

Certified Infinite Descent Criteria in Isabelle/HOL

Jamie Wright, Liron Cohen, Reuben Rowe & Andrei Popescu

July 14, 2026

Abstract

Infinite Descent is the global trace condition that underpins the soundness of cyclic reasoning and, in program analysis, the size change termination principle. Many (semi-)decision procedures for Infinite Descent are known, based on criteria ranging from automata-based constructions and relation-based characterizations, to effective (but incomplete) heuristics. We present an Isabelle/HOL mechanization of this landscape. We provide a reusable, locale-based framework of sloped graphs that defines Infinite Descent at an abstract level, independently of any concrete graph encoding. Within this framework we formalize standard *complete* criteria and prove their equivalence to the locale-level *InfiniteDescent* predicate. We also formalize tool-facing sufficient criteria, prove their soundness, and certify incompleteness where appropriate via verified counterexamples. Along the way we contribute reusable lemmas for ω -regular reasoning over streams and for Büchi-automata constructions needed by the inclusion proofs.

Contents

1	Preliminaries	2
1.1	Linear Temporal Logic and Auxillaries	2
1.2	Buchi Complementation Extended	13
2	Infinite Descent in Sloped Graphs	20
2.1	Directed Graphs	21
2.2	Sloped Graphs	39
2.3	Infinite Descent	43
3	Equivalent Criteria for Infinite Descent	54
3.1	Automata-based criteria	54
3.1.1	Vertex-language Criterion	54
3.1.2	Slope-language Criterion	66
3.2	Relation-based Criterion	79

4	Incomplete Criteria for Infinite Descent	119
4.1	Sprenger-Dam Criterion	119
4.2	Extended Sprenger-Dam Criterion	120
4.3	Sprenger-Dam Criterion Incompleteness	129
4.4	Extended Sprenger-Dam Criterion Incompleteness	131
4.5	Flat Cycles Criterion	139
4.6	Descending Unicycles Criterion	142
4.7	All Criterion	161
5	Instantiating the Abstract Framework	162
5.1	Infinite Descent Examples	162
5.1.1	Flat Aux Example	162
5.1.2	Flat Aux via Vertex-Language Automaton	163
5.1.3	Flat Aux via Sloped-Language Automaton	166
5.1.4	Descending Unicycles Example	169
5.2	Infinite Descent Counterexamples	176
5.2.1	Descending Unicycles	176
5.2.2	Flat Cycle Counterexample	181

1 Preliminaries

Some preliminaries on LTL, Transition Systems and Buchi Automata

1.1 Linear Temporal Logic and Auxillaries

theory *Preliminaries*

imports *HOL-Library.Sublist* *HOL-Library.Linear-Temporal-Logic-on-Streams*

HOL-Library.Code-Target-Int

begin

definition *any* $\equiv$ *undefined*

lemma *Suc-disj*: $j < i \implies \text{Suc } j < i \vee \text{Suc } j = i$ **by** *auto*

lemma *distinct-wrap-around*:

assumes *distinct* *c*

assumes $j > i$

shows *distinct* (*drop* *j* *c* @ *take* *i* *c*)

proof (*subst distinct-append*, *intro conjI*)

— Part 1: Individual segments are distinct

show *distinct* (*drop* *j* *c*)

using *assms*(1) *distinct-drop* **by** *blast*

```

show distinct (take i c)
  using assms(1) distinct-take by blast

— Part 2: The sets are disjoint
show set (drop j c)  $\cap$  set (take i c) = {}
proof (rule set-eqI, rule iffI)
  fix x assume  $x \in \text{set } (\text{drop } j \ c) \cap \text{set } (\text{take } i \ c)$ 
  then obtain k m where
    k:  $k < \text{length } c \ k \geq j \ x = \text{butlast } c ! k$  and
    m:  $m < \text{Suc } i \ x = \text{butlast } c ! m$ 
  by (metis assms(1,2) empty-iff inf-commute less-or-eq-imp-le set-take-disj-set-drop-if-distinct)

  from k(2) m(1) assms(2) have  $m < k$  by simp
  hence  $m \neq k$  by simp

  with assms(1) k(1) m(1) have  $\text{butlast } c ! m \neq \text{butlast } c ! k$ 
  using nth-eq-iff-index-eq[OF assms(1)]

  by (metis  $\langle x \in \text{set } (\text{drop } j \ c) \cap \text{set } (\text{take } i \ c) \rangle$  assms(2) empty-iff inf-sup-aci(1)
less-or-eq-imp-le
set-take-disj-set-drop-if-distinct)

  thus  $x \in \{\}$  using k(3) m(2) by simp
qed simp
qed

lemma distinct-outside-index:
assumes
   $0 < j \ 0 < \text{length } c \ \text{distinct } c$ 
   $\forall j < \text{length } c. \ 0 \neq j \longrightarrow c ! 0 \neq c ! j$ 
shows  $c ! 0 \notin (!) \ c \ ' \ \{j..<\text{length } c\}$ 
proof –
  {
    fix k
    assume  $k \in \{j..<\text{length } c\}$ 
    hence  $j \leq k$  and  $k < \text{length } c$  by auto
    with  $\langle 0 < j \rangle$  have  $0 < k$  by simp
    with  $\langle k < \text{length } c \rangle$  have  $c ! 0 \neq c ! k$ 
    by (metis assms length-butlast nat-neq-iff nth-butlast
nth-eq-iff-index-eq)
  }
  thus ?thesis by (auto simp: image-def)
qed

lemma distinct-outside-index':
assumes
  distinct (butlast c)
   $\text{Suc } i < \text{length } (\text{butlast } c)$ 
   $\forall j < \text{length } (\text{butlast } c). \ \text{Suc } i \neq j \longrightarrow \text{butlast } c ! \text{Suc } i \neq \text{butlast } c ! j$ 

```

shows $c ! \text{Suc } i \notin (!) (\text{butlast } c) \text{ ‘ } \{j..<i\}$
proof –
 {
 fix k
 assume $k \in \{j..<i\}$
 hence $j \leq k$ **and** $k < i$ **by** *auto*
 with *assms(1)* **have** $c ! \text{Suc } i \neq \text{butlast } c ! k$
 using *assms nth-butlast nth-eq-iff-index-eq* **by** *fastforce*
 }
thus *?thesis* **by** (*auto simp: image-def*)
qed

definition $fToList :: \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow 'a \text{ list}$ **where**
 $fToList \ n \ f \equiv \text{map } f \ [0..<n]$

lemma $\text{length-}fToList[simp]$: $\text{length } (fToList \ n \ f) = n$
unfolding $fToList\text{-def}$ **by** *simp*

lemma $\text{nth-}fToList[simp]$: $i < n \implies (fToList \ n \ f) ! i = f \ i$
unfolding $fToList\text{-def}$ **by** *simp*

lemma $fToList\text{-nth}[simp]$: $(fToList \ (\text{length } xs) \ (\text{nth } xs)) = xs$
by (*metis length-fToList nth-equalityI nth-fToList*)

lemma *list-split2*:
assumes $i < j$ **and** $j < \text{length } xs$
obtains $xs1 \ xs2 \ xs3$ **where** $xs = xs1 \ @ \ (xs ! i) \ # \ xs2 \ @ \ (xs ! j) \ # \ xs3$
proof –
 have $0 : xs = \text{take } i \ xs \ @ \ \text{drop } i \ xs$ **and** $xs ! i = \text{hd } (\text{drop } i \ xs)$
 by *simp (metis Suc-lessD assms hd-drop-conv-nth less-trans-Suc)*
 have $1 : xs ! j = (\text{drop } i \ xs) ! (j - i)$ **apply** (*subst nth-drop*) **using** *assms* **by** *auto*
 hence $2 : xs ! j \in \text{set } (\text{tl } (\text{drop } i \ xs))$
 by (*smt Cons-nth-drop-Suc assms diff-is-0-eq diff-less-mono length-Cons length-drop less-Suc-eq*
 $\text{less-imp-le-nat less-trans list.sel(3) not-less nth-equal-first-eq nth-mem set-ConsD}$)
 thus *thesis* **using** *that*
 by (*metis assms drop-Suc id-take-nth-drop less-trans split-list tl-drop*)
qed

definition $\text{repeat} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{repeat } n \ xs \equiv \text{concat } (\text{replicate } n \ xs)$

lemma $\text{repeat-0}[simp]$: $\text{repeat } 0 \ xs = []$
unfolding repeat-def **by** *simp*

lemma $\text{repeat-Suc}[simp]$: $\text{repeat } (\text{Suc } n) \ xs = xs \ @ \ \text{repeat } n \ xs$

unfolding *repeat-def* **by** *simp*

lemma *repeat-Suc2*: *repeat (Suc n) xs = repeat n xs @ xs*

unfolding *repeat-def* **by** (*metis* *append-self-conv* *concat.simps(2)* *concat-append* *replicate-Suc* *replicate-append-same*)

lemma *set-repeat[simp]*: $n > 0 \implies \text{set } (\text{repeat } n \text{ } xs) = \text{set } xs$
apply(*induct* *n*) **by** *fastforce+*

lemma *nth-repeat[simp]*: $\text{length } (\text{repeat } n \text{ } xs) = n * \text{length } xs$
by (*induct* *n*, *auto*)

lemma *repeat-nth*:

$i < n * \text{length } xs \implies \text{repeat } n \text{ } xs ! i = xs ! (i \bmod \text{length } xs)$

apply(*induct* *n* *arbitrary*: *i*)

subgoal **by** *simp*

subgoal **for** *n i* **apply**(*cases* $i < \text{length } xs$)

by (*auto* *simp*: *nth-append mod-if*) .

lemma *tl-last'*:

$tl \text{ } xs \neq [] \implies \text{last } xs = \text{last } (tl \text{ } xs)$

by (*induct* *xs*) *simp-all*

definition *fToStream* :: $(nat \Rightarrow 'a) \Rightarrow 'a \text{ stream}$ **where**

fToStream *f* $\equiv \text{smap } f \text{ nats}$

lemma *snth-fToStream[simp]*: $(fToStream \text{ } f) !! i = f \text{ } i$

unfolding *fToStream-def* **by** *simp*

lemma *shd-fToStream[simp]*: $\text{shd } (fToStream \text{ } f) = f \text{ } 0$ **unfolding** *fToStream-def*
by *simp*

lemma *stl-fToStream*: $\text{stl } (fToStream \text{ } f) !! n = (fToStream \text{ } f) !! \text{Suc } n$ **unfolding** *fToStream-def* **by** *simp*

lemma *fToStream-snth[simp]*: $(fToStream \text{ } (snth \text{ } xs)) = xs$

by (*metis* *fToStream-def* *stream-smap-nats*)

lemma *sdrop-shift-length*: $\text{length } xs = m \implies \text{sdrop } m \text{ } (xs @- ys) = ys$
by (*simp* *add*: *sdrop-shift*)

lemma *sdrop-shift-length'*: $\text{sdrop } (\text{length } xs) \text{ } (xs @- ys) = ys$
by (*simp* *add*: *sdrop-shift*)

lemma *snth-equalityI*: $(\bigwedge i. xs !! i = ys !! i) \implies xs = ys$

by (*metis* *ext* *fToStream-snth*)

lemma *hd-u-append[simp]*: $\text{hd } (u @ [\text{hd } u]) = \text{hd } u$ **by** (*simp* *add*: *hd-append*)

lemma *stake-len-append:stake* (length v) (v @- u) = v
using *stake-shift*[of length v v u] **by** *auto*

lemma *last-stake-Suc:last* (stake (Suc x') r) = r !! x' **using** *last-conv-nth*[of stake (Suc x') r] *gr0-conv-Suc* **by** (auto split: if-splits)

abbreviation *srepeat* $\equiv$ *cycle*
hide-const *cycle*
declare *cycle.simps*[simp del]
declare *snth.simps*[simp del]

lemma *stake-srepeat*:(stake (length u) (sdrop 0 (srepeat u))) = u
apply(induct length u)
subgoal by *auto*
by (metis Zero-not-Suc length-0-conv sdrop.simps(1) stake-cycle-eq)

lemma *sdrop-evI*: φ (sdrop m xs) $\implies$ ev φ xs
using *ev-iff-sdrop* **by** *auto*

lemma *srepeat-snth*[simp]: xs $\neq [] \implies$ (srepeat xs) !! i = xs!(i mod (length xs))
apply(induct i)
apply (simp add: cycle.simps hd-conv-nth snth.simps)
by (metis cycle.simps(1) hd-rotate-conv-nth rotate-conv-mod sdrop-cycle sdrop-simps(1))

lemma *sset-eq-snth*: sset xs = {xs!!i | i . True}
using *sset-range* **unfolding** *image-def* **by** (simp add: full-SetCompr-eq sset-range)

lemma *repeat-eq-stake-srepeat*:
repeat n xs = stake (n * length xs) (srepeat xs)
apply(rule nth-equalityI)
subgoal by *simp*
subgoal by *simp* (metis list.size(3) mult-0-right not-less0 repeat-nth srepeat-snth)
.

lemma *repeat-nth-eq-srepeat-snth*: i < n * length xs $\implies$ repeat n xs ! i = srepeat xs !! i
by (simp add: repeat-eq-stake-srepeat)

lemma *srepeat-repeat*:
n $\neq$ 0 $\implies$ srepeat (repeat n xs) = srepeat xs
apply(cases xs = [])
subgoal by (simp add: repeat-eq-stake-srepeat)

```

subgoal apply(rule snth-equalityI)
apply (subst srepeat-snth)
  subgoal by (simp add: repeat-eq-stake-srepeat)
  subgoal for i apply(subst srepeat-snth)
    subgoal by simp
    subgoal apply(subst repeat-nth)
      subgoal by simp
      subgoal by simp (metis (no-types, lifting) less-not-refl2 mod-mult-right-eq
mult-cancel1 mult-mod-right)
    . . . .

```

lemma last-replicate-app: $k > 0 \implies \text{last } (\text{concat } (\text{replicate } k \text{ (ls @ [l'])})) = l'$
by(induct k, auto)

lemma last-stake-Suc-app: $\text{last } (p \# \text{stake } (\text{Suc } x') \text{ } r) = \text{last } (\text{stake } (\text{Suc } x') \text{ } r)$ **by**
simp

lemma upt-Suc-hd: $\text{Suc } x' \leq y' \implies ([\text{Suc } x'..<y'] @ [y']) ! 0 = \text{Suc } x'$
unfolding le-eq-less-or-eq **using** upt-rec **apply-by**(erule disjE, auto)

lemma stl-Suc-eq: $\text{stl } r !! k = r !! \text{Suc } k$ **by** (simp add: snth.simps(2))

lemma stl-Suc: $k > 0 \implies \text{stl } r !! (k - \text{Suc } 0) = r !! k$ **by**(induct k, simp-all add:
stl-Suc-eq)

lemma stl-Suc': $k > 0 \implies \text{stl } r !! (k - 1) = r !! k$ **by**(induct k, simp-all add:
stl-Suc-eq)

lemma replicate-within-i: $(\text{Suc } i) \leq m \implies (\text{replicate } m \text{ } s @ - \text{ } x) !! i = s$ **by**(induction
(Suc i) - m arbitrary: i, simp-all)

lemma replicate-beyond-i: $(\text{Suc } i) > m \implies (\text{replicate } m \text{ } s @ - \text{ } x) !! i = x !! (i - m)$
by(induction (Suc i) - m arbitrary: i, simp-all)

lemma last-stake-i: $n > 0 \implies \text{last } (\text{stake } n \text{ } x) = x !! (n - 1)$ **using** last-conv-nth[of
stake n x] **by** auto

lemma last-stake-replicate: $(\text{Suc } i) \leq m \implies 0 < i \implies \text{last } (\text{stake } i \text{ } (\text{replicate } m \text{ } s @ - \text{ } x)) = s$
using last-conv-nth[of stake i (replicate m s @ - x)] **by** simp

lemma Suc-leq: $\text{Suc } (\text{Suc } n) \leq m \implies n \leq m - \text{Suc } (\text{Suc } 0)$ **by** auto

lemma alw-ev-iff-sdrop: $\text{alw } (\text{ev } \varphi) \text{ } xs = (\forall m. \exists n \geq m. \varphi (\text{sdrop } n \text{ } xs))$
unfolding alw-iff-sdrop ev-iff-sdrop sdrop-add

using *nat-le-iff-add* **by** *force*

lemma *ev-alw-iff-sdrop*: $ev\ (alw\ \varphi)\ xs = (\exists m. \forall n \geq m. \varphi\ (sdrop\ n\ xs))$
unfolding *alw-iff-sdrop ev-iff-sdrop sdrop-add*
using *nat-le-iff-add* **by** *force*

lemma *ev-alw-ev-iff-sdrop*: $ev\ (alw\ (ev\ \varphi))\ xs = (\exists l. \forall m \geq l. \exists n \geq m. \varphi\ (sdrop\ n\ xs))$
unfolding *alw-iff-sdrop ev-iff-sdrop sdrop-add*
using *nat-le-iff-add* **by** *force*

lemma *ev* $(alw\ (ev\ \varphi))\ xs \longleftrightarrow alw\ (ev\ \varphi)\ xs$
using *ev.base ev-alw-imp-alw-ev* **by** *fastforce*

lemma *alw-ev-sdrop*: $alw\ (ev\ \varphi)\ (sdrop\ m\ xs) \longleftrightarrow alw\ (ev\ \varphi)\ xs$
by *(metis alw-ev-sdrop alw-sdrop)*

lemma *ev-alw-sdrop*: $ev\ (alw\ \varphi)\ (sdrop\ m\ xs) \longleftrightarrow ev\ (alw\ \varphi)\ xs$
by *(metis alw.cases alw-alw ev-alw-imp-alw-ev alw-ev-sdrop)*

lemma *ev-alw-aand-alw-ev-sdrop*:
 $(ev\ (alw\ \varphi)\ aand\ alw\ (ev\ \psi))\ (sdrop\ m\ xs) \longleftrightarrow (ev\ (alw\ \varphi)\ aand\ alw\ (ev\ \psi))\ xs$
by *(simp add: ev-alw-sdrop alw-ev-sdrop)*

fun *holds2* **where** *holds2* $P\ xs \longleftrightarrow P\ (shd\ xs)\ (shd\ (stl\ xs))$

fun *second* **where** *second* $(a,b,c) = b$
lemma *second'*: $second\ x = (case\ x\ of\ (a, b, c) \Rightarrow b)$ **by** *(cases x) auto*

fun *third* **where** *third* $(a,b,c) = c$
lemma *third'*: $third\ x = (case\ x\ of\ (a, b, c) \Rightarrow c)$ **by** *(cases x) auto*

lemma *third-ssnd*: $third = (\lambda x. snd\ (snd\ x))$ **by** *auto*

lemma *third-snd*: $(third \circ snd) = (\lambda x. third\ (snd\ x))$ **by** *auto*

lemma *stake-replicate*: $stake\ n\ (replicate\ n\ X\ @- S) = replicate\ n\ X$ **by** *(induct n, auto)*

lemma *sdrop-replicate*: $sdrop\ n\ (replicate\ n\ X\ @- S) = S$ **by** *(induct n, auto)*

definition *holdsS* $S = holds\ (\lambda x. x \in S)$

lemma *alw-holds-iff-snth*: $alw\ (holds\ P)\ xs \longleftrightarrow (\forall i. P(xs!!i))$

unfolding *alw-iff-sdrop* **by** *auto*

lemma *alw-holdsS-iff-snth*: $alw\ (holdsS\ S)\ xs \longleftrightarrow (\forall i. xs!!i \in S)$
by (*simp add: alw-holds-iff-snth holdsS-def*)

lemma *alw-holds2-iff-snth*: $alw\ (holds2\ R)\ xs \longleftrightarrow (\forall i. R\ (xs!!i)\ (xs!!(Suc\ i)))$
unfolding *alw-iff-sdrop* **by** (*auto simp: snth.simps*)

lemma *ev-holds-iff-snth*: $ev\ (holds\ P)\ xs \longleftrightarrow (\exists i. P(xs!!i))$
unfolding *ev-iff-sdrop* **by** *auto*

lemma *ev-holdsS-iff-snth*: $ev\ (holdsS\ S)\ xs \longleftrightarrow (\exists i. xs!!i \in S)$
by (*simp add: ev-holds-iff-snth holdsS-def*)

lemma *ev-holds2-iff-snth*: $ev\ (holds2\ R)\ xs \longleftrightarrow (\exists i. R\ (xs!!i)\ (xs!!(Suc\ i)))$
unfolding *ev-iff-sdrop* **by** (*auto simp: snth.simps*)

lemma *alw-ev-holds-iff-snth*: $alw\ (ev\ (holds\ P))\ xs \longleftrightarrow (\forall i. \exists j \geq i. P(xs!!j))$
unfolding *alw-ev-iff-sdrop* **by** *auto*

lemma *alw-ev-holdsS-iff-snth*: $alw\ (ev\ (holdsS\ S))\ xs \longleftrightarrow (\forall i. \exists j \geq i. xs!!j \in S)$
by (*simp add: alw-ev-holds-iff-snth holdsS-def*)

lemma *alw-ev-holds2-iff-snth*: $alw\ (ev\ (holds2\ R))\ xs \longleftrightarrow (\forall i. \exists j \geq i. R\ (xs!!j)\ (xs!!(Suc\ j)))$
unfolding *alw-ev-iff-sdrop* **by** (*auto simp: snth.simps*)

lemma *ev-alw-holds-iff-snth*: $ev\ (alw\ (holds\ P))\ xs \longleftrightarrow (\exists i. \forall j \geq i. P(xs!!j))$
unfolding *ev-alw-iff-sdrop* **by** *auto*

lemma *ev-alw-holdsS-iff-snth*: $ev\ (alw\ (holdsS\ S))\ xs \longleftrightarrow (\exists i. \forall j \geq i. xs!!j \in S)$
by (*simp add: ev-alw-holds-iff-snth holdsS-def*)

lemma *ev-alw-holds2-iff-snth*: $ev\ (alw\ (holds2\ R))\ xs \longleftrightarrow (\exists i. \forall j \geq i. R\ (xs!!j)\ (xs!!(Suc\ j)))$
unfolding *ev-alw-iff-sdrop* **by** (*auto simp: snth.simps*)

lemma *ev-alw-holds2-aand-holdsS-iff-snth*:
 $ev\ (alw\ (holdsS\ S\ aand\ holds2\ R))\ xs \longleftrightarrow (\exists i. \forall j \geq i. xs!!j \in S \wedge R\ (xs!!j)\ (xs!!(Suc\ j)))$
unfolding *ev-alw-iff-sdrop* **by** (*simp add: ev-alw-holds-iff-snth holdsS-def snth.simps*)

lemma *nat-cases*: $(x::nat) = y \vee y < x \vee y > x$ **by** *auto*

lemma *snth-hd-app-eq*[*simp*]: $(x \#\# xs) !! 0 = x$ **unfolding** *snth.simps* **by** *auto*

lemma *stream-inclI*: $x \in S \implies k > 0 \longrightarrow xs!!(k-1) \in S \implies (x \#\# xs)!!k \in S$
by(*cases k, auto simp: snth-Stream*)

lemma *sdrop-1*: $(sdrop (Suc 0) nds) = stl nds$ **by** *simp*

lemma *disj-mod*: $((k::nat) \text{ div } N = 0 \wedge k \text{ mod } N = 0) \vee (0 < k \text{ div } N \wedge k \text{ mod } N = 0) \vee (0 < k \text{ mod } N)$
by *auto*

lemma *mod-bound*: $y' > x' \implies k \text{ mod } (y' - x') - Suc\ 0 < Suc\ y' - Suc\ x'$
by (*simp add: less-imp-diff-less*)

lemma *mod-contr1*: **assumes** $k \text{ mod } (y' - x') = Suc\ l'$
 $y' < Suc\ (l' + x') \quad y' - x' \neq 0$
shows *HOL.False*

proof –

have $Suc\ l' < y' - x'$ **using** *assms(1)* **by** (*metis assms(3) mod-less-divisor neq0-conv*)

also have $y' - x' < Suc\ l'$ **using** *assms(2)* **by** *auto*

ultimately show *HOL.False* **by** *simp*

qed

lemma *mod-contr2*: **assumes** $k \text{ mod } (y' - x') = Suc\ l' \quad y' = Suc\ (l' + x') \quad y' - x' \neq 0$
shows *HOL.False*

proof –

have $Suc\ l' = y' - x'$ **using** *assms* **by** *auto*

also have $k \text{ mod } (Suc\ l') = Suc\ l'$ **using** *assms* **by** *auto*

ultimately show *HOL.False* **by** (*metis Zero-not-Suc mod-mod-trivial mod-self*)

qed

lemma *mod-contr3*:

assumes $y' < k \text{ mod } (y' - x') + x' \quad Suc\ x' < y'$

shows *HOL.False*

proof –

obtain q **where** $q = k \text{ div } (y' - x')$

using *assms* **by** *simp*

hence $k = q * (y' - x') + k \text{ mod } (y' - x')$

using *assms* **by** *presburger*

define r **where** $r = k \text{ mod } (y' - x')$

then have $0 \leq r < y' - x'$ **apply** *blast* **unfolding** *r-def* **using** *pos-mod-bound assms* **by** *auto*

also have $y' - le: y' < r + x'$ **using** *assms* **unfolding** *r-def* **by** *auto*

ultimately have $r\text{-leq}: r + x' \leq (y' - x' - 1) + x'$ by *auto*
 have $y' - x' - 1 + x' = y' - 1$
 using $\langle r < y' - x' \rangle$ *less-diff-conv* $y\text{-le}$ by *fastforce*

hence $y' < y' - 1$ using $y\text{-le}$ $r\text{-leq}$ by *auto*
 thus *HOL.False* by *auto*
 qed

lemma *mod-arith1*: assumes $\text{Suc } x' < y'$
 shows $\text{Suc } (k \bmod (y' - x') + x') \leq y'$
 $(k \bmod (y' - x')) \leq y' - \text{Suc } x'$
 $\text{Suc } x' + k \bmod (y' - x') < \text{Suc } y'$

proof –
 have $x' < y' - 1$ using *assms* by *auto*
 hence $y' - x' > 0$ by *auto*
 also have $0 \leq k \bmod (y' - x') < k \bmod (y' - x') + x' < y' - x'$
 using *calculation* *mod-less-divisor* by *blast+*
 moreover have $x' \leq k \bmod (y' - x') + x' < k \bmod (y' - x') + x' + 1 < y'$
 using *calculation*(3) *less-diff-conv* by *auto*

thus $\text{Suc } (k \bmod (y' - x') + x') \leq y'$ $k \bmod (y' - x') \leq y' - \text{Suc } x'$
 $\text{Suc } x' + k \bmod (y' - x') < \text{Suc } y'$ by *auto*
 qed

lemma *le-xy-mod*: $(x'::\text{nat}) < y' \implies k \bmod (y' - x') < y' - x'$ by (*cases* k , *auto*)

definition *map-ind* $x \ i \ j \equiv \text{map } (!!) \ x \ [i..<j]$

lemma *map-ind-ne*[*simp*]: $i < j \implies \text{map-ind } x \ i \ j \neq []$ **unfolding** *map-ind-def*
 by (*cases* j , *auto*)

lemma *map-ind-len*[*simp*]: $i < j \implies \text{length } (\text{map-ind } x \ i \ j) = j - i$ **unfolding** *map-ind-def*
 by *auto*

lemma *srepeat-map-ind-eq*: $i < j \implies \text{srepeat } (\text{map-ind } x \ i \ j) \ !\! k = x \ !\! ([i..<j] \ !\ (k \bmod (j - i)))$
 using *srepeat-snth*[*OF* *map-ind-ne*, *of* $i \ j \ x \ k$] **unfolding** *map-ind-def* by *auto*

lemma *hd-map-ind*: $i < j \implies \text{hd } (\text{map-ind } x \ i \ j) = x \ !\ i$ **unfolding** *map-ind-def*
 using *list.map-sel*(1)[*of* $[i..<j]$] by *auto*

lemma *map-ind-sing*[*simp*]: $\text{map-ind } r \ i \ (\text{Suc } i) = [r \ !\ i]$
unfolding *map-ind-def* by *auto*

lemma *tl-map-ind*: $i < j \implies \text{tl} (\text{map-ind } x \ i \ j) = (\text{map-ind } x \ (\text{Suc } i) \ j)$
apply (*cases map* ((!!) x) [$i..<j$])
subgoal using *map-ind-ne* **by** *auto*
subgoal using *tl-upt*[$\text{of } i \ j$] **unfolding** *map-ind-def* **by** *fastforce* .

lemma *nth-map-ind*: $k < j - i \implies (\text{map-ind } x \ i \ j) ! k = x !! (i + k)$
using *nth-map-upt*[$\text{of } k \ j \ i$ (!!) x] **unfolding** *map-ind-def* **by** *auto*

lemma *map-ind-Suc*: $y' \geq x' \implies \text{map-ind } r \ x' \ y' @ [r !! y'] = \text{map-ind } r \ x' (\text{Suc } y')$
unfolding *map-ind-def* **by** *auto*

lemma *last-replicate-map-ind*: $0 < k \implies y' \geq x' \implies \text{last} (\text{concat} (\text{replicate } k (\text{map-ind } r \ x' (\text{Suc } y')))) = r !! y'$
using *last-replicate-app*[$\text{of } k \ \text{map-ind } r \ x' \ y' \ r !! y'$] *map-ind-Suc*[$\text{of } x' \ y' \ r$] **by** *auto*

lemma *srepeat-map-ind-snth-eq*: $i < j \implies \text{srepeat} (\text{map-ind } x \ i \ j) !! k = x !! (i + k \bmod (j - i))$
unfolding *srepeat-snth*[*OF map-ind-ne*] *map-ind-len* *nth-map-ind*[*OF le-xy-mod*] **by** *auto*

lemma *last-take-Suc*: $k = \text{Suc } n \implies \text{last} (\text{stake } k \ r) = r !! n$
apply (*cases n*)
subgoal by (*simp add: snth.simps*(1))
using *last-stake-Suc* **by** *blast*

lemma *two-ls-app*: $[y, v] = [y] @ [v]$ **by** *auto*

lemma *list-unf*: $(v \# xs' @ [y]) @ z' \# zs' @ [v] = v \# xs' @ y \# z' \# zs' @ [v]$
by *simp*

lemma *set-nemp*: $V' \neq \{\}$ $\longleftrightarrow (\exists v. v \in V')$ **by** *auto*

lemma *exI2*: $P \ a \ b \implies \exists a \ b. P \ a \ b$ **by** *auto*
lemma *exI3*: $P \ a \ b \ c \implies \exists a \ b \ c. P \ a \ b \ c$ **by** *auto*
lemma *exI4*: $P \ a \ b \ c \ d \implies \exists a \ b \ c \ d. P \ a \ b \ c \ d$ **by** *auto*

lemma *exI5*: $P \ a \ b \ c \ d \ e \implies \exists a \ b \ c \ d \ e. P \ a \ b \ c \ d \ e$ **by** (*rule exI4* [*of* - $a \ b \ c \ d$], *auto*)

end

1.2 Buchi Complementation Extended

A small extension to the Buchi Complementation AFP entry, including: useful definitions, results on omega lasso languages, finitely occurring predicates on streams and complement results

theory *Buchi-Preliminaries*

imports *Preliminaries Buchi-Complementation.Complementation-Final*

begin

lemmas *run-def = nba.run-alt-def-snth nba.target-alt-def nba.states-alt-def*

lemma *runE-alpha:*

assumes *NBA.run A (x ||| r) p*

shows $\bigwedge k. r !! k \in \text{nba.alphabet } A$

using *assms last-stake-Suc[of - r]* **unfolding** *run-def* **by**(*auto split-if-splits*)

lemma *runE-0:*

assumes *NBA.run A (x ||| r) p*

shows $r !! 0 \in \text{nba.transition } A (x !! 0) p$

using *assms last-stake-Suc[of - r]* **unfolding** *run-def* **by**(*auto split-if-splits*)

lemma *runE-Suc:*

assumes *NBA.run A (x ||| r) p*

shows $\bigwedge k. r !! \text{Suc } k \in \text{nba.transition } A (x !! \text{Suc } k) (r !! k)$

using *assms last-stake-Suc[of - r]* **unfolding** *run-def* **by**(*auto split-if-splits*)

lemma *runE-Suc':*

assumes *NBA.run A (x ||| r) p k > 0*

shows $r !! k \in \text{nba.transition } A (x !! k) (r !! (k-1))$

using *assms(2)[unfolded gr0-conv-Suc] runE-Suc[OF assms(1)]* **by** *auto*

lemma *runE-trans:*

assumes *NBA.run A (x ||| r) p*

shows $k > 0 \wedge r !! k \in \text{nba.transition } A (x !! k) (r !! (k-1)) \vee$

$k = 0 \wedge r !! k \in \text{nba.transition } A (x !! k) p$

apply(*cases k = 0*)

subgoal using *runE-0[OF assms]* **by** *auto*

subgoal using *runE-Suc'[OF assms]* **by** *auto* .

lemmas *node-def = nba.nodes-alt-def[unfolded nba.reachable-alt-def nba.target-alt-def image-def Union-eq Bex-def]*

lemma *nodeI:*

assumes $p \in \text{nba.initial } A$

$(n = p \wedge \text{NBA.path } A [] p) \vee (\exists r. r \neq [] \wedge n = \text{last } (\text{NGBA.states } r p) \wedge$

$\text{NBA.path } A r p)$

shows $n \in NBA.nodes\ A$
unfolding *node-def*
apply(*rule CollectI*, *rule exI*[*of* - $\{y. \exists r. (r = [] \longrightarrow y = p \wedge NBA.path\ A\ []\ p) \wedge (r \neq [] \longrightarrow y = last\ (NGBA.states\ r\ p) \wedge NBA.path\ A\ r\ p)\}$], *intro conjI*)
subgoal using *assms by auto*
subgoal apply(*rule CollectI*) **using** *assms by auto* .

lemma *nodeI'*:
assumes $p \in nba.initial\ A$
 $(\exists r. r \neq [] \wedge n = last\ (NGBA.states\ r\ p) \wedge NBA.path\ A\ r\ p)$
shows $n \in NBA.nodes\ A$
unfolding *node-def*
apply(*rule CollectI*, *rule exI*[*of* - $\{y. \exists r. (r = [] \longrightarrow y = p \wedge NBA.path\ A\ []\ p) \wedge (r \neq [] \longrightarrow y = last\ (NGBA.states\ r\ p) \wedge NBA.path\ A\ r\ p)\}$], *intro conjI*)
subgoal using *assms by auto*
subgoal apply(*rule CollectI*) **using** *assms by auto* .

lemma *nba-path-singl*: $fst\ a \in nba.alphabet\ A \Longrightarrow snd\ a \in nba.transition\ A\ (fst\ a)$
 $p \Longrightarrow NBA.path\ A\ [a]\ p$ **using** *nba.path.cons by auto*

lemma *nba-path-app-singl*: $fst\ a \in nba.alphabet\ A \Longrightarrow snd\ a \in nba.transition\ A\ (fst\ a)$
 $(NGBA.target\ r\ p) \Longrightarrow NBA.path\ A\ r\ p \Longrightarrow NBA.path\ A\ (r\ @\ [a])\ p$
by (*intro nba.path-append nba-path-singl, auto*)

lemma *target-last-states[simp]*: $r \neq [] \Longrightarrow NGBA.target\ r\ p = last\ (NGBA.states\ r\ p)$
unfolding *nba.target-alt-def nba.states-alt-def by auto*

lemma *target-emp[simp]*: $NGBA.target\ []\ p = p$ **unfolding** *nba.target-alt-def by auto*

lemma *last-states-singl[simp]*: $last\ (NGBA.states\ [(x,s)]\ p) = s$ **unfolding** *nba.states-alt-def by auto*

lemma *last-states-app[simp]*: $ra \neq [] \Longrightarrow last\ (NGBA.states\ (ra\ @\ [(x,s)]\ p) = s$
unfolding *nba.states-alt-def by auto*

lemma *run-nodes-closure*:

assumes *lang*: $p \in nba.initial\ A\ NBA.run\ A\ (x\ |||\ r)\ p$
shows $\bigwedge x. r \neq [] \Longrightarrow x \in NBA.nodes\ A$
subgoal for x' **proof**(*rule nodeI'*[*OF lang*(1)], *induct x'*)
case 0
then show ?*case*
apply(*intro exI*[*of* - $[(shd\ x, shd\ r)]$])
using *runE-alpha*[*OF lang*(2), *of 0*] *runE-0*[*OF lang*(2)] **by auto**
next
case (*Suc x'*)
then obtain *ra* **where** $ra: ra \neq []\ r \neq []\ x' = last\ (NGBA.states\ ra\ p)\ NBA.path\ A\ ra\ p$ **by auto**
then show ?*case*

```

apply(intro exI[of - ra @ [(x !! Suc x', r !! Suc x')]] conjI)
  subgoal by auto
  subgoal by auto
  subgoal apply (rule nba-path-app-singl)
    using runE-alpha[OF lang(2), of Suc x'] runE-Suc[OF lang(2), of x'] by
auto .
qed .

```

```

lemma map-snd-stake-szip:map snd (stake k (x ||| y)) = stake k y by(cases k,
auto)
lemma szip-unfold:(x' ||| r') = (shd x', shd r') ## (stl x' ||| stl r') by auto

```

```

lemma len-stake-szip: length (q # map snd (stake (Suc n) (x ||| r))) = Suc (Suc
n) by simp
lemma nth-stake-szip:(q # map snd (stake (Suc n) (x ||| r))) ! Suc n = r !! n
apply (induct n) by (auto,metis lessI stake.simps(2) stake-nth)

```

```

lemma last-stake-szip:last (q # map snd (stake (Suc n) (x ||| r))) = r !! n
using last-conv-nth[of q # map snd (stake (Suc n) (x ||| r)),unfolded len-stake-szip
diff-Suc-1 nth-stake-szip] by auto

```

```

lemma fst-nth-szip:fst ((x ||| r) !! k) = x !! k by auto
lemma snd-nth-szip:snd ((x ||| r) !! k) = r !! k by auto

```

```

lemma fins-finite:fins P x  $\implies$  finite {m. P (x !! m)}
  using infinite-iff-alw-ev
  unfolding infinite-iff-alw-ev[symmetric] by simp

```

```

lemma fins-imp:fins P x  $\implies$  alw (holds ( $\lambda x. \neg P x$ )) x  $\vee$  ( $\exists i. \forall k > i. P (x !! i)$ 
 $\wedge \neg P (x !! k)$ )
  apply(drule fins-finite)
  apply(cases {m. P (x !! m)} = {})
  subgoal by(rule disjI1, unfold alw-holds-iff-snth, auto)
  subgoal apply(rule disjI2, frule Max-in, clarify)
    by(rule exI[of - Max {m. P (x !! m)}], auto) .

```

```

lemma target-szip-nth:n > 0  $\implies$  NGBA.target (stake n w || stake n r') i = r' !!
(n-1)
unfolding nba.target-alt-def nba.states-alt-def using last-conv-nth[of i # map snd

```

(*stake* n w || *stake* n r)] **by** *auto*

lemma *infs-snthI*: ($\bigwedge n. \exists k \geq n. P (w !! k)$) $\implies$ *infs* P w
unfolding *infs-snth* **by** *auto*

lemma *complement-eq1*:
assumes $NBA.alphabet\ A \subseteq NBA.alphabet\ B$
assumes *finite* ($NBA.nodes\ B$)
shows $NBA.language\ A \subseteq NBA.language\ B \longleftrightarrow NBA.language\ A \cap NBA.language\ (complement\ B) = \{\}$
proof –
have $NBA.language\ A \subseteq streams\ (NBA.alphabet\ B)$
using *streams-mono2* *nba.language-alphabet* *assms(1)* **by** *blast*
then show *?thesis*
using *assms(2)* **by** (*auto simp: complement-language*)
qed

lemma *complement-eq2*: ($NBA.language\ A \cap NBA.language\ (complement\ B) = \{\}$)
 $= (NBA.language\ (intersect\ A\ (complement\ B)) = \{\})$ **by** *auto*

lemma *complement-eq3*:
assumes $NBA.alphabet\ A \subseteq NBA.alphabet\ B$
assumes *finite* ($NBA.nodes\ B$)
shows $NBA.language\ A \subseteq NBA.language\ B = (NBA.language\ (intersect\ A\ (complement\ B))) = \{\}$
unfolding *complement-eq1* [*OF assms*] *complement-eq2* **by** *auto*

lemma *omega-lasso-lang:finite* ($NBA.nodes\ A$) $\implies x \in NBA.language\ A \implies \exists v$
 $u. v @- srepeat\ u \in NBA.language\ A \wedge u \neq []$
proof (*erule nba.language-elim, unfold alw-ev-scons infs-infm Inf-many-def*)
fix $r\ p$
assume *finite:finite* ($NBA.nodes\ A$) **and**
 $lang:p \in nba.initial\ A\ NBA.run\ A\ (x ||| r)\ p\ infinite\ \{i. nba.accepting\ A\ (r$
 $!!\ i)\}$

obtain n **where** $n:n = card\ (NBA.nodes\ A)$ **using** *finite* **by** *auto*

have $p-in-A:p \in NBA.nodes\ A$ **using** *lang(1,2)* **by** (*auto split:if-splits*)
hence $pr-in-A:\bigwedge x. r!!\ x \in NBA.nodes\ A$ **subgoal for** x **using** *run-nodes-closure* [*OF lang(1,2)*] **by** *auto* .

also have *Suc-set:n* $\neq 0$ **unfolding** n **using** *calculation local.finite* **by** *auto*

obtain AS **where** $AS\text{-finite:finite } AS$ **and** $AS\text{-card:card } AS = \text{Suc } (\text{Suc } n)$ **and**
 $AS\text{-subseq:} AS \subseteq \{i. nba.\text{accepting } A (r !! i)\}$
using $\text{infinite-arbitrarily-large}[OF \text{ lang } (\beta), \text{ of } \text{Suc } (\text{Suc } n)]$ **by** *auto*

hence $AS\text{-accept:} \forall i \in AS. nba.\text{accepting } A (r !! i)$ **using** $AS\text{-subseq}$ **by** *auto*

hence $AS\text{-accept':} \bigwedge i. i \in AS \implies nba.\text{accepting } A (r !! i)$ **by** *auto*
have $((!!) r \text{ ' } AS \subseteq NBA.\text{nodes } A)$ **using** $pr\text{-in-}A$ **by** *auto*

hence $\text{card } ((!!) r \text{ ' } AS) \leq n$ **unfolding** n **by** $(\text{simp add: card-mono finite})$

moreover have $\text{card:card } ((!!) r \text{ ' } AS) < \text{card } AS$ **using** $AS\text{-card calculation}$ **by** *auto*

ultimately obtain $x' y'$ **where** $xy:r !! x' = r !! y' x' < y' nba.\text{accepting } A (r !! x')$
using $\text{pigeonhole}[of \ \lambda k. r !! k, OF \text{ card, unfolded inj-on-def}] AS\text{-accept nat-neq-iff}$
by *metis*

hence $\text{infs-r:infs } (nba.\text{accepting } A) (\text{srepeat } (\text{map-ind } r \ x' \ y'))$
apply-apply (rule infs-snthI)
subgoal for n **apply** $(\text{intro exI}[of \ - \ n * (y' - x')])$
unfolding $\text{srepeat-map-ind-snth-eq}$ **by** *auto* .
have $y\text{-ne:} y' > 0$ **using** $xy(2)$ **by** *auto*
have $\text{Suc-xy':} \text{Suc } x' \leq y'$ **using** xy **by** *auto*
obtain n **where** $y':y' = \text{Suc } n$ **using** $y\text{-ne}$ **by** $(\text{cases } y', \text{ auto})$

have $\text{take-hd:} \text{stake } x' \ x \parallel \text{stake } x' \ r = \text{stake } x' \ (x \parallel r)$ **by** *auto*
have $\text{stake-red:} \text{stake } (\text{Suc } x') \ x @- \text{srepeat } (\text{map-ind } x \ (\text{Suc } x') \ (\text{Suc } y')) \parallel$
 $\text{stake } x' \ r @- \text{srepeat } (\text{map-ind } r \ x' \ y') =$
 $(\text{stake } (\text{Suc } x') \ x \parallel \text{stake } (\text{Suc } x') \ r) @- (\text{srepeat } (\text{map-ind } x \ (\text{Suc } x') \ (\text{Suc } y')) \parallel \parallel \text{srepeat } (\text{map-ind } r \ (\text{Suc } x') \ (\text{Suc } y')))$
using $\text{tl-map-ind}[OF \ xy(2), \text{ of } r] \text{hd-map-ind}[OF \ xy(2), \text{ of } r]$
 $\text{map-ind-Suc}[OF \ \text{Suc-xy'}, \text{ of } r] \ xy(1)$
unfolding stake-Suc **by** (auto simp:)

show $\exists v \ u. v @- \text{srepeat } u \in NBA.\text{language } A \wedge u \neq []$
proof $(\text{cases } \text{Suc } x' = y')$
case *True*
hence $rx\text{-trans:} r !! x' \in nba.\text{transition } A (x !! \text{Suc } x') (r !! x')$ **using** $xy(1)$
 $\text{runE-Suc}[OF \ \text{lang}(2), \text{ of } x']$ **by** *auto*
have $\text{stl-r:} r !! \text{Suc } x' = r !! x'$ **using** *True* xy **by** *auto*
show *?thesis*
proof $(\text{intro exI2}[of \ - \ \text{stake } (\text{Suc } x') \ x \ [x !! (\text{Suc } x')]] \text{conjI } nba.\text{language}[of \ p \ A$
 $- \text{stake } x' \ r @- \text{srepeat } [r !! x'], OF \ \text{lang}(1)])$
let $?w = \text{stake } (\text{Suc } x') \ x @- \text{srepeat } [x !! \text{Suc } x']$

```

    and ?r = stake x' r @- srepeat [r !! x']
    have stake-red:?w ||| ?r = (stake (Suc x') x || stake (Suc x') r) @- (srepeat [x
!! Suc x'] ||| srepeat [r !! Suc x'])
    using xy(1) unfolding stake-Suc True[symmetric] by auto
    show infs (nba.accepting A) (p ## ?r) using infs-r unfolding True[symmetric]
by auto
    show [x !! Suc x'] ≠ [] by auto
    show NBA.run A (?w ||| ?r) p
    proof(unfold stake-red, rule nba.run-shift)
      show NBA.path A (stake (Suc x') x || stake (Suc x') r) p unfolding take-hd
using nba.run-stake[OF lang(2), of Suc x'] by auto
      show NBA.run A (srepeat [x !! Suc x'] ||| srepeat [r !! Suc x']) (NGBA.target
(stake (Suc x') x || stake (Suc x') r) p)
      unfolding target-zip-nth[of Suc x' x r p, OF zero-less-Suc, unfolded diff-Suc-1]
stl-r
      apply(rule nba.snth-run, intro conjI)
      subgoal using runE-alpha[OF lang(2), of Suc x'] by auto
      subgoal for k apply(induct k) using rx-trans by auto .
    qed
  qed
next
  case False
  hence Suc-xy:Suc x' < y' using Suc-xy'[unfolded le-eq-less-or-eq] by auto
  show ?thesis proof(intro exI2[of - stake (Suc x') x map-ind x (Suc x') (Suc
y')]) conjI
    nba.language[of p A - stake x' r @- srepeat (map-ind r x' y'), OF lang(1)]
    let ?w = stake (Suc x') x @- srepeat (map-ind x (Suc x') (Suc y'))
    and ?r = stake x' r @- srepeat (map-ind r x' y')

    show infs (nba.accepting A) (p ## ?r) using infs-r by auto
    show map-ind x (Suc x') (Suc y') ≠ [] using map-ind-ne[of Suc x'] xy(2) by
auto
    show NBA.run A (?w ||| ?r) p
    proof(unfold stake-red, rule nba.run-shift)

      show NBA.path A (stake (Suc x') x || stake (Suc x') r) p unfolding take-hd
using nba.run-stake[OF lang(2), of Suc x'] by auto

      have target-Suc:NGBA.target (stake (Suc x') x || stake (Suc x') r) p = r !!
x' using target-zip-nth[of Suc x' x r p] by auto

      have path-step:NBA.path A [(stl x !! x', stl r !! x')] (r !! y')
      unfolding xy(1)[symmetric] apply(rule nba-path-singl)
      subgoal using runE-alpha[OF lang(2), of Suc x'] by auto
      subgoal using runE-Suc[OF lang(2)] by auto .

    have SucX:Suc x' < Suc y' using Suc-xy by auto
    have SucX':Suc x' ≤ y' using Suc-xy by auto

```

note *transition-simp* = *snth-szip fst-conv snd-conv srepeat-map-ind-eq*[*OF SucX*] *nba.target-alt-def nba.states-alt-def map-snd-stake-szip*
map-snd-zip-take stake-cycle[*OF map-ind-ne*[*OF SucX*, of
r]]

show *NBA.run A (srepeat (map-ind x (Suc x') (Suc y')) ||| srepeat (map-ind
r (Suc x') (Suc y')) (NGBA.target (stake (Suc x') x || stake (Suc x') r) p)*
apply(*rule nba.stake-run, unfold target-Suc*)
subgoal for *k* **apply**(*induct k*)
subgoal by *auto*
subgoal for *k* **apply**(*unfold stake-Suc, rule nba-path-app-singl*)

subgoal using *runE-alpha*[*OF lang(2)*] **by** (*auto simp: transition-simp*)

subgoal unfolding *transition-simp map-ind-len*[*OF SucX*]
using *Suc-xy disj-mod*[*of k Suc y' - Suc x'*] **apply-apply**(*elim disjE*)
subgoal using *runE-Suc*[*OF lang(2), of x'*] **by** *auto*
subgoal using *last-replicate-map-ind*[*of k div (y' - x') - - r, OF -*
SucX]
runE-Suc[*OF lang(2), of x'*] *xy(1)* **by** *auto*
subgoal using *last-take-nth-conv*[*of k mod (y' - x') map-ind r (Suc*
x') (Suc y')]
nth-map-ind[*of (k mod (y' - x') - Suc 0) Suc y' Suc x' r,*
OF mod-bound[*OF xy(2), of k*]]
nth-upt[*of Suc x' (k mod (y' - x')) Suc y', OF mod-arith1(3)*][*OF*
Suc-xy, of k]]
runE-Suc[*OF lang(2), of (x' + k mod (y' - x'))*]
unfolding *add-Suc* **by** *auto* .
by *auto* . .
qed
qed
qed
qed

corollary *omega-lasso-lang':finite (NBA.nodes A) $\implies$ NBA.language A $\neq \{\}$ $\longleftrightarrow$
 $(\exists v u. v @- \text{srepeat } u \in \text{NBA.language } A \wedge u \neq [])$*

using *omega-lasso-lang* **by** *auto*

corollary *omega-lasso-lang'':finite (NBA.nodes A) $\implies$ NBA.language A = $\{\}$ $\longleftrightarrow$
 $(\forall v u. v @- \text{srepeat } u \notin \text{NBA.language } A)$*

using *omega-lasso-lang* **by** *auto*

lemma *prop1*:

assumes *NBA.alphabet A $\subseteq$ NBA.alphabet B*

assumes *finite (NBA.nodes A) finite (NBA.nodes B)*

assumes $\bigwedge v u. u \neq [] \implies v @- \text{repeat } u \in \text{NBA.language } A \implies v @- \text{repeat } u \in \text{NBA.language } B$
shows $\text{NBA.language } A \subseteq \text{NBA.language } B$
proof(rule ccontr)
assume $\neg \text{NBA.language } A \subseteq \text{NBA.language } B$
then obtain w **where** $w\text{-in}:w \in \text{NBA.language } A$ **and** $w\text{-nin}:w \notin \text{NBA.language } B$ **by** auto
hence $w \in \text{NBA.language (complement } B)$
using *assms* **by** (meson language-ranking nba.language-alphabet ranking-complement streams-mono subset-iff)

also have $\text{fin:finite (NBA.nodes (intersect } A \text{ (complement } B))}$
using *intersect-nodes-finite*[OF *assms*(2) *complement-finite*[OF *assms*(3)]] **by** auto

moreover have $\text{NBA.language (intersect } A \text{ (complement } B)) \neq \{\}}$ **using** *calculation w-in* **by** auto

ultimately obtain $v u$ **where** $vu:v @- \text{repeat } u \in \text{NBA.language (intersect } A \text{ (complement } B))$ $u \neq []$
using *omega-lasso-lang'*[OF *fin*] **by** auto

hence $v @- \text{repeat } u \in \text{NBA.language } A \wedge v @- \text{repeat } u \notin \text{NBA.language } B$
by (simp add: *assms*(3) *complement-language*)

thus *HOL.False* **using** *assms*(4)[OF *vu*(2)] **by** auto
qed

corollary *prop1'*:
assumes $\text{NBA.alphabet } A \subseteq \text{NBA.alphabet } B$
assumes $\text{finite (NBA.nodes } A) \text{ finite (NBA.nodes } B)$
assumes $\neg(\text{NBA.language } A \subseteq \text{NBA.language } B)$
shows $\exists v u. v @- \text{repeat } u \in \text{NBA.language } A \wedge v @- \text{repeat } u \notin \text{NBA.language } B \wedge u \neq []$

using *prop1*[OF *assms*(1,2,3)] *assms*(4) **by** auto
end

2 Infinite Descent in Sloped Graphs

We follow the formulation given in [2], in terms of sloped graphs.

theory *Directed-Graphs*
imports *Preliminaries*
begin

2.1 Directed Graphs

```

locale Graph =
fixes Node :: 'node set
and edge :: 'node  $\Rightarrow$  'node  $\Rightarrow$  bool
begin

inductive pathG :: 'node list  $\Rightarrow$  bool
where
  Base: {nd,nd'}  $\subseteq$  Node  $\implies$  edge nd nd'  $\implies$  pathG [nd,nd']
  |Step: nd  $\in$  Node  $\implies$  pathG ndl  $\implies$  edge (last ndl) nd  $\implies$  pathG (ndl @ [nd])

lemma path-length-ge2: assumes pathG ndl shows length ndl  $\geq$  2
using assms by (induct, auto)

lemma path-set:
assumes pathG ndl shows set ndl  $\subseteq$  Node
using assms by (induct, auto)

lemma path-nth-'node:
pathG ndl  $\implies$  i < length ndl  $\implies$  ndl!i  $\in$  Node
  using path-set nth-mem by blast

lemma pathG-butlast:2 < length c  $\implies$  pathG c  $\implies$  pathG (butlast c) using
pathG.cases by fastforce

lemma path-nth-edge:
assumes pathG ndl Suc i < length ndl shows edge (ndl!i) (ndl!(Suc i))
using assms apply(induct arbitrary: i) apply simp-all
by (metis Suc-lessI append-Nil2 append-eq-conv-conj butlast-snoc hd-drop-conv-nth
last-snoc nth-append-length nth-butlast take-hd-drop)

lemma path-iff-set-nth:
pathG ndl  $\longleftrightarrow$ 
  length ndl  $\geq$  2  $\wedge$  set ndl  $\subseteq$  Node  $\wedge$  ( $\forall$  i. Suc i < length ndl  $\longrightarrow$  edge (ndl!i)
(ndl!(Suc i)))
apply safe
  subgoal by (simp add: path-length-ge2)
  subgoal using path-set by blast
  subgoal by (simp add: path-nth-edge)
  subgoal proof(induction ndl rule: rev-induct)
    case Nil thus ?case by auto
  next
    case (snoc nd ndl)
    show ?case
  proof(cases length ndl = 1)
    case True then obtain nd' where ndl: ndl = [nd'] by (cases ndl, auto)
    show ?thesis unfolding ndl

```

```

    apply (simp, rule pathG.Base) using snoc by (auto simp: ndl)
  next
    case False
    show ?thesis apply (rule pathG.Step)
      subgoal using snoc by auto
      subgoal apply (rule snoc.IH) using snoc False by auto (metis Suc-lessD
nth-append)
      subgoal using snoc
      by simp (metis Suc-le-length-iff append-butlast-last-id length-append-singleton
lessI
list.simps(3) nth-append nth-append-length) .
  qed
  qed .

