A Naive Prover for First-Order Logic

Asta Halkjær From

May 26, 2024

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*)

translations
$\forall x[\in]A.\ P \rightleftharpoons CONST\ Ball\ (CONST\ set\ A)\ (\lambda x.\ P)$
$\exists x[\in]A.\ P \rightleftharpoons CONST\ Bex\ (CONST\ set\ A)\ (\lambda x.\ P)$
$\exists!x[\in]A.\ P \rightharpoonup \exists!x.\ x\ [\in]\ A \wedge P$
$LEAST\ x[:]A.\ P \rightharpoonup LEAST\ x.\ x\ [\in]\ A \wedge P$

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*)

translations
$\forall A[\subset]B.\ P \rightharpoonup \forall A.\ A\ [\subset]\ B \longrightarrow P$
$\exists A[\subset]B.\ P \rightharpoonup \exists A.\ A\ [\subset]\ B \wedge P$
$\forall A[\subseteq]B.\ P \rightharpoonup \forall A.\ A\ [\subseteq]\ B \longrightarrow P$
$\exists A[\subseteq]B.\ P \rightharpoonup \exists A.\ A\ [\subseteq]\ B \wedge P$
$\exists!A[\subseteq]B.\ P \rightharpoonup \exists!A.\ A\ [\subseteq]\ B \wedge P$

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*)›
unfolding *upt-lists-def* **by** *auto*

lemma *all-ex-upt-lists*: ‹∃ *n* ≥ *m*. *x* ∈ *set* (*upt-lists* !! *n*)›
using *upt-lists-snth* **by** (*meson dual-order.strict-trans1 gt-ex nle-le*)

```
lemma upt-lists-ne: ‹∀ xs ∈ sset upt-lists. xs ≠ []›
  unfolding upt-lists-def by (simp add: sset-range)

lemma sset-flat-stl: ‹sset (flat (stl s)) ⊆ sset (flat s)›
proof (cases s)
  case (SCons x xs)
  then show ?thesis
    by (cases x) (simp add: stl-sset subsetI, auto)
qed

lemma flat-snth-nth:
  assumes ‹x = s !! n ! m› ‹m < length (s !! n)› ‹∀ xs ∈ sset s. xs ≠ []›
  shows ‹∃ n' ≥ n. x = flat s !! n'›
  using assms
proof (induct n arbitrary: s)
  case 0
  then show ?case
    using flat-snth by fastforce
next
  case (Suc n)
  have ‹?case = (∃ n' ≥ n. x = flat s !! Suc n')›
    by (metis Suc-le-D Suc-le-mono)
  also have ‹... = (∃ n' ≥ n. x = stl (flat s) !! n')›
    by simp
  finally have ‹?case = (∃ n' ≥ n. x = (tl (shd s) @− flat (stl s)) !! n')›
    using Suc.prems flat-unfold by (simp add: shd-sset)
  then have ?case if ‹∃ n' ≥ n. x = flat (stl s) !! n'›
   using that by (metis (no-types, opaque-lifting) add.commute add-diff-cancel-left'
       dual-order.trans le-add2 shift-snth-ge)
  moreover {
    have ‹x = stl s !! n ! m› ‹ m < length (stl s !! n)›
      using Suc.prems by simp-all
    moreover have ‹∀ xs ∈ sset (stl s). xs ≠ []›
      using Suc.prems by (cases s) simp-all
    ultimately have ‹∃ n' ≥ n. x = flat (stl s) !! n'›
      using Suc.hyps by blast }
  ultimately show ?case .
qed

lemma all-ex-fair-nats: ‹∃ n ≥ m. fair-nats !! n = x›
proof −
  have ‹∃ n ≥ m. x ∈ set (upt-lists !! n)›
    using all-ex-upt-lists .
  then have ‹∃ n ≥ m. ∃ k < length (upt-lists !! n). upt-lists !! n ! k = x›
    by (simp add: in-set-conv-nth)
  then obtain n k where ‹m ≤ n› ‹k < length (upt-lists !! n)› ‹upt-lists !! n ! k
= x›
    by blast
  then obtain n' where ‹n ≤ n'› ‹x = flat upt-lists !! n'›
```

using *flat-snth-nth upt-lists-ne* **by** *metis*
moreover have ‹$m \leq n'$›
using ‹$m \leq n$› ‹$n \leq n'$› **by** *simp*
ultimately show *?thesis*
unfolding *fair-nats-def* **by** *blast*
qed

lemma *fair-surj*:
assumes ‹*surj f*›
shows ‹*fair* (*smap f fair-nats*)›
using *assms* **unfolding** *fair-def* **by** (*metis UNIV-I all-ex-fair-nats imageE snth-smap*)

