Abstract Completeness

Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel

March 17, 2025

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

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

inductive *tfinite* **where**
tfinite: $(\bigwedge t'.\ t' \mid\in\mid cont\ t \Longrightarrow tfinite\ t') \Longrightarrow tfinite\ t$

coinductive *ipath* **where**
ipath: $[\![root\ t = shd\ steps;\ t' \mid\in\mid cont\ t;\ ipath\ t'\ (stl\ steps)]\!] \Longrightarrow ipath\ t\ steps$
⟨*proof*⟩
primcorec *konig* **where**
$shd\ (konig\ t) = root\ t$
$\mid stl\ (konig\ t) = konig\ (SOME\ t'.\ t' \mid\in\mid cont\ t \wedge \neg\ tfinite\ t')$

lemma *Konig*: $\neg\ tfinite\ t \Longrightarrow ipath\ t\ (konig\ t)$
⟨*proof*⟩

2 Rule Systems

type-synonym $('state, 'rule)$ *step* $= 'state \times 'rule$**type-synonym** $('state, 'rule)$ *dtree* $= ('state, 'rule)$ *step tree*

locale *RuleSystem-Defs* =
fixes *eff* $:: 'rule \Rightarrow 'state \Rightarrow 'state\ fset \Rightarrow bool$

and *rules* $:: 'rule\ stream$
begin

abbreviation $R \equiv sset\ rules$

lemma *countable-R*: *countable R* ⟨*proof*⟩
lemma *NE-R*: $R \neq \{\}$ ⟨*proof*⟩

definition $enabled\ r\ s \equiv \exists\ sl.\ eff\ r\ s\ sl$
definition $pickEff\ r\ s \equiv if\ enabled\ r\ s\ then\ (SOME\ sl.\ eff\ r\ s\ sl)\ else\ the\ None$

lemma *pickEff*: $enabled\ r\ s \Longrightarrow eff\ r\ s\ (pickEff\ r\ s)$
⟨*proof*⟩

abbreviation $effStep\ step \equiv eff\ (snd\ step)\ (fst\ step)$
abbreviation $enabledAtStep\ r\ step \equiv enabled\ r\ (fst\ step)$
abbreviation $takenAtStep\ r\ step \equiv snd\ step = r$

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

definition $saturated\ r \equiv alw\ (holds\ (enabledAtStep\ r)\ impl\ ev\ (holds\ (takenAtStep\ r)))$
definition $Saturated\ steps \equiv \forall\ r \in R.\ saturated\ r\ steps$
coinductive *wf* **where**

wf: $[\![$*snd* (*root t*) $\in$ *R*; *effStep* (*root t*) (*fimage* (*fst o root*) (*cont t*)); $\bigwedge t'.\ t'$ |$\in$| *cont t* $\Longrightarrow$ *wf t'*$]\!]$ $\Longrightarrow$ *wf t*

coinductive *epath* **where**
epath: $[\![$*snd* (*shd steps*) $\in$ *R*; *fst* (*shd* (*stl steps*)) |$\in$| *sl*; *effStep* (*shd steps*) *sl*; *epath* (*stl steps*)$]\!]$ $\Longrightarrow$ *epath steps*

lemma *wf-ipath-epath*:
assumes *wf t ipath t steps*
shows *epath steps*
$\langle$*proof*$\rangle$

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

lemma *fair-stl*: *fair rs* $\Longrightarrow$ *fair* (*stl rs*)
$\langle$*proof*$\rangle$

lemma *sdrop-fair*: *fair rs* $\Longrightarrow$ *fair* (*sdrop m rs*)
$\langle$*proof*$\rangle$

3 A Fair Enumeration of the Rules

definition *fenum* $\equiv$ *flat* (*smap* (λ*n*. *stake n rules*) (*fromN 1*))

lemma *sset-fenum*: *sset fenum* = *R*
$\langle$*proof*$\rangle$

lemma *fair-fenum*: *fair fenum*
$\langle$*proof*$\rangle$

definition *trim rs s* = *sdrop-while* (λ*r*. *Not* (*enabled r s*)) *rs*

primcorec *mkTree* **where**
root (*mkTree rs s*) = (*s*, (*shd* (*trim rs s*)))
| *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*)))
$\langle$*proof*$\rangle$

