

# A Naive Prover for First-Order Logic

Asta Halkjær From

February 6, 2026

## 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

<b>1</b>	<b>List Syntax</b>	<b>3</b>
<b>2</b>	<b>Fair Streams</b>	<b>4</b>
<b>3</b>	<b>Syntax</b>	<b>6</b>
3.1	Terms and Formulas . . . . .	6
3.1.1	Substitution . . . . .	7
3.1.2	Variables . . . . .	7
3.2	Rules . . . . .	8
<b>4</b>	<b>Semantics</b>	<b>8</b>
4.1	Definition . . . . .	8
4.2	Substitution . . . . .	9
4.3	Variables . . . . .	9
<b>5</b>	<b>Encoding</b>	<b>10</b>
5.1	Terms . . . . .	10
5.2	Formulas . . . . .	11
5.3	Rules . . . . .	11
<b>6</b>	<b>Prover</b>	<b>12</b>
<b>7</b>	<b>Export</b>	<b>13</b>
<b>8</b>	<b>Soundness</b>	<b>14</b>
<b>9</b>	<b>Completeness</b>	<b>15</b>
9.1	Hintikka Counter Model . . . . .	15
9.2	Escape Paths Form Hintikka Sets . . . . .	17
9.3	Completeness . . . . .	21
<b>10</b>	<b>Result</b>	<b>22</b>

# 1 List Syntax

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

**abbreviation** *list-member-syntax* ::  $\langle 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$  ( $\langle - \in \rangle \rightarrow [51, 51]$  50)  
**where**

$\langle x \in \rangle A \equiv x \in \text{set } A$

**abbreviation** *list-not-member-syntax* ::  $\langle 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$  ( $\langle - \notin \rangle \rightarrow [51, 51]$  50) **where**

$\langle x \notin \rangle A \equiv x \notin \text{set } A$

**abbreviation** *list-subset-syntax* ::  $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$  ( $\langle - \subset \rangle \rightarrow [51, 51]$  50)  
**where**

$\langle A \subset \rangle B \equiv \text{set } A \subset \text{set } B$

**abbreviation** *list-subset-eq-syntax* ::  $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \rangle$  ( $\langle - \subseteq \rangle \rightarrow [51, 51]$  50) **where**

$\langle A \subseteq \rangle B \equiv \text{set } A \subseteq \text{set } B$

**abbreviation** *removeAll-syntax* ::  $\langle 'a \text{ list} \Rightarrow 'a \Rightarrow 'a \text{ list} \rangle$  (**infix**  $\langle \div \rangle$  75) **where**

$\langle A \div \rangle x \equiv \text{removeAll } x \ A$

**syntax** (*ASCII*)

*-BallList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \text{ALL } (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

*-BexList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \text{EX } (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

*-Bex1List* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \text{EX! } (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

*-BleastList* ::  $\langle \text{id} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow 'a \rangle$  ( $\langle (\exists \text{LEAST } (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

**syntax** (*input*)

*-BallList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists! (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

*-BexList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists? (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

*-Bex1List* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists?! (-/[:-]) ./ -) \rangle [0, 0, 10]$  10)

**syntax**

*-BallList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \forall (-/[ \in ]) ./ -) \rangle [0, 0, 10]$  10)

*-BexList* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \exists (-/[ \in ]) ./ -) \rangle [0, 0, 10]$  10)

*-Bex1List* ::  $\langle \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow \text{bool} \rangle$  ( $\langle (\exists \exists! (-/[ \in ]) ./ -) \rangle [0, 0, 10]$  10)

*-BleastList* ::  $\langle \text{id} \Rightarrow 'a \text{ list} \Rightarrow \text{bool} \Rightarrow 'a \rangle$  ( $\langle (\exists \text{LEAST } (-/[ \in ]) ./ -) \rangle [0, 0, 10]$  10)

### **syntax-consts**

-*BallList*  $\rightleftharpoons$  *Ball* and  
-*BexList*  $\rightleftharpoons$  *Bex* and  
-*Bex1List*  $\rightleftharpoons$  *Ex1* and  
-*BleasList*  $\rightleftharpoons$  *Least*

### **translations**

$\forall x[x \in] A. P \rightleftharpoons \text{CONST Ball (CONST set A) } (\lambda x. P)$   
 $\exists x[x \in] A. P \rightleftharpoons \text{CONST Bex (CONST set A) } (\lambda x. P)$   
 $\exists !x[x \in] A. P \rightarrow \exists !x. x [ \in ] A \wedge P$   
 $\text{LEAST } x[:] A. P \rightarrow \text{LEAST } x. x [ \in ] A \wedge P$

### **syntax (ASCII output)**

-*setlessAllList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists ALL \text{ -}[<]-./ -) \rangle [0, 0, 10] 10)$   
-*setlessExList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists EX \text{ -}[<]-./ -) \rangle [0, 0, 10] 10)$   
-*setleAllList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists ALL \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$   
-*setleExList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists EX \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$   
-*setleEx1List* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists EX! \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$

### **syntax**

-*setlessAllList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists \forall \text{ -}[<]-./ -) \rangle [0, 0, 10] 10)$   
-*setlessExList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists \exists \text{ -}[<]-./ -) \rangle [0, 0, 10] 10)$   
-*setleAllList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists \forall \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$   
-*setleExList* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists \exists \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$   
-*setleEx1List* ::  $\langle [idt, 'a, bool] \Rightarrow bool \rangle (\langle (\exists \exists! \text{ -}[<=]-./ -) \rangle [0, 0, 10] 10)$

### **syntax-consts**

-*setlessAllList* -*setleAllList*  $\rightleftharpoons$  *All* and  
-*setlessExList* -*setleExList*  $\rightleftharpoons$  *Ex* and  
-*setleEx1List*  $\rightleftharpoons$  *Ex1*

