A Naive Prover for First-Order Logic Formalized in Isabelle/HOL

Asta Halkjær From
PhD student at the Technical University of Denmark

LSD Seminar talk, 2022-05-06

Motivation

First-Order Logic is ubiquitous

Its semi-decidability is a classic result

The naive prover may be *naive* but we:

- Formalize a classic result
- Illustrate one way to formalize a logic + prover in Isabelle/HOL
- Use techniques applicable to less naive provers
- Have fun

Overview

- First-Order Logic in Isabelle/HOL
- Sequent Calculus
- Prover Idea
  - Fair Streams
  - Natural Number Encoding
- Performance on Example Proofs
- Abstract Completeness Framework
  - Formalizing the Prover
- Soundness
- Completeness
- A Less Naive Prover?

Isabelle/HOL Proof Assistant

Work in *higher-order logic* rather than English

think *functional programming + logic*

We can *give precise definitions*

We can *verify results*

We can *get help*

We can *spit out programs*

FOL Syntax

Two tiers:

- Terms s, t:
  - variables: **x**, **y**, **z**
  - functions applied to terms: **a**, **f(x)**, **g(a, h(y))**
- Formulas p, q:
  - falsity: $\bot$
  - predicates on lists of terms: **P(t)**, **Q**, **R(u, v)**
  - implication: $\mathbf{p \longrightarrow q}$
  - universal quantification: $\mathbf{\forall x.\ p(x)}$

De Bruijn: write $\mathbf{\forall x.\ \forall y.\ p(x, y)}$ as $\mathbf{\forall \forall p(1, 0)}$

FOL Syntax *in Isabelle/HOL*

We *deeply embed* the FOL syntax as objects in HOL

Terms:

```
datatype tm
  = Var nat (‹#›)
  | Fun nat ‹tm list› (‹†›)
```

Formulas:

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

FOL Semantics *in Isabelle/HOL*

Write functional program (+ logic) that *interprets* model + syntax in HOL

```
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. ⟦E⟨0:x⟩, F, G⟧ p)›
```

Sequent Calculus I

Proof system for first-order logic

Based on sequents **A** ⊢ **B:**

```
type_synonym sequent = ‹fm list × fm list›
```

Think of **A** as assumptions and **B** as possible conclusions:

```
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))›
```

Benefit: *subformula property*

Sequent Calculus II (obtusely)

From System LK on Wikipedia:

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta} \quad (\rightarrow R)$$

If we see **p ⟶ q** on the right, we just continue as above the line (with **p** and **q**).

But we only look at the first formula? What if it’s a predicate?
Then we need to move things around? Could we forget about a formula?

And how do we pick **t** here? What if we get it wrong? Do we need to copy first?

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta} \quad (\forall L)$$

Sequent Calculus III

An implication is useful once (there are only two subformulas).
A universal quantification may be useful twice, thrice, *who knows how many times*.

Should we keep applying the $\forall L$ rule?
We might *ignore* all the other formulas!

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \ (\to R)$$

Maybe we can be *smart*? Devise a *fair* strategy?
Find out exactly which **t**'s we need to instantiate with?

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta} \ (\forall L)$$

Or maybe we can be really *naive*! Wait for *divine inspiration! Exact instructions!*

Sequent Calculus With Very Specific Rules

$$\textsc{Idle}\ \frac{A \vdash B}{A \vdash B} \qquad \textsc{Axiom}\ n\ ts\ \frac{}{A \vdash B}\ \textsc{if}\ \ddagger n\ ts\ [\in]\ A\ \textsc{and}\ \ddagger n\ ts\ [\in]\ B$$

$$\textsc{FlsL}\ \frac{}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ A \qquad \textsc{FlsR}\ \frac{A \vdash B\ [\div]\ \bot}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ B$$

$$\textsc{ImpL}\ p\ q\ \frac{A\ [\div]\ (p \longrightarrow q) \vdash p\ \#\ B \qquad q\ \#\ A\ [\div]\ (p \longrightarrow q) \vdash B}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ A$$

$$\textsc{ImpR}\ p\ q\ \frac{p\ \#\ A \vdash q\ \#\ B\ [\div]\ (p \longrightarrow q)}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ B$$

$$\textsc{UniL}\ t\ p\ \frac{p\langle t/0\rangle\ \#\ A \vdash B}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ A \qquad \textsc{UniR}\ p\ \frac{A \vdash p\langle \#\mathit{fresh}(A@B)/0\rangle\ \#\ B\ [\div]\ \forall p}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ B$$

