A Concise Sequent Calculus for Teaching First-Order Logic

Asta Halkjær From and Jørgen Villadsen

Technical University of Denmark

Isabelle Workshop 2020, June 30.

Introduction

We taught a MSc course on **Automated Reasoning** in the Spring 2020 semester.

We used Isabelle formalizations heavily in the course, including the ones presented here.

This talk has three parts:

1. Natural Deduction Assistant (NaDeA).
2. Technique for showing completeness for open formulas.
3. Sequent Calculus Verifier (SeCaV).

NaDeA

Web application for proving formulas in first-order logic using natural deduction.

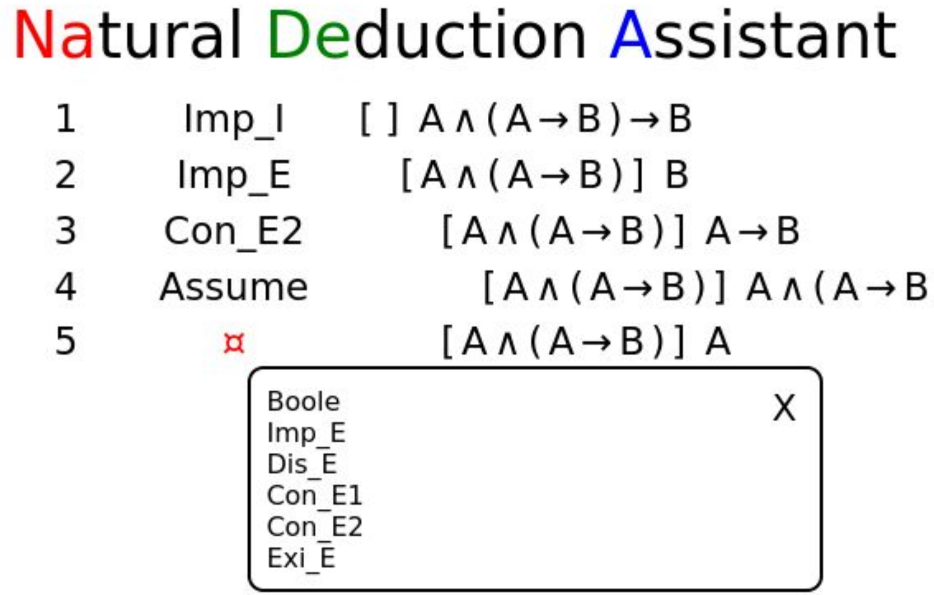
Point and click.

Only presents applicable rules.

Keeps track of assumptions.

Checks side conditions.

Undo and redo to any state.



https://nadea.compute.dtu.dk/

Natural_Deduction_Assistant.thy

NaDeA is backed by a formalization in Isabelle/HOL (6498 lines, 100+ pages): https://github.com/logic-tools/nadea

Deep embedding of the logic: syntax as datatype, semantics as function.

Inductive specification of the proof system.

Online proofs can be exported and checked in Isabelle to guarantee correctness.

1	OK (Imp (Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))) (Pre ''B'' [ ])) [ ]	Imp_I	[ ] A ∧ (A → B) → B
2	OK (Pre ''B'' [ ]) [Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))]	Imp_E	[A ∧ (A → B)] B
3	OK (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ])) [Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))]	Con_E2	[A ∧ (A → B)] A → B
4	OK (Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))) [Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))]	Assume	[A ∧ (A → B)] A ∧ (A → B)
5	OK (Pre ''A'' [ ]) [Con (Pre ''A'' [ ]) (Imp (Pre ''A'' [ ]) (Pre ''B'' [ ]))]	¤	[A ∧ (A → B)] A

Syntax and Semantics

First-order logic without negation and with de Bruijn indices for the variables.

```
type_synonym id = ‹char list›

datatype tm = Var nat | Fun id ‹tm list›

datatype fm = Falsity | Pre id ‹tm list› | Imp fm fm | Dis fm fm | Con fm fm | Exi fm | Uni fm
```