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

theorem *fair-stream*: ‹*surj f* ⟹ *fair* (*fair-stream f*)›
unfolding *fair-stream-def* **using** *fair-surj* **.**

theorem *UNIV-stream*: ‹*surj f* ⟹ *sset* (*fair-stream f*) = *UNIV*›
unfolding *fair-stream-def* **using** *all-ex-fair-nats* **by** (*metis sset-range stream.set-map surjI*)

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 ⨟ λn. 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)›
  by (induct xs) auto

lemma max-list-concat: ‹xs [∈] xss ⟹ max-list xs ≤ max-list (concat xss)›
  by (induct xss) (auto simp: max-list-append)

lemma max-list-in: ‹max-list xs < n ⟹ n [∉] xs›
  by (induct xs) auto

definition vars-fms :: ‹fm list ⇒ nat list› where
  ‹vars-fms A ≡ concat (map vars-fm A)›

lemma vars-fms-member: ‹p [∈] A ⟹ vars-fm p [⊆] vars-fms A›
  unfolding vars-fms-def by (induct A) auto

lemma max-list-mono: ‹A [⊆] B ⟹ max-list A ≤ max-list B›
 by (induct A) (simp, metis linorder-not-le list.set-intros(1) max.absorb2 max.absorb3
```

max-list.simps(*2*) *max-list-in set-subset-Cons subset-code*(*1*))

lemma *max-list-vars-fms*:
assumes ‹*max-list* (*vars-fms A*) ≤ *n*› ‹*p* [∈] *A*›
shows ‹*max-list* (*vars-fm p*) ≤ *n*›
using *assms max-list-mono vars-fms-member* **by** (*meson dual-order.trans*)

definition *fresh* :: ‹*fm list* ⇒ *nat*› **where**
‹*fresh A* ≡ *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* ⨾ λ*m*. ⦇*E*, *F*⦈ (*s m*)) *n*›
by (*induct n*) *simp-all*

lemma *lift-lemma* [*simp*]: ‹(|x ⨟ E, F|) (*lift-tm* t) = (|E, F|) t›
by (*induct* t) (*auto cong*: *map-cong*)

lemma *sub-tm-semantics* [*simp*]: ‹(|E, F|) (*sub-tm* s t) = (|λn. (|E, F|) (s n), F|) t›
by (*induct* t) (*auto cong*: *map-cong*)

lemma *sub-fm-semantics* [*simp*]: ‹[|E, F, G|] (*sub-fm* s p) = [|λn. (|E, F|) (s n), F, G|] p›
by (*induct* p *arbitrary*: E s) (*auto cong*: *map-cong*)

4.3 Variables

lemma *upd-vars-tm* [*simp*]: ‹n [∉] *vars-tm* t ⟹ (|E(n := x), F|) t = (|E, F|) t›
by (*induct* t) (*auto cong*: *map-cong*)

lemma *add-upd-commute* [*simp*]: ‹(y ⨟ E(n := x)) m = ((y ⨟ E)(*Suc* n := x)) m›
by (*induct* m) *simp-all*

lemma *upd-vars-fm* [*simp*]: ‹*max-list* (*vars-fm* p) < n ⟹ [|E(n := x), F, G|] p = [|E, F, G|] p›
proof (*induct* p *arbitrary*: E n)
case (*Pre* P *ts*)
moreover have ‹*max-list* (*concat* (*map vars-tm ts*)) < n›
using *Pre.prems max-list-concat* **by** *simp*
then have ‹n [∉] *concat* (*map vars-tm ts*)›
using *max-list-in* **by** *blast*
then have ‹∀ t [∈] *ts*. n [∉] *vars-tm* t›
by *simp*
ultimately show *?case*
using *upd-vars-tm* **by** (*metis list.map-cong semantics-fm.simps*(*2*))
next
case (*Uni* p)
have ‹*?case* = ((∀ y. [|λm. (y ⨟ E(n := x)) m, F, G|] p) = (∀ y. [|y ⨟ E, F, G|] p))›
by (*simp add*: *fun-upd-def*)
then show *?case*
using *Uni* **by** *simp*
qed (*auto simp*: *max-list-append cong*: *map-cong*)

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*: ‹$n < Inr\ \$\ n$›
unfolding *sum-encode-def* **by** *simp*

lemma *sum-prod-decode-lt* [*simp*]: ‹*sum-decode* $n = Inr\ b \Longrightarrow (x, y) =$ *prod-decode* $b \Longrightarrow y < Suc\ n$›
by (*metis le-prod-encode-2 less-Suc-eq lt-sum-encode-Inr order-le-less-trans prod-decode-inverse sum-decode-inverse*)

lemma *sum-prod-decode-lt-Suc* [*simp*]:
‹*sum-decode* $n = Inr\ b \Longrightarrow (Suc\ x, y) =$ *prod-decode* $b \Longrightarrow x < Suc\ n$›
by (*metis dual-order.strict-trans le-prod-encode-1 lessI linorder-not-less lt-sum-encode-Inr not-less-eq prod-decode-inverse sum-decode-inverse*)

lemma *lt-list-encode*: ‹n [$\in$] $ns \Longrightarrow n <$ *list-encode* ns›
proof (*induct ns*)
case (*Cons m ns*)
then show *?case*
using *le-prod-encode-1 le-prod-encode-2*
by (*metis dual-order.strict-trans1 le-imp-less-Suc less-SucI list-encode.simps*(*2*) *set-ConsD*)
qed *simp*

lemma *prod-Suc-list-decode-lt* [*simp*]:
‹(x, *Suc* y) = *prod-decode* $n \Longrightarrow y'$ [$\in$] (*list-decode* y) $\Longrightarrow y' < n$›
by (*metis Suc-le-lessD lt-list-encode le-prod-encode-2 list-decode-inverse order-less-trans prod-decode-inverse*)

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*)))›
by *pat-completeness auto*
termination by (*relation* ‹*measure id*›) *simp-all*

lemma *tm-nat*: ‹*tm-of-nat* (*nat-of-tm* t) = t›
by (*induct t*) (*simp-all add*: *map-idI*)

lemma *surj-tm-of-nat*: ‹*surj tm-of-nat*›
unfolding *surj-def* **using** *tm-nat* **by** *metis*

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)))›
  by pat-completeness auto
termination by (relation ‹measure id›) simp-all

lemma fm-nat: ‹fm-of-nat (nat-of-fm p) = p›
  using tm-nat by (induct p) (simp-all add: map-idI)

lemma surj-fm-of-nat: ‹surj fm-of-nat›
  unfolding surj-def using fm-nat by metis
```