Example Prover Output

```
|- (P --> Falsity) --> (P --> Falsity)
 + ImpR on P --> Falsity and P --> Falsity
P --> Falsity |- P --> Falsity
 + ImpR on P and Falsity
P, P --> Falsity |- Falsity
 + ImpL on P and Falsity
  P |- P, Falsity
   + FlsR
  P |- P
   + Axiom on P
  Falsity, P |- Falsity
   + FlsL
```

Prover Idea I

How do we get this *divine inspiration?* These *exact instructions?*

A *stream of rules* could tell us what to do

- Say we have the sequent ⊢ **†0 [] ⟶ †0 []**
- The rule **ImpR (†0 []) (†0 [])** says we can prove it *if*
- we can prove the sequent **†0 []** ⊢ **†0 []**

We must *always eventually* reach the rule we need

- We need to reach **Axiom 0 []** for the sequent **†0 []** ⊢ **†0 []**
- But **Axiom 1 []** doesn't harm us

Prover Idea II

Pretend numbers are rules. Consider the stream:

0 1 2 3 4 5 6 7 8 9 10 11 12 ...

Every number appears somewhere in the sequence

So we will reach the number we need at some point!

But what if we need it twice? Or we need 12 before we need 5?

Prover Idea III

Consider instead the stream of numbers

$$0\ 0\ 1\ 0\ 1\ 2\ 0\ 1\ 2\ 3\ 0\ 1\ 2\ 3\ 4\ \ldots$$

Every number *keeps appearing*

The stream is *fair* (but larger numbers are further away than before)

How to get a fair stream of *rules*?

My Theory Fair-Stream

```
definition upt_lists :: ‹nat list stream› where
  ‹upt_lists ≡ smap (upt 0) (stl nats)›
```

[0] [0, 1] [0, 1, 2] [0, 1, 2, 3] …

```
definition fair_nats :: ‹nat stream› where
  ‹fair_nats ≡ flat upt_lists›
```

0, 0, 1, 0, 1, 2, 0, 1, 2, 3, …

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

Pick any m. Any x appears after.

A handful of lemmas later…

```
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)
```

Encoding To and From the Natural Numbers

The Isabelle theory Nat-Bijection provides the following operations:

- prod_encode :: "nat × nat ⇒ nat" for **p ⟶ q**
- prod_decode :: "nat ⇒ nat × nat"
- sum_encode :: "nat + nat ⇒ nat" for **⊥** | **P(t)** | …
- sum_decode :: "nat ⇒ nat + nat"
- list_encode :: "nat list ⇒ nat" for **f(...)**
- list_decode :: "nat ⇒ nat list"

I write ‹c $ x ≡ sum_encode (c x)›

Encoding Terms as Natural Numbers

```
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
```

Encoding Formulas as Natural Numbers

```
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
```

