Abstract Soundness

Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel

February 13, 2017

Abstract

This is a formalized coinductive account of the abstract development of Brotherston et al. [2], in a slightly more general form since we work with arbitrary infinite proofs, which may be acyclic. This work is described in detail in an article by the authors [1]. The abstract proof can be instantiated for various formalisms, including first-order logic with inductive predicates.

References

[1] J. C. Blanchette, A. Popescu, and D. Traytel. Soundness and completeness proofs by coinductive methods. *J. Autom. Reasoning*, 58(1):149–179, 2017.

[2] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In R. Jhala and A. Igarashi, editors, *APLAS 2012*, volume 7705 of *Lecture Notes in Computer Science*, pages 350–367. Springer, 2012.

Contents

1 Abstract Soundness

locale *Soundness* = *RuleSystem-Defs eff rules* **for**
eff :: *'rule* ⇒ *'sequent* ⇒ *'sequent fset* ⇒ *bool*

and *rules* :: *'rule stream* +
fixes *structure* :: *'structure set*
and *sat* :: *'structure* ⇒ *'sequent* ⇒ *bool*
assumes *local-soundness*:
⋀*r s sl*.
⟦*r* ∈ *R*; *eff r s sl*; ⋀*s'*. *s'* |∈| *sl* ⟹ ∀ *S* ∈ *structure*. *sat S s'*⟧
⟹
∀ *S* ∈ *structure*. *sat S s*
begin

abbreviation *ssat s* ≡ ∀ *S* ∈ *structure*. *sat S s*

lemma *epath-shift*:
assumes *epath* (*srs* @− *steps*)
shows *epath steps*
using *assms* **by** (*induction srs arbitrary*: *steps*) (*auto elim*: *epath.cases*)

theorem *soundness*:
assumes *f*: *tfinite t* **and** *w*: *wf t*
shows *ssat* (*fst* (*root t*))
using *f w* **proof** (*induction t rule*: *tfinite.induct*)
case (*tfinite t*)
show *?case*
by (*rule local-soundness*[*of snd* (*root t*) - *fimage* (*fst o root*) (*cont t*)], *insert tfinite*)
(*fastforce elim*!: *wf.cases*)+
qed

end

2 Soundness of Infinite Proof Trees

context
begin

private definition *num P xs* ≡ *LEAST n*. *list-all* (*Not o P*) (*stake n xs*) ∧ *P* (*xs*!!*n*)

private lemma *num*:
assumes *ev*: *ev* (λ*xs*. *P* (*shd xs*)) *xs*
defines *n* ≡ *num P xs*
shows
(*list-all* (*Not o P*) (*stake n xs*) ∧ *P* (*xs*!!*n*)) ∧
(∀ *m*. *list-all* (*Not o P*) (*stake m xs*) ∧ *P* (*xs*!!*m*) ⟶ *n* ≤ *m*)
unfolding *n-def num-def*
proof (*intro conjI*[*OF LeastI-ex*] *allI impI Least-le*)
from *ev* **show** ∃ *n*. *list-all* (*Not o P*) (*stake n xs*) ∧ *P* (*xs* !! *n*)
by (*induct rule*: *ev-induct-strong*) (*auto intro*: *exI*[*of - 0*] *exI*[*of - Suc -*])

qed (*simp-all add*: *o-def*)

private lemma *num-stl*[*simp*]:
assumes *ev* (λ*xs*. *P* (*shd xs*)) *xs* **and** ¬ *P* (*shd xs*)
shows *num P xs* = *Suc* (*num P* (*stl xs*))
unfolding *num-def* **by** (*rule trans*[*OF Least-Suc*[*of* - *num P xs*]])
(*auto simp*: *num*[*OF assms*(*1*)] *assms*(*2*))

corecursive *decr0* **where**
decr0 Ord minSoFar js =
(*if* ¬ (*ev* (λ*js*. (*shd js*, *minSoFar*) ∈ *Ord* ∧ *shd js* ≠ *minSoFar*)) *js*
then undefined
else if ((*shd js*, *minSoFar*) ∈ *Ord* ∧ *shd js* ≠ *minSoFar*)
then shd js ## *decr0 Ord* (*shd js*) *js*
else decr0 Ord minSoFar (*stl js*))
by (*relation measure* (λ(*Ord*,*m*,*js*). *num* (λ*j*. (*j*, *m*) ∈ *Ord* ∧ *j* ≠ *m*) *js*)) *auto*

end

lemmas *well-order-on-defs* =
well-order-on-def linear-order-on-def partial-order-on-def
preorder-on-def trans-def antisym-def refl-on-def

lemma *sdrop-length-shift*[*simp*]:
sdrop (*length xs*) (*xs* @− *s*) = *s*
by (*simp add*: *sdrop-shift*)

lemma *ev-iff-shift*:
ev φ *xs* ⟷ (∃ *xl xs2*. *xs* = *xl* @− *xs2* ∧ φ *xs2*)
by (*meson ev.base ev-imp-shift ev-shift*)

