

Abstract Completeness

Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel

February 6, 2026

Abstract

This is a formalization of an abstract property of possibly infinite derivation trees (modeled by a codatatype), that represents the core of a Beth–Hintikka-style proof of the first-order logic completeness theorem and is independent of the concrete syntax or inference rules. This work is described in detail in a publication by the authors [2].

The abstract proof can be instantiated for a wide range of Gentzen and tableau systems as well as various flavors of FOL—e.g., with or without predicates, equality, or sorts. Here, we give only a toy example instantiation with classical propositional logic. A more serious instance—many-sorted FOL with equality—is described elsewhere [1].

References

- [1] J. C. Blanchette and A. Popescu. Mechanizing the metatheory of sledgehammer. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *FroCoS 2013*, volume 8152 of *LNCS*, pages 245–260. Springer, 2013.
- [2] J. C. Blanchette, A. Popescu, and D. Traytel. Unified classical logic completeness: A coinductive pearl. In S. Demri, D. Kapur, and C. Weidenbach, editors, *IJCAR 2014*, LNCS. Springer, 2014.

Contents

| | | |
|----------|---|----------|
| 1 | General Tree Concepts | 2 |
| 2 | Rule Systems | 2 |
| 3 | A Fair Enumeration of the Rules | 3 |
| 4 | Persistent rules | 5 |
| 5 | Code generation | 6 |
| 6 | Toy instantiation: Propositional Logic | 6 |

1 General Tree Concepts

codatatype 'a tree = Node (root: 'a) (cont: 'a tree fset)

inductive tfinite **where**

tfinite: $(\bigwedge t'. t' \in | \text{cont } t \implies \text{tfinite } t') \implies \text{tfinite } t$

coinductive ipath **where**

ipath: $\llbracket \text{root } t = \text{shd steps}; t' \in | \text{cont } t; \text{ipath } t' (\text{stl steps}) \rrbracket \implies \text{ipath } t \text{ steps}$
(proof)

primcorec konig **where**

shd (konig t) = root t
| stl (konig t) = konig (SOME t'. t' ∈ | cont t ∧ ¬ tfinite t')

lemma Konig: $\neg \text{tfinite } t \implies \text{ipath } t (\text{konig } t)$

(proof)

2 Rule Systems

type-synonym ('state, 'rule) step = 'state × 'rule **type-synonym** ('state, 'rule)

dtree = ('state, 'rule) step tree

locale RuleSystem-Defs =

fixes eff :: 'rule ⇒ 'state ⇒ 'state fset ⇒ bool

and rules :: 'rule stream

begin

abbreviation R ≡ sset rules

lemma countable-R: countable R (proof)

lemma NE-R: $R \neq \{\}$ (proof)

definition enabled r s ≡ ∃ sl. eff r s sl

definition pickEff r s ≡ if enabled r s then (SOME sl. eff r s sl) else the None

lemma pickEff: enabled r s ⇒ eff r s (pickEff r s)

(proof)

abbreviation effStep step ≡ eff (snd step) (fst step)

abbreviation enabledAtStep r step ≡ enabled r (fst step)

abbreviation takenAtStep r step ≡ snd step = r

Saturation is a very strong notion of fairness: If a rule is enabled at some point, it will eventually be taken.

definition saturated r ≡ alw (holds (enabledAtStep r) impl ev (holds (takenAtStep r)))

definition Saturated steps ≡ ∀ r ∈ R. saturated r steps

coinductive wf **where**

wf: $\llbracket \text{snd} (\text{root } t) \in R; \text{effStep} (\text{root } t) (\text{fimage} (\text{fst } o \text{ root}) (\text{cont } t));$
 $\wedge t'. t' \mid \in \mid \text{cont } t \implies \text{wf } t' \rrbracket \implies \text{wf } t$

coinductive *epath* **where**

epath: $\llbracket \text{snd} (\text{shd } \text{steps}) \in R; \text{fst} (\text{shd} (\text{stl } \text{steps})) \mid \in \mid \text{sl}; \text{effStep} (\text{shd } \text{steps}) \text{sl};$
 $\text{epath} (\text{stl } \text{steps}) \rrbracket \implies \text{epath } \text{steps}$

lemma *wf-ipath-epath*:

assumes *wf t ipath t steps*

shows *epath steps*

<proof>

definition *fair rs* $\equiv \text{sset } rs \subseteq R \wedge (\forall r \in R. \text{alw} (\text{ev} (\text{holds} ((=) r))) rs)$