```

lemma *path-iff-nth*:

```

pathG ndl  $\longleftrightarrow$ 
  length ndl  $\geq 2 \wedge (\forall i < \text{length } ndl. \text{ndl}!i \in \text{Node}) \wedge (\forall i. \text{Suc } i < \text{length } ndl \longrightarrow$ 
  edge (ndl!i) (ndl!(Suc i)))
unfolding path-iff-set-nth apply safe
  subgoal by auto
  by (auto simp: in-set-conv-nth)

```

lemma *Graph-pathG-restrict*:

```

Graph.pathG N e ndl  $\longleftrightarrow$  Graph.pathG N ( $\lambda x y. e \ x \ y \wedge x \in N \wedge y \in N$ ) ndl
unfolding Graph.path-iff-nth by auto

```

lemma *path-appendL*:

```

pathG (ndl1 @ ndl2)  $\implies$  length ndl1  $\geq 2 \implies$  pathG ndl1
unfolding path-iff-set-nth by simp (metis Suc-lessD nth-append trans-less-add1)

```

lemma *path-appendR*:

```

pathG (ndl1 @ ndl2)  $\implies$  length ndl2  $\geq 2 \implies$  pathG ndl2
unfolding path-iff-set-nth by simp (metis add-Suc-right nat-add-left-cancel-less
nth-append-length-plus)

```

lemma *path-appendM*:

```

pathG (ndl1 @ ndl2 @ ndl3)  $\implies$  length ndl2  $\geq 2 \implies$  pathG ndl2
by (smt append-Nil2 dual-order.trans path-appendL
path-appendR le-add-same-cancel1 length-append length-greater-0-conv less-le)

```

lemma *ls-app*: $[a, b, c] = [a, b] @ [c]$ **by** auto

lemma *notPathG-within*: $\neg \text{pathG } [a, b] \implies \neg \text{pathG } (ls @ [a, b] @ ls')$ **using** path-appendL[*of* $ls @ [a, b] \ ls'$] path-appendR[*of* $ls \ [a, b]$] **by** auto

lemma *notPathG-within'*: $\neg \text{pathG } [a, b] \implies \neg \text{pathG } (ls \# a \# b \# ls')$ **using** path-appendL[*of* $[ls, a, b] \ ls'$] path-appendR[*of* $[ls]$] **by** auto

lemma *pathG-tl*: $2 \leq \text{length } xs \implies \text{pathG } (x \# xs) \implies \text{pathG } xs$ **using** pathG.cases

by (metis Graph.path-appendR append-Cons append-Nil)

lemma path-append-hd:

assumes pathG (ndl1 @ [hd ndl2]) **and** pathG ndl2

shows pathG (ndl1 @ ndl2)

proof–

have 1: $\bigwedge ndl1\ ndl2\ ndl2'.\ ndl1\ @\ [ndl2,\ ndl2'] = (ndl1\ @\ [ndl2])\ @\ [ndl2']$ **by** simp

show ?thesis **using** assms(2,1) **apply** (induction arbitrary: ndl1)

subgoal for nd2 ndl2' ndl1 **unfolding** 1 **apply** (rule pathG.Step) **by** auto

subgoal for nd2 ndl2 ndl1 **unfolding** append-assoc[symmetric] **apply** (rule pathG.Step)

subgoal **by** auto

subgoal **by** (metis Nil-is-append-conv pathG.cases hd-append2 list.distinct(1))

subgoal **by** (metis Nil-is-append-conv pathG.cases last-appendR list.distinct(1))

..

qed

lemma path-append-last:

assumes pathG ndl1 **and** pathG (last ndl1 # ndl2)

shows pathG (ndl1 @ ndl2)

proof–

have [simp]: $ndl1 \neq []$ **using** assms(1) pathG.cases **by** auto

hence 1: $ndl1\ @\ ndl2 = (butlast\ ndl1)\ @\ (hd\ (last\ ndl1\ \# \ ndl2)\ \# \ ndl2)$ **by** simp

show ?thesis **unfolding** 1 **apply** (rule path-append-hd) **using** assms **by** auto

qed

lemma path-Cons:

assumes $nd \in Node$ edge nd (hd ndl) pathG ndl

shows pathG (nd # ndl)

using assms(3,1,2) **apply** induction

subgoal **by** simp (metis Base append.left-neutral append-Cons bot-least path-append-last

insert-subset snoc-eq-iff-butlast)

subgoal **by** simp (metis butlast.simps(2) pathG.simps hd-append2 last.simps list.distinct(1)

snoc-eq-iff-butlast) .

lemma not-path-Nil[simp]: $\neg pathG []$

using pathG.cases **by** blast

lemma not-path-singl[simp]: $\neg pathG [nd]$

using path-length-ge2 **by** fastforce

lemma not-path-emp: $Node = \{\} \implies \neg pathG\ ndl$

by (metis bot.extremum-uniqueI empty-iff pathG.simps insert-not-empty)

```

lemma path-singl-in:
assumes edge nd nd  $n \geq 2$   $nd \in \text{Node}$ 
shows pathG (replicate n nd)
using assms apply(induct n)
  subgoal by auto
  subgoal using Base path-Cons le-Suc-eq by auto .

lemma path-singl-set:
assumes set ndl  $\subseteq \{nd\}$   $nd \in \text{Node}$ 
shows pathG ndl  $\longleftrightarrow$  (edge nd nd  $\wedge$  ( $\exists n \geq 2. \text{ndl} = \text{replicate } n \text{ nd}$ ))
proof -
  {assume pathG ndl
   hence edge nd nd  $\wedge$  ( $\exists n \geq 2. \text{ndl} = \text{replicate } n \text{ nd}$ )
   using assms apply(induction)
   subgoal by simp (metis dual-order.refl empty-replicate numeral-2-eq-2 replicate-Suc)
   subgoal by simp (metis linear not-less-eq-eq replicate-Suc replicate-append-same)
   .
  }
  thus ?thesis using path-singl-in by (auto simp: assms(2))
qed

lemma path-two-incl:
assumes edge nd nd'  $\{nd, nd'\} \subseteq \text{Node}$ 
shows pathG [nd, nd']
using assms unfolding path-iff-set-nth by simp

lemma path-two-concat-incl:
assumes edge nd nd' edge nd' nd  $n > 0$   $\{nd, nd'\} \subseteq \text{Node}$ 
shows pathG (concat (replicate n [nd, nd']))
using assms(1-3) apply(induct n)
  subgoal by auto
  subgoal by simp (smt pathG.Base assms(4) concat.simps(1) concat.simps(2) concat-append
    path-Cons hd-append2 insert-subset list.distinct(1) list.sel(1) neq0-conv replicate-0
    replicate-append-same) .

lemma pathG-butlast-not-nil: pathG n  $\implies$  butlast n  $\neq []$  by (metis Graph.not-path-Nil
  Graph.not-path-singl append-butlast-last-id
  self-append-conv2)

```

definition *pathCon* **where**
pathCon nd nd' $\equiv \exists \text{ndl}. \text{pathG ndl} \wedge \text{hd ndl} = nd \wedge \text{last ndl} = nd'$

lemma *pathCon-trans*:
pathCon *nd* *nd'* $\implies$ *pathCon* *nd'* *nd''* $\implies$ *pathCon* *nd* *nd''*
unfolding *pathCon-def*
by (*metis path-append-last hd-Cons-tl hd-append2 last.simps last-appendR not-path-Nil*)

lemma *not-pathCon-emp*: *Node* = {} $\implies \neg$ *pathCon* *nd1* *nd2*
unfolding *pathCon-def* **by** (*simp add: Graph.not-path-emp*)

lemma *pathCon-singl*: *Node* = {*nd*} $\implies$ *pathCon* *nd1* *nd2* $\longleftrightarrow$ *nd1* = *nd* $\wedge$ *nd2* = *nd* $\wedge$ *edge* *nd1* *nd2*
unfolding *pathCon-def*
by (*metis empty-iff path-nth-'node path-set path-singl-set path-two-incl hd-conv-nth*

insert-iff last.simps last-appendR
length-greater-0-conv not-path-Nil replicate-append-same subsetI)

lemma *path-nth-pathCon*:
assumes *fp*: *pathG* *ndl* **and** *ij*: *i* < *j* *j* < *length* *ndl*
shows *pathCon* (*ndl*! *i*) (*ndl*! *j*)
proof–
obtain *ndl1* *ndl2* *ndl3* **where** *ndl* = *ndl1* @ (*ndl*! *i*) # *ndl2* @ (*ndl*! *j*) # *ndl3*
using *ij* **using** *list-split2* **by** *blast*
hence *ndl*: *pathG* (*ndl1* @ ((*ndl*! *i*) # *ndl2* @ [*ndl*! *j*]) @ *ndl3*) **using** *assms* **by**
auto
have *pathG* ((*ndl*! *i*) # *ndl2* @ [*ndl*! *j*])
using *path-appendM[OF ndl]* **by** *simp*
thus ?thesis **unfolding** *pathCon-def* **by** *auto*
qed

lemma *path-set-pathCon*:
assumes *fp*: *pathG* *ndl* **and** *nd*: {*nd*, *nd'*} $\subseteq$ *set* *ndl* *nd* $\neq$ *nd'*
shows *pathCon* *nd* *nd'* $\vee$ *pathCon* *nd'* *nd*
using *path-nth-pathCon[OF fp]* *nd* **unfolding** *set-conv-nth*
using *not-less-iff-gr-or-eq*
by (*smt (verit) insert-subset linorder-neqE-nat mem-Collect-eq*)

definition *cycleG* **where**
cycleG *ndl* $\equiv$ *pathG* *ndl* $\wedge$ *hd* *ndl* = *last* *ndl*

lemma *cycleG-not-nil*: *cycleG* *c* $\implies$ *c* $\neq$ [] **unfolding** *cycleG-def* **by** *auto*
lemma *cycleG-length-ge*: *cycleG* *c* $\implies$ *length* *c* $\geq$ 2 **unfolding** *cycleG-def* **using**
path-length-ge2 **by** *auto*

lemma *cycleG-path-drop*: *cycleG* *c* $\implies$ *j* < *length* (*butlast* *c*) $\implies$ *pathG* (*drop* *j* *c*)

```

unfolding cycleG-def by(rule path-appendR[of take j c], auto)

definition cycleFrom where
cycleFrom nd ndl  $\equiv$  cycleG ndl  $\wedge$  hd ndl = nd

lemma cycle-rotate1:
cycleG (ndl @ [nd,nd'])  $\implies$  cycleG (nd # ndl @ [nd])
unfolding cycleG-def path-iff-set-nth apply safe
  subgoal by auto
  subgoal by auto
  subgoal for i apply (cases i)
  subgoal by simp (metis append.assoc append.left-neutral append-Cons length-append-singleton

    lessI list.sel(1) neq-Nil-conv nth-Cons-0 nth-append-length)
  subgoal by simp (metis Suc-lessI less-SucI nth-append nth-append-length) .
  subgoal by simp .

lemma cycle-rotate:
assumes cycleG (ndl1 @ ndl2 @ [nd']) ndl2  $\neq$  []
shows cycleG (ndl2 @ ndl1 @ [hd ndl2])
using assms proof(induction ndl2 arbitrary: ndl1 nd' rule: rev-induct)
  case Nil thus ?case by auto
next
  case (snoc nd2 ndl2 ndl1 nd')
  show ?case
  proof(cases ndl2 = [])
    case True
    show ?thesis unfolding True apply simp
    apply(rule cycle-rotate1) using snoc True by auto
  next
  case False
  hence 1: ndl2 @ nd2 # ndl1 @ [hd ndl2] = hd ndl2 # (tl ndl2 @ nd2 #
ndl1) @ [hd ndl2] by auto
  have 2: (tl ndl2 @ nd2 # ndl1) @ [hd ndl2, hd (tl ndl2 @ [nd2])] =
    (tl ndl2 @ [nd2]) @ (ndl1 @ [hd ndl2]) @ [hd (tl ndl2 @ [nd2])]
  by auto
  show ?thesis using False apply simp unfolding append-Cons[symmetric]
  apply(rule snoc.IH[where nd' = nd2])
  subgoal unfolding append-Cons unfolding append-assoc[symmetric] ap-
ply(rule cycle-rotate1)
  using snoc by auto
  subgoal by auto .
qed
qed

lemma cycle-rotate-butlast:
assumes cycleG (ndl1 @ nd # ndl2) ndl1  $\neq$  [] ndl2  $\neq$  []
shows cycleG (nd # butlast ndl2 @ ndl1 @ [nd])
using assms cycle-rotate[where nd' = last (nd # ndl2)]

```

and ?ndl2.0 = butlast (nd # ndl2) **and** ?ndl1.0 = ndl1] **using** *assms* **by** *simp*

lemma *cycle-rotate-set*:

assumes *cycleG* ndl nd ∈ *set* ndl

shows ∃ ndl'. *set* ndl' = *set* ndl ∧ *cycleG* ndl' ∧ *hd* ndl' = nd ∧ *last* ndl' = nd ∧ *length* ndl' = *length* ndl

proof(*cases* *hd* ndl = nd ∨ *last* ndl = nd)

case *True* **thus** ?thesis **apply**(*intro* *exI*[*of* - ndl])

using *assms* **unfolding** *cycleG-def* **by** *auto*

next

case *False* **then obtain** ndl1 ndl2 **where** ndl: ndl = ndl1 @ nd # ndl2 ndl1 ≠ [] ndl2 ≠ []

by (*metis* *assms*(2) *hd-append* *last-ConsL* *last-append* *list.sel*(1) *list.simps*(3) *split-list*)

thus ?thesis **using** *cycle-rotate-butlast*[*OF* *assms*(1)[*unfolded* ndl(1)] ndl(2,3)]

apply(*intro* *exI*[*of* - nd # butlast ndl2 @ ndl1 @ [nd]]) **using** *assms* ndl

apply *safe*

subgoal **using** *in-set-butlastD* **by** *force*

subgoal **by** (*smt* *Un-iff* *append-butlast-last-id* *hd-append2* *hd-conv-nth* *in-set-conv-nth* *insert-iff*

last.simps *last-appendR* *length-greater-0-conv* *list.set*(2) *cycleG-def* *set-append*)

by *auto*

qed

lemma *cycle-set-pathCon*:

assumes *cy*: *cycleG* ndl **and** nd: {nd, nd'} ⊆ *set* ndl

shows *pathCon* nd nd'

proof—

obtain ndl' **where** ndl': *cycleG* ndl' *set* ndl' = *set* ndl *hd* ndl' = nd *last* ndl' = nd

by (*meson* *cy* *cycle-rotate-set* *insert-subset* nd)

show ?thesis **proof**(*cases* nd = nd')

case *True* **thus** ?thesis **using** *cycleG-def* ndl'(1) ndl'(3) *pathCon-def* **by** *auto*

next

case *False* **then obtain** ndl1 ndl2 **where** 1: ndl1 ≠ [] ndl' = ndl1 @ nd' # ndl2

by (*metis* *append-Nil* *insert-iff* *list.sel*(1) nd ndl'(2) ndl'(3) *split-list* *subset-eq*)

hence ndl' = ndl1 @ [nd'] @ ndl2 **by** *simp*

hence *pathG* (ndl1 @ [nd']) **using** ndl'(1) **unfolding** *cycleG-def*

by (*metis* 1(1) *append.assoc* *eq-iff* *path-appendL* *length-append-singleton* *length-greater-0-conv*

less-le *not-less-eq-eq* *numeral-2-eq-2*)

thus ?thesis **unfolding** *pathCon-def* **using** 1(1) 1(2) ndl'(3) **by** *auto*

qed

qed

lemma *cycle-iff-nth*:

cycleG ndl $\longleftrightarrow$

```

length ndl ≥ 2 ∧ ndl!0 = ndl!(length ndl - 1) ∧
(∀ i < length ndl. ndl!i ∈ Node) ∧ (∀ i. Suc i < length ndl ⟶ edge (ndl!i) (ndl!(Suc
i)))
unfolding cycleG-def path-iff-nth
by (metis One-nat-def hd-conv-nth last-conv-nth list.size(3) not-less pos2)

lemma cycleG-shape: cycleG nds ⟹ (∃ x. nds = [x,x] ∨ (∃ xs. length xs > 0 ∧
nds = [x] @ xs @ [x]))
unfolding cycleG-def
apply(rule exI[of - hd nds])
apply(cases butlast (tl nds))
subgoal apply(rule disjI1)
by (metis append.simps(1) append-butlast-last-id
butlast.simps(1,2) list.exhaust-sel
pathG-butlast-not-nil tl-last')
apply(rule disjI2)
apply(rule exI[of - butlast (tl nds)])
by (metis append.simps(1,2) append-butlast-last-id
bot-nat-0.not-eq-extremum butlast.simps(1)
hd-Cons-tl last-tl length-0-conv list.sel(2)
list.simps(3))

definition scg :: bool where
scg ≡ ∀ nd nd'. {nd,nd'} ⊆ Node ⟶ pathCon nd nd'

lemma scg-emp: Node = {} ⟹ scg unfolding scg-def by auto

lemma scg-two:
assumes Node = {nd,nd'}
shows scg ⟷ edge nd nd' ∧ edge nd' nd
proof safe
assume scg: scg
{fix nd nd' assume Node: Node = {nd,nd'}
with scg obtain ndl where pathG ndl nd = hd ndl nd' = last ndl
unfolding scg-def pathCon-def by auto
hence edge nd nd' apply(induction)
subgoal by simp
subgoal by (metis Graph.path-set Node hd-append2 insert-absorb insert-iff
last-in-set last-snoc not-path-Nil singleton-insert-inj-eq' subsetD) .
}
thus edge nd nd' edge nd' nd using assms by auto
next
assume 0: edge nd nd' edge nd' nd
show scg
unfolding scg-def proof safe
fix ndd ndd' assume 1: {ndd, ndd'} ⊆ Node
show pathCon ndd ndd'
proof(cases {ndd, ndd'} = {nd, nd'})

```

```

    case True
    show ?thesis unfolding pathCon-def
    apply(rule exI[of - [ndd, ndd']])
    apply safe
    apply (metis 0(1) 0(2) 1 pathG.Base True doubleton-eq-iff)
    by simp-all
  next
    case False hence e: ndd = ndd' and or: ndd = nd  $\vee$  ndd = nd' using 1
  assms by auto
    show ?thesis using or apply standard
    subgoal unfolding pathCon-def apply(rule exI[of - [ndd, nd', nd]])
    unfolding pathCon-def using 0 1 e assms path-Cons path-two-incl by auto
    subgoal unfolding pathCon-def apply(rule exI[of - [ndd, nd, nd']])
    using 0 1 e pathG.Base assms path-Cons by auto .
  qed
qed
qed

lemma scg-singl: Node = {nd}  $\implies$  scg  $\longleftrightarrow$  edge nd nd
by (metis insert-absorb2 scg-two)

lemma scg-iff-two:
assumes  $\exists nd\ nd'. nd \neq nd' \wedge \{nd, nd'\} \subseteq Node$ 
shows scg  $\longleftrightarrow$  ( $\forall nd\ nd'. \{nd, nd'\} \subseteq Node \wedge nd \neq nd' \longrightarrow pathCon\ nd\ nd'$ )
(is -  $\longleftrightarrow$  ?K)
proof safe
  assume 1: ?K
  show scg unfolding scg-def proof safe
    fix nd nd' assume 2:  $\{nd, nd'\} \subseteq Node$ 
    show pathCon nd nd'
    proof(cases nd = nd')
      case False thus ?thesis using 1 2 by auto
    next
      case True
      then obtain nd'' where nd'': nd''  $\in$  Node nd''  $\neq$  nd nd''  $\neq$  nd' using assms
    2 by auto
      have pathCon nd nd'' using 1 2 nd'' by auto
      moreover have pathCon nd'' nd' using 1 2 nd'' by auto
      ultimately show ?thesis using pathCon-trans by blast
    qed
  qed
qed (simp add: scg-def)

lemma scg-ex-path:
assumes scg and finite Node and Node  $\neq$  {}
shows  $\exists ndl. pathG\ ndl \wedge set\ ndl = Node$ 
proof-
  {fix S2 nd assume finite S2 and S2  $\subseteq$  Node
   hence  $\exists ndl. pathG\ ndl \wedge S2 \subseteq set\ ndl$ 

```

```

proof(induction)
  case empty thus ?case using assms pathCon-def scg-def by auto
next
  case (insert nd S2)
  then obtain ndl where ndl: pathG ndl S2 ⊆ set ndl by auto
  obtain ndl' where ndl': pathG ndl' hd ndl' = last ndl last ndl' = nd
  using insert assms(1) unfolding scg-def pathCon-def
  by simp (metis path-set last-in-set ndl(1) not-path-Nil subset-eq)

  show ?case apply(rule exI[of - ndl @ tl ndl'])
  apply safe
    subgoal by (metis path-append-last hd-Cons-tl ndl'(1) ndl'(2) ndl(1)
not-path-Nil)
    subgoal by simp (metis hd-Cons-tl last-in-set last-tl ndl'(1) ndl'(3)
not-path-Nil
not-path-singl)
    subgoal using ndl(2) by auto .
  qed
}
thus ?thesis using assms by (meson eq-iff path-set)
qed

```

lemma *scg-ex-cycle*:

assumes *scg* **and** *finite Node* **and** *Node ≠ {}*

shows $\exists ndl. \text{cycleG } ndl \wedge \text{set } ndl = \text{Node}$

proof–

```

  obtain ndl where ndl: pathG ndl ∧ set ndl = Node using scg-ex-path[OF assms]
by auto
  obtain ndl' where ndl': pathG ndl' hd ndl' = last ndl last ndl' = hd ndl
  using insert assms(1) unfolding scg-def pathCon-def
  by simp (smt last-in-set list.set-sel(1) ndl not-path-Nil)
  have 1: pathG (ndl @ tl ndl')
  by (metis append-Nil2 path-append-last hd-Cons-tl ndl ndl'(1) ndl'(2) tl-Nil)
  show ?thesis apply(rule exI[of - ndl @ tl ndl'])
  unfolding cycleG-def apply safe
    subgoal by fact
    subgoal by (metis append-Nil2 append-butlast-last-id hd-Cons-tl
hd-append2 last-appendR last-tl ndl'(1) ndl'(2) ndl'(3) not-path-Nil)
    subgoal using 1 path-set by blast
    subgoal by (simp add: ndl) .
  qed

```

lemma *Graph-scg-restrict*:

Graph.scg N e $\longleftrightarrow$ *Graph.scg N* ($\lambda x y. e \ x \ y \wedge x \in N \wedge y \in N$)

unfolding *Graph.scg-def Graph.pathCon-def*

using *Graph-pathG-restrict* **by** *fastforce*

lemma *ex-cycle-scg*:

assumes *cycleG ndl set ndl = Node*
shows *scg*
using *assms cycle-set-pathCon scg-def* **by** *auto*

lemma *scg-iff-cycle*:
assumes *finite Node* **and** *Node $\neq \{\}$*
shows *scg $\longleftrightarrow (\exists ndl. cycleG ndl \wedge set ndl = Node)$*
using *assms ex-cycle-scg scg-ex-cycle* **by** *blast*

lemma *cycle-from-path*:
assumes *p: pathG ndl*
and *ij: $j < i$ $i < length\ ndl$ $ndl!!i = ndl!!j$*
shows *cycleG (drop j (take (Suc i) ndl))*
unfolding *cycleG-def* **apply** *safe*
subgoal using *p ij unfolding path-iff-nth* **by** *auto*
subgoal
by (*metis drop-eq-Nil dual-order.strict-trans2 hd-Nil-eq-last*
hd-drop-conv-nth ij(1) ij(2) ij(3) last-drop last-snoc
lessI linorder-not-less nth-take order-less-le take-Suc-conv-app-nth) .

lemma *finite-path-containsCycle*:
assumes *f: finite Node* **and** *p: pathG ndl* **and** *l: length ndl > card Node*
shows $\exists i\ j. i < length\ ndl \wedge j < i \wedge cycleG\ (drop\ j\ (take\ (Suc\ i)\ ndl))$
proof–
have *set ndl $\subseteq$ Node* **using** *Graph.path-set p* **by** *blast*
hence *card (set ndl) $\leq$ card Node* **by** (*simp add: card-mono f*)
then obtain *i j* **where** *1: $j < i$ $i < length\ ndl$ $ndl!!i = ndl!!j$* **using** *l*
by (*metis distinct-card distinct-conv-nth not-less-iff-gr-or-eq order-le-less*)
hence *cycleG (drop j (take (Suc i) ndl))*
using *Graph.cycle-from-path p* **by** *blast*
thus *?thesis* **using** *1* **by** *auto*
qed

definition *ipath :: 'node stream $\Rightarrow$ bool* **where**
ipath $\equiv$ alw (holdsS Node) aand alw (holds2 edge)

lemma *ipath-iff-snth*: *ipath nds $\longleftrightarrow (\forall i. nds!!i \in Node \wedge edge\ (nds!!i)\ (nds!!(Suc\ i)))$*
by (*meson alw-holds2-iff-snth alw-holdsS-iff-snth ipath-def*)

lemma *ipath-stake-path*:
ipath nds $\Longrightarrow n \geq 2 \Longrightarrow pathG\ (stake\ n\ nds)$
unfolding *ipath-iff-snth path-iff-set-nth*
using *path-iff-nth path-set* **by** *auto*

lemma *ipath-iff-stake-path*:
 $ipath\ nds \longleftrightarrow (\forall n \geq 2. pathG\ (stake\ n\ nds))$
apply *safe*
subgoal by (*simp add: ipath-stake-path*)
subgoal
by (*metis Suc-le-lessD alw-holds2-iff-snth alw-holdsS-iff-snth*
ipath-def length-stake nat-le-linear not-less-eq-eq path-iff-nth stake-nth) .

lemma *sset-ipath*: $ipath\ nds \implies sset\ nds \subseteq Node$
by (*metis imageE ipath-iff-snth sset-range subsetI*)

lemma *ipath-sdrop*:
 $ipath\ nds \implies ipath\ (sdrop\ n\ nds)$
unfolding *ipath-iff-snth* **using** *path-iff-nth path-set* **by** (*auto simp add: sdrop-snth*)

lemma *ipath-shift*: $local.ipath\ (v\ @- \ srepeat\ u) \implies local.ipath\ (srepeat\ u)$
apply (*drule ipath-sdrop[of v @- srepeat u length v]*)
using *sdrop-shift-length[of v length v]* **by** *auto*

lemma *ipath-stl*: $ipath\ r1 \implies local.ipath\ (stl\ r1)$ **using** *ipath-sdrop[of - 1]* **by** *auto*

lemma *ipath-scons*: $ipath\ (r\ \#\ r') \implies local.ipath\ (r')$ **using** *ipath-stl* **by** *force*

lemma *ipath-stake-sdrop-path*:
 $ipath\ nds \implies m \geq 2 \implies pathG\ (stake\ m\ (sdrop\ n\ nds))$
by (*auto intro: ipath-sdrop ipath-stake-path*)

lemma *ipath-stake-sdrop-cycle*:
 $ipath\ nds \implies m \geq 2 \implies nds!!n = nds!!(n+m-1) \implies cycleG\ (stake\ m\ (sdrop\ n\ nds))$
by (*simp add: hd-conv-nth ipath-stake-sdrop-path last-conv-nth cycleG-def sdrop-snth*)

lemma *ipath-pathCon*:
assumes *nds: ipath nds* **and** *ij: i < j*
shows *pathCon (nds!!i) (nds!!j)*
using *ipath-stake-sdrop-path* [**where** $n = i$ **and** $m = Suc(j-i)$, *OF* *nds*]
unfolding *pathCon-def* **apply** (*intro exI[of - stake (Suc (j - i)) (sdrop i nds)]*)
using *ij not-le* **apply** *safe*
subgoal by *auto*
subgoal by *auto*
subgoal by (*metis add-diff-cancel-left' last-snoc less-or-eq-imp-le nat-le-iff-add*
sdrop-snth stake-Suc) .

lemma *ipath-infinitely-often*:
assumes *Node: finite Node* **and** *nds: ipath nds*
shows $\exists nd \in Node. \forall i. \exists j \geq i. nds!!j = nd$
proof –

let $?f = (!)nds$
have $r: \text{range } ?f \subseteq \text{Node}$ **using** $\text{ipath-iff-snth } nds$ **by** auto
hence $\text{finite } (\text{range } ?f)$ **using** $\text{Node finite-subset}$ **by** auto
then obtain nd **where** $nd: nd \in \text{Node}$ **and** $\text{infinite } (?f - \{nd\})$
by $(\text{meson } r \text{ inf-img-fin-dom infinite-UNIV-char-0 subset-eq})$
hence $\forall i. \exists j \geq i. nds!!j = nd$
by $(\text{meson finite-nat-set-iff-bounded-le nat-le-linear vimage-singleton-eq})$
thus $?thesis$ **using** nd **by** auto
qed

lemma $\text{cycle-srepeat-ipath}$:
assumes $\text{cycleG } ndl$ **shows** $\text{ipath } (\text{srepeat } (\text{butlast } ndl))$
proof–
have $ndl: \text{butlast } ndl \neq [] \wedge \text{length } ndl \geq 2$
using $\text{assms not-path-Nil path-length-ge2}$ **unfolding** cycleG-def
by $(\text{metis append-Nil append-butlast-last-id not-path-singl})$
show $?thesis$ **using** $\text{assms } ndl$ **unfolding** $\text{cycle-iff-nth ipath-iff-snth}$ **apply** safe
subgoal for i **apply** $(\text{cases } i \bmod (\text{length } ndl - \text{Suc } 0) = 0)$
subgoal by $(\text{simp add: nth-butlast})$
subgoal by $\text{simp } (\text{metis One-nat-def Suc-le-lessD diff-le-self length-butlast less-le-trans mod-less-divisor nth-butlast numeral-2-eq-2 zero-less-diff})$.
subgoal for i **apply** $(\text{cases } i \bmod (\text{length } ndl - \text{Suc } 0) = 0)$
subgoal by $\text{simp } (\text{metis One-nat-def Suc-le-lessD length-butlast mod-Suc mod-less-divisor nth-butlast numeral-2-eq-2 zero-less-diff})$
subgoal by $\text{simp } (\text{smt One-nat-def Suc-le-lessD Suc-less-eq Suc-pred length-butlast less-le-trans mod-Suc mod-less-divisor nth-butlast numeral-2-eq-2 pos2})$. .
qed

lemma cycle-repeat :
assumes $ndl: \text{cycleG } ndl$ **and** $n: n \neq 0$
shows $\text{cycleG } (\text{repeat } n (\text{butlast } ndl) @ [\text{last } ndl])$
proof–
have $ndl \neq [] \wedge \text{length } ndl \geq 2$
using $\text{cycleG-def assms cycle-iff-nth not-path-Nil}$ **by** blast
thus $?thesis$ **using** assms **unfolding** cycle-iff-nth **apply** safe
subgoal by auto
subgoal by $\text{simp } (\text{smt One-nat-def nth-repeat Suc-le-lessD Suc-pred append-butlast-last-id append-is-Nil-conv hd-append2 hd-conv-nth length-butlast length-greater-0-conv nth-append-length numeral-2-eq-2 repeat-Suc zero-less-diff})$
subgoal for i **apply** $(\text{cases } i < \text{length } (\text{repeat } n (\text{butlast } ndl)))$
subgoal apply $(\text{subst nth-append})$
by $\text{simp } (\text{metis One-nat-def in-set-butlastD length-butlast nth-mem nth-repeat path-iff-nth path-iff-set-nth set-repeat subset-code(1)})$

```

    subgoal apply(subst nth-append) by (simp add: last-conv-nth) .
  subgoal for i apply(cases i < length (repeat n (butlast ndl)))
  subgoal apply(subst nth-append)
  apply simp apply(subst repeat-nth)
  subgoal by simp
  subgoal apply simp apply(subst nth-append)
  by simp (metis (no-types, lifting) ipath-iff-snth One-nat-def Suc-le-lessD
    Suc-mono cycle-srepeat-ipath last-conv-nth length-butlast length-greater-0-conv
    less-Suc-eq mod-mult-self2-is-0 ndl nth-butlast numeral-2-eq-2
    repeat-nth-eq-srepeat-snth srepeat-snth zero-less-diff) .
  subgoal apply(subst nth-append) by simp . .
qed

```

definition *subgr* where

subgr Node1 edge1 Node2 edge2 $\equiv$ Node1 $\subseteq$ Node2 $\wedge$ ($\forall$ nd nd'. edge1 nd nd' $\longrightarrow$ edge2 nd nd')

lemma *path-subgr*:

subgr Node1 edge1 S2 R2 $\implies$ Graph.pathG Node1 edge1 ndl $\implies$ Graph.pathG S2 R2 ndl

unfolding Graph.path-iff-nth Graph.subgr-def **by** auto

lemma *cycle-subgr*:

subgr Node1 edge1 S2 R2 $\implies$ Graph.cycleG Node1 edge1 ndl $\implies$ Graph.cycleG S2 R2 ndl

by (simp add: Graph.cycleG-def Graph.path-subgr)

lemma *ipath-subgr*:

subgr Node1 edge1 S2 R2 $\implies$ Graph.ipath Node1 edge1 nds $\implies$ Graph.ipath S2 R2 nds

unfolding Graph.ipath-iff-snth Graph.subgr-def **by** auto

fun scsg :: 'node set $\Rightarrow$ ('node $\Rightarrow$ 'node $\Rightarrow$ bool) $\Rightarrow$ bool **where**

scsg Node1 edge1 $\longleftrightarrow$ subgr Node1 edge1 Node edge $\wedge$ Graph.scg Node1 edge1

lemmas scsg-def = scsg.simps[simp del]

lemma scsg-paths:scsg V' E' $\implies$ ($\forall$ nd nd'.

{nd, nd'} $\subseteq$ V' $\longrightarrow$

Graph.pathCon V' E' nd nd')

unfolding scsg-def Graph.scg-def **by** auto

definition *maximal-scsg* where

maximal-scs $Node1$ $edge1 \equiv$
 $scsg\ Node1\ edge1 \wedge (\forall\ S2\ R2. scsg\ S2\ R2 \wedge subgr\ Node1\ edge1\ S2\ R2 \longrightarrow Node1$
 $= S2 \wedge edge1 = R2)$

definition $limitS :: 'node\ stream \Rightarrow 'node\ set$ **where**
 $limitS\ nds \equiv \{nd \in Node. alw\ (ev\ (holds\ ((=)\ nd)))\ nds\}$

definition $limitR :: 'node\ stream \Rightarrow ('node \Rightarrow 'node \Rightarrow bool)$ **where**
 $limitR\ nds \equiv \lambda nd\ nd'. alw\ (ev\ (holds2\ (\lambda ndd\ ndd'. ndd = nd \wedge ndd' = nd')))\ nds$

lemma $limitS\ sset$: $limitS\ nds \subseteq sset\ nds$ **unfolding** $limitS\ def$
by (*smt alwD ev-holds-sset mem-Collect-eq subsetI*)

lemma $ipath\ ev\ alw$:

assumes $Node$: *finite* $Node$ **and** nds : $ipath\ nds$

shows $ev\ (alw\ (holdsS\ (limitS\ nds)\ aand\ holds2\ (limitR\ nds)))\ nds$

proof–

define P **where** $P \equiv \{(nd, nd') \mid nd\ nd'. \{nd, nd'\} \subseteq Node \wedge$
 $\neg alw\ (ev\ (holds2\ (\lambda ndd\ ndd'. ndd = nd \wedge ndd' = nd')))\ nds\}$

have $\forall p \in P. \exists m. \forall i \geq m. (nds!!i, nds!!(Suc\ i)) \neq p$

unfolding $P\ def\ alw\ ev\ holds2\ iff\ snth$ **by** *auto*

then obtain f **where** $f: \forall p \in P. \forall i \geq f\ p. (nds!!i, nds!!(Suc\ i)) \neq p$

using $bchoice[of\ P\ \lambda p\ m. \forall i \geq m. (nds!!i, nds!!(Suc\ i)) \neq p]$ **by** *auto*

have $P \subseteq Node \times Node$ **unfolding** $P\ def$ **by** *auto*

hence fP : *finite* $(f\ 'P)$ **using** $Node\ finite\ subset$ **by** *blast*

define $i0$ **where** $i0 \equiv Max\ (f\ 'P)$

have $\forall i \geq i0. \forall p \in P. (nds!!i, nds!!(Suc\ i)) \neq p$

by (*metis Max-ge f dual-order.trans fP image-eqI i0-def*)

hence 1 : $\forall i \geq i0. limitR\ nds\ (nds!!i)\ (nds!!(Suc\ i))$

unfolding $limitR\ def\ P\ def$ **by** *simp (metis Graph.ipath-iff-snth nds snth.simps(2))*

define Q **where** $Q \equiv \{nd \in Node. \neg alw\ (ev\ (holds\ ((=)\ nd)))\ nds\}$

have $\forall q \in Q. \exists m. \forall i \geq m. nds!!i \neq q$

unfolding $Q\ def\ alw\ ev\ holds\ iff\ snth$ **by** *auto*

then obtain g **where** $g: \forall q \in Q. \forall i \geq g\ q. nds!!i \neq q$

using $bchoice[of\ Q\ \lambda p\ m. \forall i \geq m. nds!!i \neq p]$ **by** *auto*

have $Q \subseteq Node$ **unfolding** $Q\ def$ **by** *auto*

hence gQ : *finite* $(g\ 'Q)$ **using** $Node\ finite\ subset$ **by** *blast*

```

define  $j0$  where  $j0 \equiv \text{Max } (g \text{ ' } Q)$ 
have  $\forall i \geq j0. \forall q \in Q. \text{nds}!!i \neq q$ 
by (metis Max-ge g dual-order.trans gQ image-eqI j0-def)
hence  $2: \forall i \geq j0. (\text{nds}!!i) \in \text{limitS } \text{nds}$ 
unfolding limitS-def Q-def by (metis (mono-tags, lifting) mem-Collect-eq nds
snth-sset sset-ipath subsetD)

```

```

show ?thesis
using 1 2 unfolding ev-alw-holds2-aand-holdsS-iff-snth apply(intro exI[of - max
j0 i0]) by auto
qed

```

```

lemma limitS-infinite-visits:
assumes  $x \in \text{limitS } \pi$ 
shows  $\exists n \geq m. \pi !! n = x$ 
using assms unfolding limitS-def
using infinite-nat-iff-unbounded-le mem-Collect-eq
by (metis (mono-tags, lifting) alw-ev-holds-iff-snth)

```

```

lemma ipath-sdrop-limit:
assumes Node: finite Node and nds: ipath nds
shows  $\exists i. \text{Graph.ipath } (\text{limitS } \text{nds}) (\text{limitR } \text{nds}) (\text{sdrop } i \text{ nds})$ 
using ipath-ev-alw[OF assms]
unfolding ev-alw-holds2-aand-holdsS-iff-snth Graph.ipath-iff-snth sdrop-snth
by (metis add-Suc-right le-add1)

```

```

lemma limitR-sset:  $\text{limitR } \text{nds } nd \text{ } nd' \implies \{nd, nd'\} \subseteq \text{sset } \text{nds}$ 
unfolding limitR-def unfolding alw-ev-holds2-iff-snth
by (metis empty-subsetI insert-subset snth-sset)

```

```

lemma limitR-S:  $\text{ipath } \text{nds} \implies \text{limitR } \text{nds } nd \text{ } nd' \implies \{nd, nd'\} \subseteq \text{Node}$ 
using limitR-sset sset-ipath by blast

```

```

lemma limitS-S:  $\text{limitS } \text{nds} \subseteq \text{Node}$  unfolding limitS-def by auto

```

```

lemma limitR-R:  $\text{ipath } \text{nds} \implies \text{limitR } \text{nds } nd \text{ } nd' \implies \text{edge } nd \text{ } nd'$ 
unfolding ipath-iff-snth limitR-def alw-ev-holds2-iff-snth by auto

```

```

lemma scg-limit:
assumes Node: finite Node and nds: ipath nds
shows Graph.scg (limitS nds) (limitR nds)
unfolding Graph.scg-def proof safe
  fix  $nd \text{ } nd'$  assume  $0: \{nd, nd'\} \subseteq \text{limitS } \text{nds}$ 

```

```

obtain  $m$  where  $nds-m$ :  $Graph.ipath (limitS\ nds) (limitR\ nds) (sdrop\ m\ nds)$ 
using  $ipath-sdrop-limit[OF\ assms]$  by  $auto$ 

have  $1$ :  $alw\ (ev\ (holds\ ((=)nd)))\ nds\ alw\ (ev\ (holds\ ((=)nd')))\ nds$ 
using  $0$  unfolding  $limitS-def$  by  $auto$ 

obtain  $i$  where  $i \geq m$  and  $nd = nds!!i$ 
using  $1$  unfolding  $alw-ev-holds-iff-snth$  by  $blast$ 
hence  $nd$ :  $nd = (sdrop\ m\ nds)!!(i-m)$  by  $(simp\ add: sdrop-snth)$ 

obtain  $j$  where  $j > i$  and  $nd' = nds!!j$ 
using  $1$  unfolding  $alw-ev-holds-iff-snth$  using  $Suc-le-lessD$  by  $blast$ 
hence  $nd'$ :  $nd' = (sdrop\ m\ nds)!!(j-m)$ 
by  $(metis\ add-diff-cancel-left'\ i\ less-le-trans\ less-not-refl\ nat-le-iff-add\ nat-le-linear\ sdrop-snth)$ 

show  $Graph.pathCon (limitS\ nds) (limitR\ nds) nd\ nd'$ 
unfolding  $nd\ nd'$  apply $(rule\ Graph.ipath-pathCon)$  using  $nds-m\ i\ j$  by  $auto$ 
qed

```

```

lemma  $scsg-limit$ :
assumes  $Node$ :  $finite\ Node$  and  $nds$ :  $ipath\ nds$ 
shows  $scsg (limitS\ nds) (limitR\ nds)$ 
by  $(metis\ (full-types)\ Graph.subgr-def\ Node\ limitR-R\ limitS-S\ nds\ scg-limit\ scsg.elims(3))$ 

```

```

lemma
assumes  $finite\ Node$  and  $ipath\ nds$ 
shows  $\exists Node1\ edge1.$ 
 $scsg\ Node1\ edge1 \wedge$ 
 $ev\ (alw\ (holdsS\ Node1\ a\ and\ holds2\ edge1))\ nds \wedge$ 
 $(\forall nd\ nd'.\ edge1\ nd\ nd' \longrightarrow alw\ (ev\ (holds2\ (\lambda ndd\ ndd'.\ ndd = nd \wedge ndd' = nd')))\ nds)$ 
apply $(intro\ exI[of - limitS\ nds]\ exI[of - limitR\ nds])$ 
apply  $safe$ 
 $\quad$  subgoal using  $scsg-limit[OF\ assms]$   $.$ 
 $\quad$  subgoal using  $ipath-ev-alw[OF\ assms]$   $.$ 
 $\quad$  subgoal unfolding  $limitR-def$   $.$   $.$ 

```

```

lemma  $finite-limitS$ :
assumes  $Node$ :  $finite\ Node$  and  $nds$ :  $ipath\ nds$ 
shows  $finite (limitS\ nds)$ 
using  $Node\ finite-subset\ limitS-S$  by  $blast$ 

```

lemma *S-ne*:
assumes *nds*: *ipath nds*
shows $\text{Node} \neq \{\}$
using *ipath-iff-snth nds* **by** *blast*

lemma *R-ne*:
assumes *nds*: *ipath nds*
shows $\exists nd\ nd'. \{nd, nd'\} \subseteq \text{Node} \wedge \text{edge } nd\ nd'$
using *ipath-iff-snth nds* **by** *blast*

lemma *limitS-ne*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows $\text{limitS } nds \neq \{\}$
using *ipath-sdrop-limit[OF assms]* *Graph.S-ne* **by** *blast*

lemma *limitR-ne*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows $\exists nd\ nd'. \{nd, nd'\} \subseteq \text{limitS } nds \wedge \text{limitR } nds\ nd\ nd'$
using *ipath-sdrop-limit[OF assms]* *Graph.R-ne* **by** *blast*

lemma *finite-limitR*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *finite* $\{(nd, nd') . \text{limitR } nds\ nd\ nd'\}$
proof–
 have $\{(nd, nd') . \text{limitR } nds\ nd\ nd'\} \subseteq \text{Node} \times \text{Node}$
 unfolding *limitR-def alw-ev-holds2-iff-snth*
 using *Graph.ipath-iff-snth nds* **by** *blast+*
 thus *?thesis* **using** *Node infinite-super* **by** *blast*
qed

lemma *ipath-limitR-interval*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows $\exists j1 \geq i. \exists j2 \geq j1. \forall nd\ nd'. \text{limitR } nds\ nd\ nd' \longrightarrow (\exists j. j1 \leq j \wedge j < j2 \wedge nds!!j = nd \wedge nds!!(\text{Suc } j) = nd')$
proof–
 define *P* **where** $P \equiv \{(nd, nd') . \text{limitR } nds\ nd\ nd'\}$
 have $0: \bigwedge nd\ nd'. \text{limitR } nds\ nd\ nd' \longleftrightarrow (nd, nd') \in P$ **unfolding** *P-def* **by** *simp*
 have *fP*: *finite P* $P \neq \{\}$ **unfolding** *P-def*
 using *Node finite-limitR limitR-ne nds* **by** *fastforce+*
 have $\forall p \in P. \exists j \geq i. (nds!!j, nds!!(\text{Suc } j)) = p$
 unfolding *P-def* **by** (*simp add: Graph.limitR-def alw-ev-holds2-iff-snth*)
 then obtain *f* **where** $f: \forall p \in P. f\ p \geq i \wedge (nds!!(f\ p), nds!!(\text{Suc } (f\ p))) = p$
 using *bchoice[of P $\lambda p\ j. j \geq i \wedge (nds!!j, nds!!(\text{Suc } j)) = p$]* **by** *blast*

 define *j1 j2* **where** $j1 \equiv \text{Min } (f \text{ ' } P)$ **and** $j2 \equiv \text{Suc } (\text{Max } (f \text{ ' } P))$
 show *?thesis* **apply**(*rule exI[of - j1], safe*)
 subgoal **using** *f fP* **unfolding** *j1-def* **by** *simp*
 subgoal **apply**(*rule exI[of - j2], safe*)

```

    subgoal using f fP unfolding j1-def j2-def by (simp add: le-SucI)
    subgoal for nd nd' apply (rule exI[of - f (nd,nd')])
    using f fP unfolding j1-def j2-def 0 by (auto simp: le-imp-less-Suc) . .
qed

lemma limitS-sdrop-eq[simp]: limitS (sdrop n nds) = limitS nds
unfolding limitS-def alw-ev-holds-iff-snth
by (metis (no-types, opaque-lifting) sdrop-snth add-leE less-eqE nat-add-left-cancel-le
trans-le-add2)

lemma limitR-sdrop-eq[simp]: limitR (sdrop n nds) = limitR nds
unfolding limitR-def alw-ev-holds2-iff-snth sdrop-snth fun-eq-iff apply safe
using trans-le-add2 apply fastforce
by (metis add-Suc-right add-leE nat-add-left-cancel-le nat-le-iff-add)

end

```

end

2.2 Sloped Graphs

```

theory Sloped-Graphs
imports Directed-Graphs
begin

```

```

datatype slope = Main | Decr

```

```

lemma slope-exhaust'[simp]: c ≠ Decr ⟷ c = Main by (auto, meson slope.exhaust)

```

```

instantiation slope :: linorder
begin
fun less-eq-slope :: slope ⇒ slope ⇒ bool where
less-eq-slope Decr Main = False
| less-eq-slope - - = True
fun less-slope :: slope ⇒ slope ⇒ bool where
less-slope Main Decr = True
| less-slope - - = False

```

```

instance apply standard
subgoal for x y by (cases x, (cases y, auto)+)
subgoal for x by (cases x, auto)

```

```

subgoal for  $x\ y\ z$  by (cases  $x$ , (cases  $y$ , (cases  $z$ , auto))+)
subgoal for  $x\ y$  by (cases  $x$ , (cases  $y$ , auto)+)
subgoal for  $x\ y$  by (cases  $x$ , (cases  $y$ , auto)+)
done
end

```

definition *SlopedRels* $\equiv \{P . \forall h\ h' \ sl1\ sl2. P\ h\ h' \ sl1 \wedge P\ h\ h' \ sl2 \longrightarrow sl1 = sl2\}$

```

locale Sloped-Graph = Graph Node edge
for Node :: 'node set and edge :: 'node  $\Rightarrow$  'node  $\Rightarrow$  bool
+

```

```

fixes PosOf :: 'node  $\Rightarrow$  'pos set
and RR :: ('node  $\times$  'pos)  $\Rightarrow$  ('node  $\times$  'pos)  $\Rightarrow$  slope  $\Rightarrow$  bool
assumes Node-finite: finite Node
and PosOf-finite:  $\bigwedge nd. nd \in Node \Longrightarrow finite\ (PosOf\ nd)$ 
and RR-PosOf:
 $\bigwedge nd\ P\ nd'\ P'\ sl. RR\ (nd, P)\ (nd', P')\ sl \Longrightarrow \{nd, nd'\} \subseteq Node \wedge P \in PosOf\ nd \wedge P' \in PosOf\ nd'$ 
and RR-SlopeRels:  $\bigwedge nd\ nd'. \{nd, nd'\} \subseteq Node \Longrightarrow (\lambda P\ P'. RR\ (nd, P)\ (nd', P')) \in SlopedRels$ 
begin

```

lemma *finite-Node-opt*: finite ($\{r. \exists x \in Node. r = Some\ x\}$:: 'node option set)
apply(*rule* *finite-imageI*[*unfolded* *image-def*, *of* - *Some*]) **using** *Node-finite* **by** *auto*

lemma *RR-PosOfD:RR* (nd, P) (nd', P') *Main* $\vee$ *RR* (nd, P) (nd', P') *Decr* $\Longrightarrow nd \in Node \wedge nd' \in Node \wedge P \in PosOf\ nd \wedge P' \in PosOf\ nd'$
apply(*erule* *disjE*) **using** *RR-PosOf* **by** *auto*

lemma *RR-PosOfD':RR* (nd, P) (nd', P') $s \Longrightarrow nd \in Node \wedge nd' \in Node \wedge P \in PosOf\ nd \wedge P' \in PosOf\ nd'$
apply(*cases* s) **using** *RR-PosOf* **by** *auto*

lemma *alw-shd-stl:alw* (*holdsS* *Node*) $x \Longrightarrow shd(stl\ x) \in Node$ **by**(*drule* *alw-sdrop*[*of* - $x\ Suc\ 0$], *drule* *alwD*, *unfold* *holdsS-def*, *auto*)

definition *wfLabF* $S1\ lab \equiv \forall nd \in S1. lab\ nd \in PosOf\ nd$

definition *wfLabL* $ndl\ Pl \equiv length\ ndl = length\ Pl \wedge (\forall i < length\ ndl. Pl!i \in PosOf\ (ndl!i))$

definition *wfLabS* $nds\ Ps \equiv ev\ (alw\ (holds\ (\lambda(nd, P). P \in PosOf\ nd)))\ (szip\ nds\ Ps)$

definition *wfLabFS* $Node1\ lab \equiv \forall nd \in Node1. lab\ nd \neq \{\} \wedge lab\ nd \subseteq PosOf\ nd$

lemma *wfLabFS-finite*: $wfLabFS\ Node1\ lab \implies Node1 \subseteq Node \implies nd \in Node1 \implies finite\ (lab\ nd)$
unfolding *wfLabFS-def* **using** *infinite-super* **by** (*metis in-mono PosOf-finite*)

lemma *subgr-wfLabFS*:
 $subgr\ Node1\ edge1\ S2\ R2 \implies wfLabFS\ S2\ lab \implies wfLabFS\ Node1\ lab$
unfolding *subgr-def subsetD wfLabFS-def* **by** *auto*

lemma *wfLabS-iff-snth*:
 $wfLabS\ nds\ Ps \longleftrightarrow (\exists i. \forall j \geq i. Ps!!j \in PosOf\ (nds!!j))$
unfolding *wfLabS-def ev-akw-holds-iff-snth* **by** *auto*

lemma *path-wfLabF-wfLabL*:
assumes *pathG ndl and wfLabF Node lab*
shows *wfLabL ndl (map lab ndl)*
using *assms unfolding wfLabF-def wfLabL-def path-iff-nth* **by** *auto*

lemma *ipath-wfLabF-wfLabS*:
assumes *ipath (sdrop i nds) and wfLabF Node lab*
shows *wfLabS nds (smap lab nds)*
using *assms unfolding wfLabF-def wfLabS-iff-snth ipath-iff-snth sdrop-snth*
using *nat-le-iff-add* **by** *auto*

lemma *wfLabL-tl*: $ndl \neq [] \implies wfLabL\ ndl\ Pl \implies wfLabL\ (tl\ ndl)\ (tl\ Pl)$
unfolding *wfLabL-def* **by** (*simp add: nth-tl*)

lemma *wfLabL-append*:
 $length\ ndl1 = length\ Pl1 \implies length\ ndl2 = length\ Pl2 \implies$
 $wfLabL\ (ndl1\ @\ ndl2)\ (Pl1\ @\ Pl2) \longleftrightarrow wfLabL\ ndl1\ Pl1 \wedge wfLabL\ ndl2\ Pl2$
unfolding *wfLabL-def* **apply** *safe*
subgoal by (*metis length-append nth-append trans-less-add1*)
subgoal by (*metis length-append nat-add-left-cancel-less nth-append-length-plus*)
by (*auto simp: nth-append*)

lemma *wfLabL-append-inverse*:
assumes *wfLabL (ndl1 @ ndl2) Pl*
shows $\exists Pl1\ Pl2. Pl = Pl1\ @\ Pl2 \wedge wfLabL\ ndl1\ Pl1 \wedge wfLabL\ ndl2\ Pl2$
proof –
have *0: length (ndl1 @ ndl2) = length Pl*
 $length\ ndl1 = length\ (take\ (length\ ndl1)\ Pl)$
using *assms wfLabL-def* **by** *auto*
show *?thesis*
apply (*rule exI[of - take (length ndl1) Pl]*)
apply (*rule exI[of - drop (length ndl1) Pl]*)
using *assms unfolding wfLabL-def* **apply** *safe*

subgoal by *simp*
subgoal by *simp*
subgoal for *i* **by** (*metis* *length-append nth-append nth-take trans-less-add1*)
subgoal by *simp*
subgoal for *i* **by** (*metis* *0 append-take-drop-id length-append nat-add-left-cancel-less*
nth-append-length-plus) .
qed

lemma *cycle-wfLabL-repeat*:
assumes *ndl*: *cycleG ndl* **and** *w*: *wfLabL ndl Pl*
shows *wfLabL (repeat n (butlast ndl) @ [last ndl]) (repeat n (butlast Pl) @ [last Pl])*
proof –
have *aux*: *cycleG ndl* $\implies$ *ndl* $\neq []$ $\implies$ *length ndl* = *length Pl* $\implies$ $\forall i < \text{length } ndl.$
Pl ! i $\in$ *PosOf (ndl ! i)* $\implies$ $2 \leq \text{length } ndl \implies \text{length } Pl = \text{length } ndl \implies \text{length}$
(repeat n (butlast ndl) @ [last ndl]) = *length (repeat n (butlast Pl) @ [last Pl])* **by**
simp
have *ndl* $\neq []$ $\wedge$ *length ndl* ≥ 2 $\wedge$ *length Pl* = *length ndl*
using *cycleG-def* *assms cycle-iff-nth not-path-Nil wfLabL-def* **by** *metis*
thus *?thesis* **using** *assms unfolding wfLabL-def apply safe*
subgoal by *auto*
subgoal for *i* **apply** (*cases i < length (repeat n (butlast ndl))*)
subgoal unfolding *nth-append*
using *One-nat-def Suc-le-lessD Suc-pred length-butlast less-SucI*
less-le-trans mod-less-divisor
nth-butlast numeral-2-eq-2 pos2 repeat-nth zero-less-diff
by (*smt (verit) nth-repeat*)
subgoal unfolding *nth-append* **using** *One-nat-def diff-Suc-less*
last-conv-nth length-0-conv less-le-trans pos2
by (*smt (verit, ccfv-threshold) aux diff-self-eq-0 length-Suc-conv-rev linorder-less-linear*
not-less-eq nth-Cons') . .
qed

definition *descentIpath* :: '*node stream* $\Rightarrow$ '*pos stream* $\Rightarrow$ *bool* **where**
descentIpath nds Ps $\equiv$
*(ev (alw (holds2 ($\lambda(nd,P) (nd',P'). RR (nd,P) (nd',P') Main \vee RR (nd,P)$
 $(nd',P') Decr$)))*
aand
alw (ev (holds2 ($\lambda(nd,P) (nd',P'). RR (nd,P) (nd',P') Decr$))))
)
(szip nds Ps)

lemma *descentIpath-def2*:

$\text{descentIpath } nds \ Ps \longleftrightarrow$
 $(\text{ev } (\text{alw } (\text{holds2 } (\lambda(nd,P) \ (nd',P')). \ RR \ (nd,P) \ (nd',P') \ Main \ \vee \ RR \ (nd,P) \ (nd',P') \ Decr)))$
 aand
 $\text{alw } (\text{ev } (\text{holds2 } (\lambda(nd,P) \ (nd',P'). \ RR \ (nd,P) \ (nd',P') \ Decr)))$
 $)$
 $(\text{szip } nds \ Ps)$
by $(\text{smt alw-ev-sdrop alw-sdrop descentIpath-def ev-iff-sdrop})$

lemma $\text{descentIpath-iff-snth2}$:

$\text{descentIpath } nds \ Ps \longleftrightarrow$
 $(\exists i. \ \forall j \geq i. \ RR \ (nds!!j, Ps!!j) \ (nds!!(Suc \ j), Ps!!(Suc \ j)) \ Main \ \vee$
 $\quad \quad \quad RR \ (nds!!j, Ps!!j) \ (nds!!(Suc \ j), Ps!!(Suc \ j)) \ Decr)$
 $\wedge$
 $(\forall i. \ \exists j \geq i. \ RR \ (nds!!j, Ps!!j) \ (nds!!(Suc \ j), Ps!!(Suc \ j)) \ Decr)$
unfolding $\text{descentIpath-def2 ev-alw-holds2-iff-snth alw-ev-holds2-iff-snth}$ **by** simp

lemma $\text{descentIpath-iff-snth}$:

$\text{descentIpath } nds \ Ps \longleftrightarrow$
 $(\exists i. \ (\forall j \geq i. \ RR \ (nds!!j, Ps!!j) \ (nds!!(Suc \ j), Ps!!(Suc \ j)) \ Main \ \vee$
 $\quad \quad \quad RR \ (nds!!j, Ps!!j) \ (nds!!(Suc \ j), Ps!!(Suc \ j)) \ Decr)$
 $\wedge$
 $(\forall j \geq i. \ \exists k \geq j. \ RR \ (nds!!k, Ps!!k) \ (nds!!(Suc \ k), Ps!!(Suc \ k)) \ Decr))$
unfolding $\text{descentIpath-iff-snth2}$ **by** $(\text{meson add-leE order-refl})$

2.3 Infinite Descent

definition $\text{InfiniteDescent} :: \text{bool}$ **where**

$\text{InfiniteDescent} \equiv \forall nds. \ \text{ipath } nds \longrightarrow (\exists Ps. \ \text{descentIpath } nds \ Ps)$

lemma InfiniteDescentE : $\text{InfiniteDescent} \Longrightarrow \text{ipath } nds \Longrightarrow (\bigwedge Ps. \ \text{descentIpath } nds \ Ps \Longrightarrow P) \Longrightarrow P$ **unfolding** $\text{InfiniteDescent-def}$ **by** auto

lemma InfiniteDescentI : $(\bigwedge nds. \ \text{ipath } nds \Longrightarrow \exists Ps. \ \text{descentIpath } nds \ Ps) \Longrightarrow \text{InfiniteDescent}$ **unfolding** $\text{InfiniteDescent-def}$ **by** auto

lemma $\text{descentIpath-sdrop}$: $\text{descentIpath } (\text{sdrop } m \ nds) \ (\text{sdrop } m \ Ps) \longleftrightarrow \text{descentIpath } nds \ Ps$

unfolding descentIpath-def2 **unfolding** $\text{sdrop-szip[symmetric]}$
using $\text{ev-alw-aand-alw-ev-sdrop}$.

lemma descentIpath-stl : $\text{descentIpath } (\text{stl } nds) \ (\text{stl } Ps) \longleftrightarrow \text{descentIpath } nds \ Ps$
using $\text{descentIpath-sdrop[of Suc 0 nds Ps]}$ **by** auto

lemma $\text{descentIpath-wfLabS}$:

descentIpath nds Ps $\implies$ wfLabS nds Ps
by (*meson RR-PosOf descentIpath-iff-snth2 wfLabS-iff-snth*)

lemma *descentIpath-sdrop-any:*
descentIpath (sdrop m nds) Ps' $\implies \exists$ Ps. descentIpath nds Ps
apply(*rule exI[of - replicate m any @- Ps]*)
using *sdrop-shift-length*
by (*metis descentIpath-sdrop length-replicate*)

lemma *descentIpath-grow:descentIpath r1 Ps = descentIpath (x ## r1) (y ## Ps)*
using *descentIpath-sdrop[of Suc 0 x##r1 y##Ps] unfolding sdrop-1 stream.sel(2)*
by *auto*

lemma *ipath-stake-cycle:local.ipath (srepeat u) $\implies$*
2 $\leq$ length u $\implies$
srepeat u !! 0 = sprepeat u !! (length u - 1) $\implies$
cycleG u
using *ipath-stake-sdrop-cycle[of sprepeat u length u 0] unfolding stake-srepeat*
by *auto*

lemma *descentIpath-reduceAll: $\forall x. \neg$ descentIpath (v @- sprepeat u) x $\implies \forall x. \neg$*
descentIpath (srepeat u) x
apply *safe*
subgoal for *x*
apply(*erule allE[of - replicate (length v) (shd x) @- x]*)
using *descentIpath-sdrop[of length v (v @- sprepeat u) replicate (length v) (shd x) @- x]*
using *sdrop-shift-length[of v length v sprepeat u]*
unfolding *sdrop-replicate by auto .*

definition *descentPath :: 'node list $\Rightarrow$ 'pos list $\Rightarrow$ bool where*
descentPath ndl Pl $\equiv$
($\forall i. \text{Suc } i < \text{length ndl} \longrightarrow \text{RR } (\text{ndl}!!i, \text{Pl}!!i) (\text{ndl}!(\text{Suc } i), \text{Pl}!(\text{Suc } i)) \text{ Main } \vee$
RR (ndl!!i, Pl!!i) (ndl!(Suc i), Pl!(Suc i)) Decr) $\wedge$
($\exists i. \text{Suc } i < \text{length ndl} \wedge \text{RR } (\text{ndl}!!i, \text{Pl}!!i) (\text{ndl}!(\text{Suc } i), \text{Pl}!(\text{Suc } i)) \text{ Decr}$)

lemma *descentPath-length-wfLabL:*
descentPath ndl Pl $\implies \text{length Pl} = \text{length ndl} \implies \text{wfLabL ndl Pl}$
unfolding *descentPath-def wfLabL-def*
by (*metis (no-types, lifting) RR-PosOf Suc-less-eq2 less-antisym*)

lemma *cycle-descentIPath-srepeat-imp-descentPath:*

assumes 1: *cycleG* *ndl* **and** 2: *descentIpath* (*srepeat* (*butlast* *ndl*)) *Ps*
shows $\exists Pl. \text{wfLabL } ndl \ Pl \wedge \text{descentPath } ndl \ Pl$
proof –
have *ndl*: *ndl* $\neq []$ $\wedge$ *butlast* *ndl* $\neq []$ $\wedge$ *length* *ndl* ≥ 2
using 1 *not-path-Nil* *path-length-ge2* **unfolding** *cycleG-def*
by (*metis* *append-Nil* *append-butlast-last-id* *not-path-singl*)
let *?nds* = *srepeat* (*butlast* *ndl*)
obtain *i j* **where**
a: $\bigwedge j. j \geq i \implies$
 $RR \ (?nds \ !! \ j, \ Ps \ !! \ j) \ (?nds \ !! \ Suc \ j, \ Ps \ !! \ Suc \ j) \ Main \vee$
 $RR \ (?nds \ !! \ j, \ Ps \ !! \ j) \ (?nds \ !! \ Suc \ j, \ Ps \ !! \ Suc \ j) \ Decr$ **and**
b: $j \geq i + \text{length } ndl \ RR \ (?nds \ !! \ j, \ Ps \ !! \ j) \ (?nds \ !! \ Suc \ j, \ Ps \ !! \ Suc \ j) \ Decr$
using 2 *ndl* **unfolding** *wfLabS-iff-snth* *descentIpath-iff-snth2* **by** *blast*

define *l* **where** $l \equiv (j \text{ div } (\text{length } ndl - 1)) * (\text{length } ndl - 1)$
have *l*: $l \leq j \wedge j < l + (\text{length } ndl - Suc \ 0) \wedge l \geq i$
unfolding *l-def* **apply** *simp*
apply (*metis* *One-nat-def* *Suc-le-lessD* *add commute* *dividend-less-times-div* *mult commute*
ndl numeral-2-eq-2 *zero-less-diff*)
by (*smt* *le-diff-conv2* *One-nat-def* *add commute* *b(1)* *diff-le-self* *dividend-less-div-times*
lessI *nat-le-linear*
 $ndl \text{ not-less } numeral-2-eq-2 \text{ order.trans } zero-less-diff$)

define *Pl* **where** $Pl \equiv stake \ (\text{length } ndl) \ (sdrop \ l \ Ps)$

have *length* *Pl* = *length* *ndl* **unfolding** *wfLabL-def*
unfolding *Pl-def* **using** *ndl* **by** (*auto* *simp*: *sdrop-snth*)
hence 0: $\text{wfLabL } ndl \ Pl \wedge \text{descentPath } ndl \ Pl \longleftrightarrow \text{descentPath } ndl \ Pl$
using *descentPath-length-wfLabL* **by** *blast*

show *?thesis* **apply**(*rule* *exI*[*of* - *Pl*]) **unfolding** 0 **unfolding** *descentPath-def*
proof (*intro* *allI* *impI* *conjI*)

fix *k* **assume** *sk*: *Suc* *k* < *length* *ndl*
have [*simp*]: $i \leq l + k$ **by** (*simp* *add*: *l(3)* *trans-le-add1*)

have [*simp*]: *butlast* *ndl* ! $((l + k) \bmod (\text{length } ndl - Suc \ 0)) = ndl ! k$
by (*metis* *One-nat-def* *Suc-less-eq* *Suc-pred* *l-def* *length-butlast* *less-le-trans*
mod-less
 $mod-mult-self3 \ ndl \ nth-butlast \ pos2 \ sk$)
have [*simp*]: *butlast* *ndl* ! $(Suc \ (l + k) \bmod (\text{length } ndl - Suc \ 0)) = ndl ! (Suc \ k)$
apply(*cases* *Suc* *k* < *length* *ndl* - 1)
apply (*metis* *One-nat-def* *Suc-mod-mult-self3* *l-def* *length-butlast* *mod-if*
nth-butlast)
by (*metis* 1 *One-nat-def* *Suc-diff-1* *Suc-le-lessD* *Suc-mod-mult-self3* *cycle-iff-nth*
l-def *length-butlast* *less-SucE* *less-le-trans*
 $mod-self \ nth-butlast \ numeral-2-eq-2 \ pos2 \ sk \ zero-less-diff$)
have [*simp*]: $Ps \ !! \ (l + k) = Pl \ ! \ k$

```

by (simp add: Pl-def Suc-lessD sdrop-snth sk)
have [simp]: Ps !! Suc (l + k) = Pl ! Suc k
by (simp add: Pl-def sdrop-snth sk)

show RR (ndl ! k, Pl ! k) (ndl ! Suc k, Pl ! Suc k) Main ∨
      RR (ndl ! k, Pl ! k) (ndl ! Suc k, Pl ! Suc k) Decr
using ndl a[of l+k] by simp
next
define k where k: k ≡ j mod (length ndl - Suc 0)
show ∃ k. Suc k < length ndl ∧ RR (ndl ! k, Pl ! k) (ndl ! Suc k, Pl ! Suc k)
Decr
proof(rule exI[of - k], safe)
show skk: Suc k < length ndl
by (metis (no-types, lifting) k One-nat-def Suc-le-lessD Suc-less-eq Suc-pred
    less-le-trans mod-less-divisor ndl numeral-2-eq-2 pos2)

have [simp]: butlast ndl ! k = ndl ! k
by (metis k One-nat-def Suc-le-lessD length-butlast mod-less-divisor ndl
    nth-butlast numeral-2-eq-2 zero-less-diff)
have [simp]: Ps !! j = Pl ! k
unfolding k Pl-def using ndl
by (metis One-nat-def Suc-lessD skk div-mult-mod-eq k l-def sdrop-add sdrop-simps(1)
    stake-nth)
have [simp]: Ps !! Suc j = Pl ! Suc k
unfolding k Pl-def
by (metis (no-types, lifting) ndl One-nat-def Suc-le-lessD Suc-less-eq Suc-pred
    add-Suc-right
    div-mult-mod-eq l-def less-le-trans mod-less-divisor numeral-2-eq-2 pos2 sdrop-snth
    stake-nth)
have [simp]: butlast ndl ! (Suc j mod (length ndl - Suc 0)) = ndl ! (Suc k)
apply(cases Suc k < length ndl - Suc 0)
subgoal using k by auto (smt One-nat-def length-butlast mod-Suc-eq mod-if
    nth-butlast)
subgoal by (metis 1 k One-nat-def Suc-le-lessD cycle-iff-nth length-butlast
    mod-Suc
    mod-less-divisor nth-butlast numeral-2-eq-2 zero-less-diff) .

show RR (ndl ! k, Pl ! k) (ndl ! Suc k, Pl ! Suc k) Decr
using b ndl by (simp add: k[symmetric])
qed
qed
qed

```

definition *descentIpathS* :: 'node stream ⇒ 'pos stream ⇒ bool **where**
descentIpathS nds Ps ≡

$(\forall i. RR (nds !! i, Ps !! i) (nds !! Suc i, Ps !! Suc i) Main \vee$
 $RR (nds !! i, Ps !! i) (nds !! Suc i, Ps !! Suc i) Decr)$
 $\wedge$
 $(\forall i. \exists j \geq i. RR (nds !! j, Ps !! j) (nds !! Suc j, Ps !! Suc j) Decr)$

lemma *descentIpathS-imp-descentIpath*:
descentIpathS nds Ps $\implies$ descentIpath nds Ps
unfolding *descentIpathS-def descentIpath-iff-snth2* **by** *auto*

lemma *cycle-descentIPathS-srepeat-imp-descentPath*:
cycleG ndl $\implies$ descentIpathS (srepeat (butlast ndl)) Ps $\implies$
 $\exists Pl. wfLabL ndl Pl \wedge descentPath ndl Pl$
using *cycle-descentIPathS-srepeat-imp-descentPath descentIpathS-imp-descentIpath*
by *blast*

lemma *cycle-descentPath-imp-descentIPathS-srepeat*:
assumes *cycleG ndl and w: wfLabL ndl Pl and d: descentPath ndl Pl and*
hl: hd Pl = last Pl
shows $\exists Ps. descentIpathS (srepeat (butlast ndl)) Ps$
proof–

have $ndl \neq [] \wedge length\ ndl \geq 2$ **and** $lPl: length\ Pl = length\ ndl$
using *cycleG-def assms cycle-iff-nth not-path-Nil wfLabL-def* **apply** *metis*
using *w wfLabL-def* **by** *auto*
have $bndl: butlast\ ndl \neq []$
by (*metis cycleG-def not-path-Nil not-path-singl append-Nil*
 $append-butlast-last-id\ assms(1)$)
have $bPl: butlast\ Pl \neq []$
by (*metis assms(2) bndl length-0-conv length-butlast wfLabL-def*)

show *?thesis apply(intro exI[of - srepeat (butlast Pl)])*
unfolding *descentIpathS-def* **proof**(*intro conjI allI*)
fix i
have $0: \bigwedge i. i \bmod length(butlast\ ndl) < length(butlast\ ndl)$
 $\bigwedge i. i \bmod length(butlast\ Pl) < length(butlast\ Pl)$
apply (*metis assms(2) bndl length-butlast length-greater-0-conv*
 $mod-less-divisor\ wfLabL-def$)
by (*metis One-nat-def assms(2) bndl length-butlast length-greater-0-conv*
 $mod-less-divisor\ wfLabL-def$)
define j **where** $j: j \equiv i \bmod (length(butlast\ ndl))$
have $j': j = i \bmod length(butlast\ Pl)$
using *assms(2) j wfLabL-def* **by** *auto*
have $2: \bigwedge i. (Suc\ i) \bmod (length(butlast\ ndl)) \neq 0 \implies$
 $(Suc\ i) \bmod (length(butlast\ ndl)) = Suc(i \bmod (length(butlast\ ndl)))$
using *mod-Suc* **by** *auto*
have $3[simp]: (Suc\ i) \bmod (length(butlast\ ndl)) = 0 \implies length(butlast\ ndl) - 1$
 $= j$
by (*metis Zero-not-Suc diff-Suc-1 j mod-Suc*)

```

have lj[simp]: Suc j < length ndl
by (metis 0(1) Suc-eq-plus1 j length-butlast less-diff-conv)

show RR (srepeat (butlast ndl) !! i, sprepeat (butlast Pl) !! i)
  (srepeat (butlast ndl) !! Suc i, sprepeat (butlast Pl) !! Suc i) Main ∨
  RR (srepeat (butlast ndl) !! i, sprepeat (butlast Pl) !! i)
  (srepeat (butlast ndl) !! Suc i, sprepeat (butlast Pl) !! Suc i) Decr
unfolding sprepeat-snth[OF bndl] sprepeat-snth[OF bPl]
unfolding nth-butlast[OF 0(1)] nth-butlast[OF 0(2)]
unfolding j[symmetric] j'[symmetric]
apply(cases (Suc i) mod (length (butlast ndl)) = 0)
  subgoal using hl d[unfolded descentPath-def, THEN conjunct1, rule-format,
OF lj]
  by (metis 3 One-nat-def Suc-pred assms(1) bPl butlast.simps(1) cycle-iff-nth
    hd-conv-nth lPl last-conv-nth length-butlast length-greater-0-conv)
  subgoal using d unfolding descentPath-def
  by (metis One-nat-def j lPl length-butlast lj mod-Suc) .

have b:  $\bigwedge i. \text{butlast } ndl ! (i \bmod \text{length } (butlast \ ndl)) =$ 
   $ndl ! (i \bmod \text{length } (butlast \ ndl))$ 
using 0(1) nth-butlast by blast
obtain k where k: Suc k < length ndl
RR (ndl ! k, Pl ! k) (ndl ! Suc k, Pl ! Suc k) Decr
using d[unfolded descentPath-def] by auto
define l where  $l \equiv i * \text{length } (butlast \ ndl) + k$ 
have l: l mod length (butlast ndl) = k
unfolding l-def by simp (metis Suc-pred assms(1) cycle-iff-nth k(1) mod-less
  nat-le-linear not-less not-less-eq-eq numeral-2-eq-2)
have il:  $i \leq l$  unfolding l-def
by (metis add commute add-cancel-right-right bndl le-add1 le-less-linear
  length-0-conv mod-less mod-mult-self4 mult-is-0 trans-le-add2)

show  $\exists j \geq i. RR (srepeat (butlast \ ndl) !! j, sprepeat (butlast \ Pl) !! j)$ 
   $(srepeat (butlast \ ndl) !! Suc \ j, sprepeat (butlast \ Pl) !! Suc \ j) Decr$ 
apply(rule exI[of - l])
using k unfolding sprepeat-snth[OF bndl] sprepeat-snth[OF bPl]
unfolding nth-butlast[OF 0(1)] nth-butlast[OF 0(2)] l lPl using il
by (metis Graph.cycle-iff-nth b assms(1) hd-conv-nth hl l lPl last-conv-nth
  length-0-conv length-butlast mod-Suc not-less pos2)

qed
qed

lemma cycle-descentPath-repeat-imp-descentIPathS-srepeat:
assumes ndl: cycleG ndl and n:  $n \neq 0$  and w: wflabL (repeat n (butlast ndl) @
[last ndl]) Pl
and d: descentPath (repeat n (butlast ndl) @ [last ndl]) Pl and hd Pl = last Pl
shows  $\exists Ps. \text{descentIpathS } (srepeat (butlast \ ndl)) \ Ps$ 
proof-