### **translations**

$\forall A[C] B. P \rightarrow \forall A. A [C] B \rightarrow P$   
 $\exists A[C] B. P \rightarrow \exists A. A [C] B \wedge P$   
 $\forall A[\subseteq] B. P \rightarrow \forall A. A [\subseteq] B \rightarrow P$   
 $\exists A[\subseteq] B. P \rightarrow \exists A. A [\subseteq] B \wedge P$   
 $\exists !A[\subseteq] B. P \rightarrow \exists !A. A [\subseteq] B \wedge P$

end

## **2 Fair Streams**

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

**definition** *upt-lists* ::  $\langle \text{nat list stream} \rangle$  **where**

$\langle \text{upt-lists} \equiv \text{smap (upt 0) (stl nats)} \rangle$

**definition** *fair-nats* ::  $\langle \text{nat stream} \rangle$  **where**

$\langle \text{fair-nats} \equiv \text{flat upt-lists} \rangle$

**definition** *fair* ::  $\langle 'a \text{ stream} \Rightarrow \text{bool} \rangle$  **where**  
 $\langle \text{fair } s \equiv \forall x \in \text{sset } s. \forall m. \exists n \geq m. s !! n = x \rangle$

**lemma** *upt-lists-snth*:  $\langle x \leq n \implies x \in \text{set } (\text{upt-lists } !! n) \rangle$   
**unfolding** *upt-lists-def* **by** *auto*

**lemma** *all-ex-upt-lists*:  $\langle \exists n \geq m. x \in \text{set } (\text{upt-lists } !! n) \rangle$   
**using** *upt-lists-snth* **by** (*meson dual-order.strict-trans1 gt-ex nle-le*)

**lemma** *upt-lists-ne*:  $\langle \forall xs \in \text{sset } \text{upt-lists}. xs \neq [] \rangle$   
**unfolding** *upt-lists-def* **by** (*simp add: sset-range*)

**lemma** *sset-flat-stl*:  $\langle \text{sset } (\text{flat } (\text{stl } s)) \subseteq \text{sset } (\text{flat } s) \rangle$   
**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**  $\langle x = s !! n ! m \rangle \langle m < \text{length } (s !! n) \rangle \langle \forall xs \in \text{sset } s. xs \neq [] \rangle$   
**shows**  $\langle \exists n' \geq n. x = \text{flat } s !! n' \rangle$   
**using** *assms*  
**proof** (*induct n arbitrary: s*)  
**case** *0*  
**then show** *?case*  
**using** *flat-snth* **by** *fastforce*  
**next**  
**case** (*Suc n*)  
**have**  $\langle ?case = (\exists n' \geq n. x = \text{flat } s !! \text{Suc } n') \rangle$   
**by** (*metis Suc-le-D Suc-le-mono*)  
**also have**  $\langle \dots = (\exists n' \geq n. x = \text{stl } (\text{flat } s) !! n') \rangle$   
**by** *simp*  
**finally have**  $\langle ?case = (\exists n' \geq n. x = (\text{tl } (\text{shd } s) @- \text{flat } (\text{stl } s)) !! n') \rangle$   
**using** *Suc.premis flat-unfold* **by** (*simp add: shd-sset*)  
**then have** *?case* **if**  $\langle \exists n' \geq n. x = \text{flat } (\text{stl } s) !! n' \rangle$   
**using** *that* **by** (*metis (no-types, opaque-lifting) add commute add-diff-cancel-left'*  
*dual-order.trans le-add2 shift-snth-ge*)  
**moreover** {  
**have**  $\langle x = \text{stl } s !! n ! m \rangle \langle m < \text{length } (\text{stl } s !! n) \rangle$   
**using** *Suc.premis* **by** *simp-all*  
**moreover have**  $\langle \forall xs \in \text{sset } (\text{stl } s). xs \neq [] \rangle$   
**using** *Suc.premis* **by** (*cases s*) *simp-all*  
**ultimately have**  $\langle \exists n' \geq n. x = \text{flat } (\text{stl } s) !! n' \rangle$   
**using** *Suc.hyps* **by** *blast* }  
**ultimately show** *?case* .  
**qed**

```

lemma all-ex-fair-nats:  $\langle \exists n \geq m. \text{fair-nats} !! n = x \rangle$ 
proof –
  have  $\langle \exists n \geq m. x \in \text{set} (\text{upt-lists} !! n) \rangle$ 
    using all-ex-upt-lists .
  then have  $\langle \exists n \geq m. \exists k < \text{length} (\text{upt-lists} !! n). \text{upt-lists} !! n ! k = x \rangle$ 
    by (simp add: in-set-conv-nth)
  then obtain  $n\ k$  where  $\langle m \leq n \rangle \langle k < \text{length} (\text{upt-lists} !! n) \rangle \langle \text{upt-lists} !! n ! k$ 
     $= x \rangle$ 
    by blast
  then obtain  $n'$  where  $\langle n \leq n' \rangle \langle x = \text{flat} \text{upt-lists} !! n' \rangle$ 
    using flat-snth-nth upt-lists-ne by metis
  moreover have  $\langle m \leq n' \rangle$ 
    using  $\langle m \leq n \rangle \langle n \leq n' \rangle$  by simp
  ultimately show ?thesis
    unfolding fair-nats-def by blast
qed

lemma fair-surj:
  assumes  $\langle \text{surj } f \rangle$ 
  shows  $\langle \text{fair} (\text{smap } f \text{ fair-nats}) \rangle$ 
  using assms unfolding fair-def by (metis UNIV-I all-ex-fair-nats imageE snth-smap)

definition fair-stream ::  $\langle (\text{nat} \Rightarrow 'a) \Rightarrow 'a \text{ stream} \rangle$  where
   $\langle \text{fair-stream } f \equiv \text{smap } f \text{ fair-nats} \rangle$ 

theorem fair-stream:  $\langle \text{surj } f \implies \text{fair} (\text{fair-stream } f) \rangle$ 
  unfolding fair-stream-def using fair-surj .

theorem UNIV-stream:  $\langle \text{surj } f \implies \text{sset} (\text{fair-stream } f) = \text{UNIV} \rangle$ 
  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 ( $\langle \# \rangle$ )
  | Fun nat  $\langle \text{tm list} \rangle$  ( $\langle \dagger \rangle$ )

```

```

datatype fm
  = Falsity ( $\langle \perp \rangle$ )
  | Pre nat  $\langle \text{tm list} \rangle$  ( $\langle \ddagger \rangle$ )
  | Imp fm fm (infixr  $\langle \longrightarrow \rangle$  55)

```

| *Uni fm* ( $\langle \forall \rangle$ )

**type-synonym** *sequent* =  $\langle \text{fm list} \times \text{fm list} \rangle$

### 3.1.1 Substitution

**primrec** *add-env* ::  $\langle 'a \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \rangle$  (**infix**  $\langle \circ \rangle$  0) **where**