locale *Infinite-Soundness* = *RuleSystem-Defs eff rules* **for**
eff :: *'rule* ⇒ *'sequent* ⇒ *'sequent fset* ⇒ *bool*
and *rules* :: *'rule stream*
+
fixes *structure* :: *'structure set*
and *sat* :: *'structure* ⇒ *'sequent* ⇒ *bool*
and δ :: *'sequent* ⇒ *'rule* ⇒ *'sequent* ⇒ (*'marker* × *bool* × *'marker*) *set*
and *Ord* :: *'ord rel*
and σ :: *'marker* × *'structure* ⇒ *'ord*
assumes
Ord: *well-order Ord*
and
descent:
⋀*r s sl S*.
⟦*r* ∈ *R*; *eff r s sl*; *S* ∈ *structure*; ¬ *sat S s*⟧
⟹
∃ *s' S'*.
s' |∈| *sl* ∧ *S'* ∈ *structure* ∧ ¬ *sat S' s'* ∧

$(\forall\, v\ v'\ b.$
$\quad (v,b,v') \in \delta\ s\ r\ s' \longrightarrow$
$\quad\quad (\sigma(v',S'),\ \sigma(v,S)) \in Ord \wedge (b \longrightarrow \sigma(v',S') \neq \sigma(v,S)))$

sublocale *Infinite-Soundness* < *Soundness* **where** *eff* = *eff* **and** *rules* = *rules*
and *structure* = *structure* **and** *sat* = *sat*
by *standard* (*blast dest*: *descent*)

context *Infinite-Soundness*
begin

coinductive *follow* :: *bool stream* ⇒ *'marker stream* ⇒ (*'sequent*,*'rule*)*step stream*
⇒ *bool* **where**
⟦*M'* = *shd Ms*; *s'* = *fst* (*shd steps*); (*M*,*b*,*M'*) ∈ δ *s r s'*; *follow bs Ms steps*⟧
⟹
follow (*SCons b bs*) (*SCons M Ms*) (*SCons* (*s*,*r*) *steps*)

definition *infDecr* :: *bool stream* ⇒ *bool* **where**
infDecr ≡ *alw* (*ev* (λ*bs*. *shd bs*))

definition *good* :: (*'sequent*,*'rule*)*dtree* ⇒ *bool* **where**
good t ≡ ∀ *steps*.
ipath t steps
⟶
ev (λ*steps'*. ∃ *bs Ms*. *follow bs Ms steps'* ∧ *infDecr bs*) *steps*

lemma *tfinite-good*: *tfinite t* ⟹ *good t*
using *ftree-no-ipath* **unfolding** *good-def* **by** *auto*

context
fixes *inv* :: *'sequent* × *'a* ⇒ *bool*
and *pred* :: *'sequent* × *'a* ⇒ *'rule* ⇒ *'sequent* × *'a* ⇒ *bool*
begin

primcorec *konigDtree* ::
(*'sequent*,*'rule*) *dtree* ⇒ *'a* ⇒ ((*'sequent*,*'rule*) *step* × *'a*) *stream* **where**
shd (*konigDtree t a*) = (*root t*, *a*)
|*stl* (*konigDtree t a*) =
(*let s* = *fst* (*root t*); *r* = *snd* (*root t*);
(*s'*,*a'*) = (*SOME* (*s'*,*a'*). *s'* |∈| *fimage* (*fst o root*) (*cont t*) ∧ *pred* (*s*,*a*) *r* (*s'*,*a'*)
∧ *inv* (*s'*,*a'*));
t' = (*SOME t'*. *t'* |∈| *cont t* ∧ *s'* = *fst* (*root t'*))
in konigDtree t' a'

)

lemma *stl-konigDtree*:
fixes t **defines** $s \equiv fst\ (root\ t)$ **and** $r \equiv snd\ (root\ t)$
assumes s': $s' \mid\in\mid fimage\ (fst\ o\ root)\ (cont\ t)$ **and** $pred\ (s,a)\ r\ (s',a'')$ **and** $inv\ (s',a'')$
shows $\exists t'\ a'.\ t' \mid\in\mid cont\ t \wedge pred\ (s,a)\ r\ (fst\ (root\ t'),a') \wedge inv\ (fst\ (root\ t'),a') \wedge stl\ (konigDtree\ t\ a) = konigDtree\ t'\ a'$
proof−
define P **where** $P \equiv \lambda(s',a').\ s' \mid\in\mid fimage\ (fst\ o\ root)\ (cont\ t) \wedge pred\ (s,a)\ r\ (s',a') \wedge inv\ (s',a')$
define $s'a'$ **where** $s'a' \equiv SOME\ (s',a').\ P\ (s',a')$ **let** $?s' = fst\ s'a'$ **let** $?a' = snd\ s'a'$
define t' **where** $t' \equiv SOME\ (t'::('sequent,'rule)dtree).\ t' \mid\in\mid cont\ t \wedge ?s' = fst\ (root\ t')$
have $P\ (s',a'')$ **using** *assms* **unfolding** *P-def* **by** *auto*
hence P: $P\ (?s',?a')$ **using** *someI*[*of P*] **unfolding** *s'a'-def* **by** *auto*
hence $\exists t'.\ t' \mid\in\mid cont\ t \wedge ?s' = fst\ (root\ t')$ **unfolding** *P-def* **by** *auto*
hence t': $t' \mid\in\mid cont\ t$ **and** s': $?s' = fst\ (root\ t')$
using *someI-ex*[*of* $\lambda t'.\ t' \mid\in\mid cont\ t \wedge ?s' = fst\ (root\ t')$] **unfolding** *t'-def* **by** *auto*
show *?thesis* **using** t' P s' *assms P-def s'a'-def t'-def* **by** (*intro exI*[*of - t'*] *exI*[*of - ?a'*]) *auto*
qed