```



```

define ndll where ndll  $\equiv$  repeat n (butlast ndl) @ [last ndl]
have 0: butlast ndll = repeat n (butlast ndl) unfolding ndll-def by simp
have  $\exists$  Ps. descentIpathS (srepeat (butlast ndll)) Ps
apply(rule cycle-descentPath-imp-descentIPathS-srepeat[of - Pl])
  subgoal by (simp add: cycle-repeat n ndl ndll-def)
  subgoal using w ndll-def by blast
  subgoal by (simp add: d ndll-def)
  subgoal by fact .
thus ?thesis unfolding 0 by (simp add: n srepeat-repeat)
qed

```

```

lemma srepeat-cycle-descentIpath-imp-descentIpath:
assumes ndl: cycleG ndl
and d: descentIpath (srepeat (butlast ndl)) Ps
shows  $\exists$  Ps. descentIpathS (srepeat (butlast ndl)) Ps
proof -
  define nds where nds  $\equiv$  srepeat (butlast ndl)
  define l where l = length ndl - Suc 0

  have ndl2: ndl  $\neq$  [] length ndl  $\geq$  2 butlast ndl  $\neq$  []
  using ndl cycleG-def not-path-Nil apply blast
  using cycle-iff-nth ndl apply blast
  by (metis cycleG-def not-path-Nil not-path-singl append.simps(1) append-butlast-last-id
ndl)

```

```

have l: length (butlast ndl) = l l > 0
using l-def length-butlast apply simp
using cycleG-def l-def ndl path-length-ge2 by fastforce

```

```

obtain k where nds:
 $\bigwedge i. i \geq k \implies RR (nds !! i, Ps !! i) (nds !! Suc\ i, Ps !! Suc\ i) Main \vee$ 
 $RR (nds !! i, Ps !! i) (nds !! Suc\ i, Ps !! Suc\ i) Decr$ 
 $\bigwedge i. \exists j \geq i. RR (nds !! j, Ps !! j) (nds !! Suc\ j, Ps !! Suc\ j) Decr$ 
using d unfolding descentIpath-iff-snth2 nds-def[symmetric] by auto

```

```

obtain n where nlk: n*l  $\geq$  k
by (metis One-nat-def l(2) mult.comm-neutral mult-le-mono2 not-less not-less-eq)

```

```

have nds-repeats:  $\bigwedge i. nds !! i = nds !! (n * l + i)$ 
using l(1) ndl2(3) nds-def by auto

```

```

define Qs where Qs  $\equiv$  sdrop (n * l) Ps
have nth-Qs:  $\bigwedge i. Qs !! i = Ps !! (n * l + i)$ 
by (simp add: Qs-def sdrop-snth)

```

```

show ?thesis apply(rule exI[of - Qs])
unfolding descentIpathS-def nds-def[symmetric] nth-Qs apply safe
  subgoal for i

```

```

unfolding nds-repeats[of  $i$ ] nds-repeats[of  $\text{Suc } i$ ]
by (metis nlk add-Suc-right dual-order.trans le-add1 nds(1))
subgoal for  $i$ 
using nds(2)[of  $n * l + i$ ] apply safe
  subgoal for  $j$ 
  apply(rule exI[of -  $j - n * l$ ])
  unfolding nds-repeats[of  $j - n * l$ ] nds-repeats[of  $\text{Suc } (j - n * l)$ ] by auto
  . .
qed

lemma cycle-descentIPathS-srepeat-imp-descentPath-repeat:
assumes ndl: cycleG ndl and d: descentIPathS (srepeat (butlast ndl)) Ps
shows  $\exists n \text{ Pl. } n \neq 0 \wedge \text{wfLabL } (\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}]) \text{ Pl} \wedge$ 
 $\text{descentPath } (\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}])$ 

Pl  $\wedge$  hd Pl = last Pl
proof -
  define nds where nds  $\equiv$  srepeat (butlast ndl)
  define nd where nd  $\equiv$  hd ndl
  define l where l = length ndl - Suc 0

  have ndl2: ndl  $\neq []$  length ndl  $\geq 2$  butlast ndl  $\neq []$ 
  using ndl cycleG-def not-path-Nil apply blast
  using cycle-iff-nth ndl apply blast
  by (metis cycleG-def not-path-Nil not-path-singl append.simps(1) append-butlast-last-id ndl)

  have l: length (butlast ndl) = l l > 0
  using l-def length-butlast apply simp
  using cycleG-def l-def ndl path-length-ge2 by fastforce

  have nds-repeats:  $\bigwedge n \text{ i. nds !! } (n * l + i) = \text{nds} !! i$ 
  using l(1) ndl2(3) nds-def by auto

  have snth-nds:  $\bigwedge ii. \text{nds} !! ii = \text{ndl} !! (ii \bmod l)$ 
  unfolding nds-def
  by (metis l(1) length-greater-0-conv mod-less-divisor ndl2(3) nth-butlast srepeat-snth)

  have bl-nds:  $\bigwedge ii. \text{butlast ndl} ! (ii \bmod l) = \text{nds} !! ii$ 
  using l(1) ndl2(3) nds-def by auto

  have nds:
   $\bigwedge i. \text{RR } (\text{nds} !! i, \text{Ps} !! i) (\text{nds} !! \text{Suc } i, \text{Ps} !! \text{Suc } i) \text{ Main} \vee$ 
 $\text{RR } (\text{nds} !! i, \text{Ps} !! i) (\text{nds} !! \text{Suc } i, \text{Ps} !! \text{Suc } i) \text{ Decr}$ 
 $\bigwedge i. \exists j \geq i. \text{RR } (\text{nds} !! j, \text{Ps} !! j) (\text{nds} !! \text{Suc } j, \text{Ps} !! \text{Suc } j) \text{ Decr}$ 
using d unfolding descentIPathS-def nds-def[symmetric] by auto

  have nd: nd = last ndl  $\bigwedge n. \text{nds} !! (n * l) = \text{nd}$ 
  subgoal using ndl unfolding nd-def cycleG-def by auto

```

subgoal by (*simp add: hd-conv-nth nd-def ndl2(1) snth-nds*) .

define *f* **where** $f \equiv \lambda n. Ps !! (n * l)$
have $range\ f \subseteq (\bigcup nd \in Node. PosOf\ nd)$
unfolding *f-def* **using** *RR-PosOf nds(1)* **by** *fastforce*
hence *finite (range f)* **by** (*meson Node-finite finite-UN-I finite-subset PosOf-finite*)
moreover **have** $UNIV = (\bigcup P. \{n. f\ n = P\})$ **by** *auto*
ultimately obtain *P* **where** $P: infinite\ \{n. f\ n = P\}$
by (*smt Collect-cong UNIV-I infinite-UNIV-char-0 pigeonhole-infinite*)
hence $P2: \forall n. \exists m > n. f\ m = P$
using *infinite-nat-iff-unbounded* **by** *auto*

obtain *k0* **where** $k0: Ps !! (k0 * l) = P$ **using** *P f-def not-finite-existsD* **by** *blast*
obtain *j0* **where** $j0: j0 \geq k0 * l$
 $RR\ (nds !! j0, Ps !! j0)\ (nds !! Suc\ j0, Ps !! Suc\ j0)\ Decr$
using *nds(2)* **by** *blast*
obtain *k1* **where** $aux: k1 > max\ k0\ (Suc\ j0)$ **and** $k1: Ps !! (k1 * l) = P$
using *P2 f-def* **by** *blast*
hence $k0 < k1\ j0 < k1 * l$
by *simp (metis aux One-nat-def Suc-lessD Suc-lessI dual-order.strict-trans l(2) max-less-iff-conj mult.right-neutral n-less-n-mult-m zero-less-Suc)*
note $k1 = k1\ this$

define *Pl* **where** $Pl \equiv stake\ (Suc\ ((k1 - k0) * l))\ (sdrop\ (k0 * l)\ Ps)$
define *n* **where** $n \equiv k1 - k0$

have $l-Pl[simp]: length\ Pl = Suc\ ((k1 - k0) * l)$
using *Pl-def length-stake* **by** *blast*

have $nth-Pl[simp]: \bigwedge i. i < Suc\ ((k1 - k0) * l) \implies Pl!i = Ps!!(k0 * l + i)$
unfolding *Pl-def*
by (*smt Suc-eq-plus1 Suc-le-lessD Suc-pred add.right-neutral leD linorder-neqE-nat*

not-less0 not-less-eq-eq nth-Cons'
plus-1-eq-Suc sdrop.simps(2) sdrop-add sdrop-snth stake-nth)

show *?thesis*
proof(*intro exI[of - Pl] exI[of - n], safe*)
show $0 < n$ **by** (*simp add: k1(2) n-def*)
show $wfLabL\ (repeat\ n\ (butlast\ ndl)\ @\ [last\ ndl])\ Pl$
unfolding *wfLabL-def* **apply** *safe*
subgoal by (*simp add: n-def l-def*)
subgoal for *i* **unfolding** *n-def* **apply**(*subst nth-Pl*)
subgoal by (*simp add: l-def*)
subgoal apply(*cases i < (k1 - k0) * (length ndl - Suc 0)*)
subgoal apply(*subst nth-append*) **apply**(*subst repeat-nth*)
subgoal by *simp*
subgoal apply *simp unfolding l-def[symmetric] bl-nds*
by (*metis RR-PosOf nds(1) nds-repeats*) .

```

    subgoal apply(subst nth-append)
    by simp (metis RR-PosOf add-mult-distrib l-def
        less-SucE nd(1) nd(2) nds(1)) . . .

show descentPath (repeat n (butlast ndl) @ [last ndl]) Pl
unfolding descentPath-def proof (intro conjI allI impI)
  fix i
  assume i: Suc i < length (repeat n (butlast ndl) @ [last ndl])
  hence ii: i < n * (length ndl - Suc 0) by simp
  hence T: i < n * (length (butlast ndl)) by simp
  show RR ((repeat n (butlast ndl) @ [last ndl]) ! i, Pl ! i)
    ((repeat n (butlast ndl) @ [last ndl]) ! Suc i, Pl ! Suc i) Main ∨
    RR ((repeat n (butlast ndl) @ [last ndl]) ! i, Pl ! i)
    ((repeat n (butlast ndl) @ [last ndl]) ! Suc i, Pl ! Suc i) Decr
  proof(cases Suc i < n * (length ndl - Suc 0))
    case True
    hence TT: Suc i < n * (length (butlast ndl)) by simp
    show ?thesis
    unfolding nth-append using T TT apply simp
    unfolding repeat-nth[OF T] repeat-nth[OF TT]
    unfolding bl-nds l(1) unfolding l-def[symmetric] n-def
    apply(subst nth-Pl, simp)+
    by (metis add-Suc-right nds(1) nds-repeats)
  next
  case False
  hence FF: i = n * (length (butlast ndl)) - Suc 0
  using T by auto
  show ?thesis unfolding nth-append using False T apply simp
  unfolding repeat-nth[OF T] unfolding bl-nds l(1)
  unfolding l-def[symmetric] n-def nd(1)[symmetric]
  apply(subst nth-Pl, simp)+
  by (metis Suc-lessI add-Suc-right nd(2) nds(1) nds-repeats)
  qed
next
have Suc (j0 - k0 * l) < n * length ndl - n
by (smt Nat.lessE One-nat-def Suc-diff-le Suc-mult-less-cancel1 aux diff-less-mono

    diff-mult-distrib diff-mult-distrib2 j0(1) k1(3) l(2)
    l-def le-SucI max-less-iff-conj mult.right-neutral n-def not-less-eq)
  hence [simp]: Suc (j0 - k0 * l) < n * (length ndl - Suc 0)
  by (simp add: right-diff-distrib')
  hence [simp]: j0 - k0 * l < n * (length ndl - Suc 0)
  by linarith
  have 0: Suc (j0 - k0 * l) < n * length (butlast ndl)
  j0 - k0 * l < n * length (butlast ndl)
  by auto
  have 1: Suc (j0 - k0 * l) < Suc ((k1 - k0) * l)
    using ⟨j0 - k0 * l < n * (length ndl - Suc 0)⟩ l-def n-def by blast
  hence 2: j0 - k0 * l < Suc ((k1 - k0) * l)

```

```

    by linarith
  show  $\exists i. \text{Suc } i < \text{length } (\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}]) \wedge$ 
     $RR ((\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}]) ! i, Pl ! i)$ 
     $((\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}]) ! \text{Suc } i, Pl ! \text{Suc } i) \text{ Decr}$ 
  apply (rule exI[of - j0 - k0 * l]) apply safe
  subgoal using diff-mult-distrib j0(1) k1(3) l-def n-def by auto
  subgoal unfolding nth-append apply simp
  unfolding repeat-nth[OF 0(1)] repeat-nth[OF 0(2)]
  unfolding nth-Pl[OF 1] nth-Pl[OF 2] apply (subst nth-butlast)
  subgoal using l(1) l(2) mod-less-divisor by blast
  subgoal apply (subst nth-butlast)
  subgoal using l(1) l(2) mod-less-divisor by blast
  subgoal unfolding l(1) snth-nds[symmetric]
  by (metis add-Suc-right j0(1) j0(2) le-add-diff-inverse nds-repeats)
  . . .
qed

have 0:  $hd \text{ Pl} = \text{Pl} ! 0$  last  $\text{Pl} = \text{Pl} ! ((k1 - k0) * l)$ 
apply (metis Zero-not-Suc hd-conv-nth l-Pl list.size(3))
by (metis diff-Suc-1 l-Pl last-conv-nth length-0-conv old.nat.distinct(2))
show  $hd \text{ Pl} = \text{last } \text{Pl}$  unfolding 0 unfolding Pl-def
apply (subst stake-nth)
subgoal by blast
subgoal apply (subst stake-nth)
subgoal by blast
unfolding sdrop-snth
by simp (metis add-diff-inverse-nat add-mult-distrib k0 k1(1) k1(2)
less-imp-triv)
.
qed
qed

definition RRSetChoice ::
   $'node \text{ set} \Rightarrow ('node \Rightarrow 'node \Rightarrow \text{bool}) \Rightarrow ('node \Rightarrow 'pos \text{ set}) \Rightarrow$ 
   $('node \Rightarrow 'node \Rightarrow 'pos \Rightarrow 'pos) \Rightarrow \text{bool}$  where
  RRSetChoice Node1 edge1 lab f  $\equiv$ 
   $(\forall nd \text{ nd}'. \{nd, nd'\} \subseteq \text{Node1} \longrightarrow f \text{ nd nd}' \text{ 'lab nd} \subseteq \text{lab nd}') \wedge$ 
   $(\forall nd \text{ nd}'. \{nd, nd'\} \subseteq \text{Node1} \wedge \text{edge1 nd nd}' \longrightarrow$ 
   $(\forall P \in \text{lab nd}. RR (nd, P) (nd', f \text{ nd nd}' P) \text{ Main} \vee RR (nd, P) (nd', f \text{ nd nd}' P)$ 
   $\text{Decr}))$ 

end

end

```

3 Equivalent Criteria for Infinite Descent

This subsection concerns two families of alternative criteria that are logically equivalent to Infinite Descent, and are used as the basis for decision procedures. While these criteria are already well-established, we provide the first mechanization within the sloped graph locale, and formal proofs of their soundness and completeness relative to the locale-level `InfiniteDescent` predicate.

3.1 Automata-based criteria

The first family of criteria reduces Infinite Descent to a language inclusion problem by interpreting (descending) ipaths as words in an ω -regular language [3, 1, 4, 2]. We formalize two approaches for constructing this interpretation, which (following the terminology of [2]) we refer to as the ‘vertex-language’ and ‘slope-language’ criteria, respectively.

3.1.1 Vertex-language Criterion

```

theory VLA-Criterion
  imports ../Sloped-Graphs
           ../Buchi-Preliminaries
begin

context Sloped-Graph
begin

abbreviation  $q_0$  where  $q_0 \equiv \text{None}$ 

fun  $\Delta\text{-trans} :: 'node \Rightarrow 'node\ option \Rightarrow 'node\ option\ set$  where
   $\Delta\text{-trans } tr (Some\ s) = (if\ edge\ s\ tr \wedge \{s, tr\} \subseteq Node\ then\ \{Some\ tr\}\ else\ \{\})$ 
   $\Delta\text{-trans } tr\ q_0 = (if\ tr \in Node\ then\ \{Some\ tr\}\ else\ \{\})$ 

lemma  $\Delta\text{-trans-}q_0: l \in Node \implies Some\ l \in \Delta\text{-trans } l\ q_0$  by auto

lemma  $\Delta\text{-trans-edge: edge } s\ tr \implies \{s, tr\} \subseteq Node \implies (Some\ tr) \in \Delta\text{-trans } tr$ 
   $(Some\ s)$  by auto

```

lemma Δ -trans-elim:
assumes $r \in \Delta$ -trans x q
obtains $(EdgeCase) \exists q'. q = Some\ q' \wedge edge\ q'\ x \wedge \{q', x\} \subseteq Node \wedge r = (Some\ x)$
 $\quad | (InitCase) q = q_0\ x \in Node\ r = (Some\ x)$
proof –
have $r \in \Delta$ -trans x q **using** *assms* **by** *assumption*
then consider
 $(EdgeCase) \exists q'. q = Some\ q' \wedge edge\ q'\ x \wedge \{q', x\} \subseteq Node \wedge r = (Some\ x) \mid$
 $(InitCase) q = q_0 \wedge x \in Node \wedge r = (Some\ x)$
by(*cases* q , *auto split: if-splits*)
then show *thesis* **using** *that* **by** (*cases*, *auto*)
qed

lemma Δ -trans $l\ s =$
 $\{s'. s = q_0 \wedge s' = (Some\ l) \wedge l \in Node\} \cup$
 $\{s'. \exists q'. s = Some\ q' \wedge s' = (Some\ l) \wedge edge\ q'\ l \wedge \{q', l\} \subseteq Node\}$
by(*cases* s , *auto*)

definition $Paut_V :: ('node, 'node\ option)\ nba$ **where**

$Paut_V = nba$
 $Node$
 $\{q_0\}$
 Δ -trans
 $(\lambda s. the\ s \in Node)$

lemmas $Paut_V$ -defs = $Paut_V$ -def Δ -trans.simps

lemma $Paut_V$ -alpha[simp]: $nba.alphabet\ Paut_V = Node$ **unfolding** $Paut_V$ -def **by** *auto*

lemma $Paut_V$ -accept[simp]: $nba.accepting\ Paut_V = (\lambda s. the\ s \in Node)$ **unfolding** $Paut_V$ -def **by** *auto*

lemma $Paut_V$ -init[simp]: $nba.initial\ Paut_V = \{q_0\}$ **unfolding** $Paut_V$ -def **by** *auto*

lemma $Paut_V$ -initp[intro]: $p \in nba.initial\ Paut_V \longleftrightarrow p = q_0$ **by** *auto*

lemma $Paut_V$ -trans[simp]: $nba.transition\ Paut_V\ a\ b = \Delta$ -trans $a\ b$ **unfolding** $Paut_V$ -def **by** *auto*

lemmas $Paut_V$ -trans' = $Paut_V$ -trans Δ -trans.simps

lemma $Paut_V$ -lang:NBA.language $Paut_V = \{nd. ipath\ nd\}$

proof(*safe*)

fix x

show $x \in NBA.language\ Paut_V \implies local.ipath\ x$

proof(*erule* $nba.language-elim$, *unfold* $ipath$ -def $Paut_V$ -initp $Paut_V$ -accept, *safe*)

fix $r\ p\ i$

assume $run:NBA.run\ Paut_V\ (x\ |||)\ r)\ q_0$ **and** $accept:infs\ (\lambda s. the\ s \in Node)\ r$

```

have sset  $x \subseteq \text{Node}$  using streams-sset[OF nba.run-alphabet[OF run]] by auto

thus alw (holdsS Node)  $x$  by (simp add: alw-holdsS-iff-snth subsetD)

show alw (holds2 edge)  $x$ 
proof (coinduct rule:alw-coinduct[of  $\lambda x. \exists r q. \text{NBA.run Paut}_V (x \parallel r) q$ ],safe)

  fix  $x'::\text{'node stream}$ 
  fix  $r'::\text{'node option stream}$ 
  fix  $q$ 

  show  $\exists r q. \text{NBA.run Paut}_V (x \parallel r) q$  using run streams-sset[OF nba.run-alphabet[OF run]] by auto

  show  $\text{NBA.run Paut}_V (x' \parallel r') q \implies \text{holds2 edge } x'$ 
  apply (unfold szip-unfold[of  $x' r'$ ], erule NBA.nba.run-scons-elim[of PautV],
safe)
  apply (unfold PautV-alpha PautV-trans szip-unfold[of stl  $x'$ ])
  apply (erule NBA.nba.run-scons-elim[of PautV])
  by (cases  $q$ , auto split: if-splits)

  {assume  $\neg \text{alw (holds2 edge) (stl } x') \text{ NBA.run Paut}_V (x' \parallel r') q$ 
  then show  $\exists r q. \text{NBA.run Paut}_V (stl x' \parallel r) q$ 
  by (cases  $(x' \parallel r')$ , auto simp: stl-sset subset-iff split: if-splits)}

  qed
qed
show ipath  $x \implies x \in \text{NBA.language Paut}_V$ 
proof (rule nba.language[of  $q_0 - x \text{ smap Some } x$ ])
  fix  $x$ 
  assume ipath:ipath  $x$ 

  hence  $x1\text{-prop:Some (shd } x) \in \Delta\text{-trans (shd } x) q_0 \text{ shd } x \in \text{Node}$ 
  using  $\Delta\text{-trans-}q_0$  alw-holdsS-iff-snth snth-0 ipath sset-ipath by force+

  show  $q_0 \in \text{nba.initial Paut}_V$  by auto

  define  $F\text{-valid}::(\text{'node} \times \text{'node option}) \text{ stream} \Rightarrow \text{'node option} \Rightarrow \text{bool}$  where
     $F\text{-valid} \equiv (\lambda r p. \exists r1. r = (r1 \parallel \text{smap Some } r1) \wedge \text{ipath } r1 \wedge \text{Some (shd } r1) \in \Delta\text{-trans (shd } r1) p)$ 

  have  $FF:F\text{-valid} (x \parallel \text{smap Some } x) q_0$  using  $x1\text{-prop ipath}$  by (auto simp:
 $F\text{-valid-def}$ )

  thm nba.run-coinduct
  thus  $\text{NBA.run Paut}_V (x \parallel \text{smap Some } x) q_0$ 
  apply (coinduct rule: nba.run-coinduct[of  $F\text{-valid}$ ])
  apply (unfold PautV-alpha PautV-trans fst-def snd-def  $F\text{-valid-def}$ ,safe)

```



```

subgoal using sset-ipath by auto
subgoal by simp
subgoal for - - - r1 apply(intro exI[of - stl r1] conjI, safe)
  subgoal by auto
  subgoal using ipath-stl by auto
  unfolding stream.map-sel(1)
  apply(rule  $\Delta$ -trans-edge)
    subgoal unfolding ipath-iff-snth by(erule allE[of - 0], auto)
    subgoal by (metis R-ne insert-subset sset-ipath stream.collapse stream.set)
  ..

show infs (nba.accepting PautV) (q0 ## smap Some x)
  unfolding alw-ev-scons apply(rule infs-all)
  using ipath sset-ipath[OF ipath] unfolding ipath-iff-snth
  by (auto)
qed
qed

fun  $\Delta$ -trans' :: 'node  $\Rightarrow$  ('node  $\times$  'pos  $\times$  slope) option  $\Rightarrow$ 
  ('node  $\times$  'pos  $\times$  slope) option set where
   $\Delta$ -trans' v' (Some tr) = (case tr of (v, p, s)  $\Rightarrow$  {Some (v', p', s') | p' s'. RR (v,
p) (v', p') s'}) |
   $\Delta$ -trans' v' q0 = (if v'  $\in$  Node then {q0}  $\cup$  {Some (v', p', s') | p' s'. p'  $\in$  PosOf
v'  $\wedge$  s' = Main} else {})

fun fsnd where fsnd (v,p,s) = (v, p)

lemma q0'-notDecr:third r = Decr  $\implies \neg$  (Some r)  $\in \Delta$ -trans' x q0 by(auto simp:
split: if-splits)

lemma  $\Delta$ -trans-q0'I:v  $\in$  Node  $\implies$  q0  $\in \Delta$ -trans' v q0 by auto

lemma  $\Delta$ -trans'-intro:
  assumes v'  $\in$  Node
  assumes tr = q0  $\implies$  x = q0  $\vee (\exists p' s'. x = \text{Some } (v', p', s') \wedge p' \in \text{PosOf } v' \wedge s' = \text{Main})$ 
  assumes tr  $\neq$  q0  $\implies (\exists p' s'. x = \text{Some } (v', p', s') \wedge \text{RR } (\text{fst}(\text{the } tr), \text{second}(\text{the } tr)) (v', p') s')$ 
  shows x  $\in \Delta$ -trans' v' tr
  using assms by (cases tr, auto)

lemma  $\Delta$ -trans'-elim:
  assumes v'  $\in \Delta$ -trans' vt q
  obtains
    ( $\Delta$ -trans1) q = q0 vt  $\in$  Node v' = q0
  | ( $\Delta$ -trans2) q = q0 vt  $\in$  Node  $\exists v'' p' s'. v' = \text{Some } (v'', p', s') \wedge p' \in \text{PosOf } v'' \wedge s' = \text{Main} \wedge v_t = v''$ 
  | ( $\Delta$ -trans3)  $\exists v p s v'' p' s'. q = \text{Some } (v, p, s) \wedge v' = \text{Some } (v'', p', s') \wedge \text{RR}$ 

```

$(v, p) (v'', p') s' \wedge v_t = v'' v_t \in Node$

proof –

from *assms* **consider**

$(\Delta\text{-trans}_1) q = q_0 \wedge v_t \in Node \wedge v' = q_0$

$| (\Delta\text{-trans}_2) \exists v'' p' s'. q = q_0 \wedge v_t \in Node \wedge v' = Some (v'', p', s') \wedge p' \in PosOf$
 $v'' \wedge s' = Main \wedge v_t = v''$

$| (\Delta\text{-trans}_3) \exists v p s v'' p' s'. q = Some (v, p, s) \wedge v' = Some (v'', p', s') \wedge RR$
 $(v, p) (v'', p') s' \wedge v_t = v'' \wedge v_t \in Node$

using *RR-PosOf* **by**(*cases q, auto split: if-splits*)

then show *thesis* **using** *that* **by** *cases auto*

qed

lemma $\Delta\text{-trans}'\text{-elim-}q_0'\text{-target}$:

assumes $q_0 \in \Delta\text{-trans}' v a$

obtains $q_0 = a v \in Node$

using *assms* **by**(*cases a, auto split: if-splits*)

lemma $\Delta\text{-trans}'\text{-elim-}q_0'$:

assumes $v' \in \Delta\text{-trans}' v_t q_0$

obtains

$(\Delta\text{-trans}_1) v_t \in Node v' = q_0$

$| (\Delta\text{-trans}_2) v_t \in Node \text{ second } (the v') \in PosOf (fst (the v')) \text{ third } (the v') =$
 $Main v_t = fst (the v')$

proof –

from *assms* **consider**

$(\Delta\text{-trans}_1) v_t \in Node \wedge v' = q_0$

$| (\Delta\text{-trans}_2) v_t \in Node \wedge \text{ second } (the v') \in PosOf (fst (the v')) \wedge \text{ third } (the v') =$
 $Main \wedge v_t = fst (the v')$

using *RR-PosOf* **by**(*auto split: if-splits*)

then show *thesis* **using** *that* **by** *cases auto*

qed

lemma $q_0'\text{-notReachable:} q \neq q_0 \implies v' \in \Delta\text{-trans}' v q \implies v' \neq q_0$ **by**(*cases v', cases q, auto*)

lemma $\Delta\text{-trans}'\text{-v-eq:} v' \neq q_0 \implies v' \in \Delta\text{-trans}' v q \implies fst (the v') = v$

apply(*cases v'*)

subgoal **by**(*cases q, auto*)

subgoal **by**(*cases q, auto split: if-splits*) .

lemma $\Delta\text{-trans}'\text{-Decr-not-}q_0\text{:} Some (v, p, Decr) \in \Delta\text{-trans}' vs q \implies \exists v' p s. q =$
 $Some (v', p, s) \wedge vs = v$

by(*cases q, auto split: if-splits*)

lemma $\Delta\text{-trans}'\text{-DecrRR}$:

$v' \neq None \implies$

$fst (the v') \in Node \implies$

$second\ (the\ v') \in PosOf\ (fst\ (the\ v')) \implies$
 $third\ (the\ v') = Decr \implies v' \in \Delta-trans'\ v\ q \implies$
 $RR\ (fst\ (the\ q),\ second\ (the\ q))\ (fst\ (the\ v'),\ second\ (the\ v'))\ Decr$
 $\wedge\ fst\ (the\ v') = v \wedge (fst\ (the\ q) \in Node \wedge second\ (the\ q) \in PosOf\ (fst\ (the\ q)) \wedge$
 $(third\ (the\ q) = Decr \vee third\ (the\ q) = Main))$
apply(cases v')
subgoal by (cases q , auto split:if-splits)
apply(erule $\Delta-trans'$ -elim) **using** $RR-PosOf\ slope.exhaust$ **by** (auto,meson slope.exhaust)

lemma $\Delta-trans'-DecrRR'$:
 $Some\ (v',p',Decr) \in \Delta-trans'\ v\ q \implies$
 $v' \in Node \implies p' \in PosOf\ v' \implies$
 $RR\ (fst\ (the\ q),\ second\ (the\ q))\ (v',\ p')\ Decr \wedge v' = v \wedge (fst\ (the\ q)$
 $\in Node \wedge second\ (the\ q) \in PosOf\ (fst\ (the\ q)))$
using $\Delta-trans'-DecrRR$ [of $Some\ (v',p',Decr)$] **by** auto

lemma $\Delta-trans'-ProgDRR$:
 $q \neq q_0 \implies$
 $fst\ (the\ q) \in Node \implies second\ (the\ q) \in PosOf\ (fst\ (the\ q)) \implies third$
 $(the\ q) = Decr \implies$
 $v' \in \Delta-trans'\ v\ q \implies$
 $RR\ (fst\ (the\ q),\ second\ (the\ q))\ (fst\ (the\ v'),\ second\ (the\ v'))\ (third$
 $(the\ v')) \wedge (third\ (the\ v') = Main \vee third\ (the\ v') = Decr) \wedge v' \neq q_0$
apply(cases v')
subgoal by (cases q , auto split:if-splits)
apply(erule $\Delta-trans'$ -elim) **by** auto

lemma $\Delta-trans'-ProgMRR$:
 $q \neq q_0 \implies$
 $v' \in \Delta-trans'\ v\ q \implies$
 $RR\ (fst\ (the\ q),\ second\ (the\ q))\ (fst\ (the\ v'),\ second\ (the\ v'))\ (third$
 $(the\ v')) \wedge (third\ (the\ v') = Main \vee third\ (the\ v') = Decr) \wedge v' \neq q_0$
apply(cases v')
subgoal by (cases q , auto split:if-splits)
apply(erule $\Delta-trans'$ -elim) **by** auto

lemma $\Delta-trans'-ProgMRR'-Main$:
 $x \in Node \implies y \in PosOf\ x \implies$
 $Some\ (x',y',Main) \in \Delta-trans'\ x'\ (Some\ (x,y,z)) \implies$
 $RR\ (x,\ y)\ (x',\ y')\ Main$
using $\Delta-trans'-ProgMRR$ [of $Some\ (x,y,z)\ Some\ (x',y',Main)\ x'$] $slope.exhaust$
by auto

definition $Q'-states::('node \times 'pos \times slope)\ option\ set$ **where**
 $Q'-states \equiv \{q_0\} \cup \{Some\ (v,\ p,\ s) \mid v\ p\ s.\ v \in Node \wedge p \in PosOf\ v\}$

definition $F\text{-valid}::('node \times 'pos \times slope) option \Rightarrow bool$ **where**
 $F\text{-valid} \equiv \lambda s. \exists r1\ r2. s = \text{Some } (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf } r1$

definition $Taut_V :: ('node, ('node \times 'pos \times slope) option) nba$ **where**
 $Taut_V = nba$
 Node
 $\{q_0\}$
 $\Delta\text{-trans}'$
 $F\text{-valid}$

lemma $RR\text{-red}:(\forall n \geq \text{Suc } 0. RR\ (r1 !! n, Ps !! n)\ (stl\ r1 !! n, stl\ Ps !! n)\ (stl\ Ss !! n)) \longleftrightarrow$
 $(\forall n. RR\ (stl\ r1 !! n, stl\ Ps !! n)\ (stl(stl\ r1) !! n, stl(stl\ Ps) !! n)\ (stl(stl\ Ss) !! n))$

apply *standard*

subgoal by *auto*

subgoal apply(*rule allI*)

subgoal for n **by**(*cases n, auto*) . .

lemma $Taut_V\text{-alpha}[simp]: nba.\text{alphabet } Taut_V = \text{Node}$ **unfolding** $Taut_V\text{-def}$ **by** *auto*

lemma $Taut_V\text{-accept}[simp]: nba.\text{accepting } Taut_V = (\lambda s. \exists r1\ r2. s = \text{Some } (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf } r1)$ **unfolding** $Taut_V\text{-def}$ $F\text{-valid}\text{-def}$ **by** *auto*

lemma $Taut_V\text{-init}[simp]: nba.\text{initial } Taut_V = \{q_0\}$ **unfolding** $Taut_V\text{-def}$ **by** *auto*

lemma $Taut_V\text{-initp}[intro]: p \in nba.\text{initial } Taut_V \longleftrightarrow p = q_0$ **by** *auto*

lemma $Taut_V\text{-trans}[simp]: nba.\text{transition } Taut_V\ a\ b = \Delta\text{-trans}'\ a\ b$ **unfolding** $Taut_V\text{-def}$ **by** *auto*

lemmas $\text{run-def} = nba.\text{run-alt-def-snth fst-nth-zip snd-nth-zip } Taut_V\text{-trans } Taut_V\text{-alpha}$
 $nba.\text{target-alt-def } nba.\text{states-alt-def}$

lemma $Taut_V\text{-lang}: NBA.\text{language } Taut_V = \{nds. (\exists Ps. \text{descentIpath } nds\ Ps) \wedge \text{alw } (\text{holdsS } \text{Node})\ nds\}$

proof(*safe*)

fix x

show $x \in NBA.\text{language } Taut_V \implies (\exists Ps. \text{descentIpath } x\ Ps)$

proof(*erule nba.language-elim, unfold Taut_V-accept singleton-iff*)

fix $r\ p$

assume $\text{init}: p \in nba.\text{initial } Taut_V$ **and** $\text{runs}: NBA.\text{run } Taut_V\ (x ||| r)\ p$ **and**
 $\text{infs } (\lambda s. \exists r1\ r2. s = \text{Some } (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf } r1)\ (p \#\# r)$

hence $p\text{-eq}: p = q_0$ **and** $\text{run}: NBA.\text{run } Taut_V\ (x ||| r)\ q_0$ **and** $\text{accept}: \text{infs } (\lambda s. \exists r1\ r2. s = \text{Some } (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf } r1)\ r$ **by** *auto*

have $\text{run}': \bigwedge k. x !! k \in \text{Node} \wedge r !! k \in \Delta\text{-trans}'\ (x !! k)\ (\text{last } (q_0 \# \text{map } \text{snd } (\text{stake } k\ (x ||| r))))$ **using** run **unfolding** run-def **by** *auto*

have $\text{accept-from}:\bigwedge n. \exists k \geq n. \exists r1\ r2. r \ !!\ k = \text{Some}\ (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf}\ r1 \wedge \text{Some}\ (r1, r2, \text{Decr}) \in \Delta\text{-trans}'\ (x \ !!\ k)\ (\text{last}\ (\text{stake}\ k\ r))$

subgoal for n
using $\text{accept}\ \text{unfolding}\ \text{infs-snth}$
apply-apply($\text{erule}\ \text{allE}[\text{of} - \text{Suc}\ n], \text{safe}$)
subgoal for k **using** $\text{runE-Suc}'[\text{OF}\ \text{run}, \text{of}\ k, \text{unfolded}\ \text{Taut}_V\text{-trans}]$
apply-by($\text{rule}\ \text{exI}[\text{of} - k], \text{auto}\ \text{simp}\ \text{add:}\ \text{last-stake-i}\ \text{stl-Suc}$) + . .

have $q_0' \text{-Decr}:\bigwedge r1\ r2\ k\ j. 0 < k \implies r \ !!\ k = \text{Some}\ (r1, r2, \text{Decr}) \implies r1 \in \text{Node} \implies r2 \in \text{PosOf}\ r1 \implies k \leq j \implies r \ !!\ j \neq q_0$

subgoal for $r1\ r2\ k\ j$ **proof**($\text{induct}\ j - k\ \text{arbitrary:}\ j$)

case 0

then show $?case$ **by** auto

next

case $(\text{Suc}\ xa)$

hence $\text{Suc}'\bigwedge j. 0 < k \implies xa = j - k \implies k \leq j \implies r \ !!\ j \neq q_0$ **by** auto

obtain n **where** $n:j = \text{Suc}\ n\ xa = n - k\ k \leq n$ **using** $\text{Suc}\ \text{by}(\text{cases}\ j, \text{auto})$

show $?case$

using $\text{Suc}'[\text{of}\ n, \text{OF}\ \text{Suc}(3)\ n(2,3)]$ **unfolding** n

using $\text{runE-Suc}[\text{OF}\ \text{run}, \text{of}\ n]$

using $q_0' \text{-notReachable}[\text{of}\ r \ !!\ n\ r \ !!\ \text{Suc}\ n]$ **by** auto

qed .

obtain Ps **where** $Ps:Ps = \text{smap}\ (\lambda r. \text{second}\ (\text{the}\ r))\ r$ **by** auto

have $x\text{-eq-r}:\bigwedge j. (\exists r1\ r2\ r3. r \ !!\ j = \text{Some}\ (r1, r2, r3)) \implies x \ !!\ j = \text{fst}\ (\text{the}\ (r \ !!\ j))$

subgoal for j **apply**($\text{cases}\ j$)

subgoal using $\text{runE-0}[\text{OF}\ \text{run}]$ **by** ($\text{simp}\ \text{split:}\ \text{if-splits}$)

apply safe

subgoal for $j'\ r1\ r2\ r3$

using $\text{runE-Suc}[\text{OF}\ \text{run}, \text{of}\ j', \text{unfolded}\ \text{Taut}_V\text{-trans}]$

apply-by($\text{erule}\ \Delta\text{-trans}'\text{-elim}, \text{auto}$) . .

have $x\text{-eq-r'}:\bigwedge j. (\exists r1\ r2\ r3. \text{stl}\ r \ !!\ j = \text{Some}\ (r1, r2, r3)) \implies \text{stl}\ x \ !!\ j = \text{fst}\ (\text{the}\ (\text{stl}\ r \ !!\ j))$

subgoal for j **using** $x\text{-eq-r}[\text{of}\ \text{Suc}\ j]$ **by** auto .

show $\exists Ps. \text{descentIpath}\ x\ Ps$

apply($\text{rule}\ \text{exI}[\text{of} - Ps], \text{unfold}\ \text{descentIpath-iff-snth2}, \text{intro}\ \text{conjI}$)

subgoal using $\text{accept}\ \text{unfolding}\ \text{infs-snth}$ **apply-apply**($\text{erule}\ \text{allE}[\text{of} - \text{Suc}\ 0], \text{safe}$)

subgoal for $k\ r1\ r2$ **apply**($\text{rule}\ \text{exI}[\text{of} - k], \text{intro}\ \text{allI}\ \text{impI}$)

subgoal for j **apply**($\text{drule}\ \text{Suc-le-lessD}$)

```

    apply(frule q0'-Decr[of k r1 r2 j], simp-all)
    apply(frule x-eq-r)
    using runE-Suc'[OF run zero-less-Suc, of j]
      Δ-trans'-ProgMRR[of r !! j stl r !! j fst (the (stl r !! j))]
    unfolding Ps by auto . .
  apply(rule allI)
  subgoal for i using accept-from[of Suc i] apply-apply(elim exE conjE)
  subgoal for k r1 r2 apply(cases k, simp)
    subgoal for n
      apply(frule Δ-trans'-Decr-not-q0, rule exI[of - n])
      unfolding Ps using x-eq-r[of n] last-take-Suc'[of - - r] by auto . . .
    qed
  next
  fix x
  show x ∈ NBA.language TautV ⇒ alw (holdsS Node) x
  proof(erule nba.language-elim, unfold TautV-init TautV-accept singleton-iff)
    fix r p
    assume p = q0 and run:NBA.run TautV (x ||| r) p
    thus alw (holdsS Node) x using streams-sset[OF nba.run-alphabet[OF run]] by
    (simp add: alw-holdsS-iff-snth subsetD)
  qed
  next
  fix x Ps
  assume iDPath:descentIpath x Ps and x-node:alw (holdsS Node) x

  have ev:ev (alw (holds2 (λ(nd, P) (nd', P'). RR (nd, P) (nd', P') Main ∨ RR
    (nd, P) (nd', P') Decr))) (x ||| Ps) and
    alw:alw (ev (holds2 (λ(nd, P) (nd', P'). RR (nd, P) (nd', P') Decr))) (x |||
    Ps)
    using iDPath[unfolded descentIpath-def2] by auto

  have hh:⋀x y z i. (stl x!!i, stl y!!i, stl z!!i) ∈ sset (x ||| y ||| z)
    by (metis snth-sset snth-szip stream.set-sel(2) szip.sel(2))

  define f where f = (λx' Ps' i. (if RR (x' !! i, Ps' !! i) (stl x' !! i, stl Ps' !! i)
    Decr then Decr else Main))
  define Ss where Ss = (λx Ps. Main ## fToStream (f x Ps))
  note Ss-defs = Ss-def f-def fToStream-def

  have Ss-i:⋀k m. m < Suc k ⇒ (Ss (sdrop m x) (sdrop m Ps) !! (Suc k - m))
    = f x Ps k using Suc-diff-le unfolding Ss-defs by auto

  obtain m where m:(⋀n. n ≥ m ⇒ (holds2 (λ(nd, P) (nd', P'). RR (nd, P)
    (nd', P') Main ∨ RR (nd, P) (nd', P') Decr)) (sdrop n (x ||| Ps)))
    using ev unfolding ev-alw-iff-sdrop by auto

  have alw-dropm:alw (ev (holds2 (λ(nd, P) (nd', P'). RR (nd, P) (nd', P')
    Decr))) (sdrop m (x ||| Ps))

```

```

using alw unfolding alw-ev-sdrop by auto

define r where r = replicate m q0 @- smap Some ((sdrop m x) ||| (sdrop m Ps) ||| (Ss (sdrop m x) (sdrop m Ps)))

have x-node':  $\bigwedge i. x !! i \in \text{Node}$  using x-node unfolding alw-holdsS-iff-snth by auto

have r-eq-q0':  $\bigwedge n. \text{Suc } n \leq m \longleftrightarrow r !! n = q_0$ 
apply standard unfolding r-def
subgoal using replicate-within-i by auto
subgoal for n apply(rule ccontr, unfold not-le replicate-beyond-i) using x-node'[of n] by auto .

have r-neq-q0':  $\bigwedge n. \text{Suc } n > m \longleftrightarrow r !! n = \text{Some } (x !! n, Ps !! n, Ss (sdrop m x) (sdrop m Ps) !! (n-m))$ 
apply standard
subgoal unfolding r-def using replicate-beyond-i by auto
subgoal for n using r-eq-q0'[of n] by auto .

have xn-gr:  $\bigwedge n. \text{Suc } n > m \implies x !! n = \text{fst } (\text{the } (r !! n))$  using r-neq-q0' by auto
have Psn-gr:  $\bigwedge n. \text{Suc } n > m \implies Ps !! n = \text{second } (\text{the } (r !! n))$  using r-neq-q0' by auto
have Ssn-gr:  $\bigwedge n. \text{Suc } n > m \implies \text{third } (\text{the } (r !! n)) = Ss (sdrop m x) (sdrop m Ps) !! (n-m)$  using r-neq-q0' by auto

have m-eq:  $\bigwedge k m. \text{Suc } k \leq m \implies m < \text{Suc } (\text{Suc } k) \implies m = \text{Suc } k$  by auto

show x  $\in \text{NBA.language Taut}_V$ 
proof(rule nba.language[of q0 - - r], unfold TautV-accept TautV-initp, safe)

thm ev-alw-sdrop alw-ev-sdrop
show NBA.run TautV (x ||| r) q0
proof(unfold run-def, intro allI conjI)
fix k
show x !! k  $\in \text{Node}$  using x-node' by (metis)

show r !! k  $\in \Delta\text{-trans}' (x !! k) (\text{last } (q_0 \# \text{map snd } (\text{stake } k (x ||| r))))$ 
proof(induct k)
case 0
then show ?case unfolding r-def apply(cases m)
subgoal using m[of 0] RR-PosOf[of shd x shd Ps shd (stl x) shd (stl Ps)]
unfolding Ss-def by (auto)
using  $\Delta\text{-trans-q0'I}[of x!!0, OF x-node'[of 0]]$  by auto
next
case (Suc k)

```

then show *?case*
unfolding *last-stake-szip* **apply—apply**(*erule* Δ -*trans'-elim* *exE*)

subgoal **apply**(*rule* Δ -*trans'-intro*[*OF* *x-node'*[*of Suc k*]])
subgoal using *le-less-linear*[*of Suc (Suc k) m*] **apply—apply**(*erule* *disjE*)
subgoal using *r-eq-q0'*[*of Suc k*] *x-node'*[*of Suc k*] **by** *auto*
subgoal unfolding *r-eq-q0'*[*symmetric, of k*] **apply**(*frule* *m-eq*[*of k m*], *clarify*)
using *r-neq-q0'*[*of Suc k*] *m*[*of Suc k*] *RR-PosOfD* *Ss-def* **by** *auto* .
by *auto*

subgoal **apply**(*rule* Δ -*trans'-intro*[*OF* *x-node'*[*of Suc k*]], *simp*)
using *r-neq-q0'*[*of Suc k*] *r-eq-q0'*[*of k*] *Ss-i*[*of m k*]
m[*of k*] *xn-gr*[*of k*] *xn-gr*[*of Suc k*] *Psn-gr*[*of k*] *Psn-gr*[*of Suc k*]
Ssn-gr[*of Suc k*]
unfolding *f-def* **by** *auto*

subgoal **apply**(*rule* Δ -*trans'-intro*[*OF* *x-node'*[*of Suc k*]], *simp*)
using *r-neq-q0'*[*of Suc k*] *r-eq-q0'*[*of k*] *Ss-i*[*of m k*]
m[*of k*] *xn-gr*[*of k*] *xn-gr*[*of Suc k*] *Psn-gr*[*of k*] *Psn-gr*[*of Suc k*]
Ssn-gr[*of Suc k*]
unfolding *f-def* **by** *auto* .

qed
qed

show *infs* ($\lambda s. \exists r1\ r2. s = \text{Some } (r1, r2, \text{Decr}) \wedge r1 \in \text{Node} \wedge r2 \in \text{PosOf } r1$) *r*
unfolding *r-def* *alw-ev-shift* *alw-ev-holds-iff-snth*
apply(*rule* *allI*)
subgoal for *i* **using** *alw-dropm* **unfolding** *alw-ev-holds2-iff-snth* *case-prod-beta* *Ss-defs*
apply—apply(*elim* *allE*[*of - i*] *exE* *conjE*)
subgoal for *j* **by**(*frule* *RR-PosOf*, *rule* *exI*[*of - Suc j*], *auto*) . .
qed
qed

lemma *alpha-subseq-PTaut_V*: *nba.alphabet Paut_V $\subseteq$ nba.alphabet Taut_V* **by** *simp*

lemma *Pnode-subseq-rule*: ($\bigwedge r. \forall x \in \text{Node}. \text{last } (\text{map } \text{snd } r) \neq \text{Some } x \implies$
 $r \neq [] \implies \text{NBA.path Paut}_V\ r\ \text{None} \implies \text{last } (\text{map } \text{snd } r) = \text{None} \implies$
 $(\bigcup p \in \{p. p \in \text{nba.initial Paut}_V\}. \{\text{last } (p \# \text{map } \text{snd } r) \mid r. \text{NBA.path Paut}_V\ r\ p\})$
 $\subseteq \{r. r = \text{None} \vee (\exists x \in \text{Node}. r = \text{Some } x)\}$) **by** *auto*

lemma *Paut_V-node-subseq*: *NBA.nodes Paut_V $\subseteq$ {*r*. *r* = None $\vee$ ($\exists x \in \text{Node}. r = \text{Some } x$)}*

unfolding *nba.nodes-alt-def nba.reachable-alt-def nba.target-alt-def nba.states-alt-def*

apply(*rule Pnode-subseq-rule*)

subgoal premises *p* **for** *r* **using** *p(3,1-2)* **apply**(*induct rule: nba.path.induct*)

subgoal by *auto*

subgoal for *a p* **by** (*cases p, auto simp: RR-PosOfD' split: if-splits*) . .

lemma *finite-Nodes-Paut_V:finite* (*NBA.nodes Paut_V*)

apply(*rule rev-finite-subset[OF - Paut_V-node-subseq]*) **using** *finite-Node-opt* **by** *auto*

lemma *Tnode-subseq-rule*: $(\bigwedge r. \forall a. a \in \text{Node} \longrightarrow (\forall aa. aa \in \text{PosOf } a \longrightarrow (\forall b. \text{last } (\text{map } \text{snd } r) \neq \text{Some } (a, aa, b))) \implies$

$r \neq [] \implies \text{NBA.path Taut}_V r \text{ None} \implies \text{last } (\text{map } \text{snd } r) = \text{None}$

$\implies (\bigcup_{p \in \{p. p \in \text{nba.initial Taut}_V\}. \{\text{last } (p \# \text{map } \text{snd } r) \mid r. \text{NBA.path Taut}_V r p\}}$

$\subseteq \{r. r = \text{None} \vee (\exists x \in \{(v, p, s) \mid v p s. v \in \text{Node} \wedge p \in \text{PosOf } v\}. r = \text{Some } x)\}$ **by** *auto*

lemma *Taut_V-node-subseq*:*NBA.nodes Taut_V* $\subseteq \{r. r = \text{None} \vee (\exists x \in \{(v, p, s) \mid v p s. v \in \text{Node} \wedge p \in \text{PosOf } v\}. r = \text{Some } x)\}$

unfolding *nba.nodes-alt-def nba.reachable-alt-def nba.target-alt-def nba.states-alt-def*

apply(*rule Tnode-subseq-rule*)

subgoal premises *p* **for** *r* **using** *p(3,1-2)* **apply**(*induct rule: nba.path.induct*)

subgoal by *auto*

subgoal for *a p* **by** (*cases p, auto simp: RR-PosOfD' split: if-splits*) . .

lemma *set-slope-case*: $\{(v, p, s) \mid v p s. v \in \text{Node} \wedge p \in \text{PosOf } v\} = \{(v, p, s) \mid v p s. v \in \text{Node} \wedge p \in \text{PosOf } v \wedge s \in \{\text{Main}, \text{Decr}\}\}$ **by** *auto*

lemma *finite-Node-Taut_V-gr:finite* ($\{r. \exists x \in \{(v, p, s) \mid v p s. v \in \text{Node} \wedge p \in \text{PosOf } v\}. r = \text{Some } x\}:: ('node \times 'pos \times \text{slope}) \text{ option set}$)

proof(*rule finite-imageI[unfolded image-def, of - Some], unfold set-slope-case*)

define *A* **where** *A* = $\{(v, p) \mid v p. v \in \text{Node} \wedge p \in \text{PosOf } v\}$

define *f*: $((('node \times 'pos) \times \text{slope}) \Rightarrow ('node \times 'pos \times \text{slope})$ **where** *f* = $(\lambda((n, p), s). (n, p, s))$

have *A-cart-eq*:*A* $\times \{\text{Main}, \text{Decr}\} = \{(n, p). n \in A \wedge p \in \{\text{Main}, \text{Decr}\}\}$ **by** *auto*

have *A* = $\bigcup ((\lambda v. \{v\} \times \text{PosOf } v) \text{ ` Node})$

unfolding *A-def* **by** *auto*

hence *finite A* **using** *Node-finite PosOf-finite* **by** (*simp add: finite-UN finite-cartesian-product*)

have *finiteFull:finite* (*A* $\times \{\text{Main}, \text{Decr}\}$)

using *finite-cartesian-product[OF ⟨finite A⟩, of {Main, Decr}]* **by** *simp*

have $S\text{-eq}:\{(v, p, s) \mid v \text{ pos } s. v \in \text{Node} \wedge p \in \text{PosOf } v \wedge s \in \{\text{Main}, \text{Decr}\}\} = f$
 $'(A \times \{\text{Main}, \text{Decr}\})$ **unfolding** $A\text{-def } f\text{-def case-prod-beta image-def}$ **by** *auto*

show $\text{finite } \{(v, p, s) \mid v \text{ pos } s. v \in \text{Node} \wedge p \in \text{PosOf } v \wedge s \in \{\text{Main}, \text{Decr}\}\}$ **unfolding**
 $S\text{-eq}$ **using** $\text{finite-imageI}[OF \text{ finiteFull, of } f]$ **by** *auto*
qed

lemma $\text{finite-Nodes-Taut}_V:\text{finite } (NBA.\text{nodes } \text{Taut}_V)$
apply($\text{rule rev-finite-subset}[OF \text{ Taut}_V\text{-node-subseq}]$) **using** $\text{finite-Node-Taut}_V\text{-gr}$
by *auto*

theorem $VLA\text{-Criterion}:\text{InfiniteDescent} \longleftrightarrow NBA.\text{language } Paut_V \subseteq NBA.\text{language } \text{Taut}_V$
unfolding $\text{Taut}_V\text{-lang } Paut_V\text{-lang InfiniteDescent-def ipath-def}$ **by** *auto*

corollary $VLA\text{-Criterion}':\text{InfiniteDescent} \longleftrightarrow NBA.\text{language } Paut_V \cap (NBA.\text{language } (\text{complement } \text{Taut}_V)) = \{\}$
unfolding $VLA\text{-Criterion}$ **by**($\text{rule complement-eq1, simp-all add: finite-Nodes-Taut}_V$)
end

end

3.1.2 Slope-language Criterion

theory $SLA\text{-Criterion}$
imports $../Sloped\text{-Graphs}$
 $../Buchi\text{-Preliminaries}$
begin

context $Sloped\text{-Graph}$
begin

abbreviation q_0 **where** $q_0 \equiv \text{None}$
fun $RR':: 'node \times 'node \Rightarrow 'pos \times 'pos \times \text{slope} \Rightarrow \text{bool}$ **where** $RR' (nd, nd') =$
 $(\lambda(ps, ps', s). RR (nd, ps) (nd', ps') s)$

fun $ndOf$ **where** $ndOf ((nd, ps), (nd', ps'), s) = nd$
fun $nd'Of$ **where** $nd'Of ((nd, ps), (nd', ps'), s) = nd'$

term $\{\text{Some } nd' \mid nd. \text{edge } nd \ nd'\}$

fun $\Delta_{sl}:: ('pos \times 'pos \times \text{slope} \Rightarrow \text{bool}) \Rightarrow 'node \text{ option} \Rightarrow 'node \text{ option set}$ **where**
 $\Delta_{sl} \ t \ (\text{Some } nd) = \{\text{Some } nd' \mid nd'. \{nd, nd'\} \subseteq \text{Node} \wedge \text{edge } nd \ nd' \wedge t =$
 $RR'(nd, nd') \}$

$\Delta_{sl} t q_0 = \{Some\ nd' \mid nd\ nd'. \{nd, nd'\} \subseteq Node \wedge edge\ nd\ nd' \wedge t = RR'(nd, nd')\}$

lemma Δ_{sl} -intro-Some:

assumes $edge\ s\ s'$
and $\{s, s'\} \subseteq Node$
and $tr = RR'(s, s')$
shows $Some\ s' \in \Delta_{sl}\ tr\ (Some\ s)$
using *assms* **by** *auto*

lemma Δ_{sl} -intro-q0:

assumes $edge\ s\ s'$
and $\{s, s'\} \subseteq Node$
and $tr = RR'(s, s')$
shows $Some\ s' \in \Delta_{sl}\ tr\ q_0$
using *assms* **by** *auto*

lemma Δ_{sl} -elim:

assumes $x \in \Delta_{sl}\ t\ z$
obtains $nd\ nd'$ **where**
 $x = Some\ nd' \ nd \in Node\ nd' \in Node\ edge\ nd\ nd'\ t = RR'(nd, nd')$
 $z = q_0 \vee the\ z = nd$
using *assms* **by** (*cases* *z*, *auto*)

lemma Δ_{sl} -q0-not-q0[simp]: $\neg (q_0 \in \Delta_{sl}\ x\ q_0)$ **by** *auto*

lemma Δ_{sl} -some: $r' \in \Delta_{sl}\ x\ r \implies \exists y. r' = Some\ y$ **by** (*cases* *r*, *auto*)

lemma $\Delta_{sl}\ l\ s =$

$\{Some\ v' \mid v\ v'. s = q_0 \wedge (edge\ v\ v') \wedge \{v, v'\} \subseteq Node \wedge l = RR'(v, v')\} \cup$
 $\{Some\ v' \mid v'. s \neq q_0 \wedge edge\ (the\ s)\ v' \wedge \{(the\ s), v'\} \subseteq Node \wedge l = RR'((the\ s), v')\}$
unfolding $\Delta_{sl}.simps$ **by** (*cases* *s*, *auto*)

definition $Paut_R :: ('pos \times 'pos \times slope \Rightarrow bool, 'node\ option)\ nba$ **where**

$Paut_R = nba$
 $\{RR'(nd, nd') \mid nd\ nd'. edge\ nd\ nd'\}$
 $\{q_0\}$
 Δ_{sl}
 $(\lambda s. the\ s \in Node)$

lemmas $Paut_R$ -defs = $Paut_R$ -def $\Delta_{sl}.simps$

lemma $Paut_R$ -alpha[simp]: $nba.alphabet\ Paut_R = \{RR'(nd, nd') \mid nd\ nd'. edge\ nd\ nd'\}$ **unfolding** $Paut_R$ -def **by** *auto*

lemma $Paut_R$ -accept[simp]: $nba.accepting\ Paut_R = (\lambda s. the\ s \in Node)$ **unfolding** $Paut_R$ -def **by** *auto*

lemma $Paut_R$ -init[simp]: $nba.initial\ Paut_R = \{q_0\}$ **unfolding** $Paut_R$ -def **by** *auto*

lemma $Paut_R$ -initp[intro]: $p \in nba.initial\ Paut_R \longleftrightarrow p = q_0$ **by** *auto*

lemma $Paut_R$ -trans[simp]: $nba.transition\ Paut_R\ a\ b = \Delta_{sl}\ a\ b$ **unfolding** $Paut_R$ -def

by *auto*

lemmas $Paut_R\text{-trans}' = Paut_R\text{-trans} \Delta_{sl}.sims$

lemmas $Paut_R\text{-run-def} = nba.\text{run-alt-def-snth} Paut_R\text{-trans} Paut_R\text{-alpha } nba.\text{target-alt-def} nba.\text{states-alt-def}$

lemma $Paut_R\text{-lang:NBA.language } Paut_R = \{Ri. \exists nds. ipath\ nds \wedge (\forall i. Ri \ !! \ i = RR' (nds \ !! \ i, nds \ !! \ Suc\ i))\}$

proof(*safe*)

fix x

assume $x \in NBA.\text{language } Paut_R$

then obtain r **where** $run:NBA.run\ Paut_R\ (x \ || \ r)\ None$ **and** $accept:infs\ (\lambda s. the\ s \in Node)\ r$ **apply-by**(*erule nba.language-elim, auto*)

have $q_0'\text{-notReachable}:\bigwedge k. r \ !! \ k \neq q_0$

subgoal for k **apply**(*induct k*)

subgoal using $runE\text{-}0[OF\ run]$ **apply-by**(*clarsimp*)

subgoal for k **using** $runE\text{-}Suc[OF\ run, of\ k]$ **by** *auto* .

have $x\text{-prop}:\bigwedge k. k > 0 \implies \exists nd\ nd'. x \ !! \ k = RR' (nd, nd') \wedge edge\ nd\ nd' \wedge the(r!!k) = nd' \wedge the(r!!(k-1)) = nd$

subgoal for k

using $runE\text{-}Suc[OF\ run, of\ k, unfolded\ Paut_R\text{-trans}]$

apply-apply(*drule runE-Suc'[OF run, of k, unfolded Paut_R-trans]*)

apply(*erule $\Delta_{sl}\text{-elim}$*)**+**

using $q_0'\text{-notReachable}[of]$ **by** *auto* .

have $q_0'\text{-notLast}:\bigwedge k. k > 0 \implies last\ (stake\ k\ r) \neq q_0$ **subgoal for** k

using $q_0'\text{-notReachable}[of\ k-1]$ $last\text{-conv-nth}[of\ stake\ k\ r]$ **by** *auto* .