lemma *fair-stl*: *fair rs* \implies *fair (stl rs)*

<proof>

lemma *sdrop-fair*: *fair rs* \implies *fair (sdrop m rs)*

<proof>

3 A Fair Enumeration of the Rules

definition *fenum* $\equiv \text{flat} (\text{smap} (\lambda n. \text{stake } n \text{ rules}) (\text{fromN } 1))$

lemma *sset-fenum*: *sset fenum* = *R*

<proof>

lemma *fair-fenum*: *fair fenum*

<proof>

definition *trim rs s* = *sdrop-while* ($\lambda r. \text{Not} (\text{enabled } r \text{ s})$) *rs*

primcorec *mkTree* **where**

root (*mkTree rs s*) = (*s*, (*shd* (*trim rs s*)))

\mid *cont* (*mkTree rs s*) = *fimage* (*mkTree* (*stl* (*trim rs s*))) (*pickEff* (*shd* (*trim rs s*))

s)

lemma *mkTree-unfold*[code]: *mkTree rs s* =

(*case trim rs s of SCons r s' \Rightarrow Node (s, r) (fimage (mkTree s') (pickEff r s))*)

<proof>

end

locale *RuleSystem* = *RuleSystem-Defs* *eff rules*

for *eff* :: '*rule* \Rightarrow '*state* \Rightarrow '*state* *fset* \Rightarrow *bool* **and** *rules* :: '*rule* *stream* +

fixes *S* :: '*state* *set*

assumes *eff-S*: $\bigwedge s r sl s'. \llbracket s \in S; r \in R; \text{eff } r \text{ s sl}; s' \mid \in \mid \text{sl} \rrbracket \implies s' \in S$

and *enabled-R*: $\bigwedge s. s \in S \implies \exists r \in R. \exists sl. \text{eff } r \text{ s sl}$

begin

definition *minWait rs s* $\equiv \text{LEAST } n. \text{enabled} (\text{shd} (\text{sdrop } n \text{ rs})) s$

lemma *trim-alt*:

assumes $s: s \in S$ **and** $rs: \text{fair } rs$
shows $\text{trim } rs \ s = \text{sdrop } (\text{minWait } rs \ s) \ rs$
(proof)

lemma *minWait-ex*:

assumes $s: s \in S$ **and** $rs: \text{fair } rs$
shows $\exists n. \text{enabled } (\text{shd } (\text{sdrop } n \ rs)) \ s$
(proof)

lemma **assumes** $s \in S$ **and** $\text{fair } rs$

shows $\text{trim-in-R}: \text{shd } (\text{trim } rs \ s) \in R$
and $\text{trim-enabled}: \text{enabled } (\text{shd } (\text{trim } rs \ s)) \ s$
and $\text{trim-fair}: \text{fair } (\text{trim } rs \ s)$
(proof)

lemma *minWait-least*: $\llbracket \text{enabled } (\text{shd } (\text{sdrop } n \ rs)) \ s \rrbracket \implies \text{minWait } rs \ s \leq n$
(proof)

lemma *in-cont-mkTree*:

assumes $s: s \in S$ **and** $rs: \text{fair } rs$ **and** $t': t' \in | \text{cont } (\text{mkTree } rs \ s)$
shows $\exists sl' \ s'. s' \in S \wedge \text{eff } (\text{shd } (\text{trim } rs \ s)) \ s \ sl' \wedge$
 $s' \in | \text{sl}' \wedge t' = \text{mkTree } (\text{stl } (\text{trim } rs \ s)) \ s'$
(proof)

lemma *ipath-mkTree-sdrop*:

assumes $s: s \in S$ **and** $rs: \text{fair } rs$ **and** $i: \text{ipath } (\text{mkTree } rs \ s) \ \text{steps}$
shows $\exists n \ s'. s' \in S \wedge \text{ipath } (\text{mkTree } (\text{sdrop } n \ rs) \ s') \ (\text{sdrop } n \ \text{steps})$
(proof)

lemma *wf-mkTree*:

assumes $s: s \in S$ **and** $\text{fair } rs$
shows $\text{wf } (\text{mkTree } rs \ s)$
(proof) **definition** $\text{pos } rs \ r \equiv \text{LEAST } n. \text{shd } (\text{sdrop } n \ rs) = r$

lemma *pos*: $\llbracket \text{fair } rs; r \in R \rrbracket \implies \text{shd } (\text{sdrop } (\text{pos } rs \ r) \ rs) = r$
(proof)

