A Naive Prover for First-Order Logic

Asta Halkjær From

March 17, 2025

Abstract

The AFP entry Abstract Completeness by Blanchette, Popescu and Traytel [1] formalizes the core of Beth/Hintikka-style completeness proofs for first-order logic and can be used to formalize executable sequent calculus provers. In the Journal of Automated Reasoning [2], the authors instantiate the framework with a sequent calculus for first-order logic and prove its completeness. Their use of an infinite set of proof rules indexed by formulas yields very direct arguments. A fair stream of these rules controls the prover, making its definition remarkably simple. The AFP entry, however, only contains a toy example for propositional logic. The AFP entry A Sequent Calculus Prover for First-Order Logic with Functions by From and Jacobsen [3] also uses the framework, but uses a finite set of generic rules resulting in a more sophisticated prover with more complicated proofs.

This entry contains an executable sequent calculus prover for first-order logic with functions in the style presented by Blanchette et al. The prover can be exported to Haskell and this entry includes formalized proofs of its soundness and completeness. The proofs are simpler than those for the prover by From and Jacobsen [3] but the performance of the prover is significantly worse.

The included theory *Fair-Stream* first proves that the sequence of natural numbers 0, 0, 1, 0, 1, 2, etc. is fair. It then proves that mapping any surjective function across the sequence preserves fairness. This method of obtaining a fair stream of rules is similar to the one given by Blanchette et al. [2]. The concrete functions from natural numbers to terms, formulas and rules are defined using the *Nat-Bijection* theory in the HOL-Library.

Contents

1 List Syntax

theory *List-Syntax* **imports** *Main* **begin**

abbreviation *list-member-syntax* :: ‹'*a* ⇒ '*a list* ⇒ *bool*› (‹- [∈] -› [*51*, *51*] *50*)
where
‹*x* [∈] *A* ≡ *x* ∈ *set A*›

abbreviation *list-not-member-syntax* :: ‹'*a* ⇒ '*a list* ⇒ *bool*› (‹- [∉] -› [*51*, *51*] *50*) **where**
‹*x* [∉] *A* ≡ *x* ∉ *set A*›

abbreviation *list-subset-syntax* :: ‹'*a list* ⇒ '*a list* ⇒ *bool*› (‹- [⊂] -› [*51*, *51*] *50*)
where
‹*A* [⊂] *B* ≡ *set A* ⊂ *set B*›

abbreviation *list-subset-eq-syntax* :: ‹'*a list* ⇒ '*a list* ⇒ *bool*› (‹- [⊆] -› [*51*, *51*] *50*) **where**
‹*A* [⊆] *B* ≡ *set A* ⊆ *set B*›

abbreviation *removeAll-syntax* :: ‹'*a list* ⇒ '*a* ⇒ '*a list*› (**infix** ‹[÷]› *75*) **where**
‹*A* [÷] *x* ≡ *removeAll x A*›

syntax (*ASCII*)
-BallList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3ALL* (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)
-BexList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3EX* (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1List :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3EX*! (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)
-BleastList :: ‹*id* ⇒ '*a list* ⇒ *bool* ⇒ '*a*› (‹(*3LEAST* (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)

syntax (*input*)
-BallList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3*! (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)
-BexList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3?* (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1List :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3?*! (-/[:]-)./ -)› [*0*, *0*, *10*] *10*)

syntax
-BallList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3*∀(-/[∈]-)./ -)› [*0*, *0*, *10*] *10*)
-BexList :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3*∃(-/[∈]-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1List :: ‹*pttrn* ⇒ '*a list* ⇒ *bool* ⇒ *bool*› (‹(*3*∃!(-/[∈]-)./ -)› [*0*, *0*, *10*] *10*)
-BleastList :: ‹*id* ⇒ '*a list* ⇒ *bool* ⇒ '*a*› (‹(*3LEAST*(-/[∈]-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-BallList ⇌ *Ball* **and**
-BexList ⇌ *Bex* **and**
-Bex1List ⇌ *Ex1* **and**
-BleastList ⇌ *Least*

translations
∀ *x*[∈]*A*. *P* ⇌ *CONST Ball* (*CONST set A*) (*λx*. *P*)
∃ *x*[∈]*A*. *P* ⇌ *CONST Bex* (*CONST set A*) (*λx*. *P*)
∃ !*x*[∈]*A*. *P* ⇀ ∃ !*x*. *x* [∈] *A* ∧ *P*
LEAST x[:]*A*. *P* ⇀ *LEAST x*. *x* [∈] *A* ∧ *P*