hence $i\text{-eq-iff}:\bigwedge i. (if\ i = 0\ then\ None\ else\ last\ (stake\ i\ r)) = None \longleftrightarrow i = 0$
by *simp*

obtain s **where** $s\text{-edge}:edge\ s\ (the\ (r \ !! \ 0))$ **and** $s\text{-node}:s \in Node$ **and** $s\text{-x}:x \ !! \ 0 = RR'(s, the\ (r \ !! \ 0))$

using $runE\text{-}0[OF\ run]$ **by** *auto*

show $\exists nds. local.ipath\ nds \wedge (\forall i. x \ !! \ i = RR' (nds \ !! \ i, nds \ !! \ Suc\ i))$

proof(*rule exI[of - s ## smap the r], unfold ipath-def Paut_R-initp Paut_R-accept, safe*)

fix i

have $alw\ (holdsS\ Node)\ (smap\ the\ r)$ **unfolding** $alw\text{-holdsS-iff-snth}$ **apply-apply**(*rule allI*)

subgoal for i **apply**(*induct i*)

```

      subgoal using runE-0[OF run] by auto
      subgoal for i using runE-Suc[OF run, of i, unfolded PautR-trans] ap-
ply-by(erule  $\Delta_{sl}$ -elim, auto) . .

      thus alw (holdsS Node) (s ## smap the r)
      using s-node unfolding alw-holdsS-iff-snth by (metis not0-implies-Suc snth.simps(1)
snth-Stream stream.sel(1))

      have alwEdge:alw (holds2 edge) (smap the r) unfolding alw-holds2-iff-snth
apply-apply(rule allI)
      subgoal for i using runE-Suc[OF run, of i, unfolded PautR-trans]
      runE-Suc[OF run, of Suc i, unfolded PautR-trans]
      apply-apply(erule  $\Delta_{sl}$ -elim)+
      using q0'-notReachable by auto .

      thus alw (holds2 edge) (s ## smap the r)
      using s-edge unfolding alw-holds2-iff-snth apply-apply(rule allI)
      subgoal for i apply(cases i, simp)
      subgoal for n by (erule allE[of - n], auto) . .

      show x !! i = RR' ((s ## smap the r) !! i, (s ## smap the r) !! Suc i)
      using s-x x-prop[of i] by(cases i, auto)
    qed
  next
    fix x nds
    assume ipath:ipath nds and
      Rel: $\forall i. x !! i = RR' (nds !! i, nds !! Suc i)$ 
    obtain nd1 nd2 where nds:nds = nd1 ## nd2 by (cases nds)
    have Rel': $\bigwedge i. x !! i = RR' (nds !! i, nds !! Suc i) \wedge \text{edge } (nds !! i) (nds !! Suc i)$ 
    using Rel RR'.simps ipath unfolding alw-holds2-iff-snth ipath-def by auto

    have nds-in-Node:sset nds  $\subseteq$  Node by (metis ipath sset-ipath)
    hence stl-nds-in-Node:sset (stl nds)  $\subseteq$  Node using nds by auto

    show x  $\in$  NBA.language PautR
    proof(rule nba.language[of None PautR x smap Some (stl nds)])

      show q0  $\in$  nba.initial PautR by auto

      show NBA.run PautR (x ||| smap Some (stl nds)) q0
      proof(unfold PautR-run-def fst-nth-zip snd-nth-zip, intro allI conjI, safe)
        fix k
        show  $\exists nd nd'. x !! k = RR' (nd, nd') \wedge \text{edge } nd nd'$  using Rel' by auto
        show smap Some (stl nds) !! k  $\in \Delta_{sl} (x !! k)$  (last (None # map snd (stake
k (x ||| smap Some (stl nds))))))
        proof(induct k)
          case 0

```

```

      have last:(last (None # map snd (stake 0 (x ||| smap Some (stl nds))))) =
None by auto
    show ?case unfolding snth-smap last
      apply(rule  $\Delta_{sl}$ -intro-q0[of shd nds ])
      subgoal using Rel'[of 0] by auto
      subgoal using nds-in-Node stl-nds-in-Node by auto
      subgoal using Rel'[of 0] by auto .
  next
    case (Suc k)
    then show ?case unfolding last-stake-szip snth-smap
      apply-apply(erule  $\Delta_{sl}$ -elim, rule  $\Delta_{sl}$ -intro-Some)
      using Rel'[of Suc k] ipath ipath-iff-snth ipath-stl nds-in-Node by auto
  qed
qed

```

```

      have ruleInfs:infs (nba.accepting PautR) (smap Some (stl nds))  $\implies$  infs (nba.accepting
PautR) (q0 ## smap Some (stl nds))
      using alw-ev-shift[of - [None]] by auto

      show infs (nba.accepting PautR) (q0 ## smap Some (stl nds))
      apply(rule ruleInfs, rule infs-all)
      using stl-nds-in-Node by auto
  qed
qed

```

```

lemma PautR-lang-in:  $x \in \text{NBA.language } Paut_R \iff (\exists \text{nds. ipath nds} \wedge (\forall i. x !! i = RR'(\text{nds} !! i, \text{nds} !! \text{Suc } i)))$ 
  unfolding PautR-lang by auto

```

```

definition Q'sl::('pos  $\times$  slope) set where
Q'sl  $\equiv \{(p, s) \mid p \text{ s. } p \in (\bigcup v \in \text{Node. PosOf } v)\}$ 
abbreviation  $\Sigma \equiv \{RR'(\text{nd}, \text{nd}') \mid \text{nd nd'. edge nd nd'}\}$ 

```

```

fun  $\Delta_{sl}' :: ('pos \times 'pos \times \text{slope} \Rightarrow \text{bool}) \Rightarrow ('pos \times \text{slope}) \text{ option} \Rightarrow ('pos \times \text{slope})$ 
option set where
 $\Delta_{sl}' R' (\text{Some } (p, s)) = \{\text{Some } (p', s') \mid p' \text{ s'. } (p, s) \in Q'_{sl} \wedge R' \in \Sigma \wedge R' (p, p', s')\}$ 
 $\Delta_{sl}' R' q_0 = (\text{if } R' \in \Sigma \text{ then } \{q_0\} \cup \{\text{Some } (p', s') \mid p' \text{ s'. } (p', s') \in Q'_{sl}\} \text{ else } \{\})$ 

```

```

lemma  $\Delta_{sl}'$ -intro-Some:
  assumes  $(p, s) \in Q'_{sl} \wedge R' \in \Sigma \wedge R' (p, p', s')$ 
  shows  $\text{Some } (p', s') \in \Delta_{sl}' R' (\text{Some } (p, s))$ 
  using assms by auto

```

lemma Δ_{sl}' -intro-q0:
assumes $R' \in \Sigma$ $(p', s') \in Q'_{sl}$
shows $\text{Some } (p', s') \in \Delta_{sl}' R' q_0$
using *assms* **by** *auto*

lemma Δ_{sl}' -q0-included:
assumes $R' \in \Sigma$
shows $q_0 \in \Delta_{sl}' R' q_0$
using *assms* **by** *auto*

lemma Δ_{sl}' -intro:
assumes $ns \in \Sigma$
assumes $rk \neq q_0 \implies (\exists p' s'. rk' = \text{Some } (p', s') \wedge \text{the } rk \in Q'_{sl} \wedge ns \text{ (fst(the } rk), p', s'))$
assumes $rk = q_0 \implies (rk' = q_0 \vee (\exists p' s'. rk' = \text{Some } (p', s') \wedge (p', s') \in Q'_{sl}))$
shows $rk' \in \Delta_{sl}' ns rk$
using *assms* **apply** (*cases* rk) **by** *force+*

lemma Δ_{sl}' -elim:
assumes $x \in \Delta_{sl}' R' z$
obtains (*SomeCase*) $p s p' s'$ **where**
 $z = \text{Some } (p, s) \ x = \text{Some } (p', s') \ (p, s) \in Q'_{sl} \ R' \in \Sigma \ R' (p, p', s')$
| (*q0Case*) $p' s'$ **where**
 $z = q_0 \ R' \in \Sigma \ (p', s') \in Q'_{sl} \ x = \text{Some } (p', s')$
| (*q0Self*) $z = q_0 \ R' \in \Sigma \ x = q_0$
using *assms* **by** (*cases* z , (*auto split: if-splits*)+)

lemma q_0 -notReachable: $q \neq q_0 \implies v' \in \Delta_{sl}' v q \implies v' \neq q_0$ **by** (*cases* v' , *cases* q , *auto*)

definition $F_{sl} :: ('pos \times slope) \text{ option} \Rightarrow bool$ **where**
 $F_{sl} \equiv \lambda ps. \exists p. ps = \text{Some } (p, \text{Decr}) \wedge (p, \text{Decr}) \in Q'_{sl}$

definition $Taut_R :: ('pos \times 'pos \times slope \Rightarrow bool, ('pos \times slope) \text{ option}) \text{ nba}$ **where**
 $Taut_R = nba$
 $\{RR' \ (nd, nd') \mid nd \ nd'. \text{ edge } nd \ nd'\}$
 $\{q_0\}$
 Δ_{sl}'
 F_{sl}

lemma *RR-reduce*: $(\forall n \geq \text{Suc } 0. \text{RR } (r1 !! n, Ps !! n) (\text{stl } r1 !! n, \text{stl } Ps !! n) (\text{stl } Ss !! n)) \longleftrightarrow$
 $(\forall n. \text{RR } (\text{stl } r1 !! n, \text{stl } Ps !! n) (\text{stl}(\text{stl } r1) !! n, \text{stl}(\text{stl } Ps) !! n) (\text{stl}(\text{stl } Ss) !! n))$
apply *standard*
subgoal **by** *auto*
subgoal **apply**(*rule allI*)
subgoal **for** *n* **by**(*cases n, auto*) . .

lemma *Taut_R-alpha[simp]*:*nba.alphabet* *Taut_R* = $\{RR' (nd, nd') \mid nd \text{ nd}'. \text{edge } nd \text{ nd}'\}$ **unfolding** *Taut_R-def* **by** *auto*
lemma *Taut_R-accept[simp]*:*nba.accepting* *Taut_R* = $(\lambda ps. \exists p. ps = \text{Some } (p, \text{Decr}) \wedge (p, \text{Decr}) \in Q'_{sl})$ **unfolding** *Taut_R-def* *F_{sl}-def* **by** *auto*
lemma *Taut_R-init[simp]*: *nba.initial* *Taut_R* = $\{q_0\}$ **unfolding** *Taut_R-def* **by** *auto*
lemma *Taut_R-initp[intro]*: $p \in \text{nba.initial } Taut_R \longleftrightarrow p = q_0$ **by** *auto*
lemma *Taut_R-trans[simp]*:*nba.transition* *Taut_R* *a b* = $\Delta_{sl}' a b$ **unfolding** *Taut_R-def* **by** *auto*

lemmas *run-def'* = *nba.run-alt-def-snth fst-nth-zip snd-nth-zip Taut_R-trans Taut_R-alpha*
nba.target-alt-def nba.states-alt-def

fun *Rst* **where** *Rst nds* = *smap* $(\lambda i. \text{RR}' (nds !! i, nds !! \text{Suc } i)) \text{ nats}$

lemma *stl-shift*: $(\text{stl } nds !! i, \text{stl } (nds !! i)) = (nds !! \text{Suc } i, nds !! \text{Suc } i)$ **by** *auto*

lemma *smap-shifted-eq*:
 $\text{smap } (\lambda i. \text{RR}' (nds !! i, \text{stl } nds !! i)) (\text{fromN } (\text{Suc } 0)) =$
 $\text{smap } (\lambda i. \text{RR}' (\text{stl } nds !! i, \text{stl } (nds !! i)) \text{ nats}$
unfolding *stl-shift*
apply(*rule ssubst[of smap* $(\lambda i. \text{RR}' (nds !! \text{Suc } i, \text{stl } nds !! \text{Suc } i)) \text{ nats}$
 $\text{smap } (\lambda j (ps, ps', y). \text{RR } (\text{stl } nds !! (j - \text{Suc } 0), ps) (\text{stl } (nds !! (j - \text{Suc } 0), ps') y) (\text{fromN } (\text{Suc } 0)))$))
subgoal **using** *stream-smap-fromN[of smap* $(\lambda i. \text{RR}' (\text{stl } nds !! i, \text{stl } (nds !! i)) \text{ nats } \text{Suc } 0]$ **by** *auto*
apply(*rule stream.map-cong0*)
subgoal **for** *z* **using** *stl-Suc[of z nds]* *stl-Suc[of z stl nds]* **by** *force* .

lemma *Rst-correct*: $x = \text{Rst } nds \longleftrightarrow (\forall i. x !! i = \text{RR}' (nds !! i, nds !! \text{Suc } i))$
apply(*safe*)
subgoal **for** *i* **by**(*cases i, auto*)
subgoal
apply (*coinduction arbitrary: x nds, intro conjI*)
subgoal **by** (*erule allE[of - 0], auto*)

subgoal for $x\text{-hd } x\text{-tl } nd\text{-hd } nd\text{-tl } x \text{ nds}$
apply(rule $exI2[of \text{ - } x\text{-tl } stl \text{ nds}]$, safe)
subgoal using $smap\text{-shifted-eq}$ **by** $auto$
subgoal for i **by**(erule $allE[of \text{ - } Suc \ i]$, $auto$) . . .

lemma $Rst\text{-}r:\bigwedge k. Rst \text{ nds} !! k = RR' (nds !! k, nds !! Suc \ k)$ **using** $Rst\text{-correct}$ **by** $auto$

lemma $list\text{-swap}:k>0 \implies (p \#\# \text{ smap } (\lambda r. fst (the \ r)) \ r) !! k = fst (the (r !! (k-1)))$ **by** ($cases \ k$, $auto$)

lemma $Taut_R\text{-lang-in}:ipath \text{ nds} \implies Rst \text{ nds} \in NBA.\text{language } Taut_R \longleftrightarrow (\exists Ps. \text{descentIpath } nds \ Ps)$

proof(safe)
assume $ipath:ipath \text{ nds}$

have $k\text{-edge}:\bigwedge k. edge (nds !! k) (stl \ nds !! k)$ **using** $ipath \text{ unfolding } ipath\text{-def}$
 $alw\text{-holds2-iff-snth}$ **by** $auto$

show $Rst \text{ nds} \in NBA.\text{language } Taut_R \implies \exists Ps. \text{descentIpath } nds \ Ps$

proof –

assume $Rst \text{ nds} \in NBA.\text{language } Taut_R$
then obtain r **where** $run:NBA.run \ Taut_R (Rst \ nds \ ||| \ r) \ None$ **and** $accept:infs (\lambda ps. \exists p. ps = Some (p, Decr) \wedge (p, Decr) \in Q'_{sl}) \ r$ **apply**–**by**(erule $nba.\text{language-elim}$, $auto$)

hence $run':\bigwedge k. Rst \text{ nds} !! k = RR' (nds !! k, nds !! Suc \ k) \wedge edge (nds !! k) (nds !! Suc \ k) \wedge$
 $r !! k \in \Delta_{sl}' (Rst \ nds !! k) (if \ k = 0 \text{ then } None \text{ else } last (stake \ k \ r))$
using $k\text{-edge}$ **unfolding** $Paut_R\text{-run-def}$ **by** $auto$

have $q_0'\text{-Decr}:\bigwedge p \ k \ j. 0 < k \implies r !! k = Some (p, Decr) \implies k \leq j \implies r !! j \neq q_0$

subgoal for $p \ k \ j$ **proof**(induct $j - k$ arbitrary: j)

case 0

then show $?case$ **by** $auto$

next

case ($Suc \ xa$)

hence $Suc':\bigwedge j. 0 < k \implies xa = j - k \implies k \leq j \implies r !! j \neq q_0$ **by** $auto$

obtain n **where** $n:j = Suc \ n \ xa = n - k \ k \leq n$ **using** Suc **by**(cases j , $auto$)

show $?case$

using $Suc'[of \ n, OF \ Suc(3) \ n(2,3)]$ **unfolding** n

using $runE\text{-Suc}[OF \ run, of \ n] \ q_0\text{-notReachable}[of \ r !! n \ r !! Suc \ n]$ **by** $auto$

qed .

have $\text{accept-from-notNone}:\bigwedge n. \exists k \geq n. \exists p. r !! k = \text{Some } (p, \text{Decr}) \wedge \text{Some } (p, \text{Decr}) \in \Delta_{sl}' (Rst \text{ nds } !! k) (\text{last } (\text{stake } k \ r)) \wedge (\text{last } (\text{stake } k \ r)) \neq \text{None}$
 subgoal for n
 using $\text{accept unfolding infs-snth}$
 apply-apply($\text{erule allE[of - Suc } n]$, safe)
 subgoal for $k \ p$ using $\text{accept unfolding infs-snth}$
 apply-apply($\text{erule allE[of - Suc } k]$, safe)
 subgoal for $k' \ p'$ apply($\text{intro exI[of - } k'] \text{ conjI exI[of - } p']$)
 subgoal by auto
 subgoal by auto
 subgoal using $\text{run'[of } k']$ by auto
 subgoal using $q_0' \text{-Decr[of } k \ p \ k'-1]$ $\text{last-stake-i[of } k' \ r]$ by $\text{auto} \dots$

obtain Ps where $Ps:Ps = \text{fst } (\text{the } (\text{shd } r)) \#\# \text{ smap } (\lambda r. \text{fst } (\text{the } r)) \ r$ by auto

have $PsK:\bigwedge k. k > 0 \implies Ps !! k = \text{fst}(\text{the } (r !! (k - \text{Suc } 0)))$ subgoal for k
 unfolding Ps by($\text{drule list-swap[of } k \ (\text{fst } (\text{the } (\text{shd } r))) \ r]$, simp) .

show $\exists Ps. \text{descentIpath nds } Ps$
 apply($\text{rule exI[of - } Ps]$, $\text{unfold descentIpath-iff-snth2}$, intro conjI)
 subgoal using $\text{accept-from-notNone[of Suc } 0]$ apply safe
 subgoal for $k \ p' \ a \ b$ apply($\text{rule exI[of - } k]$, intro allI impI)
 subgoal for j proof($\text{induct } j - k \text{ arbitrary: } j$)
 case 0
 then have $k:k > 0$ by auto
 hence $\text{last}:(\text{last } (\text{stake } k \ r)) = (r !! (k - 1))$ using last-stake-i by auto
 also have $j:j > 0$ using 0 by auto
 show ?case using 0 unfolding last apply-apply($\text{erule } \Delta_{sl}'\text{-elim}$)
 using $PsK[OF \ j] \ PsK[of \ Suc \ j]$ by auto
 next
 case ($\text{Suc } x$)
 hence $jk:j \neq 0 \ k \neq 0 \ j > 0 \ k > 0 \ 0 < \text{Suc } j$ by auto
 then obtain n where $j:j = \text{Suc } n$ by ($\text{cases } j$, auto)
 then have $n\text{-gr}:n > 0 \ k \leq n$ using Suc by auto
 also have $k\text{-le}:k \leq j - 1$ using $n\text{-gr } j$ by auto
 have $n\text{th-stake-szip}:r !! j \in \Delta_{sl}' (Rst \text{ nds } !! j) (r !! (j-1))$ using $\text{run'[of } j]$,
 $\text{unfolded last-stake-i[OF } jk(3)]$ jk by auto
 obtain $a' \ b'$ where $r\text{-j-min}:(r !! (j-1)) = \text{Some}(a', b')$ using $q_0' \text{-Decr[of } k \ p'$
 $j-1, OF \ jk(4) \ \text{Suc}(4) \ k\text{-le}]$ by auto
 show ?case
 subgoal using $n\text{th-stake-szip}$ apply-apply($\text{erule } \Delta_{sl}'\text{-elim}$)
 subgoal for $p \ s \ p' \ s'$ unfolding $Rst\text{-r } PsK[OF \ jk(3)] \ PsK[OF \ jk(5)]$ by
 ($\text{cases } s'$, auto)
 subgoal using $r\text{-j-min}$ by auto
 subgoal using $r\text{-j-min}$ by $\text{auto} \dots$
 qed .

```

apply(rule allI)
subgoal for  $i$  using accept-from-notNone[of Suc(Suc  $i$ )] apply clarify
  subgoal for  $k$   $p$  apply(rule exI[of -  $k$ ], intro conjI)
    subgoal by auto
    using Suc-diff-1[of  $k$ ] last-stake-i[of  $k$   $r$ ] PsK[of  $k$ ] PsK[of  $k-1$ ]
    unfolding Rst-r Ps apply-apply(erule  $\Delta_{sl}'$ -elim)
    by (auto,metis fst-eqD option.sel) . .
qed

fix Ps

{assume iDPATH:descentIpath nds Ps
  have ev:ev (alw (holds2 ( $\lambda(nd, P) (nd', P'). RR (nd, P) (nd', P') Main \vee RR$ 
    ( $nd, P) (nd', P') Decr$ ))) (nds ||| Ps)and
    alw:alw (ev (holds2 ( $\lambda(nd, P) (nd', P'). RR (nd, P) (nd', P') Decr$ ))) (nds|||
    Ps)
  using iDPATH[unfolded descentIpath-def2] by auto

  also have wf:wfLabS nds Ps using descentIpath-wfLabS[OF iDPATH] by auto

  have hh: $\bigwedge x y z i. (stl x!!i, stl y!!i, stl z!!i) \in sset (x ||| y ||| z)$ 
    by (metis snth-sset snth-szip stream.set-sel(2) szip.sel(2))
  define  $F_{sl}$  where  $F_{sl} = (\lambda x' Ps' i. (if RR (x' !! i, Ps' !! i) (stl x' !! i, stl Ps'$ 
    !!  $i) Decr$  then  $Decr$  else  $Main$ ))
  define  $Ss$  where  $Ss = (\lambda x Ps. Main \#\# fToStream (F_{sl} x Ps))$ 
  note  $Ss-defs = Ss-def F_{sl}-def fToStream-def$ 

  have  $Ss-i:\bigwedge k m. m < Suc\ k \implies (Ss (sdrop (Suc\ m) nds) (sdrop (Suc\ m) Ps) !!$ 
    ( $Suc\ k - m$ )) =  $F_{sl} nds Ps (Suc\ k)$  using Suc-diff-le unfolding  $Ss-defs$  by auto

  obtain  $m$  where  $m:(\bigwedge n. n \geq m \implies (holds2 (\lambda(nd, P) (nd', P'). RR (nd, P)$ 
    ( $nd', P') Main \vee RR (nd, P) (nd', P') Decr$ )) (sdrop  $n$  (nds ||| Ps)))
    using ev unfolding ev-alw-iff-sdrop by auto

  have  $\bigwedge n. n \geq m \implies (Rst nds !! n) (Ps !! n, Ps !! Suc\ n, Main) \vee (Rst nds !! n)$ 
    ( $Ps !! n, Ps !! Suc\ n, Decr$ )
    subgoal for  $n$  by(drule  $m$ [of  $n$ ], simp) .

  obtain  $i$  where  $i:\forall j \geq i. Ps !! j \in PosOf (nds !! j)$  using wf unfolding wfLabS-iff-snth
by auto

  define  $m'$  where  $m' = i + m$ 
  hence  $m':(\bigwedge n. n \geq m' \longrightarrow Ps !! n \in PosOf (nds !! n) \wedge ((Rst nds !! n) (Ps !! n,$ 
     $Ps !! Suc\ n, Main) \vee (Rst nds !! n) (Ps !! n, Ps !! Suc\ n, Decr)))$ 
    using  $i$   $m$  by auto

  have alw-dropm:alw (ev (holds2 ( $\lambda(nd, P) (nd', P'). RR (nd, P) (nd', P')$ 
    Decr))) (sdrop (Suc  $m'$ ) (nds ||| Ps))
    using alw unfolding alw-ev-sdrop by auto
```

define r **where** $r = \text{replicate } m' \ q_0 \ @- \ \text{smap } \text{Some } ((\text{sdrop } (\text{Suc } m') \ Ps) \ ||| \ (Ss \ (\text{sdrop } (\text{Suc } m') \ nds) \ (\text{sdrop } (\text{Suc } m') \ Ps))))$

have $x\text{-node}' : \bigwedge i. nds \ !! \ i \in \text{Node} \ \text{using } \text{ipath} \ \text{unfolding } \text{ipath-def } \text{alw-holdsS-iff-snth} \ \text{by } \text{auto}$

have $r\text{-eq-}q_0' : \bigwedge n. \text{Suc } n \leq m' \longleftrightarrow r!!n = q_0$
apply *standard* **unfolding** $r\text{-def}$
subgoal **using** *replicate-within-i* **by** *auto*
subgoal **for** n **apply**(*rule ccontr, unfold not-le replicate-beyond-i*) **using** $x\text{-node}'[of \ n]$ **by** *auto* .

have $r\text{-neg-}q_0' : \bigwedge n. \text{Suc } n > m' \longleftrightarrow r!!n = \text{Some } (Ps \ !! \ \text{Suc } n, Ss \ (\text{sdrop } (\text{Suc } m') \ nds) \ (\text{sdrop } (\text{Suc } m') \ Ps) \ !! \ (n - m'))$
apply *standard*
subgoal **unfolding** $r\text{-def}$ **using** *replicate-beyond-i* **by** *auto*
subgoal **for** n **using** $r\text{-eq-}q_0'[of \ n]$ **by** *auto* .

have $Rst\text{-all} : \bigwedge k. (Rst \ nds \ !! \ k) \in \Sigma \ \text{using } k\text{-edge} \ \text{unfolding } Rst\text{-r} \ \text{by } \text{auto}$

have $Psk\text{-node} : \bigwedge k. Ps \ !! \ k \in \text{PosOf } (nds \ !! \ k) \implies \exists x \in \text{Node}. Ps \ !! \ k \in \text{PosOf } x$
using $x\text{-node}'$ **by** *auto*
have $m'\text{-in}Q'_{sl} : \bigwedge k \ x. k \geq m' \implies (Ps \ !! \ k, x) \in Q'_{sl}$
unfolding $Q'_{sl}\text{-def}$ **using** $Psk\text{-node}$ m' **by** *auto*

have $m\text{-eq} : \bigwedge k \ m. \text{Suc } k \leq m \implies m < \text{Suc } (\text{Suc } k) \implies m = \text{Suc } k \ \text{by } \text{auto}$
have $Psn\text{-gr} : \bigwedge n. \text{Suc } n > m' \implies Ps \ !! \ (\text{Suc } n) = \text{fst } (\text{the } (r \ !! \ n)) \ \text{using } r\text{-neg-}q_0'$
by *auto*
have $Ssn\text{-gr} : \bigwedge n. \text{Suc } n > m' \implies \text{snd } (\text{the } (r \ !! \ n)) = Ss \ (\text{sdrop } (\text{Suc } m') \ nds) \ (\text{sdrop } (\text{Suc } m') \ Ps) \ !! \ (n - m') \ \text{using } r\text{-neg-}q_0' \ \text{by } \text{auto}$

show $Rst \ nds \in \text{NBA.language } Taut_R$
proof(*rule nba.language[of q_0 Taut_R - r], unfold Taut_R-accept Taut_R-initp, safe*)
show $\text{NBA.run } Taut_R \ (Rst \ nds \ ||| \ r) \ q_0$
proof(*unfold run-def', intro allI conjI*)
fix k
show $Rst \ nds \ !! \ k \in \Sigma \ \text{using } Rst\text{-all} \ \text{by } \text{auto}$

show $r \ !! \ k \in \Delta_{sl}' (Rst \ nds \ !! \ k) \ (\text{last } (\text{None } \# \ \text{map } \text{snd } (\text{stake } k \ (Rst \ nds \ ||| \ r))))$
proof(*induct k*)
case 0
have $l : (\text{last } (\text{None } \# \ \text{map } \text{snd } (\text{stake } 0 \ (Rst \ nds \ ||| \ r)))) = \text{None} \ \text{by } \text{auto}$

```

show ?case unfolding l
  apply(cases m')
    subgoal apply(rule ssubst[of r !!0 Some (Ps !! (Suc 0), Ss (sdrop (Suc
m') nds) (sdrop (Suc m') Ps) !! 0)])
      subgoal using r-neq-q0'[of 0] by auto
      subgoal apply(rule  $\Delta_{sl}'$ -intro-q0)
        subgoal using Rst-all[of 0] by auto
        subgoal using m'-inQ'sl[of Suc 0] by auto . .
      subgoal apply(rule ssubst[of r !!0 None])
        subgoal using r-eq-q0'[of 0] by auto
        apply(rule  $\Delta_{sl}'$ -q0-included) using Rst-all[of 0] by auto .
    next
    case (Suc k)
    then show ?case unfolding last-stake-szip apply—apply(erule  $\Delta_{sl}'$ -elim)

      subgoal for p s p' s' apply(rule  $\Delta_{sl}'$ -intro)
        subgoal using Rst-all[of Suc k] by auto
        subgoal apply(intro exI2[of - Ps !! Suc (Suc k) Ss (sdrop (Suc m') nds)
(sdrop (Suc m') Ps) !! (Suc k - m')] conjI)
          subgoal using r-eq-q0'[of k] r-neq-q0'[of Suc k] by auto
          subgoal using Psn-gr[of k] r-eq-q0'[of k] m'-inQ'sl[of Suc k] by auto
          subgoal using m'[of Suc k] r-eq-q0'[of k] Ss-i Psn-gr unfolding
Fsl-def by auto .
          subgoal by auto .

      subgoal for p s apply(rule  $\Delta_{sl}'$ -intro)
        subgoal using Rst-all[of Suc k] by auto
        subgoal apply(rule exI[of - Ps !! Suc (Suc k)], intro exI[of - Ss (sdrop
(Suc m') nds) (sdrop (Suc m') Ps) !! (Suc k - m')] conjI)
          subgoal using r-eq-q0'[of k] r-neq-q0'[of Suc k] by auto
          subgoal by auto
          subgoal using m'[of Suc k] r-eq-q0'[of k] Ss-i Psn-gr unfolding
Fsl-def by auto .
          subgoal by auto .

      subgoal apply(rule  $\Delta_{sl}'$ -intro)
        subgoal using Rst-all[of Suc k] by auto
        subgoal by auto
        subgoal using le-less-linear[of Suc (Suc k) m'] apply—apply(erule
disjE)
          subgoal using r-eq-q0'[of Suc k] by auto
          subgoal using r-neq-q0'[of Suc k] m'-inQ'sl[of Suc (Suc k)] by auto .
      . .
    qed
  qed

show infs (λps. ∃p. ps = Some (p, Decr) ∧ (p, Decr) ∈ Q'sl) r
  unfolding r-def alw-ev-shift infs-snth apply(clarify)
  subgoal for n using alw-droptm unfolding alw-ev-holds2-iff-snth

```

```

apply-apply(erule allE[of - n + i], clarify)
subgoal for j x y xa ya apply(intro exI[of - Suc j] conjI)
  subgoal by auto
  subgoal apply(intro exI[of - ya] conjI)
    subgoal unfolding Ss-defs by auto
    subgoal using i RR-PosOfD' unfolding Q'sl-def by fastforce . . .
qed}
qed

```

lemma *alpha-subseq-PTaut_R*: $nba.alphabet\ Paut_R \subseteq nba.alphabet\ Taut_R$ **by** simp

lemma *Paut_R-subseq-rule*: $(\bigwedge r. \forall x \in Node. last\ (map\ snd\ r) \neq Some\ x \implies r \neq [] \implies NBA.path\ Paut_R\ r\ None \implies last\ (map\ snd\ r) = None) \implies (\bigcup p \in \{p. p \in nba.initial\ Paut_R\}. \{last\ (p \# map\ snd\ r) \mid r. NBA.path\ Paut_R\ r\ p\}) \subseteq \{r. r = None \vee (\exists x \in Node. r = Some\ x)\}$ **by** auto

lemma *Paut_R-node-subseq*: $NBA.nodes\ Paut_R \subseteq \{r. r = None \vee (\exists x \in Node. r = Some\ x)\}$

unfolding nba.nodes-alt-def nba.reachable-alt-def nba.target-alt-def nba.states-alt-def

apply(rule *Paut_R-subseq-rule*)

subgoal premises p **for** r **using** p(3,1-2) **apply**(induct rule: nba.path.induct)

subgoal by auto

subgoal for a p **by** (cases p, auto simp: RR-PosOfD' split: if-splits) . .

lemma *finite-Nodes-Paut_R*: finite (NBA.nodes Paut_R)

apply(rule rev-finite-subset[OF - Paut_R-node-subseq]) **using** finite-Node-opt **by** auto

lemma *Taut_R-subseq-rule*: $(\bigwedge r. \forall a. (\forall x \in Node. a \notin PosOf\ x) \vee (\forall b. last\ (map\ snd\ r) \neq Some\ (a, b)) \implies r \neq [] \implies NBA.path\ Taut_R\ r\ None \implies last\ (map\ snd\ r) = None) \implies (\bigcup p \in \{p. p \in nba.initial\ Taut_R\}. \{last\ (p \# map\ snd\ r) \mid r. NBA.path\ Taut_R\ r\ p\}) \subseteq \{r. r = None \vee (\exists x \in \{(p, s) \mid p\ s. p \in (\bigcup v \in Node. PosOf\ v)\}. r = Some\ x)\}$

by auto

theorem *SLA-Criterion*: $InfiniteDescent \longleftrightarrow NBA.language\ Paut_R \subseteq NBA.language\ Taut_R$

apply standard

subgoal unfolding subset-iff Paut_R-lang-in InfiniteDescent-def **apply** clarify

```

    subgoal for Ri nds apply(erule allE[of - nds], safe)
    by(drule TautR-lang-in, unfold Rst-correct[of Ri nds, symmetric], auto) .
  subgoal unfolding InfiniteDescent-def apply(clarify)
  subgoal for nds
  apply(unfold subset-iff PautR-lang-in)
  by(erule allE[of - Rst nds], unfold TautR-lang-in, auto) . .

end
end

```

3.2 Relation-based Criterion

Infinite Descent also admits an equivalent *relation-based* characterization. This was first described in the context of size-change termination [3] and later generalized and refined [2]. The key idea is to summarize each finite path using a sloped relation, resulting in a finite abstraction that can be computed via a fixed-point computation. Infinite Descent can then be decided by checking a simple condition on the sloped relations that summarize loops.

```

theory Relation-Based-Criterion
  imports VLA-Criterion
begin

lemma pigeonhole-infinite-seq:
  fixes f :: nat  $\Rightarrow$  'a'
  assumes finite (range f)
  shows  $\exists i j. i < j \wedge f i = f j$ 
proof (rule ccontr)
  assume  $\neg (\exists i j. i < j \wedge f i = f j)$ 
  hence inj f unfolding inj-on-def by (metis linorder-cases)
  hence infinite (range f) using finite-imageD infinite-UNIV-nat by blast
  with assms show HOL.False by contradiction
qed

context Sloped-Graph
begin

```

```

definition MaxSl :: slope set  $\Rightarrow$  slope where
MaxSl sll  $\equiv$  if Decr  $\in$  sll then Decr else Main

```

```

lemma MaxSl-singl[simp]: MaxSl {sl} = sl
unfolding MaxSl-def by (cases sl, auto)

```

```

definition leqSl :: ('pos  $\Rightarrow$  'pos  $\Rightarrow$  slope  $\Rightarrow$  bool)  $\Rightarrow$ 

```

$(\text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where
 $\text{leqSl } P1 \ P2 \equiv \forall h \ h' \ sl1. \ P1 \ h \ h' \ sl1 \longrightarrow (\exists \ sl2. \ sl1 \leq \ sl2 \wedge P2 \ h \ h' \ sl2)$

definition $\text{lessSl} :: (\text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool}) \Rightarrow$
 $(\text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
where $\text{lessSl } P1 \ P2 \equiv \text{leqSl } P1 \ P2 \wedge P1 \neq P2$

lemma $\text{leqSl-refl}: \text{leqSl } P \ P$
unfolding leqSl-def **by** *auto*

lemma $\text{leqSl-trans}: \text{leqSl } P1 \ P2 \Longrightarrow \text{leqSl } P2 \ P3 \Longrightarrow \text{leqSl } P1 \ P3$
unfolding leqSl-def
by (*meson dual-order.trans*)

lemma leqSl-antisym-aux :
assumes $P12: \{P1, P2\} \subseteq \text{SlopedRels}$ **and** $12: \text{leqSl } P1 \ P2$ **and** $21: \text{leqSl } P2 \ P1$
and $0: P1 \ h \ h' \ sl$
shows $P2 \ h \ h' \ sl$
proof–
obtain sll **where** $1: sl \leq sll$ **and** $11: P2 \ h \ h' \ sll$
using $0 \ 12$ **unfolding** leqSl-def **by** *blast*
then obtain $slll$ **where** $2: sll \leq slll$ **and** $22: P1 \ h \ h' \ slll$
using 21 **unfolding** leqSl-def **by** *blast*
have $sl = slll$ **using** $P12 \ 0 \ 22$ **unfolding** SlopedRels-def **by** *auto*
hence $sl = sll$ **using** $1 \ 2$ **by** *auto*
thus $?thesis$ **using** 11 **by** *auto*
qed

lemma leqSl-antisym :
assumes $P12: \{P1, P2\} \subseteq \text{SlopedRels}$ **and** $12: \text{leqSl } P1 \ P2$ **and** $21: \text{leqSl } P2 \ P1$
shows $P1 = P2$
using leqSl-antisym-aux *assms*
by *fastforce*

lemma $\text{lessSl-antisym}: \{P1, P2\} \subseteq \text{SlopedRels} \Longrightarrow \neg (\text{lessSl } P1 \ P2 \wedge \text{leqSl } P2 \ P1)$
using leqSl-antisym lessSl-def **by** *auto*

lemma $\text{lessSl-trans}: \{P1, P2, P3\} \subseteq \text{SlopedRels} \Longrightarrow \text{lessSl } P1 \ P2 \Longrightarrow \text{lessSl } P2 \ P3$
 $\Longrightarrow \text{lessSl } P1 \ P3$
unfolding leqSl-def lessSl-def **by** (*metis dual-order.trans insert-subset leqSl-antisym leqSl-def*)

inductive $\text{transSl} :: (\text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool}) \Rightarrow (\text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool})$
for $K :: \text{'pos} \Rightarrow \text{'pos} \Rightarrow \text{slope} \Rightarrow \text{bool}$ **where**
Base: $K \ P \ P' \ sl \Longrightarrow \text{transSl } K \ P \ P' \ sl$

|Step: $\text{transSl } K P P' \text{ sl1} \implies K P' P'' \text{ sl2} \implies \text{transSl } K P P'' (\text{MaxSl } \{\text{sl1}, \text{sl2}\})$

lemma *transSl-mono*: **assumes** $\text{leqSl } P Q$
shows $\text{leqSl } (\text{transSl } P) (\text{transSl } Q)$
unfolding *le-fun-def leqSl-def* **apply** *clarsimp*
subgoal for $h1 h2 \text{ sl}$ **apply** (*induct rule: transSl.induct*)
subgoal using *assms leqSl-def transSl.Base* **by** *metis*
subgoal
by (*smt (verit, best) MaxSl-singl assms insert-absorb2 leqSl-def less-eq-slope.elims(3) slope.exhaust transSl.Step*) . .

definition *initFrag* ::
 $(\text{'pos} \implies \text{'pos} \implies \text{slope} \implies \text{bool}) \text{ set} \implies (\text{'pos} \implies \text{'pos} \implies \text{slope} \implies \text{bool}) \text{ set} \implies \text{bool}$
where
 $\text{initFrag } LL' LL \equiv \forall R \in LL. \exists R' \in LL'. \text{leqSl } R' R$

lemma *initFrag-Un*: $\text{initFrag } L (LL1 \cup LL2) \longleftrightarrow \text{initFrag } L LL1 \wedge \text{initFrag } L LL2$
unfolding *initFrag-def* **by** *auto*

lemma *initFrag-trans*: $\text{initFrag } L1 L2 \implies \text{initFrag } L2 L3 \implies \text{initFrag } L1 L3$
unfolding *initFrag-def* **by** (*meson leqSl-trans*)

definition *Dt* **where**
 $Dt LL \equiv \{R \in LL. \neg (\exists R' \in LL. \text{lessSl } R' R)\}$

lemma *Dt-incl*: $Dt LL \subseteq LL$ **unfolding** *Dt-def* **by** *auto*

lemma *Dt-aux*: $\bigwedge LL LL'. LL \subseteq LL' \implies Dt LL \subseteq LL'$ **unfolding** *Dt-def* **by** *auto*

lemma *initFrag-Dt*:
assumes $LL: LL \subseteq \text{SlopedRels}$ **and** $f\text{-}LL: \text{finite } LL$
shows $\text{initFrag } (Dt LL) LL$
unfolding *initFrag-def* **proof** *safe*
fix R **assume** $R[\text{simp}]: R \in LL$
define f **where** $f = \text{rec-nat } R (\lambda n R. \text{if } R \in Dt LL \text{ then } R \text{ else } (\text{SOME } R'. R' \in LL \wedge \text{lessSl } R' R))$
have $f: f 0 = R$
 $\bigwedge n. f (\text{Suc } n) =$
 $(\text{if } f n \in Dt LL \text{ then } f n \text{ else } (\text{SOME } R'. R' \in LL \wedge \text{lessSl } R' (f n)))$
unfolding *f-def* **by** *auto*

have $ff: \bigwedge n. f n \in LL \wedge$
 $((f n \in Dt LL \wedge f (\text{Suc } n) = f n) \vee$
 $(f n \notin Dt LL \wedge f (\text{Suc } n) \in LL \wedge \text{lessSl } (f (\text{Suc } n)) (f n)))$
subgoal for n **apply** (*induct n*)
subgoal by (*simp add: f Dt-def*) (*metis (no-types, lifting) someI*)
subgoal by (*simp add: f Dt-def*) (*metis (no-types, lifting) someI*) . .

hence $f\text{-UNIV-LL}$: $f \text{ ' } UNIV \subseteq LL$ by *auto*
 {assume $\forall n. f\ n \notin Dt\ LL$
 hence $\forall n. lessSl\ (f\ (Suc\ n))\ (f\ n)$ using *ff* by *auto*
 hence $\forall n\ i. lessSl\ (f\ (Suc\ (n+i)))\ (f\ n)$ apply *safe*
 subgoal for $n\ i$ apply(*induct i, simp-all*)
 by (*metis (no-types, opaque-lifting) assms(1)*
empty-subsetI ff insert-subset leqSl-trans lessSl-antisym lessSl-def order-trans)
 .
 hence *inj f* unfolding *inj-def lessSl-def*
 by (*metis le-neq-implies-less less-natE nat-le-linear*)
 hence *HOL.False* using *f-LL f-UNIV-LL*
 using *infinite-iff-countable-subset* by *blast*
 }
 then obtain n where $n: f\ n \in Dt\ LL$ by *auto*

 hence $\forall n. leqSl\ (f\ (Suc\ n))\ (f\ n)$ using *ff* unfolding *lessSl-def*
 by (*metis leqSl-refl*)
 hence $\forall n\ i. leqSl\ (f\ (Suc\ (n+i)))\ (f\ n)$ apply *safe*
 subgoal for $n\ i$ apply(*induct i, simp-all*)
 by (*metis (no-types, opaque-lifting) leqSl-trans*) .
 hence $1: leqSl\ (f\ (n))\ R$
 by (*metis add-0 f(1) f(2) n*)

 show $\exists R' \in Dt\ LL. leqSl\ R'\ R$
 apply(*rule bexI[of - f n]*)
 using *ff n 1* unfolding *lessSl-def* by *auto*
 qed

definition *ccompSl* :: $('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool) \Rightarrow ('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool) \Rightarrow$
 $'pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool$
where
ccompSl $K1\ K2\ P\ P'\ sl \equiv \exists P''\ sl1\ sl2. sl = MaxSl\ \{sl1, sl2\} \wedge K1\ P\ P''\ sl1 \wedge$
 $K2\ P''\ P'\ sl2$

lemma *finite-slope-set*: *finite* ($S::slope\ set$)
by (*metis (full-types) finite.simps insert-iff rev-finite-subset slope.exhaust subsetI*)

lemma *ccompSl-mono*: $leqSl\ P1\ Q1 \implies leqSl\ P2\ Q2 \implies leqSl\ (ccompSl\ P1\ P2)$
 $(ccompSl\ Q1\ Q2)$
unfolding *ccompSl-def le-fun-def MaxSl-def leqSl-def*
by (*smt (z3) insert-iff less-eq-slope.simps(3) slope.exhaust*)

lemma *ccompSl-SlopeRels*:

$\{P, P'\} \subseteq \text{SlopedRels} \implies \text{ccompSl } P \ P' \in \text{SlopedRels}$

unfolding *SlopedRels-def ccompSl-def MaxSl-def* **oops**

fun *Ccl-iter* :: $\text{nat} \Rightarrow 'node \Rightarrow 'node \Rightarrow ('pos \Rightarrow 'pos \Rightarrow \text{slope} \Rightarrow \text{bool}) \text{ set}$ **where**
 $\text{Ccl-iter } 0 \ nd \ nd' = (\text{if } \{nd, nd'\} \subseteq \text{Node} \wedge \text{edge } nd \ nd' \text{ then } \{\lambda P \ P' \text{ sl. } \text{RR } (nd, P) (nd', P') \text{ sl}\} \text{ else } \{\})$
 $\mid \text{Ccl-iter } (\text{Suc } i) \ nd \ nd' =$
 $\text{Ccl-iter } i \ nd \ nd' \cup$
 $\{\text{ccompSl } K1 \ K2 \mid K1 \ K2 \ nd''. \ nd'' \in \text{Node} \wedge K1 \in \text{Ccl-iter } i \ nd \ nd'' \wedge K2 \in \text{Ccl-iter } i \ nd'' \ nd'\}$

lemma *Ccl-iter-nodes*: $nd \notin \text{Node} \vee nd' \notin \text{Node} \implies \text{Ccl-iter } i \ nd \ nd' = \{\}$
apply(*induct i arbitrary: nd nd'*) **by** *auto*

lemma *Ccl-iter-PosOf*:

$K \in \text{Ccl-iter } i \ nd \ nd' \implies K \ P \ P' \text{ sl} \implies P \in \text{PosOf } nd \wedge P' \in \text{PosOf } nd'$

apply(*cases* $\{nd, nd'\} \subseteq \text{Node}$)

subgoal **apply**(*induct i arbitrary: nd nd' K sl P P'*)

using *RR-PosOf* **by** (*auto split: if-splits simp: ccompSl-def*)

subgoal **using** *Ccl-iter-nodes* **by** *auto* .

lemma *Ccl-iter-Suc-mono*: $\text{Ccl-iter } i \ nd \ nd' \subseteq \text{Ccl-iter } (\text{Suc } i) \ nd \ nd'$
by *auto*

lemma *Ccl-iter-mono*: $i \leq j \implies \text{Ccl-iter } i \ nd \ nd' \subseteq \text{Ccl-iter } j \ nd \ nd'$
apply(*induct j*)
subgoal **by** *auto*
subgoal **using** *Ccl-iter-Suc-mono le-SucE* **by** *blast* .

lemma *Ccl-iter-Suc-stable*:

assumes $\text{Ccl-iter } (\text{Suc } i) = \text{Ccl-iter } i$

shows $\text{Ccl-iter } (\text{Suc } (\text{Suc } i)) = \text{Ccl-iter } i$

using *assms unfolding fun-eq-iff Ccl-iter.simps(2)* **by** *auto*

lemma *Ccl-iter-stable*:

assumes $1: \text{Ccl-iter } (\text{Suc } i) = \text{Ccl-iter } i$ **and** $2: j \geq i$

shows $\text{Ccl-iter } j = \text{Ccl-iter } i$

using *assms apply*(*induct j-i arbitrary: j i*)

subgoal **by** *auto*

subgoal **for** $x \ j$

by (*metis Suc-diff-le Suc-leD diff-diff-cancel diff-le-self Ccl-iter-Suc-stable*) .

lemma *Ccl-iter-su-PosOf*:

$\text{Ccl-iter } i \ nd \ nd' \subseteq$

$\{R . \forall h \ h' \text{ sl. } (h, h') \notin \text{PosOf } nd \times \text{PosOf } nd' \longrightarrow \neg R \ h \ h' \text{ sl}\}$

apply(*induct i arbitrary: nd nd'*)

using *RR-PosOf Ccl-iter-PosOf ccompSl-def* **by** *auto*

lemma *finite-PosOf-prod*:
assumes $nd \in \text{Node}$ $nd' \in \text{Node}$
shows $\text{finite } \{R . \forall h h' sl. (h::'pos, h'::'pos) \notin \text{PosOf } nd \times \text{PosOf } nd' \rightarrow \neg R h h' (sl::\text{slope})\}$
(is finite ?A)
proof –
define f **where** $f \equiv \lambda R. \{(h::'pos, h'::'pos, sl::\text{slope}) \mid h h' sl. R h h' sl\}$
have $2: \text{inj-on } f \text{ ?A}$
unfolding *inj-on-def* *f-def* **by** *fastforce*
have $3: f \text{ ' ?A} \subseteq \text{Pow } (\text{PosOf } nd \times \text{PosOf } nd' \times (\text{UNIV}::\text{slope set}))$
(is - $\subseteq$?B)
unfolding *f-def* **by** *auto*
have *finite ?B* **unfolding** *finite-Pow-iff* **apply**(*intro finite-cartesian-product*)
using *PosOf-finite* *assms* *finite-slope-set* **by** *auto*
thus *?thesis* **using** $2\ 3$ **by** (*meson inj-on-finite*)
qed

lemma *finite-Ccl-iter*:
shows $\text{finite } (\text{Ccl-iter } i \text{ nd nd'})$
by (*metis* (*no-types*, *lifting*) *Ccl-iter-nodes* *Ccl-iter-su-PosOf* *finite.simps* *finite-PosOf-prod* *rev-finite-subset*)

lemma *Ccl-iter-Suc-stops*:
 $\exists i. \text{Ccl-iter } (\text{Suc } i) = \text{Ccl-iter } i$
proof –
{assume $\forall i. \exists nd nd'. \text{Ccl-iter } (\text{Suc } i) \text{ nd nd}' \neq \text{Ccl-iter } i \text{ nd nd}'$
hence $\forall i. \exists nd nd'. (nd, nd') \in \text{Node} \times \text{Node} \wedge \text{Ccl-iter } (\text{Suc } i) \text{ nd nd}' \neq \text{Ccl-iter } i \text{ nd nd}'$
by (*metis* *Ccl-iter-nodes* *SigmaI*)
hence $\forall i. \exists ndd. ndd \in \text{Node} \times \text{Node} \wedge \text{Ccl-iter } (\text{Suc } i) (\text{fst } ndd) (\text{snd } ndd) \neq \text{Ccl-iter } i (\text{fst } ndd) (\text{snd } ndd)$
by (*metis* *fst-eqD* *snd-eqD*)
then obtain ndi **where** $0:$
 $\forall i. ndi \ i \in \text{Node} \times \text{Node} \wedge$
 $\text{Ccl-iter } (\text{Suc } i) (\text{fst } (ndi \ (i::\text{nat}))) (\text{snd } (ndi \ i)) \neq \text{Ccl-iter } i (\text{fst } (ndi \ i))$
 $(\text{snd } (ndi \ i))$
by *metis*
hence $r: \text{range } ndi \subseteq \text{Node} \times \text{Node}$ **unfolding** *image-def* **by** *blast*
hence $\text{finite } (\text{range } ndi)$ **by** (*simp* *add: Node-finite* *finite-subset*)
moreover have $\{\{i. ndi \ i = ndd\} \mid ndd . ndd \in \text{range } ndi\} =$
 $(\lambda ndd. \{i. ndi \ i = ndd\}) \text{ ' } (\text{range } ndi)$ **(is ?A = -)** **by** *auto*
ultimately have $1: \text{finite } ?A$ **by** *simp*
have $\text{UNIV} = \bigcup \text{ ?A}$
by *auto* (*metis* (*mono-tags*) *mem-Collect-eq* *range-eqI* *surj-pair*)
then obtain ndd **where** $ndd: ndd \in \text{range } ndi$ **and** $\text{inf: infinite } \{i. ndi \ i = ndd\}$
using *finite-Union[OF 1]* **by** *auto*

hence $ndd\text{-}in$: $ndd \in Node \times Node$ **using** r **by** *auto*
define $nd\ nd'$ **where** $nd = fst\ ndd$ **and** $nd' = snd\ ndd$
define I **where** $I = \{i. ndi\ i = ndd\}$
have $nd\text{-}nd'$: $nd \in Node \wedge nd' \in Node$ **and** I : *infinite* I **using** *inf* **unfolding**
I-def
using $ndd\text{-}in$ **unfolding** $nd\text{-}def\ nd'\text{-}def$ **by** *auto*

hence $\forall i \in I. \exists R. R \in Ccl\text{-}iter\ (Suc\ i)\ nd\ nd' \wedge R \notin Ccl\text{-}iter\ i\ nd\ nd'$
using $Ccl\text{-}iter\text{-}mono\ 0\ I\text{-}def\ nd'\text{-}def\ nd\text{-}def$ **by** *fastforce*
then obtain Ri **where** 0 : $\forall i \in I. Ri\ i \in Ccl\text{-}iter\ (Suc\ i)\ nd\ nd' \wedge Ri\ i \notin Ccl\text{-}iter\ i\ nd\ nd'$
by *metis*
hence $\forall i\ j. \{i, j\} \subseteq I \wedge i < j \longrightarrow Ri\ i \neq Ri\ j$
by (*metis Ccl-iter-mono in-mono insert-subset less-eq-Suc-le*)
hence *inj-on* $Ri\ I$
by (*metis (mono-tags, opaque-lifting) empty-subsetI insert-subset linorder-inj-onI nle-le*)
hence *infinite* (*image* $Ri\ I$) **using** I *finite-imageD* **by** *blast*
moreover have *image* $Ri\ I \subseteq$
 $\{R. \forall h\ h'. sl. (h::'pos, h'::'pos) \notin PosOf\ nd \times PosOf\ nd' \longrightarrow \neg R\ h\ h' (sl::slope)\}$ (*is - $\subseteq$?A*)
using $0\ Ccl\text{-}iter\text{-}su\text{-}PosOf$ **by** *blast*
moreover have *finite* ?A
using $nd\text{-}nd'\text{ finite-PosOf-prod}$ **by** *presburger*
ultimately have *HOL.False*
by (*meson infinite-super*)
}
thus ?thesis **by** *blast*
qed

lemma $Ccl\text{-}iter\text{-}stops$: $\exists i. \forall j \geq i. Ccl\text{-}iter\ j = Ccl\text{-}iter\ i$
using $Ccl\text{-}iter\text{-}Suc\text{-}stops$ **apply** *safe*
subgoal for i **apply** (*rule exI[of - i]*)
using $Ccl\text{-}iter\text{-}stable$ **by** *blast* .

definition $Ccl :: 'node \Rightarrow 'node \Rightarrow ('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool)$ **set where**
 $Ccl\ nd\ nd' \equiv \bigcup i. Ccl\text{-}iter\ i\ nd\ nd'$

lemma $Ccl\text{-}nodes$: $nd \notin Node \vee nd' \notin Node \implies Ccl\ nd\ nd' = \{\}$
by (*simp add: Ccl-def Ccl-iter-nodes*)

lemma $Ccl\text{-}eq\text{-}some\text{-}Ccl\text{-}iter$: $\exists i. Ccl = Ccl\text{-}iter\ i$
proof –
obtain i **where** 0 : $\forall j \geq i. Ccl\text{-}iter\ j = Ccl\text{-}iter\ i$
using $Ccl\text{-}iter\text{-}stops$ **by** *auto*
hence 00 : $Ccl\text{-}iter\ (Suc\ i) = Ccl\text{-}iter\ i$
using *le-Suc-eq* **by** *blast*
have $\bigwedge j\ nd\ nd'. Ccl\text{-}iter\ j\ nd\ nd' \subseteq Ccl\text{-}iter\ i\ nd\ nd'$

subgoal for j nd nd' **apply**(*induct j arbitrary: nd nd'*)
subgoal using *Ccl-iter-mono* **by** *blast*
subgoal by *simp* (*smt (verit, ccfv-threshold) 00 Ccl-iter.simps(2) UnCI in-mono*
mem-Collect-eq subsetI) . .
thus *?thesis* **unfolding** *Ccl-def* **apply**(*intro exI[of - i]*)
by *fastforce*
qed

lemma *Ccl-RR[simp,intro!]*:
 $\{nd, nd'\} \subseteq \text{Node} \implies \text{edge } nd \ nd' \implies (\lambda P \ P' \ sl. \text{RR } (nd, P) \ (nd', P') \ sl) \in \text{Ccl } nd \ nd'$
unfolding *Ccl-def*
by *simp* (*metis empty-subsetI insert-subset Ccl-iter.simps(1) order-refl*)

lemma *Ccl-ccompSl[intro]*:
 $nd'' \in \text{Node} \implies K1 \in \text{Ccl } nd \ nd'' \implies K2 \in \text{Ccl } nd'' \ nd'$
 $\implies \text{ccompSl } K1 \ K2 \in \text{Ccl } nd \ nd'$
unfolding *Ccl-def* **apply** *clarsimp*
subgoal for i j **apply**(*rule exI[of - Suc (max i j)]*)
by *simp* (*metis in-mono Ccl-iter-mono max.cobounded1 max.cobounded2*) .

definition *TransitiveLoopingCcl* $\equiv \forall nd \in \text{Node}. \forall K \in \text{Ccl } nd \ nd. (\exists P. \text{transSl } K \ P \ P \ \text{Decr})$

definition *compSl* $K1 \ K2 \ P \ P' \ sl \equiv$
 $(\exists P'' \ sl1 \ sl2. sl = \text{MaxSl } \{sl1, sl2\} \wedge K1 \ P \ P'' \ sl1 \wedge K2 \ P'' \ P' \ sl2) \wedge$
 $sl = \text{Max } \{sl. \exists P'' \ sl1 \ sl2. sl = \text{MaxSl } \{sl1, sl2\} \wedge K1 \ P \ P'' \ sl1 \wedge K2 \ P'' \ P' \ sl2\}$

lemma *compSl-SlopeRels*:
 $\{P, P'\} \subseteq \text{SlopedRels} \implies \text{compSl } P \ P' \in \text{SlopedRels}$
unfolding *SlopedRels-def compSl-def MaxSl-def* **by** *auto*

lemma *compSl-leqSl-ccompSl*: $\text{leqSl } (\text{compSl } K1 \ K2) \ (\text{ccompSl } K1 \ K2)$
unfolding *leqSl-def compSl-def ccompSl-def MaxSl-def* **by** *auto*

lemma *ccompSl-leqSl-compSl*: $\text{leqSl } (\text{ccompSl } K1 \ K2) \ (\text{compSl } K1 \ K2)$
unfolding *leqSl-def* **proof** *safe*
fix $h \ h' \ sl$ **assume** *ccompSl* $K1 \ K2 \ h \ h' \ sl$
then obtain $P'' \ sl1 \ sl2$ **where** $0: sl = \text{MaxSl } \{sl1, sl2\}$
 $K1 \ h \ P'' \ sl1 \wedge K2 \ P'' \ h' \ sl2$ **unfolding** *ccompSl-def* **by** *auto*
let $?A = \{sl. \exists P'' \ sl1 \ sl2. sl = \text{MaxSl } \{sl1, sl2\} \wedge K1 \ h \ P'' \ sl1 \wedge K2 \ P'' \ h' \ sl2\}$

```

define sll where sll: sll = Max ?A
have 1: sl ≤ sll unfolding sll apply(rule Max-ge)
using 0 finite-slope-set by auto
have 2: sll ∈ ?A unfolding sll apply(rule Max-in) using 0 finite-slope-set by
auto
show ∃ sll ≥ sl. compSl K1 K2 h h' sll unfolding compSl-def
apply(rule exI[of - sll]) using 0 1 2 unfolding sll by auto
qed

```

```

fun Dcl-iter :: nat ⇒ 'node ⇒ 'node ⇒ ('pos ⇒ 'pos ⇒ slope ⇒ bool) set where
  Dcl-iter 0 nd nd' = (if {nd, nd'} ⊆ Node ∧ edge nd nd' then {λP P' sl. RR (nd, P)
  (nd', P') sl} else {})
  | Dcl-iter (Suc i) nd nd' =
    Dt (Dcl-iter i nd nd') ∪
    {compSl K1 K2 | K1 K2 nd''. nd'' ∈ Node ∧ K1 ∈ Dcl-iter i nd nd'' ∧ K2 ∈
  Dcl-iter i nd'' nd'})

```

```

lemma finite-compSl-set:
fixes D E::'node ⇒ ('pos ⇒ 'pos ⇒ slope ⇒ bool) set
assumes fin: ∧ nd''. finite (D nd'') ∧ finite (E nd'')
shows finite
  {compSl K1 K2 | K1 K2.
    ∃ nd''. nd'' ∈ Node ∧ K1 ∈ D nd'' ∧ K2 ∈ E nd''}
(is finite ?A)
proof–
  let ?B = ∪ { D nd'' × E nd'' | nd''. nd'' ∈ Node }
  define f where f ≡ λ (K1-K2::('pos ⇒ 'pos ⇒ slope ⇒ bool) × ('pos ⇒ 'pos ⇒ slope ⇒ bool)).

```

```

  compSl (fst K1-K2) (snd K1-K2)
have ?A ⊆ f ' ?B unfolding f-def image-def apply safe
  subgoal for - K1 K2 nd'' apply(rule bexI[of - (K1, K2)]) by auto .
moreover have finite ?B apply(rule finite-Union)
  subgoal using Node-finite by auto
  subgoal using fin by blast .
ultimately show finite ?A
  by (meson finite-imageI finite-subset)
qed

```

```

lemma finite-Dcl-iter: finite (Dcl-iter i nd nd')
apply(induct i arbitrary: nd nd')
  subgoal by auto
  subgoal for i nd nd' apply (auto intro!: finite-subset[OF Dt-incl]
    finite-compSl-set[of Dcl-iter i nd λnd''. Dcl-iter i nd'' nd']) . .

