
DTU

DTU

Verifying a Sequent Calculus Prover

Asta Halkjær From Frederik Krogsdal Jacobsen

Technical University of Denmark

Introduction

A sound and complete prover for first-order logic with functions

Based on a sequent calculus

All proofs are formally verified in Isabelle/HOL

Human-readable proof certificates

Background

Formalized metatheory for non-trivial sequent calculus provers

Formal verification of an executable prover

Novel analytic proof technique for completeness

Verifiable and human-readable proof certificates

A prover for the SeCaV system

Sample SeCaV Proof Rules

$$\frac{\text{Neg } p\ 2\ z}{p; z}\ \textsc{Basic} \qquad \frac{z \qquad z \qquad y}{y}\ \textsc{Ext} \qquad \frac{p; z}{\text{Neg (Neg } p); z}\ \textsc{NegNeg}$$

$$\frac{p; q; z}{\text{Dis } p\ q; z}\ \textsc{AlphaDis} \qquad \frac{\text{Neg } p; z \qquad \text{Neg } q; z}{\text{Neg (Dis } p\ q); z}\ \textsc{BetaDis}$$

$$\frac{p\,[\text{Var } 0{=}t]; z}{\text{Exi } p; z}\ \textsc{GammaExi}$$

$$\frac{\text{Neg } (p\,[\text{Var } 0{=}\text{Fun } i\ []]); z \qquad i \text{ fresh}}{\text{Neg (Exi } p); z}\ \textsc{DeltaExi}$$

Prover I

SeCaV rules affect *one formula* at a time

Our prover rules affect *every applicable formula* at once

We copy Gamma formulas and remember all terms on the branch

So no formula or instantiation is forgotten

Prover I

SeCaV rules affect *one formula* at a time

Our prover rules affect *every applicable formula* at once

We copy Gamma formulas and remember all terms on the branch

So no formula or instantiation is forgotten

Rules affect disjoint formulas

So we can apply them in any order

Prover I

SeCaV rules affect *one formula* at a time

Our prover rules affect *every applicable formula* at once

We copy Gamma formulas and remember all terms on the branch

So no formula or instantiation is forgotten

Rules affect disjoint formulas

So we can apply them in any order

We apply rules *fairly* and repeatedly

So we never miss out on a proof

Prover II

We rely on the abstract completeness framework by Blanchette, Popescu and Traytel

We need to fix a *stream* of rules from the beginning

Proof attempts are *coinductive trees* grown by applying these rules

If a tree cannot be grown further, we found a proof

A function gives the *child sequents* representing the subgoals left after applying a rule

We export code to Haskell to obtain an executable prover

Prover — proof example

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{Neg (Uni (Con } P(0)\ Q(0)))\text{; Neg } P(0)\text{; Neg } Q(0)\text{;}\ \text{Neg } P(a)\text{; Neg } Q(a)\text{; } P(a)}\ \textsc{Basic}}{\text{Neg (Uni (Con } P(0)\ Q(0)))\text{; Neg (Con } P(0)\ Q(0))\text{;}\ \text{Neg (Con } P(a)\ Q(a))\text{; } P(a)}\ \textsc{AlphaCon}}{\text{Neg (Uni (Con } P(0)\ Q(0)))\text{; Neg (Con } P(0)\ Q(0))\text{;}\ \text{Neg (Con } P(a)\ Q(a))\text{; } P(a)}\ (\ )}{\text{Neg (Uni (Con } P(0)\ Q(0)))\text{; } P(a)}\ \textsc{GammaUni}}{\text{Neg (Uni (Con } P(0)\ Q(0)))\text{; } P(a)}\ (\ ;\ ;\ )}{\text{Imp (Uni (Con } P(0)\ Q(0)))\ P(a)}\ \textsc{AlphaImp}}{\text{Imp (Uni (Con } P(0)\ Q(0)))\ P(a)}\ (\textsc{NegNeg})$$

