Certified Infinite Descent Criteria in Isabelle/HOL

Jamie Wright, Liron Cohen, Reuben Rowe & Andrei Popescu

July 5, 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

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*:$j < i \Longrightarrow Suc\ j < i \vee Suc\ j = i$ ⟨*proof*⟩

lemma *distinct-wrap-around*:
assumes *distinct c*
assumes $j > i$
shows *distinct* (*drop j c* @ *take i c*)
⟨*proof*⟩

lemma *distinct-outside-index*:
assumes

$0 < j$ $0 < length\ c$ $distinct\ c$
$\forall j{<}length\ c.\ 0 \neq j \longrightarrow c\ !\ 0 \neq c\ !\ j$
shows $c\ !\ 0 \notin (!)\ c\ `\ \{j..{<}length\ c\}$
⟨*proof*⟩

lemma *distinct-outside-index'*:
assumes
$distinct\ (butlast\ c)$
$Suc\ i < length\ (butlast\ c)$
$\forall j{<}length\ (butlast\ c).\ Suc\ i \neq j \longrightarrow butlast\ c\ !\ Suc\ i \neq butlast\ c\ !\ j$
shows $c\ !\ Suc\ i \notin (!)\ (butlast\ c)\ `\ \{j..{<}i\}$
⟨*proof*⟩

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

lemma *length-fToList*[*simp*]: $length\ (fToList\ n\ f) = n$
⟨*proof*⟩

lemma *nth-fToList*[*simp*]: $i < n \Longrightarrow (fToList\ n\ f)!i = f\ i$
⟨*proof*⟩

lemma *fToList-nth*[*simp*]: $(fToList\ (length\ xs)\ (nth\ xs)) = xs$
⟨*proof*⟩

lemma *list-split2*:
assumes $i < j$ **and** $j < length\ xs$
obtains *xs1 xs2 xs3* **where** $xs = xs1\ @\ (xs!i)\ \#\ xs2\ @\ (xs!j)\ \#\ xs3$
⟨*proof*⟩

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

lemma *repeat-0*[*simp*]: $repeat\ 0\ xs = []$
⟨*proof*⟩

lemma *repeat-Suc*[*simp*]: $repeat\ (Suc\ n)\ xs = xs\ @\ repeat\ n\ xs$
⟨*proof*⟩

lemma *repeat-Suc2*: $repeat\ (Suc\ n)\ xs = repeat\ n\ xs\ @\ xs$
⟨*proof*⟩

lemma *set-repeat*[*simp*]: $n > 0 \Longrightarrow set\ (repeat\ n\ xs) = set\ xs$
⟨*proof*⟩

lemma *nth-repeat*[*simp*]: $length\ (repeat\ n\ xs) = n * length\ xs$

⟨*proof*⟩

lemma *repeat-nth*:
$i < n * length\ xs \Longrightarrow repeat\ n\ xs\ !\ i = xs\ !\ (i\ mod\ length\ xs)$
⟨*proof*⟩

lemma *tl-last'*:
$tl\ xs \neq [] \Longrightarrow last\ xs = last\ (tl\ xs)$
⟨*proof*⟩

definition *fToStream* :: $(nat \Rightarrow 'a) \Rightarrow 'a\ stream$ **where**
$fToStream\ f \equiv smap\ f\ nats$

lemma *snth-fToStream*[*simp*]: $(fToStream\ f)!!i = f\ i$
⟨*proof*⟩

lemma *shd-fToStream*[*simp*]:*shd* $(fToStream\ f) = f\ 0$ ⟨*proof*⟩

lemma *stl-fToStream*:*stl* $(fToStream\ f)\ !!\ n = (fToStream\ f)\ !!\ Suc\ n$ ⟨*proof*⟩

lemma *fToStream-snth*[*simp*]: $(fToStream\ (snth\ xs)) = xs$
⟨*proof*⟩

lemma *sdrop-shift-length*: $length\ xs = m \Longrightarrow sdrop\ m\ (xs\ @-\ ys) = ys$
⟨*proof*⟩

lemma *sdrop-shift-length'*: $sdrop\ (length\ xs)\ (xs\ @-\ ys) = ys$
⟨*proof*⟩

lemma *snth-equalityI*: $(\bigwedge i.\ xs\ !!\ i = ys\ !!\ i) \Longrightarrow xs = ys$
⟨*proof*⟩

lemma *hd-u-append*[*simp*]:*hd* $(u\ @\ [hd\ u]) = hd\ u$ ⟨*proof*⟩

lemma *stake-len-append*:*stake* $(length\ v)\ (v\ @-\ u) = v$
⟨*proof*⟩

lemma *last-stake-Suc*:*last* $(stake\ (Suc\ x')\ r) = r\ !!\ x'$ ⟨*proof*⟩

abbreviation *srepeat* ≡ *cycle*
hide-const *cycle*
declare *cycle.simps*[*simp del*]
declare *snth.simps*[*simp del*]

lemma *stake-srepeat*:$(stake\ (length\ u)\ (sdrop\ 0\ (srepeat\ u))) = u$
⟨*proof*⟩

lemma *sdrop-evI*: $\varphi\ (sdrop\ m\ xs) \Longrightarrow ev\ \varphi\ xs$
⟨*proof*⟩

lemma *srepeat-snth*[*simp*]: $xs \neq [] \Longrightarrow (srepeat\ xs)\ !!\ i = xs!(i\ mod\ (length\ xs))$
⟨*proof*⟩

lemma *sset-eq-snth*: $sset\ xs = \{xs!!i \mid i\ .\ True\}$
⟨*proof*⟩

lemma *repeat-eq-stake-srepeat*:
$repeat\ n\ xs = stake\ (n * length\ xs)\ (srepeat\ xs)$
⟨*proof*⟩

lemma *repeat-nth-eq-srepeat-snth*: $i < n * length\ xs \Longrightarrow repeat\ n\ xs\ !\ i = srepeat\ xs\ !!\ i$
⟨*proof*⟩

lemma *srepeat-repeat*:
$n \neq 0 \Longrightarrow srepeat\ (repeat\ n\ xs) = srepeat\ xs$
⟨*proof*⟩

lemma *last-replicate-app*: $k{>}0 \Longrightarrow last\ (concat\ (replicate\ k\ (ls\ @\ [l'\rceil]))) = l'$
⟨*proof*⟩

lemma *last-stake-Suc-app*:$last\ (p\ \#\ stake\ (Suc\ x')\ r) = last\ (stake\ (Suc\ x')\ r)$
⟨*proof*⟩

lemma *upt-Suc-hd*:$Suc\ x' \leq y' \Longrightarrow ([Suc\ x'..{<}y'\rceil\ @\ [y'\rceil])\ !\ 0 = Suc\ x'$
⟨*proof*⟩

lemma *stl-Suc-eq*:$stl\ r\ !!\ k = r\ !!\ Suc\ k$ ⟨*proof*⟩

lemma *stl-Suc*:$k{>}0 \Longrightarrow stl\ r\ !!\ (k - Suc\ 0) = r\ !!\ k$ ⟨*proof*⟩
lemma *stl-Suc'*:$k{>}0 \Longrightarrow stl\ r\ !!\ (k - 1) = r\ !!\ k$ ⟨*proof*⟩
lemma *replicate-within-i*:$(Suc\ i) \leq m \Longrightarrow (replicate\ m\ s\ @{-}\ x)\ !!\ i = s$ ⟨*proof*⟩

lemma *replicate-beyond-i*:$(Suc\ i) > m \Longrightarrow (replicate\ m\ s\ @{-}\ x)\ !!\ i = x\ !!\ (i{-}m)$
⟨*proof*⟩

lemma *last-stake-i*:$n > 0 \Longrightarrow last\ (stake\ n\ x) = x\ !!\ (n{-}1)$ ⟨*proof*⟩
lemma *last-stake-replicate*:$(Suc\ i) \leq m \Longrightarrow 0 < i \Longrightarrow last\ (stake\ i\ (replicate\ m\ s\ @{-}\ x)) = s$

⟨*proof*⟩

lemma *Suc-leq*:*Suc* (*Suc n*) ≤ *m* ⟹ *n* ≤ *m*−*Suc* (*Suc 0*) ⟨*proof*⟩

lemma *alw-ev-iff-sdrop*: *alw* (*ev* φ) *xs* = (∀ *m*. ∃ *n*≥*m*. φ (*sdrop n xs*))
⟨*proof*⟩

lemma *ev-alw-iff-sdrop*: *ev* (*alw* φ) *xs* = (∃ *m*. ∀ *n*≥*m*. φ (*sdrop n xs*))
⟨*proof*⟩

lemma *ev-alw-ev-iff-sdrop*: *ev* (*alw* (*ev* φ)) *xs* = (∃ *l*. ∀ *m*≥*l*. ∃ *n*≥*m*. φ (*sdrop n xs*))
⟨*proof*⟩

lemma *ev* (*alw* (*ev* φ)) *xs* ⟷ *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *alw-ev-sdrop*: *alw* (*ev* φ) (*sdrop m xs*) ⟷ *alw* (*ev* φ) *xs*
⟨*proof*⟩

lemma *ev-alw-sdrop*: *ev* (*alw* φ) (*sdrop m xs*) ⟷ *ev* (*alw* φ) *xs*
⟨*proof*⟩

lemma *ev-alw-aand-alw-ev-sdrop*:
(*ev* (*alw* φ) *aand alw* (*ev* ψ)) (*sdrop m xs*) ⟷ (*ev* (*alw* φ) *aand alw* (*ev* ψ)) *xs*
⟨*proof*⟩

fun *holds2* **where** *holds2 P xs* ⟷ *P* (*shd xs*) (*shd* (*stl xs*))

fun *second* **where** *second* (*a*,*b*,*c*) = *b*
lemma *second'*:*second x* = (*case x of* (*a*, *b*, *c*) ⇒ *b*) ⟨*proof*⟩

fun *third* **where** *third* (*a*,*b*,*c*) = *c*
lemma *third'*:*third x* = (*case x of* (*a*, *b*, *c*) ⇒ *c*) ⟨*proof*⟩

lemma *third-ssnd*:*third* = (λ*x*. *snd* (*snd x*)) ⟨*proof*⟩

lemma *third-snd*:(*third* ∘ *snd*) = (λ*x*. *third* (*snd x*)) ⟨*proof*⟩
lemma *stake-replicate*:*stake n* (*replicate n X* @− *S*) = *replicate n X* ⟨*proof*⟩

lemma *sdrop-replicate*:*sdrop n* (*replicate n X* @− *S*) = *S* ⟨*proof*⟩

definition *holdsS S* = *holds* (λ*x*. *x* ∈ *S*)

lemma *alw-holds-iff-snth*: *alw* (*holds P*) *xs* ⟷ (∀ *i*. *P*(*xs*!!*i*))
⟨*proof*⟩

lemma *alw-holdsS-iff-snth*: *alw* (*holdsS S*) *xs* ⟷ (∀ *i*. *xs*!!*i* ∈ *S*)
⟨*proof*⟩

lemma *alw-holds2-iff-snth*: *alw* (*holds2 R*) *xs* ⟷ (∀ *i*. *R* (*xs*!!*i*) (*xs*!!(*Suc i*)))
⟨*proof*⟩

lemma *ev-holds-iff-snth*: *ev* (*holds P*) *xs* ⟷ (∃ *i*. *P*(*xs*!!*i*))
⟨*proof*⟩

lemma *ev-holdsS-iff-snth*: *ev* (*holdsS S*) *xs* ⟷ (∃ *i*. *xs*!!*i* ∈ *S*)
⟨*proof*⟩

lemma *ev-holds2-iff-snth*: *ev* (*holds2 R*) *xs* ⟷ (∃ *i*. *R* (*xs*!!*i*) (*xs*!!(*Suc i*)))
⟨*proof*⟩

lemma *alw-ev-holds-iff-snth*: *alw* (*ev* (*holds P*)) *xs* ⟷ (∀ *i*. ∃ *j*≥*i*. *P*(*xs*!!*j*))
⟨*proof*⟩

lemma *alw-ev-holdsS-iff-snth*: *alw* (*ev* (*holdsS S*)) *xs* ⟷ (∀ *i*. ∃ *j*≥*i*. *xs*!!*j* ∈ *S*)
⟨*proof*⟩

lemma *alw-ev-holds2-iff-snth*: *alw* (*ev* (*holds2 R*)) *xs* ⟷ (∀ *i*. ∃ *j*≥*i*. *R* (*xs*!!*j*)
(*xs*!!(*Suc j*)))
⟨*proof*⟩

lemma *ev-alw-holds-iff-snth*: *ev* (*alw* (*holds P*)) *xs* ⟷ (∃ *i*. ∀ *j*≥*i*. *P*(*xs*!!*j*))
⟨*proof*⟩

lemma *ev-alw-holdsS-iff-snth*: *ev* (*alw* (*holdsS S*)) *xs* ⟷ (∃ *i*. ∀ *j*≥*i*. *xs*!!*j* ∈ *S*)
⟨*proof*⟩

lemma *ev-alw-holds2-iff-snth*: *ev* (*alw* (*holds2 R*)) *xs* ⟷ (∃ *i*. ∀ *j*≥*i*. *R* (*xs*!!*j*)
(*xs*!!(*Suc j*)))
⟨*proof*⟩

lemma *ev-alw-holds2-aand-holdsS-iff-snth*:

$ev\ (alw\ (holdsS\ S\ aand\ holds2\ R))\ xs \longleftrightarrow (\exists\, i.\ \forall\, j{\geq}i.\ xs!!j \in S \wedge R\ (xs!!j)\ (xs!!(Suc\ j)))$
$\langle proof \rangle$

lemma *nat-cases*:$(x::nat) = y \vee y < x \vee y > x$ $\langle proof \rangle$

lemma *snth-hd-app-eq*[*simp*]:$(x\ \#\#\ xs)\ !!\ 0 = x$ $\langle proof \rangle$
lemma *stream-inclI*:$x \in S \Longrightarrow k{>}0 \longrightarrow xs!!(k{-}1) \in S \Longrightarrow (x\ \#\#\ xs)!!k \in S$
$\langle proof \rangle$
lemma *sdrop-1*:$(sdrop\ (Suc\ 0)\ nds) = stl\ nds$ $\langle proof \rangle$

lemma *disj-mod*:$((k::nat)\ div\ N = 0 \wedge k\ mod\ N = 0) \vee (0 < k\ div\ N \wedge k\ mod\ N = 0) \vee (0 < k\ mod\ N)$
$\langle proof \rangle$

lemma *mod-bound*:$y' > x' \Longrightarrow k\ mod\ (y' - x') - Suc\ 0 < Suc\ y' - Suc\ x'$
$\langle proof \rangle$

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

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

lemma *mod-contr3*:
assumes $y' < k\ mod\ (y' - x') + x'\ Suc\ x' < y'$
shows *HOL.False*
$\langle proof \rangle$

lemma *mod-arith1*:**assumes** $Suc\ x' < y'$
shows $Suc\ (k\ mod\ (y' - x') + x') \leq y'$
$(k\ mod\ (y' - x')) \leq y' - Suc\ x'$
$Suc\ x' + k\ mod\ (y' - x') < Suc\ y'$
$\langle proof \rangle$

lemma *le-xy-mod*:$(x'::nat) < y' \Longrightarrow k\ mod\ (y' - x') < y' - x'$ $\langle proof \rangle$

definition *map-ind x i j* ≡ *map* ((!!) *x*) [*i*..<*j*]

lemma *map-ind-ne*[*simp*]:*i* < *j* ⟹ *map-ind x i j* ≠[] ⟨*proof*⟩

lemma *map-ind-len*[*simp*]:*i* < *j* ⟹ *length* (*map-ind x i j*) = *j*−*i*⟨*proof*⟩

lemma *srepeat-map-ind-eq*:*i* < *j*⟹ *srepeat* (*map-ind x i j*) !! *k* = *x* !! ([*i*..<*j*] ! (*k mod* (*j* − *i*)))
⟨*proof*⟩

lemma *hd-map-ind*:*i* < *j* ⟹ *hd* (*map-ind x i j*) = *x* !! *i* ⟨*proof*⟩

lemma *map-ind-sing*[*simp*]:*map-ind r i* (*Suc i*) = [*r* !! *i*]
⟨*proof*⟩

lemma *tl-map-ind*:*i* < *j* ⟹*tl* (*map-ind x i j*) = (*map-ind x* (*Suc i*) *j*)
⟨*proof*⟩

lemma *nth-map-ind*: *k* < *j*−*i* ⟹ (*map-ind x i j*) ! *k* = *x* !! (*i*+*k*)
⟨*proof*⟩

lemma *map-ind-Suc*:*y*′≥*x*′ ⟹ *map-ind r x*′ *y*′ @ [*r* !! *y*′] = *map-ind r x*′ (*Suc y*′)
⟨*proof*⟩

lemma *last-replicate-map-ind*:0 < *k* ⟹ *y*′≥*x*′ ⟹ *last* (*concat* (*replicate k* (*map-ind r x*′ (*Suc y*′)))) = *r* !! *y*′
⟨*proof*⟩

lemma *srepeat-map-ind-snth-eq*:*i* < *j* ⟹ *srepeat* (*map-ind x i j*) !! *k* = *x* !! (*i* + *k mod* (*j* − *i*))
⟨*proof*⟩

lemma *last-take-Suc*′:*k* = *Suc n* ⟹ *last* (*stake k r*) = *r* !! *n*
⟨*proof*⟩

lemma *two-ls-app*:[*y*, *v*] = [*y*] @ [*v*] ⟨*proof*⟩

lemma *list-unf*:(*v* # *xs*′ @ [*y*]) @ *z*′ # *zs*′ @ [*v*] = *v* # *xs*′ @ *y* # *z*′ # *zs*′ @ [*v*]
⟨*proof*⟩

lemma *set-nemp*:*V*′ ≠ {} ⟷ (∃ *v*. *v* ∈ *V*′) ⟨*proof*⟩

lemma *exI2*:$P\ a\ b \Longrightarrow \exists\, a\ b.\ P\ a\ b$ ⟨*proof*⟩
lemma *exI3*:$P\ a\ b\ c \Longrightarrow \exists\, a\ b\ c.\ P\ a\ b\ c$ ⟨*proof*⟩
lemma *exI4*:$P\ a\ b\ c\ d \Longrightarrow \exists\, a\ b\ c\ d.\ P\ a\ b\ c\ d$ ⟨*proof*⟩

lemma *exI5*:$P\ a\ b\ c\ d\ e \Longrightarrow \exists\, a\ b\ c\ d\ e.\ P\ a\ b\ c\ d\ e$ ⟨*proof*⟩

end

1.2 Buchi Complementation Extended

A small extension to the B"uchi Complementation AFP entry, including: useful definitions, results on omega lasso languages, finitely occuring 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 $\bigwedge k.\ x\ !!\ k \in nba.alphabet\ A$
⟨*proof*⟩

lemma *runE-0*:
assumes *NBA.run A* (*x* ||| *r*) *p*
shows *r* !! *0* ∈ *nba.transition A* (*x* !! *0*) *p*
⟨*proof*⟩

lemma *runE-Suc*:
assumes *NBA.run A* (*x* ||| *r*) *p*
shows $\bigwedge k.\ r\ !!\ Suc\ k \in nba.transition\ A\ (x\ !!\ Suc\ k)\ (r\ !!k)$
⟨*proof*⟩

lemma *runE-Suc'*:
assumes *NBA.run A* (*x* ||| *r*) *p* *k* > *0*
shows *r* !! *k* ∈ *nba.transition A* (*x* !! *k*) (*r* !! (*k*−*1*))
⟨*proof*⟩

lemma *runE-trans*:
assumes *NBA.run A* (*x* ||| *r*) *p*
shows $k > 0 \wedge r\ !!\ k \in nba.transition\ A\ (x\ !!\ k)\ (r\ !!\ (k-1)) \vee$
$k = 0 \wedge r\ !!\ k \in nba.transition\ A\ (x\ !!\ k)\ p$
⟨*proof*⟩