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›
  using tm-nat fm-nat by (cases r) (simp-all add: map-idI idle-nat-def)

lemma surj-rule-of-nat: ‹surj rule-of-nat›
  unfolding surj-def using rule-nat by metis

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)›
  by pat-completeness auto
termination by (relation ‹measure size›) standard

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

lemma UNIV-rules: ‹sset rules = UNIV›
  unfolding rules-def using UNIV-stream surj-rule-of-nat .

interpretation RuleSystem ‹λr s ss. eff r s = Some ss› rules UNIV
  by unfold-locales (auto simp: UNIV-rules intro: exI[of - Idle])

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

using *assms* **by** (*cases r r′ rule*: *rule.exhaust*[*case-product rule.exhaust*])
(*unfold enabled-def*, *auto split*: *if-splits*)

lemma *per-rules*: ‹*per r*›
unfolding *per-def UNIV-rules* **using** *per-rules′* **by** *fast*

interpretation *PersistentRuleSystem* ‹λ*r s ss*. *eff r s* = *Some ss*› *rules UNIV*
using *per-rules* **by** *unfold-locales*

definition ‹*prover* ≡ *mkTree rules*›

end

7 Export

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

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

declare *Stream.smember-code* [*code del*]
lemma [*code*]: ‹*Stream.smember x* (*y* ## *s*) = (*x* = *y* ∨ *Stream.smember x s*)›
unfolding *Stream.smember-def* **by** *auto*

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*› ‹∀ *A B*. (*A*, *B*) |∈| *ss* ⟶ (∀ (*E* :: - ⇒ 'a). *sc* (*E*, *F*, *G*) (*A*, *B*))›
shows ‹*sc* (*E*, *F*, *G*) (*A*, *B*)›
using *assms*
proof (*induct r* ‹(*A*, *B*)› *rule*: *eff.induct*)
case (*5 p q*)
then have ‹*sc* (*E*, *F*, *G*) (*A* [÷] (*p* ⟶ *q*), *p* # *B*)› ‹*sc* (*E*, *F*, *G*) (*q* # *A* [÷] (*p* ⟶ *q*), *B*)›
by (*metis eff.simps*(*5*) *finsertCI option.inject option.simps*(*3*))+
then show *?case*
using *5.prems*(*1*) **by** (*force split*: *if-splits*)
next
case (*7 t p*)
then have ‹*sc* (*E*, *F*, *G*) (⟨*t*⟩*p* # *A*, *B*)›
by (*metis eff.simps*(*7*) *finsert-iff option.inject option.simps*(*3*))
then show *?case*
using *7.prems*(*1*) **by** (*fastforce split*: *if-splits*)
next
case (*8 p*)
let *?n* = ‹*fresh* (*A* @ *B*)›