$\langle (t \circ s) \ 0 = t \rangle$   
|  $\langle (t \circ s) \ (\text{Suc } n) = s \ n \rangle$

**primrec** *lift-tm* ::  $\langle \text{tm} \Rightarrow \text{tm} \rangle$  **where**

$\langle \text{lift-tm} \ (\#n) = \#(n+1) \rangle$   
|  $\langle \text{lift-tm} \ (\dagger f \ ts) = \dagger f \ (\text{map } \text{lift-tm} \ ts) \rangle$

**primrec** *sub-tm* ::  $\langle (\text{nat} \Rightarrow \text{tm}) \Rightarrow \text{tm} \Rightarrow \text{tm} \rangle$  **where**

$\langle \text{sub-tm} \ s \ (\#n) = s \ n \rangle$   
|  $\langle \text{sub-tm} \ s \ (\dagger f \ ts) = \dagger f \ (\text{map} \ (\text{sub-tm} \ s) \ ts) \rangle$

**primrec** *sub-fm* ::  $\langle (\text{nat} \Rightarrow \text{tm}) \Rightarrow \text{fm} \Rightarrow \text{fm} \rangle$  **where**

$\langle \text{sub-fm} \ - \ \perp = \perp \rangle$   
|  $\langle \text{sub-fm} \ s \ (\dagger P \ ts) = \dagger P \ (\text{map} \ (\text{sub-tm} \ s) \ ts) \rangle$   
|  $\langle \text{sub-fm} \ s \ (p \longrightarrow q) = \text{sub-fm} \ s \ p \longrightarrow \text{sub-fm} \ s \ q \rangle$   
|  $\langle \text{sub-fm} \ s \ (\forall p) = \forall (\text{sub-fm} \ (\#0 \circ \lambda n. \ \text{lift-tm} \ (s \ n)) \ p) \rangle$

**abbreviation** *inst-single* ::  $\langle \text{tm} \Rightarrow \text{fm} \Rightarrow \text{fm} \rangle$  ( $\langle \langle - \rangle \rangle$ ) **where**

$\langle \langle t \rangle \equiv \text{sub-fm} \ (t \circ \#) \rangle$

### 3.1.2 Variables

**primrec** *vars-tm* ::  $\langle \text{tm} \Rightarrow \text{nat list} \rangle$  **where**

$\langle \text{vars-tm} \ (\#n) = [n] \rangle$   
|  $\langle \text{vars-tm} \ (\dagger \ ts) = \text{concat} \ (\text{map} \ \text{vars-tm} \ ts) \rangle$

**primrec** *vars-fm* ::  $\langle \text{fm} \Rightarrow \text{nat list} \rangle$  **where**

$\langle \text{vars-fm} \ \perp = [] \rangle$   
|  $\langle \text{vars-fm} \ (\dagger \ ts) = \text{concat} \ (\text{map} \ \text{vars-tm} \ ts) \rangle$   
|  $\langle \text{vars-fm} \ (p \longrightarrow q) = \text{vars-fm} \ p \ @ \ \text{vars-fm} \ q \rangle$   
|  $\langle \text{vars-fm} \ (\forall p) = \text{vars-fm} \ p \rangle$

**primrec** *max-list* ::  $\langle \text{nat list} \Rightarrow \text{nat} \rangle$  **where**

$\langle \text{max-list} \ [] = 0 \rangle$   
|  $\langle \text{max-list} \ (x \ # \ xs) = \text{max} \ x \ (\text{max-list} \ xs) \rangle$

**lemma** *max-list-append*:  $\langle \text{max-list} \ (xs \ @ \ ys) = \text{max} \ (\text{max-list} \ xs) \ (\text{max-list} \ ys) \rangle$

**by** (*induct xs*) *auto*

**lemma** *max-list-concat*:  $\langle xs \ [\in] \ xss \Longrightarrow \ \text{max-list} \ xs \ \leq \ \text{max-list} \ (\text{concat} \ xss) \rangle$

**by** (*induct xss*) (*auto simp: max-list-append*)

**lemma** *max-list-in*:  $\langle \text{max-list} \ xs < n \Longrightarrow \ n \ [\notin] \ xs \rangle$

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)⟩

|  $\langle \llbracket E, F, G \rrbracket (\forall p) = (\forall x. \llbracket x \circ E, F, G \rrbracket p) \rangle$

**fun**  $sc :: \langle ('a \text{ var-denot} \times 'a \text{ fun-denot} \times 'a \text{ pre-denot}) \Rightarrow \text{sequent} \Rightarrow \text{bool} \rangle$  **where**  
 $\langle sc (E, F, G) (A, B) = ((\forall p [\in] A. \llbracket E, F, G \rrbracket p) \longrightarrow (\exists q [\in] B. \llbracket E, F, G \rrbracket q)) \rangle$

## 4.2 Substitution

**lemma** *add-env-semantic* [*simp*]:  $\langle \llbracket E, F \rrbracket ((t \circ s) n) = \llbracket E, F \rrbracket t \circ \lambda m. \llbracket E, F \rrbracket (s m) \circ n \rangle$   
**by** (*induct n simp-all*)

**lemma** *lift-lemma* [*simp*]:  $\langle \llbracket x \circ E, F \rrbracket (\text{lift-tm } t) = \llbracket E, F \rrbracket t \rangle$   
**by** (*induct t (auto cong: map-cong)*)

**lemma** *sub-tm-semantic* [*simp*]:  $\langle \llbracket E, F \rrbracket (\text{sub-tm } s t) = \llbracket \lambda n. \llbracket E, F \rrbracket (s n), F \rrbracket t \rangle$   
**by** (*induct t (auto cong: map-cong)*)

**lemma** *sub-fm-semantic* [*simp*]:  $\langle \llbracket E, F, G \rrbracket (\text{sub-fm } s p) = \llbracket \lambda n. \llbracket E, F \rrbracket (s n), F, G \rrbracket p \rangle$   
**by** (*induct p arbitrary: E s (auto cong: map-cong)*)

## 4.3 Variables

**lemma** *upd-vars-tm* [*simp*]:  $\langle n [\notin] \text{vars-tm } t \implies \llbracket E(n := x), F \rrbracket t = \llbracket E, F \rrbracket t \rangle$   
**by** (*induct t (auto cong: map-cong)*)

**lemma** *add-upd-commute* [*simp*]:  $\langle (y \circ E(n := x)) m = ((y \circ E)(\text{Suc } n := x)) m \rangle$   
**by** (*induct m simp-all*)