Semantics given environment *e*, function denotation *f* and predicate denotation *g*:

```
primrec
  semantics :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒ fm ⇒ bool› where
  ‹semantics e f g Falsity = False› |
  ‹semantics e f g (Pre i l) = g i (semantics_list e f l)› |
  ‹semantics e f g (Imp p q) = (if semantics e f g p then semantics e f g q else True)› |
  ‹semantics e f g (Dis p q) = (if semantics e f g p then True else semantics e f g q)› |
  ‹semantics e f g (Con p q) = (if semantics e f g p then semantics e f g q else False)› |
  ‹semantics e f g (Exi p) = (∃x. semantics (λn. if n = 0 then x else e (n - 1)) f g p)› |
  ‹semantics e f g (Uni p) = (∀x. semantics (λn. if n = 0 then x else e (n - 1)) f g p)›
```

Completeness for Open Formulas

The completeness of NadeA is based on a formalization by Stefan Berghofer:
https://www.isa-afp.org/entries/FOL-Fitting.html

The needed model existence result applies to *closed* formulas in consistent sets.
(A set is consistent if we cannot derive falsity from any finite subset.)

Open formulas are valid syntactic objects in the formalization and web application.

```
1 OK (Imp (Pre ''A'' [Var 0]) (Pre ''A'' [Var 0])) [ ]      Imp_I     [ ] A(x)→A(x)
2    OK (Pre ''A'' [Var 0]) [Pre ''A'' [Var 0]]             Assume      [A(x)] A(x)
```

We give a technique to also cover them that reuses the completeness result.

Five Steps to Success

```
theorem completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
```

We want to show strong completeness for open formulas using the existing result.

Five Steps to Success – Step 1

```
theorem completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
proof -
  let ?p = ‹put_imps p (rev z)›
  have *: ‹∀(e :: _ ⇒ 'a) f g. semantics e f g ?p›
    using assms(1) semantics_put_imps by fastforce
```

Turn the assumptions into object-level implications which preserves validity (*).

Five Steps to Success – Step 2

```
theorem completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
proof -
  let ?p = ‹put_imps p (rev z)›
  have *: ‹∀(e :: _ ⇒ 'a) f g. semantics e f g ?p›
    using assms(1) semantics_put_imps by fastforce
→ obtain m where **: ‹sentence (put_unis m ?p)›
    using ex_closure by blast
  moreover have ‹∀(e :: _ ⇒ 'a) f g. semantics e f g (put_unis m ?p)›
    using * valid_put_unis by blast
```

Close the formula with universal quantifiers (**) which **moreover** preserves validity.

Five Steps to Success – Step 3

```
theorem completeness':
  assumes ‹∀(e :: _ ⇒ 'a) f g. list_all (semantics e f g) z ⟶ semantics e f g p›
    and ‹denumerable (UNIV :: 'a set)›
  shows ‹OK p z›
proof -
  let ?p = ‹put_imps p (rev z)›
  have *: ‹∀(e :: _ ⇒ 'a) f g. semantics e f g ?p›
    using assms(1) semantics_put_imps by fastforce
  obtain m where **: ‹sentence (put_unis m ?p)›
    using ex_closure by blast
  moreover have ‹∀(e :: _ ⇒ 'a) f g. semantics e f g (put_unis m ?p)›
    using * valid_put_unis by blast
→ ultimately have ‹OK (put_unis m ?p) []›
    using assms(2) sentence_completeness by blast
```

The completeness result for closed formulas now applies.

Five Steps to Success – Step 4

```
ultimately have ‹OK (put_unis m ?p) []›
  using assms(2) sentence_completeness by blast
then have ‹OK ?p []›
  using ** remove_unis_sentence by blast
```

Work within the proof system to eliminate the universal quantifiers:

1. *Specialize* the quantifiers with fresh constants.
2. Substitute the constants with the original variables using an admissible rule.

Specialization shifts de Bruijn indices when substituting under a binder.
The two-step process sidesteps this complication.