declare *konigDtree.simps*(*2*)[*simp del*]

lemma *konigDtree*:
assumes *1*: $\bigwedge r\ s\ sl\ a.$
$[\![r \in R;\ eff\ r\ s\ sl;\ inv\ (s,a)]\!] \Longrightarrow$
$\exists s'\ a'.\ s' \mid\in\mid sl \wedge inv\ (s',a') \wedge pred\ (s,a)\ r\ (s',a')$
and *2*: $wf\ t\ inv\ (fst\ (root\ t),\ a)$
shows
alw (λ*stepas*.
let $((s,r),a) = shd\ stepas;\ ((s',\text{-}),a') = shd\ (stl\ stepas)$ *in*
$inv\ (s,a) \wedge pred\ (s,a)\ r\ (s',a')$)
($konigDtree\ t\ a$)
using *assms* **proof** (*coinduction arbitrary*: $t\ a$)
case ($alw\ t\ a$)
then obtain $s'\ a'$ **where** $s' \mid\in\mid (fst \circ root) \mid`\mid cont\ t\ inv\ (s',\ a')$
$pred\ (fst\ (root\ t),\ a)\ (snd\ (root\ t))\ (s',\ a')$
by (*auto elim*!: *wf.cases dest*!: *spec*[*of - snd* (*root t*)] *spec*[*of - fst* (*root t*)]
spec[*of* - (*fst* $\circ$ *root*) $\mid`\mid$ *cont t*] *spec*[*of - a*], *fastforce*)
with *alw stl-konigDtree*[*of s' t a a'*] **show** *?case*
by (*auto split*: *prod.splits elim*!: *wf.cases*) *fastforce*
qed

lemma *konigDtree-ipath*:
assumes $\bigwedge r\ s\ sl\ a.$

⟦$r \in R$; *eff* r s *sl*; *inv* (s,a)⟧ ⟹
∃ s' a'. s' |∈| *sl* ∧ *inv* (s',a') ∧ *pred* (s,a) r (s',a')
and *wf* t **and** *inv* (*fst* (*root* t), a)
shows *ipath* t (*smap fst* (*konigDtree* t a))
using *assms* **proof** (*coinduction arbitrary*: t a)
case (*ipath* t a)
then obtain s' a' **where** s' |∈| (*fst* ∘ *root*) |‘| *cont* t *inv* $(s',\ a')$
pred (*fst* (*root* t), a) (*snd* (*root* t)) $(s',\ a')$
by (*auto elim*!: *wf.cases dest*!: *spec*[*of* - *snd* (*root* t)] *spec*[*of* - *fst* (*root* t)]
spec[*of* - (*fst* ∘ *root*) |‘| *cont* t] *spec*[*of* - a], *fastforce*)
with *ipath stl-konigDtree*[*of* s' t a a'] **show** *?case*
by (*auto split*: *prod.splits elim*!: *wf.cases*) *force*
qed

end

lemma *follow-stl-smap-fst*[*simp*]:
follow bs Ms (*smap fst stepSs*) ⟹
follow (*stl bs*) (*stl Ms*) (*smap fst* (*stl stepSs*))
by (*erule follow.cases*) (*auto simp del*: *stream.map-sel simp add*: *stream.map-sel*[*symmetric*])

lemma *epath-stl-smap-fst*[*simp*]:
epath (*smap fst stepSs*) ⟹
epath (*smap fst* (*stl stepSs*))
by (*erule epath.cases*) (*auto simp del*: *stream.map-sel simp add*: *stream.map-sel*[*symmetric*])

lemma *infDecr-tl*[*simp*]: *infDecr bs* ⟹ *infDecr* (*stl bs*)
unfolding *infDecr-def* **by** *auto*

fun *descent* **where** *descent* (s,S) r (s',S') =
(∀ v v' b.
$(v,b,v') \in \delta$ s r s' ⟶
$(\sigma(v',S'),\ \sigma(v,S)) \in Ord \wedge (b \longrightarrow \sigma(v',S') \neq \sigma(v,S))$)