**lemma** *upd-vars-fm* [*simp*]:  $\langle \text{max-list } (\text{vars-fm } p) < n \implies \llbracket E(n := x), F, G \rrbracket p = \llbracket E, F, G \rrbracket p \rangle$

**proof** (*induct p arbitrary: E n*)

**case** (*Pre P ts*)

**moreover have**  $\langle \text{max-list } (\text{concat } (\text{map vars-tm } ts)) < n \rangle$

**using** *Pre.prem*s *max-list-concat* **by** *simp*

**then have**  $\langle n [\notin] \text{concat } (\text{map vars-tm } ts) \rangle$

**using** *max-list-in* **by** *blast*

**then have**  $\langle \forall t [\in] ts. n [\notin] \text{vars-tm } t \rangle$

**by** *simp*

**ultimately show** *?case*

**using** *upd-vars-tm* **by** (*metis list.map-cong semantics-fm.simps(2)*)

**next**

**case** (*Uni p*)

**have**  $\langle ?case = ((\forall y. \llbracket \lambda m. (y \circ E(n := x)) m, F, G \rrbracket p) = (\forall y. \llbracket y \circ E, F, G \rrbracket p)) \rangle$

**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**  $\langle \$ \rangle$  100) **where**

$\langle c \$ x \equiv \text{sum-encode } (c\ x) \rangle$

**lemma** *lt-sum-encode-Inr*:  $\langle n < \text{Inr } \$\ n \rangle$

**unfolding** *sum-encode-def* **by** *simp*

**lemma** *sum-prod-decode-lt* [*simp*]:  $\langle \text{sum-decode } n = \text{Inr } b \implies (x, y) = \text{prod-decode } b \implies y < \text{Suc } n \rangle$

**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*]:

$\langle \text{sum-decode } n = \text{Inr } b \implies (\text{Suc } x, y) = \text{prod-decode } b \implies x < \text{Suc } n \rangle$

**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*:  $\langle n [\in] ns \implies n < \text{list-encode } ns \rangle$

**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*]:

$\langle (x, \text{Suc } y) = \text{prod-decode } n \implies y' [\in] (\text{list-decode } y) \implies y' < n \rangle$

**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* ::  $\langle tm \Rightarrow nat \rangle$  **where**

$\langle \text{nat-of-tm } (\#n) = \text{prod-encode } (n, 0) \rangle$

|  $\langle \text{nat-of-tm } (\dagger f\ ts) = \text{prod-encode } (f, \text{Suc } (\text{list-encode } (\text{map } \text{nat-of-tm } ts))) \rangle$

**function** *tm-of-nat* ::  $\langle nat \Rightarrow tm \rangle$  **where**

$\langle \text{tm-of-nat } n = (\text{case } \text{prod-decode } n \text{ of}$

$(n, 0) \Rightarrow \#n$

|  $(f, \text{Suc } ts) \Rightarrow \dagger f (\text{map } \text{tm-of-nat } (\text{list-decode } ts)) \rangle$

**by** *pat-completeness auto*

**termination** **by** (*relation*  $\langle \text{measure } \text{id} \rangle$ ) *simp-all*

**lemma** *tm-nat*:  $\langle \text{tm-of-nat} (\text{nat-of-tm } t) = t \rangle$   
**by** (*induct* *t*) (*simp-all* *add*: *map-idI*)

**lemma** *surj-tm-of-nat*:  $\langle \text{surj } \text{tm-of-nat} \rangle$   
**unfolding** *surj-def* **using** *tm-nat* **by** *metis*

## 5.2 Formulas

**primrec** *nat-of-fm* ::  $\langle \text{fm} \Rightarrow \text{nat} \rangle$  **where**  
 $\langle \text{nat-of-fm } \perp = 0 \rangle$   
 $\langle \text{nat-of-fm } (\ddagger P \text{ } ts) = \text{Suc } (\text{Inl } \$ \text{prod-encode } (P, \text{list-encode } (\text{map } \text{nat-of-tm } ts))) \rangle$   
 $\langle \text{nat-of-fm } (p \longrightarrow q) = \text{Suc } (\text{Inr } \$ \text{prod-encode } (\text{Suc } (\text{nat-of-fm } p), \text{nat-of-fm } q)) \rangle$   
 $\langle \text{nat-of-fm } (\forall p) = \text{Suc } (\text{Inr } \$ \text{prod-encode } (0, \text{nat-of-fm } p)) \rangle$

**function** *fm-of-nat* ::  $\langle \text{nat} \Rightarrow \text{fm} \rangle$  **where**  
 $\langle \text{fm-of-nat } 0 = \perp \rangle$   
 $\langle \text{fm-of-nat } (\text{Suc } n) = (\text{case } \text{sum-decode } n \text{ of}$   
 $\quad \text{Inl } n \Rightarrow \text{let } (P, \text{ } ts) = \text{prod-decode } n \text{ in } \ddagger P (\text{map } \text{tm-of-nat } (\text{list-decode } ts))$   
 $\quad \text{Inr } n \Rightarrow (\text{case } \text{prod-decode } n \text{ of}$   
 $\quad \quad (\text{Suc } p, q) \Rightarrow \text{fm-of-nat } p \longrightarrow \text{fm-of-nat } q$   
 $\quad \quad (0, p) \Rightarrow \forall (\text{fm-of-nat } p))) \rangle$   
**by** *pat-completeness* *auto*  
**termination** **by** (*relation*  $\langle \text{measure } \text{id} \rangle$ ) *simp-all*

**lemma** *fm-nat*:  $\langle \text{fm-of-nat} (\text{nat-of-fm } p) = p \rangle$   
**using** *tm-nat* **by** (*induct* *p*) (*simp-all* *add*: *map-idI*)

**lemma** *surj-fm-of-nat*:  $\langle \text{surj } \text{fm-of-nat} \rangle$   
**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**  
 $\langle \text{idle-nat} \equiv 4294967295 \rangle$