have *A*: ‹∀ *p* [∈] *A*. *max-list* (*vars-fm p*) < *?n*› **and** *B*: ‹∀ *p* [∈] *B*. *max-list* (*vars-fm p*) < *?n*›
unfolding *fresh-def* **using** *max-list-vars-fms max-list-mono vars-fms-member*
by (*metis Un-iff le-imp-less-Suc set-append*)+
from *8* **have** ‹*sc* (*E*(*?n* := *x*), *F*, *G*) (*A*, ⟨#*?n*⟩*p* # *B* [÷] ∀ *p*)› **for** *x*
by (*metis eff.simps*(*8*) *finsert-iff option.inject option.simps*(*3*))
then have ‹(∀ *p* [∈] *A*. ⟦*E*, *F*, *G*⟧ *p*) ⟶
(∀ *x*. ⟦(*x* ; λ*m*. (*E*(*?n* := *x*)) *m*), *F*, *G*⟧ *p*) ∨ (∃ *q* [∈] *B* [÷] ∀ *p*. ⟦*E*, *F*, *G*⟧ *q*)›
using *A B upd-vars-fm* **by** *fastforce*
then have ‹(∀ *p* [∈] *A*. ⟦*E*, *F*, *G*⟧ *p*) ⟶
(∀ *x*. ⟦((*x* ; *E*)(*Suc ?n* := *x*)), *F*, *G*⟧ *p*) ∨ (∃ *q* [∈] *B* [÷] ∀ *p*. ⟦*E*, *F*, *G*⟧ *q*)›
unfolding *add-upd-commute* **by** *blast*
moreover have ‹*max-list* (*vars-fm p*) < *?n*›
using *B 8.prems*(*1*) **by** (*metis eff.simps*(*8*) *option.distinct*(*1*) *vars-fm.simps*(*4*))
ultimately have ‹*sc* (*E*, *F*, *G*) (*A*, ∀ *p* # (*B* [÷] ∀ *p*))›
by *auto*
moreover have ‹∀ *p* [∈] *B*›
using *8.prems*(*1*) **by** (*simp split*: *if-splits*)
ultimately show *?case*
by (*metis* (*full-types*) *Diff-iff sc.simps set-ConsD set-removeAll*)
qed (*fastforce split*: *if-splits*)+

interpretation *Soundness* ‹λ*r s ss*. *eff r s* = *Some ss*› *rules UNIV sc*
unfolding *Soundness-def* **using** *eff-sound* **by** *fast*

theorem *prover-soundness*:
assumes ‹*tfinite t*› **and** ‹*wf t*›
shows ‹*sc* (*E*, *F*, *G*) (*fst* (*root t*))›
using *assms soundness* **by** *fast*

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* ∈ *A* ⟹ ‡*P ts* ∈ *B* ⟹ *False*› **and**
FlsA: ‹⊥ ∉ *A*› **and**
ImpA: ‹*p* ⟶ *q* ∈ *A* ⟹ *p* ∈ *B* ∨ *q* ∈ *A*› **and**
ImpB: ‹*p* ⟶ *q* ∈ *B* ⟹ *p* ∈ *A* ∧ *q* ∈ *B*› **and**
UniA: ‹∀ *p* ∈ *A* ⟹ ∀ *t*. ⟨*t*⟩*p* ∈ *A*› **and**
UniB: ‹∀ *p* ∈ *B* ⟹ ∃ *t*. ⟨*t*⟩*p* ∈ *B*›

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

lemma *id-tm* [*simp*]: ‹(|#, †|) *t* = *t*›
by (*induct t*) (*auto cong*: *map-cong*)

lemma *size-sub-fm* [*simp*]: ‹*size* (*sub-fm s p*) = *size p*›
by (*induct p arbitrary*: *s*) *auto*