Encoding Rules as Natural Numbers

```
text ‹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)›
```

```
lemma ‹map rule_of_nat [0..<100] =
 [FlsL, ImpL ⊥ ⊥, FlsR, UniL (# 0) ⊥, UniR ⊥, ImpR ⊥ ⊥, ImpR (‡ 0 []) ⊥,
  ImpL ⊥ (‡ 0 []), Axiom 0 [], ImpR ⊥ (‡ 0 []), UniR (‡ 0 []),
  ImpL (‡ 0 []) ⊥, ImpR ⊥ (∀ ⊥), UniL (# 0) (‡ 0 []), Axiom 0 [# 0],
  ImpR ⊥ (∀ ⊥), Axiom 1 [], UniL († 0 []) ⊥, UniR (∀ ⊥), ImpR (‡ 0 []) ⊥,
  ImpR (‡ 0 []) (‡ 0 []), ImpL ⊥ (∀ ⊥), Axiom 0 [# 0, # 0],
  ImpR ⊥ (‡ 0 [# 0]), Axiom 1 [# 0], ImpL (‡ 0 []) (‡ 0 []), Axiom 2 [],
  ImpR (‡ 0 []) (‡ 0 []), UniR (‡ 0 [# 0]), ImpL (∀ ⊥) ⊥, ImpR (∀ ⊥) ⊥,
  UniL (# 0) (∀ ⊥), Axiom 0 [† 0 []], ImpR ⊥ (∀ (‡ 0 [])),
  Axiom 1 [# 0, # 0], UniL († 0 []) (‡ 0 []), Axiom 2 [# 0],
  ImpR (‡ 0 []) (∀ ⊥), Axiom 3 [], UniL (# 1) ⊥, UniR (∀ (‡ 0 [])),
  ImpR (∀ ⊥) ⊥, ImpR ⊥ (‡ 0 [# 0]), ImpL ⊥ (‡ 0 [# 0]),
  Axiom 0 [# 0, # 0, # 0], ImpR ⊥ (‡ 1 []), Axiom 1 [† 0 []],
  ImpL (‡ 0 []) (∀ ⊥), Axiom 2 [# 0, # 0], ImpR (‡ 0 []) (‡ 0 [# 0]),
  Axiom 3 [# 0], ImpL (∀ ⊥) (‡ 0 []), Axiom 4 [], ImpR (∀ ⊥) (‡ 0 []),
  UniR (‡ 1 []), ImpL (‡ 0 [# 0]) ⊥, ImpR (‡ 0 []) (∀ ⊥),
  UniL (# 0) (‡ 0 [# 0]), Axiom 0 [† 0 [], # 0], ImpR ⊥ (⊥ ⟶ ⊥),
  Axiom 1 [# 0, # 0, # 0], UniL († 0 []) (∀ ⊥), Axiom 2 [† 0 []],
  ImpR (‡ 0 []) (∀ (‡ 0 [])), Axiom 3 [# 0, # 0], UniL (# 1) (‡ 0 []),
  Axiom 4 [# 0], ImpR (∀ ⊥) (∀ ⊥), Axiom 5 [], UniL († 0 [# 0]) ⊥,
  UniR (⊥ ⟶ ⊥), ImpR (‡ 0 [# 0]) ⊥, ImpR (∀ ⊥) (‡ 0 []),
  ImpL ⊥ (∀ (‡ 0 [])), Axiom 0 [# 1], ImpR ⊥ (‡ 0 [# 0, # 0]),
  Axiom 1 [† 0 [], # 0], ImpL (‡ 0 []) (‡ 0 [# 0]), Axiom 2 [# 0, # 0, # 0],
  ImpR (‡ 0 []) (‡ 1 []), Axiom 3 [† 0 []], ImpL (∀ ⊥) (∀ ⊥),
```

What Does It Matter? I

```
term ‹P ⟶ P›
term ‹‡0 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡0 []) = 1› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡0 [])) = 27› by eval
lemma ‹nat_of_rule (Axiom 0 []) = 8› by eval

term ‹(∀x. P x) ⟶ P a›
term ‹∀(‡0 [#0]) ⟶ ‡0 [†0 []]›
lemma ‹nat_of_fm (‡0 [†0 []]) = 13› by eval
lemma ‹nat_of_rule (ImpR (∀(‡0 [#0])) (‡0 [†0 []])) = 1865› by eval
lemma ‹nat_of_rule (UniL (†0 []) (∀(‡0 [#0]))) = 997› by eval
lemma ‹nat_of_rule (Axiom 0 [†0 []]) = 32› by eval
```

Recall that the sequence looks like: 0 0 1 0 1 2 0 1 2 3 …

We reach 1865 only at position 1865*(1+1865)/2 = **1740045**.