**primrec** *nat-of-rule* ::  $\langle \text{rule} \Rightarrow \text{nat} \rangle$  **where**  
 $\langle \text{nat-of-rule } \text{Idle} = \text{Inl } \$ \text{prod-encode } (0, \text{idle-nat}) \rangle$   
 $\langle \text{nat-of-rule } (\text{Axiom } n \text{ } ts) = \text{Inl } \$ \text{prod-encode } (\text{Suc } n, \text{Suc } (\text{list-encode } (\text{map } \text{nat-of-tm } ts))) \rangle$   
 $\langle \text{nat-of-rule } \text{FlsL} = \text{Inl } \$ \text{prod-encode } (0, 0) \rangle$   
 $\langle \text{nat-of-rule } \text{FlsR} = \text{Inl } \$ \text{prod-encode } (0, \text{Suc } 0) \rangle$   
 $\langle \text{nat-of-rule } (\text{ImpL } p \text{ } q) = \text{Inr } \$ \text{prod-encode } (\text{Inl } \$ \text{nat-of-fm } p, \text{Inl } \$ \text{nat-of-fm } q) \rangle$   
 $\langle \text{nat-of-rule } (\text{ImpR } p \text{ } q) = \text{Inr } \$ \text{prod-encode } (\text{Inr } \$ \text{nat-of-fm } p, \text{nat-of-fm } q) \rangle$   
 $\langle \text{nat-of-rule } (\text{UniL } t \text{ } p) = \text{Inr } \$ \text{prod-encode } (\text{Inl } \$ \text{nat-of-tm } t, \text{Inr } \$ \text{nat-of-fm } p) \rangle$   
 $\langle \text{nat-of-rule } (\text{UniR } p) = \text{Inl } \$ \text{prod-encode } (\text{Suc } (\text{nat-of-fm } p), 0) \rangle$

```

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*:  $\langle \text{sset rules} = \text{UNIV} \rangle$   
**unfolding** *rules-def* **using** *UNIV-stream surj-rule-of-nat* .

**interpretation** *RuleSystem*  $\langle \lambda r s ss. \text{eff } r s = \text{Some } ss \rangle$  *rules UNIV*  
**by** *unfold-locales (auto simp: UNIV-rules intro: exI[of - Idle])*

**lemma** *per-rules'*:  
**assumes**  $\langle \text{enabled } r (A, B) \rangle \langle \neg \text{enabled } r (A', B') \rangle \langle \text{eff } r' (A, B) = \text{Some } ss' \rangle$   
 $\langle (A', B') \in ss' \rangle$   
**shows**  $\langle r' = r \rangle$   
**using** *assms* **by** (*cases r r' rule: rule.exhaust[case-product rule.exhaust]*)  
*(unfold enabled-def, auto split: if-splits)*

**lemma** *per-rules*:  $\langle \text{per } r \rangle$   
**unfolding** *per-def UNIV-rules* **using** *per-rules'* **by** *fast*

**interpretation** *PersistentRuleSystem*  $\langle \lambda r s ss. \text{eff } r s = \text{Some } ss \rangle$  *rules UNIV*  
**using** *per-rules* **by** *unfold-locales*

**definition**  $\langle \text{prover} \equiv \text{mkTree rules} \rangle$

**end**

## 7 Export

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

**definition**  $\langle \text{prove-sequent} \equiv i.\text{mkTree eff rules} \rangle$

**definition**  $\langle \text{prove} \equiv \lambda p. \text{prove-sequent} ([], [p]) \rangle$

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

**lemma** [*code*]:  $\langle \text{Stream.smember } x (y \#\#\ s) = (x = y \vee \text{Stream.smember } x s) \rangle$

**unfolding** *Stream.smember-def* **by** *auto*

**code-printing**

**constant** *the*  $\rightarrow (\text{Haskell}) (\lambda x \rightarrow \text{case } x \text{ of } \{ \text{Just } y \rightarrow y \})$   
| **constant** *Option.is-none*  $\rightarrow (\text{Haskell}) (\lambda x \rightarrow \text{case } x \text{ of } \{ \text{Just } y \rightarrow \text{False};$   
*Nothing} \rightarrow \text{True} \})*

**code-identifier**

**code-module** *Product-Type*  $\rightarrow (\text{Haskell})$  *Arith*  
| **code-module** *Orderings*  $\rightarrow (\text{Haskell})$  *Arith*  
| **code-module** *Arith*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *MaybeExt*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *List*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *Nat-Bijection*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *Syntax*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *Encoding*  $\rightarrow (\text{Haskell})$  *Prover*  
| **code-module** *HOL*  $\rightarrow (\text{Haskell})$  *Prover*

```

| code-module Set  $\rightarrow$  (Haskell) Prover
| code-module FSet  $\rightarrow$  (Haskell) Prover
| code-module Stream  $\rightarrow$  (Haskell) Prover
| code-module Fair-Stream  $\rightarrow$  (Haskell) Prover
| code-module Sum-Type  $\rightarrow$  (Haskell) Prover
| code-module Abstract-Completeness  $\rightarrow$  (Haskell) Prover
| code-module Export  $\rightarrow$  (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**  $\langle \text{eff } r (A, B) = \text{Some } ss \rangle \langle \forall A B. (A, B) \in | ss \longrightarrow (\forall (E :: - \Rightarrow 'a). \text{sc } (E, F, G) (A, B)) \rangle$

**shows**  $\langle \text{sc } (E, F, G) (A, B) \rangle$

**using** *assms*

**proof** (*induct r*  $\langle (A, B) \rangle$  *rule: eff.induct*)

**case** (*5 p q*)

**then have**  $\langle \text{sc } (E, F, G) (A [\div] (p \longrightarrow q), p \# B) \rangle \langle \text{sc } (E, F, G) (q \# A [\div] (p \longrightarrow q), B) \rangle$

**by** (*metis eff.simps(5) finsertCI option.inject option.simps(3)*)**+**

**then show** *?case*

**using** *5.prem(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.prem(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 § λm. (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.prem(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.prem(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* ‹λr 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:  $\langle \ddagger P \ ts \in A \implies \ddagger P \ ts \in B \implies \text{False} \rangle$  and
  FlsA:  $\langle \perp \notin A \rangle$  and
  ImpA:  $\langle p \longrightarrow q \in A \implies p \in B \vee q \in A \rangle$  and
  ImpB:  $\langle p \longrightarrow q \in B \implies p \in A \wedge q \in B \rangle$  and
  UniA:  $\langle \forall p \in A \implies \forall t. \langle t \rangle p \in A \rangle$  and
  UniB:  $\langle \forall p \in B \implies \exists t. \langle t \rangle p \in B \rangle$ 

abbreviation  $\langle M \ A \equiv \llbracket \#, \ddagger, \lambda P \ ts. \ddagger P \ ts \in A \rrbracket \rangle$ 