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 $p \in nba.initial\ A$
$(n = p \wedge NBA.path\ A\ []\ p) \vee (\exists r.\ r \neq [] \wedge n = last\ (NGBA.states\ r\ p) \wedge NBA.path\ A\ r\ p)$
shows $n \in NBA.nodes\ A$
⟨*proof*⟩

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 $n \in NBA.nodes\ A$
⟨*proof*⟩

lemma *nba-path-singl*:$fst\ a \in nba.alphabet\ A \Longrightarrow snd\ a \in nba.transition\ A\ (fst\ a)\ p \Longrightarrow NBA.path\ A\ [a]\ p$ ⟨*proof*⟩

lemma *nba-path-app-singl*:$fst\ a \in nba.alphabet\ A \Longrightarrow snd\ a \in nba.transition\ A\ (fst\ a)\ (NGBA.target\ r\ p) \Longrightarrow NBA.path\ A\ r\ p \Longrightarrow\ NBA.path\ A\ (r\ @\ [a])\ p$
⟨*proof*⟩

lemma *target-last-states*[*simp*]:$r \neq [] \Longrightarrow NGBA.target\ r\ p = last\ (NGBA.states\ r\ p)$
⟨*proof*⟩

lemma *target-emp*[*simp*]:$NGBA.target\ []\ p = p$ ⟨*proof*⟩
lemma *last-states-singl*[*simp*]:$last\ (NGBA.states\ [(x,s)]\ p) = s$ ⟨*proof*⟩
lemma *last-states-app*[*simp*]:$ra \neq [] \Longrightarrow last\ (NGBA.states\ (ra\ @[(x,s)])\ p) = s$ ⟨*proof*⟩

lemma *run-nodes-closure*:
assumes *lang*: $p \in nba.initial\ A\ NBA.run\ A\ (x\ |||\ r)\ p$
shows$\bigwedge x.\ r\ !!\ x \in NBA.nodes\ A$
⟨*proof*⟩

lemma *map-snd-stake-szip*:$map\ snd\ (stake\ k\ (x\ |||\ y)) = stake\ k\ y$ ⟨*proof*⟩
lemma *szip-unfold*:$(x'\ |||\ r') = (shd\ x', shd\ r')\ \#\#\ (stl\ x'\ |||\ stl\ r')$ ⟨*proof*⟩

lemma *len-stake-szip*: $length\ (q\ \#\ map\ snd\ (stake\ (Suc\ n)\ (x\ |||\ r))) = Suc\ (Suc\ n)$ ⟨*proof*⟩
lemma *nth-stake-szip*:$(q\ \#\ map\ snd\ (stake\ (Suc\ n)\ (x\ |||\ r)))\ !\ Suc\ n = r\ !!\ n$ ⟨*proof*⟩

lemma *last-stake-szip*:*last* (*q* # *map snd* (*stake* (*Suc n*) (*x* ||| *r*))) = *r* !! *n*
⟨*proof*⟩

lemma *fst-nth-zip*:*fst* ((*x* ||| *r*) !! *k*) = *x* !! *k* ⟨*proof*⟩
lemma *snd-nth-zip*:*snd* ((*x* ||| *r*) !! *k*) = *r* !! *k* ⟨*proof*⟩

lemma *fins-finite*:*fins P x* ⟹ *finite* {*m*. *P* (*x* !! *m*)}
⟨*proof*⟩

lemma *fins-imp*:*fins P x* ⟹ *alw* (*holds* (λ*x*. ¬*P x*)) *x* ∨ (∃ *i*. ∀ *k* > *i*. *P* (*x* !! *i*)
∧ ¬*P* (*x* !! *k*))
⟨*proof*⟩

lemma *target-zip-nth*:*n* > *0* ⟹ *NGBA.target* (*stake n w* || *stake n r'*) *i* = *r'* !!
(*n*−*1*)
⟨*proof*⟩

lemma *infs-snthI*: (⋀*n*. ∃ *k* ≥ *n*. *P* (*w* !! *k*)) ⟹ *infs P w*
⟨*proof*⟩

lemma *complement-eq1*:
assumes *NBA.alphabet A* ⊆ *NBA.alphabet B*
assumes *finite* (*NBA.nodes B*)
shows *NBA.language A* ⊆ *NBA.language B* ⟷ *NBA.language A* ∩ *NBA.language*
(*complement B*) = {}
⟨*proof*⟩

lemma *complement-eq2*:(*NBA.language A* ∩ *NBA.language* (*complement B*) = {})
= (*NBA.language* (*intersect A* (*complement B*)) = {}) ⟨*proof*⟩

lemma *complement-eq3*:
assumes *NBA.alphabet A* ⊆ *NBA.alphabet B*
assumes *finite* (*NBA.nodes B*)
shows *NBA.language A* ⊆ *NBA.language B* = (*NBA.language* (*intersect A* (*complement*
B)) = {})
⟨*proof*⟩

lemma *omega-lasso-lang:finite* (*NBA.nodes A*) $\Longrightarrow$ $x \in$ *NBA.language A* $\Longrightarrow$ $\exists\, v$ *u. v* @− *srepeat u* $\in$ *NBA.language A* $\wedge$ $u \neq []$
⟨*proof*⟩

corollary *omega-lasso-lang′:finite* (*NBA.nodes A*) $\Longrightarrow$ *NBA.language A* $\neq \{\}$ $\longleftrightarrow$ ($\exists\, v$ *u. v* @− *srepeat u* $\in$ *NBA.language A* $\wedge$ $u \neq []$)
⟨*proof*⟩
corollary *omega-lasso-lang″:finite* (*NBA.nodes A*) $\Longrightarrow$ *NBA.language A* $= \{\}$ $\longleftrightarrow$ ($\forall\, v$ *u. v* @− *srepeat u* $\notin$ *NBA.language A*)
⟨*proof*⟩

lemma *prop1*:
assumes *NBA.alphabet A* $\subseteq$ *NBA.alphabet B*
assumes *finite* (*NBA.nodes A*) *finite* (*NBA.nodes B*)
assumes $\bigwedge v$ *u.* $u \neq []$ $\Longrightarrow$ *v* @− *srepeat u* $\in$ *NBA.language A* $\Longrightarrow$ *v* @− *srepeat u* $\in$ *NBA.language B*
shows *NBA.language A* $\subseteq$ *NBA.language B*
⟨*proof*⟩

corollary *prop1′*:
assumes *NBA.alphabet A* $\subseteq$ *NBA.alphabet B*
assumes *finite* (*NBA.nodes A*) *finite* (*NBA.nodes B*)
assumes $\neg$(*NBA.language A* $\subseteq$ *NBA.language B*)
shows $\exists\, v$ *u. v* @− *srepeat u* $\in$ *NBA.language A* $\wedge$ *v* @− *srepeat u* $\notin$ *NBA.language B* $\wedge$ $u \neq []$

⟨*proof*⟩

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* ⇒ *bool*
where
Base: {*nd*,*nd'*} ⊆ *Node* ⟹ *edge nd nd'* ⟹ *pathG* [*nd*,*nd'*]
|*Step*: *nd* ∈ *Node* ⟹ *pathG ndl* ⟹ *edge* (*last ndl*) *nd* ⟹ *pathG* (*ndl* @ [*nd*])

lemma *path-length-ge2*: **assumes** *pathG ndl* **shows** *length ndl* ≥ *2*
⟨*proof*⟩

lemma *path-set*:
assumes *pathG ndl* **shows** *set ndl* ⊆ *Node*
⟨*proof*⟩

lemma *path-nth-'node*:
pathG ndl ⟹ *i* < *length ndl* ⟹ *ndl*!*i* ∈ *Node*
⟨*proof*⟩

lemma *pathG-butlast*:*2* < *length c* ⟹ *pathG c* ⟹ *pathG* (*butlast c*) ⟨*proof*⟩

lemma *path-nth-edge*:
assumes *pathG ndl Suc i* < *length ndl* **shows** *edge* (*ndl*!*i*) (*ndl*!(*Suc i*))
⟨*proof*⟩

lemma *path-iff-set-nth*:
pathG ndl ⟷
length ndl ≥ *2* ∧ *set ndl* ⊆ *Node* ∧ (∀ *i*. *Suc i* < *length ndl* ⟶ *edge* (*ndl*!*i*)
(*ndl*!(*Suc i*)))
⟨*proof*⟩

lemma *path-iff-nth*:
pathG ndl ⟷
length ndl ≥ *2* ∧ (∀ *i*<*length ndl*. *ndl*!*i* ∈ *Node*) ∧ (∀ *i*. *Suc i* < *length ndl* ⟶
edge (*ndl*!*i*) (*ndl*!(*Suc i*)))
⟨*proof*⟩

lemma *Graph-pathG-restrict*:
Graph.pathG N e ndl ⟷ *Graph.pathG N* (λ*x y*. *e x y* ∧ *x* ∈ *N* ∧ *y* ∈ *N*) *ndl*
⟨*proof*⟩

lemma *path-appendL*:
pathG (*ndl1* @ *ndl2*) ⟹ *length ndl1* ≥ *2* ⟹ *pathG ndl1*
⟨*proof*⟩

lemma *path-appendR*:
pathG (*ndl1* @ *ndl2*) ⟹ *length ndl2* ≥ *2* ⟹ *pathG ndl2*
⟨*proof*⟩

lemma *path-appendM*:
pathG (*ndl1* @ *ndl2* @ *ndl3*) ⟹ *length ndl2* ≥ *2* ⟹ *pathG ndl2*
⟨*proof*⟩

lemma *ls-app*:[*a*, *b*, *c*] = [*a*,*b*] @ [*c*] ⟨*proof*⟩

lemma *notPathG-within*:¬*pathG* [*a*,*b*] ⟹ ¬*pathG* (*ls* @ [*a*,*b*] @ *ls'*) ⟨*proof*⟩

lemma *notPathG-within'*:¬*pathG* [*a*,*b*] ⟹ ¬*pathG* (*ls* # *a* # *b* # *ls'*) ⟨*proof*⟩

lemma *pathG-tl*:*2* ≤ *length xs* ⟹ *pathG* (*x*#*xs*) ⟹ *pathG xs* ⟨*proof*⟩

lemma *path-append-hd*:
assumes *pathG* (*ndl1* @ [*hd ndl2*]) **and** *pathG ndl2*
shows *pathG* (*ndl1* @ *ndl2*)
⟨*proof*⟩

lemma *path-append-last*:
assumes *pathG ndl1* **and** *pathG* (*last ndl1* # *ndl2*)
shows *pathG* (*ndl1* @ *ndl2*)
⟨*proof*⟩

lemma *path-Cons*:
assumes *nd* ∈ *Node edge nd* (*hd ndl*) *pathG ndl*
shows *pathG* (*nd* # *ndl*)
⟨*proof*⟩

lemma *not-path-Nil*[*simp*]: ¬ *pathG* []
⟨*proof*⟩

lemma *not-path-singl*[*simp*]: ¬ *pathG* [*nd*]
⟨*proof*⟩

lemma *not-path-emp*: *Node* = {} ⟹ ¬ *pathG ndl*
⟨*proof*⟩

lemma *path-singl-in*:
assumes *edge nd nd n* ≥ *2 nd* ∈ *Node*
shows *pathG* (*replicate n nd*)
⟨*proof*⟩

lemma *path-singl-set*:
assumes *set ndl* ⊆ {*nd*} *nd* ∈ *Node*
shows *pathG ndl* ⟷ (*edge nd nd* ∧ (∃ *n*≥*2*. *ndl* = *replicate n nd*))
⟨*proof*⟩

lemma *path-two-incl*:
assumes *edge nd nd'* {*nd*,*nd'*} ⊆ *Node*

shows *pathG* [*nd*,*nd'*]
⟨*proof*⟩

lemma *path-two-concat-incl*:
assumes *edge nd nd' edge nd' nd n* > *0* {*nd*,*nd'*} ⊆ *Node*
shows *pathG* (*concat* (*replicate n* [*nd*,*nd'*]))
⟨*proof*⟩

lemma *pathG-butlast-not-nil*:*pathG n* ⟹ *butlast n* ≠ [] ⟨*proof*⟩

definition *pathCon* **where**
pathCon nd nd' ≡ ∃ *ndl*. *pathG ndl* ∧ *hd ndl* = *nd* ∧ *last ndl* = *nd'*

lemma *pathCon-trans*:
pathCon nd nd' ⟹ *pathCon nd' nd''* ⟹ *pathCon nd nd''*
⟨*proof*⟩

lemma *not-pathCon-emp*: *Node* = {} ⟹ ¬ *pathCon nd1 nd2*
⟨*proof*⟩

lemma *pathCon-singl*: *Node* = {*nd*} ⟹ *pathCon nd1 nd2* ⟷ *nd1* = *nd* ∧ *nd2* = *nd* ∧ *edge nd1 nd2*
⟨*proof*⟩

lemma *path-nth-pathCon*:
assumes *fp*: *pathG ndl* **and** *ij*: *i* < *j j* < *length ndl*
shows *pathCon* (*ndl*!*i*) (*ndl*!*j*)
⟨*proof*⟩

lemma *path-set-pathCon*:
assumes *fp*: *pathG ndl* **and** *nd*: {*nd*,*nd'*} ⊆ *set ndl nd* ≠ *nd'*
shows *pathCon nd nd'* ∨ *pathCon nd' nd*
⟨*proof*⟩

definition *cycleG* **where**
cycleG ndl ≡ *pathG ndl* ∧ *hd ndl* = *last ndl*

lemma *cycleG-not-nil*:*cycleG c* ⟹ *c* ≠[] ⟨*proof*⟩
lemma *cycleG-length-ge*:*cycleG c* ⟹ *length c* ≥ *2* ⟨*proof*⟩

lemma *cycleG-path-drop*: *cycleG c* ⟹ *j* < *length* (*butlast c*) ⟹ *pathG* (*drop j c*)

⟨*proof*⟩

definition *cycleFrom* **where**
cycleFrom nd ndl ≡ *cycleG ndl* ∧ *hd ndl* = *nd*

lemma *cycle-rotate1*:
cycleG (*ndl* @ [*nd*,*nd'*]) ⟹ *cycleG* (*nd* # *ndl* @ [*nd*])
⟨*proof*⟩

lemma *cycle-rotate*:
assumes *cycleG* (*ndl1* @ *ndl2* @ [*nd'*]) *ndl2* ≠ []
shows *cycleG* (*ndl2* @ *ndl1* @ [*hd ndl2*])
⟨*proof*⟩

lemma *cycle-rotate-butlast*:
assumes *cycleG* (*ndl1* @ *nd* # *ndl2*) *ndl1* ≠ [] *ndl2* ≠ []
shows *cycleG* (*nd* # *butlast ndl2* @ *ndl1* @ [*nd*])
⟨*proof*⟩

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*⟩

lemma *cycle-set-pathCon*:
assumes *cy*: *cycleG ndl* **and** *nd*: {*nd*,*nd'*} ⊆ *set ndl*
shows *pathCon nd nd'*
⟨*proof*⟩

lemma *cycle-iff-nth*:
cycleG ndl ⟷
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*)))
⟨*proof*⟩

lemma *cycleG-shape*: *cycleG nds* ⟹ (∃ *x*. *nds* = [*x*,*x*] ∨ (∃ *xs*. *length xs* > *0* ∧ *nds* = [*x*] @ *xs* @ [*x*]))
⟨*proof*⟩

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

lemma *scg-emp*: *Node* = {} ⟹ *scg* ⟨*proof*⟩

lemma *scg-two*:
assumes *Node* = {*nd*,*nd'*}

shows $scg \longleftrightarrow edge\ nd\ nd' \wedge edge\ nd'\ nd$
⟨*proof*⟩

lemma *scg-singl*: $Node = \{nd\} \Longrightarrow scg \longleftrightarrow edge\ nd\ nd$
⟨*proof*⟩

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*⟩

lemma *scg-ex-path*:
assumes *scg* **and** *finite Node* **and** $Node \neq \{\}$
shows $\exists\, ndl.\ pathG\ ndl \wedge set\ ndl = Node$
⟨*proof*⟩

lemma *scg-ex-cycle*:
assumes *scg* **and** *finite Node* **and** $Node \neq \{\}$
shows $\exists\, ndl.\ cycleG\ ndl \wedge set\ ndl = Node$
⟨*proof*⟩

lemma *Graph-scg-restrict*:
$Graph.scg\ N\ e \longleftrightarrow Graph.scg\ N\ (\lambda x\ y.\ e\ x\ y \wedge x \in N \wedge y \in N)$
⟨*proof*⟩

lemma *ex-cycle-scg*:
assumes $cycleG\ ndl\ set\ ndl = Node$
shows *scg*
⟨*proof*⟩

lemma *scg-iff-cycle*:
assumes *finite Node* **and** $Node \neq \{\}$
shows $scg \longleftrightarrow (\exists\, ndl.\ cycleG\ ndl \wedge set\ ndl = Node)$
⟨*proof*⟩

lemma *cycle-from-path*:
assumes *p*: *pathG ndl*
and *ij*: $j < i\ i < length\ ndl\ ndl!i = ndl!j$
shows $cycleG\ (drop\ j\ (take\ (Suc\ i)\ ndl))$
⟨*proof*⟩