syntax (*ASCII* **output**)
-setlessAllList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3ALL* -[<]-./ -)› [*0*, *0*, *10*] *10*)
-setlessExList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3EX* -[<]-./ -)› [*0*, *0*, *10*] *10*)
-setleAllList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3ALL* -[<=]-./ -)› [*0*, *0*, *10*] *10*)
-setleExList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3EX* -[<=]-./ -)› [*0*, *0*, *10*] *10*)
-setleEx1List :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3EX*! -[<=]-./ -)› [*0*, *0*, *10*] *10*)

syntax
-setlessAllList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3*∀ -[⊂]-./ -)› [*0*, *0*, *10*] *10*)
-setlessExList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3*∃ -[⊂]-./ -)› [*0*, *0*, *10*] *10*)
-setleAllList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3*∀ -[⊆]-./ -)› [*0*, *0*, *10*] *10*)
-setleExList :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3*∃ -[⊆]-./ -)› [*0*, *0*, *10*] *10*)
-setleEx1List :: ‹[*idt*, ′*a*, *bool*] ⇒ *bool*› (‹(*3*∃ !-[⊆]-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-setlessAllList -setleAllList ⇌ *All* **and**
-setlessExList -setleExList ⇌ *Ex* **and**
-setleEx1List ⇌ *Ex1*

translations
∀ *A*[⊂]*B*. *P* ⇀ ∀ *A*. *A* [⊂] *B* ⟶ *P*
∃ *A*[⊂]*B*. *P* ⇀ ∃ *A*. *A* [⊂] *B* ∧ *P*
∀ *A*[⊆]*B*. *P* ⇀ ∀ *A*. *A* [⊆] *B* ⟶ *P*
∃ *A*[⊆]*B*. *P* ⇀ ∃ *A*. *A* [⊆] *B* ∧ *P*
∃ !*A*[⊆]*B*. *P* ⇀ ∃ !*A*. *A* [⊆] *B* ∧ *P*

end

2 Fair Streams

theory *Fair-Stream* **imports** *HOL−Library.Stream* **begin**

definition *upt-lists* :: ‹*nat list stream*› **where**
‹*upt-lists* ≡ *smap* (*upt 0*) (*stl nats*)›

definition *fair-nats* :: ‹*nat stream*› **where**

‹*fair-nats* ≡ *flat upt-lists*›

definition *fair* :: ‹′*a stream* ⇒ *bool*› **where**
‹*fair s* ≡ ∀ *x* ∈ *sset s*. ∀ *m*. ∃ *n* ≥ *m*. *s* !! *n* = *x*›

lemma *upt-lists-snth*: ‹*x* ≤ *n* ⟹ *x* ∈ *set* (*upt-lists* !! *n*)›
⟨*proof*⟩

lemma *all-ex-upt-lists*: ‹∃ *n* ≥ *m*. *x* ∈ *set* (*upt-lists* !! *n*)›
⟨*proof*⟩

lemma *upt-lists-ne*: ‹∀ *xs* ∈ *sset upt-lists*. *xs* ≠ []›
⟨*proof*⟩

lemma *sset-flat-stl*: ‹*sset* (*flat* (*stl s*)) ⊆ *sset* (*flat s*)›
⟨*proof*⟩

lemma *flat-snth-nth*:
assumes ‹*x* = *s* !! *n* ! *m*› ‹*m* < *length* (*s* !! *n*)› ‹∀ *xs* ∈ *sset s*. *xs* ≠ []›
shows ‹∃ *n*′ ≥ *n*. *x* = *flat s* !! *n*′›
⟨*proof*⟩

lemma *all-ex-fair-nats*: ‹∃ *n* ≥ *m*. *fair-nats* !! *n* = *x*›
⟨*proof*⟩

lemma *fair-surj*:
assumes ‹*surj f*›
shows ‹*fair* (*smap f fair-nats*)›
⟨*proof*⟩

definition *fair-stream* :: ‹(*nat* ⇒ ′*a*) ⇒ ′*a stream*› **where**
‹*fair-stream f* ≡ *smap f fair-nats*›

theorem *fair-stream*: ‹*surj f* ⟹ *fair* (*fair-stream f*)›
⟨*proof*⟩

theorem *UNIV-stream*: ‹*surj f* ⟹ *sset* (*fair-stream f*) = *UNIV*›
⟨*proof*⟩

end

3 Syntax

theory *Syntax* **imports** *List-Syntax* **begin**

3.1 Terms and Formulas

datatype *tm*
= *Var nat* (‹#›)

| *Fun nat ‹tm list›* (‹†›)

datatype *fm*
= *Falsity* (‹⊥›)
| *Pre nat ‹tm list›* (‹‡›)
| *Imp fm fm* (**infixr** ‹⟶› *55*)
| *Uni fm* (‹∀›)