lemma *pos-least*: $\text{shd } (\text{sdrop } n \ rs) = r \implies \text{pos } rs \ r \leq n$
(proof)

lemma *minWait-le-pos*: $\llbracket \text{fair } rs; r \in R; \text{enabled } r \ s \rrbracket \implies \text{minWait } rs \ s \leq \text{pos } rs \ r$
(proof)

lemma *stake-pos-minWait*:

assumes $rs: \text{fair } rs$ **and** $m: \text{minWait } rs \ s < \text{pos } rs \ r$ **and** $r: r \in R$ **and** $s: s \in S$
shows $\text{pos } (\text{stl } (\text{trim } rs \ s)) \ r = \text{pos } rs \ r - \text{Suc } (\text{minWait } rs \ s)$
(proof)

lemma *ipath-mkTree-ev*:
assumes $s: s \in S$ **and** rs : fair rs
and i : *ipath* (*mkTree* rs s) *steps* **and** $r: r \in R$
and alw : alw (*holds* (*enabledAtStep* r)) *steps*
shows ev (*holds* (*takenAtStep* r)) *steps*
 $\langle proof \rangle$

4 Persistent rules

definition

$per\ r \equiv$
 $\forall s\ r1\ sl'\ s'. s \in S \wedge enabled\ r\ s \wedge r1 \in R - \{r\} \wedge eff\ r1\ s\ sl' \wedge s' \in | sl' \longrightarrow$
 $enabled\ r\ s'$

lemma *per-alw*:

assumes p : *per* r **and** e : *epath* *steps* \wedge *fst* (*shd* *steps*) $\in S$
shows alw (*holds* (*enabledAtStep* r) *impl*
(*holds* (*takenAtStep* r) *or* *next* (*holds* (*enabledAtStep* r)))) *steps*
 $\langle proof \rangle$

end — context *RuleSystem*

locale *PersistentRuleSystem* = *RuleSystem* *eff* *rules* S
for $eff :: 'rule \Rightarrow 'state \Rightarrow 'state\ fset \Rightarrow bool$ **and** $rules :: 'rule\ stream$ **and** $S +$
assumes $per: \bigwedge r. r \in R \implies per\ r$
begin

lemma *ipath-mkTree-saturated*:

assumes $s: s \in S$ **and** rs : fair rs
and i : *ipath* (*mkTree* rs s) *steps* **and** $r: r \in R$
shows *saturated* r *steps*
 $\langle proof \rangle$

theorem *ipath-mkTree-Saturated*:

assumes $s \in S$ **and** fair rs **and** *ipath* (*mkTree* rs s) *steps*
shows *Saturated* *steps*
 $\langle proof \rangle$

theorem *epath-completeness-Saturated*:

assumes $s \in S$
shows
 $(\exists t. fst\ (root\ t) = s \wedge wf\ t \wedge tfinite\ t) \vee$
 $(\exists steps. fst\ (shd\ steps) = s \wedge epath\ steps \wedge Saturated\ steps)$ (**is** $?A \vee ?B$)
 $\langle proof \rangle$

end — context *PersistentRuleSystem*

5 Code generation

```
locale RuleSystem-Code =
  fixes eff' :: 'rule  $\Rightarrow$  'state  $\Rightarrow$  'state fset option
  and rules :: 'rule stream — countably many rules
  begin

  definition eff r s sl  $\equiv$  eff' r s = Some sl

  end

  definition [code del]: effG eff' r s sl  $\equiv$  RuleSystem-Code.eff eff' r s sl

  sublocale RuleSystem-Code < RuleSystem-Defs
    where eff = effG eff' and rules = rules <proof>

  context RuleSystem-Code
  begin

  lemma enabled-eff': enabled r s  $\longleftrightarrow$  eff' r s  $\neq$  None
  <proof>

  lemma pickEff-the[code]: pickEff r s = the (eff' r s)
  <proof>

  lemmas [code-unfold] = trim-def enabled-eff' pickEff-the

  <ML>
  interpretation i: RuleSystem-Code eff' rules for eff' and rules <proof>
  declare [[lc-delete RuleSystem-Defs.mkTree (effG ?eff')]]
  declare [[lc-delete RuleSystem-Defs.trim]]
  declare [[lc-delete RuleSystem-Defs.enabled]]
  declare [[lc-delete RuleSystem-Defs.pickEff]]
  declare [[lc-add RuleSystem-Defs.mkTree (effG ?eff') i.mkTree-unfold]]
  <ML>

  code-printing
  constant the  $\rightarrow$  (Haskell) fromJust
  | constant Option.is-none  $\rightarrow$  (Haskell) isNothing

  export-code mkTree-effG-uu in Haskell module-name Tree
```