theorem *Hintikka-counter-model*:
assumes ‹*Hintikka A B*›
shows ‹(*p* ∈ *A* ⟶ *M A p*) ∧ (*p* ∈ *B* ⟶ ¬ *M A p*)›
proof (*induct p rule*: *wf-induct* [**where** *r*=‹*measure size*›])
case *1*
then show *?case* **..**
next
case (*2 x*)
then show *?case*
proof (*cases x*; *safe del*: *notI*)
case *Falsity*
show ‹⊥ ∈ *A* ⟹ *M A* ⊥› ‹⊥ ∈ *B* ⟹ ¬ *M A* ⊥›
using *Hintikka.FlsA assms* **by** *simp-all*
next
case (*Pre P ts*)
show ‹‡*P ts* ∈ *A* ⟹ *M A* (‡*P ts*)› ‹‡*P ts* ∈ *B* ⟹ ¬ *M A* (‡*P ts*)›
using *Hintikka.Basic assms* **by** (*auto cong*: *map-cong*)
next
case (*Imp p q*)
show ‹*p* ⟶ *q* ∈ *A* ⟹ *M A* (*p* ⟶ *q*)› ‹*p* ⟶ *q* ∈ *B* ⟹ ¬ *M A* (*p* ⟶ *q*)›
using *assms Hintikka.ImpA*[*of A B p q*] *Hintikka.ImpB*[*of A B p q*] *Imp 2* **by** *auto*
next
case (*Uni p*)
have ‹⟨*t*⟩*p* ∈ *A* ⟹ *M A* (⟨*t*⟩*p*)› ‹⟨*t*⟩*p* ∈ *B* ⟹ ¬ *M A* (⟨*t*⟩*p*)› **for** *t*
using *Uni 2* **by** (*metis fm.size*(*8*) *in-measure lessI less-add-same-cancel1 size-sub-fm*)+
then show ‹∀ *p* ∈ *A* ⟹ *M A* (∀ *p*)› ‹∀ *p* ∈ *B* ⟹ ¬ *M A* (∀ *p*)›
using *assms Hintikka.UniA*[*of A B p*] *Hintikka.UniB*[*of A B p*] **by** *auto*
qed
qed

9.2 Escape Paths Form Hintikka Sets

lemma *sset-sdrop*: ‹*sset* (*sdrop n s*) ⊆ *sset s*›
by (*induct n arbitrary*: *s*) (*auto intro*: *stl-sset in-mono*)

lemma *epath-sdrop*: ‹*epath steps* ⟹ *epath* (*sdrop n steps*)›
by (*induct n*) (*auto elim*: *epath.cases*)

lemma *eff-preserves-Pre*:
assumes ‹*effStep* ((*A*, *B*), *r*) *ss*› ‹(*A*′, *B*′) |∈| *ss*›

shows ‹(‡P ts [∈] $A \Longrightarrow$ ‡P ts [∈] A')› ‹‡P ts [∈] $B \Longrightarrow$ ‡P ts [∈] B'›
using *assms* **by** (*induct r* ‹(A, B)› *rule*: *eff.induct*) (*auto split*: *if-splits*)

lemma *epath-eff*:
assumes ‹*epath steps*› ‹*effStep* (*shd steps*) *ss*›
shows ‹*fst* (*shd* (*stl steps*)) |∈| *ss*›
using *assms* **by** (*auto elim*: *epath.cases*)

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*) $\Longrightarrow$ ‡P ts [∈] *lhsd* (*sdrop m steps*)›
‹‡P ts [∈] *rhs* (*shd steps*) $\Longrightarrow$ ‡P ts [∈] *rhsd* (*sdrop m steps*)›
using *assms eff-preserves-Pre*
by (*induct m arbitrary*: *steps*) (*simp*; *metis* (*no-types*, *lifting*) *epath.cases surjective-pairing*)+

lemma *Saturated-sdrop*:
assumes ‹*Saturated steps*›
shows ‹*Saturated* (*sdrop n steps*)›
using *assms* **unfolding** *Saturated-def saturated-def* **by** (*simp add*: *alw-iff-sdrop*)

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

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

lemma *treeA-snth*: ‹*p* ∈ *treeA steps* $\Longrightarrow$ ∃ *n*. *p* [∈] *lhsd* (*sdrop n steps*)›
unfolding *treeA-def* **using** *sset-range*[*of steps*] **by** *simp*

lemma *treeB-snth*: ‹*p* ∈ *treeB steps* $\Longrightarrow$ ∃ *n*. *p* [∈] *rhsd* (*sdrop n steps*)›
unfolding *treeB-def* **using** *sset-range*[*of steps*] **by** *simp*

lemma *treeA-sdrop*: ‹*treeA* (*sdrop n steps*) ⊆ *treeA steps*›
unfolding *treeA-def* **by** (*induct n*) (*simp*, *metis SUP-subset-mono order-refl sset-sdrop*)

lemma *treeB-sdrop*: ‹*treeB* (*sdrop n steps*) ⊆ *treeB steps*›
unfolding *treeB-def* **by** (*induct n*) (*simp*, *metis SUP-subset-mono order-refl sset-sdrop*)

