

# 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>5</b>
3.1	Terms and Formulas . . . . .	5
3.1.1	Substitution . . . . .	6
3.1.2	Variables . . . . .	6
3.2	Rules . . . . .	7
<b>4</b>	<b>Semantics</b>	<b>7</b>
4.1	Definition . . . . .	7
4.2	Substitution . . . . .	8
4.3	Variables . . . . .	8
<b>5</b>	<b>Encoding</b>	<b>8</b>
5.1	Terms . . . . .	9
5.2	Formulas . . . . .	9
5.3	Rules . . . . .	10
<b>6</b>	<b>Prover</b>	<b>11</b>
<b>7</b>	<b>Export</b>	<b>12</b>
<b>8</b>	<b>Soundness</b>	<b>13</b>
<b>9</b>	<b>Completeness</b>	<b>13</b>
9.1	Hintikka Counter Model . . . . .	13
9.2	Escape Paths Form Hintikka Sets . . . . .	14
9.3	Completeness . . . . .	15
<b>10</b>	<b>Result</b>	<b>16</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**

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

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

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

### **syntax**

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

### **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**

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

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

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

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

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

**lemma**  $\text{all-ex-upt-lists}$ :  $\langle \exists n \geq m. x \in \text{set } (\text{upt-lists} !! n) \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{upt-lists-ne}$ :  $\langle \forall xs \in \text{sset } \text{upt-lists}. xs \neq [] \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{sset-flat-stl}$ :  $\langle \text{sset } (\text{flat } (\text{stl } s)) \subseteq \text{sset } (\text{flat } s) \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{flat-snth-nth}$ :  
**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**  $\langle \exists n' \geq n. x = \text{flat } s !! n' \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{all-ex-fair-nats}$ :  $\langle \exists n \geq m. \text{fair-nats} !! n = x \rangle$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{fair-surj}$ :  
**assumes**  $\langle \text{surj } f \rangle$   
**shows**  $\langle \text{fair } (\text{smap } f \text{ fair-nats}) \rangle$   
 $\langle \text{proof} \rangle$

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

**theorem**  $\text{fair-stream}$ :  $\langle \text{surj } f \implies \text{fair } (\text{fair-stream } f) \rangle$   
 $\langle \text{proof} \rangle$

**theorem**  $\text{UNIV-stream}$ :  $\langle \text{surj } f \implies \text{sset } (\text{fair-stream } f) = \text{UNIV} \rangle$   
 $\langle \text{proof} \rangle$

**end**

### 3 Syntax

**theory**  $\text{Syntax}$  **imports**  $\text{List-Syntax}$  **begin**

#### 3.1 Terms and Formulas

**datatype**  $tm$   
 $= \text{Var } \text{nat } (\langle \# \rangle)$

| *Fun nat*  $\langle tm \ list \rangle$  ( $\langle \dagger \rangle$ )

**datatype** *fm*  
 = *Falsity* ( $\langle \perp \rangle$ )  
 | *Pre nat*  $\langle tm \ list \rangle$  ( $\langle \ddagger \rangle$ )  
 | *Imp fm fm* (**infixr**  $\langle \longrightarrow \rangle$  55)  
 | *Uni fm* ( $\langle \forall \rangle$ )

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

### 3.1.1 Substitution

**primrec** *add-env* ::  $\langle 'a \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \rangle$  (**infix**  $\langle \circledast \rangle$  0) **where**  
 $\langle (t \circledast s) \ 0 = t \rangle$   
 |  $\langle (t \circledast s) \ (Suc \ n) = s \ n \rangle$

**primrec** *lift-tm* ::  $\langle tm \Rightarrow tm \rangle$  **where**  
 $\langle lift-tm \ (\#n) = \#(n+1) \rangle$   
 |  $\langle lift-tm \ (\ddagger \ ts) = \ddagger \ (map \ lift-tm \ ts) \rangle$

**primrec** *sub-tm* ::  $\langle (nat \Rightarrow tm) \Rightarrow tm \Rightarrow tm \rangle$  **where**  
 $\langle sub-tm \ s \ (\#n) = s \ n \rangle$   
 |  $\langle sub-tm \ s \ (\ddagger \ ts) = \ddagger \ (map \ (sub-tm \ s) \ ts) \rangle$