type-synonym *sequent* = ‹*fm list* × *fm list*›

3.1.1 Substitution

primrec *add-env* :: ‹*'a* ⇒ (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a*› (**infix** ‹⨾› *0*) **where**
‹(*t* ⨾ *s*) *0* = *t*›
| ‹(*t* ⨾ *s*) (*Suc n*) = *s n*›

primrec *lift-tm* :: ‹*tm* ⇒ *tm*› **where**
‹*lift-tm* (#*n*) = #(*n*+*1*)›
| ‹*lift-tm* (†*f ts*) = †*f* (*map lift-tm ts*)›

primrec *sub-tm* :: ‹(*nat* ⇒ *tm*) ⇒ *tm* ⇒ *tm*› **where**
‹*sub-tm s* (#*n*) = *s n*›
| ‹*sub-tm s* (†*f ts*) = †*f* (*map* (*sub-tm s*) *ts*)›

primrec *sub-fm* :: ‹(*nat* ⇒ *tm*) ⇒ *fm* ⇒ *fm*› **where**
‹*sub-fm* - ⊥ = ⊥›
| ‹*sub-fm s* (‡*P ts*) = ‡*P* (*map* (*sub-tm s*) *ts*)›
| ‹*sub-fm s* (*p* ⟶ *q*) = *sub-fm s p* ⟶ *sub-fm s q*›
| ‹*sub-fm s* (∀*p*) = ∀(*sub-fm* (#*0* ⨾ λ*n*. *lift-tm* (*s n*)) *p*)›

abbreviation *inst-single* :: ‹*tm* ⇒ *fm* ⇒ *fm*› (‹⟨-⟩›) **where**
‹⟨*t*⟩ ≡ *sub-fm* (*t* ⨾ #)›

3.1.2 Variables

primrec *vars-tm* :: ‹*tm* ⇒ *nat list*› **where**
‹*vars-tm* (#*n*) = [*n*]›
| ‹*vars-tm* (†- *ts*) = *concat* (*map vars-tm ts*)›

primrec *vars-fm* :: ‹*fm* ⇒ *nat list*› **where**
‹*vars-fm* ⊥ = []›
| ‹*vars-fm* (‡- *ts*) = *concat* (*map vars-tm ts*)›
| ‹*vars-fm* (*p* ⟶ *q*) = *vars-fm p* @ *vars-fm q*›
| ‹*vars-fm* (∀*p*) = *vars-fm p*›

primrec *max-list* :: ‹*nat list* ⇒ *nat*› **where**
‹*max-list* [] = *0*›
| ‹*max-list* (*x* # *xs*) = *max x* (*max-list xs*)›

lemma *max-list-append*: ‹*max-list* (*xs* @ *ys*) = *max* (*max-list xs*) (*max-list ys*)›

⟨*proof*⟩

lemma *max-list-concat*: ‹xs [∈] $xss \Longrightarrow$ *max-list* $xs \leq$ *max-list* (*concat* xss)›
⟨*proof*⟩

lemma *max-list-in*: ‹*max-list* $xs < n \Longrightarrow n$ [∉] xs›
⟨*proof*⟩

definition *vars-fms* :: ‹*fm list* ⇒ *nat list*› **where**
‹*vars-fms* $A \equiv$ *concat* (*map vars-fm* A)›

lemma *vars-fms-member*: ‹p [∈] $A \Longrightarrow$ *vars-fm* p [⊆] *vars-fms* A›
⟨*proof*⟩

lemma *max-list-mono*: ‹A [⊆] $B \Longrightarrow$ *max-list* $A \leq$ *max-list* B›
⟨*proof*⟩

lemma *max-list-vars-fms*:
assumes ‹*max-list* (*vars-fms* A) $\leq n$› ‹p [∈] A›
shows ‹*max-list* (*vars-fm* p) $\leq n$›
⟨*proof*⟩

definition *fresh* :: ‹*fm list* ⇒ *nat*› **where**
‹*fresh* $A \equiv$ *Suc* (*max-list* (*vars-fms* A))›