lemma *descentE*[*elim*]:
assumes *descent* (s,S) r (s',S') **and** $(v,b,v') \in \delta$ s r s'
shows $(\sigma(v',S'),\ \sigma(v,S)) \in Ord \wedge (b \longrightarrow \sigma(v',S') \neq \sigma(v,S))$
using *assms* **by** *auto*

definition *konigDown* ≡ *konigDtree* (λ(s,S). $S \in$ *structure* ∧ ¬ *sat* S s) *descent*

lemma *konigDown*:
assumes *wf* t **and** $S \in$ *structure* **and** ¬ *sat* S (*fst* (*root* t))
shows
alw (λ*stepSs*. *let* $((s,r),S)$ = *shd stepSs*; $((s',\text{-}),S')$ = *shd* (*stl stepSs*) *in*
$S \in$ *structure* ∧ ¬ *sat* S s ∧ *descent* (s,S) r (s',S'))
(*konigDown* t S)
using *konigDtree*[*of* λ(s,S). $S \in$ *structure* ∧ ¬ *sat* S s *descent*, *unfolded konigDown-def*[*symmetric*]]

using *assms descent* **by** *auto*

lemma *konigDown-ipath*:
assumes *wf t* **and** $S \in$ *structure* **and** $\neg$ *sat S* (*fst* (*root t*))
shows
ipath t (*smap fst* (*konigDown t S*))
using *konigDtree-ipath*[*of* $\lambda(s,S)$. $S \in$ *structure* $\wedge \neg$ *sat S s descent*, *unfolded konigDown-def*[*symmetric*]]
using *assms descent* **by** *auto*

context
fixes *t S*
assumes *w*: *wf t* **and** *t*: *good t* **and** *S*: $S \in$ *structure* $\neg$ *sat S* (*fst* (*root t*))
begin