lemma id-tm [simp]:  $\langle (\#, \ddagger) \ t = t \rangle$ 
  by (induct t) (auto cong: map-cong)

lemma size-sub-fm [simp]:  $\langle \text{size} \ (\text{sub-fm} \ s \ p) = \text{size} \ p \rangle$ 
  by (induct p arbitrary: s) auto

theorem Hintikka-counter-model:
  assumes  $\langle \text{Hintikka} \ A \ B \rangle$ 
  shows  $\langle (p \in A \longrightarrow M \ A \ p) \wedge (p \in B \longrightarrow \neg M \ A \ p) \rangle$ 
proof (induct p rule: wf-induct [where r= $\langle \text{measure size} \rangle$ ])
  case 1
  then show ?case ..
next
  case (2 x)
  then show ?case
proof (cases x; safe del: notI)
  case Falsity
  show  $\langle \perp \in A \implies M \ A \ \perp \rangle \langle \perp \in B \implies \neg M \ A \ \perp \rangle$ 
  using Hintikka.FlsA assms by simp-all
next
  case (Pre P ts)
  show  $\langle \ddagger P \ ts \in A \implies M \ A \ (\ddagger P \ ts) \rangle \langle \ddagger P \ ts \in B \implies \neg M \ A \ (\ddagger P \ ts) \rangle$ 
  using Hintikka.Basic assms by (auto cong: map-cong)
next
  case (Imp p q)
  show  $\langle p \longrightarrow q \in A \implies M \ A \ (p \longrightarrow q) \rangle \langle p \longrightarrow q \in B \implies \neg M \ A \ (p \longrightarrow q) \rangle$ 
  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  $\langle \langle t \rangle p \in A \implies M \ A \ (\langle t \rangle p) \rangle \langle \langle t \rangle p \in B \implies \neg M \ A \ (\langle t \rangle p) \rangle$  for t
  using Uni 2 by (metis fm.size(8) in-measure lessI less-add-same-cancel1
size-sub-fm)+
  then show  $\langle \forall p \in A \implies M \ A \ (\forall p) \rangle \langle \forall p \in B \implies \neg M \ A \ (\forall p) \rangle$ 
  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*:  $\langle \text{sset} (\text{sdrop } n \ s) \subseteq \text{sset } s \rangle$   
**by** (*induct n arbitrary: s*) (*auto intro: stl-sset in-mono*)

**lemma** *epath-sdrop*:  $\langle \text{epath } \text{steps} \implies \text{epath} (\text{sdrop } n \ \text{steps}) \rangle$   
**by** (*induct n*) (*auto elim: epath.cases*)

**lemma** *eff-preserves-Pre*:  
**assumes**  $\langle \text{effStep} ((A, B), r) \ \text{ss} \rangle \langle (A', B') \mid \in \mid \ \text{ss} \rangle$   
**shows**  $\langle \ddagger P \ \text{ts} \mid \in \mid A \implies \ddagger P \ \text{ts} \mid \in \mid A' \rangle \langle \ddagger P \ \text{ts} \mid \in \mid B \implies \ddagger P \ \text{ts} \mid \in \mid B' \rangle$   
**using** *assms* **by** (*induct r*  $\langle (A, B) \rangle$  *rule: eff.induct*) (*auto split: if-splits*)

**lemma** *epath-eff*:  
**assumes**  $\langle \text{epath } \text{steps} \rangle \langle \text{effStep} (\text{shd } \text{steps}) \ \text{ss} \rangle$   
**shows**  $\langle \text{fst} (\text{shd} (\text{stl } \text{steps})) \mid \in \mid \ \text{ss} \rangle$   
**using** *assms* **by** (*auto elim: epath.cases*)

**abbreviation**  $\langle \text{lhs } s \equiv \text{fst} (\text{fst } s) \rangle$   
**abbreviation**  $\langle \text{rhs } s \equiv \text{snd} (\text{fst } s) \rangle$   
**abbreviation**  $\langle \text{lhsd } s \equiv \text{lhs} (\text{shd } s) \rangle$   
**abbreviation**  $\langle \text{rhsd } s \equiv \text{rhs} (\text{shd } s) \rangle$

**lemma** *epath-Pre-sdrop*:  
**assumes**  $\langle \text{epath } \text{steps} \rangle$  **shows**  
 $\langle \ddagger P \ \text{ts} \mid \in \mid \text{lhs} (\text{shd } \text{steps}) \implies \ddagger P \ \text{ts} \mid \in \mid \text{lhsd} (\text{sdrop } m \ \text{steps}) \rangle$   
 $\langle \ddagger P \ \text{ts} \mid \in \mid \text{rhs} (\text{shd } \text{steps}) \implies \ddagger P \ \text{ts} \mid \in \mid \text{rhsd} (\text{sdrop } m \ \text{steps}) \rangle$   
**using** *assms* *eff-preserves-Pre*  
**by** (*induct m arbitrary: steps*) (*simp;metis (no-types, lifting) epath.cases surjective-pairing*)+

**lemma** *Saturated-sdrop*:  
**assumes**  $\langle \text{Saturated } \text{steps} \rangle$   
**shows**  $\langle \text{Saturated} (\text{sdrop } n \ \text{steps}) \rangle$   
**using** *assms* **unfolding** *Saturated-def saturated-def* **by** (*simp add: alw-iff-sdrop*)

**definition** *treeA* ::  $\langle (\text{sequent} \times \text{rule}) \ \text{stream} \Rightarrow \text{fm } \text{set} \rangle$  **where**  
 $\langle \text{treeA } \text{steps} \equiv \bigcup s \in \text{sset } \text{steps}. \ \text{set} (\text{lhs } s) \rangle$

**definition** *treeB* ::  $\langle (\text{sequent} \times \text{rule}) \ \text{stream} \Rightarrow \text{fm } \text{set} \rangle$  **where**  
 $\langle \text{treeB } \text{steps} \equiv \bigcup s \in \text{sset } \text{steps}. \ \text{set} (\text{rhs } s) \rangle$

**lemma** *treeA-snth*:  $\langle p \in \text{treeA } \text{steps} \implies \exists n. \ p \mid \in \mid \text{lhsd} (\text{sdrop } n \ \text{steps}) \rangle$   
**unfolding** *treeA-def* **using** *sset-range[of steps]* **by** *simp*