**primrec** *sub-fm* ::  $\langle (nat \Rightarrow tm) \Rightarrow fm \Rightarrow fm \rangle$  **where**  
 $\langle sub-fm \ - \ \perp = \perp \rangle$   
 |  $\langle sub-fm \ s \ (\ddagger \ P \ ts) = \ddagger \ P \ (map \ (sub-tm \ s) \ ts) \rangle$   
 |  $\langle sub-fm \ s \ (p \longrightarrow q) = sub-fm \ s \ p \longrightarrow sub-fm \ s \ q \rangle$   
 |  $\langle sub-fm \ s \ (\forall \ p) = \forall \ (sub-fm \ (\#0 \ \circledast \ \lambda n. \ lift-tm \ (s \ n)) \ p) \rangle$

**abbreviation** *inst-single* ::  $\langle tm \Rightarrow fm \Rightarrow fm \rangle$  ( $\langle \langle - \rangle \rangle$ ) **where**  
 $\langle \langle t \rangle \equiv sub-fm \ (t \ \circledast \ \#) \rangle$

### 3.1.2 Variables

**primrec** *vars-tm* ::  $\langle tm \Rightarrow nat \ list \rangle$  **where**  
 $\langle vars-tm \ (\#n) = [n] \rangle$   
 |  $\langle vars-tm \ (\ddagger \ ts) = concat \ (map \ vars-tm \ ts) \rangle$

**primrec** *vars-fm* ::  $\langle fm \Rightarrow nat \ list \rangle$  **where**  
 $\langle vars-fm \ \perp = [] \rangle$   
 |  $\langle vars-fm \ (\ddagger \ ts) = concat \ (map \ vars-tm \ ts) \rangle$   
 |  $\langle vars-fm \ (p \longrightarrow q) = vars-fm \ p \ @ \ vars-fm \ q \rangle$   
 |  $\langle vars-fm \ (\forall \ p) = vars-fm \ p \rangle$

**primrec** *max-list* ::  $\langle nat \ list \Rightarrow nat \rangle$  **where**  
 $\langle max-list \ [] = 0 \rangle$   
 |  $\langle max-list \ (x \ # \ xs) = max \ x \ (max-list \ xs) \rangle$

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

⟨proof⟩

**lemma** *max-list-concat*: ⟨ $xs \in xss \implies \text{max-list } xs \leq \text{max-list } (\text{concat } xss)$ ⟩  
⟨proof⟩

**lemma** *max-list-in*: ⟨ $\text{max-list } xs < n \implies n \notin xs$ ⟩  
⟨proof⟩

**definition** *vars-fms* :: ⟨ $\text{fm list} \Rightarrow \text{nat list}$ ⟩ **where**  
⟨ $\text{vars-fms } A \equiv \text{concat } (\text{map } \text{vars-fm } A)$ ⟩

**lemma** *vars-fms-member*: ⟨ $p \in A \implies \text{vars-fm } p \subseteq \text{vars-fms } A$ ⟩  
⟨proof⟩

**lemma** *max-list-mono*: ⟨ $A \subseteq B \implies \text{max-list } A \leq \text{max-list } B$ ⟩  
⟨proof⟩

**lemma** *max-list-vars-fms*:  
**assumes** ⟨ $\text{max-list } (\text{vars-fms } A) \leq n$ ⟩ ⟨ $p \in A$ ⟩  
**shows** ⟨ $\text{max-list } (\text{vars-fm } p) \leq n$ ⟩  
⟨proof⟩

**definition** *fresh* :: ⟨ $\text{fm list} \Rightarrow \text{nat}$ ⟩ **where**  
⟨ $\text{fresh } A \equiv \text{Suc } (\text{max-list } (\text{vars-fms } A))$ ⟩

## 3.2 Rules

**datatype** *rule* =  
Idle | Axiom nat ⟨ $\text{tm list}$ ⟩ | 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* = ⟨ $\text{nat} \Rightarrow 'a$ ⟩  
**type-synonym** 'a *fun-denot* = ⟨ $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a$ ⟩  
**type-synonym** 'a *pre-denot* = ⟨ $\text{nat} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ ⟩

**primrec** *semantics-tm* :: ⟨'a *var-denot*  $\Rightarrow$  'a *fun-denot*  $\Rightarrow$  *tm*  $\Rightarrow$  'a⟩ (⟨[-, -]⟩)  
**where**  
| ⟨([E, F]) (#n) = E n⟩  
| ⟨([E, F]) (†f ts) = F f (map ([E, F]) ts)⟩

**primrec** *semantics-fm* ::  $\langle 'a \text{ var-denot} \Rightarrow 'a \text{ fun-denot} \Rightarrow 'a \text{ pre-denot} \Rightarrow \text{fm} \Rightarrow \text{bool} \rangle$

