A Sequent Calculus for First-Order Logic

Asta Halkjær From

March 17, 2025

Abstract

This work formalizes soundness and completeness of a one-sided sequent calculus for first-order logic. The completeness is shown via a translation from a complete semantic tableau calculus, the proof of which is based on the First-Order Logic According to Fitting theory. The calculi and proof techniques are taken from Ben-Ari's Mathematical Logic for Computer Science [1].

Contents

1 Common Notation

theory *Common* **imports** *FOL−Fitting.FOL-Fitting* **begin**

notation *FF* (‹⊥›)
notation *TT* (‹⊤›)

end

2 Tableau Calculus

theory *Tableau* **imports** *Common* **begin**

inductive *TC* :: ‹('*a*, '*b*) *form list* ⇒ *bool*› (‹⊣ -› *0*) **where**
Basic: ‹⊣ *Pred i l* # *Neg* (*Pred i l*) # *G*›
| *BasicFF*: ‹⊣ ⊥ # *G*›
| *BasicNegTT*: ‹⊣ *Neg* ⊤ # *G*›
| *AlphaNegNeg*: ‹⊣ *A* # *G* ⟹ ⊣ *Neg* (*Neg A*) # *G*›
| *AlphaAnd*: ‹⊣ *A* # *B* # *G* ⟹ ⊣ *And A B* # *G*›
| *AlphaNegOr*: ‹⊣ *Neg A* # *Neg B* # *G* ⟹ ⊣ *Neg* (*Or A B*) # *G*›
| *AlphaNegImpl*: ‹⊣ *A* # *Neg B* # *G* ⟹ ⊣ *Neg* (*Impl A B*) # *G*›
| *BetaNegAnd*: ‹⊣ *Neg A* # *G* ⟹ ⊣ *Neg B* # *G* ⟹ ⊣ *Neg* (*And A B*) # *G*›
| *BetaOr*: ‹⊣ *A* # *G* ⟹ ⊣ *B* # *G* ⟹ ⊣ *Or A B* # *G*›
| *BetaImpl*: ‹⊣ *Neg A* # *G* ⟹ ⊣ *B* # *G* ⟹ ⊣ *Impl A B* # *G*›
| *GammaForall*: ‹⊣ *subst A t 0* # *G* ⟹ ⊣ *Forall A* # *G*›
| *GammaNegExists*: ‹⊣ *Neg* (*subst A t 0*) # *G* ⟹ ⊣ *Neg* (*Exists A*) # *G*›
| *DeltaExists*: ‹⊣ *subst A* (*App n* []) *0* # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊣ *Exists A* # *G*›
| *DeltaNegForall*: ‹⊣ *Neg* (*subst A* (*App n* []) *0*) # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊣ *Neg* (*Forall A*) # *G*›
| *Order*: ‹⊣ *G* ⟹ *set G* = *set G'* ⟹ ⊣ *G'*›

lemma *Shift*: ‹⊣ *rotate1 G* ⟹ ⊣ *G*›
⟨*proof*⟩

lemma *Swap*: ‹⊣ *B* # *A* # *G* ⟹ ⊣ *A* # *B* # *G*›
⟨*proof*⟩

definition *tableauproof* :: ‹('*a*, '*b*) *form list* ⇒ ('*a*, '*b*) *form* ⇒ *bool*› **where**
‹*tableauproof ps p* ≡ (⊣ *Neg p* # *ps*)›

theorem *tableauNotAA*: ‹⊣ [*Neg* (*Pred ''A''* []), *Pred ''A''* []]›
⟨*proof*⟩

theorem *AndAnd*:
‹⊣ [*And* (*Pred ''A''* []) (*Pred ''B''* []), *Neg* (*And* (*Pred ''B''* []) (*Pred ''A''* []))]›
⟨*proof*⟩

2.1 Soundness

lemma *TC-soundness*:
‹⊣ $G \Longrightarrow \exists p \in set\ G.\ \neg\ eval\ e\ f\ g\ p$›
⟨*proof*⟩

theorem *tableau-soundness*:
‹*tableauproof ps p* $\Longrightarrow$ *list-all* (*eval e f g*) *ps* $\Longrightarrow$ *eval e f g p*›
⟨*proof*⟩