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*⟩

definition *ipath* :: *'node stream* ⇒ *bool* **where**
ipath ≡ *alw* (*holdsS Node*) *aand alw* (*holds2 edge*)

lemma *ipath-iff-snth*: *ipath nds* ⟷ (∀ *i*. *nds*!!*i* ∈ *Node* ∧ *edge* (*nds*!!*i*) (*nds*!!(*Suc i*)))
⟨*proof*⟩

lemma *ipath-stake-path*:
ipath nds ⟹ *n* ≥ *2* ⟹ *pathG* (*stake n nds*)
⟨*proof*⟩

lemma *ipath-iff-stake-path*:
ipath nds ⟷ (∀ *n* ≥ *2*. *pathG* (*stake n nds*))
⟨*proof*⟩

lemma *sset-ipath*: *ipath nds* ⟹ *sset nds* ⊆ *Node*
⟨*proof*⟩

lemma *ipath-sdrop*:
ipath nds ⟹ *ipath* (*sdrop n nds*)
⟨*proof*⟩

lemma *ipath-shift*:*local.ipath* (*v* @− *srepeat u*) ⟹ *local.ipath* (*srepeat u*)
⟨*proof*⟩

lemma *ipath-stl*:*ipath r1* ⟹ *local.ipath* (*stl r1*) ⟨*proof*⟩

lemma *ipath-scons*:*ipath* (*r*##*r'*) ⟹ *local.ipath* (*r'*) ⟨*proof*⟩

lemma *ipath-stake-sdrop-path*:
ipath nds ⟹ *m* ≥ *2* ⟹ *pathG* (*stake m* (*sdrop n nds*))
⟨*proof*⟩

lemma *ipath-stake-sdrop-cycle*:
ipath nds ⟹ *m* ≥ *2* ⟹ *nds*!!*n* = *nds*!!(*n*+*m*−*1*) ⟹ *cycleG* (*stake m* (*sdrop n nds*))
⟨*proof*⟩

lemma *ipath-pathCon*:
assumes *nds*: *ipath nds* **and** *ij*: *i* < *j*
shows *pathCon* (*nds*!!*i*) (*nds*!!*j*)
⟨*proof*⟩

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*⟩

lemma *cycle-srepeat-ipath*:
assumes *cycleG ndl* **shows** *ipath* (*srepeat* (*butlast ndl*))
⟨*proof*⟩

lemma *cycle-repeat*:
assumes *ndl*: *cycleG ndl* **and** *n*: $n \neq 0$
shows *cycleG* (*repeat n* (*butlast ndl*) @ [*last ndl*])
⟨*proof*⟩

definition *subgr* **where**
subgr Node1 edge1 Node2 edge2 ≡ *Node1* ⊆ *Node2* ∧ (∀ *nd nd′. edge1 nd nd′* ⟶ *edge2 nd nd′*)

lemma *path-subgr*:
subgr Node1 edge1 S2 R2 ⟹ *Graph.pathG Node1 edge1 ndl* ⟹ *Graph.pathG S2 R2 ndl*
⟨*proof*⟩

lemma *cycle-subgr*:
subgr Node1 edge1 S2 R2 ⟹ *Graph.cycleG Node1 edge1 ndl* ⟹ *Graph.cycleG S2 R2 ndl*
⟨*proof*⟩

lemma *ipath-subgr*:
subgr Node1 edge1 S2 R2 ⟹ *Graph.ipath Node1 edge1 nds* ⟹ *Graph.ipath S2 R2 nds*
⟨*proof*⟩

fun *scsg* :: *'node set* ⇒ (*'node* ⇒ *'node* ⇒ *bool*) ⇒ *bool* **where**
scsg Node1 edge1 ⟷ *subgr Node1 edge1 Node edge* ∧ *Graph.scg Node1 edge1*

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

lemma *scsg-paths*:*scsg V′ E′* ⟹ (∀ *nd nd′*.
{*nd*, *nd′*} ⊆ *V′* ⟶
Graph.pathCon V′ E′ nd nd′)
⟨*proof*⟩

definition *maximal-scsg* **where**

maximal-scsg Node1 edge1 ≡
scsg Node1 edge1 ∧ (∀ *S2 R2*. *scsg S2 R2* ∧ *subgr Node1 edge1 S2 R2* ⟶ *Node1* = *S2* ∧ *edge1* = *R2*)

definition *limitS* :: *'node stream* ⇒ *'node set* **where**
limitS nds ≡ {*nd* ∈ *Node*. *alw* (*ev* (*holds* ((=) *nd*))) *nds*}

definition *limitR* :: *'node stream* ⇒ (*'node* ⇒ *'node* ⇒ *bool*) **where**
limitR nds ≡ λ*nd nd'*. *alw* (*ev* (*holds2* (λ*ndd ndd'*. *ndd* = *nd* ∧ *ndd'* = *nd'*))) *nds*

lemma *limitS-sset*: *limitS nds* ⊆ *sset nds* ⟨*proof*⟩

lemma *ipath-ev-alw*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *ev* (*alw* (*holdsS* (*limitS nds*) *aand holds2* (*limitR nds*))) *nds*
⟨*proof*⟩

lemma *limitS-infinite-visits*:
assumes *x* ∈ *limitS* π
shows ∃ *n* ≥ *m*. π !! *n* = *x*
⟨*proof*⟩

lemma *ipath-sdrop-limit*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows ∃ *i*. *Graph.ipath* (*limitS nds*) (*limitR nds*) (*sdrop i nds*)
⟨*proof*⟩

lemma *limitR-sset*: *limitR nds nd nd'* ⟹ {*nd*,*nd'*} ⊆ *sset nds*
⟨*proof*⟩

lemma *limitR-S*: *ipath nds* ⟹ *limitR nds nd nd'* ⟹ {*nd*,*nd'*} ⊆ *Node*
⟨*proof*⟩

lemma *limitS-S*: *limitS nds* ⊆ *Node* ⟨*proof*⟩

lemma *limitR-R*: *ipath nds* ⟹ *limitR nds nd nd'* ⟹ *edge nd nd'*
⟨*proof*⟩

lemma *scg-limit*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *Graph.scg* (*limitS nds*) (*limitR nds*)
⟨*proof*⟩

lemma *scsg-limit*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *scsg* (*limitS nds*) (*limitR nds*)
⟨*proof*⟩

lemma
assumes *finite Node* **and** *ipath nds*
shows ∃ *Node1 edge1*.
scsg Node1 edge1 ∧
ev (*alw* (*holdsS Node1 aand holds2 edge1*)) *nds* ∧
(∀ *nd nd'*. *edge1 nd nd'* ⟶ *alw* (*ev* (*holds2* (λ*ndd ndd'*. *ndd* = *nd* ∧ *ndd'* = *nd'*))) *nds*)
⟨*proof*⟩

lemma *finite-limitS*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *finite* (*limitS nds*)
⟨*proof*⟩

lemma *S-ne*:
assumes *nds*: *ipath nds*
shows *Node* ≠ {}
⟨*proof*⟩

lemma *R-ne*:
assumes *nds*: *ipath nds*
shows ∃ *nd nd'*. {*nd*,*nd'*} ⊆ *Node* ∧ *edge nd nd'*
⟨*proof*⟩

lemma *limitS-ne*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *limitS nds* ≠ {}
⟨*proof*⟩

lemma *limitR-ne*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows ∃ *nd nd'*. {*nd*,*nd'*} ⊆ *limitS nds* ∧ *limitR nds nd nd'*
⟨*proof*⟩

lemma *finite-limitR*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows *finite* {(*nd*,*nd'*) . *limitR nds nd nd'*}
⟨*proof*⟩

lemma *ipath-limitR-interval*:
assumes *Node*: *finite Node* **and** *nds*: *ipath nds*
shows ∃*j1*≥*i*. ∃*j2*≥*j1*.
∀ *nd nd'*. *limitR nds nd nd'* ⟶
(∃*j*. *j1* ≤ *j* ∧ *j* < *j2* ∧ *nds*!!*j* = *nd* ∧ *nds*!!(*Suc j*) = *nd'*)
⟨*proof*⟩

lemma *limitS-sdrop-eq*[*simp*]: *limitS* (*sdrop n nds*) = *limitS nds*
⟨*proof*⟩

lemma *limitR-sdrop-eq*[*simp*]: *limitR* (*sdrop n nds*) = *limitR nds*
⟨*proof*⟩

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* ⟨*proof*⟩

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 ⟨*proof*⟩

end

definition *SlopedRels* ≡ {*P* . ∀ *h h′ sl1 sl2*. *P h h′ sl1* ∧ *P h h′ sl2* ⟶ *sl1* = *sl2*}

locale *Sloped-Graph* = *Graph Node edge*
for *Node* :: *′node set* **and** *edge* :: *′node* ⇒ *′node* ⇒ *bool*
+

fixes *PosOf* :: *′node* ⇒ *′pos set*
and *RR* :: (*′node* × *′pos*) ⇒ (*′node* × *′pos*) ⇒ *slope* ⇒ *bool*
assumes *Node-finite*: *finite Node*
and *PosOf-finite*: ⋀*nd*. *nd* ∈ *Node* ⟹ *finite* (*PosOf nd*)
and *RR-PosOf*:
⋀*nd P nd′ P′ sl*. *RR* (*nd*,*P*) (*nd′*,*P′*) *sl* ⟹ {*nd*,*nd′*} ⊆ *Node* ∧ *P* ∈ *PosOf nd* ∧ *P′* ∈ *PosOf nd′*
and *RR-SlopeRels*: ⋀*nd nd′*.
{*nd*,*nd′*} ⊆ *Node* ⟹ (λ*P P′*. *RR* (*nd*,*P*) (*nd′*,*P′*)) ∈ *SlopedRels*
begin

lemma *finite-Node-opt*: *finite* ({*r*. ∃ *x*∈*Node*. *r* = *Some x*}:: *′node option set*)
⟨*proof*⟩

lemma *RR-PosOfD*:*RR* (*nd*,*P*) (*nd′*,*P′*) *Main* ∨ *RR* (*nd*,*P*) (*nd′*,*P′*) *Decr* ⟹ *nd* ∈ *Node* ∧ *nd′* ∈ *Node* ∧ *P* ∈ *PosOf nd* ∧ *P′* ∈ *PosOf nd′*
⟨*proof*⟩

lemma *RR-PosOfD′*:*RR* (*nd*,*P*) (*nd′*,*P′*) *s* ⟹ *nd* ∈ *Node* ∧ *nd′* ∈ *Node* ∧ *P* ∈ *PosOf nd* ∧ *P′* ∈ *PosOf nd′*
⟨*proof*⟩

lemma *alw-shd-stl*:*alw* (*holdsS Node*) *x* ⟹ *shd*(*stl x*) ∈ *Node* ⟨*proof*⟩

definition *wfLabF S1 lab* ≡ ∀ *nd*∈*S1*. *lab nd* ∈ *PosOf nd*
definition *wfLabL ndl Pl* ≡ *length ndl* = *length Pl* ∧ (∀ *i*<*length ndl*. *Pl*!*i* ∈ *PosOf* (*ndl*!*i*))
definition *wfLabS nds Ps* ≡ *ev* (*alw* (*holds* (λ(*nd*,*P*). *P* ∈ *PosOf nd*))) (*szip nds Ps*)

definition *wfLabFS Node1 lab* ≡ ∀ *nd*∈*Node1*. *lab nd* ≠ {} ∧ *lab nd* ⊆ *PosOf nd*

lemma *wfLabFS-finite*: *wfLabFS Node1 lab* ⟹ *Node1* ⊆ *Node* ⟹ *nd*∈*Node1* ⟹ *finite* (*lab nd*)
⟨*proof*⟩

lemma *subgr-wfLabFS*:

subgr Node1 edge1 S2 R2 $\Longrightarrow$ *wfLabFS S2 lab* $\Longrightarrow$ *wfLabFS Node1 lab*
⟨*proof*⟩

lemma *wfLabS-iff-snth*:
wfLabS nds Ps $\longleftrightarrow$ ($\exists\, i.\ \forall\, j{\geq}i.$ *Ps*!!*j* $\in$ *PosOf* (*nds*!!*j*))
⟨*proof*⟩

lemma *path-wfLabF-wfLabL*:
assumes *pathG ndl* **and** *wfLabF Node lab*
shows *wfLabL ndl* (*map lab ndl*)
⟨*proof*⟩

lemma *ipath-wfLabF-wfLabS*:
assumes *ipath* (*sdrop i nds*) **and** *wfLabF Node lab*
shows *wfLabS nds* (*smap lab nds*)
⟨*proof*⟩

lemma *wfLabL-tl*: *ndl* $\neq$ [] $\Longrightarrow$ *wfLabL ndl Pl* $\Longrightarrow$ *wfLabL* (*tl ndl*) (*tl Pl*)
⟨*proof*⟩

lemma *wfLabL-append*:
length ndl1 = *length Pl1* $\Longrightarrow$ *length ndl2* = *length Pl2* $\Longrightarrow$
wfLabL (*ndl1* @ *ndl2*) (*Pl1* @ *Pl2*) $\longleftrightarrow$ *wfLabL ndl1 Pl1* $\wedge$ *wfLabL ndl2 Pl2*
⟨*proof*⟩

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*⟩

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*⟩

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

)
(*szip nds Ps*)

lemma *descentIpath-def2*:
descentIpath nds Ps $\longleftrightarrow$
(*ev* (*alw* (*holds2* (λ(*nd*,*P*) (*nd*′,*P*′). *RR* (*nd*,*P*) (*nd*′,*P*′) *Main* ∨ *RR* (*nd*,*P*) (*nd*′,*P*′) *Decr*)))
aand
alw (*ev* (*holds2* (λ(*nd*,*P*) (*nd*′,*P*′). *RR* (*nd*,*P*) (*nd*′,*P*′) *Decr*)))
)
(*szip nds Ps*)
⟨*proof*⟩

lemma *descentIpath-iff-snth2*:
descentIpath nds Ps $\longleftrightarrow$
(∃ *i*. ∀ *j*≥*i*. *RR* (*nds*!!*j*,*Ps*!!*j*) (*nds*!!(*Suc j*),*Ps*!!(*Suc j*)) *Main* ∨
RR (*nds*!!*j*,*Ps*!!*j*) (*nds*!!(*Suc j*),*Ps*!!(*Suc j*)) *Decr*)
∧
(∀ *i*. ∃ *j*≥*i*. *RR* (*nds*!!*j*,*Ps*!!*j*) (*nds*!!(*Suc j*),*Ps*!!(*Suc j*)) *Decr*)
⟨*proof*⟩

lemma *descentIpath-iff-snth*:
descentIpath nds Ps $\longleftrightarrow$
(∃ *i*. (∀ *j*≥*i*. *RR* (*nds*!!*j*,*Ps*!!*j*) (*nds*!!(*Suc j*),*Ps*!!(*Suc j*)) *Main* ∨
RR (*nds*!!*j*,*Ps*!!*j*) (*nds*!!(*Suc j*),*Ps*!!(*Suc j*)) *Decr*)
∧
(∀ *j*≥*i*. ∃ *k*≥*j*. *RR* (*nds*!!*k*,*Ps*!!*k*) (*nds*!!(*Suc k*),*Ps*!!(*Suc k*)) *Decr*))
⟨*proof*⟩

2.3 Infinite Descent

definition *InfiniteDescent* :: *bool* **where**
InfiniteDescent ≡ ∀ *nds*. *ipath nds* ⟶ (∃ *Ps*. *descentIpath nds Ps*)

lemma *InfiniteDescentE*:*InfiniteDescent* ⟹ *ipath nds* ⟹ (⋀*Ps*. *descentIpath nds Ps* ⟹ *P*) ⟹ *P* ⟨*proof*⟩
lemma *InfiniteDescentI*:(⋀*nds*. *ipath nds* ⟹ ∃ *Ps*. *descentIpath nds Ps*)⟹ *InfiniteDescent* ⟨*proof*⟩

lemma *descentIpath-sdrop*: *descentIpath* (*sdrop m nds*) (*sdrop m Ps*) $\longleftrightarrow$ *descentIpath nds Ps*
⟨*proof*⟩

lemma *descentIpath-stl*: *descentIpath* (*stl nds*) (*stl Ps*) $\longleftrightarrow$ *descentIpath nds Ps*

⟨*proof*⟩

lemma *descentIpath-wfLabS*:
descentIpath nds Ps ⟹ *wfLabS nds Ps*
⟨*proof*⟩

lemma *descentIpath-sdrop-any*:
descentIpath (*sdrop m nds*) *Ps'* ⟹ ∃ *Ps*. *descentIpath nds Ps*
⟨*proof*⟩

lemma *descentIpath-grow*:*descentIpath r1 Ps* = *descentIpath* (*x* ## *r1*) (*y* ## *Ps*)
⟨*proof*⟩

lemma *ipath-stake-cycle*:*local.ipath* (*srepeat u*) ⟹
2 ≤ *length u* ⟹
srepeat u !! *0* = *srepeat u* !! (*length u* − *1*) ⟹
cycleG u
⟨*proof*⟩

lemma *descentIpath-reduceAll*:∀ *x*. ¬ *descentIpath* (*v* @− *srepeat u*) *x* ⟹ ∀ *x*. ¬ *descentIpath* (*srepeat u*) *x*
⟨*proof*⟩

definition *descentPath* :: *'node list* ⇒ *'pos list* ⇒ *bool* **where**
descentPath ndl Pl ≡
(∀ *i*. *Suc i* < *length ndl* ⟶ *RR* (*ndl*!*i*,*Pl*!*i*) (*ndl*!(*Suc i*),*Pl*!(*Suc i*)) *Main* ∨
RR (*ndl*!*i*,*Pl*!*i*) (*ndl*!(*Suc i*),*Pl*!(*Suc i*)) *Decr*) ∧
(∃ *i*. *Suc i* < *length ndl* ∧ *RR* (*ndl*!*i*,*Pl*!*i*) (*ndl*!(*Suc i*),*Pl*!(*Suc i*)) *Decr*)

lemma *descentPath-length-wfLabL*:
descentPath ndl Pl ⟹ *length Pl* = *length ndl* ⟹ *wfLabL ndl Pl*
⟨*proof*⟩

lemma *cycle-descentIPath-srepeat-imp-descentPath*:
assumes *1*: *cycleG ndl* **and** *2*: *descentIpath* (*srepeat* (*butlast ndl*)) *Ps*
shows ∃ *Pl*. *wfLabL ndl Pl* ∧ *descentPath ndl Pl*
⟨*proof*⟩

definition *descentIpathS* :: *'node stream ⇒ 'pos stream ⇒ bool* **where**
descentIpathS nds Ps ≡
(∀ i. RR (nds !! i, Ps !! i) (nds !! Suc i, Ps !! Suc i) Main ∨
RR (nds !! i, Ps !! i) (nds !! Suc i, Ps !! Suc i) Decr)
∧
(∀ i. ∃ j≥i. RR (nds !! j, Ps !! j) (nds !! Suc j, Ps !! Suc j) Decr)

lemma *descentIpathS-imp-descentIpath*:
descentIpathS nds Ps ⟹ descentIpath nds Ps
⟨*proof*⟩

lemma *cycle-descentIPathS-srepeat-imp-descentPath*:
cycleG ndl ⟹ descentIpathS (srepeat (butlast ndl)) Ps ⟹
∃ Pl. wfLabL ndl Pl ∧ descentPath ndl Pl
⟨*proof*⟩