6 Toy instantiation: Propositional Logic

```
datatype fmla = Atom nat | Neg fmla | Conj fmla fmla

primrec max-depth where
  max-depth (Atom _) = 0
```

| $\text{max-depth } (\text{Neg } \varphi) = \text{Suc } (\text{max-depth } \varphi)$
| $\text{max-depth } (\text{Conj } \varphi \ \psi) = \text{Suc } (\text{max } (\text{max-depth } \varphi) (\text{max-depth } \psi))$

lemma *max-depth-0*: $\text{max-depth } \varphi = 0 = (\exists n. \varphi = \text{Atom } n)$
⟨*proof*⟩

lemma *max-depth-Suc*: $\text{max-depth } \varphi = \text{Suc } n = ((\exists \psi. \varphi = \text{Neg } \psi \wedge \text{max-depth } \psi = n) \vee$
 $(\exists \psi1 \ \psi2. \varphi = \text{Conj } \psi1 \ \psi2 \wedge \text{max } (\text{max-depth } \psi1) (\text{max-depth } \psi2) = n))$
⟨*proof*⟩

abbreviation *atoms* $\equiv \text{smap } \text{Atom } \text{nats}$

abbreviation *depth1* \equiv

$\text{sinterleave } (\text{smap } \text{Neg } \text{atoms}) (\text{smap } (\text{case-prod } \text{Conj}) (\text{sproduct } \text{atoms } \text{atoms}))$

abbreviation *sinterleaves* $\equiv \text{fold } \text{sinterleave}$

fun *extendLevel* **where** *extendLevel* (*belowN*, *N*) =
(*let* *Next* = *sinterleaves*
(*map* (*smap* (*case-prod* *Conj*)) [*sproduct* *belowN* *N*, *sproduct* *N* *belowN*, *sproduct* *N* *N*])
(*smap* *Neg* *N*)
in (*sinterleave* *belowN* *N*, *Next*))

lemma *extendLevel-step*:

$\llbracket \text{sset } \text{belowN} = \{\varphi. \text{max-depth } \varphi < n\};$
 $\text{sset } N = \{\varphi. \text{max-depth } \varphi = n\}; \text{st} = (\text{belowN}, N) \rrbracket \implies$
 $\exists \text{belowNext } \text{Next}. \text{extendLevel } \text{st} = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset } \text{belowNext} = \{\varphi. \text{max-depth } \varphi < \text{Suc } n\} \wedge \text{sset } \text{Next} = \{\varphi. \text{max-depth } \varphi = \text{Suc } n\}$
⟨*proof*⟩

lemma *sset-atoms*: $\text{sset } \text{atoms} = \{\varphi. \text{max-depth } \varphi < 1\}$
⟨*proof*⟩

lemma *sset-depth1*: $\text{sset } \text{depth1} = \{\varphi. \text{max-depth } \varphi = 1\}$
⟨*proof*⟩

lemma *extendLevel-Nsteps*:

$\llbracket \text{sset } \text{belowN} = \{\varphi. \text{max-depth } \varphi < n\}; \text{sset } N = \{\varphi. \text{max-depth } \varphi = n\} \rrbracket \implies$
 $\exists \text{belowNext } \text{Next}. (\text{extendLevel } \widetilde{\widetilde{m}}) (\text{belowN}, N) = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset } \text{belowNext} = \{\varphi. \text{max-depth } \varphi < n + m\} \wedge \text{sset } \text{Next} = \{\varphi. \text{max-depth } \varphi = n + m\}$
⟨*proof*⟩

corollary *extendLevel*:

$\exists \text{belowNext } \text{Next}. (\text{extendLevel } \widetilde{\widetilde{m}}) (\text{atoms}, \text{depth1}) = (\text{belowNext}, \text{Next}) \wedge$
 $\text{sset } \text{belowNext} = \{\varphi. \text{max-depth } \varphi < 1 + m\} \wedge \text{sset } \text{Next} = \{\varphi. \text{max-depth } \varphi = 1 + m\}$