What Does It Matter? II

The numbers in the formulas matter:

```
term ‹P ⟶ P ⟶ P›
term ‹‡0 [] ⟶ ‡0 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡0 [] ⟶ ‡0 []) = 18› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡0 [] ⟶ ‡0 [])) = 469› by eval

term ‹P ⟶ Q ⟶ P›
term ‹‡0 [] ⟶ ‡1 [] ⟶ ‡0 []›
lemma ‹nat_of_fm (‡1 [] ⟶ ‡0 []) = 70› by eval
lemma ‹nat_of_rule (ImpR (‡0 []) (‡1 [] ⟶ ‡0 [])) = 5409› by eval
```

We reach 469 at position **110215**

We reach 5409 at position **14631345**

Example Proofs I

```
time ./Main "Imp (Pre 0 []) (Pre 0 [])"
|- (P) --> (P)
 + ImpR on P and P
P |- P
 + Axiom on P
```

```
Executed in    9.80 millis
```

Example Proofs II

```
time ./Main "Imp (Uni (Pre 0 [Var 0])) (Pre 0 [Fun 0 []])"
|- (forall P(0)) --> (P(a))
 + ImpR on forall P(0) and P(a)
forall P(0) |- P(a)
 + UniL on 0 and P(0)
P(0), forall P(0) |- P(a)
 + UniL on a and P(0)
P(a), P(0), forall P(0) |- P(a)
 + UniL on 1 and P(0)
P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0) and P(0)
P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on b and P(0)
P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 2 and P(0)
P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0, 0) and P(0)
P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(0) and P(0)
P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on c and P(0)
P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 3 and P(0)
P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(a) and P(0)
P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(0, 0) and P(0)
P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on h(0) and P(0)
P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on d and P(0)
P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on 4 and P(0)
P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on f(0, 0, 0) and P(0)
P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on g(a) and P(0)
P(g(a)), P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1), P(a), P(0), forall P(0) |- P(a)
 + UniL on h(0, 0) and P(0)
P(h(0, 0)), P(g(a)), P(f(0, 0, 0)), P(4), P(d), P(h(0)), P(g(0, 0)), P(f(a)), P(3), P(c), P(g(0)), P(f(0, 0)), P(2), P(b), P(f(0)), P(1),
P(a), P(0), forall P(0) |- P(a)
 + Axiom on P(a)
________________________________________

Executed in   3.51 secs
```

We need to get to **1865** to hit the ImpR rule.

Then we start back at 0.

The UniL rule we need is at **997**.

But then we keep running from **997** to **1866**.

And hit lots of **UniL** rules in between…

In the end: *a very silly derivation*.

Example Proofs III

```
time ./Main "Imp (Pre 0 []) (Imp (Pre 0 []) (Pre 0 []))"
|- (P) --> ((P) --> (P))
 + ImpR on P and (P) --> (P)
P |- (P) --> (P)
 + ImpR on P and P
P, P |- P
 + Axiom on P
```

(position **110215**)

```
Executed in  192.72 millis
```

Example Proofs IV

```
time ./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
```

(position **14631345**)

Executed in **43.01 secs**

The Abstract Completeness Framework

We base the prover on a framework by Blanchette, Popescu and Traytel

Their code includes a *naive prover* for propositional logic (no proofs)
Their paper includes ideas for a *naive prover* for first-order logic

My entry realizes those ideas

https://www.isa-afp.org/entries/Abstract_Completeness.html

Blanchette, J.C., Popescu, A. & Traytel, D. Soundness and Completeness Proofs by Coinductive Methods. Journal of Automated Reasoning 58, 149–179 (2017). https://doi.org/10.1007/s10817-016-9391-3

What’s In A Prover?