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 *∃ Ps. descentIpathS (srepeat (butlast ndl)) Ps*
⟨*proof*⟩

lemma *cycle-descentPath-repeat-imp-descentIPathS-srepeat*:
assumes *ndl*: *cycleG ndl* **and** *n*: *n ≠ 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 *∃ Ps. descentIpathS (srepeat (butlast ndl)) Ps*
⟨*proof*⟩

lemma *srepeat-cycle-descentIpath-imp-descentIpath*:
assumes *ndl*: *cycleG ndl*
and *d*: *descentIpath (srepeat (butlast ndl)) Ps*
shows *∃ Ps. descentIpathS (srepeat (butlast ndl)) Ps*
⟨*proof*⟩

lemma *cycle-descentIPathS-srepeat-imp-descentPath-repeat*:
assumes *ndl*: *cycleG ndl* **and** *d*: *descentIpathS (srepeat (butlast ndl)) Ps*
shows *∃ n Pl. n ≠ 0 ∧ wfLabL (repeat n (butlast ndl) @ [last ndl]) Pl ∧*
descentPath (repeat n (butlast ndl) @ [last ndl])
Pl ∧ hd Pl = last Pl
⟨*proof*⟩

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

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** $q_0 \equiv None$

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

lemma Δ-*trans-q*$_0$:*l* $\in$ *Node* $\Longrightarrow$ *Some l* $\in$ Δ-*trans l* q_0 $\langle proof \rangle$

lemma Δ-*trans-edge*:*edge s tr* $\Longrightarrow$ $\{s, tr\} \subseteq Node \Longrightarrow$ (*Some tr*) $\in$ Δ-*trans tr* (*Some s*) $\langle proof \rangle$

lemma Δ-*trans-elim*:
assumes $r \in \Delta$-*trans x q*
obtains (*EdgeCase*) $\exists\, q'.\ q = Some\ q' \wedge edge\ q'\ x \wedge \{q', x\} \subseteq Node \wedge r = (Some\ x)$
| (*InitCase*) $q = q_0\ x \in Node\ r = (Some\ x)$
$\langle proof \rangle$

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

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* $\langle proof \rangle$
lemma $Paut_V$-*accept*[*simp*]:*nba.accepting* $Paut_V$ = ($\lambda s.$ *the s* $\in$ *Node*) $\langle proof \rangle$
lemma $Paut_V$-*init*[*simp*]: *nba.initial* $Paut_V$ = $\{q_0\}$ $\langle proof \rangle$
lemma $Paut_V$-*initp*[*intro*]:*p* $\in$ *nba.initial* $Paut_V$ $\longleftrightarrow$ *p* = q_0 $\langle proof \rangle$
lemma $Paut_V$-*trans*[*simp*]:*nba.transition* $Paut_V$ *a b* = Δ-*trans a b* $\langle proof \rangle$

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

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

⟨*proof*⟩

fun Δ-*trans′* :: *′node* ⇒ (*′node* × *′pos* × *slope*) *option* ⇒
(*′node* × *′pos* × *slope*) *option set* **where**
Δ-*trans′ v′* (*Some tr*) = (*case tr of* (*v*, *p*, *s*) ⇒ {*Some* (*v′*, *p′*, *s′*)| *p′ s′*. *RR* (*v*, *p*) (*v′*, *p′*) *s′*})|
Δ-*trans′ v′ q*$_0$ = (*if v′* ∈ *Node then* {*q*$_0$} ∪ {*Some* (*v′*, *p′*, *s′*)| *p′ s′*. *p′* ∈ *PosOf v′* ∧ *s′* = *Main*} *else* {})

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

lemma *q*$_0$*′-notDecr*:*third r* = *Decr* ⟹ ¬ (*Some r*) ∈ Δ-*trans′ x q*$_0$ ⟨*proof*⟩

lemma Δ-*trans-q*$_0$*′I*:*v* ∈ *Node* ⟹ *q*$_0$ ∈ Δ-*trans′ v q*$_0$ ⟨*proof*⟩

lemma Δ-*trans′-intro*:
assumes *v′* ∈ *Node*
assumes *tr* = *q*$_0$ ⟹ *x* = *q*$_0$ ∨ (∃ *p′ s′*. *x* = *Some* (*v′*, *p′*, *s′*) ∧ *p′* ∈ *PosOf v′* ∧ *s′* = *Main*)
assumes *tr* ≠ *q*$_0$ ⟹ (∃ *p′ s′*. *x* = *Some* (*v′*, *p′*, *s′*) ∧ *RR* (*fst*(*the tr*), *second*(*the tr*)) (*v′*, *p′*) *s′*)
shows *x* ∈ Δ-*trans′ v′ tr*
⟨*proof*⟩

lemma Δ-*trans′-elim*:
assumes *v′* ∈ Δ-*trans′ v*$_t$ *q*
obtains
(Δ-*trans*$_1$) *q* = *q*$_0$ *v*$_t$ ∈ *Node v′* = *q*$_0$
| (Δ-*trans*$_2$) *q* = *q*$_0$ *v*$_t$ ∈ *Node* ∃ *v″ p′ s′*. *v′* = *Some* (*v″*, *p′*,*s′*) ∧ *p′* ∈ *PosOf v″* ∧ *s′* = *Main* ∧ *v*$_t$ = *v″*
| (Δ-*trans*$_3$) ∃ *v p s v″ p′ s′*. *q* = *Some* (*v*, *p*, *s*) ∧ *v′* = *Some* (*v″*, *p′*,*s′*) ∧ *RR* (*v*, *p*) (*v″*, *p′*) *s′* ∧ *v*$_t$ = *v″ v*$_t$ ∈ *Node*
⟨*proof*⟩

lemma Δ-*trans′-elim-q*$_0$*′-target*:
assumes *q*$_0$ ∈ Δ-*trans′ v a*
obtains *q*$_0$ = *a v* ∈ *Node*
⟨*proof*⟩

lemma Δ-*trans′-elim-q*$_0$*′*:
assumes *v′* ∈ Δ-*trans′ v*$_t$ *q*$_0$
obtains
(Δ-*trans*$_1$) *v*$_t$ ∈ *Node v′* = *q*$_0$
| (Δ-*trans*$_2$) *v*$_t$ ∈ *Node second* (*the v′*) ∈ *PosOf* (*fst* (*the v′*)) *third* (*the v′*) = *Main v*$_t$ = *fst* (*the v′*)
⟨*proof*⟩

lemma q_0'*-notReachable*:$q \neq q_0 \Longrightarrow v' \in \Delta$*-trans'* $v\ q \Longrightarrow v' \neq q_0$ ⟨*proof*⟩

lemma Δ*-trans'-v-eq*:$v' \neq q_0 \Longrightarrow v' \in \Delta$*-trans'* $v\ q \Longrightarrow$ *fst* (*the* v') $= v$
⟨*proof*⟩

lemma Δ*-trans'-Decr-not-*q_0:*Some* (v, p, *Decr*) $\in \Delta$*-trans'* $vs\ q \Longrightarrow \exists\, v'\ p\ s.\ q =$ *Some* (v',p,s) $\wedge\ vs = v$
⟨*proof*⟩

lemma Δ*-trans'-DecrRR*:
$v' \neq$ *None* $\Longrightarrow$
fst (*the* v') $\in$ *Node* $\Longrightarrow$
second (*the* v') $\in$ *PosOf* (*fst* (*the* v')) $\Longrightarrow$
third (*the* v') $=$ *Decr* $\Longrightarrow$ $v' \in \Delta$*-trans'* $v\ q \Longrightarrow$
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*))
⟨*proof*⟩

lemma Δ*-trans'-DecrRR'*:
Some (v',p',*Decr*) $\in \Delta$*-trans'* $v\ q \Longrightarrow$
$v' \in$ *Node* $\Longrightarrow$ $p' \in$ *PosOf* $v' \Longrightarrow$
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)))
⟨*proof*⟩

lemma Δ*-trans'-ProgDRR*:
$q \neq q_0 \Longrightarrow$
fst (*the* q) $\in$ *Node* $\Longrightarrow$ *second* (*the* q) $\in$ *PosOf* (*fst* (*the* q)) $\Longrightarrow$ *third* (*the* q) $=$ *Decr* $\Longrightarrow$
$v' \in \Delta$*-trans'* $v\ q \Longrightarrow$
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*) $\wedge\ v' \neq q_0$
⟨*proof*⟩

lemma Δ*-trans'-ProgMRR*:
$q \neq q_0 \Longrightarrow$
$v' \in \Delta$*-trans'* $v\ q \Longrightarrow$
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*) $\wedge\ v' \neq q_0$
⟨*proof*⟩

lemma Δ*-trans'-ProgMRR'-Main*:
$x \in$ *Node* $\Longrightarrow$ $y \in$ *PosOf* $x \Longrightarrow$
Some (x',y',*Main*) $\in \Delta$*-trans'* x' (*Some* (x,y,z)) $\Longrightarrow$
RR (x, y) (x', y') *Main*
⟨*proof*⟩

definition *Q′-states*::(*′node* × *′pos* × *slope*) *option set* **where**
Q′-states ≡ {q_0} ∪ {*Some* (*v*, *p*, *s*) |*v p s*. *v* ∈ *Node* ∧ *p* ∈ *PosOf v*}

definition *F-valid*::(*′node* × *′pos* × *slope*) *option* ⇒ *bool* **where**
F-valid ≡ λ*s*. ∃ *r1 r2*. *s* = *Some* (*r1*, *r2*, *Decr*) ∧ *r1* ∈ *Node* ∧ *r2* ∈ *PosOf r1*

definition $Taut_V$ :: (*′node*, (*′node* × *′pos* × *slope*) *option*) *nba* **where**
$Taut_V$ = *nba*
Node
{q_0}
Δ-*trans′*
F-valid

lemma *RR-red*:(∀ *n*≥*Suc 0*. *RR* (*r1* !! *n*, *Ps* !! *n*) (*stl r1* !! *n*, *stl Ps* !! *n*) (*stl Ss* !! *n*)) ⟷
(∀ *n*. *RR* (*stl r1* !! *n*, *stl Ps* !! *n*) (*stl*(*stl r1*) !! *n*, *stl*(*stl Ps*) !! *n*) (*stl*(*stl Ss*) !! *n*))
⟨*proof*⟩
lemma $Taut_V$-*alpha*[*simp*]:*nba.alphabet* $Taut_V$ = *Node* ⟨*proof*⟩
lemma $Taut_V$-*accept*[*simp*]:*nba.accepting* $Taut_V$ = (λ*s*. ∃ *r1 r2*. *s* = *Some* (*r1*, *r2*, *Decr*) ∧ *r1* ∈ *Node* ∧ *r2* ∈ *PosOf r1*) ⟨*proof*⟩
lemma $Taut_V$-*init*[*simp*]: *nba.initial* $Taut_V$ = {q_0} ⟨*proof*⟩
lemma $Taut_V$-*initp*[*intro*]:*p* ∈ *nba.initial* $Taut_V$ ⟷ *p* = q_0 ⟨*proof*⟩
lemma $Taut_V$-*trans*[*simp*]:*nba.transition* $Taut_V$ *a b* = Δ-*trans′ a b* ⟨*proof*⟩

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

lemma $Taut_V$-*lang*:*NBA.language* $Taut_V$ = {*nds*. (∃ *Ps*. *descentIpath nds Ps*) ∧ *alw* (*holdsS Node*) *nds*}
⟨*proof*⟩

lemma *alpha-subseq-PTaut*$_V$: *nba.alphabet* $Paut_V$ ⊆ *nba.alphabet* $Taut_V$ ⟨*proof*⟩

lemma *Pnode-subseq-rule*:(⋀*r*. ∀ *x*∈*Node*. *last* (*map snd r*) ≠ *Some x* ⟹
r ≠ [] ⟹ *NBA.path* $Paut_V$ *r None* ⟹ *last* (*map snd r*) = *None*) ⟹
(⋃ *p*∈{*p*. *p* ∈ *nba.initial* $Paut_V$}. {*last* (*p* # *map snd r*) |*r*. *NBA.path* $Paut_V$ *r p*})
⊆ {*r*. *r* = *None* ∨ (∃ *x*∈*Node*. *r* = *Some x*)} ⟨*proof*⟩

lemma *Paut$_V$-node-subseq*:*NBA.nodes Paut$_V$* $\subseteq$ $\{r.\ r = None \vee (\exists x{\in}\ Node.\ r = Some\ x)\}$
⟨*proof*⟩

lemma *finite-Nodes-Paut$_V$*:*finite* (*NBA.nodes Paut$_V$*)
⟨*proof*⟩

lemma *Tnode-subseq-rule*:$(\bigwedge r.\ \forall a.\ a \in Node \longrightarrow (\forall aa.\ aa \in PosOf\ a \longrightarrow (\forall b.\ last\ (map\ snd\ r) \neq Some\ (a,\ aa,\ b))) \Longrightarrow$
$r \neq [] \Longrightarrow NBA.path\ Taut_V\ r\ None \Longrightarrow last\ (map\ snd\ r) = None)$
$\Longrightarrow (\bigcup p{\in}\{p.\ p \in nba.initial\ Taut_V\}.\ \{last\ (p\ \#\ map\ snd\ r)\ |r.\ NBA.path\ Taut_V\ r\ p\})$
$\subseteq \{r.\ r = None \vee (\exists x{\in}\{(v,\ p,\ s)\ |v\ p\ s.\ v \in Node \wedge p \in PosOf\ v\}.\ r = Some\ x)\}$ ⟨*proof*⟩

lemma *Taut$_V$-node-subseq*:*NBA.nodes Taut$_V$* $\subseteq$ $\{r.\ r = None \vee (\exists x{\in}\{(v,\ p,\ s)\ |v\ p\ s.\ v \in Node \wedge p \in PosOf\ v\}.\ r = Some\ x)\}$
⟨*proof*⟩

lemma *set-slope-case*:$\{(v,\ p,\ s)\ |v\ p\ s.\ v \in Node \wedge p \in PosOf\ v\} = \{(v,\ p,\ s)\ |v\ p\ s.\ v \in Node \wedge p \in PosOf\ v \wedge s \in\{Main,\ Decr\}\}$ ⟨*proof*⟩

lemma *finite-Node-Taut$_V$-gr*:*finite* $(\{r.\ \exists x{\in}\{(v,\ p,\ s)\ |v\ p\ s.\ v \in Node \wedge p \in PosOf\ v\}.\ r = Some\ x\}$:: $('node \times 'pos \times slope)\ option\ set)$
⟨*proof*⟩

lemma *finite-Nodes-Taut$_V$*:*finite* (*NBA.nodes Taut$_V$*)
⟨*proof*⟩

theorem *VLA-Criterion*:*InfiniteDescent* $\longleftrightarrow$ *NBA.language Paut$_V$* $\subseteq$ *NBA.language Taut$_V$*
⟨*proof*⟩

corollary *VLA-Criterion'*:*InfiniteDescent* $\longleftrightarrow$ *NBA.language Paut$_V$* $\cap$ (*NBA.language* (*complement Taut$_V$*)) = {}
⟨*proof*⟩
end

end

3.1.2 Slope-language Criterion

theory *SLA-Criterion*
imports *../Sloped-Graphs*

../Buchi-Preliminaries
begin

context *Sloped-Graph*
begin

abbreviation q_0 **where** $q_0 \equiv None$
fun RR':: $'node \times 'node \Rightarrow 'pos \times 'pos \times slope \Rightarrow bool$ **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 $\{Some\ nd' |\ nd.\ edge\ nd\ nd'\}$

fun Δ_{sl} :: $('pos \times 'pos \times slope \Rightarrow bool) \Rightarrow 'node\ option \Rightarrow 'node\ option\ set$ **where**
$\Delta_{sl}\ t\ (Some\ nd) = \{Some\ nd' |\ nd'.\ \{nd,\ nd'\} \subseteq Node \wedge edge\ nd\ nd' \wedge t = RR'(nd,nd')\ \}|$
$\Delta_{sl}\ t\ q_0 = \{Some\ nd' |\ nd\ nd'.\ \{nd,\ nd'\} \subseteq Node \wedge edge\ nd\ nd' \wedge t = RR'(nd,nd')\}$

lemma Δ_{sl}-*intro-Some*:
assumes *edge s s'*
and $\{s,\ s'\} \subseteq Node$
and $tr = RR'(s,s')$
shows $Some\ s' \in \Delta_{sl}\ tr\ (Some\ s)$
⟨*proof*⟩

lemma Δ_{sl}-*intro-q0*:
assumes *edge s s'*
and $\{s,\ s'\} \subseteq Node$
and $tr = RR'(s,s')$
shows $Some\ s' \in \Delta_{sl}\ tr\ q_0$
⟨*proof*⟩

lemma Δ_{sl}-*elim*:
assumes $x \in \Delta_{sl}\ t\ z$
obtains *nd nd'* **where**
$x = Some\ nd'\ nd \in Node\ nd' \in Node\ edge\ nd\ nd'\ t = RR'\ (nd,\ nd')$
$z = q_0 \vee the\ z = nd$
⟨*proof*⟩

lemma Δ_{sl}-q_0-*not*-q_0[*simp*]: $\neg\ (q_0 \in \Delta_{sl}\ x\ q_0)$ ⟨*proof*⟩

lemma Δ_{sl}-*some*:$r' \in \Delta_{sl}\ x\ r \Longrightarrow \exists\, y.\ r' = Some\ y$ ⟨*proof*⟩

lemma $\Delta_{sl}\ l\ s =$
$\{Some\ v' |\ v\ v'.\ s = q_0 \wedge (edge\ v\ v') \wedge \{v,\ v'\} \subseteq Node \wedge l = RR'(v,v')\} \cup$