```

```

lemma finite-compSl-Dcl-iter:
finite {compSl K1 K2 | K1 K2. ∃ nd''. nd'' ∈ Node ∧ K1 ∈ Dcl-iter i nd nd'' ∧ K2
  ∈ Dcl-iter i nd'' nd'}
apply(rule finite-compSl-set) by (simp add: finite-Dcl-iter)

```

```

lemma Dcl-iter-SlopedRels:  $Dcl\text{-}iter\ i\ nd\ nd' \subseteq SlopedRels$ 
proof(induct i arbitrary: nd nd')
  case 0 thus ?case using RR-SlopeRels by auto
next
  case (Suc i)
  thus ?case using compSl-SlopeRels Dt-incl by simp (blast intro: Dt-aux)
qed

lemma Dcl-iter-subseqI:
  assumes  $\bigwedge R\ h\ h'\ sl.\ R \in Dcl\text{-}iter\ (Suc\ i)\ nd\ nd' \implies$ 
     $(h, h') \notin PosOf\ nd \times PosOf\ nd' \implies$ 
     $R\ h\ h'\ sl \implies$ 
    HOL.False
shows  $Dcl\text{-}iter\ (Suc\ i)\ nd\ nd' \subseteq \{R.\ \forall h\ h'\ sl.\ (h, h') \notin PosOf\ nd \times PosOf\ nd' \longrightarrow \neg R\ h\ h'\ sl\}$ 
using assms by auto

lemma Dcl-iter-su-PosOf:
 $Dcl\text{-}iter\ i\ nd\ nd' \subseteq$ 
 $\{R.\ \forall h\ h'\ sl.\ (h, h') \notin PosOf\ nd \times PosOf\ nd' \longrightarrow \neg R\ h\ h'\ sl\}$ 
proof (induct i arbitrary: nd nd')
  case 0 thus ?case using RR-PosOf by auto
next
  case (Suc i nd nd')
  show ?case
  proof (rule Dcl-iter-subseqI)
    fix  $R\ h\ h'\ sl$ 
    assume R-in:  $R \in Dcl\text{-}iter\ (Suc\ i)\ nd\ nd'$ 
    and h-notin:  $(h, h') \notin PosOf\ nd \times PosOf\ nd'$ 
    and R-h:  $R\ h\ h'\ sl$ 

    from R-in have  $R \in Dt\ (Dcl\text{-}iter\ i\ nd\ nd' \cup \{compSl\ K1\ K2\ |\ K1\ K2\ nd''.\ nd'' \in Node \wedge K1 \in Dcl\text{-}iter\ i\ nd\ nd'' \wedge K2 \in Dcl\text{-}iter\ i\ nd''\ nd'\})$ 
    by simp
    hence  $R \in Dcl\text{-}iter\ i\ nd\ nd' \cup \{compSl\ K1\ K2\ |\ K1\ K2\ nd''.\ nd'' \in Node \wedge K1 \in Dcl\text{-}iter\ i\ nd\ nd'' \wedge K2 \in Dcl\text{-}iter\ i\ nd''\ nd'\}$ 
    using Dt-incl by blast
    thus HOL.False
  proof
    assume  $R \in Dcl\text{-}iter\ i\ nd\ nd'$ 
    with Suc[of nd nd'] h-notin R-h show HOL.False by blast
  next
    assume  $R \in \{compSl\ K1\ K2\ |\ K1\ K2\ nd''.\ nd'' \in Node \wedge K1 \in Dcl\text{-}iter\ i\ nd\ nd'' \wedge K2 \in Dcl\text{-}iter\ i\ nd''\ nd'\}$ 
    then obtain  $K1\ K2\ nd''$  where  $R = compSl\ K1\ K2$ 
    and  $K1 \in Dcl\text{-}iter\ i\ nd\ nd''$ 
    and  $K2 \in Dcl\text{-}iter\ i\ nd''\ nd'$  by blast
    with R-h obtain  $P''\ sl1\ sl2$  where  $K1\ h\ P''\ sl1$  and  $K2\ P''\ h'\ sl2$ 

```



```

    unfolding compSl-def by blast

  have h ∈ PosOf nd
    using ⟨K1 ∈ Dcl-iter i nd nd'⟩ ⟨K1 h P'' sl1⟩ Suc[of nd nd'] by blast
  moreover have h' ∈ PosOf nd'
    using ⟨K2 ∈ Dcl-iter i nd'' nd'⟩ ⟨K2 P'' h' sl2⟩ Suc[of nd'' nd'] by blast
  ultimately have (h, h') ∈ PosOf nd × PosOf nd' by simp
  with h-notin show HOL.False by contradiction
qed
qed
qed

lemma Dcl-iter-nodes-out:
  nd ∉ Node ∨ nd' ∉ Node ⟹ Dcl-iter i nd nd' = {}
  apply(induct i arbitrary: nd nd')
  subgoal by auto
  subgoal by (auto simp add: Dt-def) .

lemma initFrag-Dt-trans:
  L ⊆ SlopedRels ⟹ finite L ⟹ initFrag L L' ⟹ initFrag (Dt L) L'
  using initFrag-Dt initFrag-trans by blast

lemma Dt-initFrag-aux: initFrag LL LL' ⟹ initFrag LL (Dt LL')
  using Dt-incl initFrag-def by auto

lemma initFrag-Un-left: initFrag LL LL' ⟹ initFrag (LL ∪ KK) LL'
  using initFrag-def by auto

lemma Dcl-iter-initFrag-Ccl-iter: initFrag (Dcl-iter i nd nd') (Ccl-iter i nd nd')
  proof(induct i arbitrary: nd nd')
    case 0 thus ?case unfolding initFrag-def leqSl-def by auto
  next
    case(Suc i nd nd')
    show ?case unfolding Ccl-iter.simps initFrag-Un proof safe
      have initFrag
        (Dcl-iter i nd nd' ∪
         {compSl K1 K2 | K1 K2.
          ∃ nd''. nd'' ∈ Node ∧ K1 ∈ Dcl-iter i nd nd'' ∧ K2 ∈ Dcl-iter i nd'' nd'})
        (Ccl-iter i nd nd')
        using Suc by (meson initFrag-Un-left)

    then show initFrag (Dcl-iter (Suc i) nd nd') (Ccl-iter i nd nd')
      using Suc unfolding initFrag-Un

```

```

    using finite-compSl-Dcl-iter finite-Dcl-iter initFrag-trans Dcl-iter-SlopedRels
    by(auto intro!: initFrag-Dt-trans compSl-SlopeRels,(blast+))
  next
    note 1 = Dcl-iter.simps(2)[of i nd nd']
    show initFrag (Dcl-iter (Suc i) nd nd')
      {ccompSl K1 K2 | K1 K2 nd''. nd'' ∈ Node ∧ K1 ∈ Ccl-iter i nd nd'' ∧ K2
    ∈ Ccl-iter i nd'' nd'}
    unfolding 1
    apply(rule initFrag-Dt-trans)
    subgoal using Dcl-iter-SlopedRels by (auto intro!: compSl-SlopeRels)
    subgoal unfolding finite-Un using finite-Dcl-iter finite-compSl-Dcl-iter by
  auto
    subgoal unfolding initFrag-def
    apply(intro ballI, clarsimp)
    apply(drule Suc[unfolded initFrag-def, rule-format])+
    apply safe
    subgoal for K1 K2 nd'' K1' K2' apply(rule bexI[of - compSl K1' K2'])
    by (auto simp: ccompSl-mono,meson compSl-leqSl-ccompSl ccompSl-mono
  leqSl-trans) . .
    qed
  qed

```

lemma *Ccl-iter-initFrag-Dcl-iter*: *initFrag (Ccl-iter i nd nd') (Dcl-iter i nd nd')*

proof(*induct i arbitrary: nd nd'*)

case 0 thus ?case unfolding *initFrag-def leqSl-def* by auto

next

case(Suc i nd nd')

show ?case unfolding *Dcl-iter.simps* apply(rule *Dt-initFrag-aux*)

unfolding *initFrag-Un* **proof** safe

show *initFrag (Ccl-iter (Suc i) nd nd') (Dcl-iter i nd nd')* **apply** *simp*

apply(rule *initFrag-Un-left*) **using** *Suc* **by** *auto*

next

note 1 = *Dcl-iter.simps*(2)[of i nd nd']

show *initFrag (Ccl-iter (Suc i) nd nd')*

{*compSl K1 K2 | K1 K2 nd''. nd'' ∈ Node ∧ K1 ∈ Dcl-iter i nd nd'' ∧ K2*

∈ Dcl-iter i nd'' nd'}

unfolding 1 **apply** *simp* unfolding *initFrag-def*

apply(*intro ballI, clarsimp*)

apply(*drule Suc[unfolded initFrag-def, rule-format]*)+

apply safe **subgoal** for *K1 K2 nd'' K1' K2'*

apply(rule *bexI*[of - *ccompSl K1' K2'*])

subgoal by (*meson ccompSl-leqSl-compSl ccompSl-mono compSl-leqSl-ccompSl*

leqSl-trans)

subgoal by *auto* . .

qed

qed

lemma *Dcl-iter-finite-range*:
assumes $nd \in \text{Node}$ $nd' \in \text{Node}$
shows $\text{finite } (\text{range } (\lambda i. \text{Dcl-iter } i \text{ } nd \text{ } nd'))$
proof –
let $?S = \{R. (\forall h \ h' (sl::\text{slope}). (h, h') \notin \text{PosOf } nd \times \text{PosOf } nd' \longrightarrow \neg R \ h \ h' \ sl)\}$
have $\text{range } (\lambda i. \text{Dcl-iter } i \text{ } nd \text{ } nd') \subseteq \text{Pow } ?S$
using *Dcl-iter-su-PosOf* **by** *blast*
moreover have $\text{finite } (\text{Pow } ?S)$
unfolding *finite-Pow-iff* **using** *finite-PosOf-prod*[*OF assms*] .
ultimately show $?thesis$ **by** (*metis* (*mono-tags*, *lifting*) *rev-finite-subset*)
qed

lemma *Dcl-iter-finite-range-all*:
 $\text{finite } (\text{range } (\lambda i. \text{Dcl-iter } i \text{ } nd \text{ } nd'))$
proof (*cases* $nd \in \text{Node} \wedge nd' \in \text{Node}$)
case *True* **thus** $?thesis$ **using** *Dcl-iter-finite-range* **by** *auto*
next
case *False*
hence $\forall i. \text{Dcl-iter } i \text{ } nd \text{ } nd' = \{\}$ **using** *Dcl-iter-nodes-out* **by** *auto*
hence $\text{range } (\lambda i. \text{Dcl-iter } i \text{ } nd \text{ } nd') = \{\{\}\}$ **by** *auto*
thus $?thesis$ **by** *simp*
qed

lemma *Dt-idem*: $Dt (Dt \ X) = Dt \ X$
unfolding *Dt-def* **by** *auto*

lemma *Dcl-iter-thin-0*: $Dt (\text{Dcl-iter } 0 \text{ } nd \text{ } nd') = \text{Dcl-iter } 0 \text{ } nd \text{ } nd'$
proof (*cases* $\{nd, nd'\} \subseteq \text{Node} \wedge \text{edge } nd \text{ } nd'$)
case *True*
hence $\text{Dcl-iter } 0 \text{ } nd \text{ } nd' = \{\lambda P \ P' \ sl. RR \ (nd, P) \ (nd', P') \ sl\}$ **by** *simp*
thus $?thesis$ **unfolding** *Dt-def lessSl-def leqSl-def* **by** *auto*
next
case *False*
hence $\text{Dcl-iter } 0 \text{ } nd \text{ } nd' = \{\}$ **by** *simp*
thus $?thesis$ **unfolding** *Dt-def* **by** *auto*
qed

lemma *Dcl-iter-thin*: $Dt (\text{Dcl-iter } i \text{ } nd \text{ } nd') = \text{Dcl-iter } i \text{ } nd \text{ } nd'$
proof (*cases* i)
case 0 **thus** $?thesis$ **using** *Dcl-iter-thin-0* **by** *simp*
next
case (*Suc* n)
have $\text{Dcl-iter } (\text{Suc } n) \text{ } nd \text{ } nd' = Dt (\text{Dcl-iter } n \text{ } nd \text{ } nd' \cup \{\text{compSl } K1 \ K2 \mid K1 \ K2 \text{ } nd'', nd'' \in \text{Node} \wedge K1 \in \text{Dcl-iter } n \text{ } nd \text{ } nd'' \wedge K2 \in \text{Dcl-iter } n \text{ } nd'' \text{ } nd'\})$
by *simp*
thus $?thesis$ **using** *Dt-idem Suc* **by** *simp*
qed

lemma *Dcl-iter-initFrag-Suc*:
initFrag (Dcl-iter (Suc i) nd nd') (Dcl-iter i nd nd')
proof –
 let $?X = \text{Dcl-iter } i \text{ nd nd}' \cup \{ \text{compSl } K1 \ K2 \mid K1 \ K2 \text{ nd}'', \text{ nd}'' \in \text{Node} \wedge K1 \in \text{Dcl-iter } i \text{ nd nd}'' \wedge K2 \in \text{Dcl-iter } i \text{ nd}'' \text{ nd}' \}$
 have $?X \subseteq \text{SlopedRels}$
 using *Dcl-iter-SlopedRels compSl-SlopeRels* **by** *blast*
 moreover have *finite* $?X$
 using *finite-Dcl-iter finite-compSl-Dcl-iter* **by** *auto*
 ultimately have *initFrag (Dt ?X) ?X*
 using *initFrag-Dt* **by** *blast*
 thus *?thesis*
 unfolding *Dcl-iter.simps initFrag-Un* **by** *blast*
qed

lemma *Dcl-iter-initFrag-le*:
 $i \leq j \implies \text{initFrag (Dcl-iter } j \text{ nd nd')} (\text{Dcl-iter } i \text{ nd nd}')$
apply(*induct j*)
subgoal **by** (*simp add: initFrag-def leqSl-refl*)
subgoal **for** j
 using *Ccl-iter-initFrag-Dcl-iter Dcl-iter-initFrag-Ccl-iter*
Dcl-iter-initFrag-Suc initFrag-trans le-Suc-eq
by *blast* .

lemma *Dt-initFrag-antisym*:
 assumes $A \subseteq \text{SlopedRels } B \subseteq \text{SlopedRels}$
 and $\text{Dt } A = A \text{ Dt } B = B$
 and $AB: \text{initFrag } A \ B$ and $BA: \text{initFrag } B \ A$
 shows $A = B$
proof (*rule subset-antisym; rule subsetI*)
 fix R assume $R \in A$
 from $\langle R \in A \rangle BA$ obtain R' where $R' \in B$ and $\text{leqSl } R' \ R$ unfolding *initFrag-def* **by** *blast*
 from $\langle R' \in B \rangle AB$ obtain R'' where $R'' \in A$ and $\text{leqSl } R'' \ R'$ unfolding *initFrag-def* **by** *blast*
 have $\text{leqSl } R'' \ R$ using $\langle \text{leqSl } R'' \ R' \rangle \langle \text{leqSl } R' \ R \rangle$ *leqSl-trans* **by** *blast*
 have $R'' = R$
proof (*rule ccontr*)
 assume $R'' \neq R$
 hence *lessSl* $R'' \ R$ using $\langle \text{leqSl } R'' \ R \rangle$ unfolding *lessSl-def* **by** *simp*
 with $\langle R'' \in A \rangle \langle R \in A \rangle$ have $R \notin \text{Dt } A$ unfolding *Dt-def* **by** *blast*
 with $\langle \text{Dt } A = A \rangle \langle R \in A \rangle$ show *HOL.False* **by** *simp*
qed
 hence $\text{leqSl } R \ R'$ using $\langle \text{leqSl } R'' \ R' \rangle$ **by** *simp*
 have $R = R'$ using $\langle \text{leqSl } R \ R' \rangle \langle \text{leqSl } R' \ R \rangle \langle R \in A \rangle \langle R' \in B \rangle$ *assms(1,2)*
leqSl-antisym **by** *blast*
 thus $R \in B$ using $\langle R' \in B \rangle$ **by** *simp*
next
 fix R assume $R \in B$

from $\langle R \in B \rangle$ *AB* **obtain** R' **where** $R' \in A$ **and** $\text{leqSl } R' R$ **unfolding** *initFrag-def* **by** *blast*
from $\langle R' \in A \rangle$ *BA* **obtain** R'' **where** $R'' \in B$ **and** $\text{leqSl } R'' R'$ **unfolding** *initFrag-def* **by** *blast*
have $\text{leqSl } R'' R$ **using** $\langle \text{leqSl } R'' R' \rangle \langle \text{leqSl } R' R \rangle$ *leqSl-trans* **by** *blast*
have $R'' = R$
proof (*rule ccontr*)
assume $R'' \neq R$
hence $\text{lessSl } R'' R$ **using** $\langle \text{leqSl } R'' R \rangle$ **unfolding** *lessSl-def* **by** *simp*
with $\langle R'' \in B \rangle \langle R \in B \rangle$ **have** $R \notin \text{Dt } B$ **unfolding** *Dt-def* **by** *blast*
with $\langle \text{Dt } B = B \rangle \langle R \in B \rangle$ **show** *HOL.False* **by** *simp*
qed
hence $\text{leqSl } R R'$ **using** $\langle \text{leqSl } R'' R' \rangle$ **by** *simp*
have $R = R'$ **using** $\langle \text{leqSl } R R' \rangle \langle \text{leqSl } R' R \rangle \langle R \in B \rangle \langle R' \in A \rangle$ *assms(1,2)*
leqSl-antisym **by** *blast*
thus $R \in A$ **using** $\langle R' \in A \rangle$ **by** *simp*
qed

lemma *Dcl-iter-Suc-stops-pair*: $\exists i. \text{Dcl-iter } (\text{Suc } i) \text{ nd nd}' = \text{Dcl-iter } i \text{ nd nd}'$
proof –

have *finite* (*range* ($\lambda i. \text{Dcl-iter } i \text{ nd nd}'$)) **by** (*rule Dcl-iter-finite-range-all*)
then obtain $i j$ **where** $i < j$ **and** $\text{eq: Dcl-iter } i \text{ nd nd}' = \text{Dcl-iter } j \text{ nd nd}'$
using *pigeonhole-infinite-seq* **by** *blast*

have *initFrag* ($\text{Dcl-iter } j \text{ nd nd}'$) ($\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}'$)
using $\langle i < j \rangle$ *Dcl-iter-initFrag-le Suc-leI* **by** *blast*
moreover have *initFrag* ($\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}'$) ($\text{Dcl-iter } i \text{ nd nd}'$)
using *Dcl-iter-initFrag-Suc* **by** *simp*
ultimately have *BA*: *initFrag* ($\text{Dcl-iter } i \text{ nd nd}'$) ($\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}'$)
unfolding *eq[symmetric]* **by** *simp*

have *AB*: *initFrag* ($\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}'$) ($\text{Dcl-iter } i \text{ nd nd}'$)
using *Dcl-iter-initFrag-Suc* **by** *simp*

have $\text{Dcl-iter } i \text{ nd nd}' \subseteq \text{SlopedRels}$ **and** $\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}' \subseteq \text{SlopedRels}$
using *Dcl-iter-SlopedRels* **by** *blast+*
moreover have $\text{Dt } (\text{Dcl-iter } i \text{ nd nd}') = \text{Dcl-iter } i \text{ nd nd}'$
and $\text{Dt } (\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}') = \text{Dcl-iter } (\text{Suc } i) \text{ nd nd}'$
using *Dcl-iter-thin* **by** *blast+*
ultimately have $\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}' = \text{Dcl-iter } i \text{ nd nd}'$
using *Dt-initFrag-antisym* [*of Dcl-iter } (\text{Suc } i) \text{ nd nd}' \text{ Dcl-iter } i \text{ nd nd}'*] *AB BA*

by *simp*

thus *?thesis* **by** *blast*

qed

lemma *Dcl-iter-eq-implies-Suc-eq*:

assumes $i < j$ $\text{Dcl-iter } i \text{ nd nd}' = \text{Dcl-iter } j \text{ nd nd}'$

shows $\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}' = \text{Dcl-iter } i \text{ nd nd}'$

proof –

have $\text{initFrag } (Dcl\text{-}iter\ j\ nd\ nd')\ (Dcl\text{-}iter\ (Suc\ i)\ nd\ nd')$
using $\langle i < j \rangle\ Dcl\text{-}iter\text{-}initFrag\text{-}le\ Suc\text{-}leI$ **by** *blast*
moreover have $\text{initFrag } (Dcl\text{-}iter\ (Suc\ i)\ nd\ nd')\ (Dcl\text{-}iter\ i\ nd\ nd')$
using $Dcl\text{-}iter\text{-}initFrag\text{-}Suc$ **by** *simp*
ultimately have $BA: \text{initFrag } (Dcl\text{-}iter\ i\ nd\ nd')\ (Dcl\text{-}iter\ (Suc\ i)\ nd\ nd')$
unfolding $\text{assms}(2)[\text{symmetric}]$ **by** *simp*
have $AB: \text{initFrag } (Dcl\text{-}iter\ (Suc\ i)\ nd\ nd')\ (Dcl\text{-}iter\ i\ nd\ nd')$
using $Dcl\text{-}iter\text{-}initFrag\text{-}Suc$ **by** *simp*
have $Dcl\text{-}iter\ i\ nd\ nd' \subseteq SlopedRels$ **and** $Dcl\text{-}iter\ (Suc\ i)\ nd\ nd' \subseteq SlopedRels$
using $Dcl\text{-}iter\text{-}SlopedRels$ **by** *blast+*
moreover have $Dt\ (Dcl\text{-}iter\ i\ nd\ nd') = Dcl\text{-}iter\ i\ nd\ nd'$
and $Dt\ (Dcl\text{-}iter\ (Suc\ i)\ nd\ nd') = Dcl\text{-}iter\ (Suc\ i)\ nd\ nd'$
using $Dcl\text{-}iter\text{-}thin$ **by** *blast+*
ultimately show *?thesis*
using $Dt\text{-}initFrag\text{-}antisym[\text{of } Dcl\text{-}iter\ (Suc\ i)\ nd\ nd'\ Dcl\text{-}iter\ i\ nd\ nd']\ AB\ BA$
by *simp*
qed

lemma $Dcl\text{-}iter\text{-}Suc\text{-}stops: \exists i. Dcl\text{-}iter\ (Suc\ i) = Dcl\text{-}iter\ i$
proof (*rule ccontr*)
assume $\neg(\exists i. Dcl\text{-}iter\ (Suc\ i) = Dcl\text{-}iter\ i)$
hence $\forall i. \exists nd\ nd'. Dcl\text{-}iter\ (Suc\ i)\ nd\ nd' \neq Dcl\text{-}iter\ i\ nd\ nd'$ **by** (*metis ext*)
hence $\forall i. \exists ndd. ndd \in Node \times Node \wedge Dcl\text{-}iter\ (Suc\ i)\ (fst\ ndd)\ (snd\ ndd) \neq Dcl\text{-}iter\ i\ (fst\ ndd)\ (snd\ ndd)$
by (*metis Dcl\text{-}iter\text{-}nodes\text{-}out SigmaI fst\text{-}eqD snd\text{-}eqD*)
then obtain ndi **where** $ndi: \forall i. ndi\ i \in Node \times Node \wedge Dcl\text{-}iter\ (Suc\ i)\ (fst\ (ndi\ i))\ (snd\ (ndi\ i)) \neq Dcl\text{-}iter\ i\ (fst\ (ndi\ i))\ (snd\ (ndi\ i))$
by *metis*

have $range\ ndi \subseteq Node \times Node$ **using** ndi **by** *blast*
hence $finite\ (range\ ndi)$ **using** $Node\text{-}finite\ finite\text{-}cartesian\text{-}product$ **by** (*metis rev\text{-}finite\text{-}subset*)

have $UNIV = (\bigcup ndd \in range\ ndi. \{i. ndi\ i = ndd\})$ **by** *auto*
moreover have $finite\ (range\ ndi)$ **by** *fact*
ultimately obtain ndd **where** $ndd \in range\ ndi$ **and** $\inf\text{-}I: infinite\ \{i. ndi\ i = ndd\}$
using $infinite\text{-}UNIV\text{-}nat\ finite\text{-}Union$ **by** (*metis (no\text{-}types, lifting) finite\text{-}UN\text{-}I*)

define $nd\ nd'\ I$ **where** $nd = fst\ ndd$ **and** $nd' = snd\ ndd$ **and** $I = \{i. ndi\ i = ndd\}$
have $I\text{-}prop: \forall i \in I. Dcl\text{-}iter\ (Suc\ i)\ nd\ nd' \neq Dcl\text{-}iter\ i\ nd\ nd'$
using $I\text{-}def\ nd\text{-}def\ nd'\text{-}def\ ndi$ **by** *auto*

have $I = (\bigcup R \in (\lambda i. Dcl\text{-}iter\ i\ nd\ nd')\ 'I. \{k \in I. Dcl\text{-}iter\ k\ nd\ nd' = R\})$ **by** *auto*
moreover have $finite\ ((\lambda i. Dcl\text{-}iter\ i\ nd\ nd')\ 'I)$
using $Dcl\text{-}iter\text{-}finite\text{-}range\text{-}all[\text{of } nd\ nd']$
by (*metis Set.basic\text{-}monos(1) range\text{-}subsetD finite\text{-}subset image\text{-}subset\text{-}iff*)

ultimately obtain R where $R \in (\lambda i. \text{Dcl-iter } i \text{ nd nd}') \text{ ' } I$ and $\text{inf-}R$: $\text{infinite } \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\}$

proof –

{ assume $\forall R \in (\lambda i. \text{Dcl-iter } i \text{ nd nd}') \text{ ' } I. \text{finite } \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\}$
 hence $\text{finite } (\bigcup R \in (\lambda i. \text{Dcl-iter } i \text{ nd nd}') \text{ ' } I. \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\})$
 using $\langle \text{finite } ((\lambda i. \text{Dcl-iter } i \text{ nd nd}') \text{ ' } I) \rangle$ by *blast*
 hence $\text{finite } I$
 using $\langle I = (\bigcup R \in (\lambda i. \text{Dcl-iter } i \text{ nd nd}') \text{ ' } I. \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\}) \rangle$

by *simp*

hence HOL.False

using $\text{inf-}I \text{ I-def}$ by *simp* }

thus $?thesis$

using *that* by *blast*

qed

from $\text{inf-}R$ have $\forall m. \exists n \geq m. n \in \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\}$

by (*simp add: infinite-nat-iff-unbounded-le*)

then obtain i where $i\text{-in}$: $i \in I$ and $i\text{-}R$: $\text{Dcl-iter } i \text{ nd nd}' = R$

by *blast*

have $\exists j \geq \text{Suc } i. j \in \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\}$

using $\langle \forall m. \exists n \geq m. n \in \{k \in I. \text{Dcl-iter } k \text{ nd nd}' = R\} \rangle$ by *blast*

then obtain j where $j\text{-in}$: $j \in I$ and $j\text{-}R$: $\text{Dcl-iter } j \text{ nd nd}' = R$ and $i < j$

by *auto*

have $\text{eq: Dcl-iter } i \text{ nd nd}' = \text{Dcl-iter } j \text{ nd nd}'$ using $i\text{-}R \ j\text{-}R$ by *simp*

have $\text{Dcl-iter } (\text{Suc } i) \text{ nd nd}' = \text{Dcl-iter } i \text{ nd nd}'$

using $\langle i < j \rangle \text{ eq Dcl-iter-eq-implies-Suc-eq}$ by *blast*

thus HOL.False using $I\text{-prop } i\text{-in}$ by *simp*

qed

lemma *Dcl-iter-Suc-stable*:

assumes $\text{Dcl-iter } (\text{Suc } i) = \text{Dcl-iter } i$

shows $\text{Dcl-iter } (\text{Suc } (\text{Suc } i)) = \text{Dcl-iter } i$

using *assms* unfolding *fun-eq-iff Dcl-iter.simps(2)* by *auto*

lemma *Dcl-iter-stable*:

assumes $\text{Dcl-iter } (\text{Suc } i) = \text{Dcl-iter } i$ and $j \geq i$

shows $\text{Dcl-iter } j = \text{Dcl-iter } i$

using *assms* apply(*induct j-i arbitrary: j i*)

subgoal by *auto*

subgoal for $x \ j$

by (*metis Suc-diff-le Suc-leD diff-diff-cancel diff-le-self Dcl-iter-Suc-stable*)

done

lemma *Dcl-iter-stops*: $\exists k. \forall j \geq k. \text{Dcl-iter } j = \text{Dcl-iter } k$

using *Dcl-iter-Suc-stops Dcl-iter-stable* by *blast*

definition $Dcl :: 'node \Rightarrow 'node \Rightarrow ('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool)$ set **where**
 $Dcl\ nd\ nd' \equiv Dcl\text{-}iter\ (LEAST\ k.\ \forall j \geq k.\ Dcl\text{-}iter\ j = Dcl\text{-}iter\ k)\ nd\ nd'$

lemma $Dcl\text{-}eq\text{-}some\text{-}Dcl\text{-}iter$: $\exists i.\ \forall j \geq i.\ Dcl = Dcl\text{-}iter\ j$

proof –

obtain k **where** $k\text{-}def$: $\forall j \geq k.\ Dcl\text{-}iter\ j = Dcl\text{-}iter\ k$ **using** $Dcl\text{-}iter\text{-}stops$ **by**
 $blast$
let $?k\text{-}least = LEAST\ k.\ \forall j \geq k.\ Dcl\text{-}iter\ j = Dcl\text{-}iter\ k$
have $\forall j \geq ?k\text{-}least.\ Dcl\text{-}iter\ j = Dcl\text{-}iter\ ?k\text{-}least$
using $k\text{-}def\ LeastI$ [of $\lambda k.\ \forall j \geq k.\ Dcl\text{-}iter\ j = Dcl\text{-}iter\ k$] **by** $blast$
thus $?thesis$ **unfolding** $Dcl\text{-}def\ fun\text{-}eq\text{-}iff$ **by** $blast$
qed

lemma $Dcl\text{-}initFrag\text{-}Ccl$: $initFrag\ (Dcl\ nd\ nd')\ (Ccl\ nd\ nd')$

proof –

obtain $k\text{-}dcl$ **where** $k\text{-}dcl$: $\forall j \geq k\text{-}dcl.\ Dcl = Dcl\text{-}iter\ j$ **using** $Dcl\text{-}eq\text{-}some\text{-}Dcl\text{-}iter$
by $blast$
obtain $k\text{-}ccl$ **where** $k\text{-}ccl$: $Ccl = Ccl\text{-}iter\ k\text{-}ccl$ **using** $Ccl\text{-}eq\text{-}some\text{-}Ccl\text{-}iter$ **by**
 $blast$
define $max\text{-}k$ **where** $max\text{-}k \equiv \max\ k\text{-}dcl\ k\text{-}ccl$
have 1 : $Dcl\ nd\ nd' = Dcl\text{-}iter\ max\text{-}k\ nd\ nd'$ **using** $k\text{-}dcl\ max\text{-}k\text{-}def$ **by** ($simp$
 add : $fun\text{-}eq\text{-}iff$)
have 2 : $Ccl\text{-}iter\ k\text{-}ccl\ nd\ nd' \subseteq Ccl\text{-}iter\ max\text{-}k\ nd\ nd'$ **using** $Ccl\text{-}iter\text{-}mono$
 $max\text{-}k\text{-}def$ **by** $auto$
have 3 : $Ccl\ nd\ nd' = Ccl\text{-}iter\ k\text{-}ccl\ nd\ nd'$ **using** $k\text{-}ccl$ **by** $simp$
moreover **have** $Ccl\ nd\ nd' = Ccl\text{-}iter\ max\text{-}k\ nd\ nd'$
using $2\ 3$ **unfolding** $Ccl\text{-}def$ **by** $auto$
ultimately show $?thesis$ **using** $Dcl\text{-}iter\text{-}initFrag\text{-}Ccl\text{-}iter$ [of $max\text{-}k$] 1 **by** $metis$
qed

lemma $Ccl\text{-}initFrag\text{-}Dcl$: $initFrag\ (Ccl\ nd\ nd')\ (Dcl\ nd\ nd')$

proof –

obtain $k\text{-}dcl$ **where** $k\text{-}dcl$: $\forall j \geq k\text{-}dcl.\ Dcl = Dcl\text{-}iter\ j$ **using** $Dcl\text{-}eq\text{-}some\text{-}Dcl\text{-}iter$
by $blast$
obtain $k\text{-}ccl$ **where** $k\text{-}ccl$: $Ccl = Ccl\text{-}iter\ k\text{-}ccl$ **using** $Ccl\text{-}eq\text{-}some\text{-}Ccl\text{-}iter$ **by**
 $blast$
define $max\text{-}k$ **where** $max\text{-}k \equiv \max\ k\text{-}dcl\ k\text{-}ccl$
have 1 : $Dcl\ nd\ nd' = Dcl\text{-}iter\ max\text{-}k\ nd\ nd'$ **using** $k\text{-}dcl\ max\text{-}k\text{-}def$ **by** ($simp$
 add : $fun\text{-}eq\text{-}iff$)
have 2 : $Ccl\text{-}iter\ k\text{-}ccl\ nd\ nd' \subseteq Ccl\text{-}iter\ max\text{-}k\ nd\ nd'$ **using** $Ccl\text{-}iter\text{-}mono$
 $max\text{-}k\text{-}def$ **by** $auto$
have 3 : $Ccl\ nd\ nd' = Ccl\text{-}iter\ k\text{-}ccl\ nd\ nd'$ **using** $k\text{-}ccl$ **by** $simp$
moreover **have** $Ccl\ nd\ nd' = Ccl\text{-}iter\ max\text{-}k\ nd\ nd'$
using $2\ 3$ **unfolding** $Ccl\text{-}def$ **by** $auto$
ultimately show $?thesis$ **using** $Ccl\text{-}iter\text{-}initFrag\text{-}Dcl\text{-}iter$ [of $max\text{-}k$] 1 **by** $metis$
qed

definition *TransitiveLooping* $\equiv \forall nd \in \text{Node}. \forall K \in \text{Dcl } nd \text{ nd}. (\exists P. \text{transSl } K \ P \ P \text{ Decr})$

lemma *TransitiveLooping-iff-TransitiveLoopingCcl*:
TransitiveLooping $\longleftrightarrow$ *TransitiveLoopingCcl*

proof *standard*

assume 1: *TransitiveLooping*
 show *TransitiveLoopingCcl*
 unfolding *TransitiveLoopingCcl-def* **proof** *safe*
 fix *nd K* **assume** *nd: nd ∈ Node and K ∈ Ccl nd nd*
 moreover **have** *initFrag (Dcl nd nd) (Ccl nd nd)*
 using *Dcl-initFrag-Ccl* **by** *blast*
 ultimately obtain *K'* **where** *le: leqSl K' K and K' ∈ Dcl nd nd*
 unfolding *initFrag-def* **by** *auto*
 then obtain *P* **where** *transSl K' P P Decr* **using** 1 *nd* **unfolding** *Transi-*
tiveLooping-def **by** *auto*
 with *le* **have** *transSl K P P Decr*
 using *transSl-mono*
 by (*metis leqSl-def less-eq-slope.elims(2) slope.distinct(1)*)
 thus $\exists P. \text{transSl } K \ P \ P \text{ Decr}$ **..**
 qed
next
 assume 1: *TransitiveLoopingCcl*
 show *TransitiveLooping*
 unfolding *TransitiveLooping-def* **proof** *safe*
 fix *nd K* **assume** *nd: nd ∈ Node and K ∈ Dcl nd nd*
 moreover **have** *initFrag (Ccl nd nd) (Dcl nd nd)*
 using *Ccl-initFrag-Dcl* **by** *blast*
 ultimately obtain *K'* **where** *le: leqSl K' K and K' ∈ Ccl nd nd*
 unfolding *initFrag-def* **by** *auto*
 then obtain *P* **where** *transSl K' P P Decr* **using** 1 *nd* **unfolding** *Transi-*
tiveLoopingCcl-def **by** *auto*
 with *le* **have** *transSl K P P Decr*
 using *transSl-mono*
 by (*metis leqSl-def less-eq-slope.elims(2) slope.distinct(1)*)
 thus $\exists P. \text{transSl } K \ P \ P \text{ Decr}$ **..**
 qed
qed

definition *descentPathSl* :: 'node list $\Rightarrow$ 'pos list $\Rightarrow$ slope list $\Rightarrow$ bool **where**
descentPathSl ndl Pl sll $\equiv$
length Pl = length ndl $\wedge$ length sll = length ndl - 1 $\wedge$
 $(\forall i < \text{length ndl} - 1. \text{RR } (ndl!i, Pl!i) (ndl!(\text{Suc } i), Pl!(\text{Suc } i)) (sll!i))$

lemma *descentPathSl-append*:

assumes *w*: *descentPathSl* ndl1 Pl1 sll1 *descentPathSl* ndl2 Pl2 sll2 **and** *l*: last
ndl1 = hd ndl2 last Pl1 = hd Pl2

and *lndl*: length ndl1 ≥ 2 length ndl2 ≥ 2

shows *descentPathSl* (ndl1 @ tl ndl2) (Pl1 @ tl Pl2) (sll1 @ sll2)

proof –

have *lPl1*: length Pl1 = length ndl1 length Pl2 = length ndl2

using *w*(1) *w*(2) *descentPathSl-def* **by** *auto*

hence *lPl*: length Pl1 ≥ 2 length Pl2 ≥ 2 **using** *lndl* **by** *auto*

have *lsll*: length sll1 = length ndl1 - 1 length sll2 = length ndl2 - 1

using *w*(1) *w*(2) *descentPathSl-def* **by** *auto*

hence *sll*: length sll1 ≥ 1 length sll2 ≥ 1 **using** *lndl* **by** *auto*

have [*simp*]: hd ndl2 = ndl2!0 hd (tl ndl2) = ndl2!(Suc 0)

hd Pl2 = Pl2!0 hd (tl Pl2) = Pl2!(Suc 0)

hd sll2 = sll2!0

apply (metis *diff-is-0-eq'* *hd-conv-nth* *length-0-conv* *lsll*(2) *lsll*(2) *not-one-le-zero* *zero-le-one*)

apply (metis *One-nat-def* *Suc-le-eq* *hd-conv-nth* *length-greater-0-conv* *length-tl* *lsll*(2) *lsll*(2) *List.nth-tl*)

apply (metis *diff-is-0-eq'* *hd-conv-nth* *lPl*(2) *list.size*(3) *lsll*(2) *lsll*(2) *not-one-le-zero* *zero-le-one*)

apply (metis *One-nat-def* *Suc-le-eq* *hd-conv-nth* *lPl*(2) *length-greater-0-conv* *length-tl* *lsll*(2) *lsll*(2) *List.nth-tl*)

by (metis *hd-conv-nth* *length-0-conv* *lsll*(2) *not-one-le-zero*)

show *?thesis* **unfolding** *descentPathSl-def* **apply** *safe*

subgoal using *assms* **unfolding** *descentPathSl-def* **by** *simp*

subgoal using *assms* **unfolding** *descentPathSl-def* **by** *auto*

subgoal for *i* **proof** –

assume *ii*: *i* < length (ndl1 @ tl ndl2) - 1

show *RR* ((ndl1 @ tl ndl2) ! *i*, (Pl1 @ tl Pl2) ! *i*) ((ndl1 @ tl ndl2) ! *Suc i*,
(Pl1 @ tl Pl2) ! *Suc i*) ((sll1 @ sll2) ! *i*)

proof(*cases i* < length ndl1 - 1)

case *True*

thus *?thesis* **using** *assms* **unfolding** *descentPathSl-def*

by *simp*

next

case *False* **note** *i* = *False*

show *?thesis*

proof(*cases i* = length ndl1 - 1)

case *True*

hence 1: (ndl1 @ tl ndl2) ! *i* = last ndl1

```

      by (metis One-nat-def diff-Suc-less last-conv-nth length-greater-0-conv
less-imp-diff-less
      less-le-trans lsl(1) lssl(1) not-less nth-append zero-less-Suc)
    have 2: (Pl1 @ tl Pl2) ! i = last Pl1
    by (metis One-nat-def Suc-n-not-le-n True add-diff-inverse-nat lPPl(1)
last-conv-nth lessI
      list.size(3) lsl(1) lssl(1) nat-diff-split nth-append plus-1-eq-Suc)
    have 3: (ndl1 @ tl ndl2) ! Suc i = hd (tl ndl2) unfolding nth-append
  by (metis (full-types) True add-diff-inverse-nat cancel-comm-monoid-add-class.diff-cancel

      hd-conv-nth le-imp-less-Suc length-tl less-not-refl3 list.size(3) lsl
      lssl nat-diff-split plus-1-eq-Suc)
    have 4: (Pl1 @ tl Pl2) ! Suc i = hd (tl Pl2) unfolding nth-append
  by (metis (full-types) True add-diff-inverse-nat cancel-comm-monoid-add-class.diff-cancel
hd-conv-nth
      lPPl(1) lPPl(2) le-imp-less-Suc length-tl less-not-refl3
      list.size(3) lsl lssl nat-diff-split plus-1-eq-Suc)
    have 5: (sll1 @ sll2) ! i = hd sll2
  by (metis True hd-Cons-tl length-greater-0-conv less-le-trans less-numeral-extra(1)

      lsl(2) lssl(1) nth-append-length)
  show ?thesis using w(2) lsl(2) unfolding 1 2 3 4 5 l descentPathSl-def
by auto
next
case False hence i:  $i \geq \text{length } ndl1$   $i \geq \text{length } Pl1$  using i lPPl by auto
hence [simp]:  $\neg i < \text{length } sll1$  using lssl(1) by linarith
define j where j:  $j \equiv \text{Suc } (i - \text{length } ndl1)$ 
have [simp]:  $tl\ ndl2 ! (i - \text{length } ndl1) = ndl2!j$ 
by (metis Suc-leD Suc-le-mono add-diff-inverse-nat hd-Cons-tl j list.size(3)
lsl(2)
      lssl(2) nat-diff-split not-one-le-zero nth-Cons-Suc plus-1-eq-Suc)
have [simp]:  $tl\ Pl2 ! (i - \text{length } Pl1) = Pl2!j$ 
by (metis diff-Suc-1 diff-is-0-eq hd-Cons-tl j lPPl(1) lPPl(2)
      list.size(3) lsl(2) lssl(2) not-less-eq-eq nth-Cons-Suc zero-diff)
have [simp]:  $tl\ ndl2 ! (\text{Suc } i - \text{length } ndl1) = ndl2!(\text{Suc } j)$ 
by (metis Suc-diff-le Suc-leD Suc-le-mono add-diff-inverse-nat hd-Cons-tl
i(1) j list.size(3)
      lsl(2) lssl(2) nat-diff-split not-one-le-zero nth-Cons-Suc plus-1-eq-Suc)
have [simp]:  $tl\ Pl2 ! (\text{Suc } i - \text{length } Pl1) = Pl2!(\text{Suc } j)$ 
by (metis Suc-diff-le hd-Cons-tl i(2) j lPPl(1) lPPl(2) length-greater-0-conv
length-tl less-le-trans
      less-numeral-extra(1) lsl(2) lssl(2) nth-Cons-Suc tl-Nil)
have [simp]:  $sll2 ! (i - \text{length } sll1) = sll2!j$ 
using Suc-diff-le i(1) j lsl(1) lssl(1)
by (metis One-nat-def Suc-n-minus-m-eq le-simps(3) zero-less-diff)
have  $j < \text{length } ndl2 - 1$  unfolding j using ii i(1) by auto
thus ?thesis unfolding nth-append using i w(2) unfolding descent-
PathSl-def by simp
qed

```

qed
 qed .
 qed

lemma *descentPathSl-append-invert-singl*:
assumes *pathG ndl descentPathSl (ndl @ [nd]) Pl' sll'*
shows $\exists Pl\ P\ sll\ sl.$
 $descentPathSl\ ndl\ Pl\ sll \wedge Pl' = Pl @ [P] \wedge sll' = sll @ [sl] \wedge RR\ (last\ ndl, last\ Pl)\ (nd, P)\ sl$
proof –
obtain $P\ Pl$ **where** $Pl': Pl' = Pl @ [P]$ **using** *assms unfolding descentPathSl-def*

by (*metis append-butlast-last-id append-is-Nil-conv length-0-conv not-Cons-self2*)
obtain $sl\ sll$ **where** $sll': sll' = sll @ [sl]$ **using** *assms unfolding descentPathSl-def Pl' apply simp*
by (*metis Graph.not-path-Nil append-Nil2 append-butlast-last-id append-eq-append-conv*
 $length-append-singleton\ length-butlast$)
show *?thesis* **apply** (*intro exI[of - P] exI[of - Pl] exI[of - sl] exI[of - sll]*)
unfolding $Pl'\ sll'$ **using** *assms unfolding descentPathSl-def*
by (*smt One-nat-def Pl' Suc-mono append-is-Nil-conv diff-Suc-1 last-conv-nth length-0-conv*
 $length-append-singleton\ lessI\ less-SucI\ nth-append\ nth-append-length\ sll'$)
 qed

lemma *descentPathSl-append-invert*:
assumes $p1: pathG\ ndl1\ \text{and}\ pathG\ ndl2\ last\ ndl1 = hd\ ndl2\ descentPathSl\ (ndl1 @ (tl\ ndl2))\ Pl\ sll$
shows $\exists Pl1\ Pl2\ sll1\ sll2. last\ Pl1 = hd\ Pl2 \wedge descentPathSl\ ndl1\ Pl1\ sll1 \wedge descentPathSl\ ndl2\ Pl2\ sll2 \wedge Pl = Pl1 @ (tl\ Pl2) \wedge sll = sll1 @ sll2$
using *assms(2-4) proof (induction arbitrary: Pl sll)*
case (*Base nd nd' Pl Sll*)
then show *?case* **using** *descentPathSl-append-invert-singl[OF p1 Base(4)][simplified]*
apply *safe*
subgoal **for** $Pl1\ Pl2\ sll1\ sl2$
apply (*rule exI[of - Pl1]*) **apply** (*rule exI[of - [last Pl1, P2]]*)
apply (*rule exI[of - sll1]*) **apply** (*rule exI[of - [sl2]]*)
by (*simp add: descentPathSl-def*) .
next
case (*Step nd2 ndl2 Pl' sll'*)
have $0: last\ ndl1 = hd\ ndl2$ **using** *Step* **by** (*metis hd-append not-path-Nil*)
obtain $P2\ Pl$ **where** $Pl': Pl' = Pl @ [P2]$
by (*metis Nil-is-append-conv Step.hyps(2) Step.prem(2)*
 $length-0-conv\ list.distinct(1)\ not-path-Nil\ rev-exhaust\ tl-append2\ descentPathSl-def$)
have $l: length\ Pl = length\ (ndl1 @ tl\ ndl2)$
by (*metis Pl' Step.hyps(2) Step.prem(2) append.assoc butlast-snoc length-butlast not-path-Nil*)

```

    tl-append2 descentPathSl-def)
  have sll' ≠ []
  using Step.premis(2) unfolding Pl'
  by (metis One-nat-def Step.hyps Step.premis(1)
      append-is-Nil-conv diff-is-0-eq hd-Cons-tl list.size(3) not-Cons-self2
      not-less-eq-eq numeral-2-eq-2 p1 pathG.Step path-append-last path-length-ge2
      descentPathSl-def)
  then obtain sl2 sll where sll': sll' = sll @ [sl2] using rev-exhaust by blast
  have 1: descentPathSl (ndl1 @ tl ndl2) Pl sll using Step.premis(2) unfolding
  Pl' sll' unfolding descentPathSl-def apply safe
  subgoal using l by auto
  subgoal using l by auto
  subgoal for i apply (cases i < length ndl1)
  subgoal using One-nat-def Step.hyps(2) Suc-mono add-diff-cancel-left' ap-
  pend.assoc butlast-snoc l
  List.length-append List.length-tl less-SucI not-path-Nil nth-butlast plus-1-eq-Suc
  tl-append2
  by (smt (verit, ccfv-SIG) length-append-singleton)

  subgoal by (smt One-nat-def Step.hyps(2) Step.premis(2) Suc-mono add-diff-cancel-left'
  append.assoc
  butlast-snoc length-append length-append-singleton
  less-SucI not-path-Nil nth-butlast plus-1-eq-Suc sll' tl-append2 descent-
  PathSl-def l) . .
  from Step.IH[OF 0 1] obtain Pl1 Pl2 sll1 sll2 where
  la: last Pl1 = hd Pl2 and w: descentPathSl ndl1 Pl1 sll1 descentPathSl ndl2 Pl2
  sll2
  and Pl: Pl = Pl1 @ tl Pl2 and sll: sll = sll1 @ sll2 by auto
  have lndl: length ndl1 ≥ 2 length ndl2 ≥ 2 by (simp-all add: Step.hyps(2) p1
  path-length-ge2)
  have lPPl: length Pl1 = length ndl1 length Pl2 = length ndl2
  using w(1) w(2) descentPathSl-def by auto
  hence lPl: length Pl1 ≥ 2 length Pl2 ≥ 2 using lndl by auto
  have lssl: length sll1 = length ndl1 - 1 length sll2 = length ndl2 - 1
  using w(1) w(2) descentPathSl-def by auto
  hence lsl: length sll1 ≥ 1 length sll2 ≥ 1 using lndl by auto
  have [simp]: ¬ length ndl1 + length ndl2 - 2 < length ndl1 using lndl by
  linarith
  have [simp]: ¬ length ndl1 + length ndl2 - 2 < length Pl1 using w(1) descent-
  PathSl-def by auto
  have [simp]: length ndl1 + length ndl2 - Suc (Suc (length Pl1)) < length Pl2
  - Suc 0
  using lPPl(1) lPPl(2) lndl(2) by linarith
  have [simp]: ¬ Suc (length ndl1 + length ndl2 - 2) < length ndl1
  using Suc-lessD <¬ length ndl1 + length ndl2 - 2 < length ndl1> by blast
  have [simp]: ¬ Suc (length ndl1 + length ndl2 - 2) < length Pl1
  by (simp add: lPPl(1))
  have [simp]: ¬ length ndl1 + length ndl2 - Suc (length Pl1) < length Pl2 -
  Suc 0

```

```

    by (simp add: lPPl(1) lPPl(2))
  have [simp]:  $\neg \text{length } \text{ndl1} + \text{length } \text{ndl2} - 2 < \text{length } \text{sll1}$ 
    using less-diff-conv lssl(1) by auto
  have [simp]:  $\neg \text{length } \text{ndl1} + \text{length } \text{ndl2} - \text{Suc } (\text{Suc } (\text{length } \text{sll1})) < \text{length } \text{sll2}$ 
    using lndl(1) lssl(1) lssl(2) by auto

  have a:  $\neg \text{Suc } (\text{length } \text{ndl1} + \text{length } \text{ndl2} - 2) < \text{length } \text{ndl1}$ 
     $\neg \text{length } \text{ndl1} + \text{length } \text{ndl2} - \text{Suc } (\text{Suc } (\text{length } \text{sll1})) < \text{length } \text{sll2}$ 
     $\langle \neg \text{length } \text{ndl1} + \text{length } \text{ndl2} - 2 < \text{length } \text{ndl1} \rangle$ 
     $\langle \neg \text{length } \text{ndl1} + \text{length } \text{ndl2} - 2 < \text{length } \text{sll1} \rangle$  by simp-all

  have aux:  $RR \text{ (last } \text{ndl2}, \text{last } \text{Pl2}) (\text{nd2}, \text{P2}) \text{sl2}$ 
  using Step.premis(2) unfolding Pl' sll' Pl sll descentPathSl-def apply clarsimp

  apply(erule allE[of -  $\text{length } \text{ndl1} + \text{length } \text{ndl2} - 2$ ])
  unfolding nth-append
  using One-nat-def Suc-diff-Suc Suc-le-lessD add-diff-cancel-left' butlast-snoc diff-Suc-Suc

    lPPl(1) last-conv-nth last-tl length-greater-0-conv length-tl lessI less-add-Suc1
  less-add-same-cancel1
    lssl(2) lssl(2) nth-append-length nth-butlast numeral-2-eq-2 plus-1-eq-Suc
  tl-append2
    List.nth-append by (smt (z3) a)

  show ?case
  apply(rule exI[of - Pl1]) apply(rule exI[of - Pl2 @ [P2]])
  apply(rule exI[of - sll1]) apply(rule exI[of - sll2 @ [sl2]])
  apply safe
  subgoal by (metis Graph.not-path-Nil Step.hyps(2) hd-append2 la length-greater-0-conv
w(2) descentPathSl-def)
  subgoal using w(1) by blast
  subgoal unfolding descentPathSl-def apply safe
  subgoal using w(2) descentPathSl-def by auto
  subgoal by (metis Step.hyps(2) length-append-singleton length-tl not-path-Nil
tl-append2 w(2) descentPathSl-def)
  subgoal for i apply(subst nth-append) apply(cases i < length ndl2-1)
  subgoal using w unfolding descentPathSl-def by simp
  subgoal by simp (metis (no-types, lifting) One-nat-def Suc-pred aux but-
last-snoc gr-implies-not0
    last-conv-nth length-greater-0-conv
    not-less-less-Suc-eq nth-append-length nth-butlast w(2) descentPathSl-def
zero-less-Suc) . .
  subgoal by (metis Pl Pl' Step.hyps(2) append-eq-append-conv2 lPPl(2) length-0-conv
not-path-Nil tl-append2)
  subgoal by (simp add: sll sll') .
qed

```

lemma *descentPathSl-wfLabL*:
assumes *pathG ndl* **and** *descentPathSl ndl Pl sll*
shows *wfLabL ndl Pl*
using *assms unfolding descentPathSl-def wfLabL-def*
by (*metis (no-types, lifting) Nitpick.size-list-simp(2) RR-PosOf length-tl less-Suc-eq not-less numeral-2-eq-2 path-length-ge2*)

definition *tracksPers* :: ('pos $\Rightarrow$ 'pos $\Rightarrow$ slope $\Rightarrow$ bool) $\Rightarrow$ 'node list $\Rightarrow$ bool **where**

tracksPers K ndl $\equiv$
 $(\forall Pl\ sll. \text{descentPathSl } ndl\ Pl\ sll \longrightarrow K\ (hd\ Pl)\ (last\ Pl)\ (MaxSl\ (set\ sll))) \wedge$
 $(\forall P\ P'\ sl. K\ P\ P'\ sl \longrightarrow (\exists Pl\ sll. \text{descentPathSl } ndl\ Pl\ sll \wedge hd\ Pl = P \wedge last\ Pl = P' \wedge sl = MaxSl\ (set\ sll)))$

proposition *tracksPers-leftTotal*:

assumes *ndl*: *pathG ndl*

shows $\exists K \in Ccl\ (hd\ ndl)\ (last\ ndl). \text{tracksPers } K\ ndl$

using *ndl proof induct*

case (*Base nd nd'*)

show ?*case apply*(*rule* *bexI[of - $\lambda P\ P'\ sl. RR\ (nd, P)\ (nd', P')\ sl]$*)

subgoal unfolding *tracksPers-def* **apply** *safe*

subgoal for *Pl sll* **unfolding** *descentPathSl-def MaxSl-def*

by *simp (metis (full-types) One-nat-def diff-Suc-1 hd-conv-nth in-set-conv-nth*

last-conv-nth less-one list.size(3) old.nat.distinct(2) slope.exhaust)

subgoal for *P P' sl* **apply**(*intro exI[of - $[P, P']$ exI[of - $[sl]$]*)

unfolding *descentPathSl-def* **by** *auto* .

subgoal using *Base* **by** *auto* .

next

case (*Step nd ndl*)

then obtain *K* **where** *K*: $K \in Ccl\ (hd\ ndl)\ (last\ ndl)$ **and** *cor*: *tracksPers K ndl*

by *auto*

have *lndl*: $length\ ndl \geq 2$ **using** *Step.hyps(2) path-iff-nth* **by** *blast*

have [*simp*]: $hd\ (ndl\ @\ [nd]) = hd\ ndl$

by (*metis Graph.not-path-Nil Step.hyps(2) hd-append2*)

define *K1* **where** *K1* $\equiv \lambda P\ P'\ sl. RR\ (last\ ndl, P)\ (nd, P')\ sl$

have *K1*: $K1 \in Ccl\ (last\ ndl)\ nd$ **unfolding** *K1-def* **apply**(*rule Ccl-RR*)

using *Step Ccl-nodes edge (last ndl) nd* **by** *blast+*

show ?*case apply*(*rule bexI[of - ccompSl K K1]*)

subgoal unfolding *tracksPers-def* **proof** *safe*

fix *Pl' sll'* **assume** *descentPathSl (ndl @ [nd]) Pl' sll'*

then obtain *P Pl sl sll* **where** *w*: *descentPathSl ndl Pl sll*

and *1*: $Pl' = Pl\ @\ [P]\ sll' = sll\ @\ [sl]$ **and** *edge*: $RR\ (last\ ndl, last\ Pl)\ (nd,$

P) sl
using *descentPathSl-append-invert-singl*[$OF \langle pathG \ ndl \rangle$] **by** *blast*
have lPl : $length \ Pl \geq 2$ **using** w **by** (*simp add: lndl descentPathSl-def*)

have 2 : $hd \ Pl' = hd \ Pl \ last \ Pl' = P$ **unfolding** 1
subgoal
by (*metis Cons-eq-appendI Step.hyps(2) length-0-conv list.exhaust-sel list.sel(1)*
not-path-Nil w descentPathSl-def)
subgoal by simp .

from w **have** K : $K \ (hd \ Pl) \ (last \ Pl) \ (MaxSl \ (set \ sll))$ **using** *cor unfolding*
tracksPers-def by auto
show $ccompSl \ K \ K1 \ (hd \ Pl') \ (last \ Pl') \ (MaxSl \ (set \ sll'))$
unfolding *ccompSl-def* **apply**(*rule exI[of - last Pl]*)
apply(*rule exI[of - MaxSl (set sll)]*) **apply**(*rule exI[of - sl]*) **apply** *safe*
subgoal unfolding 1 *MaxSl-def* **by** *auto*
subgoal unfolding 2 **by** *fact*
subgoal unfolding $K1$ -def 2 **by** *fact .*
next
fix $P \ P' \ sl$ **assume** 0 : $ccompSl \ K \ K1 \ P \ P' \ sl$
then obtain $P'' \ sl1 \ sl2$ **where** sl : $sl = MaxSl \ \{sl1, \ sl2\}$
and K : $K \ P \ P'' \ sl1$ **and** $K1 \ P'' \ P' \ sl2$ **unfolding** *ccompSl-def* **by** *auto*
hence $edge$: $RR \ (last \ ndl, \ P'') \ (nd, \ P') \ sl2$ **unfolding** $K1$ -def **by** *simp*
obtain $Pl \ sll$ **where** w : *descentPathSl ndl Pl sll* **and** 1 : $hd \ Pl = P \ last \ Pl =$
 $P'' \ sl1 = MaxSl \ (set \ sll)$
using K *cor unfolding tracksPers-def by blast*
show $\exists \ Pl' \ sll'. \ descentPathSl \ (ndl \ @ \ [ndl]) \ Pl' \ sll' \wedge \ hd \ Pl' = P \wedge \ last \ Pl' =$
 $P' \wedge \ sl = MaxSl \ (set \ sll')$
apply(*intro exI[of - Pl @ [P']] exI[of - sll @ [sl2]]*) **apply** *safe*
subgoal using w **unfolding** *descentPathSl-def*
by *simp (smt 1(2) One-nat-def Step.hyps(2) Suc-lessI Suc-pred edge but-*
last-snoc last-conv-nth
length-greater-0-conv less-Suc-eq not-path-Nil nth-append-length nth-butlast)
subgoal by (*metis 1(1) Step.hyps(2) hd-append2 length-greater-0-conv*
not-path-Nil w descentPathSl-def)
subgoal by simp
subgoal unfolding $1 \ sl$ **unfolding** *MaxSl-def* **by** *auto .*
qed
subgoal using *Ccl-ccompSl*[$OF - K \ K1$] **using** K *Ccl-nodes* **by** *auto .*
qed

proposition *tracksPers-rightTotal*:
assumes $K \in Ccl \ nd \ nd'$
shows $\exists \ ndl. \ pathG \ ndl \wedge \ hd \ ndl = nd \wedge \ last \ ndl = nd' \wedge \ tracksPers \ K \ ndl$
proof –
obtain i **where** $K \in Ccl\text{-}iter \ i \ nd \ nd'$ **using** *assms unfolding Ccl-def* **by** *auto*
thus *?thesis* **proof**(*induction i arbitrary: K nd nd'*)
case $(0 \ K \ nd \ nd')$
hence $K : K = (\lambda P \ P'. \ RR \ (nd, \ P) \ (nd', \ P'))$ **and** $edge: \{nd, nd'\} \subseteq Node$


```

edge nd nd' by (auto split: if-splits)
  show ?case apply(intro exI[of - [nd, nd']]) apply safe
  subgoal using Base edge by (simp add: path-two-incl)
  subgoal by simp
  subgoal by simp
  subgoal unfolding tracksPers-def apply safe
    subgoal for Pl sll unfolding descentPathSl-def K
      by simp (metis (full-types) MaxSl-def One-nat-def diff-Suc-1 hd-conv-nth
in-set-conv-nth last-conv-nth less-one list.size(3) nat.distinct(1) slope.exhaust)
    subgoal for P P' sl apply(intro exI[of - [P, P']] exI[of - [sl]])
      unfolding descentPathSl-def K by simp . .
next
case (Suc i K nd nd')
show ?case proof(cases K ∈ Ccl-iter i nd nd')
  case True
  show ?thesis using Suc.IH[OF True] .
next
case False
with ⟨K ∈ Ccl-iter (Suc i) nd nd'⟩ obtain K1 K2 nd'' where
  K: K = ccompSl K1 K2 and nd'': nd'' ∈ Node
  and K1: K1 ∈ Ccl-iter i nd nd'' and K2: K2 ∈ Ccl-iter i nd'' nd'
by auto
from Suc.IH[OF K1] Suc.IH[OF K2] obtain ndl1 ndl2 where
  ndl1: pathG ndl1 hd ndl1 = nd last ndl1 = nd'' and cor1: tracksPers K1
ndl1 and
  ndl2: pathG ndl2 hd ndl2 = nd'' last ndl2 = nd' and cor2: tracksPers K2
ndl2
by auto
define ndl where ndl ≡ ndl1 @ tl ndl2

show ?thesis apply(rule exI[of - ndl]) apply safe
  subgoal by (metis ndl-def list.exhaust-sel ndl1(1) ndl1(3) ndl2(1) ndl2(2)
not-path-Nil path-append-last)
  subgoal by (metis ndl-def hd-append ndl1(1) ndl1(2) not-path-Nil)
  subgoal by (metis append-Nil2 append-butlast-last-id hd-Cons-tl last-appendR
last-tl ndl1(1) ndl1(3)
ndl2(1) ndl2(2) ndl2(3) ndl-def not-path-Nil)
  subgoal unfolding tracksPers-def proof safe
    fix Pl sll assume w: descentPathSl ndl Pl sll
    from descentPathSl-append-invert[OF ndl1(1) ndl2(1) - w[unfolded
ndl-def]]
    obtain Pl1 Pl2 sll1 sll2 where
      w1: descentPathSl ndl1 Pl1 sll1 and w2: descentPathSl ndl2 Pl2 sll2 and
      4: last Pl1 = hd Pl2 Pl = Pl1 @ (tl Pl2) sll = sll1 @ sll2
    using ndl1(3) ndl2(2) by auto
    from w1 cor1 have K1: K1 (hd Pl1) (last Pl1) (MaxSl (set sll1))
unfolding tracksPers-def by auto
    from w2 cor2 have K2: K2 (hd Pl2) (last Pl2) (MaxSl (set sll2))
unfolding tracksPers-def by auto

```

```

show  $K$  ( $hd\ Pl$ ) ( $last\ Pl$ ) ( $MaxSl\ (set\ sll)$ ) unfolding  $K\ ccompSl-def$ 
apply( $rule\ exI[of - last\ Pl1]$ )
apply( $rule\ exI[of - MaxSl\ (set\ sll1)]$ ) apply( $rule\ exI[of - MaxSl\ (set\ sll2)]$ )

apply safe
  subgoal unfolding  $4\ MaxSl-def$  by simp

    subgoal by ( $metis\ 4(2)\ K1\ hd-append2\ list.size(3)\ ndl1(1)\ not-less$ 
       $path-length-ge2\ pos2\ w1\ descentPathSl-def$ )
    subgoal by ( $metis\ 4(1)\ 4(2)\ K2\ append-Nil2\ hd-Cons-tl\ last-ConsL$ 
       $last-appendR\ last-tl$ 
       $list.size(3)\ ndl2(1)\ not-less\ path-length-ge2\ pos2\ w2\ descentPathSl-def$ )
    .

  next
    fix  $P\ P'\ sl$  assume  $K\ P\ P'\ sl$ 
    then obtain  $P''\ sl1\ sl2$  where  $sl: sl = MaxSl\ \{sl1, sl2\}$ 
    and  $K: K1\ P\ P''\ sl1\ K2\ P''\ P'\ sl2$  unfolding  $K\ ccompSl-def$  by auto
    from  $K(1)$  obtain  $Pl1\ sll1$  where
       $1: descentPathSl\ ndl1\ Pl1\ sll1\ hd\ Pl1 = P\ last\ Pl1 = P''\ sl1 = MaxSl\ (set$ 
 $sll1)$ 
    using cor1 unfolding tracksPers-def by blast
    from  $K(2)$  obtain  $Pl2\ sll2$  where
       $2: descentPathSl\ ndl2\ Pl2\ sll2\ hd\ Pl2 = P''\ last\ Pl2 = P'\ sl2 = MaxSl$ 
 $(set\ sll2)$ 
    using cor2 unfolding tracksPers-def by blast

    show  $\exists\ Pl\ sll. descentPathSl\ ndl\ Pl\ sll \wedge hd\ Pl = P \wedge last\ Pl = P' \wedge sl$ 
 $= MaxSl\ (set\ sll)$ 
    apply( $intro\ exI[of - Pl1\ @\ tl\ Pl2]\ exI[of - sll1\ @\ sll2]$ ) apply safe
    subgoal using  $1(1)\ 2(1)\ 1(3)\ 2(2)$ 
    by ( $simp\ add: ndl1(1)\ ndl1(3)\ ndl2(1)\ ndl2(2)\ ndl-def\ path-length-ge2$ 
 $descentPathSl-append$ )
    subgoal by ( $metis\ 1(1,2)\ hd-append2\ length-0-conv\ ndl1(1)\ not-path-Nil$ 
 $descentPathSl-def$ )
    subgoal by ( $metis\ 1(3)\ 2(1-3)\ append-Nil2\ hd-Cons-tl\ last-ConsL$ 
 $last-appendR\ last-tl$ 
 $length-0-conv\ ndl2(1)\ not-path-Nil\ descentPathSl-def$ )
    subgoal unfolding  $sl\ 1\ 2\ MaxSl-def$  by simp .
  qed .
qed
qed
qed

```

definition $descentPathParam\ ndl\ Pl\ sl \equiv$

$(\forall i. \text{Suc } i < \text{length } \text{ndl} \longrightarrow$
 $\quad \text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ Main} \vee$
 $\quad \neg \text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ Main} \wedge$
 $\quad \text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ Decr} \wedge \text{sl} = \text{Decr})$
 $\wedge$
 $(\exists i. \text{Suc } i < \text{length } \text{ndl} \wedge \text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ sl})$

lemma *descentPath-descentPathParam*:
descentPath *ndl Pl* $\longleftrightarrow$ *descentPathParam* *ndl Pl Decr*
using *descentPathParam-def descentPath-def* **by** *auto*

lemma *descentPathSl-descentPathParam*:
assumes *pathG* *ndl* **and** *descentPathSl* *ndl Pl sll*
shows *descentPathParam* *ndl Pl (MaxSl (set sll))*
using *assms unfolding descentPathSl-def descentPathParam-def MaxSl-def*
using *assms unfolding descentPathSl-def descentPathParam-def MaxSl-def*
apply *safe*
subgoal by (*metis (full-types) Nitpick.size-list-simp(2)*
One-nat-def Suc-less-eq length-tl not-path-Nil slope.exhaust)
subgoal by (*metis (full-types) Suc-lessE diff-Suc-1 in-set-conv-nth slope.exhaust*)
subgoal by (*smt Nitpick.size-list-simp(2) One-nat-def Suc-n-not-le-n diff-Suc-less*
in-set-conv-nth length-greater-0-conv length-tl less-trans-Suc not-path-Nil
numeral-2-eq-2 path-length-ge2 slope.exhaust) .