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'*. $[\![$*s* $\in$ *S*; *r* $\in$ *R*; *eff r s sl*; *s'* |$\in$| *sl*$]\!]$ $\Longrightarrow$ *s'* $\in$ *S*
and *enabled-R*: $\bigwedge$ *s*. *s* $\in$ *S* $\Longrightarrow$ $\exists$ *r* $\in$ *R*. $\exists$ *sl*. *eff r s sl*
begin
definition *minWait rs s* $\equiv$ *LEAST n*. *enabled* (*shd* (*sdrop n rs*)) *s*

lemma *trim-alt*:
assumes *s*: $s \in S$ **and** *rs*: *fair rs*
shows *trim rs s* = *sdrop* (*minWait rs s*) *rs*
⟨*proof*⟩

lemma *minWait-ex*:
assumes *s*: $s \in S$ **and** *rs*: *fair rs*
shows ∃ *n*. *enabled* (*shd* (*sdrop n rs*)) *s*
⟨*proof*⟩

lemma assumes $s \in S$ **and** *fair rs*
shows *trim-in-R*: *shd* (*trim rs s*) ∈ *R*
and *trim-enabled*: *enabled* (*shd* (*trim rs s*)) *s*
and *trim-fair*: *fair* (*trim rs s*)
⟨*proof*⟩

lemma *minWait-least*: ⟦*enabled* (*shd* (*sdrop n rs*)) *s*⟧ ⟹ *minWait rs s* ≤ *n*
⟨*proof*⟩

lemma *in-cont-mkTree*:
assumes *s*: $s \in S$ **and** *rs*: *fair rs* **and** *t'*: *t'* |∈| *cont* (*mkTree rs s*)
shows ∃ *sl' s'*. *s'* ∈ *S* ∧ *eff* (*shd* (*trim rs s*)) *s sl'* ∧
s' |∈| *sl'* ∧ *t'* = *mkTree* (*stl* (*trim rs s*)) *s'*
⟨*proof*⟩

lemma *ipath-mkTree-sdrop*:
assumes *s*: $s \in S$ **and** *rs*: *fair rs* **and** *i*: *ipath* (*mkTree rs s*) *steps*
shows ∃ *n s'*. *s'* ∈ *S* ∧ *ipath* (*mkTree* (*sdrop n rs*) *s'*) (*sdrop m steps*)
⟨*proof*⟩

lemma *wf-mkTree*:
assumes *s*: $s \in S$ **and** *fair rs*
shows *wf* (*mkTree rs s*)
⟨*proof*⟩**definition** *pos rs r* ≡ *LEAST n*. *shd* (*sdrop n rs*) = *r*

lemma *pos*: ⟦*fair rs*; *r* ∈ *R*⟧ ⟹ *shd* (*sdrop* (*pos rs r*) *rs*) = *r*
⟨*proof*⟩

lemma *pos-least*: *shd* (*sdrop n rs*) = *r* ⟹ *pos rs r* ≤ *n*
⟨*proof*⟩

lemma *minWait-le-pos*: ⟦*fair rs*; *r* ∈ *R*; *enabled r s*⟧ ⟹ *minWait rs s* ≤ *pos rs r*
⟨*proof*⟩

lemma *stake-pos-minWait*:
assumes *rs*: *fair rs* **and** *m*: *minWait rs s* < *pos rs r* **and** *r*: *r* ∈ *R* **and** *s*: *s* ∈ *S*
shows *pos* (*stl* (*trim rs s*)) *r* = *pos rs r* − *Suc* (*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*
⟨*proof*⟩