2.2 Completeness for Closed Formulas

theorem *infinite-nonempty*: ‹*infinite A* $\Longrightarrow \exists x.\ x \in A$›
⟨*proof*⟩

theorem *TCd-consistency*:
assumes *inf-param*: ‹*infinite* (*UNIV*::'*a set*)›
shows ‹*consistency* {*S*::('*a*, '*b*) *form set*. $\exists G.\ S = set\ G \wedge \neg\ (\dashv G)$}›
⟨*proof*⟩

theorem *tableau-completeness'*:
fixes *p* :: ‹(*nat*, *nat*) *form*›
assumes ‹*closed 0 p*›
and ‹*list-all* (*closed 0*) *ps*›
and *mod*: ‹$\forall$ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) *ps* ⟶ *eval e f g p*›
shows ‹*tableauproof ps p*›
⟨*proof*⟩

2.3 Open Formulas

lemma *TC-psubst*:
fixes *f* :: ‹'*a* ⇒ '*a*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
shows ‹⊣ $G \Longrightarrow \dashv$ *map* (*psubst f*) *G*›
⟨*proof*⟩

lemma *subcs-map*: ‹*subcs c s G* = *map* (*subc c s*) *G*›
⟨*proof*⟩

lemma *TC-subcs*:
fixes *G* :: ‹('*a*, '*b*) *form list*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
shows ‹⊣ $G \Longrightarrow \dashv$ *subcs c s G*›
⟨*proof*⟩

lemma *TC-map-subc*:
fixes *G* :: ‹('*a*, '*b*) *form list*›
assumes *inf-params*: ‹*infinite* (*UNIV* :: '*a set*)›
shows ‹⊣ $G \Longrightarrow \dashv$ *map* (*subc c s*) *G*›
⟨*proof*⟩

lemma *ex-all-closed*: ‹$\exists m.$ *list-all* (*closed m*) *G*›
⟨*proof*⟩

primrec *sub-consts* :: ‹'*a list* ⇒ ('*a*, '*b*) *form* ⇒ ('*a*, '*b*) *form*› **where**
  ‹*sub-consts* [] *p* = *p*›
| ‹*sub-consts* (*c* # *cs*) *p* = *sub-consts cs* (*subst p* (*App c* []) (*length cs*))›

lemma *valid-sub-consts*:
  assumes ‹$\forall$ (*e* :: *nat* ⇒ '*a*) *f g*. *eval e f g p*›
  shows ‹*eval* (*e* :: *nat* => '*a*) *f g* (*sub-consts cs p*)›
  ⟨*proof*⟩

lemma *closed-sub'* [*simp*]:
  assumes ‹*k* ≤ *m*› **shows**
    ‹*closedt* (*Suc m*) *t* = *closedt m* (*substt t* (*App c* []) *k*)›
    ‹*closedts* (*Suc m*) *l* = *closedts m* (*substs l* (*App c* []) *k*)›
  ⟨*proof*⟩

lemma *closed-sub*: ‹*k* ≤ *m* ⟹ *closed* (*Suc m*) *p* = *closed m* (*subst p* (*App c* [])
k)›
  ⟨*proof*⟩

lemma *closed-sub-consts*: ‹*length cs* = *k* ⟹ *closed m* (*sub-consts cs p*) = *closed*
(*m* + *k*) *p*›
⟨*proof*⟩

lemma *map-sub-consts-Nil*: ‹*map* (*sub-consts* []) *G* = *G*›
  ⟨*proof*⟩

primrec *conjoin* :: ‹('*a*, '*b*) *form list* ⇒ ('*a*, '*b*) *form*› **where**
  ‹*conjoin* [] = *Neg* ⊥›
| ‹*conjoin* (*p* # *ps*) = *And p* (*conjoin ps*)›

lemma *eval-conjoin*: ‹*list-all* (*eval e f g*) *G* = *eval e f g* (*conjoin G*)›
  ⟨*proof*⟩

lemma *valid-sub*:
  fixes *e* :: ‹*nat* ⇒ '*a*›
  assumes ‹$\forall$ (*e* :: *nat* ⇒ '*a*) *f g*. *eval e f g p* ⟶ *eval e f g q*›
  shows ‹*eval e f g* (*subst p t m*) ⟶ *eval e f g* (*subst q t m*)›
  ⟨*proof*⟩

lemma *eval-sub-consts*:
  fixes *e* :: ‹*nat* ⇒ '*a*›
  assumes ‹$\forall$ (*e* :: *nat* ⇒ '*a*) *f g*. *eval e f g p* ⟶ *eval e f g q*›
    and ‹*eval e f g* (*sub-consts cs p*)›
  shows ‹*eval e f g* (*sub-consts cs q*)›
  ⟨*proof*⟩

lemma *sub-consts-And* [*simp*]: ‹*sub-consts cs* (*And p q*) = *And* (*sub-consts cs p*) (*sub-consts cs q*)›
⟨*proof*⟩