lemma *descentPathParam-append*:
assumes *ndl*: *ndl1* $\neq []$ *ndl2* $\neq []$ *length* *ndl1* = *length* *Pl1* *length* *ndl2* = *length* *Pl2*
and *d*: *descentPathParam* *ndl1 Pl1 sl1 descentPathParam* *ndl2 Pl2 sl2*
and *l*: *last* *ndl1* = *hd* *ndl2* *last* *Pl1* = *hd* *Pl2*
shows *descentPathParam* (*ndl1* @ (*tl* *ndl2*)) (*Pl1* @ (*tl* *Pl2*)) (*MaxSl* {*sl1*, *sl2*})
proof–
define *ndl Pl sl* **where** *ndl* $\equiv$ *ndl1* @ *tl* *ndl2* **and** *Pl* $\equiv$ *Pl1* @ *tl* *Pl2* **and** *sl* $\equiv$ *MaxSl* {*sl1*, *sl2*}
have *sl-Decr[simp]*: *sl* = *Decr* $\longleftrightarrow$ *sl1* = *Decr* $\vee$ *sl2* = *Decr*
unfolding *sl-def MaxSl-def* **using** *slope.exhaust* **by** *auto*
show *?thesis*
unfolding *descentPathParam-def ndl-def[symmetric] Pl-def[symmetric] sl-def[symmetric]*

proof(*intro allI conjI impI*)
fix *i* **assume** *Suc i* < *length* *ndl*
show *RR* (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *Main* $\vee$
 $\neg$ *RR* (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *Main* $\wedge$
 $\text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ Decr} \wedge \text{sl} = \text{Decr}$
proof(*cases* *Suc i* < *length* *ndl1*)
case *True*
hence *1*: *RR* (*ndl1* ! *i*, *Pl1* ! *i*) (*ndl1* ! *Suc i*, *Pl1* ! *Suc i*) *Main* $\vee$

```

    ¬ RR (ndl1 ! i, Pl1 ! i) (ndl1 ! Suc i, Pl1 ! Suc i) Main ∧
    RR (ndl1 ! i, Pl1 ! i) (ndl1 ! Suc i, Pl1 ! Suc i) Decr ∧ sl1 = Decr
  using d(1) unfolding descentPathParam-def by auto
  have 2: ndl1 ! (Suc i) = ndl ! (Suc i) Pl1 ! (Suc i) = Pl ! (Suc i)
    ndl1 ! i = ndl ! i Pl1 ! i = Pl ! i
  apply (metis True ndl(3) ndl-def nth-append)
  apply (metis Pl-def True ndl(3) nth-append)
  apply (metis Suc-lessD True ndl(3) ndl-def nth-append)
  by (metis Pl-def Suc-lessD True ndl(3) nth-append)
  show ?thesis using 1 unfolding 2 by auto
next
case False
show ?thesis proof(cases i < length ndl1)
  case True hence i: i = length ndl1 - Suc 0 using False by linarith
  have 2: ndl ! (Suc i) = ndl2 ! (Suc 0) Pl ! (Suc i) = Pl2 ! (Suc 0)
    ndl ! i = ndl2 ! 0 Pl ! i = Pl2 ! 0
  apply (metis False Suc-diff-Suc diff-self-eq-0 i length-greater-0-conv
    list.exhaust-sel minus-nat.diff-0 ndl(1) ndl(2) ndl-def nth-Cons-Suc
    nth-append)
    apply (metis False Pl-def Suc-lessI True ‹Suc i < length ndl› append-Nil2
      cancel-comm-monoid-add-class.diff-cancel length-greater-0-conv length-tl
      ndl(3) ndl(4)
      ndl-def nth-append nth-tl assms(2))
    apply (metis One-nat-def True hd-conv-nth i l(1) last-conv-nth ndl(1) ndl(2)
      ndl-def nth-append)
    by (metis One-nat-def Pl-def True hd-conv-nth i l(2) last-conv-nth length-0-conv
      ndl nth-append)
    have Suc 0 < length ndl2 using False ‹Suc i < length ndl› ndl-def by auto
    thus ?thesis using d(2) unfolding 2 descentPathParam-def by auto
  next
  case False
  define j where j ≡ Suc (i - length ndl1)
  have 2: ndl ! i = ndl2 ! j Pl ! i = Pl2 ! j
    ndl ! (Suc i) = ndl2 ! (Suc j)
    Pl ! (Suc i) = Pl2 ! (Suc j)
  unfolding j-def
  apply (metis False hd-Cons-tl ndl(2) ndl-def nth-Cons-Suc nth-append)
  apply (metis False Pl-def hd-Cons-tl length-0-conv ndl(2) ndl(3) ndl(4)
    nth-Cons-Suc nth-append)
  apply (metis False Suc-diff-le Suc-lessD hd-Cons-tl ndl(2) ndl-def not-less
    nth-Cons-Suc nth-append)
  by (metis (mono-tags, lifting) False Pl-def Suc-diff-le Suc-lessD hd-Cons-tl
    length-0-conv
    ndl(2) ndl(3) ndl(4) not-less nth-Cons-Suc nth-append)
  have Suc j < length ndl2 using False ‹Suc i < length ndl› j-def ndl-def by
  auto
  thus ?thesis using d(2) unfolding 2 descentPathParam-def by auto
qed
qed

```

```

next
  show  $\exists i. \text{Suc } i < \text{length } \text{ndl} \wedge \text{RR } (\text{ndl} ! i, \text{Pl} ! i) (\text{ndl} ! \text{Suc } i, \text{Pl} ! \text{Suc } i) \text{ sl}$ 
  proof (cases  $\text{sl} = \text{sl1}$ )
    case True
      obtain  $i$  where  $\text{Suc } i < \text{length } \text{ndl1} \wedge \text{RR } (\text{ndl1} ! i, \text{Pl1} ! i) (\text{ndl1} ! \text{Suc } i, \text{Pl1} ! \text{Suc } i) \text{ sl1}$ 
      using  $d(1)$  descentPathParam-def by auto
      thus ?thesis apply (intro exI[of -  $i$ ])
      by (metis Pl-def Suc-lessD add-lessD1 length-append ndl(3) ndl-def not-less-eq nth-append True)
    next
      case False note  $\text{sl} = \text{False}$ 
      hence  $\text{sl}: \text{sl} = \text{sl2}$  unfolding sl-def MaxSl-def by (cases  $\text{sl1}$ , auto)
      obtain  $i$  where  $\text{si}: \text{Suc } i < \text{length } \text{ndl2} \wedge \text{RR } (\text{ndl2} ! i, \text{Pl2} ! i) (\text{ndl2} ! \text{Suc } i, \text{Pl2} ! \text{Suc } i) \text{ sl2}$ 
      using  $d(2)$  descentPathParam-def by auto
      show ?thesis proof (cases  $i=0$ )
        case True
          define  $j$  where  $j \equiv (\text{length } \text{ndl1} + i - \text{Suc } 0)$ 
          have 2:  $\text{ndl} ! j = \text{ndl2} ! i \text{ Pl} ! j = \text{Pl2} ! i$ 
               $\text{ndl} ! (\text{Suc } j) = \text{ndl2} ! (\text{Suc } i) \text{ Pl} ! (\text{Suc } j) = \text{Pl2} ! (\text{Suc } i)$ 
          unfolding j-def
          apply (metis True Nat.add-0-right One-nat-def diff-less hd-conv-nth  $l(1)$  last-conv-nth length-greater-0-conv
               $\text{ndl}(1,2)$  ndl-def not-less-eq not-less-zero nth-append)
          apply (metis True Nat.add-0-right One-nat-def Pl-def diff-Suc-less hd-conv-nth  $l(2)$  last-conv-nth length-greater-0-conv
              ndl nth-append)
          apply (metis True Nat.add-0-right Nitpick.size-list-simp(2) One-nat-def length-tl list.exhaust-sel ndl(1) ndl(2) ndl-def
              nth-Cons-Suc nth-append-length-plus)
          by (metis True Nat.add-0-right Nitpick.size-list-simp(2) Pl-def diff-add-zero length-tl
               $\text{ndl}(1,3,4)$  not-add-less1 nth-append nth-tl plus-1-eq-Suc si zero-less-diff list.sel(2))
          have  $\text{Suc } j < \text{length } \text{ndl}$  using j-def ndl(1) ndl-def si by auto
          thus ?thesis apply (intro exI[of -  $j$ ]) using si unfolding 2 sl by auto
        next
          case False
            define  $j$  where  $j \equiv \text{length } \text{ndl1} + i - \text{Suc } 0$ 
            have  $\text{ndl} ! j = (\text{tl } \text{ndl2}) ! (i - \text{Suc } 0) \text{ Pl} ! j = \text{Pl2} ! i$ 
            and 2:  $\text{ndl} ! (\text{Suc } j) = \text{ndl2} ! (\text{Suc } i) \text{ Pl} ! (\text{Suc } j) = \text{Pl2} ! (\text{Suc } i)$ 
            unfolding j-def
            apply (metis False Nat.add-diff-assoc One-nat-def Suc-pred diff-add-zero diff-is-0-eq
                less-Suc-eq ndl-def nth-append-length-plus plus-1-eq-Suc zero-less-Suc)
            apply (metis False Nat.add-diff-assoc Pl-def Suc-leI Suc-pred length-0-conv list.exhaust-sel
                nat-neq-iff ndl(2-4) not-less-eq nth-Cons-Suc nth-append-length-plus)

```

zero-less-Suc)
apply (*metis Nitpick.size-list-simp*(2) *Suc-less-eq Suc-pred add-diff-cancel-left'*
add-gr-0
length-greater-0-conv ndl(1) ndl(2) ndl-def not-add-less1 nth-append nth-tl
si)
by (*metis Nitpick.size-list-simp*(2) *Pl-def Suc-less-eq Suc-pred add-diff-cancel-left'*

add-gr-0 length-greater-0-conv ndl not-add-less1 nth-append nth-tl si)
hence $\exists j: \text{ndl} ! j = \text{ndl2} ! i \text{ Pl} ! j = \text{Pl2} ! i$
unfolding *j-def*
apply (*metis False Nitpick.size-list-simp*(2) *Suc-lessD Suc-pred less-Suc-eq*
ndl(2) nth-tl si zero-less-Suc)
using $\langle \text{Pl} ! j = \text{Pl2} ! i \rangle$ *j-def* **by** *blast*
have *Suc j < length ndl* **using** *j-def ndl(1) ndl-def si* **by** *auto*
thus *?thesis* **apply**(*intro exI[of - j]*) **using** *si unfolding 2 3 sl* **by** *auto*
qed
qed
qed
qed

lemma *tracksPers-transSl-impl-ex-descentPath-repeat*:
assumes *ndl: cycleG ndl* **and** *tr: tracksPers K ndl* **and** *P: transSl K P P' sl*
shows $\exists \text{Pl } n. n \neq 0 \wedge \text{wfLabL } (\text{repeat } n \text{ (butlast ndl)} @ [\text{last ndl}]) \text{Pl} \wedge$
descentPathParam (repeat n (butlast ndl) @ [last ndl]) Pl sl $\wedge$
hd Pl = P $\wedge$ *last Pl = P'*

proof–
have *ndlNe[simp]: ndl $\neq []$* **and** *pndl[simp]: pathG ndl*
and *ndlhl[simp]: hd ndl = last ndl*
using *cycleG-def ndl not-path-Nil* **by** *auto*
show *?thesis* **using** *P* **proof**(*induction*)
case (*Base P P' sl*)
then obtain *Pl sll* **where** *Pl: descentPathSl ndl Pl sll* *hd Pl = P* *last Pl = P'*
and *sl: sl = MaxSl (set sll)* **using** *tr* **unfolding** *tracksPers-def* **by** *blast*
then show *?case* **apply**(*intro exI[of - Pl] exI[of - Suc 0]*)
using *descentPathSl-wfLabL descentPathSl-descentPathParam* **by** *auto*
next
case (*Step P P' sl1 P'' sl2*)
then obtain *Pl1 n1* **where** *n1: n1 $\neq 0$* **and** *Pl:*
wfLabL (repeat n1 (butlast ndl) @ [last ndl]) Pl1
descentPathParam (repeat n1 (butlast ndl) @ [last ndl]) Pl1 sl1
hd Pl1 = P *last Pl1 = P'* **by** *auto*
from $\langle K P' P'' sl2 \rangle$
obtain *Pl2 sll2* **where** *descentPathSl ndl Pl2 sll2* **and** *htPl2: hd Pl2 = P'* *last*
Pl2 = P''
and *sl2: sl2 = MaxSl (set sll2)* **using** *tr* **unfolding** *tracksPers-def* **by** *blast*
hence *Pl2: wfLabL ndl Pl2 descentPathParam ndl Pl2 sl2*
using *descentPathSl-wfLabL descentPathSl-descentPathParam* **by** *auto*

have *0: repeat (Suc n1) (butlast ndl) @ [last ndl] =*

```

      (repeat n1 (butlast ndl) @ [last ndl]) @ tl ndl
unfolding repeat-Suc2
by (metis append.assoc append-Cons append-Nil
      append-butlast-last-id list.exhaust-sel ndlNe ndlhl)

have Pl1facts[simp]: length Pl1 = Suc (n1 * (length ndl - Suc 0))
using Pl(1) wfLabL-def by auto

have Pl2facts[simp]: Pl2 ≠ [] length Pl2 = length ndl
      length (tl Pl2) = length ndl - Suc 0
using ⟨descentPathSl ndl Pl2 sl2⟩ descentPathSl-def by auto

show ?case apply (intro exI[of - Pl1 @ tl Pl2] exI[of - Suc n1]) apply safe
  subgoal unfolding 0 apply (subst wfLabL-append)
    subgoal by simp
    subgoal by simp
    subgoal apply safe
      subgoal by fact
      subgoal using Pl2(1) ndlNe wfLabL-tl by blast . .
    subgoal unfolding 0 apply (rule descentPathParam-append)
      subgoal by simp
      subgoal by simp
      subgoal by simp
      subgoal by simp
      subgoal by fact
      subgoal by fact
      subgoal by simp
      subgoal by (simp add: Pl(4) htPl2(1)) .
    subgoal by (metis Pl(3) Pl1facts hd-append list.size(3) old.nat.distinct(2))
    subgoal by (metis Pl(4) Pl2facts(1) append.right-neutral
      htPl2(1) htPl2(2) last.simps last-appendR list.exhaust-sel) .
qed
qed

lemma tracksPers-transSl-impl-ex-descentPathSlS:
assumes ndl: cycleG ndl and tr: tracksPers K ndl and K: (∃ P. transSl K P P
  Decr)
shows ∃ Ps. descentIPathS (srepeat (butlast ndl)) Ps
using cycle-descentPath-repeat-imp-descentIPathS-srepeat[OF ndl]
using tracksPers-transSl-impl-ex-descentPath-repeat[OF ndl tr, where sl = Decr]
using K unfolding descentPath-descentPathParam[symmetric] by metis

lemma wfLabL-descentPathParam-append-inverse:
fixes ndl1 ndl2
defines ndl ≡ ndl1 @ tl ndl2
assumes ndl: length ndl1 ≥ 2 length ndl2 ≥ 2 last ndl1 = hd ndl2
and w: wfLabL ndl Pl and d: descentPathParam ndl Pl sl
shows ∃ Pl1 Pl2 sl1 sl2.
  Pl = Pl1 @ (tl Pl2) ∧ sl = MaxSl {sl1, sl2} ∧ last Pl1 = hd Pl2 ∧

```

```

wfLabL ndl1 Pl1 ∧ descentPathParam ndl1 Pl1 sl1 ∧
wfLabL ndl2 Pl2 ∧ descentPathParam ndl2 Pl2 sl2
proof –
  have ndl1 ≠ [] ndl2 ≠ [] using ndl by force+
  note ndl = this ndl(3) ndl(1,2)
  define l1 where l1: l1 ≡ length ndl1
  define Pl1 Pl2 where Pl1: Pl1 ≡ take l1 Pl and Pl2: Pl2 ≡ drop (l1 – Suc 0)
Pl
  have lPl: length Pl = length ndl1 + length ndl2 – Suc 0
  using w unfolding wfLabL-def by (simp add: ndl-def Suc-leI ndl(2))
  have Plne: Pl ≠ [] Pl1 ≠ [] Pl2 ≠ []
  using Suc-diff-Suc lPl ndl(1) ndl(2) apply fastforce
  apply (metis Pl1 Suc-pred add-pos-pos l1 lPl length-greater-0-conv list.size(3)
ndl(1–2) one-is-add take-eq-Nil)
  by (metis Pl2 Suc-pred add-diff-cancel-left' add-pos-pos diff-Suc-Suc l1 lPl length-drop

    length-greater-0-conv ndl(1) ndl(2))
  have lPl1: length Pl1 = l1
  by (metis Pl1 Suc-pred add-diff-cancel-left' add-gr-0 drop-all l1 lPl length-drop
    length-greater-0-conv length-take min.absorb2 ndl(2) not-less-eq-eq)
  have lPl2: length Pl2 = length ndl2 by (simp add: Pl2 l1 lPl ndl(1))

  have Pl: Pl = Pl1 @ tl Pl2
  by (metis One-nat-def Pl1 Pl2 Suc-pred' append-take-drop-id drop-Suc l1 length-greater-0-conv
ndl(1) tl-drop)
  have lstPl1: last Pl1 = Pl!(l1 – Suc 0)
  by (metis One-nat-def Pl Plne(2) diff-less lPl1 last-conv-nth length-greater-0-conv
lessI nth-append)
  have hdPl2: hd Pl2 = Pl!(l1 – Suc 0) using Pl2 Plne(3) drop-all hd-drop-conv-nth
not-less by blast
  have lh: last Pl1 = hd Pl2 by (simp add: hdPl2 lstPl1)

  have s0l1: 0 < l1 Suc 0 < l1 using assms(2) l1 by linarith+
  have s0l2: 0 < length ndl2 Suc 0 < length ndl2 using assms(3) by auto
  have s0l: 0 < length ndl Suc 0 < length ndl using l1 ndl-def s0l1 by auto

  have Pli[simp]: ∧i. i < l1 ⇒ Pl1 ! i = Pl ! i ∧ i. i < l1 ⇒ ndl1 ! i = ndl ! i
  ∧i. i < length ndl2 ⇒ Pl2 ! i = Pl ! (l1 + i – Suc 0)
  ∧i. i < length ndl2 ⇒ ndl2 ! i = ndl ! (l1 + i – Suc 0)
  subgoal by (simp add: Pl1 l1 ndl-def nth-append)
  subgoal by (simp add: Pl1 l1 ndl-def nth-append)
  subgoal for i using Pl1 Pl2 Plne(2) Plne(3) by auto
  subgoal for i unfolding ndl-def apply(cases i=0)
  subgoal apply(subst nth-append)
  by simp (metis One-nat-def Plne(2) diff-Suc-1 gr0-implies-Suc hd-conv-nth l1
lPl1 last-conv-nth
    length-greater-0-conv less-Suc-eq ndl(3))
  subgoal apply(subst nth-append)
  by simp (metis One-nat-def Suc-eq-plus1 Suc-pred add-diff-cancel-left add-gr-0

```



```

hd-Cons-tl l1
  less-add-same-cancel1 ndl(2) not-less-eq nth-Cons-Suc) . .

  have wfLabL ndl1 Pl1 using w lPl1 unfolding wfLabL-def by (metis l1 ndl-def
    Pl add-strict-mono lPl less-diff-conv nth-append-left s0l2(2))
  have wfLabL ndl2 Pl2 using w Pl2 Pl1(2) lPl2 s0l1 unfolding wfLabL-def by
    (auto simp add: lPl2)

  show ?thesis proof(rule exI[of - Pl1], rule exI[of - Pl2])
  show  $\exists s1\ s2.$ 
    Pl = Pl1 @ (tl Pl2)  $\wedge$  sl = MaxSl {s1,s2}  $\wedge$  last Pl1 = hd Pl2  $\wedge$ 
    wfLabL ndl1 Pl1  $\wedge$  descentPathParam ndl1 Pl1 s1  $\wedge$ 
    wfLabL ndl2 Pl2  $\wedge$  descentPathParam ndl2 Pl2 s2
  proof(cases sl)
  case Main note sl = Main[simp]
  have 0:  $\bigwedge ii. \text{Suc } ii < \text{length } ndl \implies RR\ (ndl ! ii, Pl ! ii)\ (ndl ! \text{Suc } ii, Pl !$ 
    Suc ii) Main
  using d unfolding descentPathParam-def by (simp add: Pl l1 lPl1 lPl2
    ndl-def)
  show ?thesis proof(intro exI[of - Main], safe)
  show Pl = Pl1 @ tl Pl2 last Pl1 = hd Pl2 wfLabL ndl1 Pl1 wfLabL ndl2
    Pl2 by fact+
  show sl = MaxSl {Main, Main} unfolding MaxSl-def by simp
  show descentPathParam ndl1 Pl1 Main unfolding descentPathParam-def
    apply(intro conjI allI impI)
  subgoal for i unfolding l1[symmetric]
  by (smt 0 Pl Pl1(1,2) Suc-lessD Suc-less-eq Suc-pred add-gr-0 l1 lPl lPl1
    lPl2 length-append
    length-greater-0-conv length-tl less-add-same-cancel1 less-trans-Suc ndl(2)
    ndl-def)
  subgoal apply(rule exI[of - 0]) apply safe
  subgoal using ndl by linarith
  subgoal using 0[of 0] using s0l1 s0l by (simp add: l1) . .
  show descentPathParam ndl2 Pl2 Main unfolding descentPathParam-def
    apply(intro conjI allI impI)
  subgoal for i by simp (smt 0 Nat.add-diff-assoc2 One-nat-def Pl Suc-leI
    add-Suc-right add-eq-if
    l1 lPl lPl1 lPl2 length-append length-tl nat-add-left-cancel-less ndl-def
    not-gr-zero s0l1(1))
  subgoal apply(rule exI[of - 0]) apply safe
  subgoal using ndl by linarith
  subgoal using 0[of 0] using s0l2 s0l
  by simp (metis 0 One-nat-def Suc-pred l1 length-append length-tl
    less-Suc-eq less-add-same-cancel1 ndl-def plus-1-eq-Suc s0l1(1)) . .
  qed
next
case Decr note sl = Decr
then obtain ii where
  ii: Suc ii < length ndl RR (ndl ! ii, Pl ! ii) (ndl ! Suc ii, Pl ! Suc ii) Decr

```

```

using d unfolding descentPathParam-def by auto

define sl1 sl2 where
  sl1  $\equiv$  if ( $\exists i. \text{Suc } i < \text{length } \text{ndl1} \wedge \text{RR } (\text{ndl1} ! i, \text{Pl1} ! i) (\text{ndl1} ! \text{Suc } i, \text{Pl1} ! \text{Suc } i) \text{ Decr}$ )
    then Decr else Main and
  sl2  $\equiv$  if ( $\exists i. \text{Suc } i < \text{length } \text{ndl2} \wedge \text{RR } (\text{ndl2} ! i, \text{Pl2} ! i) (\text{ndl2} ! \text{Suc } i, \text{Pl2} ! \text{Suc } i) \text{ Decr}$ )
    then Decr else Main
have sl12: sl1 = Decr  $\vee$  sl2 = Decr
using ii apply (cases ii < l1)
subgoal apply (cases Suc ii = l1)
subgoal apply(subgoal-tac sl2 = Decr)
subgoal by simp
subgoal using s0l2(2) unfolding sl2-def by (auto intro!: exI[of - 0])
.
subgoal apply(subgoal-tac sl1 = Decr)
subgoal by simp
subgoal using Suc-lessD l1 unfolding sl1-def by (auto intro!: exI[of - ii]) . .
subgoal apply(subgoal-tac sl2 = Decr)
subgoal by simp
subgoal using Suc-lessD l1 unfolding sl2-def
by (auto intro!: exI[of - Suc ii - l1], simp add: ndl-def) . .

show ?thesis proof(rule exI[of - sl1], rule exI[of - sl2], safe)
show Pl = Pl1 @ tl Pl2 last Pl1 = hd Pl2 wfLabL ndl1 Pl1 wfLabL ndl2
Pl2 by fact+
show sl = MaxSl {sl1, sl2} using sl12 sl unfolding MaxSl-def by auto
have 1:  $\bigwedge i. \text{Suc } i < \text{length } \text{ndl1} \implies$ 
   $\text{RR } (\text{ndl1} ! i, \text{Pl1} ! i) (\text{ndl1} ! \text{Suc } i, \text{Pl1} ! \text{Suc } i) \text{ Main} \vee$ 
 $\neg \text{RR } (\text{ndl1} ! i, \text{Pl1} ! i) (\text{ndl1} ! \text{Suc } i, \text{Pl1} ! \text{Suc } i) \text{ Main} \wedge$ 
 $\text{RR } (\text{ndl1} ! i, \text{Pl1} ! i) (\text{ndl1} ! \text{Suc } i, \text{Pl1} ! \text{Suc } i) \text{ Decr} \wedge \text{sl1} =$ 
Decr
using d sl unfolding sl1-def unfolding l1[symmetric] unfolding descentPathParam-def
by (smt Pl Suc-lessD Suc-less-eq Suc-pred add-gr-0 l1 lPl lPl1 lPl2
  length-append length-tl less-add-same-cancel1 less-trans-Suc ndl-def
nth-append s0l2(1))
show descentPathParam ndl1 Pl1 sl1 unfolding descentPathParam-def
apply(intro conjI allI impI)
subgoal by fact
subgoal apply (cases sl1)
subgoal using 1 l1 s0l1(2) by blast
subgoal using sl1-def slope.distinct(1) by presburger . .
have 2:  $\bigwedge i. \text{Suc } i < \text{length } \text{ndl2} \implies$ 
   $\text{RR } (\text{ndl2} ! i, \text{Pl2} ! i) (\text{ndl2} ! \text{Suc } i, \text{Pl2} ! \text{Suc } i) \text{ Main} \vee$ 
 $\neg \text{RR } (\text{ndl2} ! i, \text{Pl2} ! i) (\text{ndl2} ! \text{Suc } i, \text{Pl2} ! \text{Suc } i) \text{ Main} \wedge$ 
 $\text{RR } (\text{ndl2} ! i, \text{Pl2} ! i) (\text{ndl2} ! \text{Suc } i, \text{Pl2} ! \text{Suc } i) \text{ Decr} \wedge \text{sl2} = \text{Decr}$ 

```

```

      using d sl unfolding sl2-def unfolding l1[symmetric] unfolding de-
scentPathParam-def
    by simp (smt Nat.add-diff-assoc Nat.add-diff-assoc2 wfLabL-def Suc-pred
add-diff-cancel-left'
      add-gr-0 l1 lPl le-add1 nat-add-left-cancel-less plus-1-eq-Suc s0l1(1)
w)
    show descentPathParam ndl2 Pl2 sl2 unfolding descentPathParam-def
apply(intro conjI allI impI)
  subgoal by fact
  subgoal apply (cases sl2)
  subgoal using 2 s0l2(2) by blast
  subgoal using sl2-def slope.distinct(1) by presburger . .
qed
qed
qed
qed

```

lemma *wfLabL-descentPathParam-imp-descentPathSl*:
assumes *w*: *wfLabL* *ndl* *Pl* **and** *d*: *descentPathParam* *ndl* *Pl* *sl*
shows $\exists sll. \text{descentPathSl } ndl \ Pl \ sll \wedge sl = \text{MaxSl } (\text{set } sll)$
proof–

```

  have l[simp]: length Pl = length ndl
  using assms(1) wfLabL-def by auto

```

```

define sll where sll =
  fToList (length ndl – Suc 0)
  (λi. if RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) sl then sl else
    if RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) Main then Main else Decr
  )
  have lsll[simp]: length sll = length ndl – Suc 0 unfolding sll-def by simp
  have sll:
    ∧i. i < length ndl – Suc 0 ⇒
      RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) sl ⇒ sll!i = sl
    ∧i. i < length ndl – Suc 0 ⇒
      RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) Main ⇒
        ¬RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) Decr
        ⇒ sll!i = Main
    ∧i. i < length ndl – Suc 0 ⇒
      ¬RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) Main ⇒
        RR (ndl!i,Pl!i) (ndl!(Suc i),Pl!(Suc i)) Decr
        ⇒ sll!i = Decr
  unfolding sll-def using slope.exhaust by auto

```

```

show ?thesis apply(rule exI[of - sll]) apply safe
  subgoal unfolding descentPathSl-def apply safe
  subgoal by fact
  subgoal by simp
  subgoal using d sll(2,3) unfolding descentPathParam-def
  by (metis (full-types) One-nat-def add.commute less-diff-conv plus-1-eq-Suc

```

slope.exhaust) .

subgoal using *d sll(1) unfolding descentPathParam-def MaxSl-def*
by (*smt Nitpick.size-list-simp(2) One-nat-def Suc-less-eq add-lessD1*
in-set-conv-nth length-greater-0-conv length-tl lsll plus-1-eq-Suc slope.exhaust)
.
qed

lemma *ex-descentPath-repeat-impl-tracksPers-transSl*:
assumes *ndl: cycleG ndl and tr: tracksPers K ndl and n: n ≠ 0 and*
Pl: wfLabL (repeat n (butlast ndl) @ [last ndl]) Pl
descentPathParam (repeat n (butlast ndl) @ [last ndl]) Pl sl
shows *transSl K (hd Pl) (last Pl) sl*
proof –
have *ndlNe[simp]: ndl ≠ [] and pndl[simp]: pathG ndl*
and *ndlhl[simp]: hd ndl = last ndl*
using *cycleG-def ndl not-path-Nil* **by** *auto*
have *tlndl[simp]: tl ndl ≠ []*
by (*metis hd-Cons-tl ndlNe not-path-singl pndl*)
show *?thesis using n Pl proof(induction n arbitrary: Pl sl)*
case (*0 Pl sl*)
then show *?case* **by** *auto*
next
case (*Suc n1 Pl sl*)
have *0: repeat (Suc n1) (butlast ndl) @ [last ndl] =*
(repeat n1 (butlast ndl) @ [last ndl]) @ tl ndl
unfolding *repeat-Suc2*
by (*metis append.assoc append-Cons append-Nil*
append-butlast-last-id list.exhaust-sel ndlNe ndlhl)

define *ndll where ndll ≡ repeat n1 (butlast ndl) @ [last ndl]*

have *ndll[simp]: ndll ≠ [] and lndll: last ndll = hd ndl*
using *ndll-def apply blast* **by** (*simp add: ndll-def*)

show *?case proof(cases n1=0)*
case *True*
hence *0: repeat (Suc n1) (butlast ndl) @ [last ndl] = ndl* **by** *simp*
hence *wfLabL ndl Pl descentPathParam ndl Pl sl* **using** *Suc* **by** *auto*
then obtain *sll where descentPathSl ndl Pl sll and sl: sl = MaxSl (set sll)*
using *wfLabL-descentPathParam-imp-descentPathSl* **by** *blast*
hence *K (hd Pl) (last Pl) sl* **using** *tr unfolding tracksPers-def* **by** *auto*
thus *?thesis* **by** (*intro transSl.Base*)
next
case *False*

have *[simp]: 2 ≤ length ndl 2 ≤ length ndll*
using *cycle-iff-nth ndl* **unfolding** *ndll-def* **using** *False* **by** *auto*

obtain *Pl1 sl1 Pl2 sl2 where Pl: Pl = Pl1 @ tl Pl2*

and $sl: sl = \text{MaxSl } \{sl1, sl2\}$ **and** $l: \text{last } Pl1 = \text{hd } Pl2$
and $Pl1: \text{wfLabL } ndll \text{ } Pl1 \text{ } \text{descentPathParam } ndll \text{ } Pl1 \text{ } sl1$
and $Pl2: \text{wfLabL } ndl \text{ } Pl2 \text{ } \text{descentPathParam } ndl \text{ } Pl2 \text{ } sl2$
using $\text{wfLabL-descentPathParam-append-inverse}[OF - - \text{ } lndll$
 $\text{ } Suc.\text{prems}(2,3)[\text{unfolded } 0 \text{ } ndll\text{-def}[\text{symmetric}]]]$ **by** *auto*
then obtain $sll2$ **where** $\text{descentPathSl } ndl \text{ } Pl2 \text{ } sll2$ **and** $sl2: sl2 = \text{MaxSl}$
 $(\text{set } sll2)$
using $\text{wfLabL-descentPathParam-imp-descentPathSl}$ **by** *blast*
hence $K2: K (\text{hd } Pl2) (\text{last } Pl2) \text{ } sl2$ **using** *tr* **unfolding** tracksPers-def **by**
auto

have $ne: Pl1 \neq [] \wedge tl \text{ } Pl2 \neq []$ **using** $Pl1(1) \text{ } Pl2(1) \text{ } \text{wfLabL-def}$
by $(\text{metis } \text{length-greater-0-conv } ndlNe \text{ } ndll \text{ } tlndl \text{ } \text{wfLabL-tl})$
have $h12: \text{hd } Pl1 = \text{hd } Pl \text{ } \text{last } Pl2 = \text{last } Pl$
unfolding Pl **by** $(\text{auto } \text{simp: } ne \text{ } \text{last-tl})$

from $Suc.IH[\text{unfolded } ndll\text{-def}[\text{symmetric}], OF \text{ } False \text{ } Pl1]$
have $K1: \text{transSl } K (\text{hd } Pl1) (\text{last } Pl1) \text{ } sl1$.
show $?thesis$ **using** $\text{transSl.Step}[OF \text{ } K1[\text{unfolded } l] \text{ } K2, \text{ } \text{unfolded } sl[\text{symmetric}]$
 $h12]$.
qed
qed
qed

lemma $\text{tracksPers-ex-descentPathSlS-impl-loopsDecr}$:
assumes $ndl: \text{cycleG } ndl$ **and** $tr: \text{tracksPers } K \text{ } ndl$
and $d: \text{descentIpathS } (\text{srepeat } (\text{butlast } ndl)) \text{ } Ps$
shows $\exists P. \text{transSl } K \text{ } P \text{ } P \text{ } \text{Decr}$
using $\text{cycle-descentIpathS-srepeat-imp-descentPath-repeat}[OF \text{ } ndl \text{ } d]$
using $\text{ex-descentPath-repeat-impl-tracksPers-transSl}[OF \text{ } ndl \text{ } tr, \text{ } \text{where } sl = \text{Decr}]$
unfolding $\text{descentPath-descentPathParam}[\text{symmetric}]$ **by** *metis*

lemma $\text{tracksPers-transSl-iff-ex-descentIpathS}$:
assumes $\text{cycleG } ndl \text{ } \text{tracksPers } K \text{ } ndl$
shows $(\exists P. \text{transSl } K \text{ } P \text{ } P \text{ } \text{Decr}) \longleftrightarrow (\exists Ps. \text{descentIpathS } (\text{srepeat } (\text{butlast } ndl)) \text{ } Ps)$
using $\text{assms } \text{Node-finite } \text{tracksPers-transSl-impl-ex-descentPathSlS } \text{tracksPers-ex-descentPathSlS-impl-loopsDecr}$

by *blast*

definition $\text{allOmegaCyclesDescendS} \equiv$
 $\forall ndl. \text{cycleG } ndl \longrightarrow (\exists Ps. \text{descentIpathS } (\text{srepeat } (\text{butlast } ndl)) \text{ } Ps)$

proposition *TransitiveLoopingCcl-iff-allOmegaCyclesDescendS*:
TransitiveLoopingCcl $\longleftrightarrow$ *allOmegaCyclesDescendS*
using *tracksPers-transSl-iff-ex-descentIpathS*
unfolding *TransitiveLoopingCcl-def allOmegaCyclesDescendS-def*
using *Ccl-nodes cycleG-def tracksPers-leftTotal Node-finite*
by (*metis* (*no-types*, *opaque-lifting*) *empty-iff tracksPers-rightTotal*)

lemma *InfiniteDescent-imp-InfiniteDescent*:
assumes *InfiniteDescent*
shows *allOmegaCyclesDescendS*
using *InfiniteDescent-def allOmegaCyclesDescendS-def assms cycle-srepeat-ipath*
srepeat-cycle-descentIpath-imp-descentIpath **by** *blast*

proposition *allOmegaCyclesDescendS-implies-InfiniteDescent*:
allOmegaCyclesDescendS $\longrightarrow$ *InfiniteDescent*
proof(*rule impI*, *rule ccontr*, *unfold VLA-Criterion*)
assume *omega*: *allOmegaCyclesDescendS* **and**
lang-inc: $\neg$ *NBA.language Paut_V* $\subseteq$ *NBA.language Taut_V*
then obtain *v u* **where** *lasso-in*: *v @- srepeat u* $\in$ *NBA.language Paut_V*
and *lasso-nin*: *v @- srepeat u* $\notin$ *NBA.language Taut_V*
and *u-ne*: *u* $\neq []$
apply-by(*drule prop1* '[*OF alpha-subseq-PTaut_V finite-Nodes-Paut_V finite-Nodes-Taut_V*]',
elim exE conjE)

also have *alw-Node*: *alw* (*holdsS Node*) (*v @- srepeat u*)
and *ipath*: *ipath* (*v @- srepeat u*)
using *lasso-in* **unfolding** *Paut_V-lang ipath-def* **by** *auto*

have *nn*: $2 \leq \text{length } v + \text{length } (u @ [hd \ u])$ **using** *u-ne* **by** (*cases u*, *auto*)

have *path-v*: *pathG* (*v @ u @ [hd u]*)
using *ipath-stake-path*[*OF ipath nn*]
unfolding *stake-add*[*symmetric*] *stake-len-append* *sdrop-shift-length'*
length-append-singleton *stake-Suc* *stake-cycle-eq*[*OF u-ne*]
srepeat-snth[*OF u-ne*] *hd-conv-nth*[*OF u-ne*] **by** *auto*

have *cycleG* (*u @ [hd u]*)
unfolding *cycleG-def* **using** *u-ne path-appendR*[*OF path-v*] **by** *auto*

then obtain *Ps* **where** *descentS*: *descentIpathS* (*srepeat u*) *Ps*
using *omega* **unfolding** *allOmegaCyclesDescendS-def* **by** *auto*
hence *descent*: *descentIpath* (*srepeat u*) *Ps* **using** *descentIpathS-imp-descentIpath*
by *auto*

then obtain *Ps'* **where** *Ps'*: *descentIpath* (*v @- srepeat u*) *Ps'*
using *descentIpath-sdrop-any*[*of length v*, *of v @- srepeat u Ps*]
unfolding *sdrop-shift-length'* **by** *auto*

thus *HOL.False* **using** *lasso-nin alw-Node* **unfolding** *Taut_V-lang* **by** *auto*

qed

corollary *Relation-Based-Criterion:*

InfiniteDescent $\longleftrightarrow$ *TransitiveLoopingCcl*

using *allOmegaCyclesDescendS*-*implies-InfiniteDescent* *InfiniteDescent-imp-InfiniteDescent*

TransitiveLoopingCcl-iff-allOmegaCyclesDescendS **by** *blast*

theorem *Relation-Based-Criterion':*

InfiniteDescent $\longleftrightarrow$ *TransitiveLooping*

using *TransitiveLooping-iff-TransitiveLoopingCcl* *Relation-Based-Criterion*

by *fastforce*

end

end

4 Incomplete Criteria for Infinite Descent

We next formalize some sufficient criteria for deciding Infinite Descent that are incomplete, but useful in practice. We adapt a known Sprenger-Dam (SD) criterion [5] to the general setting of sloped graphs, and then presents a novel theoretical contribution: an extension that strictly generalizes SD, which we call XSD.

4.1 Sprenger-Dam Criterion

theory *Incomplete-Criteria*

imports *../Sloped-Graphs*

begin

context *Sloped-Graph*

begin

definition *decreasingPCC* :: ('node $\Rightarrow$ 'node $\Rightarrow$ bool) $\Rightarrow$ ('node $\Rightarrow$ 'pos) $\Rightarrow$ bool

where

decreasingPCC *edge1* *lab* $\equiv$

$(\forall nd\ nd'.\ edge1\ nd\ nd' \longrightarrow RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Main \vee$
 $RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Decr) \wedge$
 $(\exists nd\ nd'.\ edge1\ nd\ nd' \wedge RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Decr)$

lemma *decreasingPCC-ipath-alw-holds2*:
assumes *lab*: *decreasingPCC edge1 lab* **and** *nds*: *Graph.ipath Node1 edge1 nds*
shows
 $alw\ (holds2\ (\lambda(nd, P)\ (nd', P').\ RR\ (nd, P)\ (nd', P')\ Main \vee RR\ (nd, P)\ (nd', P')\ Decr))$
 $(szip\ nds\ (smap\ lab\ nds))$
using *assms unfolding Graph.ipath-iff-snth decreasingPCC-def*
unfolding *alw-holds2-iff-snth* **by** *auto*

lemma *decreasingPCC-ipath-alw-ev-holds2*:
assumes *lab*: *decreasingPCC edge1 lab* **and** *nds*: *Graph.ipath Node1 edge1 nds* **and**
 $\forall nd\ nd'.\ edge1\ nd\ nd' \longrightarrow alw\ (ev\ (holds2\ (\lambda ndd\ ndd'.\ ndd = nd \wedge ndd' = nd')))$
 nds
shows
 $alw\ (ev\ (holds2\ (\lambda(nd, P)\ (nd', P').\ RR\ (nd, P)\ (nd', P')\ Decr)))$
 $(szip\ nds\ (smap\ lab\ nds))$
using *assms unfolding Graph.ipath-iff-snth decreasingPCC-def*
unfolding *alw-ev-holds2-iff-snth* **by** *fastforce*

lemma *decreasingPCC-imp-descentIpath*:
assumes *nds*: *ipath nds*
and *lim*: *decreasingPCC (limitR nds) lab*
shows *descentIpath nds (smap lab nds)*
proof–
obtain *i* **where** *0*: *Graph.ipath (limitS nds) (limitR nds) (sdrop i nds)*
using *ipath-sdrop-limit[OF Node-finite nds]* **by** *auto*
show *?thesis*
unfolding *descentIpath-def* **apply**(*intro sdrop-evI[where m = i]*)
unfolding *sdrop-szip sdrop-smap* **apply** *safe*
subgoal **using** *decreasingPCC-ipath-alw-holds2[OF lim 0]* .
subgoal **apply**(*rule decreasingPCC-ipath-alw-ev-holds2[OF lim 0]*)
apply *safe* **apply**(*rule alw-sdrop*) **unfolding** *limitR-def* . .
qed

definition *SDdescending* :: *bool* **where**
 $SDdescending \equiv \forall\ Node1\ edge1.\ scsg\ Node1\ edge1 \longrightarrow (\exists\ lab.\ wfLabF\ Node1\ lab \wedge decreasingPCC\ edge1\ lab)$

proposition *SDdescending-imp-InfiniteDescent*:
 $SDdescending \implies InfiniteDescent$
unfolding *SDdescending-def InfiniteDescent-def*
using *decreasingPCC-imp-descentIpath scsg-limit Node-finite* **by** *blast*

4.2 Extended Sprenger-Dam Criterion

definition *ExtG-Nodes* :: $('node \times 'pos)\ set$ **where**

$ExtG\text{-}Nodes \equiv \{(nd, P). nd \in Node \wedge P \in PosOf\ nd\}$

definition $ExtG\text{-}Edges :: ('node \times 'pos) \Rightarrow ('node \times 'pos) \Rightarrow bool$ **where**
 $ExtG\text{-}Edges \equiv \lambda(nd, P) (nd', P').$
 $edge\ nd\ nd' \wedge (RR\ (nd, P)\ (nd', P')\ Main \vee RR\ (nd, P)\ (nd', P')\ Decr)$

definition $is\text{-}slice ::$
 $'node\ set \Rightarrow ('node \Rightarrow 'node \Rightarrow bool) \Rightarrow$
 $('node \Rightarrow 'pos\ set) \Rightarrow ('node \Rightarrow 'node \Rightarrow 'pos \Rightarrow 'pos) \Rightarrow$
 $('node \times 'pos)\ set \Rightarrow (('node \times 'pos) \Rightarrow ('node \times 'pos) \Rightarrow bool) \Rightarrow bool$ **where**
 $is\text{-}slice\ Node1\ edge1\ lab\ f\ NNode\ eedge \equiv$
 $NNode \subseteq \{(nd, P). nd \in Node1 \wedge P \in lab\ nd\} \wedge$
 $Graph.subgr\ NNode\ eedge\ ExtG\text{-}Nodes\ ExtG\text{-}Edges \wedge$
 $(\forall nd\ P\ nd'\ P'. eedge\ (nd, P)\ (nd', P') \longrightarrow$
 $\{(nd, P), (nd', P')\} \subseteq NNode \wedge edge1\ nd\ nd' \wedge f\ nd\ nd'\ P = P') \wedge$
 $(\forall nd\ nd'. \{nd, nd'\} \subseteq Node1 \wedge edge1\ nd\ nd' \longrightarrow$
 $(\exists P\ P'. \{(nd, P), (nd', P')\} \subseteq NNode \wedge eedge\ (nd, P)\ (nd', P'))))$

definition $decreasing\text{-}slice :: ('node \times 'pos)\ set \Rightarrow (('node \times 'pos) \Rightarrow ('node \times 'pos) \Rightarrow bool) \Rightarrow bool$ **where**
 $decreasing\text{-}slice\ NNode\ eedge \equiv$
 $\exists nd\ P\ nd'\ P'. \{(nd, P), (nd', P')\} \subseteq NNode \wedge eedge\ (nd, P)\ (nd', P') \wedge RR$
 $(nd, P)\ (nd', P')\ Decr$

definition $descending\text{-}PCSC\text{-}sliced ::$
 $'node\ set \Rightarrow ('node \Rightarrow 'node \Rightarrow bool) \Rightarrow$
 $('node \Rightarrow 'pos\ set) \Rightarrow ('node \Rightarrow 'node \Rightarrow 'pos \Rightarrow 'pos) \Rightarrow bool$ **where**
 $descending\text{-}PCSC\text{-}sliced\ Node1\ edge1\ lab\ f \equiv$
 $RRSetChoice\ Node1\ edge1\ lab\ f \wedge$
 $(\forall NNode\ eedge.$
 $is\text{-}slice\ Node1\ edge1\ lab\ f\ NNode\ eedge \wedge$
 $Graph.scg\ NNode\ eedge$
 $\longrightarrow decreasing\text{-}slice\ NNode\ eedge)$

definition $XSDdescending :: bool$ **where**
 $XSDdescending \equiv$
 $\forall Node1\ edge1. scsg\ Node1\ edge1 \longrightarrow$

$(\exists \text{lab } f. \text{wfLabFS Node1 lab} \wedge \text{descending-PCSC-sliced Node1 edge1 lab } f)$

lemma *stake-sdrop-szip-nth*:

$k < m \implies \text{stake } m \text{ (sdrop } j \text{ (szip } A \ B))} \ ! \ k = (A \ !! \ (j + k), B \ !! \ (j + k))$
by (*metis sdrop-snth snth-szip stake-nth*)

lemma *set-stake-sdrop-szipD*:

$(x, y) \in \text{set} \text{ (stake } m \text{ (sdrop } j \text{ (szip } A \ B)))} \implies$
 $\exists k < m. x = A \ !! \ (j + k) \wedge y = B \ !! \ (j + k)$
by (*metis in-set-conv-nth length-stake prod.inject stake-sdrop-szip-nth*)

lemma *eq-stake-sdrop-szip-tuple*:

$k < m \implies (x, y) = \text{stake } m \text{ (sdrop } j \text{ (szip } A \ B))} \ ! \ k \implies x = A \ !! \ (j + k) \wedge y = B \ !! \ (j + k)$
by (*metis prod.inject stake-sdrop-szip-nth*)

lemma *descending-PCSC-sliced-imp-descentIpath*:

assumes *nds*: *ipath nds* **and** *lab*: *wfLabFS (limitS nds) lab*
and *lim*: *descending-PCSC-sliced (limitS nds) (limitR nds) lab f*
shows $\exists Ps. \text{descentIpath nds } Ps$
proof–

define *Node1 edge1* **where** *Sedge1-def*: $\text{Node1} \equiv \text{limitS nds}$ $\text{edge1} \equiv \text{limitR nds}$
obtain *n* **where** *a*: *alw (holdsS Node1 a and holds2 edge1) (sdrop n nds)*
using *ipath-ev-alw[OF Node-finite nds]* **unfolding** *ev-iff-sdrop Sedge1-def* **by** *auto*

define *nnds* **where** $\text{nnds} \equiv \text{sdrop } n \text{ nds}$
have *lndss*: $\text{limitR nds} = \text{limitR nnds}$ **unfolding** *nnds-def* **by** *auto*
have *nnds*: *ipath nnds* **by** (*simp add: Graph.ipath-sdrop nds nnds-def*)

have *nnds-Sedge1*: $\forall i. \text{nnds}!!i \in \text{Node1} \wedge \text{edge1} \text{ (nnds}!!i) \text{ (nnds}!!(\text{Suc } i))$
using *a* **unfolding** *nnds-def[symmetric]*
using *alw-holds2-iff-snth alw-holdsS-iff-snth alw-mono* **by** *blast*

obtain *P0* **where** *P0*: $P0 \in \text{lab} \text{ (shd nnds)}$
using *lab* **unfolding** *wfLabFS-def*
by (*metis Sedge1-def(1) equalsOI nnds-Sedge1 snth.simps(1)*)

define *Pi* **where** $Pi \equiv \text{rec-nat } P0 \ (\lambda i \ P. f \text{ (nnds}!!i) \text{ (nnds}!!(\text{Suc } i)) } P)$
define *Ps* **where** $Ps \equiv f\text{ToStream } Pi$

have *00*: $\bigwedge i. Pi \ i \in \text{lab} \text{ (nnds}!!i) \wedge$
 $\text{edge1} \text{ (nnds}!!i) \text{ (nnds}!!(\text{Suc } i)) \wedge$
 $Pi \text{ (Suc } i) = f \text{ (nnds}!!i) \text{ (nnds}!!(\text{Suc } i)) } (Pi \ i)$
subgoal for *i* **apply**(*induct i*) **apply** *simp-all*

subgoal using $P0$ **unfolding** $Pi\text{-def}$
by (*metis* (*no-types*, *lifting*) *nnds-Sedge1* *old.nat.simps*(6) *old.nat.simps*(7) *snth.simps*(1))
subgoal for i **unfolding** $Pi\text{-def}$ **apply** *simp*
by (*smt* (*verit*, *best*) *Graph.limitR-S* *descending-PCSC-sliced-def* *RRSetChoice-def* *Node-finite* *Sedge1-def*(2) *image-subset-iff* *ipath-sdrop-limit* *lim*
limitR-sdrop-eq *nds* *nnds-Sedge1*) . .

hence $0: \bigwedge i. Ps!!i \in lab\ (nnds!!i) \wedge Ps!!(Suc\ i) = f\ (nnds\ !!\ i)\ (nnds\ !!\ Suc\ i)$
 $(Ps\ !!\ i)$
by (*simp* *add: Ps-def*)

define φ **where** $\varphi \equiv \lambda i\ P\ P'. P' \in lab(nnds!!(Suc\ i)) \wedge$
 $(RR\ (nnds!!i, P)\ (nnds!!(Suc\ i), P')\ Main \vee$
 $RR\ (nnds!!i, P)\ (nnds!!(Suc\ i), P')\ Decr)$

have $2: \forall i. \varphi\ i\ (Pi\ i)\ (Pi\ (Suc\ i))$
using 0 **unfolding** $\varphi\text{-def}$
by (*smt* (*verit*, *best*) *00* *descending-PCSC-sliced-def* *RRSetChoice-def* *Sedge1-def*(1) *Sedge1-def*(2) *empty-subsetI* *insert-subset* *lim* *nnds-Sedge1*)

have $Node1: finite\ Node1\ Node1 \subseteq Node\ wfLabFS\ Node1\ lab$
unfolding *Sedge1-def*(1) **using** *Node-finite* *infinite-super* *limitS-S*
apply *blast* **by** (*auto* *simp: Sedge1-def*(1) *limitS-S* *lab*)

define *StateSpace* **where** $StateSpace \equiv \{(nd, P). nd \in Node1 \wedge P \in lab\ nd\}$
have *eNode1: finite StateSpace*
unfolding *StateSpace-def*
proof –
have $\{(nd, P). nd \in Node1 \wedge P \in lab\ nd\} = Sigma\ Node1\ lab$ **by** *auto*
thus *finite* $\{(nd, P). nd \in Node1 \wedge P \in lab\ nd\}$
using *wfLabFS-finite*[*OF* *Node1*(3) *Node1*(2)] *Node1*(1) *finite-SigmaI* **by** *auto*
qed

have *ipath-states: $\forall i. (nnds!!i, Ps!!i) \in StateSpace$*
unfolding *StateSpace-def* **using** 0 *nnds-Sedge1* **by** *auto*

obtain $nd\ P$ **where** $nd\text{-}P: (nd, P) \in StateSpace\ \forall i. \exists j \geq i. nnds!!j = nd \wedge Ps!!j = P$
proof –
have $range\ (\lambda i. (nnds!!i, Ps!!i)) \subseteq StateSpace$ **using** *ipath-states* **by** *auto*
hence *fin-range: finite* ($range\ (\lambda i. (nnds!!i, Ps!!i))$)
using *eNode1* *finite-subset* **by** *auto*

```

obtain  $nd-P$  where  $infinite \{i. (nnds!!i, Ps!!i) = nd-P\}$ 
  using  $pigeonhole-infinite[OF ] fin-range$  by  $fastforce$ 
moreover obtain  $nd P$  where  $nd-P = (nd, P)$  by  $fastforce$ 
ultimately have  $inf: infinite \{i. (nnds!!i, Ps!!i) = (nd, P)\}$  by  $simp$ 

have  $\forall i. \exists j \geq i. nnds!!j = nd \wedge Ps!!j = P$ 
  using  $inf$  unfolding  $infinite-nat-iff-unbounded$ 
  by  $(meson mem-Collect-eq order-less-imp-le prod.inject)$ 
moreover have  $(nd, P) \in StateSpace$ 
  using  $inf ipath-states not-finite-existsD$  by  $blast$ 
ultimately show  $?thesis$  using  $that$  by  $blast$ 
qed

have  $d-nnds: descentIpath nnds Ps$ 
unfolding  $descentIpath-iff-snth2$  apply  $(intro conjI)$ 
  subgoal using  $2$  unfolding  $\varphi-def$  by  $(simp add: Ps-def)$ 
  subgoal proof  $safe$ 
    fix  $i$ 

    obtain  $j0$  where  $j0: j0 \geq i \ nnds!!j0 = nd \wedge Ps!!j0 = P$  using  $nd-P$  by  $auto$ 

    obtain  $j1 j2$  where  $j12: j1 \geq Suc\ j0 \ j2 \geq j1$ 
       $\wedge nd\ nd'. limitR\ nnds\ nd\ nd' \implies (\exists j \geq j1. j < j2 \wedge nnds!!j = nd \wedge nnds!!$ 
 $Suc\ j = nd')$ 
      using  $ipath-limitR-interval[OF Node-finite nnds]$  by  $blast$ 

    obtain  $j3$  where  $j3: j3 \geq Suc\ j2 \ nnds!!j3 = nd \wedge Ps!!j3 = P$  using  $nd-P$  by
 $auto$ 

    define  $nd-Pl$  where  $nd-Pl \equiv stake\ (j3-j0+1)\ (sdrop\ j0\ (szip\ nnds\ Ps))$ 

    have  $cyc: Graph.cycleG\ StateSpace\ (\lambda-. True)\ nd-Pl$ 
    unfolding  $nd-Pl-def$  apply  $(rule\ Graph.ipath-stake-sdrop-cycle)$ 
      subgoal by  $(simp\ add: Graph.ipath-iff-snth\ ipath-states)$ 
      subgoal using  $j12(1)\ j12(2)\ j3(1)$  by  $linarith$ 
      subgoal by  $simp\ (metis\ add-diff-cancel-left'\ add-leE\ j0(2)\ j12(1)\ j12(2)\$ 
 $j3(1)\ j3(2)\$ 
 $nat-le-iff-add\ plus-1-eq-Suc)\ .$ 

    define  $NNode$  where  $NNode \equiv set\ (nd-Pl)$ 
    define  $eedge$  where  $eedge \equiv \lambda\ nd-P\ nd-P'.$ 
 $(\exists k. Suc\ k < length\ nd-Pl \wedge nd-P = nd-Pl!!k \wedge nd-P' = nd-Pl!(Suc\ k))$ 

    have  $subgr: Graph.subgr\ NNode\ eedge\ ExtG-Nodes\ ExtG-Edges$ 
    unfolding  $Graph.subgr-def\ NNode-def\ eedge-def\ ExtG-Nodes-def\ ExtG-Edges-def$ 

```

```

apply safe
subgoal for nd P
  unfolding nd-Pl-def
  apply (drule set-stake-sdrop-szipD)
  using ipath-iff-snth nnds StateSpace-def by fastforce
subgoal for nd P
  unfolding nd-Pl-def
  apply (drule set-stake-sdrop-szipD)
  using ipath-states Node1(3) unfolding StateSpace-def wfLabFS-def
  by auto
subgoal for nd P nd' P' k
  apply(rule ssubst[of nd nnds !! (j0 + k)])
apply (metis Suc-lessD length-stake nd-Pl-def prod.inject stake-sdrop-szip-nth)
  apply(rule ssubst[of nd' nnds !! Suc (j0 + k)])
  apply (unfold nd-Pl-def, metis prod.inject stake-sdrop-szip-nth nd-Pl-def
length-stake add-Suc-right)
  using ipath-iff-snth nnds by blast
subgoal for nd P nd' P' k
  apply(rule ssubst[of (nd,P) (nnds !! (j0 + k), Ps !! (j0 + k))])
  apply(unfold nd-Pl-def, metis Suc-lessD length-stake sdrop-snth snth-szip
stake-nth)
  apply(rule ssubst[of (nd', P') (nnds !! Suc (j0 + k), Ps !! Suc (j0 + k))])
  apply(metis add-Suc-right length-stake sdrop-snth snth-szip stake-nth)
  using 2[THEN spec, of j0 + k]
  unfolding nd-Pl-def  $\varphi$ -def Ps-def
  by (metis Suc-lessD add-Suc-right length-stake snth-fToStream
stake-sdrop-szip-nth) .

have proj:  $\forall x P y P'. \text{edge } (x, P) (y, P') \longrightarrow \{(x, P), (y, P')\} \subseteq \text{NNode} \wedge$ 
edge1 x y  $\wedge f x y P = P'$ 
  unfolding edge-def apply clarify
subgoal for x P y P' k
  using nnds-Sedge1[THEN spec, of j0 + k] 0[of j0 + k]
  unfolding nd-Pl-def
  by (metis Suc-lessD add-Suc-right eq-stake-sdrop-szip-tuple
length-stake NNode-def Suc-lessD empty-subsetI in-set-conv-nth in-
sert-subset nd-Pl-def) .

have cover:  $\forall x y. \{x, y\} \subseteq \text{Node1} \wedge \text{edge1 } x y \longrightarrow$ 
 $(\exists P P'. \{(x,P),(y,P')\} \subseteq \text{NNode} \wedge \text{eedge } (x,P) (y,P'))$ 
unfolding NNode-def eedge-def Sedge1-def lndss
proof clarify
  fix x y
  assume nodes:  $\{x, y\} \subseteq \text{limitS nds}$ 
  assume edge: limitR nnds x y

from j12(3)[OF edge] obtain j where j-bound:  $j1 \leq j < j2$ 
and x-eq:  $x = \text{nnds} !! j$  and y-eq:  $y = \text{nnds} !! \text{Suc } j$  by blast

```

```

define  $k$  where  $k \equiv j - j0$ 

have  $j0-j$ :  $j0 \leq j$  using  $j12(1)$   $j$ -bound(1) by linarith
have  $k$ -len1:  $k < j3 - j0 + 1$  and  $k$ -len2:  $Suc\ k < j3 - j0 + 1$ 
  unfolding  $k$ -def using  $j12(1)$   $j$ -bound(2)  $j3(1)$   $j0-j$  by linarith+
have  $len$ -Pl:  $length\ nd\text{-}Pl = j3 - j0 + 1$  unfolding  $nd$ -Pl-def by simp

have  $nth$ -k:  $nd\text{-}Pl ! k = (x, Ps !! j)$ 
  unfolding  $nd$ -Pl-def  $k$ -def using  $j0-j$   $k$ -len1
  by (metis Nat.add-diff-assoc add-diff-cancel-left'  $k$ -def
    stake-sdrop-szip-nth  $x$ -eq)
have  $nth$ -Suc-k:  $nd\text{-}Pl ! Suc\ k = (y, Ps !! Suc\ j)$ 
  unfolding  $nd$ -Pl-def  $k$ -def using  $j0-j$   $k$ -len2
  by (metis Nat.add-diff-assoc add-Suc-right add-diff-cancel-left'
     $k$ -def stake-sdrop-szip-nth  $y$ -eq)

show  $\exists P\ P'. \{(x, P), (y, P')\} \subseteq set\ nd\text{-}Pl \wedge$ 
  ( $\exists i. Suc\ i < length\ nd\text{-}Pl \wedge (x, P) = nd\text{-}Pl ! i \wedge (y, P') = nd\text{-}Pl$ 
  !  $Suc\ i$ )
  proof (intro exI2[of -  $Ps !! j$   $Ps !! Suc\ j$ ] conjI)
    show  $\{(x, Ps !! j), (y, Ps !! Suc\ j)\} \subseteq set\ nd\text{-}Pl$ 
    using  $nth$ -k  $nth$ -Suc-k  $k$ -len1  $k$ -len2  $len$ -Pl  $nth$ -mem by (metis empty-subsetI
      insert-subset)
    show  $\exists i. Suc\ i < length\ nd\text{-}Pl \wedge (x, Ps !! j) = nd\text{-}Pl ! i \wedge (y, Ps !! Suc\ j) = nd\text{-}Pl ! Suc\ i$ 
    using  $k$ -len2  $nth$ -k  $nth$ -Suc-k  $len$ -Pl by (metis exI[of -  $k$ ])
  qed
qed

have  $nnode$ -bound:  $NNode \subseteq \{(nd, P). nd \in Node1 \wedge P \in lab\ nd\}$ 
  unfolding  $NNode$ -def  $StateSpace$ -def[symmetric]
proof
  fix  $x$  assume  $x \in set\ nd\text{-}Pl$ 
  then obtain  $k$  where  $x = nd\text{-}Pl ! k\ k < length\ nd\text{-}Pl$ 
  by (metis  $\langle x \in set\ nd\text{-}Pl \rangle$  in-set-conv-nth)
  thus  $x \in StateSpace$ 
  unfolding  $nd$ -Pl-def using ipath-states
  by (metis length-stake sdrop-snth snth-szip stake-nth)
qed

have  $is$ -slice:  $is$ -slice  $Node1$   $edge1$   $lab\ f\ NNode$   $eedge$ 
unfolding  $is$ -slice-def using subgr proj cover  $nnode$ -bound by blast

have  $scg$ :  $Graph.scg\ NNode\ eedge$  apply(subst  $Graph.scg$ -iff-cycle)
  subgoal unfolding  $NNode$ -def by auto

```

```

    subgoal unfolding NNode-def
  by simp (metis Graph.cycle-iff-nth cyc less-nat-zero-code list.size(3) not-path-Nil
    path-iff-nth)
    subgoal apply(rule exI[of - nd-Pl], standard)
      subgoal using cyc unfolding Graph.cycleG-def NNode-def eedge-def
        unfolding Graph.path-iff-set-nth by auto
      subgoal unfolding NNode-def .. .

  with is-slice lim obtain nd-d P-d nd-d' P-d' where
    decr-edge: eedge (nd-d, P-d) (nd-d', P-d') RR (nd-d, P-d) (nd-d', P-d') Decr
  unfolding descending-PCSC-sliced-def decreasing-slice-def
  by (metis scg Sedge1-def(2) Sedge1-def(1))

  then obtain k where k: k < length nd-Pl - 1 RR (nd-Pl ! k) (nd-Pl ! Suc k)
Decr
  by (metis Suc-lessE diff-Suc-1 eedge-def)

  show  $\exists j \geq i. RR (nnds !! j, Ps !! j) (nnds !! Suc j, Ps !! Suc j) Decr$ 
  apply(rule exI[of - j0+k], safe)
  subgoal by (simp add: j0(1) trans-le-add1)
  subgoal using k unfolding nd-Pl-def sdrop-snth
    by (metis Suc-eq-plus1 Suc-mono add-Suc-right add-diff-cancel-right'
      length-stake less-SucI sdrop-snth snth-szip stake-nth) .
  qed .

  show ?thesis using d-nnds by (simp add: descentIpath-sdrop-any nnds-def)
  qed

proposition XSDdescending-implies-InfiniteDescent:
  XSDdescending  $\implies$  InfiniteDescent
  unfolding XSDdescending-def InfiniteDescent-def
  using descending-PCSC-sliced-imp-descentIpath scsg-limit Node-finite by blast

lemmas Incomplete-Criterion = SDdescending-imp-InfiniteDescent
  XSDdescending-implies-InfiniteDescent

theorem SDdescending-implies-XSDdescending:
  SDdescending  $\implies$  XSDdescending
  unfolding SDdescending-def XSDdescending-def
proof clarify
  fix Node1 edge1
  assume SD:  $\forall Node1\ edge1. scsg\ Node1\ edge1 \longrightarrow (\exists lab. wfLabF\ Node1\ lab \wedge$ 
    decreasingPCC edge1 lab)
  assume scsg: scsg Node1 edge1

  define edge1' where edge1'  $\equiv \lambda x\ y. edge1\ x\ y \wedge x \in Node1 \wedge y \in Node1$ 

```

```

have scsg': scsg Node1 edge1'
  using scsg Graph-scg-restrict unfolding scsg-def edge1'-def Graph.subgr-def
  by auto

from SD scsg' obtain lab where wf: wfLabF Node1 lab and pcc: decreasingPCC
edge1' lab
  by blast

define lab' where lab' ≡ λnd. {lab nd}
define f where f ≡ λ(nd::'node) nd' (P::'pos). lab nd'

show  $\exists lab' f. wfLabFS\ Node1\ lab' \wedge descending-PCSC-sliced\ Node1\ edge1\ lab' f$ 
proof (intro exI conjI)
  show wfLabFS Node1 lab'
    using wf unfolding wfLabF-def wfLabFS-def lab'-def by blast

  show descending-PCSC-sliced Node1 edge1 lab' f
    unfolding descending-PCSC-sliced-def
  proof (intro conjI allI impI)

    show RRSetChoice Node1 edge1 lab' f
      using pcc unfolding decreasingPCC-def RRSetChoice-def lab'-def f-def
      edge1'-def by auto

  fix NNode eedge
  assume is-slice Node1 edge1 lab' f NNode eedge  $\wedge$  Graph.scg NNode eedge
  hence is-slice: is-slice Node1 edge1 lab' f NNode eedge
    and scg: Graph.scg NNode eedge by auto

  from pcc obtain nd-d nd-d' where decr: edge1' nd-d nd-d' RR (nd-d, lab
nd-d) (nd-d', lab nd-d') Decr
    unfolding decreasingPCC-def by blast

  have nodes-in: {nd-d, nd-d'} ⊆ Node1 and orig-edge: edge1 nd-d nd-d'
    using decr(1) unfolding edge1'-def by auto

  have cover: ∀ x y. {x, y} ⊆ Node1 ∧ edge1 x y ⟶
     $(\exists P P'. \{(x, P), (y, P')\} \subseteq NNode \wedge eedge (x, P) (y, P'))$ 
    using is-slice unfolding is-slice-def by blast

  obtain P P' where in-slice: {(nd-d, P), (nd-d', P')} ⊆ NNode eedge (nd-d,
P) (nd-d', P')
    using cover nodes-in orig-edge by blast

```



```

have nnode-bound:  $NNode \subseteq \{(nd, P). nd \in Node1 \wedge P \in lab' nd\}$ 
  using is-slice unfolding is-slice-def by auto

have  $P = lab\ nd-d$  and  $P' = lab\ nd-d'$ 
  using in-slice(1) nnode-bound unfolding lab'-def by auto

show decreasing-slice NNode eedge
  unfolding decreasing-slice-def
  using in-slice decr(2)  $\langle P = lab\ nd-d \rangle \langle P' = lab\ nd-d' \rangle$  by blast
qed
qed
qed

end

end

```

4.3 Sprenger-Dam Criterion Incompleteness

```

theory SD-Incomplete
imports ../Incomplete-Criteria
begin

```

```

datatype node = One
datatype pos = h1 | h1'

```

```

definition Node  $\equiv \{One\}$ 

```

```

lemma O-Node[simp]:  $One \in Node$  by (simp add: Node-def)

```

```

lemma alw-nodes: alw (holdsS Node) nds
  unfolding alw-holdsS-iff-snth Node-def apply (rule allI)
  subgoal for i apply (induct i)
  using node.exhaust by auto .