lemma *alw-ev-Ord*:
obtains *ks* **where** *alw* (λks. (*shd* (*stl ks*), *shd ks*) $\in$ *Ord*) *ks*
and *alw* (*ev* (λks. *shd* (*stl ks*) $\neq$ *shd ks*)) *ks*
proof$-$
define *P* **where** $P \equiv \lambda stepSs$. *let* ((*s*,*r*),*S*) = *shd stepSs*; ((*s*′,-),*S*′) = *shd* (*stl stepSs*) *in*
$S \in$ *structure* $\wedge \neg$ *sat S s* $\wedge$ *descent* (*s*,*S*) *r* (*s*′,*S*′)
have *alw P* (*konigDown t S*) **using** *konigDown*[*OF w S*] **unfolding** *P-def* **by** *auto*
obtain *srs steps bs Ms* **where** *0*: *smap fst* (*konigDown t S*) = *srs* @$-$ *steps* **and**
f: *follow bs Ms steps* **and** *i*: *infDecr bs*
using *konigDown-ipath*[*OF w S*] *t* **unfolding** *good-def ev-iff-shift* **by** *auto*
define *stepSs* **where** *stepSs* = *sdrop* (*length srs*) (*konigDown t S*)
have *steps*: *steps* = *smap fst stepSs* **unfolding** *stepSs-def sdrop-smap*[*symmetric*] *0* **by** *simp*
have *e*: *epath steps*
using *wf-ipath-epath*[*OF w konigDown-ipath*[*OF w S*]] *0 epath-shift* **by** *simp*
have *alw P* (*konigDown t S*) **using** *konigDown*[*OF w S*] **unfolding** *P-def* **by** *auto*
hence *P*: *alw P stepSs* **using** *alw-sdrop* **unfolding** *stepSs-def* **by** *auto*
let *?ks* = *smap* σ (*szip Ms* (*smap snd stepSs*))
show *?thesis* **proof**(*rule that*[*of ?ks*])
show *alw* (λks. (*shd* (*stl ks*), *shd ks*) $\in$ *Ord*) *?ks*
using *e f P* **unfolding** *steps* **proof**(*coinduction arbitrary*: *bs Ms stepSs rule*: *alw-coinduct*)
case (*alw bs Ms stepSs*)
let *?steps* = *smap fst stepSs* **let** *?Ss* = *smap snd stepSs*
let *?MSs* = *szip Ms* (*smap snd stepSs*)
let *?s* = *fst* (*shd ?steps*) **let** *?s*′ = *fst* (*shd* (*stl ?steps*))
let *?r* = *snd* (*shd ?steps*)
let *?S* = *snd* (*shd stepSs*) **let** *?S*′ = *snd* (*shd* (*stl stepSs*))
let *?M* = *shd Ms* **let** *?M*′ = *shd* (*stl Ms*) **let** *?b* = *shd bs*
have *1*: (*?M*, *?b*, *?M*′) $\in \delta$ *?s ?r ?s*′
using ‹*follow bs Ms* (*smap fst stepSs*)› **by** (*cases rule*: *follow.cases*) *auto*

```
    have 2: descent (?s,?S) ?r (?s',?S')
      using ‹alw P stepSs› unfolding P-def by (cases rule: alw.cases) auto
    have (σ(?M',?S'), σ(?M,?S)) ∈ Ord using descentE[OF 2 1] by simp
    thus ?case by simp
  next
    case (stl bs Ms stepSs)
    thus ?case
      by (intro exI[of - stl bs] exI[of - stl Ms] exI[of - stl stepSs])
        (auto elim: epath.cases)
  qed
next
  show alw (ev (λks. shd (stl ks) ≠ shd ks)) ?ks
    using e f P i unfolding steps proof(coinduction arbitrary: bs Ms stepSs
rule: alw-coinduct)
    case (alw bs Ms stepSs)
    let ?steps = smap fst stepSs let ?Ss = smap snd stepSs
    let ?MSs = szip Ms (smap snd stepSs)
    let ?s = fst (shd ?steps) let ?s' = fst (shd (stl ?steps))
    let ?r = snd (shd ?steps)
    let ?S = snd (shd stepSs) let ?S' = snd (shd (stl stepSs))
    let ?M = shd Ms let ?M' = shd (stl Ms) let ?b = shd bs
    have 1: (?M, ?b, ?M') ∈ δ ?s ?r ?s'
      using ‹follow bs Ms (smap fst stepSs)› by (cases rule: follow.cases) auto
    have 2: descent (?s,?S) ?r (?s',?S')
      using ‹alw P stepSs› unfolding P-def by (cases rule: alw.cases) auto
    have (σ(?M',?S'), σ(?M,?S)) ∈ Ord using descentE[OF 2 1] by simp
    have ev shd bs using ‹infDecr bs› unfolding infDecr-def by auto
    thus ?case using ‹epath ?steps› ‹follow bs Ms ?steps› ‹alw P stepSs›
    proof (induction arbitrary: Ms stepSs)
      case (base bs Ms stepSs)
      let ?steps = smap fst stepSs let ?Ss = smap snd stepSs
      let ?MSs = szip Ms (smap snd stepSs)
      let ?s = fst (shd ?steps) let ?s' = fst (shd (stl ?steps))
      let ?r = snd (shd ?steps)
      let ?S = snd (shd stepSs) let ?S' = snd (shd (stl stepSs))
      let ?M = shd Ms let ?M' = shd (stl Ms) let ?b = shd bs
      have 1: (?M, ?b, ?M') ∈ δ ?s ?r ?s'
        using ‹follow bs Ms (smap fst stepSs)› by (cases rule: follow.cases) auto
      have 2: descent (?s,?S) ?r (?s',?S')
        using ‹alw P stepSs› unfolding P-def by (cases rule: alw.cases) auto
      have σ(?M',?S') ≠ σ(?M,?S) using descentE[OF 2 1] ‹shd bs› by simp
      thus ?case by auto
    next
      case (step bs Ms stepSs)
      have ev (λks. shd (stl ks) ≠ shd ks)
            (smap σ
            (szip (stl Ms) (smap snd (stl stepSs))))
        using step(3−5) step(2)[of stl stepSs stl Ms] by auto
      thus ?case by auto
```

```
    qed
  next
    case (stl bs Ms stepSs)
    thus ?case
      by (intro exI[of - stl bs] exI[of - stl Ms] exI[of - stl stepSs])
        (auto elim: epath.cases)
  qed
 qed
qed

definition
 ks ≡ SOME ks.
       alw (λks. (shd (stl ks), shd ks) ∈ Ord) ks ∧
       alw (ev (λks. shd (stl ks) ≠ shd ks)) ks

lemma alw-ks: alw (λks. (shd (stl ks), shd ks) ∈ Ord) ks
 and alw-ev-ks: alw (ev (λks. shd (stl ks) ≠ shd ks)) ks
 unfolding ks-def using alw-ev-Ord someI-ex[of λks.
       alw (λks. (shd (stl ks), shd ks) ∈ Ord) ks ∧
       alw (ev (λks. shd (stl ks) ≠ shd ks)) ks]
 by auto

abbreviation decr where decr ≡ decr0 Ord

lemmas decr-simps = decr0.code[of Ord]

context
 fixes js
 assumes a: alw (λjs. (shd (stl js), shd js) ∈ Ord) js
   and ae: alw (ev (λjs. shd (stl js) ≠ shd js)) js
begin

lemma decr-ev:
 assumes m: (shd js, m) ∈ Ord
 shows ev (λjs. (shd js, m) ∈ Ord ∧ shd js ≠ m) js
   (is ev (λjs. ?φ m js) js)
proof−
 have ev (λjs. shd (stl js) ≠ shd js) js using ae by auto
 thus ?thesis
   using a m proof induction
   case (base ls)
   hence ev (?φ (shd ls)) ls by auto
   moreover have ⋀js. ?φ (shd ls) js ⟹ ?φ m js
     using ‹(shd ls, m) ∈ Ord› Ord unfolding well-order-on-defs by blast
   ultimately show ?case using ev-mono[of ?φ (shd ls) - ?φ m] by auto
 qed auto
qed

lemma decr-simps-diff[simp]:
```