$\langle \llbracket -, -, - \rrbracket \rangle$  **where**  
 $\langle \llbracket -, -, - \rrbracket \perp = \text{False} \rangle$   
 $\mid \langle \llbracket E, F, G \rrbracket (\dagger P \text{ ts}) = G \ P \ (\text{map } \llbracket E, F \rrbracket \text{ ts}) \rangle$   
 $\mid \langle \llbracket E, F, G \rrbracket (p \longrightarrow q) = (\llbracket E, F, G \rrbracket p \longrightarrow \llbracket E, F, G \rrbracket q) \rangle$   
 $\mid \langle \llbracket E, F, G \rrbracket (\forall p) = (\forall x. \llbracket x \circ E, F, G \rrbracket p) \rangle$

**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**  
 $\langle \text{sc } (E, F, G) (A, B) = ((\forall p [\in] A. \llbracket E, F, G \rrbracket p) \longrightarrow (\exists q [\in] B. \llbracket E, F, G \rrbracket q)) \rangle$

## 4.2 Substitution

**lemma** *add-env-semantics [simp]*:  $\langle \llbracket E, F \rrbracket ((t \circ s) \ n) = (\llbracket E, F \rrbracket t \circ \lambda m. \llbracket E, F \rrbracket (s \ m)) \ n \rangle$   
 $\langle \text{proof} \rangle$

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

**lemma** *sub-tm-semantics [simp]*:  $\langle \llbracket E, F \rrbracket (\text{sub-tm } s \ t) = (\lambda n. \llbracket E, F \rrbracket (s \ n), F) \ t \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *sub-fm-semantics [simp]*:  $\langle \llbracket E, F, G \rrbracket (\text{sub-fm } s \ p) = \llbracket \lambda n. \llbracket E, F \rrbracket (s \ n), F, G \rrbracket p \rangle$   
 $\langle \text{proof} \rangle$

## 4.3 Variables

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

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

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

**end**

## 5 Encoding

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

**abbreviation** *infix-sum-encode (infixr <\$> 100) where*  
 $\langle c \ \$ \ x \equiv \text{sum-encode } (c \ x) \rangle$

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

⟨proof⟩

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

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

**lemma** *lt-list-encode*: ⟨ $n \in ns \implies n < \text{list-encode } ns$ ⟩  
⟨proof⟩

**lemma** *prod-Suc-list-decode-lt* [simp]:  
⟨ $(x, \text{Suc } y) = \text{prod-decode } n \implies y' \in (\text{list-decode } y) \implies y' < n$ ⟩  
⟨proof⟩

## 5.1 Terms

**primrec** *nat-of-tm* :: ⟨ $tm \Rightarrow nat$ ⟩ **where**  
⟨*nat-of-tm*  $(\#n) = \text{prod-encode } (n, 0)$ ⟩  
| ⟨*nat-of-tm*  $(\dagger f \ ts) = \text{prod-encode } (f, \text{Suc } (\text{list-encode } (\text{map } \text{nat-of-tm } \ ts)))$ ⟩

**function** *tm-of-nat* :: ⟨ $nat \Rightarrow tm$ ⟩ **where**  
⟨*tm-of-nat*  $n = (\text{case } \text{prod-decode } n \text{ of}$   
   $(n, 0) \Rightarrow \#n$   
  |  $(f, \text{Suc } \ ts) \Rightarrow \dagger f (\text{map } \text{tm-of-nat } (\text{list-decode } \ ts))$ )⟩  
⟨proof⟩

**termination** ⟨proof⟩

**lemma** *tm-nat*: ⟨*tm-of-nat*  $(\text{nat-of-tm } t) = t$ ⟩  
⟨proof⟩

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

## 5.2 Formulas

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

**function** *fm-of-nat* :: ⟨ $nat \Rightarrow fm$ ⟩ **where**  
⟨*fm-of-nat*  $0 = \perp$ ⟩  
| ⟨*fm-of-nat*  $(\text{Suc } n) = (\text{case } \text{sum-decode } n \text{ of}$   
   $\text{Inl } n \Rightarrow \text{let } (P, \ ts) = \text{prod-decode } n \text{ in } \dagger P (\text{map } \text{tm-of-nat } (\text{list-decode } \ ts))$   
  |  $\text{Inr } n \Rightarrow (\text{case } \text{prod-decode } n \text{ of}$   
     $(\text{Suc } p, q) \Rightarrow \text{fm-of-nat } p \longrightarrow \text{fm-of-nat } q$ )