Prover — certificate example

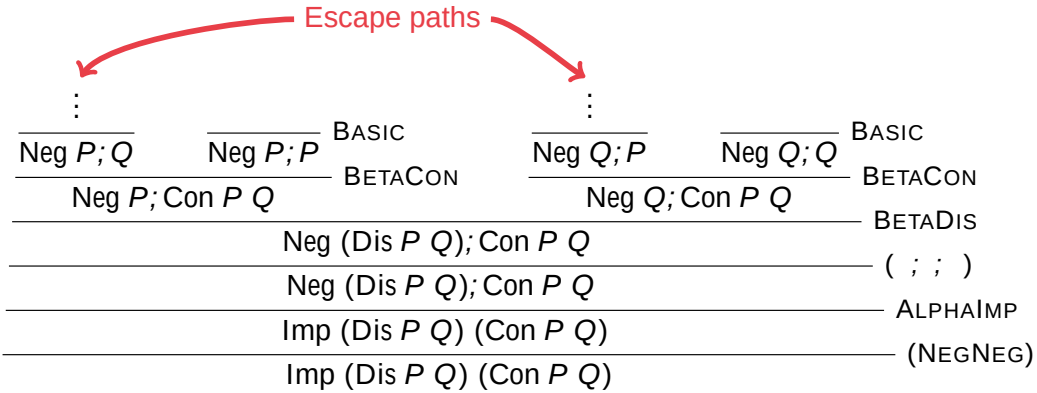
```
Imp (Uni (Con (P [0]) (Q [0]))) (P [a])

AlphaImp
  Neg (Uni (Con (P [0]) (Q [0])))
  P [a]
Ext
  Neg (Uni (Con (P [0]) (Q [0])))
  Neg (Uni (Con (P [0]) (Q [0])))
  P [a]
GammaUni[0]
  Neg (Con (P [0]) (Q [0]))
  Neg (Uni (Con (P [0]) (Q [0])))
  P [a]
```

```
Ext
  Neg (Uni (Con (P [0]) (Q [0])))
  Neg (Uni (Con (P [0]) (Q [0])))
  P [a]
  Neg (Con (P [0]) (Q [0]))
GammaUni[a]
  Neg (Con (P [a]) (Q [a]))
  Neg (Uni (Con (P [0]) (Q [0])))
  P [a]
  Neg (Con (P [0]) (Q [0]))
Ext
  Neg (Con (P [0]) (Q [0]))
  Neg (Con (P [a]) (Q [a]))
  P [a]
  Neg (Uni (Con (P [0]) (Q [0])))
AlphaCon
  Neg (P [0])
  Neg (Q [0])
  Neg (Con (P [a]) (Q [a]))
  P [a]
  Neg (Uni (Con (P [0]) (Q [0])))
```

```
Ext
  Neg (Con (P [a]) (Q [a]))
  P [a]
  Neg (Uni (Con (P [0]) (Q [0])))
  Neg (P [0])
  Neg (Q [0])
AlphaCon
  Neg (P [a])
  Neg (Q [a])
  P [a]
  Neg (Uni (Con (P [0]) (Q [0])))
  Neg (P [0])
  Neg (Q [0])
Ext
  P [a]
  Neg (Uni (Con (P [0]) (Q [0])))
  Neg (P [0])
  Neg (Q [0])
  Neg (P [a])
  Neg (Q [a])
Basic
```

Prover — escape path example



Soundness I

If our prover returns a proof, we can build a SeCaV proof

The SeCaV proof system is sound, so the prover is sound

We use the abstract soundness framework by Blanchette et al.

If the *children* of a sequent all have SeCaV proofs, so does the sequent

Soundness II

If the *children* of a sequent all have SeCaV proofs, so does the sequent:

1. Assume all child sequents have a proof
2. Induction on sequent: use appropriate SeCaV rule for each formula

Example: Our sequent looks like Dis $P\ Q;:::$, so $P;Q;:::$ is a child sequent with a SeCaV proof. We apply the AlphaDis rule to prove the sequent using the proof of $P;Q;:::$ (and possibly some reordering).

$$\dfrac{\dfrac{\vdots}{P;Q;:::}\ \text{ASSUMPTION} \qquad :::}{\text{Dis}\ P\ Q;:::}\ \text{AlphaDis}$$

$$\vdots$$

Completeness

Framework: prover either produces a finite, well formed proof tree or an infinite tree with a saturated escape path