$\{Some\ v' |\ v'.\ s \neq q_0 \wedge edge\ (the\ s)\ v' \wedge \{(the\ s),\ v'\} \subseteq Node \wedge l = RR'((the\ s),v')\}$
$\langle proof \rangle$

definition $Paut_R :: ('pos \times 'pos \times slope \Rightarrow bool,\ 'node\ option)\ nba$ **where**
$Paut_R = nba$
$\{RR'\ (nd,\ nd') |\ nd\ nd'.\ edge\ nd\ nd'\}$
$\{q_0\}$
Δ_{sl}
$(\lambda s.\ the\ s \in Node)$

lemmas $Paut_R\text{-}defs = Paut_R\text{-}def\ \Delta_{sl}.simps$
lemma $Paut_R\text{-}alpha[simp]:nba.alphabet\ Paut_R = \{RR'\ (nd,\ nd') |\ nd\ nd'.\ edge\ nd\ nd'\}\ \langle proof \rangle$
lemma $Paut_R\text{-}accept[simp]:nba.accepting\ Paut_R = (\lambda s.\ the\ s \in Node)\ \langle proof \rangle$
lemma $Paut_R\text{-}init[simp]:\ nba.initial\ Paut_R = \{q_0\}\ \langle proof \rangle$
lemma $Paut_R\text{-}initp[intro]:p \in nba.initial\ Paut_R \longleftrightarrow p = q_0\ \langle proof \rangle$
lemma $Paut_R\text{-}trans[simp]:nba.transition\ Paut_R\ a\ b = \Delta_{sl}\ a\ b\ \langle proof \rangle$

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

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

lemma $Paut_R\text{-}lang\text{-}in:x \in NBA.language\ Paut_R \longleftrightarrow (\exists\, nds.\ ipath\ nds \wedge (\forall\, i.\ x\ !!\ i = RR'\ (nds\ !!\ i,\ nds\ !!\ Suc\ i)))$
$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

lemma Δ_{sl}'*-intro-q0*:
assumes $R' \in \Sigma$ $(p', s') \in Q'_{sl}$
shows *Some* $(p', s') \in \Delta_{sl}'\ R'\ q_0$
$\langle proof \rangle$

lemma Δ_{sl}'*-q0-included*:
assumes $R' \in \Sigma$
shows $q_0 \in \Delta_{sl}'\ R'\ q_0$
$\langle proof \rangle$

lemma Δ_{sl}'*-intro*:
assumes $ns \in \Sigma$
assumes $rk \neq q_0 \Longrightarrow (\exists\ p'\ s'.\ rk' = Some\ (p', s') \wedge the\ rk \in Q'_{sl} \wedge ns\ (fst(the\ rk), p', s'))$
assumes $rk = q_0 \Longrightarrow (rk' = q_0 \vee (\exists\, p'\ s'.\ rk' = Some\ (p', s') \wedge (p', s') \in Q'_{sl}))$
shows $rk' \in \Delta_{sl}'\ ns\ rk$
$\langle proof \rangle$

lemma Δ_{sl}'*-elim*:
assumes $x \in \Delta_{sl}'\ R'\ z$
obtains (*SomeCase*) $p\ s\ p'\ s'$ **where**
$z = Some\ (p, s)\ x = Some\ (p', s')\ (p, s) \in Q'_{sl}\ R' \in \Sigma\ R'\ (p, p', s')$
| (*q0Case*) $p'\ s'$ **where**
$z = q_0\ R' \in \Sigma\ (p', s') \in Q'_{sl}\ x = Some\ (p', s')$
| (*q0Self*) $z = q_0\ R' \in \Sigma\ x = q_0$
$\langle proof \rangle$

lemma q_0*-notReachable*:$q \neq q_0 \Longrightarrow v' \in \Delta_{sl}'\ v\ q \Longrightarrow v' \neq q_0$ $\langle proof \rangle$

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

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

lemma *RR-reduce*:$(\forall\, n{\geq}Suc\ 0.\ RR\ (r1\ !!\ n,\ Ps\ !!\ n)\ (stl\ r1\ !!\ n,\ stl\ Ps\ !!\ n)\ (stl\ Ss\ !!\ n)) \longleftrightarrow$
$(\forall\, n.\ RR\ (stl\ r1\ !!\ n,\ stl\ Ps\ !!\ n)\ (stl(stl\ r1)\ !!\ n,\ stl(stl\ Ps)\ !!\ n)\ (stl(stl\ Ss)\ !!\ n))$
$\langle proof\rangle$

lemma *Taut*$_R$*-alpha*[*simp*]:*nba.alphabet Taut*$_R$ $= \{RR'\ (nd,\ nd')|\ nd\ nd'.\ edge\ nd\ nd'\}$ $\langle proof\rangle$
lemma *Taut*$_R$*-accept*[*simp*]:*nba.accepting Taut*$_R$ $= (\lambda ps.\ \exists\, p.\ ps = Some\ (p,\ Decr) \wedge (p,\ Decr) \in Q'_{sl})$ $\langle proof\rangle$
lemma *Taut*$_R$*-init*[*simp*]: *nba.initial Taut*$_R$ $= \{q_0\}$ $\langle proof\rangle$
lemma *Taut*$_R$*-initp*[*intro*]:$p \in nba.initial\ Taut_R \longleftrightarrow p = q_0$ $\langle proof\rangle$
lemma *Taut*$_R$*-trans*[*simp*]:*nba.transition Taut*$_R$ $a\ b = \Delta_{sl}'\ a\ b$ $\langle proof\rangle$

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* $(\lambda i.\ RR'\ (nds\ !!\ i,\ nds\ !!\ Suc\ i))\ nats$

lemma *stl-shift*:$(stl\ nds\ !!\ i, stl\ (stl\ nds)\ !!\ i) = (nds\ !!\ Suc\ i, (stl\ nds)\ !!\ Suc\ i)$ $\langle proof\rangle$

lemma *smap-shifted-eq*:
$smap\ (\lambda i.\ RR'\ (nds\ !!\ i, stl\ nds\ !!\ i))\ (fromN\ (Suc\ 0)) =$
$smap\ (\lambda i.\ RR'\ (stl\ nds\ !!\ i, stl\ (stl\ nds)\ !!\ i))\ nats$
$\langle proof\rangle$

lemma *Rst-correct*:$x = Rst\ nds \longleftrightarrow (\forall\, i.\ x\ !!\ i = RR'\ (nds\ !!\ i,\ nds\ !!\ Suc\ i))$
$\langle proof\rangle$

lemma *Rst-r*:$\bigwedge k.\ Rst\ nds\ !!\ k = RR'\ (nds\ !!\ k,\ nds\ !!\ Suc\ k)\langle proof\rangle$

lemma *list-swap*:$k{>}0 \Longrightarrow (p\ \#\#\ smap\ (\lambda r.\ fst\ (the\ r))\ r)\ !!\ k = fst\ (the\ (r\ !!\ (k{-}1)))$ $\langle proof\rangle$
lemma *Taut*$_R$*-lang-in*:*ipath nds* $\Longrightarrow$ *Rst nds* $\in$ *NBA.language Taut*$_R \longleftrightarrow (\exists\, Ps.$ *descentIpath nds Ps*)
$\langle proof\rangle$

lemma *alpha-subseq-PTaut$_R$*: *nba.alphabet Paut$_R$* $\subseteq$ *nba.alphabet Taut$_R$* ⟨*proof*⟩

lemma *Paut$_R$-subseq-rule*:($\bigwedge r.\ \forall x{\in}Node.\ last\ (map\ snd\ r) \neq Some\ x \Longrightarrow$
$r \neq [] \Longrightarrow NBA.path\ Paut_R\ r\ None \Longrightarrow last\ (map\ snd\ r) = None) \Longrightarrow$
$(\bigcup p{\in}\{p.\ p \in nba.initial\ Paut_R\}.\ \{last\ (p\ \#\ map\ snd\ r)\ |r.\ NBA.path\ Paut_R\ r\ p\})$
$\subseteq \{r.\ r = None \vee (\exists x{\in}Node.\ r = Some\ x)\}$ ⟨*proof*⟩

lemma *Paut$_R$-node-subseq*:*NBA.nodes Paut$_R$* $\subseteq \{r.\ r = None \vee (\exists x{\in}\ Node.\ r = Some\ x)\}$
⟨*proof*⟩

lemma *finite-Nodes-Paut$_R$*:*finite* (*NBA.nodes Paut$_R$*)
⟨*proof*⟩

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*)) $\Longrightarrow$
$r \neq [] \Longrightarrow NBA.path\ Taut_R\ r\ None \Longrightarrow last\ (map\ snd\ r) = None)$
$\Longrightarrow (\bigcup p{\in}\{p.\ p \in nba.initial\ Taut_R\}.\ \{last\ (p\ \#\ map\ snd\ r)\ |r.\ NBA.path\ Taut_R$
$r\ p\})$
$\subseteq \{r.\ r = None \vee (\exists x{\in}\{(p,\ s)|\ p\ s.\ p \in (\bigcup v{\in}Node.\ PosOf\ v)\}.\ r = Some\ x)\}$
⟨*proof*⟩

theorem *SLA-Criterion*:*InfiniteDescent* $\longleftrightarrow$ *NBA.language Paut$_R$* $\subseteq$ *NBA.language Taut$_R$*
⟨*proof*⟩

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*⟩

context *Sloped-Graph*
begin

definition *MaxSl* :: *slope set* ⇒ *slope* **where**
MaxSl sll ≡ *if Decr* ∈ *sll then Decr else Main*

lemma *MaxSl-singl*[*simp*]: *MaxSl* {*sl*} = *sl*
⟨*proof*⟩

definition *leqSl* :: (*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) ⇒
(*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) ⇒ *bool*
where
leqSl P1 P2 ≡ ∀ *h h' sl1*. *P1 h h' sl1* ⟶ (∃ *sl2*. *sl1* ≤ *sl2* ∧ *P2 h h' sl2*)

definition *lessSl* :: (*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) ⇒
(*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) ⇒ *bool*
where *lessSl P1 P2* ≡ *leqSl P1 P2* ∧ *P1* ≠ *P2*

lemma *leqSl-refl*: *leqSl P P*
⟨*proof*⟩

lemma *leqSl-trans*: *leqSl P1 P2* ⟹ *leqSl P2 P3* ⟹ *leqSl P1 P3*
⟨*proof*⟩

lemma *leqSl-antisym-aux*:
assumes *P12*: {*P1*,*P2*} ⊆ *SlopedRels* **and** *12*: *leqSl P1 P2* **and** *21*: *leqSl P2 P1*
and *0*: *P1 h h' sl*
shows *P2 h h' sl*
⟨*proof*⟩

lemma *leqSl-antisym*:
assumes *P12*: {*P1*,*P2*} ⊆ *SlopedRels* **and** *12*: *leqSl P1 P2* **and** *21*: *leqSl P2 P1*
shows *P1* = *P2*
⟨*proof*⟩

lemma *lessSl-antisym*: {*P1*,*P2*} ⊆ *SlopedRels* ⟹ ¬ (*lessSl P1 P2* ∧ *leqSl P2 P1*)

⟨*proof*⟩

lemma *lessSl-trans*: $\{P1,P2,P3\} \subseteq SlopedRels \Longrightarrow lessSl\ P1\ P2 \Longrightarrow lessSl\ P2\ P3 \Longrightarrow lessSl\ P1\ P3$
⟨*proof*⟩

inductive *transSl* :: $('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool) \Rightarrow ('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool)$
for $K :: 'pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool$ **where**
Base: $K\ P\ P'\ sl \Longrightarrow transSl\ K\ P\ P'\ sl$
|*Step*: $transSl\ K\ P\ P'\ sl1 \Longrightarrow K\ P'\ P''\ sl2 \Longrightarrow transSl\ K\ P\ P''\ (MaxSl\ \{sl1,sl2\})$

lemma *transSl-mono*: **assumes** *leqSl P Q*
shows *leqSl* (*transSl P*) (*transSl Q*)
⟨*proof*⟩

definition *initFrag* ::
$('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool)\ set \Rightarrow ('pos \Rightarrow 'pos \Rightarrow slope \Rightarrow bool)\ set \Rightarrow bool$
where
$initFrag\ LL'\ LL \equiv \forall R{\in}\ LL.\ \exists R' \in LL'.\ leqSl\ R'\ R$

lemma *initFrag-Un*: $initFrag\ L\ (LL1 \cup LL2) \longleftrightarrow initFrag\ L\ LL1 \wedge initFrag\ L\ LL2$
⟨*proof*⟩

lemma *initFrag-trans*: $initFrag\ L1\ L2 \Longrightarrow initFrag\ L2\ L3 \Longrightarrow initFrag\ L1\ L3$
⟨*proof*⟩

definition *Dt* **where**
$Dt\ LL \equiv \{R \in LL.\ \neg\ (\exists R' \in LL.\ lessSl\ R'\ R)\}$

lemma *Dt-incl*: $Dt\ LL \subseteq LL$ ⟨*proof*⟩

lemma *Dt-aux*: $\bigwedge LL\ LL'.\ LL \subseteq LL' \Longrightarrow Dt\ LL \subseteq LL'$ ⟨*proof*⟩

lemma *initFrag-Dt*:
assumes *LL*: $LL \subseteq SlopedRels$ **and** *f-LL*: *finite LL*
shows *initFrag* (*Dt LL*) *LL*
⟨*proof*⟩

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 ≡ ∃ *P'' sl1 sl2*. *sl* = *MaxSl* {*sl1*, *sl2*} ∧ *K1 P P'' sl1* ∧ *K2 P'' P' sl2*

lemma *finite-slope-set*: *finite* (*S*::*slope set*)
⟨*proof*⟩

lemma *ccompSl-mono*: *leqSl P1 Q1* ⟹ *leqSl P2 Q2* ⟹ *leqSl* (*ccompSl P1 P2*) (*ccompSl Q1 Q2*)
⟨*proof*⟩

lemma *ccompSl-SlopeRels*:
{*P*,*P'*} ⊆ *SlopedRels* ⟹ *ccompSl P P'* ∈ *SlopedRels*
⟨*proof*⟩

fun *Ccl-iter* :: *nat* ⇒ *'node* ⇒ *'node* ⇒ (*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) *set* **where**
Ccl-iter 0 nd nd' = (*if* {*nd*,*nd'*} ⊆ *Node* ∧ *edge nd nd' then* {λ*P P' sl*. *RR* (*nd*,*P*) (*nd'*,*P'*) *sl*} *else* {})
|*Ccl-iter* (*Suc i*) *nd nd'* =
Ccl-iter i nd nd' ∪
{*ccompSl K1 K2* | *K1 K2 nd''*. *nd''* ∈ *Node* ∧ *K1* ∈ *Ccl-iter i nd nd''* ∧ *K2* ∈ *Ccl-iter i nd'' nd'*}

lemma *Ccl-iter-nodes*: *nd* ∉ *Node* ∨ *nd'* ∉ *Node* ⟹ *Ccl-iter i nd nd'* = {}
⟨*proof*⟩

lemma *Ccl-iter-PosOf*:
K ∈ *Ccl-iter i nd nd'* ⟹ *K P P' sl* ⟹ *P* ∈ *PosOf nd* ∧ *P'* ∈ *PosOf nd'*
⟨*proof*⟩

lemma *Ccl-iter-Suc-mono*: *Ccl-iter i nd nd'* ⊆ *Ccl-iter* (*Suc i*) *nd nd'*
⟨*proof*⟩

lemma *Ccl-iter-mono*: *i* ≤ *j* ⟹ *Ccl-iter i nd nd'* ⊆ *Ccl-iter j nd nd'*
⟨*proof*⟩

lemma *Ccl-iter-Suc-stable*:
assumes *Ccl-iter* (*Suc i*) = *Ccl-iter i*
shows *Ccl-iter* (*Suc* (*Suc i*)) = *Ccl-iter i*
⟨*proof*⟩

lemma *Ccl-iter-stable*:
assumes *1*: *Ccl-iter* (*Suc i*) = *Ccl-iter i* **and** *2*: *j* ≥ *i*
shows *Ccl-iter j* = *Ccl-iter i*
⟨*proof*⟩

lemma *Ccl-iter-su-PosOf*:
Ccl-iter i nd nd' ⊆
{*R* . ∀ *h h' sl*. (*h*,*h'*) ∉ *PosOf nd* × *PosOf nd'* ⟶ ¬ *R h h' sl*}

⟨*proof*⟩

lemma *finite-PosOf-prod*:
assumes *nd* ∈ *Node nd′* ∈ *Node*
shows *finite* {*R* . ∀ *h h′ sl*. (*h*::′*pos*,*h*′::′*pos*) ∉ *PosOf nd* × *PosOf nd′*
⟶ ¬ *R h h′* (*sl*::*slope*)}
(**is** *finite ?A*)
⟨*proof*⟩

lemma *finite-Ccl-iter*:
shows *finite* (*Ccl-iter i nd nd′*)
⟨*proof*⟩

lemma *Ccl-iter-Suc-stops*:
∃ *i*. *Ccl-iter* (*Suc i*) = *Ccl-iter i*
⟨*proof*⟩

lemma *Ccl-iter-stops*: ∃ *i*. ∀ *j* ≥ *i*. *Ccl-iter j* = *Ccl-iter i*
⟨*proof*⟩

definition *Ccl* :: ′*node* ⇒ ′*node* ⇒ (′*pos* ⇒ ′*pos* ⇒ *slope* ⇒ *bool*) *set* **where**
Ccl nd nd′ ≡ ⋃ *i*. *Ccl-iter i nd nd′*

lemma *Ccl-nodes*: *nd* ∉ *Node* ∨ *nd′* ∉ *Node* ⟹ *Ccl nd nd′* = {}
⟨*proof*⟩

lemma *Ccl-eq-some-Ccl-iter*: ∃ *i*. *Ccl* = *Ccl-iter i*
⟨*proof*⟩

lemma *Ccl-RR*[*simp*,*intro*!]:
{*nd*,*nd′*} ⊆ *Node* ⟹ *edge nd nd′* ⟹ (λ*P P′ sl*. *RR* (*nd*,*P*) (*nd′*,*P′*) *sl*) ∈ *Ccl nd nd′*
⟨*proof*⟩

lemma *Ccl-ccompSl*[*intro*]:
nd″ ∈ *Node* ⟹ *K1* ∈ *Ccl nd nd″* ⟹ *K2* ∈ *Ccl nd″ nd′*
⟹ *ccompSl K1 K2* ∈ *Ccl nd nd′*
⟨*proof*⟩

definition *TransitiveLoopingCcl* ≡ ∀ *nd*∈*Node*. ∀ *K*∈*Ccl nd nd*. (∃ *P*. *transSl K P P Decr*)

definition *compSl K1 K2 P P′ sl* ≡
(∃ *P″ sl1 sl2*. *sl* = *MaxSl* {*sl1*, *sl2*} ∧ *K1 P P″ sl1* ∧ *K2 P″ P′ sl2*) ∧
sl = *Max* {*sl*. ∃ *P″ sl1 sl2*. *sl* = *MaxSl* {*sl1*, *sl2*} ∧ *K1 P P″ sl1* ∧ *K2 P″ P′ sl2*}

lemma *compSl-SlopeRels*:
{*P*,*P′*} ⊆ *SlopedRels* ⟹ *compSl P P′* ∈ *SlopedRels*
⟨*proof*⟩

lemma *compSl-leqSl-ccompSl*: *leqSl* (*compSl K1 K2*) (*ccompSl K1 K2*)
⟨*proof*⟩

lemma *ccompSl-leqSl-compSl*: *leqSl* (*ccompSl K1 K2*) (*compSl K1 K2*)
⟨*proof*⟩