3.2 Rules

datatype *rule* =
Idle | *Axiom nat* ‹*tm list*› | *FlsL* | *FlsR* | *ImpL fm fm* | *ImpR fm fm* | *UniL tm fm* | *UniR fm*

end

4 Semantics

theory *Semantics* **imports** *Syntax* **begin**

4.1 Definition

type-synonym ′*a var-denot* = ‹*nat* ⇒ ′*a*›
type-synonym ′*a fun-denot* = ‹*nat* ⇒ ′*a list* ⇒ ′*a*›
type-synonym ′*a pre-denot* = ‹*nat* ⇒ ′*a list* ⇒ *bool*›

primrec *semantics-tm* :: ‹′*a var-denot* ⇒ ′*a fun-denot* ⇒ *tm* ⇒ ′*a*› (‹(⦇-, -⦈)›)
where
‹⦇E, F⦈ (#n) = E n›
| ‹⦇E, F⦈ (†f ts) = F f (*map* ⦇E, F⦈ ts)›

primrec *semantics-fm* :: ‹′a var-denot ⇒ ′a fun-denot ⇒ ′a pre-denot ⇒ fm ⇒ bool›
  (‹⟦-, -, -⟧›) **where**
  ‹⟦-, -, -⟧ ⊥ = False›
| ‹⟦E, F, G⟧ (‡P ts) = G P (map ⦇E, F⦈ ts)›
| ‹⟦E, F, G⟧ (p ⟶ q) = (⟦E, F, G⟧ p ⟶ ⟦E, F, G⟧ q)›
| ‹⟦E, F, G⟧ (∀ p) = (∀ x. ⟦x ⨟ E, F, G⟧ p)›

fun *sc* :: ‹(′a var-denot × ′a fun-denot × ′a pre-denot) ⇒ sequent ⇒ bool› **where**
  ‹sc (E, F, G) (A, B) = ((∀ p [∈] A. ⟦E, F, G⟧ p) ⟶ (∃ q [∈] B. ⟦E, F, G⟧ q))›

4.2 Substitution

lemma *add-env-semantics* [*simp*]: ‹⦇E, F⦈ ((t ⨟ s) n) = (⦇E, F⦈ t ⨟ λm. ⦇E, F⦈ (s m)) n›
  ⟨*proof*⟩

lemma *lift-lemma* [*simp*]: ‹⦇x ⨟ E, F⦈ (lift-tm t) = ⦇E, F⦈ t›
  ⟨*proof*⟩

lemma *sub-tm-semantics* [*simp*]: ‹⦇E, F⦈ (sub-tm s t) = ⦇λn. ⦇E, F⦈ (s n), F⦈ t›
  ⟨*proof*⟩

lemma *sub-fm-semantics* [*simp*]: ‹⟦E, F, G⟧ (sub-fm s p) = ⟦λn. ⦇E, F⦈ (s n), F, G⟧ p›
  ⟨*proof*⟩

4.3 Variables

lemma *upd-vars-tm* [*simp*]: ‹n [∉] vars-tm t ⟹ ⦇E(n := x), F⦈ t = ⦇E, F⦈ t›
  ⟨*proof*⟩

lemma *add-upd-commute* [*simp*]: ‹(y ⨟ E(n := x)) m = ((y ⨟ E)(Suc n := x)) m›
  ⟨*proof*⟩

lemma *upd-vars-fm* [*simp*]: ‹max-list (vars-fm p) < n ⟹ ⟦E(n := x), F, G⟧ p = ⟦E, F, G⟧ p›
⟨*proof*⟩

end

5 Encoding

theory *Encoding* **imports** *HOL−Library.Nat-Bijection Syntax* **begin**

abbreviation *infix-sum-encode* (**infixr** ‹$› *100*) **where**
  ‹c $ x ≡ sum-encode (c x)›

lemma *lt-sum-encode-Inr*: ‹n < Inr $ n›

⟨*proof*⟩

lemma *sum-prod-decode-lt* [*simp*]: ‹*sum-decode n* = *Inr b* ⟹ (*x*, *y*) = *prod-decode b* ⟹ *y* < *Suc n*›
⟨*proof*⟩

lemma *sum-prod-decode-lt-Suc* [*simp*]:
‹*sum-decode n* = *Inr b* ⟹ (*Suc x*, *y*) = *prod-decode b* ⟹ *x* < *Suc n*›
⟨*proof*⟩

lemma *lt-list-encode*: ‹*n* [∈] *ns* ⟹ *n* < *list-encode ns*›
⟨*proof*⟩

lemma *prod-Suc-list-decode-lt* [*simp*]:
‹(*x*, *Suc y*) = *prod-decode n* ⟹ *y'* [∈] (*list-decode y*) ⟹ *y'* < *n*›
⟨*proof*⟩