Need to show that root sequent of a saturated escape path is not valid:

- Formulas on saturated escape paths form Hintikka sets
- Hintikka sets induce a well formed countermodel

. . . so valid sequents result in finite, well formed proof trees

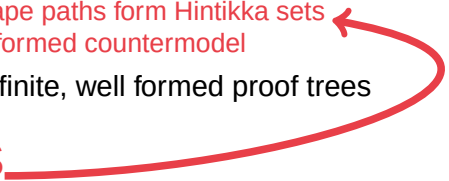
Completeness

Framework: prover either produces a finite, well formed proof tree or an infinite tree with a saturated escape path

Need to show that root sequent of a saturated escape path is not valid:

Formulas on saturated escape paths form Hintikka sets
Hintikka sets induce a well formed countermodel

. . . so valid sequents result in finite, well formed proof trees



HERE BE DRAGONS

(need to build a bounded countermodel over only the terms in the sequent and ensure functions stay inside its domain)

Bounded semantics

In a completeness proof for a *calculus* we can assume that Gamma formulas are instantiated with all possible terms

Thus, we can build a countermodel in the full Herbrand domain

Our prover only uses terms from the given sequent (and fresh ones)

So we must build a *bounded* countermodel over this restricted domain

We must ensure that our function denotation stays inside this domain

Subtypes fail us

The SeCaV semantics represents the domain as a type variable.

We cannot build the subtype of terms from a *local* sequent (yet?[1])

So we represent the domain as an explicit parameter to the semantics

We have $u; E; F; G \models \text{Uni } P$ iff $u; E; F; G \models P(x)$ for all $x \ 2 \ u$

We reprove soundness of SeCaV under this (u)semantics

[1] Kunčar and Popescu ITP 2014

Hintikka sets

We always need at least one term

terms H = *if* (⋃ *p 2 H: set* (*subtermFm p*)) = *f g then f Fun 0* [] *g*
else (⋃ *p 2 H: set* (*subtermFm p*))

To quantify over in our Hintikka sets

locale *Hintikka* =
fixes *H* :: *fm set*
assumes
Basic: *Pre n ts 2 H* =) *Neg* (*Pre n ts*) ~2 *H* **and**
AlphaDis: *Dis p q 2 H* =) *p 2 H* ^ *q 2 H* **and**
BetaDis: *Neg* (*Dis p q*) *2 H* =) *Neg p 2 H* _ *Neg q 2 H* **and**
GammaExi: *Exi p 2 H* =) 8 *t 2 terms H: sub 0 t p 2 H* **and**
DeltaExi: *Neg* (*Exi p*) *2 H* =) 9 *t 2 terms H: Neg* (*sub 0 t p*) *2 H* **and**
⋮

Bounded countermodel

We carefully build the countermodel

```
E S n     if Var n 2 terms S then Var n else SOME t: t 2 terms S
F S i l     if Fun i l 2 terms S then Fun i l else SOME t: t 2 terms S
G S n ts     Neg (Pre n ts) 2 S
M S     usemantics (terms S) (E S) (F S) (G S)
```

terms is downwards closed, so members evaluate to themselves

```
t 2 terms S = )  semantics-term (E S) (F S) t = t
```

We have a countermodel to any formula in a Hintikka set

```
Hintikka S = )  (p 2 S     !  :  M S p) ^ (Neg p 2 S     !  M S p)
```

Saturated escape paths form Hintikka sets

Final step is to inspect the saturated escape paths

We need to show that the formulas constitute a Hintikka set

On paper, this follows straightforwardly from our rules

In practice, it requires fiddly reasoning about the coinductive paths

In the end: any saturated escape path has a (bounded) countermodel, contradicting the validity of its root sequent

Results and future work

We have verified soundness and completeness in Isabelle/HOL

Verification helped find actual bugs in our implementation

The performance is limited, but optimizations are possible

Generation of proof certificates is not (yet) verified

Potentially consider extensions to the logic such as equality