$\langle (0, p) \Rightarrow \forall (fm\text{-of-nat } p) \rangle$   
 $\langle proof \rangle$   
**termination**  $\langle proof \rangle$

**lemma** *fm-nat*:  $\langle fm\text{-of-nat } (nat\text{-of-fm } p) = p \rangle$   
 $\langle proof \rangle$

**lemma** *surj-fm-of-nat*:  $\langle surj\text{-of-nat} \rangle$   
 $\langle proof \rangle$

### 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**  
 $\langle idle\text{-nat} \equiv 4294967295 \rangle$

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

**fun** *rule-of-nat* ::  $\langle nat \Rightarrow rule \rangle$  **where**  
 $\langle rule\text{-of-nat } n = (case\ sum\text{-decode } n\ of$   
 $\quad Inl\ n \Rightarrow (case\ prod\text{-decode } n\ of$   
 $\quad\quad (0, 0) \Rightarrow FlsL$   
 $\quad\quad | (0, Suc\ 0) \Rightarrow FlsR$   
 $\quad\quad | (0, n2) \Rightarrow if\ n2 = idle\text{-nat} then\ Idle\ else$   
 $\quad\quad\quad let\ (p, q) = prod\text{-decode } n2\ in\ ImpR\ (fm\text{-of-nat } p)\ (fm\text{-of-nat } q)$   
 $\quad\quad | (Suc\ n, Suc\ ts) \Rightarrow Axiom\ n\ (map\ tm\text{-of-nat } (list\text{-decode } ts))$   
 $\quad\quad | (Suc\ p, 0) \Rightarrow UniR\ (fm\text{-of-nat } p)$   
 $\quad Inr\ n \Rightarrow (let\ (n1, n2) = prod\text{-decode } n\ in$   
 $\quad case\ sum\text{-decode } n1\ of$   
 $\quad\quad Inl\ n1 \Rightarrow (case\ sum\text{-decode } n2\ of$   
 $\quad\quad\quad Inl\ q \Rightarrow ImpL\ (fm\text{-of-nat } n1)\ (fm\text{-of-nat } q)$   
 $\quad\quad\quad | Inr\ p \Rightarrow UniL\ (tm\text{-of-nat } n1)\ (fm\text{-of-nat } p)$   
 $\quad\quad\quad | Inr\ p \Rightarrow ImpR\ (fm\text{-of-nat } p)\ (fm\text{-of-nat } n2))) \rangle$

**lemma** *rule-nat*:  $\langle rule\text{-of-nat } (nat\text{-of-rule } r) = r \rangle$   
 $\langle proof \rangle$

**lemma** *surj-rule-of-nat*:  $\langle \text{surj rule-of-nat} \rangle$   
 $\langle \text{proof} \rangle$

**end**

## 6 Prover

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

**function** *eff* ::  $\langle \text{rule} \Rightarrow \text{sequent} \Rightarrow (\text{sequent fset}) \text{ option} \rangle$  **where**  
 $\langle \text{eff Idle } (A, B) = \text{Some } \{| (A, B) |\} \rangle$   
 $| \langle \text{eff } (\text{Axiom } P \text{ ts}) (A, B) = (\text{if } \ddagger P \text{ ts } [\in] A \wedge \ddagger P \text{ ts } [\in] B \text{ then } \text{Some } \{|\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff FlsL } (A, B) = (\text{if } \perp [\in] A \text{ then } \text{Some } \{|\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff FlsR } (A, B) = (\text{if } \perp [\in] B \text{ then } \text{Some } \{| (A, B [\div] \perp) |\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff } (\text{ImpL } p \ q) (A, B) = (\text{if } (p \longrightarrow q) [\in] A \text{ then } \text{Some } \{| (A [\div] (p \longrightarrow q), p \# B), (q \# A [\div] (p \longrightarrow q), B) |\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff } (\text{ImpR } p \ q) (A, B) = (\text{if } (p \longrightarrow q) [\in] B \text{ then } \text{Some } \{| (p \# A, q \# B [\div] (p \longrightarrow q)) |\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff } (\text{UniL } t \ p) (A, B) = (\text{if } \forall p [\in] A \text{ then } \text{Some } \{| (\langle t \rangle p \# A, B) |\} \text{ else } \text{None}) \rangle$   
 $| \langle \text{eff } (\text{UniR } p) (A, B) = (\text{if } \forall p [\in] B \text{ then } \text{Some } \{| (A, \langle \#(\text{fresh } (A \ @ \ B)) \rangle p \# B [\div] \forall p) |\} \text{ else } \text{None}) \rangle$   
 $\langle \text{proof} \rangle$