assumes m: $(shd\ js,\ m) \in Ord$
and $shd\ js \neq m$
shows $decr\ m\ js = shd\ js\ \#\#\ decr\ (shd\ js)\ js$
using $decr\text{-}ev[OF\ m]\ assms$ **by** $(subst\ decr\text{-}simps)\ simp$

lemma *decr-simps-eq*[*simp*]:
$decr\ (shd\ js)\ js = decr\ (shd\ js)\ (stl\ js)$
proof−
have m: $(shd\ js,\ shd\ js) \in Ord$ **using** Ord
unfolding *well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def refl-on-def* **by** *auto*
show *?thesis* **using** $decr\text{-}ev[OF\ m]$ **by** $(subst\ decr\text{-}simps)\ simp$
qed

end

lemma *stl-decr*:
assumes a: $alw\ (\lambda js.\ (shd\ (stl\ js),\ shd\ js) \in Ord)\ js$
and ae: $alw\ (ev\ (\lambda js.\ shd\ (stl\ js) \neq shd\ js))\ js$
and m: $(shd\ js,\ m) \in Ord$
shows
$\exists js1\ js2.\ js = js1\ @-\ js2 \wedge set\ js1 \subseteq \{m\} \wedge$
$(shd\ js2,\ m) \in Ord \wedge shd\ js2 \neq m \wedge$
$shd\ (decr\ m\ js) = shd\ js2 \wedge stl\ (decr\ m\ js) = decr\ (shd\ js2)\ js2$
(**is** $\exists js1\ js2.\ ?\varphi\ js\ js1\ js2$)
using $decr\text{-}ev[OF\ assms]\ m\ a\ ae$ **proof** (*induction rule*: *ev-induct-strong*)
case (*base js*)
thus *?case* **by** (*intro exI*[*of* - []] *exI*[*of* - *js*]) *auto*
next
case (*step js*)
then obtain *js1 js2* **where** *1*: $?\varphi\ (stl\ js)\ js1\ js2$ **and** [*simp*]: $shd\ js = m$ **by** *auto*
thus *?case*
by (*intro exI*[*of* - *shd js # js1*] *exI*[*of* - *js2*],
simp, *metis* (*lifting*) *decr-simps-eq step*(*2*,*4*,*5*,*6*) *stream.collapse*)
qed

corollary *stl-decr-shd*:
assumes a: $alw\ (\lambda js.\ (shd\ (stl\ js),\ shd\ js) \in Ord)\ js$ **and**
ae: $alw\ (ev\ (\lambda js.\ shd\ (stl\ js) \neq shd\ js))\ js$
shows
$\exists js1\ js2.\ js = js1\ @-\ js2 \wedge set\ js1 \subseteq \{shd\ js\} \wedge$
$(shd\ js2,\ shd\ js) \in Ord \wedge shd\ js2 \neq shd\ js \wedge$
$shd\ (decr\ (shd\ js)\ js) = shd\ js2 \wedge stl\ (decr\ (shd\ js)\ js) = decr\ (shd\ js2)\ js2$
using Ord **unfolding** *well-order-on-defs* **by** (*intro stl-decr*[*OF assms*]) *blast*