Five Steps to Success – Step 4.5

Specialization does two things:

- It increments the inserted variable when going under a binder.
- It decrements existing variables that point beyond the removed binder.

If we try to specialize directly with the original variables, then we need to reason about a tricky sequence of substitutions.

$$(\forall\forall p(0,1,2))[2/0] \rightsquigarrow \forall((\forall p(0,1,2))[3/1]) \rightsquigarrow \forall\forall(p(0,1,2)[4/2]) \rightsquigarrow \forall\forall p(0,1,4)$$

$$(\forall p(0,1,4))[1/0] \rightsquigarrow \forall(p(0,1,4)[2/1]) \rightsquigarrow \forall p(0,2,3)$$

$$p(0,2,3)[0/0] \rightsquigarrow p(0,1,2)$$

When substituting for fresh constants, the closure is already specialized away.

Five Steps to Success – Step 5

```
  ultimately have ‹OK (put_unis m ?p) []›
    using assms(2) sentence_completeness by blast
  then have ‹OK ?p []›
    using ** remove_unis_sentence by blast
→ then show ‹OK p z›
    using remove_imps by fastforce
qed
```

Turn the introduced implications back into assumptions with a deduction theorem.

We do this by (for each implication):

1. Weakening the assumptions with the antecedent.
2. Eliminating the implication with modus ponens.

Five Steps to Success – Step Wait a Second

The universal closure is unnecessary!
(We may want to introduce it for teaching purposes.)

Substitute fresh constants directly for the free variables to close the formula.

It is just as easy to show that this preserves validity.

We save the *specialization* step.

This is how we show completeness for open formulas in SeCaV.

We are curious to hear other solutions to cover open formulas.

SeCaV – Sequent Calculus Verifier

One-sided sequent calculus for first-order logic formalized in Isabelle/HOL.

Uses the same syntax and helper functions as NaDeA.

```
inductive sequent_calculus (‹⊩ _› 0) where
  Basic: ‹⊩ p # z› if ‹member (Neg p) z› |
  AlImp: ‹⊩ Imp p q # z› if ‹⊩ Neg p # q # z› |
  AlDis: ‹⊩ Dis p q # z› if ‹⊩ p # q # z› |
  AlCon: ‹⊩ Neg (Con p q) # z› if ‹⊩ Neg p # Neg q # z› |
  BeImp: ‹⊩ Neg (Imp p q) # z› if ‹⊩ p # z› and ‹⊩ Neg q # z› |
  BeDis: ‹⊩ Neg (Dis p q) # z› if ‹⊩ Neg p # z› and ‹⊩ Neg q # z› |
  BeCon: ‹⊩ Con p q # z› if ‹⊩ p # z› and ‹⊩ q # z› |
  GaExi: ‹⊩ Exi p # z› if ‹⊩ sub 0 t p # z› |
  GaUni: ‹⊩ Neg (Uni p) # z› if ‹⊩ Neg (sub 0 t p) # z› |
  DeExi: ‹⊩ Neg (Exi p) # z› if ‹⊩ Neg (sub 0 (Fun c []) p) # z› and ‹news c (p # z)› |
  DeUni: ‹⊩ Uni p # z› if ‹⊩ sub 0 (Fun c []) p # z› and ‹news c (p # z)› |
  Extra: ‹⊩ z› if ‹⊩ p # z› and ‹member p z›
```

SeCaV – Notes on Rules

No rules for negation since it is an abbreviation:

```
abbreviation Neg :: ‹fm ⇒ fm› where ‹Neg p ≡ Imp p Falsity›
```

Every regular rule works on the head of the list:

- Easy to write down the rules.
- Works well with the simplifier.