5.1 Terms

primrec *nat-of-tm* :: ‹*tm* ⇒ *nat*› **where**
‹*nat-of-tm* (#*n*) = *prod-encode* (*n*, *0*)›
| ‹*nat-of-tm* (†*f ts*) = *prod-encode* (*f*, *Suc* (*list-encode* (*map nat-of-tm ts*)))›

function *tm-of-nat* :: ‹*nat* ⇒ *tm*› **where**
‹*tm-of-nat n* = (*case prod-decode n of*
(*n*, *0*) ⇒ #*n*
| (*f*, *Suc ts*) ⇒ †*f* (*map tm-of-nat* (*list-decode ts*)))›
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *tm-nat*: ‹*tm-of-nat* (*nat-of-tm t*) = *t*›
⟨*proof*⟩

lemma *surj-tm-of-nat*: ‹*surj tm-of-nat*›
⟨*proof*⟩

5.2 Formulas

primrec *nat-of-fm* :: ‹*fm* ⇒ *nat*› **where**
‹*nat-of-fm* ⊥ = *0*›
| ‹*nat-of-fm* (‡*P ts*) = *Suc* (*Inl* $ *prod-encode* (*P*, *list-encode* (*map nat-of-tm ts*)))›
| ‹*nat-of-fm* (*p* ⟶ *q*) = *Suc* (*Inr* $ *prod-encode* (*Suc* (*nat-of-fm p*), *nat-of-fm q*))›
| ‹*nat-of-fm* (∀ *p*) = *Suc* (*Inr* $ *prod-encode* (*0*, *nat-of-fm p*))›

function *fm-of-nat* :: ‹*nat* ⇒ *fm*› **where**
‹*fm-of-nat 0* = ⊥›
| ‹*fm-of-nat* (*Suc n*) = (*case sum-decode n of*
Inl n ⇒ *let* (*P*, *ts*) = *prod-decode n in* ‡*P* (*map tm-of-nat* (*list-decode ts*))
| *Inr n* ⇒ (*case prod-decode n of*
(*Suc p*, *q*) ⇒ *fm-of-nat p* ⟶ *fm-of-nat q*

```
  | (0, p) ⇒ ∀ (fm-of-nat p)))›
  ⟨proof⟩
termination ⟨proof⟩

lemma fm-nat: ‹fm-of-nat (nat-of-fm p) = p›
  ⟨proof⟩

lemma surj-fm-of-nat: ‹surj fm-of-nat›
  ⟨proof⟩
```

5.3 Rules

Pick a large number to help encode the Idle rule, so that we never hit it in practice.

```
definition idle-nat :: nat where
  ‹idle-nat ≡ 4294967295›

primrec nat-of-rule :: ‹rule ⇒ nat› where
  ‹nat-of-rule Idle = Inl $ prod-encode (0, idle-nat)›
| ‹nat-of-rule (Axiom n ts) = Inl $ prod-encode (Suc n, Suc (list-encode (map nat-of-tm ts)))›
| ‹nat-of-rule FlsL = Inl $ prod-encode (0, 0)›
| ‹nat-of-rule FlsR = Inl $ prod-encode (0, Suc 0)›
| ‹nat-of-rule (ImpL p q) = Inr $ prod-encode (Inl $ nat-of-fm p, Inl $ nat-of-fm q)›
| ‹nat-of-rule (ImpR p q) = Inr $ prod-encode (Inr $ nat-of-fm p, nat-of-fm q)›
| ‹nat-of-rule (UniL t p) = Inr $ prod-encode (Inl $ nat-of-tm t, Inr $ nat-of-fm p)›
| ‹nat-of-rule (UniR p) = Inl $ prod-encode (Suc (nat-of-fm p), 0)›

fun rule-of-nat :: ‹nat ⇒ rule› where
  ‹rule-of-nat n = (case sum-decode n of
    Inl n ⇒ (case prod-decode n of
      (0, 0) ⇒ FlsL
    | (0, Suc 0) ⇒ FlsR
    | (0, n2) ⇒ if n2 = idle-nat then Idle else
      let (p, q) = prod-decode n2 in ImpR (fm-of-nat p) (fm-of-nat q)
    | (Suc n, Suc ts) ⇒ Axiom n (map tm-of-nat (list-decode ts))
    | (Suc p, 0) ⇒ UniR (fm-of-nat p))
  | Inr n ⇒ (let (n1, n2) = prod-decode n in
    case sum-decode n1 of
      Inl n1 ⇒ (case sum-decode n2 of
        Inl q ⇒ ImpL (fm-of-nat n1) (fm-of-nat q)
      | Inr p ⇒ UniL (tm-of-nat n1) (fm-of-nat p))
    | Inr p ⇒ ImpR (fm-of-nat p) (fm-of-nat n2)))›

lemma rule-nat: ‹rule-of-nat (nat-of-rule r) = r›
  ⟨proof⟩
```

lemma *surj-rule-of-nat*: ‹*surj rule-of-nat*›
⟨*proof*⟩

end

6 Prover

theory *Prover* **imports** *Abstract-Completeness.Abstract-Completeness Encoding Fair-Stream* **begin**