**lemma** *treeB-snth*:  $\langle p \in \text{treeB } \text{steps} \implies \exists n. \ p \mid \in \mid \text{rhsd} (\text{sdrop } n \ \text{steps}) \rangle$   
**unfolding** *treeB-def* **using** *sset-range[of steps]* **by** *simp*

**lemma** *treeA-sdrop*:  $\langle \text{treeA} (\text{sdrop } n \ \text{steps}) \subseteq \text{treeA } \text{steps} \rangle$   
**unfolding** *treeA-def* **by** (*induct n*) (*simp,metis SUP-subset-mono order-refl*)

*sset-sdrop*)

**lemma** *treeB-sdrop*:  $\langle \text{treeB } (\text{sdrop } n \text{ steps}) \subseteq \text{treeB steps} \rangle$   
  **unfolding** *treeB-def* **by** (*induct n*) (*simp, metis SUP-subset-mono order-refl sset-sdrop*)

**lemma** *enabled-ex-taken*:  
  **assumes**  $\langle \text{epath steps} \rangle \langle \text{Saturated steps} \rangle \langle \text{enabled } r \text{ (fst (shd steps))} \rangle$   
  **shows**  $\langle \exists k. \text{takenAtStep } r \text{ (shd (sdrop } k \text{ steps))} \rangle$   
  **using** *assms unfolding Saturated-def saturated-def UNIV-rules* **by** (*auto simp: ev-iff-sdrop*)

**lemma** *Hintikka-epath*:  
  **assumes**  $\langle \text{epath steps} \rangle \langle \text{Saturated steps} \rangle$   
  **shows**  $\langle \text{Hintikka (treeA steps) (treeB steps)} \rangle$

**proof**

**fix** *P ts*  
  **assume**  $\langle \dagger P \text{ ts} \in \text{treeA steps} \rangle$   
  **then obtain** *m* **where** *m*:  $\langle \dagger P \text{ ts} [\in] \text{lhsd (sdrop } m \text{ steps)} \rangle$   
    **using** *treeA-snth* **by** *auto*

**assume**  $\langle \dagger P \text{ ts} \in \text{treeB steps} \rangle$   
  **then obtain** *k* **where** *k*:  $\langle \dagger P \text{ ts} [\in] \text{rhsd (sdrop } k \text{ steps)} \rangle$   
    **using** *treeB-snth* **by** *auto*

**let** *?j* =  $\langle m + k \rangle$   
  **let** *?jstep* =  $\langle \text{shd (sdrop } ?j \text{ steps)} \rangle$

**have**  $\langle \dagger P \text{ ts} [\in] \text{lhs } ?jstep \rangle$   
    **using** *assms m epath-sdrop epath-Pre-sdrop* **by** (*metis (no-types, lifting) sdrop-add*)  
  **moreover have**  $\langle \dagger P \text{ ts} [\in] \text{rhs } ?jstep \rangle$   
    **using** *assms k epath-sdrop epath-Pre-sdrop* **by** (*metis (no-types, lifting) add commute sdrop-add*)

**ultimately have**  $\langle \text{enabled (Axiom } P \text{ ts) (fst } ?jstep) \rangle$   
    **unfolding** *enabled-def* **by** (*metis eff.simps(2) prod.exhaust-sel*)  
  **then obtain** *j'* **where**  $\langle \text{takenAtStep (Axiom } P \text{ ts) (shd (sdrop } j' \text{ steps))} \rangle$   
    **using** *enabled-ex-taken[OF epath-sdrop[OF assms(1)] Saturated-sdrop[OF assms(2)]]*

**by** *auto*

**then have**  $\langle \text{eff (snd (shd (sdrop } j' \text{ steps))) (fst (shd (sdrop } j' \text{ steps)))} = \text{None} \rangle$   
    **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**  $\langle \perp \notin \text{treeA steps} \rangle$

**proof**

**assume**  $\langle \perp \in \text{treeA steps} \rangle$   
  **then have**  $\langle \exists j. \text{enabled FlsL (fst (shd (sdrop } j \text{ steps)))} \rangle$

```

    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 ‹take-
nAtStep - (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 ‹take-
nAtStep - (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  $\langle p \in \text{treeA } (stl ?s) \wedge q \in \text{treeB } (stl ?s) \rangle$ 
  unfolding treeA-def treeB-def by (meson UN-I shd-sset)
then show  $\langle p \in \text{treeA steps} \wedge q \in \text{treeB steps} \rangle$ 
  using treeA-sdrop treeB-sdrop by (metis sdrop-simps(2) subsetD)
next
  fix p
  assume *:  $\langle \forall p \in \text{treeA steps} \rangle$ 
  show  $\langle \forall t. \langle t \rangle p \in \text{treeA steps} \rangle$ 
  proof
    fix t
    from * have  $\langle \exists k. \text{enabled } (UniL t p) (fst (shd (sdrop k steps))) \rangle$ 
    unfolding enabled-def using treeA-snth by (metis eff.simps(7) prod.exhaust-sel sdrop-simps(1))
    then obtain j where  $\langle \text{takenAtStep } (UniL t p) (shd (sdrop j steps)) \rangle$  (is  $\langle \text{takenAtStep} - (shd ?s) \rangle$ 
      using enabled-ex-taken[OF epath-sdrop[OF assms(1)] Saturated-sdrop[OF assms(2)]] by auto
    then have  $\langle fst (shd (stl ?s)) \in \{ | \langle t \rangle p \# lhsd ?s, rhsd ?s \} \rangle$ 
      using assms(1) epath-sdrop epath-eff
    by (metis (no-types, lifting) eff.simps(7) epath.cases option.distinct(1) prod.collapse)
    then have  $\langle \langle t \rangle p \in lhs (shd (stl ?s)) \rangle$ 
      by auto
    then have  $\langle \langle t \rangle p \in \text{treeA } (stl ?s) \rangle$ 
      unfolding treeA-def by (meson UN-I shd-sset)
    then show  $\langle \langle t \rangle p \in \text{treeA steps} \rangle$ 
      using treeA-sdrop by (metis sdrop-simps(2) subsetD)
    qed
  next
    fix p
    assume *:  $\langle \forall p \in \text{treeB steps} \rangle$ 
    then have  $\langle \exists k. \text{enabled } (UniR p) (fst (shd (sdrop k steps))) \rangle$ 
    unfolding enabled-def using treeB-snth by (metis eff.simps(8) prod.exhaust-sel sdrop-simps(1))
    then obtain j where  $\langle \text{takenAtStep } (UniR p) (shd (sdrop j steps)) \rangle$  (is  $\langle \text{takenAtStep} - (shd ?s) \rangle$ 
      using enabled-ex-taken[OF epath-sdrop[OF assms(1)] Saturated-sdrop[OF assms(2)]]
    by auto
    then have  $\langle fst (shd (stl ?s)) \in \{ | (lhsd ?s, \langle \#(\text{fresh } (lhsd ?s @ rhsd ?s)) p \# rhsd ?s [\div] \forall p) \} \rangle$ 
      using assms(1) epath-sdrop epath-eff
    by (metis (no-types, lifting) eff.simps(8) epath.cases option.distinct(1) prod.collapse)
    then have  $\langle \exists t. \langle t \rangle p \in rhs (shd (stl ?s)) \rangle$ 
      by auto
    then have  $\langle \exists t. \langle t \rangle p \in \text{treeB } (stl ?s) \rangle$ 
      unfolding treeB-def by (meson UN-I shd-sset)
    then show  $\langle \exists t. \langle t \rangle p \in \text{treeB steps} \rangle$ 
      using treeB-sdrop by (metis sdrop-simps(2) subsetD)
    qed