lemma *decr*:
assumes a: $alw\ (\lambda js.\ (shd\ (stl\ js),\ shd\ js) \in Ord)\ js$ (**is** $?a\ js$)
and ae: $alw\ (ev\ (\lambda js.\ shd\ (stl\ js) \neq shd\ js))\ js$ (**is** $?ae\ js$)

```
  shows
    alw (λjs. (shd (stl js), shd js) ∈ Ord ∧ shd (stl js) ≠ shd js) (decr (shd js) js)
    (is alw ?φ -)
proof−
  let ?ξ = λls js. ls = decr (shd js) js ∧ ?a js ∧ ?ae js
  {fix ls assume ∃ js. ?ξ ls js
    hence alw ?φ ls proof(elim alw-coinduct)
      fix ls assume ∃ js. ?ξ ls js
      then obtain js where 1: ?ξ ls js by auto
      then obtain js1 js2 where js: js = js1 @− js2 ∧ set js1 ⊆ {shd js} ∧
       (shd js2, shd js) ∈ Ord ∧ shd js2 ≠ shd js ∧
       shd ls = shd js2 ∧ stl ls = decr (shd js2) js2
        using stl-decr-shd by blast
      then obtain js3 js4 where js2: js2 = js3 @− js4 ∧ set js3 ⊆ {shd js2} ∧
       (shd js4, shd js2) ∈ Ord ∧ shd js4 ≠ shd js2 ∧
       shd (decr (shd js2) js2) = shd js4 ∧ stl ((decr (shd js2) js2)) = decr (shd
js4) js4
        using stl-decr-shd[of js2] a ae using 1 alw-shift by blast
      show ?φ ls using 1 js js2 by metis
    qed (metis (no-types, lifting) alw-shift stl-decr-shd)
  }
  thus ?thesis using assms by blast
qed

lemma alw-snth:
  assumes alw (λxs. P (shd (stl xs)) (shd xs)) xs
  shows P (xs!!(Suc n)) (xs!! n)
  using assms
  by (induction n, auto, metis (mono-tags) alw.cases alw-iff-sdrop sdrop-simps(1)
sdrop-stl)

lemma F: False
proof−
  define ls where ls = decr (shd ks) ks
  have 0: alw (λjs. (shd (stl js), shd js) ∈ Ord ∧ shd (stl js) ≠ shd js) ls
    using decr[OF alw-ks alw-ev-ks] unfolding ls-def .
  define Q where Q = range (snth ls)  let ?wf = Wellfounded.wf
  have Q: Q ≠ {} unfolding Q-def by auto
  have 1: ?wf (Ord − Id) using Ord unfolding well-order-on-def by auto
  obtain q where q: q ∈ Q and 2: ∀ q'. (q',q) ∈ Ord − Id ⟶ q' ∉ Q
    using wfE-min[OF 1] Q by auto
  obtain n where ls!!n = q using q unfolding Q-def by auto
  hence (ls!!(Suc n),q) ∈ Ord − Id using alw-snth[OF 0] by auto
  thus False using 2 Q-def by blast
qed

end
```

theorem *infinite-soundness*:
assumes *wf t* **and** *good t* **and** $S \in$ *structure*
shows *sat S* (*fst* (*root t*))
using *F*[*OF assms*] **by** *auto*

end

3 Soundness of Cyclic Proof Trees

datatype (*discs-sels*) ('*sequent*, '*rule*, '*link*) *ctree* =
Link '*link* |
cNode ('*sequent*,'*rule*) *step* ('*sequent*, '*rule*, '*link*) *ctree fset*

corecursive *treeOf* **where**
treeOf pointsTo ct =
(*if* $\exists$ *l l'*. *pointsTo l* = *Link l'*
(∗ *makes sense only if backward links point to normal nodes, not to backwards links*: ∗)
then undefined
else (*case ct of*
Link l ⇒ *treeOf pointsTo* (*pointsTo l*)
|*cNode step cts* ⇒ *Node step* (*fimage* (*treeOf pointsTo*) *cts*)
)
)
by (*relation measure* (λ(*p*,*t*). *case t of Link l'* => *Suc 0* | - => *0*)) (*auto split*: *ctree.splits*)

declare *treeOf.code*[*simp*]

context *Infinite-Soundness*
begin

context
fixes *pointsTo* :: '*link* ⇒ ('*sequent*, '*rule*, '*link*)*ctree*
assumes *pointsTo*: ∀ *l l'*. *pointsTo l* ≠ *Link l'*
begin

function *seqOf* **where**
seqOf (*Link l*) = *seqOf* (*pointsTo l*)
|
seqOf (*cNode* (*s*,*r*) -) = *s*
by *pat-completeness auto*
termination
by (*relation measure* (λ*t*. *case t of Link l'* => *Suc 0* | - => *0*))
(*auto split*: *ctree.splits simp*: *pointsTo*)