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

using *assms* **unfolding** *Saturated-def saturated-def UNIV-rules* **by** (*auto simp*: *ev-iff-sdrop*)

lemma *Hintikka-epath*:
assumes ‹*epath steps*› ‹*Saturated steps*›
shows ‹*Hintikka* (*treeA steps*) (*treeB steps*)›
proof
fix *P ts*
assume ‹‡*P ts* ∈ *treeA steps*›
then obtain *m* **where** *m*: ‹‡*P ts* [∈] *lhsd* (*sdrop m steps*)›
using *treeA-snth* **by** *auto*

assume ‹‡*P ts* ∈ *treeB steps*›
then obtain *k* **where** *k*: ‹‡*P ts* [∈] *rhsd* (*sdrop k steps*)›
using *treeB-snth* **by** *auto*

let *?j* = ‹*m* + *k*›
let *?jstep* = ‹*shd* (*sdrop ?j steps*)›

have ‹‡*P ts* [∈] *lhs ?jstep*›
using *assms m epath-sdrop epath-Pre-sdrop* **by** (*metis* (*no-types*, *lifting*) *sdrop-add*)
moreover have ‹‡*P ts* [∈] *rhs ?jstep*›
using *assms k epath-sdrop epath-Pre-sdrop* **by** (*metis* (*no-types*, *lifting*) *add.commute sdrop-add*)
ultimately have ‹*enabled* (*Axiom P ts*) (*fst ?jstep*)›
unfolding *enabled-def* **by** (*metis eff.simps*(*2*) *prod.exhaust-sel*)
then obtain *j′* **where** ‹*takenAtStep* (*Axiom P ts*) (*shd* (*sdrop j′ steps*))›
using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(*1*)] *Saturated-sdrop*[*OF assms*(*2*)]] **by** *auto*
then have ‹*eff* (*snd* (*shd* (*sdrop j′ steps*))) (*fst* (*shd* (*sdrop j′ steps*))) = *None*›
using *assms*(*1*) *epath-sdrop epath-eff*
by (*metis* (*no-types*, *lifting*) *eff.simps*(*2*) *epath.simps equalsffemptyD surjective-pairing*)
then show *False*
using *assms*(*1*) *epath-sdrop* **by** (*metis epath.cases option.discI*)
next
show ‹⊥ ∉ *treeA steps*›
proof
assume ‹⊥ ∈ *treeA steps*›
then have ‹∃*j*. *enabled FlsL* (*fst* (*shd* (*sdrop j steps*)))›
unfolding *enabled-def* **using** *treeA-snth* **by** (*metis eff.simps*(*3*) *prod.exhaust-sel sdrop-simps*(*1*))
then obtain *j* **where** ‹*takenAtStep FlsL* (*shd* (*sdrop j steps*))› (**is** ‹*takenAtStep* - (*shd ?steps*)›)
using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(*1*)] *Saturated-sdrop*[*OF assms*(*2*)]] **by** *auto*
then have ‹*eff* (*snd* (*shd ?steps*)) (*fst* (*shd ?steps*)) = *None*›
using *assms*(*1*) *epath-sdrop epath-eff*
by (*metis* (*no-types*, *lifting*) *eff.simps*(*3*) *epath.simps equalsffemptyD surjec-*