**termination**  $\langle \text{proof} \rangle$

**definition** *rules* ::  $\langle \text{rule stream} \rangle$  **where**  
 $\langle \text{rules} \equiv \text{fair-stream rule-of-nat} \rangle$

**lemma** *UNIV-rules*:  $\langle \text{sset rules} = \text{UNIV} \rangle$   
 $\langle \text{proof} \rangle$

**interpretation** *RuleSystem*  $\langle \lambda r \ s \ ss. \ \text{eff } r \ s = \text{Some } ss \rangle$  *rules* *UNIV*  
 $\langle \text{proof} \rangle$

**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**  $\langle r' = r \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *per-rules*:  $\langle \text{per } r \rangle$   
 $\langle \text{proof} \rangle$

**interpretation** *PersistentRuleSystem*  $\langle \lambda r \ s \ ss. \ \text{eff } r \ s = \text{Some } ss \rangle$  *rules* *UNIV*  
 $\langle \text{proof} \rangle$

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

end

## 7 Export

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

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

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

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

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

**code-printing**

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

**code-identifier**

**code-module** *Product-Type*  $\rightarrow (Haskell)\ \text{Arith}$   
| **code-module** *Orderings*  $\rightarrow (Haskell)\ \text{Arith}$   
| **code-module** *Arith*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *MaybeExt*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *List*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Nat-Bijection*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Syntax*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Encoding*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *HOL*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Set*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *FSet*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Stream*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Fair-Stream*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Sum-Type*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Abstract-Completeness*  $\rightarrow (Haskell)\ \text{Prover}$   
| **code-module** *Export*  $\rightarrow (Haskell)\ \text{Prover}$

**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)>
<proof>

```

**interpretation** *Soundness* <λr s ss. eff r s = Some ss> *rules UNIV sc*  
<proof>

**theorem** *prover-soundness*:

```

assumes <tfinite t> and <wf t>
shows <sc (E, F, G) (fst (root t))>
<proof>

```

end

## 9 Completeness

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

### 9.1 Hintikka Counter Model

**locale** *Hintikka* =

**fixes** *A B* :: <fm set>

**assumes**

*Basic*: <⊥ P ts ∈ 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**  $\langle M A \equiv \llbracket \#, \dagger, \lambda P ts. \ddagger P ts \in A \rrbracket \rangle$

**lemma** *id-tm* [*simp*]:  $\langle (\#, \dagger) t = t \rangle$   
 $\langle proof \rangle$

**lemma** *size-sub-fm* [*simp*]:  $\langle size (sub-fm s p) = size p \rangle$   
 $\langle proof \rangle$

**theorem** *Hintikka-counter-model*:

**assumes**  $\langle Hintikka A B \rangle$   
**shows**  $\langle (p \in A \longrightarrow M A p) \wedge (p \in B \longrightarrow \neg M A p) \rangle$   
 $\langle proof \rangle$

## 9.2 Escape Paths Form Hintikka Sets

**lemma** *sset-sdrop*:  $\langle sset (sdrop n s) \subseteq sset s \rangle$   
 $\langle proof \rangle$

**lemma** *epath-sdrop*:  $\langle epath steps \implies epath (sdrop n steps) \rangle$   
 $\langle proof \rangle$

**lemma** *eff-preserves-Pre*:

**assumes**  $\langle effStep ((A, B), r) ss \rangle \langle (A', B') | \in | ss \rangle$   
**shows**  $\langle \ddagger P ts [\in] A \implies \ddagger P ts [\in] A' \rangle, \langle \ddagger P ts [\in] B \implies \ddagger P ts [\in] B' \rangle$   
 $\langle proof \rangle$

**lemma** *epath-eff*:

**assumes**  $\langle epath steps \rangle \langle effStep (shd steps) ss \rangle$   
**shows**  $\langle fst (shd (stl steps)) | \in | ss \rangle$   
 $\langle proof \rangle$

**abbreviation**  $\langle lhs s \equiv fst (fst s) \rangle$

**abbreviation**  $\langle rhs s \equiv snd (fst s) \rangle$

**abbreviation**  $\langle lhsd s \equiv lhs (shd s) \rangle$

**abbreviation**  $\langle rhsd s \equiv rhs (shd s) \rangle$

**lemma** *epath-Pre-sdrop*:

**assumes**  $\langle epath steps \rangle$  **shows**  
 $\langle \ddagger P ts [\in] lhs (shd steps) \implies \ddagger P ts [\in] lhsd (sdrop m steps) \rangle$   
 $\langle \ddagger P ts [\in] rhs (shd steps) \implies \ddagger P ts [\in] rhsd (sdrop m steps) \rangle$   
 $\langle proof \rangle$

**lemma** *Saturated-sdrop*:

**assumes**  $\langle Saturated steps \rangle$   
**shows**  $\langle Saturated (sdrop n steps) \rangle$   
 $\langle proof \rangle$

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

**definition** *treeB* ::  $\langle (\text{sequent} \times \text{rule}) \text{ stream} \Rightarrow \text{fm set} \rangle$  **where**

$\langle \text{treeB steps} \equiv \bigcup s \in \text{sset steps. set (rhs s)} \rangle$

**lemma** *treeA-snth*:  $\langle p \in \text{treeA steps} \implies \exists n. p [\in] \text{lhsd (sdrop n steps)} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *treeB-snth*:  $\langle p \in \text{treeB steps} \implies \exists n. p [\in] \text{rhsd (sdrop n steps)} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *treeA-sdrop*:  $\langle \text{treeA (sdrop n steps)} \subseteq \text{treeA steps} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *treeB-sdrop*:  $\langle \text{treeB (sdrop n steps)} \subseteq \text{treeB steps} \rangle$   
 $\langle \text{proof} \rangle$

**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**  $\langle \exists k. \text{takenAtStep } r \text{ (shd (sdrop k steps))} \rangle$

$\langle \text{proof} \rangle$

**lemma** *Hintikka-epath*:

**assumes**  $\langle \text{epath steps} \rangle \langle \text{Saturated steps} \rangle$

**shows**  $\langle \text{Hintikka (treeA steps) (treeB steps)} \rangle$

$\langle \text{proof} \rangle$

### 9.3 Completeness

**lemma** *fair-stream-rules*:  $\langle \text{Fair-Stream.fair rules} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *fair-rules*:  $\langle \text{fair rules} \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *epath-prover*:

**fixes**  $A B :: \langle \text{fm list} \rangle$

**defines**  $\langle t \equiv \text{prover (A, B)} \rangle$

**shows**  $\langle (\text{fst (root } t) = (A, B) \wedge \text{wf } t \wedge \text{tfinite } t) \vee$

$(\exists \text{steps. fst (shd steps) = (A, B) \wedge \text{epath steps} \wedge \text{Saturated steps}) \rangle$  (**is**  $\langle ?A \vee ?B \rangle$ )

$\langle \text{proof} \rangle$

**lemma** *epath-countermodel*:

**assumes**  $\langle \text{fst (shd steps) = (A, B)} \rangle \langle \text{epath steps} \rangle \langle \text{Saturated steps} \rangle$

**shows**  $\langle \exists (E :: - \Rightarrow \text{tm}) F G. \neg \text{sc (E, F, G) (A, B)} \rangle$

$\langle \text{proof} \rangle$

**theorem** *prover-completeness*:

**assumes**  $\langle \forall (E :: - \Rightarrow \text{tm}) F G. \text{sc (E, F, G) (A, B)} \rangle$

```

defines < $t \equiv prover (A, B)$ >
shows < $fst (root t) = (A, B) \wedge wf t \wedge tfinite t$ >
<proof>

```

**corollary**

```

assumes < $\forall (E :: - \Rightarrow tm) F G. \llbracket E, F, G \rrbracket p$ >
defines < $t \equiv prover ([], [p])$ >
shows < $fst (root t) = ([], [p]) \wedge wf t \wedge tfinite t$ >
<proof>

```

**end**

## 10 Result

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

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

```

fixes  $A B :: \langle fm list \rangle$ 
defines < $t \equiv prover (A, B)$ >
shows < $tfinite t \wedge wf t \longleftrightarrow (\forall (E :: - \Rightarrow tm) F G. sc (E, F, G) (A, B))$ >
<proof>

```

**corollary**

```

fixes  $p :: fm$ 
defines < $t \equiv prover ([], [p])$ >
shows < $tfinite t \wedge wf t \longleftrightarrow (\forall (E :: - \Rightarrow tm) F G. \llbracket E, F, G \rrbracket p)$ >
<proof>

```

**end**

## References

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