<proof>

definition $fmlas = sinterleave\ atoms\ (smerge\ (smap\ snd\ (siterate\ extendLevel\ (atoms,\ depth1))))$

lemma $fmlas\text{-}UNIV: sset\ fmlas = (UNIV :: fmla\ set)$
<proof>

datatype $rule = Idle \mid Ax\ nat \mid NegL\ fmla \mid NegR\ fmla \mid ConjL\ fmla\ fmla \mid ConjR\ fmla\ fmla$

abbreviation $mkRules\ f \equiv smap\ f\ fmlas$

abbreviation $mkRulePairs\ f \equiv smap\ (case\text{-}prod\ f)\ (sproduct\ fmlas\ fmlas)$

definition $rules\ where$

$rules = Idle\ \#\#\ sinterleaves\ [mkRules\ NegL,\ mkRules\ NegR,\ mkRulePairs\ ConjL,\ mkRulePairs\ ConjR]$
 $(smap\ Ax\ nats)$

lemma $rules\text{-}UNIV: sset\ rules = (UNIV :: rule\ set)$
<proof>

type-synonym $state = fmla\ fset * fmla\ fset$

fun $eff' :: rule \Rightarrow state \Rightarrow state\ fset\ option\ where$

$eff'\ Idle\ (\Gamma,\ \Delta) = Some\ \{(\Gamma,\ \Delta)\}$
 $| eff'\ (Ax\ n)\ (\Gamma,\ \Delta) =$
 $\quad (if\ Atom\ n\ |\in|\ \Gamma \wedge Atom\ n\ |\in|\ \Delta\ then\ Some\ \{\}\ else\ None)$
 $| eff'\ (NegL\ \varphi)\ (\Gamma,\ \Delta) =$
 $\quad (if\ Neg\ \varphi\ |\in|\ \Gamma\ then\ Some\ \{(\Gamma\ |-|\ \{\ |\ Neg\ \varphi\ |\},\ finsert\ \varphi\ \Delta)\}\ else\ None)$
 $| eff'\ (NegR\ \varphi)\ (\Gamma,\ \Delta) =$
 $\quad (if\ Neg\ \varphi\ |\in|\ \Delta\ then\ Some\ \{(\finsert\ \varphi\ \Gamma,\ \Delta\ |-|\ \{\ |\ Neg\ \varphi\ |\})\}\ else\ None)$
 $| eff'\ (ConjL\ \varphi\ \psi)\ (\Gamma,\ \Delta) =$
 $\quad (if\ Conj\ \varphi\ \psi\ |\in|\ \Gamma$
 $\quad\ then\ Some\ \{(\finsert\ \varphi\ (\finsert\ \psi\ (\Gamma\ |-|\ \{\ |\ Conj\ \varphi\ \psi\ |\})),\ \Delta)\}$
 $\quad\ else\ None)$
 $| eff'\ (ConjR\ \varphi\ \psi)\ (\Gamma,\ \Delta) =$
 $\quad (if\ Conj\ \varphi\ \psi\ |\in|\ \Delta$
 $\quad\ then\ Some\ \{(\Gamma,\ finsert\ \varphi\ (\Delta\ |-|\ \{\ |\ Conj\ \varphi\ \psi\ |\})),\ (\Gamma,\ finsert\ \psi\ (\Delta\ |-|\ \{\ |\ Conj$
 $\varphi\ \psi\ |\}))\}$
 $\quad\ else\ None)$

abbreviation $Disj\ \varphi\ \psi \equiv Neg\ (Conj\ (Neg\ \varphi)\ (Neg\ \psi))$

abbreviation $Imp\ \varphi\ \psi \equiv Disj\ (Neg\ \varphi)\ \psi$

abbreviation $Iff\ \varphi\ \psi \equiv Conj\ (Imp\ \varphi\ \psi)\ (Imp\ \psi\ \varphi)$

definition *thm1* \equiv ($\{|Conj (Atom\ 0) (Neg (Atom\ 0))|\}$, $\{|\}$)

declare *Stream.smember-code* [*code del*]

lemma [*code*]: *Stream.smember* *x* (*y* $\#\#$ *s*) = (*x* = *y* \vee *Stream.smember* *x* *s*)
<proof>

interpretation *RuleSystem* $\lambda r\ s\ ss.$ *eff'* *r s* = *Some ss rules UNIV*
<proof>

interpretation *PersistentRuleSystem* $\lambda r\ s\ ss.$ *eff'* *r s* = *Some ss rules UNIV*
<proof>

definition *rho* \equiv *i.fenum rules*

definition *propTree* \equiv *i.mkTree eff' rho*

export-code *propTree thm1* **in** *Haskell* **module-name** *PropInstance*