coinductive *cwf* **where**
Node[*intro*!]: *cwf* (*pointsTo l*) ⟹ *cwf* (*Link l*)

```
|
  cNode[intro]:
  ⟦r ∈ R; eff r s (fimage seqOf cts); ⋀ct'. ct' |∈| cts ⟹ cwf ct'⟧
⟹
cwf (cNode (s,r) cts)

definition cgood ct ≡ good (treeOf pointsTo ct)

lemma cwf-Link: cwf (Link l) ⟷ cwf (pointsTo l)
  by (auto elim: cwf.cases)

lemma cwf-cNode-seqOf:
  cwf (cNode (s, r) cts) ⟹ eff r s (fimage seqOf cts)
  by (auto elim: cwf.cases)

lemma treeOf-seqOf[simp]:
  fst ∘ root ∘ treeOf pointsTo = seqOf
proof(rule ext, unfold o-def)
  fix ct show fst (root (treeOf pointsTo ct)) = seqOf ct
    by induct (auto split: ctree.splits simp: pointsTo)
qed

lemma wf-treeOf:
  assumes cwf ct
  shows wf (treeOf pointsTo ct)
proof−
  {fix t let ?φ = λct t. cwf ct ∧ t = treeOf pointsTo ct
    assume ∃ ct. ?φ ct t hence wf t
    proof(elim wf.coinduct, safe)
      fix ct let ?t = treeOf pointsTo ct
      assume ct: cwf ct
      show
       ∃ t. treeOf pointsTo ct = t ∧
           snd (root t) ∈ R ∧
           effStep (root t) (fimage (fst ∘ root) (cont t)) ∧
           (∀ t'. t' |∈| cont t ⟶ (∃ ct'. ?φ ct' t') ∨ wf t')
      proof(rule exI[of - ?t], safe)
        show snd (root ?t) ∈ R using pointsTo ct
          by (auto elim: cwf.cases split: ctree.splits simp: cwf-Link)
        show effStep (root ?t) (fimage (fst ∘ root) (cont ?t))
            using pointsTo ct by (auto  elim: cwf.cases split: ctree.splits simp:
cwf-Link)
        {fix t' assume t': t' |∈| cont ?t
          show ∃ ct'. ?φ ct' t'
          proof(cases ct)
            case (Link l)
            then obtain s r cts where pl: pointsTo l = cNode (s,r) cts
              using pointsTo by (cases pointsTo l) auto
            obtain ct' where ct': ct' |∈| cts and t' = treeOf pointsTo ct'
```

```
          using t' by (auto simp: Link pl pointsTo split: ctree.splits)
        moreover have cwf ct' using ct' ct pl unfolding Link
          by (auto simp: cwf-Link elim: cwf.cases)
        ultimately show ?thesis by blast
      next
        case (cNode step cts)
         then obtain s r where cNode: ct = cNode (s,r) cts by (cases step)
auto
        obtain ct' where ct': ct' |∈| cts and t' = treeOf pointsTo ct'
          using t' by (auto simp: cNode pointsTo split: ctree.splits)
        moreover have cwf ct' using ct' ct unfolding cNode
          by (auto simp: cwf-Link elim: cwf.cases)
        ultimately show ?thesis by blast
      qed
    }
   qed
  qed
 }
 thus ?thesis using assms by blast
qed

theorem cyclic-soundness:
 assumes cwf ct and cgood ct and S ∈ structure
 shows sat S (seqOf ct)
 using infinite-soundness wf-treeOf assms
 unfolding cgood-def treeOf-seqOf[symmetric] comp-def
 by blast

end

end
```

4 Appendix: The definition of treeOf under more flexible assumptions about pointsTo

```
definition rels where
 rels pointsTo ≡ {((pointsTo, pointsTo l'), (pointsTo, Link l')) | l'. True}

definition rel :: (('link ⇒ ('sequent, 'rule, 'link) ctree) × ('sequent, 'rule, 'link)
ctree) rel where
 rel ≡ UNION {pointsTo . wf {(l,l'). pointsTo l' = Link l}} rels

lemma wf-rels[simp]:
 assumes wf {(l,l'). (pointsTo :: 'link ⇒ ('sequent, 'rule, 'link)ctree) l' = Link
l}
   (is wf ?w)
 shows wf (rels pointsTo) using wf-map-prod-image
proof −
```

```
  define r1 :: (('link ⇒ ('sequent, 'rule, 'link) ctree) × ('sequent, 'rule, 'link)
ctree) rel where
    r1 = {((pointsTo,pointsTo l'), (pointsTo, Link l'::('sequent, 'rule, 'link) ctree))
| l'.
                (∀ l''. pointsTo l' ≠ Link l'')}
  define r2 :: (('link ⇒ ('sequent, 'rule, 'link) ctree) × ('sequent, 'rule, 'link)
ctree) rel where
    r2 = image (map-prod (map-prod id Link) (map-prod id Link)) (inv-image ?w
snd)
  have 0: rels pointsTo ⊆ r1 ∪ r2
    unfolding rels-def r1-def r2-def unfolding inv-image-def image-Collect by
auto
  let ?m = measure (λ(tOfL,t). case t of Link l' => Suc 0 | - => 0)
 have 1: wf r1 unfolding r1-def by (rule wf-subset[of ?m]) (auto split: ctree.splits)
  have 2: wf r2 using assms unfolding r2-def
   by (intro wf-map-prod-image wf-inv-image) (auto simp: inj-on-def)
  have 3: Domain r1 ∩ Range r2 = {} unfolding r1-def r2-def by auto
  show ?thesis using 1 2 3 by (intro wf-subset[OF - 0] wf-Un) auto
qed

lemma rel: wf rel
  unfolding rel-def
  apply(rule wf-UN)
  subgoal by (auto intro: wf-UN)
  unfolding rels-def by auto

corecursive treeOf' where
  treeOf' pointsTo ct =
   (if ¬ wf {(l',l). pointsTo l = Link l'}
    (* makes sense only if backward links point to normal nodes, not to backwards
links: *)
     then undefined
     else (case ct of
            Link l ⇒ treeOf' pointsTo (pointsTo l)
           |cNode step cts ⇒ Node step (fimage (treeOf' pointsTo) cts)
         )
   )
  apply(relation rel) using rel unfolding rel-def rels-def[abs-def] by auto
```