function *eff* :: ‹*rule* ⇒ *sequent* ⇒ (*sequent fset*) *option*› **where**
‹*eff Idle* (*A*, *B*) = *Some* {| (*A*, *B*) |}›
| ‹*eff* (*Axiom P ts*) (*A*, *B*) = (*if* ‡*P ts* [∈] *A* ∧ ‡*P ts* [∈] *B then Some* {||} *else None*)›
| ‹*eff FlsL* (*A*, *B*) = (*if* ⊥ [∈] *A then Some* {||} *else None*)›
| ‹*eff FlsR* (*A*, *B*) = (*if* ⊥ [∈] *B then Some* {| (*A*, *B* [÷] ⊥) |} *else None*)›
| ‹*eff* (*ImpL p q*) (*A*, *B*) = (*if* (*p* ⟶ *q*) [∈] *A then*
Some {| (*A* [÷] (*p* ⟶ *q*), *p* # *B*), (*q* # *A* [÷] (*p* ⟶ *q*), *B*) |} *else None*)›
| ‹*eff* (*ImpR p q*) (*A*, *B*) = (*if* (*p* ⟶ *q*) [∈] *B then*
Some {| (*p* # *A*, *q* # *B* [÷] (*p* ⟶ *q*)) |} *else None*)›
| ‹*eff* (*UniL t p*) (*A*, *B*) = (*if* ∀ *p* [∈] *A then Some* {| (⟨*t*⟩*p* # *A*, *B*) |} *else None*)›
| ‹*eff* (*UniR p*) (*A*, *B*) = (*if* ∀ *p* [∈] *B then*
Some {| (*A*, ⟨#(*fresh* (*A* @ *B*))⟩*p* # *B* [÷] ∀ *p*) |} *else None*)›
⟨*proof*⟩
termination ⟨*proof*⟩

definition *rules* :: ‹*rule stream*› **where**
‹*rules* ≡ *fair-stream rule-of-nat*›

lemma *UNIV-rules*: ‹*sset rules* = *UNIV*›
⟨*proof*⟩

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

lemma *per-rules′*:
assumes ‹*enabled r* (*A*, *B*)› ‹¬ *enabled r* (*A′*, *B′*)› ‹*eff r′* (*A*, *B*) = *Some ss′*› ‹(*A′*, *B′*) |∈| *ss′*›
shows ‹*r′* = *r*›
⟨*proof*⟩

lemma *per-rules*: ‹*per r*›
⟨*proof*⟩

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

definition ‹*prover* ≡ *mkTree rules*›

end

7 Export

theory *Export* **imports** *Prover* **begin**

definition ‹*prove-sequent* ≡ *i.mkTree eff rules*›
definition ‹*prove* ≡ *λp. prove-sequent* ([], [*p*])›

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

code-printing
constant *the* ⇀ (*Haskell*) (*x* −> *case x of* { *Just y* −> *y* })
| **constant** *Option.is-none* ⇀ (*Haskell*) (*x* −> *case x of* { *Just y* −> *False*; *Nothing* −> *True* })

code-identifier
code-module *Product-Type* ⇀ (*Haskell*) *Arith*
| **code-module** *Orderings* ⇀ (*Haskell*) *Arith*
| **code-module** *Arith* ⇀ (*Haskell*) *Prover*
| **code-module** *MaybeExt* ⇀ (*Haskell*) *Prover*
| **code-module** *List* ⇀ (*Haskell*) *Prover*
| **code-module** *Nat-Bijection* ⇀ (*Haskell*) *Prover*
| **code-module** *Syntax* ⇀ (*Haskell*) *Prover*
| **code-module** *Encoding* ⇀ (*Haskell*) *Prover*
| **code-module** *HOL* ⇀ (*Haskell*) *Prover*
| **code-module** *Set* ⇀ (*Haskell*) *Prover*
| **code-module** *FSet* ⇀ (*Haskell*) *Prover*
| **code-module** *Stream* ⇀ (*Haskell*) *Prover*
| **code-module** *Fair-Stream* ⇀ (*Haskell*) *Prover*
| **code-module** *Sum-Type* ⇀ (*Haskell*) *Prover*
| **code-module** *Abstract-Completeness* ⇀ (*Haskell*) *Prover*
| **code-module** *Export* ⇀ (*Haskell*) *Prover*

export-code open *prove* **in** *Haskell*

To export the Haskell code run:

```
> isabelle build -e -D .
```

To compile the exported code run:

```
> ghc -O2 -i./program Main.hs
```

To prove a formula, supply it using raw constructor names, e.g.:

```
> ./Main "Imp (Pre 0 []) (Imp (Pre 1 []) (Pre 0 []))"
```

```
  |- (P) --> ((Q) --> (P))
  + ImpR on P and (Q) --> (P)
  P |- (Q) --> (P)
  + ImpR on Q and P
  Q, P |- P
 + Axiom on P
```