We encourage students to use the following admissible rule instead of `Extra`:

```
theorem Ext: ‹⊩ y› if ‹⊩ z› and ‹ext y z›
primrec ext where
  ‹ext y [] = True› |
  ‹ext y (p # z) = (if member p y then ext y z else False)›
```

Derivations

We use **from** and **with** to bring the applied rules to the forefront.
We use **if ?thesis** to gradually break down the formula towards `Basic` sequents.

```
lemma ‹⊩ [Imp (Pre '''' []) (Pre '''' [])]›
proof -
  from AlphaImp have ?thesis if ‹⊩ [Neg (Pre '''' []), Pre '''' []]›
    using that by simp
  with Ext have ?thesis if ‹⊩ [Pre '''' [], Neg (Pre '''' [])]›
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

The simplifier can handle every application we have encountered but…

Derivations with Substitutions

Sometimes the simplifier needs a bit of help in the form a **where** attribute.

```
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1])))),
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
  ]
  ›
  using that by simp
with GammaExi[where t=‹Fun ''a'' []›] have ?thesis if ‹⊩
  [
    Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Fun ''a'' []]))),
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
  ]
  ›
  using that by simp
```

Duplicating Gamma Formulas

Gamma formulas apply to all instances but our rules “destroy” them.

Solution: Start the derivation by duplicating the formula.

```
proposition ‹∃x. ∀y. p y ∨ ¬ p x› by metis

lemma ‹⊩
  [
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
  ]
  ›
proof -
  from Ext have ?thesis if ‹⊩
    [
      Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1])))),
      Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
    ]
    ›
    using that by simp
```

Exam

24 students did the take-home exam which included nine SeCaV proofs.
In general the solutions were excellent (some were longer than necessary).

```
proposition ‹(p ⟶ q) ⟶ p ⟶ q› by metis
proposition ‹p ⟶ (p ⟶ q) ⟶ q› by metis
proposition ‹p ∧ (p ⟶ q) ⟶ q› by metis
proposition ‹p a ∧ (p a ⟶ (∀x. p x)) ⟶ (∀x. p x)› by metis
proposition ‹(∀x. p x) ⟶ p a› by metis
proposition ‹p ⟶ q ⟶ p› by metis
proposition ‹(p ⟶ q ⟶ r) ⟶ (p ⟶ q) ⟶ p ⟶ r› by metis
proposition ‹(∀x. p x) ⟶ (∃x. p x)› by metis
proposition ‹p ∨ (p ⟶ q)› by metis
```

The final grades for the course were as follows (in the ECTS grading scale):
10 As, 10 Bs, 4 Cs and 2 Fs.

The course evaluation is available online: https://kurser.dtu.dk/course/02256/info

Abridged Bibliography

[1] Stefan Berghofer. First-Order Logic According to Fitting. Archive of Formal Proofs, August 2007. http://isa-afp.org/entries/FOL-Fitting.html, Formal proof development.

[3] Andreas Halkjær From. Formalized Soundness and Completeness of Natural Deduction for First-Order Logic. Tenth Scandinavian Logic Symposium (SLS 2018), http://scandinavianlogic.org/material/book_of_abstracts_sls2018.pdf, 2018.

[4] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. Teaching a Formalized Logical Calculus. In Proceedings of the 8th International Workshop on Theorem proving components for Educational software (ThEdu'19), 2020.

[7] Jørgen Villadsen. ProofJudge: Automated Proof Judging Tool for Learning Mathematical Logic. In Proceedings of the Exploring Teaching for Active Learning in Engineering Education Conference, pages 39–44, Copenhagen, Denmark, 2015.

[8] Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural Deduction and the Isabelle Proof Assistant. In Proceedings of the 6th International Workshop on Theorem proving components for Educational software (ThEdu'17), 2018.

[9] Jørgen Villadsen, Andreas Halkjær From, and Anders Schlichtkrull. Natural Deduction Assistant (NaDeA). In Proceedings of the 7th International Workshop on Theorem proving components for Educational software (ThEdu'18), 2019.

[10] Jørgen Villadsen, Alexander Birch Jensen, and Anders Schlichtkrull. NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle. IFCoLog Journal of Logics and their Applications, 4(1):55–82, 2017.