tive-pairing)
    then show *False*
      using *assms*(*1*) *epath-sdrop* **by** (*metis epath.cases option.discI*)
  qed
next
  fix *p q*
  assume ‹*p* ⟶ *q* ∈ *treeA steps*›
  then have ‹∃ *k*. *enabled* (*ImpL p q*) (*fst* (*shd* (*sdrop k steps*)))›
    unfolding *enabled-def* **using** *treeA-snth* **by** (*metis eff.simps*(*5*) *prod.exhaust-sel sdrop-simps*(*1*))
  then obtain *j* **where** ‹*takenAtStep* (*ImpL p q*) (*shd* (*sdrop j steps*))› (**is** ‹*takenAtStep* - (*shd ?s*)›)
    using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(*1*)] *Saturated-sdrop*[*OF assms*(*2*)]] **by** *auto*
  then have ‹*fst* (*shd* (*stl ?s*)) |∈|
      {| (*lhsd ?s* [÷] (*p* ⟶ *q*), *p* # *rhsd ?s*), (*q* # *lhsd ?s* [÷] (*p* ⟶ *q*), *rhsd ?s*) |}›
    using *assms*(*1*) *epath-sdrop epath-eff*
    by (*metis* (*no-types*, *lifting*) *eff.simps*(*5*) *epath.cases option.distinct*(*1*) *prod.collapse*)
  then have ‹*p* [∈] *rhs* (*shd* (*stl ?s*)) ∨ *q* [∈] *lhs* (*shd* (*stl ?s*))›
    by *auto*
  then have ‹*p* ∈ *treeB* (*stl ?s*) ∨ *q* ∈ *treeA* (*stl ?s*)›
    unfolding *treeA-def treeB-def* **by** (*meson UN-I shd-sset*)
  then show ‹*p* ∈ *treeB steps* ∨ *q* ∈ *treeA steps*›
    using *treeA-sdrop treeB-sdrop* **by** (*metis sdrop-simps*(*2*) *subsetD*)
next
  fix *p q*
  assume ‹*p* ⟶ *q* ∈ *treeB steps*›
  then have ‹∃ *k*. *enabled* (*ImpR p q*) (*fst* (*shd* (*sdrop k steps*)))›
    unfolding *enabled-def* **using** *treeB-snth* **by** (*metis eff.simps*(*6*) *prod.exhaust-sel sdrop-simps*(*1*))
  then obtain *j* **where** ‹*takenAtStep* (*ImpR p q*) (*shd* (*sdrop j steps*))› (**is** ‹*takenAtStep* - (*shd ?s*)›)
    using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(*1*)] *Saturated-sdrop*[*OF assms*(*2*)]] **by** *auto*
  then have ‹*fst* (*shd* (*stl ?s*)) |∈| {| (*p* # *lhsd ?s*, *q* # *rhsd ?s* [÷] (*p* ⟶ *q*)) |}›
    using *assms*(*1*) *epath-sdrop epath-eff*
    by (*metis* (*no-types*, *lifting*) *eff.simps*(*6*) *epath.cases option.distinct*(*1*) *prod.collapse*)
  then have ‹*p* [∈] *lhs* (*shd* (*stl ?s*)) ∧ *q* [∈] *rhs* (*shd* (*stl ?s*))›
    by *auto*
  then have ‹*p* ∈ *treeA* (*stl ?s*) ∧ *q* ∈ *treeB* (*stl ?s*)›
    unfolding *treeA-def treeB-def* **by** (*meson UN-I shd-sset*)
  then show ‹*p* ∈ *treeA steps* ∧ *q* ∈ *treeB steps*›
    using *treeA-sdrop treeB-sdrop* **by** (*metis sdrop-simps*(*2*) *subsetD*)
next
  fix *p*
  assume ∗: ‹∀ *p* ∈ *treeA steps*›
  show ‹∀ *t*. ⟨*t*⟩*p* ∈ *treeA steps*›
  proof

fix *t*
from ∗ **have** ‹∃ *k*. *enabled* (*UniL t p*) (*fst* (*shd* (*sdrop k steps*)))›
unfolding *enabled-def* **using** *treeA-snth* **by** (*metis eff.simps*(7) *prod.exhaust-sel sdrop-simps*(1))
then obtain *j* **where** ‹*takenAtStep* (*UniL t p*) (*shd* (*sdrop j steps*))›(**is** ‹*takenAtStep* - (*shd ?s*)›)
using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(1)] *Saturated-sdrop*[*OF assms*(2)]] **by** *auto*
then have ‹*fst* (*shd* (*stl ?s*)) |∈| {| (⟨*t*⟩*p* # *lhsd ?s*, *rhsd ?s*) |}›
using *assms*(1) *epath-sdrop epath-eff*
by (*metis* (*no-types*, *lifting*) *eff.simps*(7) *epath.cases option.distinct*(1) *prod.collapse*)
then have ‹⟨*t*⟩*p* [∈] *lhs* (*shd* (*stl ?s*))›
by *auto*
then have ‹⟨*t*⟩*p* ∈ *treeA* (*stl ?s*)›
unfolding *treeA-def* **by** (*meson UN-I shd-sset*)
then show ‹⟨*t*⟩*p* ∈ *treeA steps*›
using *treeA-sdrop* **by** (*metis sdrop-simps*(2) *subsetD*)
qed
next
fix *p*
assume ∗: ‹∀ *p* ∈ *treeB steps*›
then have ‹∃ *k*. *enabled* (*UniR p*) (*fst* (*shd* (*sdrop k steps*)))›
unfolding *enabled-def* **using** *treeB-snth* **by** (*metis eff.simps*(8) *prod.exhaust-sel sdrop-simps*(1))
then obtain *j* **where** ‹*takenAtStep* (*UniR p*) (*shd* (*sdrop j steps*))›(**is** ‹*takenAtStep* - (*shd ?s*)›)
using *enabled-ex-taken*[*OF epath-sdrop*[*OF assms*(1)] *Saturated-sdrop*[*OF assms*(2)]] **by** *auto*
then have ‹*fst* (*shd* (*stl ?s*)) |∈|
{| (*lhsd ?s*, ⟨#(*fresh* (*lhsd ?s* @ *rhsd ?s*))⟩*p* # *rhsd ?s* [÷] ∀ *p*) |}›
using *assms*(1) *epath-sdrop epath-eff*
by (*metis* (*no-types*, *lifting*) *eff.simps*(8) *epath.cases option.distinct*(1) *prod.collapse*)
then have ‹∃ *t*. ⟨*t*⟩*p* [∈] *rhs* (*shd* (*stl ?s*))›
by *auto*
then have ‹∃ *t*. ⟨*t*⟩*p* ∈ *treeB* (*stl ?s*)›
unfolding *treeB-def* **by** (*meson UN-I shd-sset*)
then show ‹∃ *t*. ⟨*t*⟩*p* ∈ *treeB steps*›
using *treeB-sdrop* **by** (*metis sdrop-simps*(2) *subsetD*)
qed