4 Persistent rules

definition
  per r ≡
    $\forall\, s\ r1\ sl'\ s'.\ s \in S \wedge \mathit{enabled}\ r\ s \wedge r1 \in R - \{r\} \wedge \mathit{eff}\ r1\ s\ sl' \wedge s' \mathrel{|\in|} sl' \longrightarrow$
enabled r s'

lemma *per-alw*:
  assumes *p*: *per r* **and** *e*: *epath steps* ∧ *fst* (*shd steps*) ∈ *S*
  shows *alw* (*holds* (*enabledAtStep r*) *impl*
    (*holds* (*takenAtStep r*) *or nxt* (*holds* (*enabledAtStep r*)))) *steps*
⟨*proof*⟩

end — context RuleSystem

locale *PersistentRuleSystem* = *RuleSystem eff rules S*
for *eff* :: *'rule* ⇒ *'state* ⇒ *'state fset* ⇒ *bool* **and** *rules* :: *'rule stream* **and** *S* +
assumes *per*: $\bigwedge r.\ r \in R \Longrightarrow \mathit{per}\ r$
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*
⟨*proof*⟩

theorem *ipath-mkTree-Saturated*:
  assumes $s \in S$ **and** *fair rs* **and** *ipath* (*mkTree rs s*) *steps*
  shows *Saturated steps*
  ⟨*proof*⟩

theorem *epath-completeness-Saturated*:
  assumes $s \in S$
  shows
  (∃ *t*. *fst* (*root t*) = *s* ∧ *wf t* ∧ *tfinite t*) ∨
   (∃ *steps*. *fst* (*shd steps*) = *s* ∧ *epath steps* ∧ *Saturated steps*) (**is** *?A* ∨ *?B*)
⟨*proof*⟩

end — context PersistentRuleSystem

5 Code generation

locale *RuleSystem-Code* =
fixes *eff'* :: *'rule* ⇒ *'state* ⇒ *'state fset option*
and *rules* :: *'rule stream* — countably many rules
begin

definition *eff r s sl* ≡ *eff' r s* = *Some sl*

end

definition [*code del*]: *effG eff' r s sl* ≡ *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* ⟷ *eff' r s* ≠ *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* ⇀ (*Haskell*) *fromJust*
| **constant** *Option.is-none* ⇀ (*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*

| *max-depth* (*Neg* φ) = *Suc* (*max-depth* φ)
| *max-depth* (*Conj* φ ψ) = *Suc* (*max* (*max-depth* φ) (*max-depth* ψ))

lemma *max-depth-0*: *max-depth* φ = *0* = (∃ *n*. φ = *Atom n*)
⟨*proof*⟩

lemma *max-depth-Suc*: *max-depth* φ = *Suc n* = ((∃ ψ. φ = *Neg* ψ ∧ *max-depth* ψ = *n*) ∨
(∃ ψ*1* ψ*2*. φ = *Conj* ψ*1* ψ*2* ∧ *max* (*max-depth* ψ*1*) (*max-depth* ψ*2*) = *n*))
⟨*proof*⟩

abbreviation *atoms* ≡ *smap Atom nats*
abbreviation *depth1* ≡
sinterleave (*smap Neg atoms*) (*smap* (*case-prod Conj*) (*sproduct atoms atoms*))

abbreviation *sinterleaves* ≡ *fold 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*:
⟦*sset belowN* = {φ. *max-depth* φ < *n*};
sset N = {φ. *max-depth* φ = *n*}; *st* = (*belowN*, *N*)⟧ ⟹
∃ *belowNext Next*. *extendLevel st* = (*belowNext*, *Next*) ∧
sset belowNext = {φ. *max-depth* φ < *Suc n*} ∧ *sset Next* = {φ. *max-depth* φ = *Suc n*}
⟨*proof*⟩

lemma *sset-atoms*: *sset atoms* = {φ. *max-depth* φ < *1*}
⟨*proof*⟩

lemma *sset-depth1*: *sset depth1* = {φ. *max-depth* φ = *1*}
⟨*proof*⟩

lemma *extendLevel-Nsteps*:
⟦*sset belowN* = {φ. *max-depth* φ < *n*}; *sset N* = {φ. *max-depth* φ = *n*}⟧ ⟹
∃ *belowNext Next*. (*extendLevel* ^^ *m*) (*belowN*, *N*) = (*belowNext*, *Next*) ∧
sset belowNext = {φ. *max-depth* φ < *n* + *m*} ∧ *sset Next* = {φ. *max-depth* φ = *n* + *m*}
⟨*proof*⟩