lemma *sub-consts-conjoin*:
‹*eval e f g* (*sub-consts cs* (*conjoin G*)) = *eval e f g* (*conjoin* (*map* (*sub-consts cs*) *G*))›
⟨*proof*⟩

lemma *all-sub-consts-conjoin*:
‹*list-all* (*eval e f g*) (*map* (*sub-consts cs*) *G*) = *eval e f g* (*sub-consts cs* (*conjoin G*))›
⟨*proof*⟩

lemma *valid-all-sub-consts*:
fixes *e* :: ‹*nat* ⇒ *'a*›
assumes ‹∀(*e* :: *nat* ⇒ *'a*) *f g*. *list-all* (*eval e f g*) *G* ⟶ *eval e f g p*›
shows ‹*list-all* (*eval e f g*) (*map* (*sub-consts cs*) *G*) ⟶ *eval e f g* (*sub-consts cs p*)›
⟨*proof*⟩

lemma *TC-vars-for-consts*:
fixes *G* :: ‹(*'a*, *'b*) *form list*›
assumes ‹*infinite* (*UNIV* :: *'a set*)›
shows ‹⊣ *G* ⟹ ⊣ *map* (λ*p*. *vars-for-consts p cs*) *G*›
⟨*proof*⟩

lemma *vars-for-consts-sub-consts*:
‹*closed* (*length cs*) *p* ⟹ *list-all* (λ*c*. *new c p*) *cs* ⟹ *distinct cs* ⟹
vars-for-consts (*sub-consts cs p*) *cs* = *p*›
⟨*proof*⟩

lemma *all-vars-for-consts-sub-consts*:
‹*list-all* (*closed* (*length cs*)) *G* ⟹ *list-all* (λ*c*. *list-all* (*new c*) *G*) *cs* ⟹ *distinct cs* ⟹
map (λ*p*. *vars-for-consts p cs*) (*map* (*sub-consts cs*) *G*) = *G*›
⟨*proof*⟩

lemma *new-conjoin*: ‹*new c* (*conjoin G*) ⟹ *list-all* (*new c*) *G*›
⟨*proof*⟩

lemma *all-fresh-constants*:
fixes *G* :: ‹(*'a*, *'b*) *form list*›
assumes ‹*infinite* (*UNIV* :: *'a set*)›
shows ‹∃ *cs*. *length cs* = *m* ∧ *list-all* (λ*c*. *list-all* (*new c*) *G*) *cs* ∧ *distinct cs*›
⟨*proof*⟩

lemma *sub-consts-Neg*: ‹*sub-consts cs* (*Neg p*) = *Neg* (*sub-consts cs p*)›

⟨*proof*⟩

2.4 Completeness

theorem *tableau-completeness*:
fixes G :: ‹(*nat*, *nat*) *form list*›
assumes ‹∀(*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) *G* ⟶ *eval e f g p*›
shows ‹*tableauproof G p*›
⟨*proof*⟩

corollary
fixes p :: ‹(*nat*, *nat*) *form*›
assumes ‹∀(*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g p*›
shows ‹⊣ [*Neg p*]›
⟨*proof*⟩

end

3 Sequent Calculus

theory *Sequent* **imports** *Tableau* **begin**

inductive *SC* :: ‹('*a*, '*b*) *form list* ⇒ *bool*› (‹⊢ -› *0*) **where**
Basic: ‹⊢ *Pred i l* # *Neg* (*Pred i l*) # *G*›
| *BasicNegFF*: ‹⊢ *Neg* ⊥ # *G*›
| *BasicTT*: ‹⊢ ⊤ # *G*›
| *AlphaNegNeg*: ‹⊢ *A* # *G* ⟹ ⊢ *Neg* (*Neg A*) # *G*›
| *AlphaNegAnd*: ‹⊢ *Neg A* # *Neg B* # *G* ⟹ ⊢ *Neg* (*And A B*) # *G*›
| *AlphaOr*: ‹⊢ *A* # *B* # *G* ⟹ ⊢ *Or A B* # *G*›
| *AlphaImpl*: ‹⊢ *Neg A* # *B* # *G* ⟹ ⊢ *Impl A B* # *G*›
| *BetaAnd*: ‹⊢ *A* # *G* ⟹ ⊢ *B* # *G* ⟹ ⊢ *And A B* # *G*›
| *BetaNegOr*: ‹⊢ *Neg A* # *G* ⟹ ⊢ *Neg B* # *G* ⟹ ⊢ *Neg* (*Or A B*) # *G*›
| *BetaNegImpl*: ‹⊢ *A* # *G* ⟹ ⊢ *Neg B* # *G* ⟹ ⊢ *Neg* (*Impl A B*) # *G*›
| *GammaExists*: ‹⊢ *subst A t 0* # *G* ⟹ ⊢ *Exists A* # *G*›
| *GammaNegForall*: ‹⊢ *Neg* (*subst A t 0*) # *G* ⟹ ⊢ *Neg* (*Forall A*) # *G*›
| *DeltaForall*: ‹⊢ *subst A* (*App n* []) *0* # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊢ *Forall A* # *G*›
| *DeltaNegExists*: ‹⊢ *Neg* (*subst A* (*App n* []) *0*) # *G* ⟹ *news n* (*A* # *G*) ⟹ ⊢ *Neg* (*Exists A*) # *G*›
| *Order*: ‹⊢ *G* ⟹ *set G* = *set G'* ⟹ ⊢ *G'*›