```

```

fun edge::node  $\Rightarrow$  node  $\Rightarrow$  bool where
  edge One One = HOL.True

```

```

lemma edge-into-zero: edge nd nd'  $\longleftrightarrow nd = One \wedge nd' = One$  by (cases nd, cases nd', auto)

```

lemma *edgeTrue*: *edge nd nd'* **by**(*cases nd, cases nd', auto*)

fun *PosOf*::*node* $\Rightarrow$ *pos set* **where**
PosOf One = {*h1, h1'*}

definition *RR-set* :: ((*node* $\times$ *pos*) $\times$ (*node* $\times$ *pos*) $\times$ *slope*) *set* **where**
RR-set = {
 ((*One, h1*), (*One, h1'*), *Decr*),

 ((*One, h1'*), (*One, h1*), *Main*)
}

definition *RR* :: *node* $\times$ *pos* $\Rightarrow$ *node* $\times$ *pos* $\Rightarrow$ *slope* $\Rightarrow$ *bool* **where**
RR np1 np2 s $\equiv$ ((*np1, np2, s*) $\in$ *RR-set*)

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *RR-ZO[simp]*: *RR (One, h1) (One, h1') Decr* **unfolding** *RR-defs* **by** *auto*
lemma *RR-OZ[simp]*: *RR (One, h1') (One, h1) Main* **unfolding** *RR-defs* **by** *auto*

lemma *P-inPosOf*: *RR (nd, P) (nd', P') sl* $\Longrightarrow$ *P* $\in$ *PosOf nd*
RR (nd, P) (nd', P') sl $\Longrightarrow$ *P'* $\in$ *PosOf nd'* **by**(*auto simp: RR-defs*)

interpretation *Sloped-Graph* **where**
Node = *Node* **and** *edge* = *edge* **and** *PosOf* = *PosOf*
and *RR* = *RR* **apply** *standard*
subgoal **by**(*simp add: Node-def*)
subgoal **by**(*unfold Node-def, auto elim: PosOf.cases*)
by(*auto simp: RR-defs SlopedRels-def Node-def*)

lemma *allNodesOne*: $\forall i. nds \text{ !! } i = One$ **using** *alw-nodes[unfolded Node-def, of nds]* *PosOf.cases* **by** *auto*

lemma *ipath-isOne*: *ipath nds* $\Longrightarrow$ *nds* = *sconst One*
using *allNodesOne* **unfolding** *ipath-iff-snth*
by (*simp add: snth-equalityI*)

lemma *j-mod2-cases*: ((*j mod Suc (Suc 0)*) = 0 $\wedge$ (*Suc j mod Suc (Suc 0)*) = 1)
 $\vee$ ((*j mod Suc (Suc 0)*) = 1 $\wedge$ (*Suc j mod Suc (Suc 0)*) = 0)
by(*induct j, simp, metis One-nat-def Suc-1 mod2-Suc-Suc*)

lemma *j-mod2-cases'*: (*j mod Suc (Suc 0)*) = 0 $\vee$ (*j mod Suc (Suc 0)*) = 1
apply(*induct j, simp*)
using *j-mod2-cases* **by** *blast*

lemma *j-mod2-suc*: $j \bmod \text{Suc } (\text{Suc } 0) = 0 \implies \text{Suc } j \bmod \text{Suc } (\text{Suc } 0) = 1$ **by**
(metis j-mod2-cases)

lemma *descentPathEx*: *descentIpath* (*sconst One*) (*srepeat [h1, h1']*)
unfolding *descentIpath-iff-snth*
apply(*rule exI[of - 0], clarsimp*)
apply(*intro conjI allI*)
subgoal for j by(*rule disjE[OF j-mod2-cases[of j]], simp-all*)
subgoal for j
apply(*rule disjE[OF j-mod2-cases[of j]]*)
subgoal by(*rule exI[of - j], simp*)
subgoal by(*rule exI[of - Suc j], simp add: j-mod2-suc*) . .

lemma *pathConEx*: *Graph.pathCon* {*One*} ($\lambda u v. \text{True}$) *One One*
unfolding *Graph.pathCon-def*
using *Graph.pathG.Base[of One One {One} ($\lambda u v. \text{True}$)]* **by** *auto*

lemma *scsgEx*: *scsg* {*One*} ($\lambda u v. \text{True}$)
unfolding *scsg-def* **apply** *safe*
subgoal unfolding *subgr-def* **by** (*auto simp: edgeTrue*)
subgoal unfolding *Graph.scg-def* **using** *pathConEx* **by** *auto* .

proposition *InfiniteDescent*
apply(*rule InfiniteDescentI*)
apply(*drule ipath-isOne*)
apply(*rule exI[of - srepeat [h1, h1']]*)
using *descentPathEx* **by** *auto*

proposition $\neg \text{SDdescending}$
unfolding *SDdescending-def* **apply** *safe*
apply(*erule allE[of - {One}]*)
apply(*elim allE[of - $\lambda u v. \text{True}$] impE, simp add: scsgEx*)
apply(*elim exE conjE*)
unfolding *decreasingPCC-def wfLabF-def* **apply** *clarify*
subgoal for lab nd nd' **apply**(*cases nd, cases nd', simp*)
by(*elim allE[of - One], simp add: RR-defs*) .

end

4.4 Extended Sprenger-Dam Criterion Incompleteness

theory *XSD-Incomplete*
imports *../Incomplete-Criteria*
begin

```

datatype node = One | Two | Three
datatype pos = p1 | p1' | p2 | p3

```

```

definition Node  $\equiv$  {One, Two, Three}

```

```

lemma nd-notOne:  $nd \neq One \longleftrightarrow nd = Two \vee nd = Three$  by(cases nd, auto)

```

```

lemma O-Node[simp]:  $One \in Node$  by(simp add: Node-def)
lemma T-Node[simp]:  $Two \in Node$  by(simp add: Node-def)
lemma Tr-Node[simp]:  $Three \in Node$  by(simp add: Node-def)

```

```

lemma alw-nodes: alw (holdsS Node) nds
  unfolding alw-holdsS-iff-snth Node-def apply(rule allI)
  subgoal for i apply(induct i)
  using node.exhaust by auto .

```

```

fun edge::node  $\Rightarrow$  node  $\Rightarrow$  bool where
  edge One Two = HOL.True|
  edge Two One = HOL.True|
  edge One Three = HOL.True|
  edge Three One = HOL.True|
  edge - - = HOL.False

```

```

fun PosOf::node  $\Rightarrow$  pos set where
  PosOf One = {p1, p1'}|
  PosOf Two = {p2}|
  PosOf Three = {p3}

```

```

definition RR-set :: ((node  $\times$  pos)  $\times$  (node  $\times$  pos)  $\times$  slope) set where
  RR-set = {
    ((One, p1), (Two, p2), Main),
    ((Two, p2), (One, p1), Decr),
    ((Two, p2), (One, p1'), Decr),
    ((One, p1'), (Three, p3), Main),
    ((Three, p3), (One, p1'), Decr),
    ((Three, p3), (One, p1), Decr)
  }

```

definition *EE-set* :: $((node \times pos) \times (node \times pos))$ set **where**

```
EE-set = {
  ((One, p1), (Two, p2)),
  ((Two, p2), (One, p1)),
  ((Two, p2), (One, p1')),
  ((One, p1'), (Three, p3)),
  ((Three, p3), (One, p1')),
  ((Three, p3), (One, p1))
}
```

definition *EE* :: $node \times pos \Rightarrow node \times pos \Rightarrow bool$ **where**

EE np1 np2 $\equiv ((np1, np2) \in EE\text{-set})$

lemmas *EE-defs* = *EE-def EE-set-def*

definition *RR* :: $node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool$ **where**

RR np1 np2 s $\equiv ((np1, np2, s) \in RR\text{-set})$

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *RR-OT[simp]:RR* (One, p1) (Two, p2) *Main unfolding RR-defs by auto*

lemma *RR-TO[simp]:RR* (Two, p2) (One, p1) *Decr unfolding RR-defs by auto*

lemma *RR-TO'[simp]:RR* (Two, p2) (One, p1') *Decr unfolding RR-defs by auto*

lemma *RR-OTr[simp]:RR* (One, p1') (Three, p3) *Main unfolding RR-defs by auto*

lemma *RR-TrO[simp]:RR* (Three, p3) (One, p1') *Decr unfolding RR-defs by auto*

lemma *RR-TrO'[simp]:RR* (Three, p3) (One, p1) *Decr unfolding RR-defs by auto*

lemma *P-inPosOf:RR* (nd, P) (nd', P') *sl* $\implies P \in PosOf\ nd$

RR (nd, P) (nd', P') *sl* $\implies P' \in PosOf\ nd'$ **by**(*auto simp: RR-defs*)

interpretation *Sloped-Graph* **where**

Node = *Node* **and** *edge* = *edge* **and** *PosOf* = *PosOf*

and *RR* = *RR* **apply** *standard*

subgoal **by**(*simp add: Node-def*)

subgoal **by**(*unfold Node-def, auto elim: PosOf.cases*)

by(*auto simp: RR-defs SlopedRels-def Node-def*)

definition *ipath12* :: $node\ stream \Rightarrow bool$ **where**

ipath12 s $\equiv ev\ (alw\ (holds\ (\lambda n. n = One \vee n = Two)))\ s$

definition *ipath13* :: $node\ stream \Rightarrow bool$ **where**

$ipath13\ s \equiv ev\ (alw\ (holds\ (\lambda n. n = One \vee n = Three)))\ s$

definition $ipath-mixed :: node\ stream \Rightarrow bool$ **where**

$ipath-mixed\ s \equiv alw\ (ev\ (holds\ (\lambda n. n = Two)))\ s \wedge alw\ (ev\ (holds\ (\lambda n. n = Three)))\ s$

lemmas $ipaths = ipath12-def\ ipath13-def\ ipath-mixed-def$

lemma $ipath-cases: ipath\ nds \Longrightarrow ipath12\ nds \vee ipath13\ nds \vee ipath-mixed\ nds$

unfolding $ipaths$

apply $(cases\ ev\ (alw\ (holds\ (\lambda n. n \neq Three))))\ nds)$

apply $(cases\ ev\ (alw\ (holds\ (\lambda n. n \neq Two))))\ nds)$

subgoal unfolding $ev-alw-holds-iff-snth\ alw-ev-holds-iff-snth$ **by** $(meson\ PosOf.cases)$

subgoal unfolding $ev-alw-holds-iff-snth\ alw-ev-holds-iff-snth$ **by** $(meson\ PosOf.cases)$

subgoal unfolding $ev-alw-holds-iff-snth\ alw-ev-holds-iff-snth$ **by** $(meson\ PosOf.cases)$

.

lemma $ipath-dist21:$

assumes $ipath\ nds$

shows $nds\ !!\ i = Two \Longrightarrow nds\ !!\ Suc\ i = One$

using $assms$ **unfolding** $ipath-def$

apply $(induct\ i)$

apply $(unfold\ alw-holds2-iff-snth)$

by $(metis\ edge.elims(2)\ node.distinct(1))+$

lemma $ipath-dist31:$

assumes $ipath\ nds$

shows $nds\ !!\ i = Three \Longrightarrow nds\ !!\ Suc\ i = One$

using $assms$ **unfolding** $ipath-def$

apply $(induct\ i)$

apply $(unfold\ alw-holds2-iff-snth)$

by $(metis\ edge.elims(2)\ edge.simps(7))+$

lemma $ipath-dist1:$

assumes $ipath\ nds$

shows $nds\ !!\ i = One \Longrightarrow nds\ !!\ Suc\ i = Two \vee nds\ !!\ Suc\ i = Three$

using $assms$ **unfolding** $ipath-def$

apply $(induct\ i)$

apply $(unfold\ alw-holds2-iff-snth)$

by $(metis\ edge.elims(2)\ edge.simps(7))+$

lemma $ipath12-descent:$

```

assumes ipath nds
shows ipath12 nds  $\implies \exists Ps. \text{descentIpath } nds \ Ps$ 
unfolding ipaths ev-alw-holds-iff-snth
proof(erule exE)
  fix i
  assume assm:  $\forall j \geq i. nds !! j = One \vee nds !! j = Two$ 

  let ?Ps = smap ( $\lambda n. \text{if } n = One \text{ then } p1 \text{ else } p2$ ) nds

  show ?thesis
    apply (unfold descentIpath-def, intro exI[of - ?Ps] sdrop-evI[of - i] conjI)

    subgoal unfolding alw-holds2-iff-snth
      apply (clarsimp split: if-splits prod.splits, intro conjI impI)
      subgoal for i' using ipath-dist1[OF assms, of i' + i] by (simp add:
add commute sdrop-snth)
      subgoal for i' using ipath-dist1[OF assms, of i' + i]
        unfolding RR-defs
        by (auto simp add: add commute sdrop-snth,metis assm le-Suc-eq le-add2
node.distinct(3,5))
      subgoal for i' unfolding nd-notOne RR-defs
        using ipath-dist21[OF assms, of i' + i]
        ipath-dist31[OF assms, of i' + i]
      by (auto simp add: add commute sdrop-snth,metis assm le-add2 node.distinct(3,5))
      subgoal for i' unfolding nd-notOne RR-defs
        using ipath-dist21[OF assms, of i' + i]
        ipath-dist31[OF assms, of i' + i]
      by (auto simp add: add commute sdrop-snth) .

    unfolding alw-ev-holds2-iff-snth
    apply(rule allI, simp)
    subgoal for i'
      apply(cases sdrop i nds !! i')
      subgoal apply(rule exI[of - Suc i'])
        using ipath-dist1[OF assms, of i' + i]
        ipath-dist21[OF assms, of Suc (i' + i)]
        ipath-dist31[OF assms, of Suc (i' + i)]
        by (auto simp add: add commute sdrop-snth,metis assm less-add-Suc2
nat-less-le node.distinct(3,5))
      subgoal apply(rule exI[of - i'])
        using ipath-dist21[OF assms, of i' + i]
        by (simp add: add commute sdrop-snth)
      subgoal using assm by (metis le-add1 node.distinct(3,5) sdrop-snth) . .
  qed

lemma ipath13-descent:
  assumes ipath nds
  shows ipath13 nds  $\implies \exists Ps. \text{descentIpath } nds \ Ps$ 

```

```

unfolding ipaths ev-alw-holds-iff-snth
proof(erule exE)
  fix i
  assume assm: $\forall j \geq i. \text{nds} !! j = \text{One} \vee \text{nds} !! j = \text{Three}$ 

  let ?Ps = smap ( $\lambda n. \text{if } n = \text{One} \text{ then } p1' \text{ else } p3$ ) nds

  show ?thesis
    apply (unfold descentIpath-def, intro exI[of - ?Ps] sdrop-evI[of - i] conjI)

    subgoal unfolding alw-holds2-iff-snth
      apply (clarsimp split: if-splits prod.splits, intro conjI impI)
      subgoal for i' using ipath-dist1[OF assms, of i' + i] by (simp add:
add commute sdrop-snth)
      subgoal for i' using ipath-dist1[OF assms, of i' + i]
      unfolding RR-defs
      by (auto simp add: add commute sdrop-snth, metis assm less-add-Suc2
less-or-eq-imp-le node.distinct(1,5))
      subgoal for i' unfolding nd-notOne RR-defs
      using ipath-dist21[OF assms, of i' + i]
      ipath-dist31[OF assms, of i' + i]
      by (auto simp add: add commute sdrop-snth, metis add commute assm
le-iff-add node.distinct(1,5))
      subgoal for i' unfolding nd-notOne RR-defs
      using ipath-dist21[OF assms, of i' + i]
      ipath-dist31[OF assms, of i' + i]
      by (auto simp add: add commute sdrop-snth) .

    unfolding alw-ev-holds2-iff-snth
    apply(rule allI, simp)
    subgoal for i'
      apply(cases sdrop i nds !! i')
      subgoal apply(rule exI[of - Suc i'])
        using ipath-dist1[OF assms, of i' + i]
        ipath-dist21[OF assms, of Suc (i' + i)]
        ipath-dist31[OF assms, of Suc (i' + i)]
        by (auto simp add: add commute sdrop-snth, metis assm less-add-Suc2
nat-less-le node.distinct(1,5))
      subgoal by (metis le-add1 node.distinct(1,5) sdrop-snth assm)
      apply(rule exI[of - i'])
      using ipath-dist31[OF assms, of i' + i]
      by (simp add: add commute sdrop-snth) .

    qed

lemma ipath-mixed-descent:
  assumes ipath nds
  shows ipath-mixed nds  $\implies \exists Ps. \text{descentIpath nds Ps}$ 

```



```

unfolding ipaths alw-ev-holds-iff-snth
proof(elim conjE)
  assume assm: $\forall i. \exists j \geq i. nds !! j = Two$ 
     $\forall i. \exists j \geq i. nds !! j = Three$ 

  let ?Ps = smap2 ( $\lambda n next-n.$ 
    if n = Two then p2
    else if n = Three then p3
    else if next-n = Two then p1
    else p1 ') nds (stl nds)

  show ?thesis find-theorems sdrop 0
    apply (unfold descentIpath-def, intro exI[of - ?Ps] sdrop-evI[of - 0, unfolded
sdrop.simps(1)] conjI)

  subgoal unfolding alw-holds2-iff-snth
    using ipath-dist31[OF assms]
      ipath-dist21[OF assms]
      ipath-dist1[OF assms]
    by (auto simp add: stl-Suc-eq,(metis PosOf.elims RR-OTr RR-OT))+

  unfolding alw-ev-holds2-iff-snth
  apply(rule allI, simp add: RR-defs)
  using assm(2) assms ipath-dist31 by auto
qed

lemma pathConEx11:Graph.pathCon {One,Two,Three} ( $\lambda u v. edge\ u\ v$ ) One One
  unfolding Graph.pathCon-def
  using Graph.pathG.Step[of One {One,Two,Three} ( $\lambda u v. edge\ u\ v$ ) [One, Two]]
    Graph.pathG.Base[of One Two {One,Two,Three} ( $\lambda u v. edge\ u\ v$ )]
  by auto

lemma pathConEx12:Graph.pathCon {One,Two,Three} ( $\lambda u v. edge\ u\ v$ ) One Two
  unfolding Graph.pathCon-def
  using Graph.pathG.Base[of One Two {One,Two,Three} ( $\lambda u v. edge\ u\ v$ )]
  by auto

lemma pathConEx13:Graph.pathCon {One,Two,Three} ( $\lambda u v. edge\ u\ v$ ) One Three
  unfolding Graph.pathCon-def
  using Graph.pathG.Base[of One Three {One,Two,Three} ( $\lambda u v. edge\ u\ v$ )]
  by auto

lemma pathConEx21:Graph.pathCon {One,Two,Three} ( $\lambda u v. edge\ u\ v$ ) Two One
  unfolding Graph.pathCon-def
  using Graph.pathG.Base[of Two One {One,Two,Three} ( $\lambda u v. edge\ u\ v$ )]
  by auto

```

```

lemma pathConEx31: Graph.pathCon {One, Two, Three} (λu v. edge u v) Three
One
  unfolding Graph.pathCon-def
  using Graph.pathG.Base[of Three One {One, Two, Three} (λu v. edge u v)]
  by auto

lemma pathConEx23: Graph.pathCon {One, Two, Three} (λu v. edge u v) Two Three
  using Graph.pathCon-trans[OF pathConEx21 pathConEx13] .

lemma pathConEx32: Graph.pathCon {One, Two, Three} (λu v. edge u v) Three
Two
  using Graph.pathCon-trans[OF pathConEx31 pathConEx12] .

lemmas pathConEx = pathConEx11 pathConEx12
  pathConEx13 pathConEx21
  pathConEx31 pathConEx23
  pathConEx32

lemma scsgEx: scsg {One, Two, Three} (λu v. edge u v)
  unfolding scsg-def apply safe
  subgoal unfolding subgr-def by (auto)
  subgoal unfolding Graph.scg-def using pathConEx
  by (metis (full-types) Graph.pathCon-trans nd-notOne) .

lemma wfLabFSCases:
  wfLabFS {One, Two, Three} lab ⇒
  (lab One = {p1} ∨ lab One = {p1'} ∨ lab One = {p1, p1'}) ∧
  lab Three = {p3} ∧ lab Two = {p2} unfolding wfLabFS-def by auto

lemma noHCSC: wfLabFS {One, Two, Three} lab ⇒
  RRSetChoice {One, Two, Three} edge lab f ⇒ False
  apply(drule wfLabFSCases, unfold RRSetChoice-def, safe)
  subgoal
    apply(erule allE[of - One], erule allE[of - One])
    apply(erule allE[of - Three], erule allE[of - Three])
    by(auto simp: RR-defs)
  subgoal
    apply(erule allE[of - One], erule allE[of - One])
    apply(erule allE[of - Two], erule allE[of - Two])
    by(auto simp: RR-defs)
  subgoal premises prem
    using prem(1) prem(1)
    apply—apply(erule allE[of - One], erule allE[of - One])
    apply—apply(erule allE[of - Two], erule allE[of - Three], simp add: prem)
    using prem(2,5)
    apply—by(erule allE[of - One], erule allE[of - Three], simp add: RR-defs) .

```

```

proposition InfiniteDescent
  apply(rule InfiniteDescentI)
  apply(frule ipath-cases, elim disjE)
  subgoal using ipath12-descent by auto
  subgoal using ipath13-descent by auto
  subgoal using ipath-mixed-descent by auto .

proposition  $\neg$ XSDdescending
  unfolding XSDdescending-def apply safe
  apply(erule allE[of - {One,Two,Three}])
  apply(elim allE[of -  $\lambda u v. \text{edge } u v$  impE, simp add: scsgEx])
  apply(elim exE conjE)
  unfolding descending-PCSC-sliced-def apply clarsimp
  subgoal for lab f using noHCSC[of lab f] by auto .

```

end

4.5 Flat Cycles Criterion

```

theory Flat-Cycles-Criterion
imports ../Sloped-Graphs
begin

```

```

context Sloped-Graph
begin

```

```

definition FlatEdges::('node  $\times$  'node) set where
FlatEdges  $\equiv \{(u, v). \text{edge } u v \wedge (\forall p \in \text{PosOf } u. \forall q \in \text{PosOf } v. \neg \text{RR } (u,p) (v,q) \text{ Decr})\}$ 

```

```

lemma FlatEdges-no-Decr:: $(u,v) \in \text{FlatEdges} \implies \neg \text{RR } (u,p) (v,q) \text{ Decr}$ 
  unfolding FlatEdges-def
  apply(rule notI, frule RR-PosOf, safe)
  apply(drule bspec[of PosOf u  $\lambda p. \forall q \in \text{PosOf } v. \neg \text{RR } (u, p) (v, q) \text{ Decr } p$ ], simp))
  apply(drule bspec[of PosOf v - q], simp)
  using RR-SlopeRels[of u v, unfolded SlopedRels-def]
  by auto

```

definition *FlatCycle* :: bool **where**

FlatCycle $\equiv \exists nds. (\forall i < \text{length } nds - 1. (nds ! i, nds ! (\text{Suc } i)) \in \text{FlatEdges}) \wedge \text{cycleG } nds$

lemma *FlatCycle-properties*: *FlatCycle* $\implies \exists xs. \text{cycleG } xs \wedge (\forall i < \text{length } xs - 1.$

$\forall p q. \neg \text{RR } (xs ! i, p) (xs ! (i+1), q) \text{ Decr})$

apply (*unfold FlatCycle-def, erule exE*)

subgoal for *nds* **apply** (*intro exI[of - nds] conjI allI impI*)

subgoal by *auto*

subgoal for *i* **apply** (*elim conjE allE[of - i] impE, simp*)

using *FlatEdges-no-Decr[of nds ! i nds ! Suc i]* **by** *auto . .*

lemma *FlatCycle-imp-flat-ipath*:

assumes *FlatCycle*

shows $\exists S. \text{ipath } S \wedge (\forall i p q. \neg \text{RR } (S !! i, p) (S !! (\text{Suc } i), q) \text{ Decr})$

proof –

obtain *xs* **where** *cyc*: *cycleG xs*

and *no-decr-finite*: $\forall i < \text{length } xs - 1. \forall p q. \neg \text{RR } (xs ! i, p) (xs ! (i+1), q) \text{ Decr}$

using *FlatCycle-properties[OF assms]* **by** *blast*

let *?ys* = *butlast xs*

let *?S* = *srepeat ?ys*

have *xs-nemp*: *xs* $\neq []$ **using** *cyc unfolding cycleG-def* **by** *auto*

have *xs-len*: *length xs* ≥ 2 **using** *cyc unfolding cycleG-def* **by** (*simp add: path-length-ge2*)

have *len-pos*: *length ?ys* > 0

using *xs-nemp cyc*

by (*meson Graph.cycleG-def bot-nat-0.not-eq-extremum length-0-conv pathG-butlast-not-nil*)

have *ipaths*: *ipath ?S*

apply (*unfold ipath-def, intro allI conjI*)

subgoal using *cyc* **by** (*meson ipath-def cycle-srepeat-ipath*)

subgoal using *cyc cycle-srepeat-ipath ipath-def* **by** *auto .*

have *no-decr-stream*: $\forall i p q. \neg \text{RR } (?S !! i, p) (?S !! (\text{Suc } i), q) \text{ Decr}$

proof (*intro allI*)

fix *i p q*

let *?n* = *length ?ys*

let *?k* = *i mod ?n*

have *wrap*: *i mod (length xs - Suc 0) = length xs - Suc (Suc 0) $\implies$ Suc i mod (length xs - Suc 0) = 0*

by (*metis One-nat-def Suc-pred' diff-Suc-eq-diff-pred len-pos length-butlast mod-Suc*)

have *len-le*: $\text{length } xs - \text{Suc } (\text{Suc } 0) < \text{length } xs - \text{Suc } 0$ **using** *xs-len* **by** *fastforce*

have *butlast-wrap*: $\text{butlast } xs \neq 0 = xs \neq \text{Suc } (\text{length } xs - \text{Suc } (\text{Suc } 0))$
using *cyc xs-len last-conv-nth[OF xs-nemp]* **unfolding** *cycleG-def*
by (*metis One-nat-def Suc-diff-Suc Suc-le-lessD cyc cycle-iff-nth len-pos*
nth-butlast numeral-2-eq-2)

have *cases-mod*: $i \bmod \text{length } (\text{butlast } xs) = \text{length } (\text{butlast } xs) - 1 \vee i \bmod$
 $\text{length } (\text{butlast } xs) < \text{length } (\text{butlast } xs) - 1$
by (*metis Suc-diff-1 len-pos mod-less-divisor nat-cases*
not-less-eq)

have $\neg RR (?ys \neq ?k, p) (?ys \neq ((\text{Suc } i) \bmod ?n), q)$ *Decr*
using *no-decr-finite cases-mod*
apply—**apply**(*erule disjE*)

subgoal **apply**(*elim allE[of - length xs - Suc (Suc 0)] impE*)
subgoal **using** *xs-len* **by** *auto*
subgoal **apply**(*erule allE[of - p], erule allE[of - q]*)
using *nth-butlast[of length xs - Suc (Suc 0) xs]* **by** (*simp add: len-le*
butlast-wrap wrap) .

subgoal **apply**(*erule allE[of - i mod length (butlast xs)]*)
by (*simp,metis One-nat-def len-le len-pos length-butlast mod-Suc*
mod-less-divisor not-less-eq nth-butlast) .

thus $\neg RR (?S \neq i, p) (?S \neq (\text{Suc } i), q)$ *Decr*
using *len-pos* **by** *force*

qed
show *?thesis*
using *ipaths no-decr-stream* **by** *blast*
qed

lemma *FlatCycleE*: $\text{FlatCycle} \implies (\bigwedge xs. \text{ipath } xs \implies (\forall i p q. \neg RR (xs \neq i, p)$
 $(xs \neq (i+1), q) \text{ Decr}) \implies P) \implies P$
apply(*drule FlatCycle-imp-flat-ipath*)
using *cycle-srepeat-ipath* **by** *auto*

lemma *notFlatCycleI*:
 $(\bigwedge nds. \forall i < \text{length } nds - 1. (nds \neq i, nds \neq \text{Suc } i) \in \text{FlatEdges} \implies$
 $\text{cycleG } nds \implies \text{HOL.False}) \implies \neg \text{FlatCycle}$

```

by(unfold FlatCycle-def, rule notI, elim exE conjE, assumption)

lemma notFlatCycleI':
  ( $\bigwedge nds. \text{pathG } nds \implies$ 
     $\forall i < \text{length } nds - 1. (nds ! i, nds ! \text{Suc } i) \in \text{FlatEdges} \implies$ 
     $hd \text{ } nds = last \text{ } nds \implies HOL.False$ )  $\implies \neg \text{FlatCycle}$ 
by(unfold FlatCycle-def cycleG-def, rule notI, elim exE conjE, assumption)

```

```

theorem Flat-Cycles-Criterion:  $\text{FlatCycle} \implies \neg \text{InfiniteDescent}$ 
apply(rule notI, erule FlatCycleE)
unfolding InfiniteDescent-def
subgoal for xs apply(erule allE[of - xs], simp)
unfolding descentIpath-iff-snth by auto .

```

end

end

4.6 Descending Unicycles Criterion

```

theory Descending-Unicycles-Criterion
imports ../Sloped-Graphs
         ../Buchi-Preliminaries
begin

```

```

context Graph
begin

```

```

definition basicCycle :: 'node list  $\Rightarrow$  bool where
  basicCycle c  $\equiv \text{cycleG } c \wedge \text{distinct } (\text{butlast } c)$ 

```

```

lemmas basicCycle-defs = basicCycle-def[unfolded cycleG-def]

```

```

lemma basicCycleI:  $\text{pathG } c \implies hd \text{ } c = last \text{ } c \implies \text{distinct } (\text{butlast } c) \implies \text{basicCycle } c$ 
unfolding basicCycle-def cycleG-def by auto

```

```

lemma basicCycle-path-drop:  $\text{basicCycle } c \implies j < \text{length } (\text{butlast } c) \implies \text{pathG } (\text{drop } j \text{ } c)$ 
using cycleG-path-drop unfolding basicCycle-def by auto

```

```

lemma basicCycle-not-nil:  $\text{basicCycle } c \implies c \neq []$  unfolding basicCycle-def cycleG-def by auto

```

```

lemma basicCycle-ge2:  $\text{basicCycle } c \implies \text{length } c \geq 2$  unfolding basicCycle-def

```

cycleG-def **using** *path-length-ge2* **by** *auto*

lemma *cycle-elim*: $\neg(\exists c. \text{basicCycle } c \wedge \text{set } c = V') \implies (\forall c. \text{basicCycle } c \longrightarrow \text{set } c \neq V' \implies P) \implies P$ **by** *auto*

lemma *basicCycle-set-eq*: $\text{basicCycle } c \implies \text{set } c = \text{set } (\text{butlast } c)$

apply (*standard*)

unfolding *basicCycle-def* *cycleG-def*

using *path-length-ge2* [of *c*]

apply (*metis* *append-butlast-last-id* *basicCycleI*

basicCycle-not-nil *cycle-Cons* *hd-append2* *hd-in-set*

list.simps(3) *not-path-singl* *self-append-conv2*

set-ConsD *sset-cycle* *stream.set-intros*(2)

subset-code(1))

by (*meson* *in-set-butlastD* *subset-code*(1))

lemma *cycleG-contains-basicCycle*:

assumes *cycleG* *ndl*

shows $\exists c. \text{basicCycle } c \wedge \text{set } c \subseteq \text{set } ndl$

using *assms* **proof** (*induct* *length* *ndl* *arbitrary*: *ndl* *rule*: *less-induct*)

case *less*

note *cycle* = *less*(2)

then have *ndl-ne*: *ndl* $\neq []$ *length* *ndl* ≥ 2 **unfolding** *cycleG-def* **using** *path-length-ge2*

by *auto*

then show *?case*

proof (*cases* *distinct* (*butlast* *ndl*))

case *True*

then show *?thesis* **unfolding** *basicCycle-def* **using** *cycle* **by** *auto*

next

case *False*

then obtain *xs* *y* *ys* *zs* **where** *split*: *butlast* *ndl* = *xs* @ *y* # *ys* @ *y* # *zs* **using** *not-distinct-decomp* **by** *fastforce*

let *?c'* = *y* # *ys* @ [*y*]

have *pathG-butlast*: *pathG* (*butlast* *ndl*) **using** *cycle* [*unfolded* *cycleG-def*] *False* *pathG.cases* **by** *fastforce*

hence *pathG* *?c'* **unfolding** *split*

using *path-appendR* [of *xs* *y* # *ys* @ *y* # *zs*] *path-appendL* [of *y* # *ys* @ [*y*]] **by** *auto*

hence *cycleG* *?c'* **unfolding** *cycleG-def* **by** *auto*

also have *length* *?c'* < *length* *ndl* **using** *split* *length-butlast* [of *ndl*] *ndl-ne* **by** *auto*

moreover have $set\ ?c' \subseteq set\ ndl$ **by**(*rule subsetI, rule in-set-butlastD, unfold split, auto*)

ultimately show $?thesis$ **using** $less(1)[of\ ?c']$ **by** *auto*
qed
qed

lemma *basicCycle-rotate1*:

$basicCycle\ (ndl\ @\ [nd, nd']) \implies basicCycle\ (nd\ \# \ ndl\ @\ [nd])$

unfolding *basicCycle-defs path-iff-set-nth* **apply** *safe*

subgoal by *auto*

subgoal by *auto*

subgoal for i **apply** (*cases i*)

subgoal by *simp* (*metis append.assoc append.left-neutral append-Cons length-append-singleton*

$lessI\ list.sel(1)\ neq-Nil-conv\ nth-Cons-0\ nth-append-length$)

subgoal by *simp* (*metis Suc-lessI less-SucI nth-append nth-append-length*) .

subgoal by *simp*

by (*simp add: butlast-append*)

lemma *basicCycle-rotate*:

assumes $basicCycle\ (ndl1\ @\ ndl2\ @\ [nd'])\ ndl2 \neq []$

shows $basicCycle\ (ndl2\ @\ ndl1\ @\ [hd\ ndl2])$

using *assms* **proof**(*induction ndl2 arbitrary: ndl1 nd' rule: rev-induct*)

case Nil **thus** $?case$ **by** *auto*

next

case (*snoc nd2 ndl2 ndl1 nd'*)

show $?case$

proof(*cases ndl2 = []*)

case True

show $?thesis$ **unfolding** *True* **apply** *simp*

apply(*rule basicCycle-rotate1*) **using** *snoc True* **by** *auto*

next

case False

hence $1: ndl2\ @\ nd2\ \# \ ndl1\ @\ [hd\ ndl2] = hd\ ndl2\ \# \ (tl\ ndl2\ @\ nd2\ \# \ ndl1)\ @\ [hd\ ndl2]$ **by** *auto*

have $2: (tl\ ndl2\ @\ nd2\ \# \ ndl1)\ @\ [hd\ ndl2, hd\ (tl\ ndl2\ @\ [nd2])] =$
 $(tl\ ndl2\ @\ [nd2])\ @\ (ndl1\ @\ [hd\ ndl2])\ @\ [hd\ (tl\ ndl2\ @\ [nd2])]$

by *auto*

show $?thesis$ **using** *False* **apply** *simp* **unfolding** *append-Cons[symmetric]*

apply(*rule snoc.IH[where nd' = nd2]*)

subgoal unfolding *append-Cons* **unfolding** *append-assoc[symmetric]* **ap-**
ply(*rule basicCycle-rotate1*)

using *snoc* **by** *auto*

subgoal by *auto* .

qed

qed

lemma *basicCycle-rotate-butlast*:

assumes *basicCycle* (*ndl1* @ *nd* # *ndl2*) *ndl1* ≠ [] *ndl2* ≠ []
shows *basicCycle* (*nd* # *butlast* *ndl2* @ *ndl1* @ [*nd*])
using *assms* *basicCycle-rotate*[**where** *nd'* = *last* (*nd* # *ndl2*)
and ?*ndl2*.0 = *butlast* (*nd* # *ndl2*) **and** ?*ndl1*.0 = *ndl1*] **using** *assms* **by**
simp

lemma *basicCycle-rotate-set*:
assumes *basicCycle* *ndl* *nd* ∈ *set* *ndl*
shows ∃ *ndl'*. *set* *ndl'* = *set* *ndl* ∧ *basicCycle* *ndl'* ∧ *hd* *ndl'* = *nd* ∧ *last* *ndl'* =
nd ∧ *length* *ndl'* = *length* *ndl*
proof(*cases* *hd* *ndl* = *nd* ∨ *last* *ndl* = *nd*)
case *True* **thus** ?*thesis* **apply**(*intro* *exI*[*of* - *ndl*])
using *assms* **unfolding** *basicCycle-defs* **by** *auto*
next
case *False* **then obtain** *ndl1* *ndl2* **where** *ndl*: *ndl* = *ndl1* @ *nd* # *ndl2* *ndl1*
≠ [] *ndl2* ≠ []
by (*metis* *assms*(2) *hd-append* *last-ConsL* *last-append* *list.sel*(1) *list.simps*(3)
split-list)
thus ?*thesis* **using** *basicCycle-rotate-butlast*[*OF* *assms*(1)[*unfolded* *ndl*(1)] *ndl*(2,3)]

apply(*intro* *exI*[*of* - *nd* # *butlast* *ndl2* @ *ndl1* @ [*nd*]]) **using** *assms* *ndl*
apply *safe*
subgoal **using** *in-set-butlastD* **by** *force*
subgoal **by** (*smt* *Un-iff* *append-butlast-last-id* *hd-append2* *hd-conv-nth* *in-set-conv-nth*
insert-iff
last.simps *last-appendR* *length-greater-0-conv* *list.set*(2) *basicCycle-defs* *set-append*)
by *auto*
qed

lemma *basicCycle-smaller-alt*:
assumes *basicCycle* *c*
assumes *i* < *length* *c* - 1
assumes *edge* (*c* ! *i*) *v* *v* ∈ *set* (*butlast* *c*)
assumes *assm*: *v* ≠ *c* ! *Suc* *i*
shows ∃ *c-alt*. *basicCycle* *c-alt* ∧ *set* *c-alt* ⊆ *set* *c* ∧ *v* ∈ *set* *c-alt* ∧ (*c* ! *i*) ∈ *set*
c-alt ∧ (*c* ! *Suc* *i*) ∉ *set* *c-alt*
proof -
from *assms*(1) **have** *pG*: *pathG* *c* **and** *dist*: *distinct* (*butlast* *c*) **and** *cyc*: *cycleG*
c
unfolding *basicCycle-def* **using** *cycleG-def* **by** *auto*

hence *c-Suc-in*: *c* ! *Suc* *i* ∈ *set* (*butlast* *c*) **using** *assms* *basicCycle-set-eq*[*OF*
assms(1)]
by (*metis* *diff-Suc-1* *diff-less-Suc* *in-set-conv-nth* *less-trans-Suc*
not-less-less-Suc-eq)
have *hd-eq*: *hd* (*butlast* *c*) = *hd* *c* **by** (*metis* *append-butlast-last-id* *assms*(4)
hd-append2 *list.set*(1) *set-nemp* *not-path-Nil* *pG*)

```

obtain  $j$  where  $j: j < \text{length } c - 1 \vee v = c \wedge j \neq \text{Suc } i$ 
using  $\text{assms}(4)$   $\text{length-butlast}[of\ c]$  by ( $\text{metis } \text{assm } \text{in-set-conv-nth } \text{nth-butlast}$ )

show  $?thesis$  proof( $\text{cases } j \leq i$ )
case  $\text{True}$ 

define  $c\text{-alt}$  where  $c\text{-alt}: c\text{-alt} = \text{map } (!) (\text{butlast } c) [j..<\text{Suc } i] @ [c!j]$ 
hence  $c\text{-alt}'$ :  $c\text{-alt} = \text{drop } j (\text{take } (\text{Suc } i) (\text{butlast } c)) @ [c!j]$ 
using  $\text{map-nth-upt-drop-take-conv}[of\ \text{Suc } i\ \text{butlast } c\ j]$ 
using  $\text{assms}(2)$  by  $\text{auto}$ 

hence  $c\text{-eq}: c = \text{take } j\ c @ \text{drop } j (\text{take } (\text{Suc } i) (\text{butlast } c)) @ \text{drop } (\text{Suc } i)\ c$ 
by ( $\text{metis } \text{All-less-Suc } \text{True } \text{add-diff-cancel-left}'\ \text{add-increasing } \text{append-take-drop-id}$ 
 $\text{assms}(2)\ \text{diff-Suc-1}'$ 
 $\text{drop-take-drop-unsplit } \text{le-simps}(2)\ \text{less-diff-conv } \text{less-eqE } \text{nat-geq-1-eq-neqz}$ 
 $\text{nat-in-between-eq}(2)\ \text{plus-1-eq-Suc}$ 
 $\text{semiring-norm}(174)\ \text{take-butlast}$ )

have  $c\text{-alt-dist}: \text{distinct } (\text{butlast } c\text{-alt})$ 
unfolding  $c\text{-alt}'$ 
using  $\text{distinct-drop}[of\ -\ j,\ OF\ \text{distinct-take}[OF\ \text{dist},\ of\ \text{Suc } i]]$  by  $\text{auto}$ 

from  $c\text{-alt}$  have  $\text{last}: c \wedge i = \text{last } (\text{butlast } c\text{-alt})$ 
apply ( $\text{simp } \text{add}: \text{True}$ )
by ( $\text{metis } \text{assms}(2)\ \text{butlast.simps}(2)\ \text{butlast-append } \text{last-snoc } \text{length-butlast}$ 
 $\text{list.distinct}(1)\ \text{nth-butlast}$ )

also have  $\text{first}: \text{hd } c\text{-alt} = c \wedge j \vee v \in \text{set } c\text{-alt}$ 
unfolding  $c\text{-alt}$  apply ( $\text{simp-all } \text{add}: \text{hd-append } \text{hd-drop-conv-nth } j\ \text{True}$ )
by ( $\text{metis } \text{True } \text{butlast-conv-take } \text{hd-map } \text{hd-upt } j(1)\ \text{le-0-eq } \text{nat-less-le } \text{nth-take}$ 
 $\text{upt-eq-Nil-conv}$ )

hence  $c\text{-alt} \neq []$  by  $\text{fastforce}$ 
then have  $c \wedge j \in \text{set } (\text{butlast } c\text{-alt})$  using  $\text{first } c\text{-alt } \text{True}$ 
by ( $\text{simp,metis } \text{butlast-snoc } c\text{-alt } \text{hd-append2 } \text{hd-in-set } \text{list.discI } \text{list-se-match}(2)$ 
 $\text{not-Cons-self2}$ )

hence  $c\text{-in}: c \wedge i \in \text{set } (\text{butlast } c\text{-alt})$  apply-by( $\text{unfold } \text{last}, \text{rule } \text{last-in-set},$ 
 $\text{fastforce}$ )

```

```

have c-alt-path:pathG c-alt
proof(cases i = j)
  case True
  then show ?thesis
    unfolding c-alt apply (simp split:if-splits add: take-Suc hd-drop-conv-nth)
    using assms(3) j(1,2) pG pathG.intros(1) path-nth-'node by (simp add:
nth-butlast)
  next
  case False
  hence j<i using True by auto
  thus ?thesis
    apply-apply(unfold c-alt,rule pathG.Step)
    subgoal using assms(1,4) basicCycle-set-eq j(2) pG path-set by auto
    subgoal using assms(1)[unfolded basicCycle-defs]
      c-alt[symmetric, unfolded c-alt] c-eq True apply simp
      apply(rule path-appendM[of take j c drop j (take (Suc i) (butlast c))
drop (Suc i) c], simp)
      by (metis Suc-leI Suc-mono length-append-singleton length-map length-upt
numerals(2) zero-less-diff)
      using last by (metis assms(3) butlast-snoc c-alt j(2))
    qed

moreover have basicCycle c-alt
  using calculation c-alt

  unfolding basicCycle-defs
  apply(intro conjI)
  subgoal using c-alt-path by auto
  subgoal using first by auto
  subgoal using c-alt-dist by auto .

moreover have c ! Suc i ∉ set c-alt
proof(cases Suc i = length (butlast c))
  case True
  hence c-Suc:c ! Suc i = c ! 0
    using assms(1)[unfolded basicCycle-defs]
    by (metis append-butlast-last-id cyc cycleG-not-nil
hd-conv-nth nth-append-length)

  hence j-gr:j > 0 using j assm[unfolded c-Suc] bot-nat-0.not-eq-extremum by
blast
  show ?thesis
    unfolding c-alt c-Suc apply (simp add: True, intro conjI)
    subgoal using assm[unfolded c-Suc] j(2) c-Suc by (metis)
    subgoal using j-gr dist unfolding distinct-conv-nth[of butlast c] ap-
ply-apply(elim allE[of - 0], erule impE)
    subgoal using True by linarith

```

```

      subgoal using distinct-outside-index[of j, OF - - dist]
      by (metis One-nat-def True length-butlast nth-butlast zero-less-Suc)
    ..
  next
  case False
  hence Suc i < length (butlast c) using assms(2) by fastforce
  then show ?thesis
  unfolding c-alt apply (simp add: True, intro conjI)
  subgoal using nth-butlast[of i c, unfolded length-butlast, OF assms(2)]
    dist distinct-conv-nth[of butlast c]
    by (metis assms(2) length-butlast n-not-Suc-n nth-butlast)
  subgoal using j dist distinct-conv-nth[of butlast c] assm by fastforce
  subgoal using dist unfolding distinct-conv-nth[of butlast c] apply-apply(elim
alle[of - Suc i], erule impE)
    subgoal by auto
    subgoal using distinct-outside-index'[OF dist] by blast . .
  qed

  ultimately show ?thesis
  apply-apply(rule exI[of - c-alt], simp, intro conjI)
  subgoal using c-alt-dist unfolding c-alt' by (metis assms(1) basicCy-
cle-set-eq in-set-dropD in-set-takeD snoc-eq-iff-butlast'
    subset-code(1))
  subgoal using j first(2) by blast
  subgoal by (metis ci-in in-set-butlastD) .
next
case False
hence False':j > Suc i using j by auto
define c-alt where c-alt:c-alt = (drop j (butlast c)) @ (take (Suc i) (butlast
c)) @ [c!j]

hence butlast-alt:butlast c-alt = (drop j (butlast c)) @ (take (Suc i) (butlast c))
by (simp add: butlast-append)
from c-alt have last:last (take (Suc i) (butlast c)) = c ! i
using last-take-nth-conv assms(2) bot-nat-0.extremum-strict diff-Suc-1 diff-le-self

by (metis (no-types, lifting) butlast-conv-take length-butlast nth-take le-trans
lessI less-eq-Suc-le )

hence first:hd c-alt = c ! j unfolding c-alt
by (metis butlast-conv-take drop-eq-Nil2 hd-append hd-drop-conv-nth j(1)
length-butlast nth-take
  verit-comp-simplify1(3))
hence hd-last:hd c-alt = last c-alt by (simp add: c-alt)

have vin:v ∈ set c-alt unfolding j c-alt by auto
have cin:c ! i ∈ set c-alt unfolding c-alt using last
by (metis Misc.last-in-set Zero-not-Suc assms(4) butlast-snoc gr0-conv-Suc

```

```

in-set-butlast-appendI length-0-conv
  length-pos-if-in-set take-eq-Nil2)

  have set-incls:set (drop j (butlast c))  $\subseteq$  set c  $\wedge j \in$  set c set (take (Suc i)
(butlast c))  $\subseteq$  set c
  apply (simp add: assms(1) basicCycle-set-eq set-drop-subset)
  apply (metis assms(4) in-set-butlastD j(2))
  by (meson in-set-butlastD in-set-takeD subset-code(1))
  hence set-in-alt:set c-alt  $\subseteq$  set c unfolding c-alt by auto

  have c-alt-path:pathG c-alt
  unfolding c-alt
  apply(cases length c > 2)
  subgoal apply(rule path-append-hd)
  subgoal using basicCycle-path-drop[OF assms(1) j(1)[unfolded length-butlast[symmetric]]]
    assms(1)[unfolded basicCycle-defs]
    by (simp,metis assms(4) hd-append2 length-greater-0-conv length-pos-if-in-set
list.size(3) take-eq-Nil
  zero-less-Suc append-take-drop-id basicCycleI basicCycle-not-nil distinct-drop
drop-butlast hd-Nil-eq-last last-appendR
  snoc-eq-iff-butlast)
  subgoal apply(cases i, simp add: take-Suc hd-eq)
  subgoal apply(intro conjI)
    subgoal by (metis assms(4) list.set(1) memb-imp-not-empty)
    apply(rule pathG.Base)
    subgoal using path-set[OF pG] set-incls by force
    subgoal using assms j by (simp add: hd-conv-nth) .
  apply(rule pathG.Step)
  subgoal using pG path-iff-set-nth set-incls(2) by auto
  subgoal apply(rule path-appendL[of take (Suc i) (butlast c) drop (Suc i)
(butlast c)])
    subgoal by (simp add: pG pathG-butlast)
    subgoal by fastforce .
    subgoal unfolding last using assms j by (simp add: hd-conv-nth) . .

  subgoal using False' apply(cases i, simp-all add: take-Suc)
  subgoal using assms(2)[unfolded basicCycle-defs] j(1) by force
  subgoal using j(1) by auto . .

  have c-alt-dist:distinct (butlast c-alt)
  using distinct-wrap-around[OF dist False'] unfolding butlast-alt by auto

  have cS-notin:c ! Suc i  $\notin$  set c-alt
  proof(cases Suc i = length (butlast c))
  case True
    show ?thesis using j(1) True False by auto
  next
  case False
    hence succ:Suc i < length (butlast c) using assms(2) by fastforce

```

```

then show ?thesis
  unfolding c-alt apply (simp add: False, intro conjI)
  subgoal using nth-butlast[of i c, unfolded length-butlast, OF assms(2)]
    dist distinct-conv-nth[of butlast c]
    using assm j(2) by fastforce
  subgoal using False' atLeastLessThan-empty atLeastLessThan-empty-iff2
    dist nth-butlast nth-eq-iff-index-eq succ in-set-drop-conv-nth by
metis
  subgoal by (metis Cons-nth-drop-Suc append-take-drop-id dist distinct-take
not-distinct-conv-prefix nth-butlast) .
qed

note all = set-in-alt vin cin cS-notin c-alt-dist c-alt-path hd-last

show ?thesis unfolding basicCycle-defs using all by auto
qed
qed

lemma scsg-has-basicCycle:  $V' \neq \{\}$   $\implies$   $scsg\ V'\ E' \implies \exists c. basicCycle\ c \wedge set\ c \subseteq V'$ 
  unfolding scsg-def Graph.scg-def Graph.pathCon-def set-nemp
  apply (clarsimp)
  subgoal for v
    using Graph.cycleG-def[of  $V'\ E'$ , symmetric]
      cycleG-contains-basicCycle[of ]
    by (metis Graph.path-set basic-trans-rules(23)
cycle-subgr) .

lemma scsg-node-in-basic-cycle:
  assumes  $v \in V'$ 
  assumes  $scsg\ V'\ E'$ 
  shows  $\exists c. basicCycle\ c \wedge v \in set\ c \wedge set\ c \subseteq V'$ 
proof -
  have pathConn:  $Graph.pathCon\ V'\ E'\ v\ v$  using scsg-paths[OF assms(2)] assms(1)
  by auto

  then obtain p where  $p: Graph.pathG\ V'\ E'\ p\ hd\ p = v\ last\ p = v\ length\ p > 1$ 
  unfolding Graph.pathCon-def using Graph.not-path-Nil path-length-ge2
  by (metis Graph.path-length-ge2 Suc-1 Suc-to-right
not-numeral-le-zero nz-le-conv-less
verit-comp-simplify1(3))

  hence cycFrom:  $Graph.cycleFrom\ V'\ E'\ v\ p$  unfolding Graph.cycleFrom-def
  Graph.cycleG-def by auto

```

```

let ?Cycles = {p. Graph.cycleFrom V' E' v p}

let ?min-len = Least (λp. p ∈ length ‘ ?Cycles)

from cycFrom have cyc-nemp: ?Cycles ≠ {} by blast
then have len-nemp: length ‘ ?Cycles ≠ {} by auto

then obtain p-min where
  pmin-mem: p-min ∈ ?Cycles and
  pmin-len: length p-min = ?min-len
  using cyc-nemp LeastI-ex unfolding image-iff
  by (smt (verit, best) equals0I)

hence p-min-least: ∀ p'. p' ∈ ?Cycles ⟶ length p-min ≤ length p' by (simp add:
Least-le)
also have p-min-cyc: Graph.cycleG V' E' p-min
  using pmin-mem unfolding Graph.cycleFrom-def
  by (metis Graph.scsg.elims(2) assms(2) cycle-subgr
    mem-Collect-eq)

have p-min-cycF: cycleG p-min
  using pmin-mem unfolding Graph.cycleFrom-def
  by (metis Graph.scsg.elims(2) assms(2) cycle-subgr
    mem-Collect-eq)

have v-in-pmin: v ∈ set p-min
  using pmin-mem unfolding Graph.cycleFrom-def
  by (metis (lifting) Graph.cycleG-def Graph.not-path-Nil
    list.set-sel(1) mem-Collect-eq)

hence v-in-pmin: v ∈ set (butlast p-min) using p-min-cyc[unfolded Graph.cycleG-def]

  by (smt (verit, ccfv-threshold) Graph.cycleFrom-def Graph.not-path-Nil
    Graph.pathG.simps butlast.simps(2) butlast-snoc hd-append2 list.sel(1)
    list.set-intros(1) list.set-sel(1) mem-Collect-eq pmin-mem split-list-first)

have v-loc: hd p-min = v last p-min = v using pmin-mem unfolding Graph.cycleFrom-def
Graph.cycleG-def by auto

have basicCycle p-min
proof (rule ccontr)
  assume ¬ basicCycle p-min
  hence ¬ distinct (butlast p-min) unfolding basicCycle-def using p-min-cycF

```

```

by auto
  then obtain  $xs\ y\ ys\ zs$  where  $split: butlast\ p-min = xs @ y \# ys @ y \# zs$ 
  using  $not-distinct-decomp$  by fastforce

  let  $?c' = y \# ys @ [y]$ 

  have  $pathG-butlast: Graph.pathG\ V'\ E'\ (butlast\ p-min)$  using  $p-min-cyc[unfolded\ Graph.cycleG-def]$ 
  split  $Graph.pathG.cases$  by fastforce
  hence  $Graph.pathG\ V'\ E'\ ?c'$  unfolding  $split$  using  $Graph.path-appendR[of\ V'\ E'\ xs\ y \# ys @ y \# zs]$ 
   $Graph.path-appendL[of\ V'\ E'\ y \# ys @ [y]]$  by auto

  hence  $cyc-c': Graph.cycleG\ V'\ E'\ ?c'$  unfolding  $Graph.cycleG-def$  by auto

  have  $cyc-l: Graph.cycleG\ V'\ E'\ (xs @ y \# zs @ [v])$ 
  unfolding  $Graph.cycleG-def$ 
  using  $v-loc\ split$ 
  using  $p-min-cyc[unfolded\ Graph.cycleG-def]$ 
  using  $append-butlast-last-id[of\ p-min, OF\ Graph.cycleG-not-nil[OF\ p-min-cyc],$ 
   $symmetric]$ 
  apply-apply(rule  $conjI$ )
  subgoal apply(cases  $xs$ )
  subgoal apply(cases  $zs$ )
    subgoal using  $Graph.path-appendR[of\ V'\ E'\ y \# ys [y, v]]$  by auto
    subgoal for  $a\ ls$  using  $Graph.path-appendR[of\ V'\ E'\ y \# ys y \# a \# ls$ 
    @  $[v]]$  by auto .
    subgoal for  $x'\ xs'$  apply(cases  $zs$ )
    subgoal using  $Graph.path-append-hd[of\ V'\ E'\ v \# xs' [y, v]]$ 
       $Graph.path-appendR[of\ V'\ E'\ x' \# xs' @ y \# ys [y, v]]$ 
       $Graph.path-appendL[of\ V'\ E'\ x' \# xs' @ [y] ys @ [y, v]]$  by auto
    subgoal for  $z'\ zs'$  apply simp
    apply(rule  $Graph.path-append-hd[of\ V'\ E'\ v \# xs' @ [y] z' \# zs' @ [v],$ 
       $OF - Graph.path-appendR[of\ V'\ E'\ x' \# xs' @ y \# ys$ 
    @  $[y] z' \# zs' @ [v]],$ 
       $unfolded\ list-unf])$ 
    using  $Graph.path-appendM[of\ V'\ E'\ x' \# xs' @ y \# ys [y, z']\ zs' @ [v]]$ 
    using  $Graph.path-appendL[of\ V'\ E'\ x' \# xs' @ [y] ys @ y \# z' \# zs'$ 
    @  $[v]]$ 
    using  $Graph.path-append-hd[of\ V'\ E'\ v \# xs' [y, z']]$  by auto . .
  by (metis  $Nil-is-append-conv\ hd-append\ last.simps\ last-append\ list.distinct(1)$ 
     $list.sel(1)$ )

  have  $length-smaller: length\ ?c' < length\ p-min$ 
     $length\ (xs @ y \# zs @ [v]) < length\ p-min$ 
  using  $append-butlast-last-id[of\ p-min, OF\ cycleG-not-nil[OF\ p-min-cycF],$ 
   $unfolded\ split\ v-loc]$ 
  by auto

```


have $v \in \text{set } ?c' \vee v \in \text{set } (xs @ y \# zs @ [v])$ **using** $v\text{-in-pmin}[\text{unfolded split}]$
by *auto*

thus *HOL.False*

apply–**apply**(*elim disjE*)

subgoal **apply**(*drule Graph.cycle-rotate-set[OF cyc-c', of v], clarify*)

subgoal **for** ndl' **using** $p\text{-min-least}$

apply–**apply**(*elim allE[of - ndl'] impE, simp add: Graph.cycleFrom-def*)
using length-smaller **by** *auto* .

subgoal **apply**(*drule Graph.cycle-rotate-set[OF cyc-l, of v], clarify*)

subgoal **for** ndl' **using** $p\text{-min-least}$

apply–**apply**(*elim allE[of - ndl'] impE, simp add: Graph.cycleFrom-def*)
using length-smaller **by** *auto* . .

qed

with $\langle p\text{-min} \in ?Cycles \rangle$ **show** $?thesis$

apply–**apply**(*rule exI[of - p-min], intro conjI, simp*)

apply (*metis Graph.cycleFrom-def basicCycle-not-nil*
 $hd\text{-in-set mem-Collect-eq}$)

by (*meson Graph.cycleFrom-def Graph.cycleG-def*
 $Graph.path\text{-set mem-Collect-eq}$)

qed

definition $\text{connectedCycles} :: 'node\ list \Rightarrow 'node\ list \Rightarrow bool$ **where**
 $\text{connectedCycles } c1\ c2 \equiv (\exists p. \text{pathG } p \wedge hd\ p \in \text{set } c1 \wedge last\ p \in \text{set } c2)$

lemma $\text{connectedCyclesE} : \text{connectedCycles } c1\ c2 \Longrightarrow (\bigwedge p. \text{pathG } p \Longrightarrow$
 $hd\ p \in \text{set } c1 \Longrightarrow last\ p \in \text{set } c2 \Longrightarrow P) \Longrightarrow P$
unfolding $\text{connectedCycles-def}$ **by** *auto*

definition $\text{unicyclesGraph} :: bool$ **where**

$\text{unicyclesGraph} \equiv$

$\forall c\ c'. \text{basicCycle } c \wedge \text{basicCycle } c' \longrightarrow$

$(\text{connectedCycles } c\ c' \wedge \text{connectedCycles } c'\ c) \longrightarrow \text{set } c = \text{set } c'$

lemma $\text{unicyclesGraphI} : (\bigwedge c\ c'. \text{connectedCycles } c\ c' \Longrightarrow$

$\text{basicCycle } c \Longrightarrow$

$\text{basicCycle } c' \Longrightarrow$

$\text{connectedCycles } c'\ c \Longrightarrow \text{set } c = \text{set } c') \Longrightarrow \text{unicyclesGraph}$

unfolding $\text{unicyclesGraph-def}$ **by** *auto*

lemma $\text{unicyclesGraphI}' : (\bigwedge c\ c'. p.$

```

    pathG p  $\implies$ 
    hd p  $\in$  set c  $\implies$ 
    last p  $\in$  set c'  $\implies$ 
    basicCycle c  $\implies$ 
    basicCycle c'  $\implies$ 
    connectedCycles c' c  $\implies$  set c = set c'  $\implies$  unicyclesGraph
unfolding unicyclesGraph-def connectedCycles-def by auto

lemma basicCycle-unique-successor:
  assumes unicyclesGraph
  assumes basicCycle c
  assumes i < length c - 1
  assumes edge (c ! i) v v  $\in$  set (butlast c)
  shows v = c ! (Suc i)
proof (rule ccontr)
  assume assm: v  $\neq$  c ! Suc i

  from assms(2) have pG: pathG c and dist: distinct (butlast c) and cyc: cycleG
  c
  unfolding basicCycle-def using cycleG-def by auto

  hence c-Suc-in: c ! Suc i  $\in$  set (butlast c) using assms basicCycle-set-eq[OF
  assms(2)]
  by (metis diff-Suc-1 diff-less-Suc in-set-conv-nth less-trans-Suc
  not-less-less-Suc-eq)
  have hd-eq: hd (butlast c) = hd c by (metis append-butlast-last-id assms(5)
  hd-append2 list.set(1) memb-imp-not-empty not-path-Nil pG)

  obtain c-alt where alt: basicCycle c-alt set c-alt  $\subseteq$  set c
  v  $\in$  set c-alt (c ! i)  $\in$  set c-alt
  (c ! Suc i)  $\notin$  set c-alt using basicCycle-smaller-alt[OF assms(2 -)
  assm] by blast

  have connectedCycles c c-alt connectedCycles c-alt c
  unfolding connectedCycles-def
  using alt assms(2)  $\langle v \in$  set c-alt  $\rangle$  assms(5)
  by (metis alt(1,2) basicCycle-def basicCycle-defs cycleG-not-nil list.set-sel(1)
  subset-code(1) basicCycle-set-eq)+

  with assms(1) alt(1) assms(2) have set c-alt = set c unfolding unicycles-
  Graph-def by blast

  thus HOL.False using  $\langle v \neq$  c ! Suc i  $\rangle$  using alt
  by (metis One-nat-def Suc-diff-Suc Suc-mono assms(3) cyc cycleG-not-nil
  length-greater-0-conv minus-nat.diff-0 nth-mem)