9.3 Completeness

lemma *fair-stream-rules*: ‹*Fair-Stream.fair rules*›
unfolding *rules-def* **using** *fair-stream surj-rule-of-nat* **.**

lemma *fair-rules*: ‹*fair rules*›
using *fair-stream-rules* **unfolding** *Fair-Stream.fair-def fair-def alw-iff-sdrop ev-holds-sset*
by (*metis dual-order.refl le-Suc-ex sdrop-snth snth-sset*)

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 −
{ assume ‹¬ *?A*›
with *assms* **have** ‹¬ *tfinite* (*mkTree rules* (*A*, *B*))›
unfolding *prover-def* **using** *wf-mkTree fair-rules* **by** *simp*
then obtain *steps* **where** ‹*ipath* (*mkTree rules* (*A*, *B*)) *steps*› **using** *Konig* **by** *blast*
with *assms* **have** ‹*fst* (*shd steps*) = (*A*, *B*) ∧ *epath steps* ∧ *Saturated steps*›
by (*metis* (*no-types*, *lifting*) *fair-rules UNIV-I fst-conv ipath.cases*
ipath-mkTree-Saturated mkTree.simps(*1*) *wf-ipath-epath wf-mkTree*)
then have *?B* **by** *blast*
}
then show *?thesis* **by** *blast*
qed

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 −
have ‹*Hintikka* (*treeA steps*) (*treeB steps*)› (**is** ‹*Hintikka ?A ?B*›)
using *assms Hintikka-epath assms* **by** *simp*
moreover have ‹∀ *p* [∈] *A*. *p* ∈ *?A*› ‹∀ *p* [∈] *B*. *p* ∈ *?B*›
using *assms shd-sset* **unfolding** *treeA-def treeB-def* **by** *fastforce*+
ultimately have ‹∀ *p* [∈] *A*. *M ?A p*› ‹∀ *p* [∈] *B*. ¬ *M ?A p*›
using *Hintikka-counter-model assms* **by** *blast*+
then show *?thesis*
by *auto*
qed

theorem *prover-completeness*:
assumes ‹∀ (*E* :: - ⇒ *tm*) *F G*. *sc* (*E*, *F*, *G*) (*A*, *B*)›
defines ‹*t* ≡ *prover* (*A*, *B*)›
shows ‹*fst* (*root t*) = (*A*, *B*) ∧ *wf t* ∧ *tfinite t*›
using *assms epath-prover epath-countermodel* **by** *blast*

corollary
assumes ‹∀ (*E* :: - ⇒ *tm*) *F G*. ⟦*E*, *F*, *G*⟧ *p*›
defines ‹*t* ≡ *prover* ([], [*p*])›
shows ‹*fst* (*root t*) = ([], [*p*]) ∧ *wf t* ∧ *tfinite t*›
using *assms prover-completeness* **by** *simp*

end

10 Result

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

theorem *prover-soundness-completeness*:
fixes *A B* :: ‹*fm list*›
defines ‹*t* ≡ *prover* (*A*, *B*)›
shows ‹*tfinite t* ∧ *wf t* ⟷ (∀(*E* :: - ⇒ *tm*) *F G*. *sc* (*E*, *F*, *G*) (*A*, *B*))›
using *assms prover-soundness prover-completeness* **unfolding** *prover-def* **by** *fastforce*

corollary
fixes *p* :: *fm*
defines ‹*t* ≡ *prover* ([], [*p*])›
shows ‹*tfinite t* ∧ *wf t* ⟷ (∀(*E* :: - ⇒ *tm*) *F G*. ⟦*E*, *F*, *G*⟧ *p*)›
using *assms prover-soundness-completeness* **by** *simp*

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.