lemma *Shift*: ‹⊢ *rotate1 G* ⟹ ⊢ *G*›
⟨*proof*⟩

lemma *Swap*: ‹⊢ *B* # *A* # *G* ⟹ ⊢ *A* # *B* # *G*›
⟨*proof*⟩

lemma ‹⊢ [*Neg* (*Pred ''A''* []), *Pred ''A''* []]›
⟨*proof*⟩

lemma ‹⊢ [*And* (*Pred* ''*A*'' []) (*Pred* ''*B*'' []), *Neg* (*And* (*Pred* ''*B*'' []) (*Pred* ''*A*'' []))]›
⟨*proof*⟩

3.1 Soundness

lemma *SC-soundness*: ‹⊢ *G* ⟹ ∃ *p* ∈ *set G*. *eval e f g p*›
⟨*proof*⟩

3.2 Tableau Calculus Equivalence

fun *compl* :: ‹('*a*, '*b*) *form* ⇒ ('*a*, '*b*) *form*› **where**
‹*compl* (*Neg p*) = *p*›
| ‹*compl p* = *Neg p*›

lemma *compl*: ‹*compl p* = *Neg p* ∨ (∃ *q*. *compl p* = *q* ∧ *p* = *Neg q*)›
⟨*proof*⟩

lemma *new-compl*: ‹*new n p* ⟹ *new n* (*compl p*)›
⟨*proof*⟩

lemma *news-compl*: ‹*news n G* ⟹ *news n* (*map compl G*)›
⟨*proof*⟩

theorem *TC-SC*: ‹⊣ *G* ⟹ ⊢ *map compl G*›
⟨*proof*⟩

3.3 Completeness

theorem *SC-completeness*:
fixes *p* :: ‹(*nat*, *nat*) *form*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-all* (*eval e f g*) *ps* ⟶ *eval e f g p*›
shows ‹⊢ *p* # *map compl ps*›
⟨*proof*⟩

corollary
fixes *p* :: ‹(*nat*, *nat*) *form*›
assumes ‹∀ (*e* :: *nat* ⇒ *nat hterm*) *f g*. *eval e f g p*›
shows ‹⊢ [*p*]›
⟨*proof*⟩

end
theory *Sequent2* **imports** *Sequent* **begin**

4 Completeness Revisited

lemma ‹∃ *p*. *q* = *compl p*›
⟨*proof*⟩

definition *compl′* **where**
‹*compl′* = (λ*q*. (*SOME p*. *q* = *compl p*))›

lemma *comp′-sem*:
‹*eval e f g* (*compl′ p*) ⟷ ¬ *eval e f g p*›
⟨*proof*⟩

lemma *comp′-sem-list*: ‹*list-ex* (λ*p*. ¬ *eval e f g p*) (*map compl′ ps*) ⟷ *list-ex* (*eval e f g*) *ps*›
⟨*proof*⟩

theorem *SC-completeness′*:
fixes *ps* :: ‹(*nat*, *nat*) *form list*›
assumes ‹∀(*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-ex* (*eval e f g*) (*p* # *ps*)›
shows ‹⊢ *p* # *ps*›
⟨*proof*⟩

corollary
fixes *ps* :: ‹(*nat*, *nat*) *form list*›
assumes ‹∀(*e* :: *nat* ⇒ *nat hterm*) *f g*. *list-ex* (*eval e f g*) *ps*›
shows ‹⊢ *ps*›
⟨*proof*⟩

end

References

[1] M. Ben-Ari. *Mathematical Logic for Computer Science, 3rd Edition.* Springer, 2012.