```

qed

lemma *scsg-in-unicycles-is-basicCycle*:

assumes *unicyclesGraph*

$V' \neq \{\}$

scsg $V' E'$

shows $\exists c. \text{basicCycle } c \wedge \text{set } c = V'$

proof (*rule ccontr, erule cycle-elim*)

assume *notFull*: $\forall c. \text{basicCycle } c \longrightarrow \text{set } c \neq V'$

obtain *c1* **where** *cycle1*: *basicCycle* *c1* *set* *c1* $\subseteq V'$ **using** *scsg-has-basicCycle*[*OF* *assms*(2,3)] **by** *auto*

then obtain *c'* **where** *c':c' ∈ set c1* **using** *basicCycle-not-nil* **by** *blast*

from *cycle1* **obtain** *v* **where** *v:v ∈ V' v ∉ set c1* **using** *notFull* **by** *auto*

then obtain *c2* **where** *cycle2*:*basicCycle* *c2* *v ∈ set c2 set c2 ⊆ V'*
using *scsg-node-in-basic-cycle*[*OF* *v*(1) *assms*(3)] **by** *blast*

then have *cycles-neq*:*set c1 ≠ set c2* **using** *v*(2) **by** *auto*

note *basicCycles* = *cycle1*(1) *cycle2*(1)

have *cc1*:*connectedCycles* *c1 c2*

unfolding *connectedCycles-def*

using *scsg-paths*[*OF* *assms*(3), *simplified*]

apply—apply(*erule allE*[*of* - *c'*], *elim allE*[*of* - *v*] *impE*)

subgoal using *c'* *cycle1*(2) *v*(1) **by** *blast*

subgoal unfolding *Graph.pathCon-def* **using** *Graph.scsg-def* *assms*(3) *c'*
cycle2(2) *path-subgr* **by** *blast* .

have *cc2*:*connectedCycles* *c2 c1*

unfolding *connectedCycles-def*

using *scsg-paths*[*OF* *assms*(3), *simplified*]

apply—apply(*erule allE*[*of* - *v*], *elim allE*[*of* - *c'*] *impE*)

subgoal using *c'* *cycle1*(2) *v*(1) **by** *blast*

subgoal unfolding *Graph.pathCon-def* **using** *Graph.scsg-def* *assms*(3) *c'*
cycle2(2) *path-subgr* **by** *blast* .

note *connectedCycles* = *cc1 cc2*

show *HOL.False*

using *assms*(1)[*unfolded unicyclesGraph-def*] *basicCycles* *cc1 cc2*

apply—by(*erule allE*[*of* - *c1*], *elim allE*[*of* - *c2*] *impE conjE*, *auto simp: cycles-neq*)

qed

lemma *unicycle-limit-is-basic-cycle*:

assumes *unicyclesGraph*
assumes *finite Node*
assumes *ipath* π
shows $\exists c. \text{basicCycle } c \wedge \text{limitS } \pi = \text{set } (\text{butlast } c)$
using *scsg-in-unicycles-is-basicCycle* [*OF* *assms*(1) *limitS-ne* [*OF* *assms*(2,3)] *scsg-limit* [*OF* *assms*(2,3)]]
basicCycle-set-eq **by** *auto*

lemma *limitS-limitR-edges*:

assumes *unicyclesGraph*
assumes *finite Node*
assumes *ipath* π
assumes $\text{limitS } \pi = \text{set } (\text{butlast } c)$
assumes *basicCycle* c
shows $\forall i < \text{length } c - 1. (\text{limitR } \pi) (c ! i) (c ! \text{Suc } i)$
proof (*rule allI*, *rule impI*)
fix i **assume** *assm*: $i < \text{length } c - 1$
let $?u = c ! i$
let $?v = c ! \text{Suc } i$

have *u-in-lim*: $?u \in \text{limitS } \pi$ **using** *assms*(4) **by** (*metis* *assm* *length-butlast* *nth-butlast* *nth-mem*)

have *edge-dist*: $\forall v'. \text{edge } ?u \ v' \wedge v' \in \text{limitS } \pi \longrightarrow v' = ?v$
using *basicCycle-unique-successor* [*OF* *assms*(1,5) *assm*, *unfolded* *assms*(4)] [*symmetric*]
by *auto*

obtain i' **where** *ipaths*: *Graph.ipath* (*limitS* π) (*limitR* π) (*sdrop* $i' \pi$)
using *ipath-sdrop-limit* [*OF* *assms*(2,3)] **by** *blast*

hence $n': \forall n \geq i'. \pi !! n \in \text{limitS } \pi$ **unfolding** *Graph.ipath-iff-snth* *sdrop-snth* **by** (*metis* *less-eqE*)

then obtain n **where** $n: n \geq i' \ ?u = \pi !! n$
using *limitS-infinite-visits* [*OF* *u-in-lim*, *of* i'] **by** *auto*

have *lim*: $\pi !! (\text{Suc } n) \in \text{limitS } \pi$
using $n' \ n(1)$ *le-Suc-eq* **by** *blast*

```

have edge: edge ( $\pi !! n$ ) ( $\pi !! (Suc\ n)$ )
  using assms(3) unfolding ipath-def by (simp add: alw-holds2-iff-snth)

hence edge':edge ?u ( $\pi !! (Suc\ n)$ ) using n(2) by simp

hence eq:?v =  $\pi !! (Suc\ n)$  using edge-dist lim by auto

show (limitR  $\pi$ ) ?u ?v
  using n(1) ipaths[unfolded Graph.ipath-iff-snth]
  unfolding eq n(2) sdrop-snth apply by(erule allE[of - n-i'], simp)

qed

lemma unicycle-lasso:
  assumes unicyclesGraph
  assumes finite Node
  assumes ipath  $\pi$ 
  shows  $\exists v\ u.\ \pi = v @- srepeat\ u \wedge basicCycle\ (u @ [hd\ u])$ 
proof -

  have scsg (limitS  $\pi$ ) (limitR  $\pi$ ) using scsg-limit[OF assms(2,3)] .

  then obtain c where c-prop:basicCycle c set (butlast c) = limitS  $\pi$ 
    using unicycle-limit-is-basic-cycle[OF assms] by blast

  also obtain i where final-ipath:Graph.ipath (set (butlast c)) (limitR  $\pi$ ) (sdrop i
   $\pi$ )
    using ipath-sdrop-limit[OF assms(2,3)] calculation by auto

  hence hd-in:shd (sdrop i  $\pi$ )  $\in$  set c using basicCycle-set-eq[OF c-prop(1)]
  Graph.sset-ipath by blast

  obtain c' where c'-prop:basicCycle c' set c' = set c hd c' = shd (sdrop i  $\pi$ )
    last c' = shd (sdrop i  $\pi$ ) length c' = length c
    using basicCycle-rotate-set[OF c-prop(1), of shd (sdrop i  $\pi$ ), OF hd-in] by blast

  hence lim-c':limitS  $\pi$  = set (butlast c')
    using c-prop unfolding basicCycle-def
    using Graph.basicCycle-set-eq c'-prop(1) c-prop(1)
    by blast

  have  $\bigwedge n.\ (sdrop\ i\ \pi) !! n = (butlast\ c') ! (n\ mod\ length\ (butlast\ c'))$ 
  subgoal for n proof (induct n rule: nat-less-induct)
    fix n
    assume IH:  $\forall m < n.\ (sdrop\ i\ \pi) !! m = (butlast\ c') ! (m\ mod\ length\ (butlast\ c'))$ 

```

$c')$
show $(sdrop\ i\ \pi) !! n = (butlast\ c') ! (n\ mod\ length\ (butlast\ c'))$
proof $(cases\ n)$
 case 0
 then show $?thesis\ using\ c'\text{-prop}$
 by $(metis\ List.set\ empty\ assms(2,3)\ basicCycle\ set\ eq\ c\text{-prop}(2)\ hd\ conv\ nth$
 $length\ greater\ 0\ conv\ limitS\ ne\ mod\ less\ nth\ butlast\ snth.simps(1))$
next
 case $(Suc\ k)$
 have $c'\text{-eq}: butlast\ c' ! (n\ mod\ length\ (butlast\ c')) = c' ! Suc\ (k\ mod\ length$
 $(butlast\ c'))$
 by $(metis\ Graph.basicCycle\ set\ eq\ Suc\ c'\text{-prop}(1,3,4)\ hd\ conv\ nth\ last\ conv\ nth$
 $length\ butlast\ length\ greater\ 0\ conv\ list.size(3)\ mod\ Suc\ mod\ less\ divisor$
 $nth\ butlast\ set\ empty2)$
 also have $k\ le: k\ mod\ length\ (butlast\ c') < length\ c' - 1\ using\ Suc$
 $length\ butlast$
 by $(metis\ Graph.ipath\ iff\ snth\ c\text{-prop}(2)\ final\ ipath\ length\ butlast$
 $length\ pos\ if\ in\ set\ lim\ c'\ mod\ less\ divisor\ numeral\ nat(7))$

 have $drop\ in: (sdrop\ i\ \pi) !! n \in set\ (butlast\ c')$
 by $(metis\ Graph.ipath\ iff\ snth\ c\text{-prop}(2)\ final\ ipath\ lim\ c'\ sdrop\ snth)$

 then have $sdrop\ eq: (sdrop\ i\ \pi) !! k = (butlast\ c') ! (k\ mod\ length\ (butlast$
 $c'))$
 using $IH\ Suc\ by\ auto$

 then have $(limitR\ \pi) ((sdrop\ i\ \pi) !! k) ((sdrop\ i\ \pi) !! Suc\ k)$
 using $final\ ipath\ Graph.ipath\ iff\ snth\ by\ blast$

 moreover have $(limitR\ \pi) ((butlast\ c') ! (k\ mod\ length\ (butlast\ c')))$
 $((butlast\ c') ! (Suc\ k\ mod\ length\ (butlast\ c')))$
 using $limitS\ limitR\ edges[OF\ assms(1,2,3)\ lim\ c'\ c'\text{-prop}(1)]$
 by $(metis\ Suc\ calculation(1)\ k\ le\ length\ butlast\ nth\ butlast)$

 ultimately show $?thesis\ unfolding\ c'\text{-eq}$
 apply $\text{--} apply(rule\ basicCycle\ unique\ successor[OF\ assms(1)\ c'\text{-prop}(1),$
 $of\ (k\ mod\ length\ (butlast\ c'))\ (sdrop\ i\ \pi !! n))$
 subgoal by $(metis\ assms(2,3)\ length\ butlast\ length\ greater\ 0\ conv\ lim\ c'$
 $limitS\ ne\ list.set(1)\ mod\ less\ divisor)$
 subgoal using $limitR\ R[OF\ assms(3)]\ sdrop\ eq\ by\ (metis\ Suc\ k\ le$
 $length\ butlast\ nth\ butlast)$
 subgoal using $drop\ in\ by\ auto\ .$
 qed
 qed .

 hence $cyc\ repeat: srepeat\ (butlast\ c') = sdrop\ i\ \pi$

```

    by (metis Graph.basicCycle-set-eq c'-prop(1,2) hd-in list.set(1) set-nemp
        snth-equalityI srepeat-snth)

  show ?thesis apply(rule exI[of - stake i  $\pi$ ], intro exI[of - butlast c'] conjI)
  subgoal unfolding cyc-repeat by simp
  subgoal using basicCycle-ge2[OF c'-prop(1)] c'-prop(1)[unfolded basicCycle-
    cle-defs]
  by (simp add: c'-prop(1) hd-butlast le-simps(3) numeral-2-eq-2) .
qed

end

context Sloped-Graph
begin

definition cycleDescends ndl  $\equiv (\exists n\ Pl.\ n \neq 0 \wedge wfLabL\ (repeat\ n\ (butlast\ ndl))$ 
  @ [last ndl]) Pl  $\wedge$ 
  descentPath (repeat n (butlast ndl) @ [last ndl])
  Pl  $\wedge$  hd Pl = last Pl)

lemma cycle-descentIPathS-imp-cycleDescends:
assumes ndl: cycleG ndl and d: descentIPathS (srepeat (butlast ndl)) Ps
shows cycleDescends ndl
  using cycle-descentIPathS-srepeat-imp-descentPath-repeat[OF assms]
  by (auto simp: cycleDescends-def)

lemma cycle-descentIPath-imp-cycleDescends:
assumes ndl: cycleG ndl and d: descentIPath (srepeat (butlast ndl)) Ps
shows cycleDescends ndl
  using srepeat-cycle-descentIPath-imp-descentIPath[OF assms]
  cycle-descentIPathS-imp-cycleDescends[OF assms(1)] by auto

lemma cycleDescends-imp-descentIPathS:
assumes ndl: cycleG ndl and desc: cycleDescends ndl
shows  $\exists Ps.\ descentIPathS\ (srepeat\ (butlast\ ndl))\ Ps$ 
  using assms(2)[unfolded cycleDescends-def]
  apply-apply(clarify)
  using cycle-descentPath-repeat-imp-descentIPathS-srepeat[OF assms(1)] by auto

definition SimplyDescendingGraph :: bool where SimplyDescendingGraph  $\equiv \forall\ ndl.\$ 
  basicCycle ndl  $\longrightarrow$  cycleDescends ndl

lemma SimplyDescendingGraphD: SimplyDescendingGraph  $\Longrightarrow$  basicCycle c  $\Longrightarrow$ 
  cycleDescends c

```

unfolding *SimplyDescendingGraph-def* **by** *auto*

lemma *SimplyDescendingGraphI*: $(\bigwedge c. \text{ipath } (\text{srepeat } (\text{butlast } c)) \implies \text{basicCycle } c \implies \text{cycleDescends } c) \implies \text{SimplyDescendingGraph}$

unfolding *SimplyDescendingGraph-def* *basicCycle-def*

by (*simp add: cycle-srepeat-ipath*)

lemma *SimplyDescendingGraphI'*: $(\bigwedge c. \text{basicCycle } c \implies \text{cycleDescends } c) \implies \text{SimplyDescendingGraph}$

unfolding *SimplyDescendingGraph-def* *basicCycle-def* **by** *auto*

lemma *allCyclesDescendI*: $(\bigwedge c. \text{ipath } (\text{srepeat } (\text{butlast } c)) \implies \text{basicCycle } c \implies (\exists n \text{ Pl. } n \neq 0 \wedge \text{wfLabL } (\text{repeat } n \text{ (butlast } c) @ [\text{last } c]) \text{ Pl } \wedge \text{descentPath } (\text{repeat } n \text{ (butlast } c) @ [\text{last } c]) \text{ Pl } \wedge \text{hd Pl} = \text{last Pl})) \implies \text{SimplyDescendingGraph}$

unfolding *SimplyDescendingGraph-def* *cycleDescends-def*

using *basicCycle-def*

by (*simp add: cycle-srepeat-ipath*)

proposition *InfiniteDescent-implies-SimplyDescendingGraph*:
InfiniteDescent $\implies$ *SimplyDescendingGraph*

unfolding *SimplyDescendingGraph-def* *InfiniteDescent-def*

using *cycle-srepeat-ipath* *cycle-descentIPath-imp-cycleDescends* *basicCycle-def* **by** *blast*

proposition *SimplyDescendingUnicyclesGraph-implies-InfiniteDescent*:
assumes *unicyclesGraph*
shows *SimplyDescendingGraph* $\implies$ *InfiniteDescent*

proof(*rule InfiniteDescentI*)

fix *nds*

assume *SDG*: *SimplyDescendingGraph* **and**
ipath:ipath nds

obtain *v u* **where** *nds:nds = v @- srepeat u* **and** *cyc:basicCycle (u @ [hd u])*

using *unicycle-lasso[OF assms Node-finite ipath]* **by** *auto*

then obtain *Ps* **where** *descentS:descentIPathS (srepeat u) Ps*

using *SDG[unfolded SimplyDescendingGraph-def]*

apply—apply(*elim allE[of - u @ [hd u]] impE exE conjE, assumption*)

using *cycleDescends-imp-descentIPathS[of u @ [hd u]]* *basicCycle-def* **by** *auto*

hence *descent:descentIpath (srepeat u) Ps* **using** *descentIpathS-imp-descentIpath*
by *auto*

then obtain *Ps'* **where** *Ps':descentIpath (v @- srepeat u) Ps'*
using *descentIpath-sdrop-any[of length v, of v @- srepeat u Ps]*
unfolding *sdrop-shift-length'* **by** *auto*

thus $\exists Ps. \text{descentIpath nds } Ps$ **unfolding** *nds* **by** *auto*
qed

theorem *DescendingUnicyclesCriterion:*
assumes *unicyclesGraph*
shows *InfiniteDescent $\longleftrightarrow$ SimplyDescendingGraph*
using *InfiniteDescent-implies-SimplyDescendingGraph*
SimplyDescendingUnicyclesGraph-implies-InfiniteDescent[OF assms]
by *auto*

end

end

4.7 All Criterion

theory *All* **imports**

Flat-Cycles-Criterion
Descending-Unicycles-Criterion
Incomplete-Criteria
SLA-Criterion
VLA-Criterion
Relation-Based-Criterion

begin

context *Sloped-Graph*
begin
thm *Flat-Cycles-Criterion*
DescendingUnicyclesCriterion
Incomplete-Criterion
SLA-Criterion
VLA-Criterion
Relation-Based-Criterion
Relation-Based-Criterion'

end

end

5 Instantiating the Abstract Framework

We now provide concrete examples of this framework in action applying a range of the criterion shown

5.1 Infinite Descent Examples

5.1.1 Flat Aux Example

```
theory Flat-Aux
imports ../Criterion/All
begin

fun flat and aux where
  flat [] = []
| flat (l#ls) = aux l ls
| aux [] ls = flat ls
| aux (x#xs) ls = x # aux xs ls

datatype node = Flat | Aux
type-synonym pos = nat

definition Node  $\equiv \{Flat, Aux\}$ 

lemma nd-aux[simp]:  $nd \neq Aux \longleftrightarrow nd = Flat$  by(cases nd, auto)
lemma nd-flat[simp]:  $nd \neq Flat \longleftrightarrow nd = Aux$  by(cases nd, auto)

lemma alw-nodes:alw (holdsS Node) nds
  unfolding alw-holdsS-iff-snth Node-def apply(rule allI)
  subgoal for i by(induct i, auto) .

fun edge::node  $\Rightarrow$  node  $\Rightarrow$  bool where
  edge Flat Aux = HOL.True|
  edge Aux - = HOL.True|
  edge Flat Flat = HOL.False

lemma edge-Flat[simp]:  $edge\ Flat\ x \longleftrightarrow x = Aux$  by(cases x, auto)

lemma edge-Flat'[simp]:  $edge\ nd\ Flat \longleftrightarrow nd = Aux$  by(cases nd, auto)

fun PosOf::node  $\Rightarrow$  pos set where
```

$PosOf\ Flat = \{0\}$
 $PosOf\ Aux = \{0,1\}$

definition $RR\text{-}set :: ((node \times pos) \times (node \times pos) \times slope) \text{ set where}$

$RR\text{-}set = \{$
 $((Flat, 0), (Aux, 0), Decr),$
 $((Flat, 0), (Aux, 1), Decr),$
 $((Aux, 1), (Flat, 0), Main),$
 $((Aux, 0), (Aux, 0), Decr),$
 $((Aux, 1), (Aux, 1), Main)$
 $\}$

definition $RR :: node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool \text{ where}$

$RR\ np1\ np2\ s \equiv ((np1, np2), s) \in RR\text{-}set)$

definition $RRs :: node \Rightarrow node \Rightarrow (pos \Rightarrow pos \Rightarrow slope \Rightarrow bool) \text{ where}$

$RRs\ n\ n' \equiv (\lambda p\ p'\ s. (((n, p), (n, p')), s) \in RR\text{-}set))$

lemmas $RRs\text{-}defs = RRs\text{-}def\ RR\text{-}set\text{-}def$

lemmas $RR\text{-}defs = RR\text{-}def\ RR\text{-}set\text{-}def$

lemma $RR\text{-}aux\text{-}aux\text{-}1[simp]: RR\ (Aux, 0)\ (Aux, 0)\ Decr \text{ unfolding } RR\text{-}defs \text{ by } auto$

lemma $P\text{-}inPosOf: RR\ (nd, P)\ (nd', P')\ sl \implies P \in PosOf\ nd$

$RR\ (nd, P)\ (nd', P')\ sl \implies P' \in PosOf\ nd' \text{ by } (auto\ simp: RR\text{-}defs)$

interpretation $Sloped\text{-}Graph \text{ where}$

$Node = Node \text{ and } edge = edge \text{ and } PosOf = PosOf \text{ and } RR = RR$

apply $unfold\text{-}locales$

by $(auto\ simp: RR\text{-}defs\ SlopedRels\text{-}def\ Node\text{-}def\ elim: PosOf.cases)$

lemma $lang\text{-}run\text{-}inj: x \in NBA.language\ Paut_V \implies \exists r. NBA.run\ Paut_V\ (x \parallel r)$

$None$

by $(erule\ nba.language\text{-}elim, auto)$

end

5.1.2 Flat Aux via Vertex-Language Automaton

theory $Flat\text{-}Aux\text{-}VLA$

imports $Flat\text{-}Aux$

begin

lemma $InfiniteDescentI: (\bigwedge nds. alw\ (holds2\ edge)\ nds \implies \exists x. (descentIpath\ nds))$

$\Rightarrow$ *InfiniteDescent*
unfolding *VLA-Criterion Paut_V-lang Taut_V-lang ipath-def* **by** *auto*

lemma *InfiniteDescent*

proof(*rule InfiniteDescentI*)
fix *x*
assume *alw (holds2 edge) x*
hence *edge-always: $\bigwedge i. \text{edge } (x !! i) (x !! \text{Suc } i)$*
and *Node-always: $\bigwedge i. (x !! i) \in \text{Node}$*
using *alw-nodes*
unfolding *alw-holds2-iff-snth alw-holdsS-iff-snth* **by** *auto*

have *flat-trans-is-aux: $\bigwedge i. x !! i = \text{Flat} \Rightarrow \text{stl } x !! i = \text{Aux}$*
subgoal for *i* **using** *edge-always[of i]* **by** (*auto*) .

define *Pss::node stream $\Rightarrow$ nat $\Rightarrow$ nat* **where**
Pss = ($\lambda x n. (\text{if } x !! n = \text{Flat} \text{ then } 0 \text{ else } 1)$)

define *Ps* **where** *Ps:Ps = fToStream (Pss x)*
note *Ps-defs = Ps[unfolded Pss-def fToStream-def]*

show *Ex (descentIpath x)*

proof(*cases infs ($\lambda x. x = \text{Flat}$) x*)
case *True*

show *?thesis*
apply(*rule exI[of - Ps], unfold descentIpath-iff-snth*)
apply(*rule exI[of - 0], intro conjI allI impI*)

subgoal for *j* **apply**(*cases x !! j*)
subgoal apply(*cases x !! Suc j*)
using *edge-always[of j]* **by** (*auto simp: RR-defs Ps-defs*)
by(*cases x !! Suc j, auto simp: RR-defs Ps-defs*)

using *True flat-trans-is-aux* **unfolding** *alw-ev-holds-iff-snth*
by(*auto simp: RR-defs Ps-defs*)

next

case *False*
then consider (*alw-aux*) *alw (holds ($\lambda x. x = \text{Aux}$) x*
 $| (\text{some-flat}) \exists i. \forall k > i. (x !! i) = \text{Flat} \wedge (x !! k) = \text{Aux}$
apply—**by**(*drule fins-imp, auto*)

thus *?thesis* **proof**(*cases*)

```

case alw-aux
hence alw-aux:  $\bigwedge i. x !! i = Aux$  unfolding alw-holds-iff-snth by auto
also have alw-aux':  $\bigwedge i. stl x !! i = Aux$  subgoal for  $i$  using calculation[of
Suc i] by auto .

show ?thesis
  apply(rule exI[of - sconst 0], unfold descentIpath-iff-snth)
  apply(rule exI[of - 0], intro conjI allI impI)
  using alw-aux alw-aux' by auto
next
case some-flat
then obtain  $i$  where  $i: \forall n > i. x !! n = Aux$   $x !! i = Flat$  by blast

hence always-aux:  $\bigwedge j. j \geq Suc\ 0 \implies x !! (i + j) = Aux$ 
                  $\bigwedge j. j \geq Suc\ 0 \implies stl x !! (i + j) = Aux$ 
by (auto,metis less-add-Suc1 stl-Suc-eq)

have flat-aux-not-i-le:
 $\bigwedge j. x !! j = Flat \implies x !! Suc\ j = Aux \implies j \neq i \implies Suc\ j < i \wedge j < i$ 
by (metis Suc-lessI i(1,2) linorder-neqE-nat node.simps(2))

have aux-flat-not-i-le:
 $\bigwedge j. x !! j = Aux \implies x !! Suc\ j = Flat \implies j < i$ 
by (metis i(1) node.simps(2) not-less-eq)

have aux-ngr:  $\bigwedge j. x !! j = Aux \implies x !! Suc\ j = Aux \implies \neg i < j \implies Suc\ j$ 
 $< i \wedge j < i$ 
by (metis Suc-lessI i(2) linorder-neqE-nat node.distinct(1))

define newPs where newPs: newPs = stake  $i$  Ps @- sconst 0
note newPs-defs = newPs[unfolded Ps-defs]

hence always-1:  $\bigwedge j. j \geq Suc\ 0 \implies newPs !! (i + j) = 0$ 
                $\bigwedge j. j \geq Suc\ 0 \implies stl newPs !! (i + j) = 0$  by auto

also have newPs-le:  $\bigwedge j. j < i \implies newPs !! j = (stake\ i\ Ps) ! j$ 
by (simp add: newPs)

have newPs-gr:  $\bigwedge j. i < j \implies newPs !! j = 0$  unfolding newPs by auto

show ?thesis
  apply(rule exI[of - newPs], unfold descentIpath-iff-snth)
  apply(rule exI[of - Suc 0], intro conjI allI impI)

  subgoal for  $j$  apply(cases  $x !! j$ )
    subgoal apply(cases  $x !! Suc\ j$ )

```

```

subgoal using edge-always[of j] by auto

apply(cases j = i)
subgoal by (auto simp: RR-defs newPs-defs)

apply(frule flat-aux-not-i-le[of j], safe)
apply(frule newPs-le[of j], frule newPs-le[of Suc j])
using nth-map-upt by (auto simp: RR-defs Ps-defs)

subgoal apply(cases x !! Suc j)

subgoal apply(frule aux-flat-not-i-le, safe)
apply(frule newPs-le[of j], drule Suc-disj)
apply (erule disjE, frule newPs-le[of Suc j])
subgoal using nth-map-upt by (auto simp: RR-defs Ps-defs)
subgoal by (auto simp: RR-defs newPs-defs) .

apply(cases i < j)

subgoal using newPs-gr newPs-gr[of Suc j] by auto

subgoal apply(frule aux-ngr, safe)
apply(frule newPs-le[of j], frule newPs-le[of Suc j])
using nth-map-upt by (auto simp: RR-defs Ps-defs) . .

subgoal for j apply(rule exI[of - i + j]) using always-aux always-1 by
(auto simp: RR-defs) .

qed

qed
qed

end

```

5.1.3 Flat Aux via Sloped-Language Automaton

```

theory Flat-Aux-SLA
imports Flat-Aux
begin

```

```

lemma InfiniteDescentI: ( $\bigwedge nds. alw (holds2 \text{ edge}) \ nds \implies \exists x (descentIpath \ nds)$ )
 $\implies InfiniteDescent$ 
unfolding SLA-Criterion PautR-lang Rst-correct[symmetric]
using ipath-def TautR-lang-in by auto

```

lemma *InfiniteDescent*

proof(*rule InfiniteDescentI*)

fix x
 assume $alw\ (holds2\ edge)\ x$
 hence $edge-always:\bigwedge i. edge\ (x\ !!\ i)\ (x\ !!\ Suc\ i)$
 and $Node-always:\bigwedge i. (x\ !!\ i) \in Node$
 using $alw-nodes$
 unfolding $alw-holds2-iff-snth\ alw-holdsS-iff-snth$ **by** $auto$

have $flat-trans-is-aux:\bigwedge i. x\ !!\ i = Flat \implies stl\ x\ !!\ i = Aux$
 subgoal for i **using** $edge-always[of\ i]$ **by** $(auto)$.

define $Pss::node\ stream \Rightarrow nat \Rightarrow nat$ **where**

$Pss = (\lambda x\ n. (if\ x\ !!\ n = Flat\ then\ 0\ else\ 1))$

define Ps **where** $Ps:Ps = fToStream\ (Pss\ x)$

note $Ps-defs = Ps[unfolded\ Pss-def\ fToStream-def]$

show $Ex\ (descentIpath\ x)$

proof(*cases infs* $(\lambda x. x = Flat)\ x$)

case $True$

show *?thesis*

apply(*rule exI*[*of* - Ps], *unfold descentIpath-iff-snth*)

apply(*rule exI*[*of* - 0], *intro conjI allI impI*)

subgoal for j **apply**(*cases* $x\ !!\ j$)

subgoal apply(*cases* $x\ !!\ Suc\ j$)

using $edge-always[of\ j]$ **by** $(auto\ simp: RR-defs\ Ps-defs)$

by(*cases* $x\ !!\ Suc\ j$, *auto simp: RR-defs Ps-defs*)

using $True\ flat-trans-is-aux$ **unfolding** $alw-ev-holds-iff-snth$

by(*auto simp: RR-defs Ps-defs*)

next

case $False$

then consider $(alw-aux)\ alw\ (holds\ (\lambda x. x = Aux))\ x$

$| (some-flat)\ \exists i. \forall k > i. (x\ !!\ i) = Flat \wedge (x\ !!\ k) = Aux$

apply-by(*drule fins-imp, auto*)

thus *?thesis* **proof**(*cases*)

case $alw-aux$

hence $alw-aux:\bigwedge i. x\ !!\ i = Aux$ **unfolding** $alw-holds-iff-snth$ **by** $auto$

also have $alw-aux':\bigwedge i. stl\ x\ !!\ i = Aux$ **subgoal for** i **using** $calculation[of\ Suc\ i]$ **by** $auto$.

```

show ?thesis
  apply(rule exI[of - sconst 0], unfold descentIpath-iff-snth)
  apply(rule exI[of - 0], intro conjI allI impI)
  using alw-aux alw-aux' by auto
next
case some-flat
then obtain i where i:  $\forall n > i. x !! n = Aux\ x !! i = Flat$  by blast

hence always-aux:  $\bigwedge j. j \geq Suc\ 0 \implies x !! (i + j) = Aux$ 
                $\bigwedge j. j \geq Suc\ 0 \implies stl\ x !! (i + j) = Aux$ 
  by (auto, metis less-add-Suc1 stl-Suc-eq)

have flat-aux-not-i-le:
   $\bigwedge j. x !! j = Flat \implies x !! Suc\ j = Aux \implies j \neq i \implies Suc\ j < i \wedge j < i$ 
  by (metis Suc-lessI i(1,2) linorder-neqE-nat node.simps(2))

have aux-flat-not-i-le:
   $\bigwedge j. x !! j = Aux \implies x !! Suc\ j = Flat \implies j < i$ 
  by (metis i(1) node.simps(2) not-less-eq)

have aux-ngr:  $\bigwedge j. x !! j = Aux \implies x !! Suc\ j = Aux \implies \neg i < j \implies Suc\ j$ 
< i  $\wedge j < i$ 
  by (metis Suc-lessI i(2) linorder-neqE-nat node.distinct(1))

define newPs where newPs: newPs = stake i Ps @- sconst 0
note newPs-defs = newPs[unfolded Ps-defs]

hence always-1:  $\bigwedge j. j \geq Suc\ 0 \implies newPs !! (i + j) = 0$ 
                $\bigwedge j. j \geq Suc\ 0 \implies stl\ newPs !! (i + j) = 0$  by auto

also have newPs-le:  $\bigwedge j. j < i \implies newPs !! j = (stake\ i\ Ps) ! j$ 
  by (simp add: newPs)

have newPs-gr:  $\bigwedge j. i < j \implies newPs !! j = 0$  unfolding newPs by auto

show ?thesis
  apply(rule exI[of - newPs], unfold descentIpath-iff-snth)
  apply(rule exI[of - Suc 0], intro conjI allI impI)

  subgoal for j apply(cases x !! j)
    subgoal apply(cases x !! Suc j)

      subgoal using edge-always[of j] by auto

      apply(cases j = i)
      subgoal by (auto simp: RR-defs newPs-defs)

```



```

apply(frule flat-aux-not-i-le[of j], safe)
apply(frule newPs-le[of j], frule newPs-le[of Suc j])
using nth-map-upt by (auto simp: RR-defs Ps-defs)

subgoal apply(cases x !! Suc j)

  subgoal apply(frule aux-flat-not-i-le, safe)
  apply(frule newPs-le[of j], drule Suc-disj)
  apply (erule disjE, frule newPs-le[of Suc j])
  subgoal using nth-map-upt by (auto simp: RR-defs Ps-defs)
  subgoal by (auto simp: RR-defs newPs-defs) .

apply(cases i < j)

subgoal using newPs-gr newPs-gr[of Suc j] by auto

subgoal apply(frule aux-ngr, safe)
apply(frule newPs-le[of j], frule newPs-le[of Suc j])
using nth-map-upt by (auto simp: RR-defs Ps-defs) . .

subgoal for j apply(rule exI[of - i + j]) using always-aux always-1 by
(auto simp: RR-defs) .

  qed

  qed
qed

end

```

5.1.4 Descending Unicycles Example

```

theory Descending-Unicycles-Example
  imports ../Criterion/All
begin

```

```

datatype node = Zero | One | Two | Three
datatype pos = p0 | p1 | p2 | p2' | p3 | p3'

```

```

definition Node  $\equiv$  {Zero, One, Two, Three}

```

```

lemma Z-Node[simp]: Zero  $\in$  Node by(simp add: Node-def)
lemma O-Node[simp]: One  $\in$  Node by(simp add: Node-def)
lemma T-Node[simp]: Two  $\in$  Node by(simp add: Node-def)

```

lemma *Th-Node*[simp]: *Three* ∈ *Node* **by**(*simp add: Node-def*)

lemma *alw-nodes:alw* (*holdsS Node*) *nds*
unfolding *alw-holdsS-iff-snth Node-def* **apply**(*rule allI*)
subgoal for i **apply**(*induct i*)
using *node.exhaust* **by** *auto* .

fun *edge::node ⇒ node ⇒ bool* **where**
edge Zero One = *HOL.True* |
edge One Zero = *HOL.True* |
edge Zero Two = *HOL.True* |
edge Two Three = *HOL.True* |
edge Three Two = *HOL.True* |
edge - - = *HOL.False*

lemma *edge-into-zero*[simp]: *edge nd Zero* $\longleftrightarrow$ *nd* = *One* **by**(*cases nd, auto*)
lemma *edge-into-one*[simp]: *edge nd One* $\longleftrightarrow$ *nd* = *Zero* **by**(*cases nd, auto*)
lemma *edge-into-three*[simp]: *edge nd Three* $\longleftrightarrow$ *nd* = *Two* **by**(*cases nd, auto*)
lemma *edge-three-into*[simp]: *edge Three nd* $\longleftrightarrow$ *nd* = *Two* **by**(*cases nd, auto*)
lemma *edge-two-into*[simp]: *edge Two nd* $\longleftrightarrow$ *nd* = *Three* **by**(*cases nd, auto*)
lemma *edge-zero-into*[simp]: *edge Zero nd* $\longleftrightarrow$ *nd* = *One* $\vee$ *nd* = *Two* **by**(*cases nd, auto*)

fun *PosOf::node ⇒ pos set* **where**
PosOf Zero = {*p0*} |
PosOf One = {*p1*} |
PosOf Two = {*p2*, *p2'*} |
PosOf Three = {*p3*, *p3'*}

definition *RR-set* :: ((*node* × *pos*) × (*node* × *pos*) × *slope*) *set* **where**
RR-set = {
 ((*Zero*, *p0*), (*One*, *p1*), *Decr*),

 ((*One*, *p1*), (*Zero*, *p0*), *Main*),

 ((*Zero*, *p0*), (*Two*, *p2*), *Main*),
 ((*Zero*, *p0*), (*Two*, *p2'*), *Main*),

 ((*Two*, *p2*), (*Three*, *p3*), *Main*),
 ((*Two*, *p2'*), (*Three*, *p3'*), *Main*),

 ((*Three*, *p3*), (*Two*, *p2*), *Decr*),

```

    ((Three, p3'), (Two, p2'), Main)
  }

```

definition $RR :: node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool$ **where**
 $RR\ np1\ np2\ s \equiv ((np1, np2, s) \in RR\text{-}set)$

lemmas $RR\text{-}defs = RR\text{-}def\ RR\text{-}set\text{-}def$

lemma $RR\text{-}ZO[simp]: RR\ (Zero, p0)\ (node.One, p1)\ Decr$ **unfolding** $RR\text{-}defs$ **by** $auto$

lemma $RR\text{-}OZ[simp]: RR\ (node.One, p1)\ (Zero, p0)\ Main$ **unfolding** $RR\text{-}defs$ **by** $auto$

lemma $RR\text{-}TTr[simp]: RR\ (Two, p2)\ (Three, p3)\ Main$ **unfolding** $RR\text{-}defs$ **by** $auto$

lemma $RR\text{-}TrT[simp]: RR\ (Three, p3)\ (Two, p2)\ Decr$ **unfolding** $RR\text{-}defs$ **by** $auto$

lemma $P\text{-}inPosOf: RR\ (nd, P)\ (nd', P')\ sl \implies P \in PosOf\ nd$
 $RR\ (nd, P)\ (nd', P')\ sl \implies P' \in PosOf\ nd'$ **by** $(auto\ simp: RR\text{-}defs)$

interpretation $Sloped\text{-}Graph$ **where**

$Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$

and $RR = RR$ **apply** $standard$

subgoal **by** $(simp\ add: Node\text{-}def)$

subgoal **by** $(unfold\ Node\text{-}def,\ auto\ elim: PosOf.cases)$

by $(auto\ simp: RR\text{-}defs\ SlopedRels\text{-}def\ Node\text{-}def)$

lemma $notTZ: \neg pathG\ [node.Two, Zero]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.Two, Zero]],\ auto)$

lemma $notTO: \neg pathG\ [node.Two, One]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.Two, One]],\ auto)$

lemma $notTrZ: \neg pathG\ [node.Three, Zero]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.Three, Zero]],\ auto)$

lemma $notTrO: \neg pathG\ [node.Three, One]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.Three, One]],\ auto)$

lemma $notOT: \neg pathG\ [node.One, node.Two]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.One, node.Two]],\ auto)$

lemma $notOTr: \neg pathG\ [node.One, node.Three]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [node.One, node.Three]],\ auto)$

lemma $notZTr: \neg pathG\ [Zero, node.Three]$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [Zero, node.Three]],\ auto)$

lemma $notZTZ: \neg pathG\ ([Zero, node.Two, Zero])$ **by** $(rule\ notI,\ cases\ rule: pathG.cases[of\ [Zero, node.Two, Zero]],\ auto)$

lemma $pathNWF\text{-}Three: pathG\ p \implies hd\ p = Three \implies last\ p \neq Zero \wedge last\ p \neq$

```

One
  apply(induct rule: pathG.induct)
  subgoal by auto
  subgoal for nd ndl apply (cases ndl, simp)
    subgoal for l ls by(cases last ndl, simp-all split: if-splits) . .

lemma pathNWF-Two:pathG p  $\implies$  hd p = Two  $\implies$  last p  $\neq$  Zero  $\wedge$  last p  $\neq$  One
  apply(induct rule: pathG.induct)
  subgoal by auto
  subgoal for nd ndl apply (cases ndl, simp)
    subgoal for l ls by(cases last ndl, simp-all split: if-splits) . .

lemma pathNWF-disj:pathG pa  $\implies$ 
  hd pa = node.Two  $\vee$  hd pa = node.Three  $\implies$ 
  last pa = Zero  $\vee$  last pa = node.One  $\implies$  HOL.False
  apply(elim disjE) using pathNWF-Two pathNWF-Three by auto

lemma ZZ-FlatEdge: $\neg$ (Zero, Zero)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)
lemma OO-FlatEdge: $\neg$ (One, One)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)
lemma TT-FlatEdge: $\neg$ (Two, Two)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)
lemma ThTh-FlatEdge: $\neg$ (Three, Three)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)
lemmas same-FlatEdge = ZZ-FlatEdge OO-FlatEdge TT-FlatEdge ThTh-FlatEdge

lemma ZO-FlatEdge: $\neg$ (Zero, One)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)
lemma ThT-FlatEdge: $\neg$ (Three, Two)  $\in$  FlatEdges unfolding FlatEdges-def by (simp add: RR-defs)

lemma aux-ls:xs  $\neq$  []  $\implies$  (xs @ [Zero]) ! Suc 0 = node.Three  $\implies$  length xs > 1
by(induct xs, auto)

theorem  $\neg$ FlatCycle
  apply(rule notFlatCycleI, frule cycleG-shape, elim exE disjE)
  subgoal for - x using same-FlatEdge by(cases x, auto)
  subgoal for nds x xs apply(cases x)

    subgoal using path-nth-edge[of Zero # xs @ [Zero] 0]
      apply(cases xs ! 0)
      subgoal by (auto simp add: cycleG-def)
      subgoal by(erule allE[of - 0], auto simp add: cycleG-def ZO-FlatEdge)
      subgoal using pathNWF-Two[of xs @ [Zero]]
        pathG-tl[of xs @ [Zero] Zero]

```

```

      hd-conv-nth[of xs] by (auto simp add: cycleG-def)
subgoal by (auto simp add: cycleG-def) .

subgoal apply(erule allE[of - length nds - 2])
  using path-nth-edge[of One # xs @ [One] length nds - 2]
  by (simp add: cycleG-def ZO-FlatEdge)

subgoal apply(erule allE[of - 1])
  using path-nth-edge[of Two # xs @ [Two] 0]
  using path-nth-edge[of Two # xs @ [Two] 1]
  by (simp add: cycleG-def ThT-FlatEdge)

subgoal apply(erule allE[of - 0])
  using path-nth-edge[of Three # xs @ [Three] 0]
  by (simp add: cycleG-def ThT-FlatEdge) . .

lemma i-disj:i < Suc (Suc (Suc 0))  $\longleftrightarrow$  i = 0  $\vee$  i = 1  $\vee$  i = 2 by auto

lemma i-disj':i < Suc (Suc 0)  $\longleftrightarrow$  i = 0  $\vee$  i = 1 by auto

lemma allCycles:basicCycle c  $\longleftrightarrow$  c = [Zero, One, Zero]  $\vee$  c = [One, Zero, One]
 $\vee$  c = [Two, Three, Two]  $\vee$  c = [Three, Two, Three]
apply(standard)
subgoal unfolding basicCycle-def[unfolded cycleG-def] apply(elim conjE)
  apply(cases rule: pathG.cases[of c], simp)
subgoal for nd by(cases nd, auto)
subgoal for nd'' ndl
  apply(cases ndl, simp)
subgoal for nd ndl' apply(cases ndl', simp)
subgoal for nd' ls
  apply(cases ls)
subgoal apply(cases nd)
  subgoal by(cases nd', auto)
  subgoal by(cases nd', auto)
  subgoal by(cases nd', auto simp: notTZ)
  subgoal by(cases nd', auto simp: notTZ) .

subgoal for nnd ls' apply(cases nd)

  subgoal apply(cases nd')
    subgoal by(cases nnd,cases nd'', auto)
    subgoal by(cases nnd,cases nd'', auto simp: notPathG-within' notOTr

```

```

notOT)
    subgoal apply(cases nnd,cases nd'', auto simp: notPathG-within'
notTO)
    by (smt (verit, ccfv-threshold) PosOf.cases
        butlast.simps(2) edge.elims(2) hd-last-singletonI
        last.simps list.distinct(1) list.set-sel(1)
        node.distinct(9) node.simps(4,6) pathG.cases
        snoc-eq-iff-butlast)

    subgoal apply(cases nnd,cases nd'', auto simp: notPathG-within'
notTrO notOT) using notZTr path-appendL[of [Zero, Three]] by auto .

    subgoal apply(cases nd')
    subgoal by(cases nnd,cases nd'', auto simp: notPathG-within' notZTr,
(metis List.last-in-set edge.elims(2) node.distinct(9) node.simps(4,8)))
    subgoal by(cases nnd,cases nd'', auto)
    subgoal by(cases nnd,cases nd'', auto simp: notPathG-within'
notTZ,metis append-Cons empty-append-eq-id notOT notPathG-within)
    subgoal apply(cases nnd,cases nd'', auto simp: notPathG-within'
notTrZ,metis append-Cons empty-append-eq-id notOTr notPathG-within) . .

    subgoal apply(cases nd')
    subgoal apply(cases nnd,cases nd'', auto) using notTZ path-appendL[of
[Two, Zero]] by auto
    subgoal apply(cases nnd,cases nd'', auto) using notTO path-appendL[of
[Two, One]] by auto
    subgoal by(cases nnd,cases nd'', auto simp: notPathG-within' notTZ)
    subgoal by(cases nnd,cases nd'', auto simp: notTrO notTrZ
notPathG-within') .

    subgoal apply(cases nd')
    subgoal apply(cases nnd,cases nd'', auto) using notTrZ path-appendL[of
[Three, Zero]] by auto
    subgoal apply(cases nnd,cases nd'', auto) using notTrO path-appendL[of
[Three, One]] by auto
    subgoal by(cases nnd,cases nd'', auto simp: notTZ notTO not-
PathG-within')
    subgoal by(cases nnd,cases nd'', auto simp: notTrZ notTrO
notPathG-within') . . . . .

    subgoal apply(elim disjE, simp-all add: basicCycle-def cycleG-def)
    subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
    subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
    subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
    subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base) . .

```

lemma *notConnectedCyclesI*: $(\bigwedge p. \text{pathG } p \implies \text{hd } p \in \text{set } c' \implies \text{last } p \in \text{set } c \implies \text{HOL.False}) \implies \neg \text{connectedCycles } c' \ c$

unfolding *connectedCycles-def* **by** *auto*

lemma *unicycle:unicyclesGraph*

apply(*rule unicyclesGraphI'*, *unfold allCycles connectedCycles-def*, *elim exE conjE disjE*)

subgoal **by** *simp*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

subgoal **using** *pathNWF-disj* **by** *auto*

lemma *SDG:SimplyDescendingGraph*

apply(*rule SimplyDescendingGraphI'*, *unfold allCycles cycleDescends-def*, *elim disjE*)

subgoal **for** *c* **apply**(*intro exI[of - 1] exI[of - [p0,p1,p0]] conjI, simp-all*)

subgoal **unfolding** *wfLabL-def* **by** (*clarsimp, unfold i-disj, force*)

subgoal **unfolding** *descentPath-def* **by**(*clarsimp, unfold i-disj', force*) .

subgoal **for** *c* **apply**(*intro exI[of - 1] exI[of - [p1,p0,p1]] conjI, simp-all*)

subgoal **unfolding** *wfLabL-def* **by** (*clarsimp, unfold i-disj, force*)

subgoal **unfolding** *descentPath-def* **by**(*clarsimp, unfold i-disj', force*) .

subgoal **for** *c* **apply**(*intro exI[of - 1] exI[of - [p2,p3,p2]] conjI, simp-all*)

subgoal **unfolding** *wfLabL-def* **by** (*clarsimp, unfold i-disj, force*)

subgoal **unfolding** *descentPath-def* **by**(*clarsimp, unfold i-disj', force*) .

subgoal **for** *c* **apply**(*intro exI[of - 1] exI[of - [p3,p2,p3]] conjI, simp-all*)

subgoal **unfolding** *wfLabL-def* **by** (*clarsimp, unfold i-disj, force*)

subgoal **unfolding** *descentPath-def* **by**(*clarsimp, unfold i-disj', force*) . .

theorem *InfiniteDescent*

apply(*unfold DescendingUnicyclesCriterion[OF unicycle]*)

using *SDG* **by** *auto*

end

5.2 Infinite Descent Counterexamples

5.2.1 Descending Unicycles

```
theory Descending-Unicycles-CounterExample
  imports ../Criterion/All
begin
```

```
datatype node = Zero | One | Two | Three
datatype pos = p0 | p1 | p2 | p2' | p3 | p3'
```

```
definition Node  $\equiv \{Zero, One, Two, Three\}$ 
```

```
lemma Z-Node[simp]: Zero  $\in$  Node by(simp add: Node-def)
lemma O-Node[simp]: One  $\in$  Node by(simp add: Node-def)
lemma T-Node[simp]: Two  $\in$  Node by(simp add: Node-def)
lemma Th-Node[simp]: Three  $\in$  Node by(simp add: Node-def)
```

```
lemma alw-nodes: alw (holdsS Node) nds
  unfolding alw-holdsS-iff-snth Node-def apply(rule allI)
  subgoal for i apply(induct i)
  using node.exhaust by auto .
```

```
fun edge::node  $\Rightarrow$  node  $\Rightarrow$  bool where
  edge Zero One = HOL.True|
  edge One Zero = HOL.True|
  edge Zero Two = HOL.True|
  edge Two Three = HOL.True|
  edge Three Two = HOL.True|
  edge - - = HOL.False
```

```
lemma edge-into-zero: edge nd Zero  $\longleftrightarrow$  nd = One by(cases nd, auto)
```

```
fun PosOf::node  $\Rightarrow$  pos set where
  PosOf Zero = {p0}|
  PosOf One = {p1}|
  PosOf Two = {p2, p2'}|
  PosOf Three = {p3, p3'}
```

```
definition RR-set :: ((node  $\times$  pos)  $\times$  (node  $\times$  pos)  $\times$  slope) set where
  RR-set = {
```



```

((Zero, p0), (One, p1), Decr),

((One, p1), (Zero, p0), Main),

((Zero, p0), (Two, p2), Main),
((Zero, p0), (Two, p2'), Main),

((Two, p2), (Three, p3), Main),
((Two, p2'), (Three, p3'), Main),

((Three, p3), (Two, p2), Main),

((Three, p3'), (Two, p2'), Main)
}

```

definition $RR :: node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool$ **where**
 $RR\ np1\ np2\ s \equiv ((np1, np2, s) \in RR\text{-}set)$

lemmas $RR\text{-}defs = RR\text{-}def\ RR\text{-}set\text{-}def$

lemma $RR\text{-}ZO[simp]: RR\ (Zero, p0)\ (node.One, p1)\ Decr$ **unfolding** $RR\text{-}defs$ **by** *auto*

lemma $RR\text{-}OZ[simp]: RR\ (node.One, p1)\ (Zero, p0)\ Main$ **unfolding** $RR\text{-}defs$ **by** *auto*

lemma $RR\text{-}TTr[simp]: RR\ (Two, p2)\ (Three, p3)\ Main$ **unfolding** $RR\text{-}defs$ **by** *auto*

lemma $RR\text{-}TrT[simp]: RR\ (Three, p3)\ (Two, p2)\ Main$ **unfolding** $RR\text{-}defs$ **by** *auto*

lemma $P\text{-}inPosOf: RR\ (nd, P)\ (nd', P')\ sl \implies P \in PosOf\ nd$
 $RR\ (nd, P)\ (nd', P')\ sl \implies P' \in PosOf\ nd'$ **by** (*auto simp: RR-defs*)

interpretation *Sloped-Graph* **where**

$Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$

and $RR = RR$ **apply** *standard*

subgoal **by** (*simp add: Node-def*)

subgoal **by** (*unfold Node-def, auto elim: PosOf.cases*)

by (*auto simp: RR-defs SlopedRels-def Node-def*)

lemma $i\text{-}disj: i < Suc\ (Suc\ (Suc\ 0)) \longleftrightarrow i = 0 \vee i = 1 \vee i = 2$ **by** *auto*

lemma $i\text{-}disj': i < Suc\ (Suc\ 0) \longleftrightarrow i = 0 \vee i = 1$ **by** *auto*

lemma *notTZ*: $\neg$ *pathG* [*node.Two*, *Zero*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.Two*, *Zero*]], *auto*)
lemma *notTO*: $\neg$ *pathG* [*node.Two*, *One*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.Two*, *One*]], *auto*)
lemma *notTrZ*: $\neg$ *pathG* [*node.Three*, *Zero*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.Three*, *Zero*]], *auto*)
lemma *notTrO*: $\neg$ *pathG* [*node.Three*, *One*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.Three*, *One*]], *auto*)
lemma *notOT*: $\neg$ *pathG* [*node.One*, *node.Two*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.One*, *node.Two*]], *auto*)
lemma *notOTr*: $\neg$ *pathG* [*node.One*, *node.Three*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*node.One*, *node.Three*]], *auto*)
lemma *notZTr*: $\neg$ *pathG* [*Zero*, *node.Three*] **by**(*rule notI*, *cases rule: pathG.cases*[*of* [*Zero*, *node.Three*]], *auto*)

lemma *allCycles:basicCycle* $c \longleftrightarrow c = [\textit{Zero}, \textit{One}, \textit{Zero}] \vee c = [\textit{One}, \textit{Zero}, \textit{One}] \vee c = [\textit{Two}, \textit{Three}, \textit{Two}] \vee c = [\textit{Three}, \textit{Two}, \textit{Three}]$

apply(*standard*)
subgoal unfolding *basicCycle-def*[*unfolded cycleG-def*] **apply**(*elim conjE*)
apply(*cases rule: pathG.cases*[*of c*], *simp*)
subgoal for *nd* **by**(*cases nd*, *auto*)
subgoal for *nd'' ndl*
apply(*cases ndl*, *simp*)
subgoal for *nd ndl'* **apply**(*cases ndl'*, *simp*)
subgoal for *nd' ls*
apply(*cases ls*)
subgoal apply(*cases nd*)
subgoal by(*cases nd'*, *auto*)
subgoal by(*cases nd'*, *auto*)
subgoal by(*cases nd'*, *auto simp: notTZ*)
subgoal by(*cases nd'*, *auto simp: notTZ*) .

subgoal for *nnd ls'* **apply**(*cases nd*)

subgoal apply(*cases nd'*)
subgoal by(*cases nnd, cases nd'', auto*)
subgoal by(*cases nnd, cases nd'', auto simp: notPathG-within' notOTr*
notOT)
subgoal apply(*cases nnd, cases nd'', auto simp: notPathG-within'*
notTO)
by (*smt (verit, ccfv-threshold) PosOf.cases*
butlast.simps(2) edge.elims(2) hd-last-singletonI
last.simps list.distinct(1) list.set-sel(1)
node.distinct(9) node.simps(4,6) pathG.cases
snoc-eq-iff-butlast)

subgoal apply(*cases nnd, cases nd'', auto simp: notPathG-within'*
notTrO notOT) **using** *notZTr path-appendL*[*of* [*Zero*, *Three*]] **by** *auto* .

```

      subgoal apply(cases nd')
      subgoal by(cases nnd,cases nd'', auto simp: notPathG-within' notZTr,
        (metis List.last-in-set edge.elims(2) node.distinct(9) node.simps(4,8)))
      subgoal by(cases nnd,cases nd'', auto)
      subgoal by(cases nnd,cases nd'', auto simp: notPathG-within'
        notTZ,metis append-Cons empty-append-eq-id notOT notPathG-within)
      subgoal apply(cases nnd,cases nd'', auto simp: notPathG-within'
        notTrZ,metis append-Cons empty-append-eq-id notOTr notPathG-within) . .

```

```

      subgoal apply(cases nd')
      subgoal apply(cases nnd,cases nd'', auto) using notTZ path-appendL[of
        [Two, Zero]] by auto
      subgoal apply(cases nnd,cases nd'', auto) using notTO path-appendL[of
        [Two, One]] by auto
      subgoal by(cases nnd,cases nd'', auto simp: notPathG-within' notTZ)
      subgoal by(cases nnd,cases nd'', auto simp: notTrO notTrZ
        notPathG-within') .

```

```

      subgoal apply(cases nd')
      subgoal apply(cases nnd,cases nd'', auto) using notTrZ path-appendL[of
        [Three, Zero]] by auto
      subgoal apply(cases nnd,cases nd'', auto) using notTrO path-appendL[of
        [Three, One]] by auto
      subgoal by(cases nnd,cases nd'', auto simp: notTZ notTO not-
        PathG-within')
      subgoal by(cases nnd,cases nd'', auto simp: notTrZ notTrO
        notPathG-within') . . . . .

```

```

subgoal apply(elim disjE, simp-all add: basicCycle-def cycleG-def)
subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base)
subgoal unfolding ls-app by(rule pathG.Step, auto simp: pathG.Base) . .

```

```

lemma pathNWF-Three:pathG p  $\implies$  hd p = Three  $\implies$  last p  $\neq$  Zero  $\wedge$  last p  $\neq$ 
One
  apply(induct rule: pathG.induct)
  subgoal by auto
  subgoal for nd ndl apply (cases ndl, simp)
    subgoal for l ls by(cases last ndl, simp-all split: if-splits, auto) . .

```

```

lemma pathNWF-Two:pathG p  $\implies$  hd p = Two  $\implies$  last p  $\neq$  Zero  $\wedge$  last p  $\neq$  One
  apply(induct rule: pathG.induct)
  subgoal by auto
  subgoal for nd ndl apply (cases ndl, simp)
    subgoal for l ls by(cases last ndl, simp-all split: if-splits, auto) . .

```

```

lemma pathNWF-disj:pathG pa  $\implies$ 

```

$hd\ pa = node.Two \vee hd\ pa = node.Three \implies$
 $last\ pa = Zero \vee last\ pa = node.One \implies HOL.False$
apply(*elim disjE*) **using** *pathNWF-Two pathNWF-Three* **by** *auto*

lemma *unicycle:unicyclesGraph*
apply(*rule unicyclesGraphI', unfold allCycles connectedCycles-def, elim exE conjE disjE*)
subgoal by simp
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
subgoal using pathNWF-disj by auto
by auto

lemma *repeat-within:0 < n $\implies$ i < n + n $\implies$*
 $repeat\ n\ [node.Two, node.Three] ! i \in \{Two, Three\} \wedge$
 $(repeat\ n\ [node.Two, node.Three] @ [node.Two]) ! Suc\ i \in \{Two, Three\}$
using *set-repeat[of n [node.Two, node.Three]]*
 $nth_mem[of\ i\ repeat\ n\ [node.Two, node.Three]]$
 $nth_mem[of\ Suc\ i\ repeat\ n\ [node.Two, node.Three] @ [node.Two]]$
by auto

lemma *noRR:0 < n $\implies$ i < n + n $\implies$*
 $RR\ (repeat\ n\ [node.Two, node.Three] ! i, Pl ! i)$
 $((repeat\ n\ [node.Two, node.Three] @ [node.Two]) ! Suc\ i, Pl ! Suc\ i)$
 $Decr \implies$
 $HOL.False$
using *repeat-within[of n i] unfolding RR-defs* **by auto**

lemma *noDescentPath:0 < n $\implies$ descentPath*
 $(repeat\ n\ (butlast\ [node.Two, node.Three, node.Two]) @$
 $[last\ [node.Two, node.Three, node.Two]])$
 $Pl \implies HOL.False$
using *noRR unfolding descentPath-def* **by force**

lemma *SDG: $\neg$ SimplyDescendingGraph*
apply(*rule notI, unfold SimplyDescendingGraph-def*)
apply(*elim allE[of - [Two, Three, Two]] impE*)
subgoal using allCycles by auto
subgoal unfolding cycleDescends-def using noDescentPath by auto .

```

theorem  $\neg$ InfiniteDescent
  apply(unfold DescendingUnicyclesCriterion[OF unicycle])
  using SDG by auto

end

```

5.2.2 Flat Cycle Counterexample

```

theory Flat-Cycle-Example
  imports ../Criterion/All
begin

  datatype node = Zero | One | Two
  datatype pos = p0 | p0' | p1 | p1' | p2 | p2'

  definition Node  $\equiv$  {Zero, One, Two}

  lemma Z-Node[simp]: Zero  $\in$  Node by(simp add: Node-def)
  lemma T-Node[simp]: Two  $\in$  Node by(simp add: Node-def)

  lemma alw-nodes:alw (holdsS Node) nds
    unfolding alw-holdsS-iff-snth Node-def apply(rule allI)
    subgoal for i apply(induct i)
    using node.exhaust by auto .

  fun edge::node  $\Rightarrow$  node  $\Rightarrow$  bool where
    edge Zero One = HOL.True|
    edge One Zero = HOL.True|
    edge Zero Two = HOL.True|
    edge Two Zero = HOL.True|
    edge - - = HOL.False

  fun PosOf::node  $\Rightarrow$  pos set where
    PosOf Zero = {p0,p0'}|
    PosOf One = {p1,p1'}|
    PosOf Two = {p2,p2'}

  definition RR-set :: ((node  $\times$  pos)  $\times$  (node  $\times$  pos)  $\times$  slope) set where
    RR-set = {
      ((Zero, p0), (One, p1), Main),
      ((Zero, p0'), (One, p1'), Main),

```

```

((One, p1), (Zero, p0), Main),
((One, p1'), (Zero, p0'), Decr),

((Zero, p0), (Two, p2), Main),
((Zero, p0'), (Two, p2'), Main),

((Two, p2), (Zero, p0), Main),
((Two, p2'), (Zero, p0'), Main)
}

```

definition $RR :: node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool$ **where**
 $RR\ np1\ np2\ s \equiv ((np1, np2), s) \in RR\text{-}set$

lemmas $RR\text{-}defs = RR\text{-}def\ RR\text{-}set\text{-}def$

lemma $P\text{-}inPosOf:RR\ (nd, P)\ (nd', P')\ sl \implies P \in PosOf\ nd$
 $RR\ (nd, P)\ (nd', P')\ sl \implies P' \in PosOf\ nd'$ **by** $(auto\ simp: RR\text{-}defs)$

interpretation *Sloped-Graph* **where**
 $Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$
and $RR = RR$ **apply** *standard*
subgoal **by** $(simp\ add: Node\text{-}def)$
subgoal **by** $(unfold\ Node\text{-}def, auto\ elim: PosOf.cases)$
by $(auto\ simp: RR\text{-}defs\ SlopedRels\text{-}def\ Node\text{-}def)$

lemma $listE:(i < length\ ([Zero, node.Two]\ @\ [Zero]) - 1) \implies (i = 0 \implies P) \implies$
 $(i = 1 \implies P) \implies P$ **by** *fastforce*

lemma $ZT\text{-}FlatEdge:(Zero, Two) \in FlatEdges$ **unfolding** $FlatEdges\text{-}def$ **by** $(simp\ add: RR\text{-}defs)$

lemma $TZ\text{-}FlatEdge:(Two, Zero) \in FlatEdges$ **unfolding** $FlatEdges\text{-}def$ **by** $(simp\ add: RR\text{-}defs)$

lemma $\neg InfiniteDescent$
apply $(rule\ Flat\text{-}Cycles\text{-}Criterion[unfolded\ FlatCycle\text{-}def])$

apply $(intro\ exI[of\ -\ ([Zero, Two]\ @\ [Zero])\ conjI])$

subgoal **apply** $(intro\ allI\ impI, erule\ listE)$
using $ZT\text{-}FlatEdge\ TZ\text{-}FlatEdge$ **by** *auto*

subgoal **unfolding** $cycleG\text{-}def$
apply $(rule\ conjI)$
subgoal **by** $(rule\ pathG.Step, simp\text{-}all\ add: pathG.Base)$
by *auto* .

end

References

- [1] J. Brotherston, “Sequent Calculus Proof Systems for Inductive Definitions,” Ph.D. dissertation, University of Edinburgh, November 2006. [Online]. Available: <https://era.ed.ac.uk/handle/1842/1458>
- [2] L. Cohen, A. Jabarin, A. Popescu, and R. N. S. Rowe, “The Complex(ity) Landscape of Checking Infinite Descent,” *Proc. ACM Program. Lang.*, vol. 8, no. POPL, jan 2024. [Online]. Available: <https://doi.org/10.1145/3632888>
- [3] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, “The Size-change Principle for Program Termination,” in *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, C. Hankin and D. Schmidt, Eds. ACM, 2001, pp. 81–92.
- [4] A. Simpson, “Cyclic Arithmetic Is Equivalent to Peano Arithmetic,” in *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, ser. Lecture Notes in Computer Science, J. Esparza and A. S. Murawski, Eds., vol. 10203, 2017, pp. 283–300.
- [5] C. Sprenger and M. Dam, “On the Structure of Inductive Reasoning: Circular and Tree-Shaped Proofs in the μ -calculus,” in *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed., vol. 2620. Springer, 2003, pp. 425–440.