The output is pretty-printed.

end

8 Soundness

theory *Soundness* **imports** *Abstract-Soundness.Finite-Proof-Soundness Prover Semantics* **begin**

lemma *eff-sound*:
assumes ‹*eff r* (A, B) = *Some ss*› ‹$\forall A\ B.\ (A, B)$ |∈| *ss* $\longrightarrow$ $(\forall (E :: \text{-} \Rightarrow 'a).$ *sc* $(E, F, G)\ (A, B))$›
shows ‹*sc* $(E, F, G)\ (A, B)$›
⟨*proof*⟩

interpretation *Soundness* ‹$\lambda r\ s\ ss.$ *eff r s* = *Some ss*› *rules UNIV sc*
⟨*proof*⟩

theorem *prover-soundness*:
assumes ‹*tfinite t*› **and** ‹*wf t*›
shows ‹*sc* (E, F, G) (*fst* (*root t*))›
⟨*proof*⟩

end

9 Completeness

theory *Completeness* **imports** *Prover Semantics* **begin**

9.1 Hintikka Counter Model

locale *Hintikka* =
fixes $A\ B$ :: ‹*fm set*›
assumes
Basic: ‹‡P *ts* $\in A \Longrightarrow$ ‡P *ts* $\in B \Longrightarrow$ *False*› **and**
FlsA: ‹$\bot \notin A$› **and**
ImpA: ‹$p \longrightarrow q \in A \Longrightarrow p \in B \vee q \in A$› **and**
ImpB: ‹$p \longrightarrow q \in B \Longrightarrow p \in A \wedge q \in B$› **and**
UniA: ‹$\forall p \in A \Longrightarrow \forall t.\ \langle t\rangle p \in A$› **and**
UniB: ‹$\forall p \in B \Longrightarrow \exists t.\ \langle t\rangle p \in B$›

abbreviation ‹*M A* ≡ ⟦#, †, *λP ts*. ‡*P ts* ∈ *A*⟧›

lemma *id-tm* [*simp*]: ‹⦇#, †⦈ *t* = *t*›
⟨*proof*⟩

lemma *size-sub-fm* [*simp*]: ‹*size* (*sub-fm s p*) = *size p*›
⟨*proof*⟩

theorem *Hintikka-counter-model*:
assumes ‹*Hintikka A B*›
shows ‹(*p* ∈ *A* ⟶ *M A p*) ∧ (*p* ∈ *B* ⟶ ¬ *M A p*)›
⟨*proof*⟩

9.2 Escape Paths Form Hintikka Sets

lemma *sset-sdrop*: ‹*sset* (*sdrop n s*) ⊆ *sset s*›
⟨*proof*⟩

lemma *epath-sdrop*: ‹*epath steps* ⟹ *epath* (*sdrop n steps*)›
⟨*proof*⟩

lemma *eff-preserves-Pre*:
assumes ‹*effStep* ((*A*, *B*), *r*) *ss*› ‹(*A*′, *B*′) |∈| *ss*›
shows ‹(‡*P ts* [∈] *A* ⟹ ‡*P ts* [∈] *A*′)› ‹‡*P ts* [∈] *B* ⟹ ‡*P ts* [∈] *B*′›
⟨*proof*⟩

lemma *epath-eff*:
assumes ‹*epath steps*› ‹*effStep* (*shd steps*) *ss*›
shows ‹*fst* (*shd* (*stl steps*)) |∈| *ss*›
⟨*proof*⟩

abbreviation ‹*lhs s* ≡ *fst* (*fst s*)›
abbreviation ‹*rhs s* ≡ *snd* (*fst s*)›
abbreviation ‹*lhsd s* ≡ *lhs* (*shd s*)›
abbreviation ‹*rhsd s* ≡ *rhs* (*shd s*)›

lemma *epath-Pre-sdrop*:
assumes ‹*epath steps*› **shows**
‹‡*P ts* [∈] *lhs* (*shd steps*) ⟹ ‡*P ts* [∈] *lhsd* (*sdrop m steps*)›
‹‡*P ts* [∈] *rhs* (*shd steps*) ⟹ ‡*P ts* [∈] *rhsd* (*sdrop m steps*)›
⟨*proof*⟩

lemma *Saturated-sdrop*:
assumes ‹*Saturated steps*›
shows ‹*Saturated* (*sdrop n steps*)›
⟨*proof*⟩

definition *treeA* :: ‹(*sequent* × *rule*) *stream* ⇒ *fm set*› **where**
‹*treeA steps* ≡ ⋃*s* ∈ *sset steps*. *set* (*lhs s*)›