```

### 9.3 Completeness

**lemma** *fair-stream-rules*:  $\langle \text{Fair-Stream.fair rules} \rangle$   
**unfolding** *rules-def* **using** *fair-stream surj-rule-of-nat* .

**lemma** *fair-rules*:  $\langle \text{fair rules} \rangle$   
**using** *fair-stream-rules* **unfolding** *Fair-Stream.fair-def fair-def alw-iff-sdrop ev-holds-sset*  
**by**  $(\text{metis dual-order.refl le-Suc-ex sdrop-snth snth-sset})$

**lemma** *epath-prover*:  
**fixes**  $A B :: \langle \text{fm list} \rangle$   
**defines**  $\langle t \equiv \text{prover } (A, B) \rangle$   
**shows**  $\langle \text{fst } (\text{root } t) = (A, B) \wedge \text{wf } t \wedge \text{tfinite } t \rangle \vee$   
 $\langle \exists \text{ steps. } \text{fst } (\text{shd } \text{steps}) = (A, B) \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps} \rangle$  **(is**  $\langle ?A \vee$   
 $?B \rangle$   
**proof** –  
**{** **assume**  $\langle \neg ?A \rangle$   
**with** *assms* **have**  $\langle \neg \text{tfinite } (\text{mkTree rules } (A, B)) \rangle$   
**unfolding** *prover-def* **using** *wf-mkTree fair-rules* **by** *simp*  
**then obtain** *steps* **where**  $\langle \text{ipath } (\text{mkTree rules } (A, B)) \text{ steps} \rangle$  **using** *Konig* **by**  
*blast*  
**with** *assms* **have**  $\langle \text{fst } (\text{shd } \text{steps}) = (A, B) \wedge \text{epath } \text{steps} \wedge \text{Saturated } \text{steps} \rangle$   
**by**  $(\text{metis } (\text{no-types, lifting}) \text{ fair-rules UNIV-I fst-conv ipath.cases}$   
 $\text{ipath-mkTree-Saturated mkTree.simps(1) wf-ipath-epath wf-mkTree})$   
**then have**  $?B$  **by** *blast*  
**}**  
**then show** *thesis* **by** *blast*  
**qed**

**lemma** *epath-countermodel*:  
**assumes**  $\langle \text{fst } (\text{shd } \text{steps}) = (A, B) \rangle \langle \text{epath } \text{steps} \rangle \langle \text{Saturated } \text{steps} \rangle$   
**shows**  $\langle \exists (E :: - \Rightarrow \text{tm}) F G. \neg \text{sc } (E, F, G) (A, B) \rangle$   
**proof** –  
**have**  $\langle \text{Hintikka } (\text{treeA } \text{steps}) (\text{treeB } \text{steps}) \rangle$  **(is**  $\langle \text{Hintikka } ?A ?B \rangle$   
**using** *assms Hintikka-epath assms* **by** *simp*  
**moreover have**  $\langle \forall p [\in] A. p \in ?A \rangle \langle \forall p [\in] B. p \in ?B \rangle$   
**using** *assms shd-sset* **unfolding** *treeA-def treeB-def* **by** *fastforce+*  
**ultimately have**  $\langle \forall p [\in] A. M ?A p \rangle \langle \forall p [\in] B. \neg M ?A p \rangle$   
**using** *Hintikka-counter-model assms* **by** *blast+*  
**then show** *thesis*  
**by** *auto*  
**qed**

**theorem** *prover-completeness*:  
**assumes**  $\langle \forall (E :: - \Rightarrow \text{tm}) F G. \text{sc } (E, F, G) (A, B) \rangle$   
**defines**  $\langle t \equiv \text{prover } (A, B) \rangle$   
**shows**  $\langle \text{fst } (\text{root } t) = (A, B) \wedge \text{wf } t \wedge \text{tfinite } t \rangle$   
**using** *assms epath-prover epath-countermodel* **by** *blast*

**corollary**

```

assumes  $\langle \forall (E :: - \Rightarrow tm) F G. \llbracket E, F, G \rrbracket p \rangle$ 
defines  $\langle t \equiv prover (\ [], [p] ) \rangle$ 
shows  $\langle fst (root\ t) = (\ [], [p] ) \wedge wf\ t \wedge tfinite\ t \rangle$ 
using assms prover-completeness by simp

```

**end**

## 10 Result

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

**theorem** *prover-soundness-completeness*:

```

fixes  $A\ B :: \langle fm\ list \rangle$ 
defines  $\langle t \equiv prover (A, B) \rangle$ 
shows  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\forall (E :: - \Rightarrow tm) F G. sc\ (E, F, G)\ (A, B)) \rangle$ 
using assms prover-soundness prover-completeness unfolding prover-def by fastforce

```

**corollary**

```

fixes  $p :: fm$ 
defines  $\langle t \equiv prover (\ [], [p] ) \rangle$ 
shows  $\langle tfinite\ t \wedge wf\ t \longleftrightarrow (\forall (E :: - \Rightarrow tm) F G. \llbracket E, F, G \rrbracket p) \rangle$ 
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](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](https://isa-afp.org/entries/FOL_Seq_Calc2.html), Formal proof development.