Our sequent calculus prover attempts to build a proof tree using a stream of rules:

```
definition ‹prover ≡ mkTree rules›
```

Such an attempt can be infinite so we need codatatypes:

```
codatatype 'a tree = Node (root: 'a) (cont: "'a tree fset")
```

The root of the proof tree is our sequent + the first applicable rule

The children are the proof trees for the sequents obtained by *applying* this rule

```
primcorec mkTree where
  "root (mkTree rs s) = (s, (shd (trim rs s)))"
| "cont (mkTree rs s) = fimage (mkTree (stl (trim rs s))) (pickEff (shd (trim rs s)) s)"
```

What’s Required of a Prover?

The framework requires that:

- We explain what rules do and give a stream of rules
- We pick a set S of proof states that our prover stays within
- Some rule always applies (hint: **Idle**!)
- Our rules are persistent: they do not step on each other’s toes

```
locale RuleSystem = RuleSystem_Defs eff rules
for eff :: "'rule ⇒ 'state ⇒ 'state fset ⇒ bool" and rules :: "'rule stream" +
fixes S :: "'state set"
assumes eff_S: "⋀ s r sl s'. ⟦s ∈ S; r ∈ R; eff r s sl; s' |∈| sl⟧ ⟹ s' ∈ S"
and enabled_R: "⋀ s. s ∈ S ⟹ ∃ r ∈ R. ∃ sl. eff r s sl"
assumes per: "⋀ r. r ∈ R ⟹ per r"
```

What's Delivered by a Prover?

The framework tells us the prover produces one of two things:

```
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)›
```

- A finite, well formed proof tree
  - Soundness: show that this guarantees *validity* of the formula
- a saturated escape path (epath)
  - Completeness: show that this induces a *counter model* for the formula

Sequent Calculus With Very Specific Rules *Reprise*

$$\textsc{Idle}\ \frac{A \vdash B}{A \vdash B} \qquad \textsc{Axiom}\ n\ ts\ \frac{}{A \vdash B}\ \textsc{if}\ \ddagger n\ ts\ [\in]\ A\ \textsc{and}\ \ddagger n\ ts\ [\in]\ B$$

$$\textsc{FlsL}\ \frac{}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ A \qquad \textsc{FlsR}\ \frac{A \vdash B\ [\div]\ \bot}{A \vdash B}\ \textsc{if}\ \bot\ [\in]\ B$$

$$\textsc{ImpL}\ p\ q\ \frac{A\ [\div]\ (p \longrightarrow q) \vdash p \# B \qquad q \# A\ [\div]\ (p \longrightarrow q) \vdash B}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ A$$

$$\textsc{ImpR}\ p\ q\ \frac{p \# A \vdash q \# B\ [\div]\ (p \longrightarrow q)}{A \vdash B}\ \textsc{if}\ (p \longrightarrow q)\ [\in]\ B$$

$$\textsc{UniL}\ t\ p\ \frac{p\langle t/0\rangle \# A \vdash B}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ A \qquad \textsc{UniR}\ p\ \frac{A \vdash p\langle \# \mathit{fresh}(A @ B)/0\rangle \# B\ [\div]\ \forall p}{A \vdash B}\ \textsc{if}\ \forall p\ [\in]\ B$$

What Our Rules Do

```
function eff :: ‹rule ⇒ sequent ⇒ (sequent fset) option› where
  ‹eff Idle (A, B) =
    Some {| (A, B) |}›
| ‹eff (Axiom n ts) (A, B) = (if ‡n ts [∈] A ∧ ‡n 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 {| (p⟨t/0⟩ # A, B) |} else None)›
| ‹eff (UniR p) (A, B) = (if ∀p [∈] B then
    Some {| (A, p⟨#(fresh (A @ B))/0⟩ # B [÷] ∀p) |} else None)›
```

Our Stream of Rules

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

A datatype for our rules

```
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 .
```

Our fair stream of rules

which includes every rule (so also **Idle**)

Instantiating the Framework

We can easily instantiate the framework:

```
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 ‹λr s ss. eff r s = Some ss› rules UNIV
  using per_rules by unfold_locales
```

Soundness

Prove that valid premises ensure valid conclusions.