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* {λ*P 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*⟩

lemma *finite-Dcl-iter*: *finite* (*Dcl-iter i nd nd′*)
⟨*proof*⟩

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′*}
⟨*proof*⟩

lemma *Dcl-iter-SlopedRels*: *Dcl-iter i nd nd′* ⊆ *SlopedRels*
⟨*proof*⟩

lemma *Dcl-iter-subseqI*:
assumes ⋀*R h h′ sl*. *R* ∈ *Dcl-iter* (*Suc i*) *nd nd′* ⟹
(*h*, *h′*) ∉ *PosOf nd* × *PosOf nd′* ⟹
R h h′ sl ⟹
HOL.False

shows *Dcl-iter (Suc i) nd nd′ ⊆ {R. ∀ h h′ sl. (h, h′) ∉ PosOf nd × PosOf nd′ ⟶ ¬ R h h′ sl}*
⟨*proof*⟩

lemma *Dcl-iter-su-PosOf*:
Dcl-iter i nd nd′ ⊆
{R . ∀ h h′ sl. (h,h′) ∉ PosOf nd × PosOf nd′ ⟶ ¬ R h h′ sl}
⟨*proof*⟩

lemma *Dcl-iter-nodes-out*:
nd ∉ Node ∨ nd′ ∉ Node ⟹ Dcl-iter i nd nd′ = {}
⟨*proof*⟩

lemma *initFrag-Dt-trans*:
L ⊆ SlopedRels ⟹ finite L ⟹ initFrag L L′ ⟹ initFrag (Dt L) L′
⟨*proof*⟩

lemma *Dt-initFrag-aux*: *initFrag LL LL′ ⟹ initFrag LL (Dt LL′)*
⟨*proof*⟩

lemma *initFrag-Un-left*: *initFrag LL LL′ ⟹ initFrag (LL ∪ KK) LL′*
⟨*proof*⟩

lemma *Dcl-iter-initFrag-Ccl-iter*: *initFrag (Dcl-iter i nd nd′) (Ccl-iter i nd nd′)*
⟨*proof*⟩

lemma *Ccl-iter-initFrag-Dcl-iter*: *initFrag (Ccl-iter i nd nd′) (Dcl-iter i nd nd′)*
⟨*proof*⟩

lemma *Dcl-iter-finite-range*:
assumes *nd ∈ Node nd′ ∈ Node*
shows *finite (range (λi. Dcl-iter i nd nd′))*
⟨*proof*⟩

lemma *Dcl-iter-finite-range-all*:
finite (range (λi. Dcl-iter i nd nd′))
⟨*proof*⟩

lemma *Dt-idem*: *Dt (Dt X) = Dt X*
⟨*proof*⟩

lemma *Dcl-iter-thin-0*: *Dt* (*Dcl-iter 0 nd nd′*) = *Dcl-iter 0 nd nd′*
⟨*proof*⟩

lemma *Dcl-iter-thin*: *Dt* (*Dcl-iter i nd nd′*) = *Dcl-iter i nd nd′*
⟨*proof*⟩

lemma *Dcl-iter-initFrag-Suc*:
initFrag (*Dcl-iter* (*Suc i*) *nd nd′*) (*Dcl-iter i nd nd′*)
⟨*proof*⟩

lemma *Dcl-iter-initFrag-le*:
$i \leq j \Longrightarrow$ *initFrag* (*Dcl-iter j nd nd′*) (*Dcl-iter i nd nd′*)
⟨*proof*⟩

lemma *Dt-initFrag-antisym*:
assumes $A \subseteq$ *SlopedRels* $B \subseteq$ *SlopedRels*
and *Dt A* = *A Dt B* = *B*
and *AB*: *initFrag A B* **and** *BA*: *initFrag B A*
shows *A* = *B*
⟨*proof*⟩

lemma *Dcl-iter-Suc-stops-pair*: $\exists\, i.$ *Dcl-iter* (*Suc i*) *nd nd′* = *Dcl-iter i nd nd′*
⟨*proof*⟩

lemma *Dcl-iter-eq-implies-Suc-eq*:
assumes $i < j$ *Dcl-iter i nd nd′* = *Dcl-iter j nd nd′*
shows *Dcl-iter* (*Suc i*) *nd nd′* = *Dcl-iter i nd nd′*
⟨*proof*⟩

lemma *Dcl-iter-Suc-stops*: $\exists\, i.$ *Dcl-iter* (*Suc i*) = *Dcl-iter i*
⟨*proof*⟩

lemma *Dcl-iter-Suc-stable*:
assumes *Dcl-iter* (*Suc i*) = *Dcl-iter i*
shows *Dcl-iter* (*Suc* (*Suc i*)) = *Dcl-iter i*
⟨*proof*⟩

lemma *Dcl-iter-stable*:
assumes *Dcl-iter* (*Suc i*) = *Dcl-iter i* **and** $j \geq i$
shows *Dcl-iter j* = *Dcl-iter i*
⟨*proof*⟩

lemma *Dcl-iter-stops*: $\exists\, k.\ \forall\, j \geq k.$ *Dcl-iter j* = *Dcl-iter k*
⟨*proof*⟩

definition *Dcl* :: *'node* ⇒ *'node* ⇒ (*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) *set* **where**
Dcl nd nd′ ≡ *Dcl-iter* (*LEAST k.* $\forall\, j \geq k.$ *Dcl-iter j* = *Dcl-iter k*) *nd nd′*

lemma *Dcl-eq-some-Dcl-iter*: $\exists\, i.\ \forall\, j \geq i.$ *Dcl* = *Dcl-iter j*

⟨*proof*⟩

lemma *Dcl-initFrag-Ccl*: *initFrag* (*Dcl nd nd'*) (*Ccl nd nd'*)
⟨*proof*⟩

lemma *Ccl-initFrag-Dcl*: *initFrag* (*Ccl nd nd'*) (*Dcl nd nd'*)
⟨*proof*⟩

definition *TransitiveLooping* ≡ ∀ *nd*∈*Node*. ∀ *K*∈*Dcl nd nd*. (∃ *P*. *transSl K P P Decr*)

lemma *TransitiveLooping-iff-TransitiveLoopingCcl*:
TransitiveLooping ⟷ *TransitiveLoopingCcl*
⟨*proof*⟩

definition *descentPathSl* :: *'node list* ⇒ *'pos list* ⇒ *slope list* ⇒ *bool* **where**
descentPathSl ndl Pl sll ≡
length Pl = *length ndl* ∧ *length sll* = *length ndl*−*1* ∧
(∀ *i*<*length ndl*−*1*. *RR* (*ndl*!*i*,*Pl*!*i*) (*ndl*!(*Suc i*),*Pl*!(*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*⟩

lemma *descentPathSl-append-invert-singl*:
assumes *pathG ndl descentPathSl* (*ndl* @ [*nd*]) *Pl' sll'*
shows ∃ *Pl P sll sl*.
descentPathSl ndl Pl sll ∧ *Pl'* = *Pl* @ [*P*] ∧ *sll'* = *sll* @ [*sl*] ∧ *RR* (*last ndl*,*last Pl*) (*nd*,*P*) *sl*
⟨*proof*⟩

lemma *descentPathSl-append-invert*:

assumes *p1*: *pathG ndl1* **and** *pathG ndl2 last ndl1 = hd ndl2 descentPathSl* (*ndl1* @ (*tl ndl2*)) *Pl sll*
shows ∃ *Pl1 Pl2 sll1 sll2*. *last Pl1 = hd Pl2* ∧
descentPathSl ndl1 Pl1 sll1 ∧ *descentPathSl ndl2 Pl2 sll2* ∧ *Pl = Pl1* @ (*tl Pl2*)
∧ *sll = sll1* @ *sll2*
⟨*proof*⟩

lemma *descentPathSl-wfLabL*:
assumes *pathG ndl* **and** *descentPathSl ndl Pl sll*
shows *wfLabL ndl Pl*
⟨*proof*⟩

definition *tracksPers* :: (*'pos* ⇒ *'pos* ⇒ *slope* ⇒ *bool*) ⇒ *'node list* ⇒ *bool* **where**

tracksPers K ndl ≡
(∀ *Pl sll*. *descentPathSl ndl Pl sll* ⟶ *K* (*hd Pl*) (*last Pl*) (*MaxSl* (*set sll*))) ∧
(∀ *P P' sl*. *K P P' sl* ⟶ (∃ *Pl sll*. *descentPathSl ndl Pl sll* ∧ *hd Pl = P* ∧ *last Pl = P'* ∧ *sl = MaxSl* (*set sll*)))

proposition *tracksPers-leftTotal*:
assumes *ndl*: *pathG ndl*
shows ∃ *K* ∈ *Ccl* (*hd ndl*) (*last ndl*). *tracksPers K ndl*
⟨*proof*⟩

proposition *tracksPers-rightTotal*:
assumes *K* ∈ *Ccl nd nd'*
shows ∃ *ndl*. *pathG ndl* ∧ *hd ndl = nd* ∧ *last ndl = nd'* ∧ *tracksPers K ndl*
⟨*proof*⟩

definition *descentPathParam ndl Pl sl* ≡
(∀ *i*. *Suc i* < *length ndl* ⟶
RR (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *Main* ∨
¬ *RR* (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *Main* ∧
RR (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *Decr* ∧ *sl = Decr*)
∧
(∃ *i*. *Suc i* < *length ndl* ∧ *RR* (*ndl* ! *i*, *Pl* ! *i*) (*ndl* ! *Suc i*, *Pl* ! *Suc i*) *sl*)

lemma *descentPath-descentPathParam*:
descentPath ndl Pl ⟷ *descentPathParam ndl Pl Decr*
⟨*proof*⟩

lemma *descentPathSl-descentPathParam*:
assumes *pathG ndl* **and** *descentPathSl ndl Pl sll*
shows *descentPathParam ndl Pl* (*MaxSl* (*set sll*))
⟨*proof*⟩

lemma *descentPathParam-append*:
assumes *ndl*: *ndl1* ≠ [] *ndl2* ≠ []*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*⟩

lemma *tracksPers-transSl-impl-ex-descentPath-repeat*:
assumes *ndl*: *cycleG ndl* **and** *tr*: *tracksPers K ndl* **and** *P*: *transSl K P P′ sl*
shows ∃ *Pl n*. *n* ≠ *0* ∧ *wfLabL* (*repeat n* (*butlast ndl*) @ [*last ndl*]) *Pl* ∧
descentPathParam (*repeat n* (*butlast ndl*) @ [*last ndl*]) *Pl sl* ∧
hd Pl = *P* ∧ *last Pl* = *P′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *wfLabL-descentPathParam-imp-descentPathSl*:
assumes *w*: *wfLabL ndl Pl* **and** *d*: *descentPathParam ndl Pl sl*
shows ∃ *sll*. *descentPathSl ndl Pl sll* ∧ *sl* = *MaxSl* (*set sll*)
⟨*proof*⟩

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*⟩

lemma *tracksPers-ex-descentPathSlS-impl-loopsDecr*:
assumes *ndl*: *cycleG ndl* **and** *tr*: *tracksPers K ndl*
and *d*: *descentIpathS* (*srepeat* (*butlast ndl*)) *Ps*
shows ∃ *P*. *transSl K P P Decr*
⟨*proof*⟩

lemma *tracksPers-transSl-iff-ex-descentIpathS*:
assumes *cycleG ndl tracksPers K ndl*
shows (∃ *P*. *transSl K P P Decr*) ⟷ (∃ *Ps*. *descentIpathS* (*srepeat* (*butlast ndl*)) *Ps*)
⟨*proof*⟩

definition *allOmegaCyclesDescendS* ≡
∀ *ndl*. *cycleG ndl* ⟶ (∃ *Ps*. *descentIpathS* (*srepeat* (*butlast ndl*)) *Ps*)

proposition *TransitiveLoopingCcl-iff-allOmegaCyclesDescendS*:
TransitiveLoopingCcl ⟷ *allOmegaCyclesDescendS*
⟨*proof*⟩

lemma *InfiniteDescent-imp-InfiniteDescent*:
assumes *InfiniteDescent*
shows *allOmegaCyclesDescendS*
⟨*proof*⟩

proposition *allOmegaCyclesDescendS-implies-InfiniteDescent*:
allOmegaCyclesDescendS ⟶ *InfiniteDescent*
⟨*proof*⟩

corollary *Relation-Based-Criterion*:
InfiniteDescent ⟷ *TransitiveLoopingCcl*
⟨*proof*⟩

theorem *Relation-Based-Criterion′*:
InfiniteDescent ⟷ *TransitiveLooping*
⟨*proof*⟩

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 ≡
  $(\forall nd\ nd'.\ edge1\ nd\ nd' \longrightarrow RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Main \vee$
  $RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Decr) \wedge$
  $(\exists nd\ nd'.\ edge1\ nd\ nd' \wedge RR\ (nd,lab\ nd)\ (nd',lab\ nd')\ Decr)$

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *decreasingPCC-imp-descentIpath*:
assumes *nds*: *ipath nds*
and *lim*: *decreasingPCC* (*limitR nds*) *lab*
shows *descentIpath nds* (*smap lab nds*)
⟨*proof*⟩

definition *SDdescending* :: *bool* **where**
SDdescending ≡ ∀ *Node1 edge1*. *scsg Node1 edge1* ⟶ (∃ *lab*. *wfLabF Node1 lab* ∧ *decreasingPCC edge1 lab*)

proposition *SDdescending-imp-InfiniteDescent*:
SDdescending ⟹ *InfiniteDescent*
⟨*proof*⟩

4.2 Extended Sprenger-Dam Criterion

definition *ExtG-Nodes* :: (*'node* × *'pos*) *set* **where**
ExtG-Nodes ≡ {(*nd*, *P*). *nd* ∈ *Node* ∧ *P* ∈ *PosOf nd*}

definition *ExtG-Edges* :: (*'node* × *'pos*) ⇒ (*'node* × *'pos*) ⇒ *bool* **where**
ExtG-Edges ≡ λ(*nd*, *P*) (*nd'*, *P'*).
  edge nd nd' ∧ (*RR* (*nd*,*P*) (*nd'*,*P'*) *Main* ∨ *RR* (*nd*,*P*) (*nd'*,*P'*) *Decr*)

definition *is-slice* ::
'node set ⇒ (*'node* ⇒ *'node* ⇒ *bool*) ⇒
 (*'node* ⇒ *'pos set*) ⇒ (*'node* ⇒ *'node* ⇒ *'pos* ⇒ *'pos*) ⇒
 (*'node* × *'pos*) *set* ⇒ ((*'node* × *'pos*) ⇒ (*'node* × *'pos*) ⇒ *bool*) ⇒ *bool* **where**
is-slice Node1 edge1 lab f NNode eedge ≡
  NNode ⊆ {(*nd*, *P*). *nd* ∈ *Node1* ∧ *P* ∈ *lab nd*} ∧
  Graph.subgr NNode eedge ExtG-Nodes ExtG-Edges ∧
  (∀ *nd P nd' P'*. *eedge* (*nd*, *P*) (*nd'*, *P'*) ⟶
    {(*nd*, *P*), (*nd'*, *P'*)} ⊆ *NNode* ∧ *edge1 nd nd'* ∧ *f nd nd' P* = *P'*) ∧
  (∀ *nd nd'*. {*nd*,*nd'*} ⊆ *Node1* ∧ *edge1 nd nd'* ⟶
    (∃ *P P'*. {(*nd*, *P*), (*nd'*, *P'*)} ⊆ *NNode* ∧ *eedge* (*nd*, *P*) (*nd'*, *P'*)))

definition *decreasing-slice* :: (*'node* × *'pos*) *set* ⇒ ((*'node* × *'pos*) ⇒ (*'node* × *'pos*) ⇒ *bool*) ⇒ *bool* **where**
decreasing-slice NNode eedge ≡

$\exists\, nd\ P\ nd'\ P'.\ \{(nd,\ P),\ (nd',\ P')\} \subseteq NNode \wedge eedge\ (nd,\ P)\ (nd',\ P') \wedge RR\ (nd,\ P)\ (nd',\ P')\ Decr$

definition *descending-PCSC-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-PCSC-sliced Node1 edge1 lab f $\equiv$
RRSetChoice Node1 edge1 lab f $\wedge$
$(\forall$ *NNode eedge.*
is-slice Node1 edge1 lab f NNode eedge $\wedge$
Graph.scg NNode eedge
$\longrightarrow$ *decreasing-slice NNode eedge*)

definition *XSDdescending* :: *bool* **where**
XSDdescending $\equiv$
$\forall$ *Node1 edge1. scsg Node1 edge1* $\longrightarrow$
$(\exists$ *lab f. wfLabFS Node1 lab* $\wedge$ *descending-PCSC-sliced Node1 edge1 lab f*$)$

lemma *stake-sdrop-szip-nth*:
$k < m \Longrightarrow$ *stake m (sdrop j (szip A B)) ! k* $=$ *(A !! (j + k), B !! (j + k))*
⟨*proof*⟩

lemma *set-stake-sdrop-szipD*:
(x, y) $\in$ *set (stake m (sdrop j (szip A B)))* $\Longrightarrow$
$\exists\, k{<}m.$ *x* $=$ *A !! (j + k)* $\wedge$ *y* $=$ *B !! (j + k)*
⟨*proof*⟩

lemma *eq-stake-sdrop-szip-tuple*:
k $<$ *m* $\Longrightarrow$ *(x, y)* $=$ *stake m (sdrop j (szip A B)) ! k* $\Longrightarrow$ *x* $=$ *A !! (j + k)* $\wedge$ *y* $=$ *B !! (j + k)*
⟨*proof*⟩