definition *treeB* :: ‹(*sequent* × *rule*) *stream* ⇒ *fm set*› **where**
‹*treeB steps* ≡ ⋃ *s* ∈ *sset steps*. *set* (*rhs s*)›

lemma *treeA-snth*: ‹*p* ∈ *treeA steps* ⟹ ∃ *n*. *p* [∈] *lhsd* (*sdrop n steps*)›
⟨*proof*⟩

lemma *treeB-snth*: ‹*p* ∈ *treeB steps* ⟹ ∃ *n*. *p* [∈] *rhsd* (*sdrop n steps*)›
⟨*proof*⟩

lemma *treeA-sdrop*: ‹*treeA* (*sdrop n steps*) ⊆ *treeA steps*›
⟨*proof*⟩

lemma *treeB-sdrop*: ‹*treeB* (*sdrop n steps*) ⊆ *treeB steps*›
⟨*proof*⟩

lemma *enabled-ex-taken*:
assumes ‹*epath steps*› ‹*Saturated steps*› ‹*enabled r* (*fst* (*shd steps*))›
shows ‹∃ *k*. *takenAtStep r* (*shd* (*sdrop k steps*))›
⟨*proof*⟩

lemma *Hintikka-epath*:
assumes ‹*epath steps*› ‹*Saturated steps*›
shows ‹*Hintikka* (*treeA steps*) (*treeB steps*)›
⟨*proof*⟩

9.3 Completeness

lemma *fair-stream-rules*: ‹*Fair-Stream.fair rules*›
⟨*proof*⟩

lemma *fair-rules*: ‹*fair rules*›
⟨*proof*⟩

lemma *epath-prover*:
fixes *A B* :: ‹*fm list*›
defines ‹*t* ≡ *prover* (*A*, *B*)›
shows ‹(*fst* (*root t*) = (*A*, *B*) ∧ *wf t* ∧ *tfinite t*) ∨
(∃ *steps*. *fst* (*shd steps*) = (*A*, *B*) ∧ *epath steps* ∧ *Saturated steps*)› (**is** ‹*?A* ∨ *?B*›)
⟨*proof*⟩

lemma *epath-countermodel*:
assumes ‹*fst* (*shd steps*) = (*A*, *B*)› ‹*epath steps*› ‹*Saturated steps*›
shows ‹∃ (*E* :: - ⇒ *tm*) *F G*. ¬ *sc* (*E*, *F*, *G*) (*A*, *B*)›
⟨*proof*⟩

theorem *prover-completeness*:
assumes ‹∀ (*E* :: - ⇒ *tm*) *F G*. *sc* (*E*, *F*, *G*) (*A*, *B*)›

defines ‹$t \equiv$ *prover* $(A,\ B)$›
shows ‹*fst* (*root* t) $= (A,\ B) \wedge$ *wf* $t \wedge$ *tfinite* t›
⟨*proof*⟩

corollary
assumes ‹$\forall (E :: - \Rightarrow tm)\ F\ G.\ [\![E,\ F,\ G]\!]\ p$›
defines ‹$t \equiv$ *prover* $([],\ [p])$›
shows ‹*fst* (*root* t) $= ([],\ [p]) \wedge$ *wf* $t \wedge$ *tfinite* t›
⟨*proof*⟩

end

10 Result

theory *Result* **imports** *Soundness Completeness* **begin**

theorem *prover-soundness-completeness*:
fixes $A\ B$ :: ‹*fm list*›
defines ‹$t \equiv$ *prover* $(A,\ B)$›
shows ‹*tfinite* $t \wedge$ *wf* $t \longleftrightarrow (\forall (E :: - \Rightarrow tm)\ F\ G.\ sc\ (E,\ F,\ G)\ (A,\ B))$›
⟨*proof*⟩

corollary
fixes p :: *fm*
defines ‹$t \equiv$ *prover* $([],\ [p])$›
shows ‹*tfinite* $t \wedge$ *wf* $t \longleftrightarrow (\forall (E :: - \Rightarrow tm)\ F\ G.\ [\![E,\ F,\ G]\!]\ p)$›
⟨*proof*⟩

end

References

[1] J. C. Blanchette, A. Popescu, and D. Traytel. Abstract completeness. *Archive of Formal Proofs*, Apr. 2014. https://isa-afp.org/entries/Abstract_Completeness.html, Formal proof development.

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

[3] A. H. From and F. K. Jacobsen. A sequent calculus prover for first-order logic with functions. *Archive of Formal Proofs*, Jan. 2022. https://isa-afp.org/entries/FOL_Seq_Calc2.html, Formal proof development.