corollary *extendLevel*:
∃ *belowNext Next*. (*extendLevel* ^^ *m*) (*atoms*, *depth1*) = (*belowNext*, *Next*) ∧
sset belowNext = {φ. *max-depth* φ < *1* + *m*} ∧ *sset Next* = {φ. *max-depth* φ = *1* + *m*}

⟨*proof*⟩

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

lemma *fmlas-UNIV*: *sset fmlas* = (*UNIV* :: *fmla set*)
⟨*proof*⟩

datatype *rule* = *Idle* | *Ax nat* | *NegL fmla* | *NegR fmla* | *ConjL fmla fmla* | *ConjR fmla fmla*

abbreviation *mkRules f* ≡ *smap f fmlas*
abbreviation *mkRulePairs f* ≡ *smap* (*case-prod f*) (*sproduct fmlas fmlas*)

definition *rules* **where**
rules = *Idle* ##
sinterleaves [*mkRules NegL*, *mkRules NegR*, *mkRulePairs ConjL*, *mkRulePairs ConjR*]
(*smap Ax nats*)

lemma *rules-UNIV*: *sset rules* = (*UNIV* :: *rule set*)
⟨*proof*⟩

type-synonym *state* = *fmla fset* ∗ *fmla fset*

fun *eff′* :: *rule* ⇒ *state* ⇒ *state fset option* **where**
eff′ Idle (Γ, Δ) = *Some* {|(Γ, Δ)|}
| *eff′* (*Ax n*) (Γ, Δ) =
(*if Atom n* |∈| Γ ∧ *Atom n* |∈| Δ *then Some* {||} *else None*)
| *eff′* (*NegL* φ) (Γ, Δ) =
(*if Neg* φ |∈| Γ *then Some* {|(Γ |−| {| *Neg* φ |}, *finsert* φ Δ)|} *else None*)
| *eff′* (*NegR* φ) (Γ, Δ) =
(*if Neg* φ |∈| Δ *then Some* {|(*finsert* φ Γ, Δ |−| {| *Neg* φ |})|} *else None*)
| *eff′* (*ConjL* φ ψ) (Γ, Δ) =
(*if Conj* φ ψ |∈| Γ
then Some {|(*finsert* φ (*finsert* ψ (Γ |−| {| *Conj* φ ψ |})), Δ)|}
else None)
| *eff′* (*ConjR* φ ψ) (Γ, Δ) =
(*if Conj* φ ψ |∈| Δ
then Some {|(Γ, *finsert* φ (Δ |−| {| *Conj* φ ψ |})), (Γ, *finsert* ψ (Δ |−| {| *Conj* φ ψ |}))|}
else None)

abbreviation *Disj* φ ψ ≡ *Neg* (*Conj* (*Neg* φ) (*Neg* ψ))
abbreviation *Imp* φ ψ ≡ *Disj* (*Neg* φ) ψ
abbreviation *Iff* φ ψ ≡ *Conj* (*Imp* φ ψ) (*Imp* ψ φ)

definition *thm1* ≡ ({|*Conj* (*Atom 0*) (*Neg* (*Atom 0*))|}, {||})

declare *Stream.smember-code* [*code del*]
lemma [*code*]: *Stream.smember x* (*y* ## *s*) = (*x* = *y* ∨ *Stream.smember x s*)
⟨*proof*⟩

interpretation *RuleSystem* λ*r s ss. eff′ r s* = *Some ss rules UNIV*
⟨*proof*⟩

interpretation *PersistentRuleSystem* λ*r s ss. eff′ r s* = *Some ss rules UNIV*
⟨*proof*⟩

definition *rho* ≡ *i.fenum rules*
definition *propTree* ≡ *i.mkTree eff′ rho*

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