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. descentIpath nds Ps*
⟨*proof*⟩

proposition *XSDdescending-implies-InfiniteDescent*:
XSDdescending $\Longrightarrow$ *InfiniteDescent*
⟨*proof*⟩

lemmas *Incomplete-Criterion* $=$ *SDdescending-imp-InfiniteDescent*

theorem *SDdescending-implies-XSDdescending*:
SDdescending $\Longrightarrow$ *XSDdescending*
⟨*proof*⟩

end

end

4.3 Sprenger-Dam Criterion Incompleteness

theory *SD-Incomplete*
imports *../Incomplete-Criteria*
begin

datatype *node* = *One*
datatype *pos*= *h1* | *h1′*

definition *Node* ≡ {*One*}

lemma *O-Node*[*simp*]:*One* ∈ *Node* ⟨*proof*⟩

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

fun *edge*::*node* ⇒ *node* ⇒ *bool* **where**
edge One One = *HOL.True*

lemma *edge-into-zero*:*edge nd nd′* ⟷ *nd* = *One* ∧ *nd′* = *One* ⟨*proof*⟩

lemma *edgeTrue*:*edge nd nd′* ⟨*proof*⟩

fun *PosOf*::*node* ⇒ *pos set* **where**
PosOf One = {*h1*, *h1′*}

definition *RR-set* :: ((*node* × *pos*) × (*node* × *pos*) × *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**
$RR\ np1\ np2\ s \equiv ((np1,\ np2,\ s) \in RR\text{-}set)$

lemmas $RR\text{-}defs = RR\text{-}def\ RR\text{-}set\text{-}def$

lemma $RR\text{-}ZO[simp]{:}RR\ (One,\ h1)\ (One,\ h1')\ Decr\ \langle proof\rangle$
lemma $RR\text{-}OZ[simp]{:}RR\ (One,\ h1')\ (One,\ h1)\ Main\ \langle proof\rangle$

lemma $P\text{-}inPosOf{:}RR\ (nd,\ P)\ (nd',\ P')\ sl \Longrightarrow P \in PosOf\ nd$
$RR\ (nd,\ P)\ (nd',\ P')\ sl \Longrightarrow P' \in PosOf\ nd'\ \langle proof\rangle$

interpretation *Sloped-Graph* **where**
$Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$
and $RR = RR\ \langle proof\rangle$

lemma $allNodesOne{:}\forall\, i.\ nds\ !!\ i = One\ \langle proof\rangle$

lemma $ipath\text{-}isOne{:}ipath\ nds \Longrightarrow nds = sconst\ One$
$\langle proof\rangle$

lemma $j\text{-}mod2\text{-}cases{:}((j\ mod\ Suc\ (Suc\ 0)) = 0 \wedge (Suc\ j\ mod\ Suc\ (Suc\ 0)) = 1)$
$\vee\ ((j\ mod\ Suc\ (Suc\ 0)) = 1 \wedge (Suc\ j\ mod\ Suc\ (Suc\ 0)) = 0)$
$\langle proof\rangle$

lemma $j\text{-}mod2\text{-}cases'{:}(j\ mod\ Suc\ (Suc\ 0)) = 0 \vee (j\ mod\ Suc\ (Suc\ 0)) = 1$
$\langle proof\rangle$

lemma $j\text{-}mod2\text{-}suc{:}j\ mod\ Suc\ (Suc\ 0) = 0 \Longrightarrow Suc\ j\ mod\ Suc\ (Suc\ 0) = 1\ \langle proof\rangle$

lemma $descentPathEx{:}descentIpath\ (sconst\ One)\ (srepeat\ [h1,\ h1'])$
$\langle proof\rangle$

lemma $pathConEx{:}Graph.pathCon\ \{One\}\ (\lambda u\ v.\ True)\ One\ One$
$\langle proof\rangle$

lemma $scsgEx{:}scsg\ \{One\}\ (\lambda u\ v.\ True)$
$\langle proof\rangle$

proposition *InfiniteDescent*
$\langle proof\rangle$

proposition $\neg SDdescending$

⟨*proof*⟩

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* ≡ {*One*, *Two*, *Three*}

lemma *nd-notOne*:*nd* ≠ *One* ⟷ *nd* = *Two* ∨ *nd* = *Three* ⟨*proof*⟩

lemma *O-Node*[*simp*]:*One* ∈ *Node* ⟨*proof*⟩
lemma *T-Node*[*simp*]:*Two* ∈ *Node* ⟨*proof*⟩
lemma *Tr-Node*[*simp*]:*Three* ∈ *Node* ⟨*proof*⟩

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

fun *edge*::*node* ⇒ *node* ⇒ *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* ⇒ *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-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-set*)

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *RR-OT*[*simp*]:*RR* (*One*, *p1*) (*Two*, *p2*) *Main* ⟨*proof*⟩
lemma *RR-TO*[*simp*]:*RR* (*Two*, *p2*) (*One*, *p1*) *Decr* ⟨*proof*⟩
lemma *RR-TO'*[*simp*]:*RR* (*Two*, *p2*) (*One*, *p1'*) *Decr* ⟨*proof*⟩
lemma *RR-OTr*[*simp*]:*RR* (*One*, *p1'*) (*Three*, *p3*) *Main* ⟨*proof*⟩
lemma *RR-TrO*[*simp*]:*RR* (*Three*, *p3*) (*One*, *p1'*) *Decr* ⟨*proof*⟩
lemma *RR-TrO'*[*simp*]:*RR* (*Three*, *p3*) (*One*, *p1*) *Decr* ⟨*proof*⟩

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'* ⟨*proof*⟩

interpretation *Sloped-Graph* **where**
Node = *Node* **and** *edge* = *edge* **and** *PosOf* = *PosOf*
and *RR* = *RR* ⟨*proof*⟩

definition *ipath12* :: *node stream* $\Rightarrow$ *bool* **where**

ipath12 s ≡ *ev* (*alw* (*holds* (λ*n*. *n* = *One* ∨ *n* = *Two*))) *s*

definition *ipath13* :: *node stream* ⇒ *bool* **where**
ipath13 s ≡ *ev* (*alw* (*holds* (λ*n*. *n* = *One* ∨ *n* = *Three*))) *s*

definition *ipath-mixed* :: *node stream* ⇒ *bool* **where**
ipath-mixed s ≡ *alw* (*ev* (*holds* (λ*n*. *n* = *Two*))) *s* ∧ *alw* (*ev* (*holds* (λ*n*. *n* = *Three*))) *s*

lemmas *ipaths* = *ipath12-def ipath13-def ipath-mixed-def*

lemma *ipath-cases*:*ipath nds* ⟹ *ipath12 nds* ∨ *ipath13 nds* ∨ *ipath-mixed nds*
⟨*proof*⟩

lemma *ipath-dist21*:
assumes *ipath nds*
shows *nds* !! *i* = *Two* ⟹ *nds* !! *Suc i* = *One*
⟨*proof*⟩

lemma *ipath-dist31*:
assumes *ipath nds*
shows *nds* !! *i* = *Three* ⟹ *nds* !! *Suc i* = *One*
⟨*proof*⟩

lemma *ipath-dist1*:
assumes *ipath nds*
shows *nds* !! *i* = *One* ⟹ *nds* !! *Suc i* = *Two* ∨ *nds* !! *Suc i* = *Three*
⟨*proof*⟩

lemma *ipath12-descent*:
assumes *ipath nds*
shows *ipath12 nds* ⟹ ∃ *Ps*. *descentIpath nds Ps*
⟨*proof*⟩

lemma *ipath13-descent*:
assumes *ipath nds*
shows *ipath13 nds* ⟹ ∃ *Ps*. *descentIpath nds Ps*
⟨*proof*⟩

lemma *ipath-mixed-descent*:
assumes *ipath nds*
shows *ipath-mixed nds* ⟹ ∃ *Ps*. *descentIpath nds Ps*
⟨*proof*⟩

lemma *pathConEx11*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *One One*
$\langle proof \rangle$

lemma *pathConEx12*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *One Two*
$\langle proof \rangle$

lemma *pathConEx13*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *One Three*
$\langle proof \rangle$

lemma *pathConEx21*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *Two One*
$\langle proof \rangle$

lemma *pathConEx31*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *Three One*
$\langle proof \rangle$

lemma *pathConEx23*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *Two Three*
$\langle proof \rangle$

lemma *pathConEx32*:*Graph.pathCon* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$ *Three Two*
$\langle proof \rangle$

lemmas *pathConEx* = *pathConEx11 pathConEx12*
pathConEx13 pathConEx21
pathConEx31 pathConEx23
pathConEx32

lemma *scsgEx*:*scsg* $\{One, Two, Three\}$ $(\lambda u\ v.\ edge\ u\ v)$
$\langle proof \rangle$

lemma *wfLabFSCases*:
wfLabFS $\{One,\ Two,\ Three\}$ *lab* $\Longrightarrow$
$(lab\ One = \{p1\} \vee lab\ One = \{p1'\} \vee lab\ One = \{p1, p1'\}) \wedge$
$lab\ Three = \{p3\} \wedge lab\ Two = \{p2\}$ $\langle proof \rangle$

lemma *noHCSC*:*wfLabFS* $\{One,\ Two,\ Three\}$ *lab* $\Longrightarrow$
RRSetChoice $\{One,\ Two,\ Three\}$ *edge lab f* $\Longrightarrow$ *False*
$\langle proof \rangle$

proposition *InfiniteDescent*
$\langle proof \rangle$

proposition $\neg$*XSDdescending*

⟨*proof*⟩

end

4.5 Flat Cycles Criterion

theory *Flat-Cycles-Criterion*
imports *../Sloped-Graphs*
begin

context *Sloped-Graph*
begin

definition *FlatEdges*::(*'node* × *'node*) *set* **where**
FlatEdges ≡ {(*u*, *v*). *edge u v* ∧ (∀ *p* ∈ *PosOf u*. ∀ *q* ∈ *PosOf v*. ¬*RR* (*u*,*p*) (*v*,*q*) *Decr*)}

lemma *FlatEdges-no-Decr*:(*u*,*v*) ∈ *FlatEdges* ⟹ ¬ *RR* (*u*,*p*) (*v*,*q*) *Decr*
⟨*proof*⟩

definition *FlatCycle* :: *bool* **where**
FlatCycle ≡ ∃ *nds*. (∀ *i* < *length nds* − *1*. (*nds* ! *i*, *nds* ! (*Suc i*)) ∈ *FlatEdges*) ∧ *cycleG nds*

lemma *FlatCycle-properties*:*FlatCycle* ⟹ ∃ *xs*. *cycleG xs* ∧ (∀ *i* < *length xs* − *1*. ∀ *p q*. ¬ *RR* (*xs*!*i*,*p*) (*xs*!(*i*+*1*),*q*) *Decr*)
⟨*proof*⟩

lemma *FlatCycle-imp-flat-ipath*:
assumes *FlatCycle*
shows ∃ *S*. *ipath S* ∧ (∀ *i p q*. ¬ *RR* (*S* !! *i*, *p*) (*S* !! (*Suc i*), *q*) *Decr*)
⟨*proof*⟩

lemma *FlatCycleE*:*FlatCycle* ⟹ (⋀*xs*. *ipath xs* ⟹ (∀ *i p q*. ¬ *RR* (*xs*!! *i*,*p*) (*xs*!!(*i*+*1*),*q*) *Decr*) ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *notFlatCycleI*:
(⋀*nds*. ∀ *i*<*length nds* − *1*. (*nds* ! *i*, *nds* ! *Suc i*) ∈ *FlatEdges* ⟹

$cycleG\ nds \Longrightarrow HOL.False) \Longrightarrow \neg FlatCycle$
⟨*proof*⟩

lemma *notFlatCycleI'*:
$(\bigwedge nds.\ pathG\ nds \Longrightarrow$
$\forall\, i{<}length\ nds - 1.\ (nds\ !\ i, nds\ !\ Suc\ i) \in FlatEdges \Longrightarrow$
$hd\ nds = last\ nds \Longrightarrow HOL.False) \Longrightarrow \neg FlatCycle$
⟨*proof*⟩

theorem *Flat-Cycles-Criterion:FlatCycle* $\Longrightarrow \neg InfiniteDescent$
⟨*proof*⟩

end

end

4.6 Descending Unicycles Criterion

theory *Descending-Unicycles-Criterion*
imports *../Sloped-Graphs*
../Buchi-Preliminaries
begin

context *Graph*
begin

definition *basicCycle* :: *'node list* ⇒ *bool* **where**
basicCycle c ≡ *cycleG c* ∧ *distinct* (*butlast c*)

lemmas *basicCycle-defs* = *basicCycle-def*[*unfolded cycleG-def*]

lemma *basicCycleI:pathG c* ⟹ *hd c* = *last c* ⟹ *distinct* (*butlast c*) ⟹ *basicCycle c*
⟨*proof*⟩

lemma *basicCycle-path-drop*: *basicCycle c* ⟹ *j* < *length* (*butlast c*) ⟹ *pathG* (*drop j c*)
⟨*proof*⟩

lemma *basicCycle-not-nil:basicCycle c* ⟹ *c* ≠[] ⟨*proof*⟩
lemma *basicCycle-ge2:basicCycle c* ⟹ *length c* ≥ *2* ⟨*proof*⟩

lemma *cycle-elim:*¬(∃ *c. basicCycle c* ∧ *set c* = *V'*) ⟹ (∀ *c. basicCycle c* ⟶ *set c* ≠ *V'* ⟹ *P*) ⟹ *P* ⟨*proof*⟩

lemma *basicCycle-set-eq*:*basicCycle c* $\Longrightarrow$ *set c* = *set* (*butlast c*)
⟨*proof*⟩

lemma *cycleG-contains-basicCycle*:
assumes *cycleG ndl*
shows ∃ *c*. *basicCycle c* ∧ *set c* ⊆ *set ndl*
⟨*proof*⟩

lemma *basicCycle-rotate1*:
basicCycle (*ndl* @ [*nd*,*nd'*]) $\Longrightarrow$ *basicCycle* (*nd* # *ndl* @ [*nd*])
⟨*proof*⟩

lemma *basicCycle-rotate*:
assumes *basicCycle* (*ndl1* @ *ndl2* @ [*nd'*]) *ndl2* ≠ []
shows *basicCycle* (*ndl2* @ *ndl1* @ [*hd ndl2*])
⟨*proof*⟩

lemma *basicCycle-rotate-butlast*:
assumes *basicCycle* (*ndl1* @ *nd* # *ndl2*) *ndl1* ≠ [] *ndl2* ≠ []
shows *basicCycle* (*nd* # *butlast ndl2* @ *ndl1* @ [*nd*])
⟨*proof*⟩

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*⟩

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*⟩

lemma *scsg-has-basicCycle*: *V'* ≠ {} $\Longrightarrow$ *scsg V' E'* $\Longrightarrow$ ∃ *c*. *basicCycle c* ∧ *set c* ⊆ *V'*
⟨*proof*⟩

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*⟩

definition *connectedCycles* :: *'node list* ⇒ *'node list* ⇒ *bool* **where**
connectedCycles c1 c2 ≡ (∃ *p*. *pathG p* ∧ *hd p* ∈ *set c1* ∧ *last p* ∈ *set c2*)

lemma *connectedCyclesE*:*connectedCycles c1 c2* ⟹ (⋀*p*. *pathG p* ⟹
hd p ∈ *set c1* ⟹ *last p* ∈ *set c2* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

definition *unicyclesGraph* :: *bool* **where**
unicyclesGraph ≡
∀ *c c'*. *basicCycle c* ∧ *basicCycle c'* ⟶
(*connectedCycles c c'* ∧ *connectedCycles c' c*) ⟶ *set c* = *set c'*

lemma *unicyclesGraphI*:(⋀*c c'*.
connectedCycles c c' ⟹
basicCycle c ⟹
basicCycle c' ⟹
connectedCycles c' c ⟹ *set c* = *set c'*) ⟹ *unicyclesGraph*
⟨*proof*⟩

lemma *unicyclesGraphI'*:(⋀*c c' p*.
pathG p ⟹
hd p ∈ *set c* ⟹
last p ∈ *set c'* ⟹
basicCycle c ⟹
basicCycle c' ⟹
connectedCycles c' c ⟹ *set c* = *set c'*) ⟹ *unicyclesGraph*
⟨*proof*⟩

lemma *basicCycle-unique-successor*:
assumes *unicyclesGraph*
assumes *basicCycle c*
assumes *i* < *length c* − *1*
assumes *edge* (*c* ! *i*) *v v* ∈ *set* (*butlast c*)
shows *v* = *c* ! (*Suc i*)
⟨*proof*⟩

lemma *scsg-in-unicycles-is-basicCycle*:
assumes *unicyclesGraph*
$V' \neq \{\}$
scsg V' E'
shows $\exists\, c.$ *basicCycle* $c \wedge$ *set* $c =$ V'
⟨*proof*⟩

lemma *unicycle-limit-is-basic-cycle*:
assumes *unicyclesGraph*
assumes *finite Node*
assumes *ipath* π
shows $\exists\, c.$ *basicCycle* $c \wedge$ *limitS* $\pi =$ *set* (*butlast* c)
⟨*proof*⟩

lemma *limitS-limitR-edges*:
assumes *unicyclesGraph*
assumes *finite Node*
assumes *ipath* π
assumes *limitS* $\pi =$ *set* (*butlast* c)
assumes *basicCycle* c
shows $\forall\, i <$ *length* $c - 1.$ (*limitR* π) (c ! i) (c ! *Suc* i)
⟨*proof*⟩

lemma *unicycle-lasso*:
assumes *unicyclesGraph*
assumes *finite Node*
assumes *ipath* π
shows $\exists\, v\ u.\ \pi = v$ @− *srepeat* $u \wedge$ *basicCycle* (u @ [*hd* u])
⟨*proof*⟩

end

context *Sloped-Graph*
begin

definition *cycleDescends ndl* ≡ ($\exists\, n$ *Pl*. $n \neq 0 \wedge$ *wfLabL* (*repeat* n (*butlast ndl*) @ [*last ndl*]) *Pl* ∧
descentPath (*repeat* n (*butlast ndl*) @ [*last ndl*]) *Pl* ∧ *hd Pl* = *last Pl*)

lemma *cycle-descentIPathS-imp-cycleDescends*:
assumes *ndl*: *cycleG ndl* **and** *d*: *descentIpathS* (*srepeat* (*butlast ndl*)) *Ps*

shows *cycleDescends ndl*
⟨*proof*⟩

lemma *cycle-descentIPath-imp-cycleDescends*:
assumes *ndl*: *cycleG ndl* **and** *d*: *descentIpath* (*srepeat* (*butlast ndl*)) *Ps*
shows *cycleDescends ndl*
⟨*proof*⟩

lemma *cycleDescends-imp-descentIPathS*:
assumes *ndl*: *cycleG ndl* **and** *desc*: *cycleDescends ndl*
shows ∃ *Ps*. *descentIpathS* (*srepeat* (*butlast ndl*)) *Ps*
⟨*proof*⟩

definition *SimplyDescendingGraph* :: *bool* **where** *SimplyDescendingGraph* ≡ ∀ *ndl*. *basicCycle ndl* ⟶ *cycleDescends ndl*

lemma *SimplyDescendingGraphD*: *SimplyDescendingGraph* ⟹ *basicCycle c* ⟹ *cycleDescends c*
⟨*proof*⟩

lemma *SimplyDescendingGraphI*:(⋀*c*. *ipath* (*srepeat* (*butlast c*)) ⟹ *basicCycle c* ⟹ *cycleDescends c*) ⟹ *SimplyDescendingGraph*
⟨*proof*⟩

lemma *SimplyDescendingGraphI'*:(⋀*c*. *basicCycle c* ⟹ *cycleDescends c*) ⟹ *SimplyDescendingGraph*
⟨*proof*⟩

lemma *allCyclesDescendI*:(⋀*c*. *ipath* (*srepeat* (*butlast c*)) ⟹ *basicCycle c* ⟹ (∃ *n Pl*.
n ≠ *0* ∧
wfLabL
(*repeat n* (*butlast c*) @ [*last c*])
Pl ∧
descentPath
(*repeat n* (*butlast c*) @ [*last c*])
Pl ∧
hd Pl = *last Pl*)) ⟹ *SimplyDescendingGraph*
⟨*proof*⟩

proposition *InfiniteDescent-implies-SimplyDescendingGraph*:
InfiniteDescent ⟹ *SimplyDescendingGraph*
⟨*proof*⟩

proposition *SimplyDescendingUnicyclesGraph-implies-InfiniteDescent*:

assumes *unicyclesGraph*
shows *SimplyDescendingGraph* $\Longrightarrow$ *InfiniteDescent*
⟨*proof*⟩

theorem *DescendingUnicyclesCriterion*:
assumes *unicyclesGraph*
shows *InfiniteDescent* $\longleftrightarrow$ *SimplyDescendingGraph*
⟨*proof*⟩