The framework lifts local soundness:

```
lemma eff_sound:
  fixes E :: ‹_ ⇒ 'a›
  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)›
```

To global soundness:

```
theorem prover_soundness:
  assumes ‹tfinite t› and ‹wf t›
  shows ‹sc (E, F, G) (fst (root t))›
```

Completeness

Way more fun!

We need to transform a *failed proof attempt* (epath) into a *counter model*

- Look at every sequent on the infinite branch
- Satisfy every predicate in assumptions (no predicate in conclusions)
  - No overlap or an **Axiom** rule would have terminated the branch
- Satisfiability lifts to each connective
  - Because our rules are sensible

We can characterize the properties of an epath *syntactically*

Hintikka Sets

A: set of assumption formulas on epath, **B**: set of conclusion formulas on epath

```
locale Hintikka =
  fixes A B :: ‹fm set›
  assumes
    Basic: ‹‡n ts ∈ A ⟹ ‡n 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. p⟨t/0⟩ ∈ A› and
    UniB: ‹∀p ∈ B ⟹ ∃t. p⟨t/0⟩ ∈ B›
```

Axiom would have kicked in
FlsL would have kicked
ImpL has kicked in
ImpR has kicked in
UniL has kicked in
UniR has kicked in

Now think of **A** as formulas to satisfy and **B** as formulas to falsify

Occurrence of a formula demands presence of *corresponding evidence*

Counter Model

We will use the *term universe*: terms are interpreted as themselves

```
lemma id_tm [simp]: ‹⦇#, †⦈ t = t›
  by (induct t) (auto cong: map_cong)
```

The counter model for epath sets **A**, **B** is given by:

```
abbreviation ‹M A ≡ ⟦#, †, λn ts. ‡n ts ∈ A⟧›
```

A predicate is true when it occurs in **A**.

This gives a counter model:

```
theorem Hintikka_counter_model:
  assumes ‹Hintikka A B›
  shows ‹(p ∈ A ⟶ M A p) ∧ (p ∈ B ⟶ ¬ M A p)›
```

Saturated Escape Paths Form Hintikka Sets

Most gnarly part of the formalization due to the use of codatatypes

```
lemma Hintikka_epath:
  assumes ‹epath steps› ‹Saturated steps›
  shows ‹Hintikka (treeA steps) (treeB steps)›
```

But! We are helped by having very specific rules.

Say we need to show **ImpB**: if some **p ⟶ q** is in **B** then **p** is in **A** and **q** is in **B**

- If **p ⟶ q** is in the proof tree then **ImpR p q** is enabled at some point
- So at some later point it will be *applied* and have the desired effect
- Profit

Result

We have a sound and complete prover:

```
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
```

We can export it to Haskell and run on it real examples

A Less Naive Prover?

Frederik Krogsdal Jacobsen and I have used similar techniques for another prover

The rules there are based on SeCaV: https://secav.compute.dtu.dk/

There we are *smart*. 3000 lines of formalization instead of 900.
We do not instantiate with every term, so we need a custom *bounded* semantics.
We apply rules to *every* matching formula to ensure fairness. Many concerns!

Accepted at ITP 2022 (Interactive Theorem Proving) and to the AFP:
https://www.isa-afp.org/entries/FOL_Seq_Calc2.html

References

My prover + formalization:
https://www.isa-afp.org/entries/FOL_Seq_Calc3.html

The abstract completeness framework by Blanchette, Popescu and Traytel:
https://www.isa-afp.org/entries/Abstract_Completeness.html

Blanchette, J.C., Popescu, A. & Traytel, D. Soundness and Completeness Proofs by Coinductive Methods. Journal of Automated Reasoning 58, 149–179 (2017). https://doi.org/10.1007/s10817-016-9391-3