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* ≡ {*Flat*, *Aux*}

lemma *nd-aux*[*simp*]:*nd* ≠ *Aux* ⟷ *nd* = *Flat* ⟨*proof*⟩
lemma *nd-flat*[*simp*]:*nd* ≠ *Flat* ⟷ *nd* = *Aux* ⟨*proof*⟩

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

fun *edge*::*node* ⇒ *node* ⇒ *bool* **where**
edge Flat Aux = *HOL.True*|
edge Aux - = *HOL.True*|
edge Flat Flat = *HOL.False*

lemma *edge-Flat*[*simp*]:*edge Flat x* ⟷ *x* = *Aux* ⟨*proof*⟩

lemma *edge-Flat'*[*simp*]:*edge nd Flat* ⟷ *nd* = *Aux* ⟨*proof*⟩

fun *PosOf*::*node* ⇒ *pos set* **where**
PosOf Flat = {*0*}|
PosOf Aux = {*0*,*1*}

definition *RR-set* :: ((*node* × *pos*) × (*node* × *pos*) × *slope*) *set* **where**
RR-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 $RR :: node \times pos \Rightarrow node \times pos \Rightarrow slope \Rightarrow bool$ **where**
$RR\ np1\ np2\ s \equiv ((np1,\ np2,\ s) \in RR\text{-}set)$

definition $RRs :: node \Rightarrow node \Rightarrow (pos \Rightarrow pos \Rightarrow slope \Rightarrow bool)$ **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 $RR\text{-}aux\text{-}aux\text{-}1[simp]{:}RR\ (Aux,\ 0)\ (Aux,\ 0)\ Decr\ \langle proof\rangle$

lemma $P\text{-}inPosOf{:}RR\ (nd,\ P)\ (nd',\ P')\ sl \Longrightarrow P \in PosOf\ nd$
$RR\ (nd,\ P)\ (nd',\ P')\ sl \Longrightarrow P' \in PosOf\ nd'\ \langle proof\rangle$

interpretation *Sloped-Graph* **where**
$Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$ **and** $RR = RR$
$\langle proof\rangle$

lemma $lang\text{-}run\text{-}inj{:}x \in NBA.language\ Paut_V \Longrightarrow \exists r.\ NBA.run\ Paut_V\ (x\ |||\ r)$
None
$\langle proof\rangle$

end

5.1.2 Flat Aux via Vertex-Language Automaton

theory *Flat-Aux-VLA*
imports *Flat-Aux*
begin

lemma $InfiniteDescentI{:}(\bigwedge nds.\ alw\ (holds2\ edge)\ nds \Longrightarrow Ex\ (descentIpath\ nds))$
$\Longrightarrow InfiniteDescent$
$\langle proof\rangle$

lemma *InfiniteDescent*

$\langle proof\rangle$

end

5.1.3 Flat Aux via Sloped-Language Automaton

theory *Flat-Aux-SLA*
imports *Flat-Aux*
begin

lemma *InfiniteDescentI*:$(\bigwedge nds.\ alw\ (holds2\ edge)\ nds \Longrightarrow Ex\ (descentIpath\ nds))$
$\Longrightarrow$ *InfiniteDescent*
⟨*proof*⟩

lemma *InfiniteDescent*

⟨*proof*⟩

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* ≡ {*Zero*, *One*, *Two*, *Three*}

lemma *Z-Node*[*simp*]:*Zero* ∈ *Node* ⟨*proof*⟩
lemma *O-Node*[*simp*]:*One* ∈ *Node* ⟨*proof*⟩
lemma *T-Node*[*simp*]:*Two* ∈ *Node* ⟨*proof*⟩
lemma *Th-Node*[*simp*]:*Three* ∈ *Node* ⟨*proof*⟩

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

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* ⟨*proof*⟩
lemma *edge-into-one*[*simp*]:*edge nd One* $\longleftrightarrow$ *nd* = *Zero* ⟨*proof*⟩
lemma *edge-into-three*[*simp*]:*edge nd Three* $\longleftrightarrow$ *nd* = *Two* ⟨*proof*⟩
lemma *edge-three-into*[*simp*]:*edge Three nd* $\longleftrightarrow$ *nd* = *Two* ⟨*proof*⟩
lemma *edge-two-into*[*simp*]:*edge Two nd* $\longleftrightarrow$ *nd* = *Three* ⟨*proof*⟩
lemma *edge-zero-into*[*simp*]:*edge Zero nd* $\longleftrightarrow$ *nd* = *One* ∨ *nd* = *Two* ⟨*proof*⟩

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* × *pos* ⇒ *node* × *pos* ⇒ *slope* ⇒ *bool* **where**
RR np1 np2 s ≡ ((*np1*, *np2*, *s*) ∈ *RR-set*)

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *RR-ZO*[*simp*]:*RR* (*Zero*, *p0*) (*node.One*, *p1*) *Decr* ⟨*proof*⟩
lemma *RR-OZ*[*simp*]:*RR* (*node.One*, *p1*) (*Zero*, *p0*) *Main* ⟨*proof*⟩

lemma *RR-TTr*[*simp*]:*RR* (*Two*, *p2*) (*Three*, *p3*) *Main* ⟨*proof*⟩
lemma *RR-TrT*[*simp*]:*RR* (*Three*, *p3*) (*Two*, *p2*) *Decr* ⟨*proof*⟩

lemma *P-inPosOf*:*RR* (*nd*, *P*) (*nd'*, *P'*) *sl* ⟹ *P* ∈ *PosOf nd*

$$RR\ (nd,\ P)\ (nd',\ P')\ sl \Longrightarrow P' \in PosOf\ nd'\ \langle proof \rangle$$

interpretation *Sloped-Graph* **where**
  $Node = Node$ **and** $edge = edge$ **and** $PosOf = PosOf$
  and $RR = RR\ \langle proof \rangle$

lemma *notTZ*:$\neg pathG\ [node.Two,\ Zero]\ \langle proof \rangle$
lemma *notTO*:$\neg pathG\ [node.Two,\ One]\ \langle proof \rangle$
lemma *notTrZ*:$\neg pathG\ [node.Three,\ Zero]\ \langle proof \rangle$
lemma *notTrO*:$\neg pathG\ [node.Three,\ One]\ \langle proof \rangle$
lemma *notOT*:$\neg pathG\ [node.One,\ node.Two]\ \langle proof \rangle$
lemma *notOTr*:$\neg pathG\ [node.One,\ node.Three]\ \langle proof \rangle$
lemma *notZTr*:$\neg pathG\ [Zero,\ node.Three]\ \langle proof \rangle$
lemma *notZTZ*:$\neg pathG\ ([Zero,node.Two,\ Zero])\langle proof \rangle$

lemma *pathNWF-Three*:$pathG\ p \Longrightarrow hd\ p = Three \Longrightarrow last\ p \neq Zero \wedge last\ p \neq One$
  $\langle proof \rangle$

lemma *pathNWF-Two*:$pathG\ p \Longrightarrow hd\ p = Two \Longrightarrow last\ p \neq Zero \wedge last\ p \neq One$
  $\langle proof \rangle$

lemma *pathNWF-disj*:$pathG\ pa \Longrightarrow$
    $hd\ pa = node.Two \vee hd\ pa = node.Three \Longrightarrow$
    $last\ pa = Zero \vee last\ pa = node.One \Longrightarrow HOL.False$
  $\langle proof \rangle$

lemma *ZZ-FlatEdge*:$\neg(Zero,\ Zero) \in FlatEdges\ \langle proof \rangle$
lemma *OO-FlatEdge*:$\neg(One,\ One) \in FlatEdges\ \langle proof \rangle$
lemma *TT-FlatEdge*:$\neg(Two,\ Two) \in FlatEdges\ \langle proof \rangle$
lemma *ThTh-FlatEdge*:$\neg(Three,\ Three) \in FlatEdges\ \langle proof \rangle$
lemmas *same-FlatEdge* = *ZZ-FlatEdge OO-FlatEdge TT-FlatEdge ThTh-FlatEdge*

lemma *ZO-FlatEdge*:$\neg(Zero,\ One) \in FlatEdges\ \langle proof \rangle$
lemma *ThT-FlatEdge*:$\neg(Three,\ Two) \in FlatEdges\ \langle proof \rangle$

lemma *aux-ls*:$xs \neq [] \Longrightarrow (xs\ @\ [Zero])\ !\ Suc\ 0 = node.Three \Longrightarrow length\ xs > 1$
$\langle proof \rangle$

theorem $\neg FlatCycle$
  $\langle proof \rangle$

lemma *i-disj*:$i < Suc\ (Suc\ (Suc\ 0)) \longleftrightarrow i = 0 \vee i = 1 \vee i = 2\ \langle proof \rangle$

lemma *i-disj'*:$i < Suc\ (Suc\ 0) \longleftrightarrow i = 0 \vee i = 1\ \langle proof \rangle$

lemma *allCycles*:*basicCycle* $c \longleftrightarrow c = [Zero, One, Zero] \vee c = [One, Zero, One]$
$\vee\ c = [Two, Three, Two] \vee c = [Three, Two, Three]$
$\langle proof \rangle$

lemma *notConnectedCyclesI*:$(\bigwedge p.\ pathG\ p \Longrightarrow hd\ p \in set\ c' \Longrightarrow last\ p \in set\ c$
$\Longrightarrow HOL.False) \Longrightarrow \neg\ connectedCycles\ c'\ c$
$\langle proof \rangle$

lemma *unicycle*:*unicyclesGraph*
$\langle proof \rangle$

lemma *SDG*:*SimplyDescendingGraph*
$\langle proof \rangle$

theorem *InfiniteDescent*
$\langle proof \rangle$

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*]:$Zero \in Node\ \langle proof \rangle$
lemma *O-Node*[*simp*]:$One \in Node\ \langle proof \rangle$
lemma *T-Node*[*simp*]:$Two \in Node\ \langle proof \rangle$
lemma *Th-Node*[*simp*]:$Three \in Node\ \langle proof \rangle$

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

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*:*edge nd Zero* ⟷ *nd* = *One* ⟨*proof*⟩

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*), *Main*),

((*Three*, *p3'*), (*Two*, *p2'*), *Main*)
}

definition *RR* :: *node* × *pos* ⇒ *node* × *pos* ⇒ *slope* ⇒ *bool* **where**
RR np1 np2 s ≡ ((*np1*, *np2*, *s*) ∈ *RR-set*)

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *RR-ZO*[*simp*]:*RR* (*Zero*, *p0*) (*node.One*, *p1*) *Decr* ⟨*proof*⟩
lemma *RR-OZ*[*simp*]:*RR* (*node.One*, *p1*) (*Zero*, *p0*) *Main* ⟨*proof*⟩

lemma *RR-TTr*[*simp*]:*RR* (*Two*, *p2*) (*Three*, *p3*) *Main* ⟨*proof*⟩
lemma *RR-TrT*[*simp*]:*RR* (*Three*, *p3*) (*Two*, *p2*) *Main* ⟨*proof*⟩

lemma *P-inPosOf*:*RR* (*nd*, *P*) (*nd′*, *P′*) *sl* ⟹ *P* ∈ *PosOf nd*
RR (*nd*, *P*) (*nd′*, *P′*) *sl* ⟹ *P′* ∈ *PosOf nd′* ⟨*proof*⟩

interpretation *Sloped-Graph* **where**
Node = *Node* **and** *edge* = *edge* **and** *PosOf* = *PosOf*
and *RR* = *RR* ⟨*proof*⟩

lemma *i-disj*:*i* < *Suc* (*Suc* (*Suc 0*)) ⟷ *i* = *0* ∨ *i* = *1* ∨ *i* = *2* ⟨*proof*⟩

lemma *i-disj′*:*i* < *Suc* (*Suc 0*) ⟷ *i* = *0* ∨ *i* = *1* ⟨*proof*⟩

lemma *notTZ*:¬*pathG* [*node.Two*, *Zero*] ⟨*proof*⟩
lemma *notTO*:¬*pathG* [*node.Two*, *One*] ⟨*proof*⟩
lemma *notTrZ*:¬*pathG* [*node.Three*, *Zero*] ⟨*proof*⟩
lemma *notTrO*:¬*pathG* [*node.Three*, *One*] ⟨*proof*⟩
lemma *notOT*:¬*pathG* [*node.One*, *node.Two*] ⟨*proof*⟩
lemma *notOTr*:¬*pathG* [*node.One*, *node.Three*] ⟨*proof*⟩
lemma *notZTr*:¬*pathG* [*Zero*, *node.Three*] ⟨*proof*⟩

lemma *allCycles*:*basicCycle c* ⟷ *c* = [*Zero*, *One*, *Zero*] ∨ *c* = [*One*, *Zero*, *One*]
∨ *c* = [*Two*, *Three*, *Two*] ∨ *c* = [*Three*, *Two*, *Three*]
⟨*proof*⟩

lemma *pathNWF-Three*:*pathG p* ⟹ *hd p* = *Three* ⟹ *last p* ≠ *Zero* ∧ *last p* ≠ *One*
⟨*proof*⟩

lemma *pathNWF-Two*:*pathG p* ⟹ *hd p* = *Two* ⟹ *last p* ≠ *Zero* ∧ *last p* ≠ *One*
⟨*proof*⟩

lemma *pathNWF-disj*:*pathG pa* ⟹
hd pa = *node.Two* ∨ *hd pa* = *node.Three* ⟹
last pa = *Zero* ∨ *last pa* = *node.One* ⟹ *HOL.False*
⟨*proof*⟩

lemma *unicycle*:*unicyclesGraph*
⟨*proof*⟩

lemma *repeat-within*:*0* < *n* ⟹ *i*< *n* + *n* ⟹
repeat n [*node.Two*, *node.Three*] ! *i* ∈ {*Two*, *Three*} ∧

$(repeat\ n\ [node.Two,\ node.Three]\ @\ [node.Two])\ !\ Suc\ i \in \{Two,\ Three\}$
⟨*proof*⟩

lemma *noRR*:$0 < n \Longrightarrow i < n + n \Longrightarrow$
$RR\ (repeat\ n\ [node.Two,\ node.Three]\ !\ i,\ Pl\ !\ i)$
$((repeat\ n\ [node.Two,\ node.Three]\ @\ [node.Two])\ !\ Suc\ i,\ Pl\ !\ Suc\ i)$
$Decr \Longrightarrow$
$HOL.False$
⟨*proof*⟩

lemma *noDescentPath*:$0 < n \Longrightarrow descentPath$
$(repeat\ n\ (butlast\ [node.Two,\ node.Three,\ node.Two])\ @$
$[last\ [node.Two,\ node.Three,\ node.Two]])$
$Pl \Longrightarrow HOL.False$
⟨*proof*⟩

lemma *SDG*:¬*SimplyDescendingGraph*
⟨*proof*⟩

theorem ¬*InfiniteDescent*
⟨*proof*⟩

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* ≡ {*Zero*, *One*, *Two*}

lemma *Z-Node*[*simp*]:*Zero* ∈ *Node* ⟨*proof*⟩
lemma *T-Node*[*simp*]:*Two* ∈ *Node* ⟨*proof*⟩

lemma *alw-nodes*:*alw* (*holdsS Node*) *nds*
⟨*proof*⟩

fun *edge*::*node* ⇒ *node* ⇒ *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* ⇒ *pos set* **where**
PosOf Zero = {*p0*,*p0'*}|
PosOf One = {*p1*,*p1'*}|
PosOf Two = {*p2*,*p2'*}

definition *RR-set* :: ((*node* × *pos*) × (*node* × *pos*) × *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* × *pos* ⇒ *node* × *pos* ⇒ *slope* ⇒ *bool* **where**
RR np1 np2 s ≡ ((*np1*, *np2*, *s*) ∈ *RR-set*)

lemmas *RR-defs* = *RR-def RR-set-def*

lemma *P-inPosOf*:*RR* (*nd*, *P*) (*nd'*, *P'*) *sl* ⟹ *P* ∈ *PosOf nd*
RR (*nd*, *P*) (*nd'*, *P'*) *sl* ⟹ *P'* ∈ *PosOf nd'* ⟨*proof*⟩

interpretation *Sloped-Graph* **where**
Node = *Node* **and** *edge* = *edge* **and** *PosOf* = *PosOf*
and *RR* = *RR* ⟨*proof*⟩

lemma *listE*:(*i* < *length* ([*Zero*, *node.Two*] @ [*Zero*]) − *1*) ⟹ (*i* = *0* ⟹ *P*) ⟹
(*i* = *1* ⟹ *P*) ⟹ *P* ⟨*proof*⟩

lemma *ZT-FlatEdge*:(*Zero*, *Two*) ∈ *FlatEdges* ⟨*proof*⟩
lemma *TZ-FlatEdge*:(*Two*, *Zero*) ∈ *FlatEdges* ⟨*proof*⟩

lemma ¬*InfiniteDescent*
⟨*proof*⟩

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.