A Concise Sequent Calculus for Teaching First-Order Logic

From, Asta Halkjær; Villadsen, Jørgen

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

A Concise Sequent Calculus for Teaching First-Order Logic

Asta Halkjær From and Jørgen Villadsen

Technical University of Denmark

Abstract

We have formalized soundness and completeness for a sequent calculus for classical first-order logic in the proof assistant Isabelle/HOL. We first describe a technique for extending a completeness result for closed formulas to completeness for open formulas as well. This requires some tricky reasoning due to our use of de Bruijn indices. We then describe a concise sequent calculus for first-order logic and introduce a useful way to write up the sequent calculus derivations in Isabelle/HOL. Our formalization has recently been used in a computer science course on automated reasoning.

Keywords: First-Order Logic · Sequent Calculus · Isabelle/HOL

1 Introduction

We present a sequent calculus for classical first-order logic formalized in the proof assistant Isabelle/HOL. The calculus is sound and complete as it can show exactly the same formulas as our Natural Deduction Assistant. The contributions of this paper are:

- An overview of our work on the Natural Deduction Assistant (NaDeA) and a related Sequent Calculus Verifier (SeCaV), both used in our courses (Section 2).
- A technique for extending a completeness result for closed formulas to completeness for open formulas as well (Section 3).
- A concise sequent calculus for first-order logic, including a way to write up derivations that we have used in teaching (Sections 4 and 5).

Our work is part of the IsaFoL (Isabelle Formalization of Logic) project that aims at developing formalizations in Isabelle/HOL of logics, proof systems, and automatic/interactive provers [2]. The formalization can be obtained from the NaDeA online web application as described in Section 2 (in total 6498 lines in Isabelle/HOL).

2 Our Previous Work

There are several parts to our Natural Deduction Assistant (NaDeA).

First of all, there is a formalization in Isabelle of the syntax and semantics of classical first-order logic and a natural deduction proof system. Our syntax has falsity as a primitive and defines negation in terms of this and implication. We represent constants as functions taking no arguments and variables as natural numbers referring to them using de Bruijn indices. The proof rules are defined inductively and we have formalized proofs of soundness and completeness.

Since the proof system is formalized in Isabelle we can be extremely precise about the side conditions of rules and the substitution procedure [8,10].

Besides the formalization, NaDeA consists of an online web application:

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

The formalization document `Natural_Deduction_Assistant.thy` can be obtained by clicking on the verification button in the top right corner when the help window has been cancelled. Note that the verification button itself shows the number of ¤ symbols in the current proof state (initially 1) and if this becomes 0 then the proof is finished and a proof in Isabelle/HOL is generated. The formalization document `Natural_Deduction_Assistant.thy` is found in the so-called "Base theory" tab.

The online web application is written in the TypeScript programming language and allows the user to input a formula and to prove it using the proof system. It has several features [10]:

- The user is only presented with rules applicable to solve the chosen subgoal.
- The application automatically keeps track of assumptions and appeals to them.
- Side conditions of quantifier rules are checked, i.e. that Skolem constants are new.
- The user can undo and redo to any previous state.
- Proofs, whether finished or in-progress, can be exported to a textual format that can be loaded again later or on another computer. This preserves all the steps taken in the proof.
- The user can switch back and forth between the standard notation and the abstract syntax used in the formalization.
- A version of the formalization can be viewed inside the application with comments alongside the proof system definitions and the soundness proof.

Another aspect of the web application is a system dubbed ProofJudge [7]. This allows the teacher of a logic course to pose formulas as exercises or assignments that the students can access, try to prove, and hand in. The teacher and teaching assistants can then see how many people have solved a given task, see individual solutions and so on. ProofJudge integrates with the assistant so that you can see how many steps a student used to prove a given formula or how many subgoals are still left. The system also allows students to save their work online and return to it later [7].

While ProofJudge is designed to facilitate the human grading of submissions, we also run an automatic tableau prover in the background as the user works on their proof. This prover checks every current subgoal and if it seems like the subgoal is unprovable, it unobtrusively gives a warning by making the corresponding line number orange [8]. We have verified the soundness of the prover's kernel in Isabelle [5,6] and used the code export facilities to generate SML code for the full prover. From this we generate the JavaScript that runs on the page [8].

In the direction from NaDeA to Isabelle we have a feature that allows users to export their finished online proofs to a corresponding proof in the Isabelle formalization. There, each rule application including side conditions is checked by the proof assistant and our formalized soundness proof then guarantees the validity of the formula [8]. The possibility to export proofs in this way mitigates the fact that the web application is not formally verified: If you are in doubt of the validity of your proof you can export it to Isabelle and have it checked there [8].

We have evaluated our use of NaDeA in the classroom and note that a problem for small proofs is that students can potentially find them by just clicking blindly but that this is not a problem for more complicated examples [9]. One feature appreciated by several students is the ability to access any previous state through undoing and redoing, a feature that is not present in most applications where taking an action after undoing makes it impossible to go back. A number of examples and hints are available in the system to get started [9].

As counterpart to our Natural Deduction Assistant (NaDeA) we also have a Sequent Calculus Verifier (SeCaV) with a formalization in Isabelle [4]. We use exactly the same syntax as in NaDeA but the proof system is a one-sided sequent calculus. The proof of completeness is derived from a completeness proof for a tableau system whose rules are the dual of the sequent calculus. These tableau rules correspond closely to the consistency property conditions used in a formalization by Berghofer [1], allowing us to easily show completeness for closed formulas by building on that work. We then show completeness of the sequent calculus by translating any closing tableau into a sequent calculus derivation [4]. We have used the SeCaV system for teaching [4].

3 Open Formulas

In this section we show how to extend a completeness result for sentences to one for open formulas. The context is natural deduction but we have applied the same technique for tableau proofs. We presented this technique at the Tenth Scandinavian Logic Symposium in 2018 [3] (abstract only).

We illustrate the five-step technique by showing the lemma that combines them. The result we prove is the following:

```
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 −
```

That is, we assume that the formula p is valid under assumptions z and that the domain is denumerable, and need to show that we can derive $OK\ p\ z$.

1. First, we reduce the problem by turning the assumptions into implications in the syntax:

```
  let ?p = ⟨put-imps p (rev z)⟩
```

Importantly, this preserves validity:

```
  have ∗: ⟨∀(e :: - ⇒ 'a) f g. semantics e f g ?p⟩
    using assms(1) semantics-put-imps by fastforce
```

2. Next, we universally close the formula by putting a sufficient number of universal quantifiers in front and note that this too preserves validity:

```
  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
```

3. The resulting formula is a valid sentence so we know from the existing completeness result that it can be derived:

ultimately have ‹*OK* (*put-unis m ?p*) []›
using *assms*(*2*) *sentence-completeness* **by** *blast*

4. By working within the proof system we can then derive the open formula:

then have ‹*OK ?p* []›
using ∗∗ *remove-unis-sentence* **by** *blast*

5. And finally we use a deduction theorem to turn the introduced implications into assumptions for the judgement.

then show ‹*OK p z*›
using *remove-imps* **by** *fastforce*
qed

Steps 1 through 3 are not particularly difficult so we focus instead on steps 4 and 5.

3.1 Deriving the open formula

We have a derivation of a formula with m universal quantifiers in front and want a derivation without them. We could use the *Uni-E* rule to substitute the original variables for the freshly quantified ones but this requires some tricky reasoning due to our use of de Bruijn indices. The following example starts from the formula $\forall\forall\forall p(0,1,2)$ and illustrates how the variables shift when eliminating the quantifiers:

$$\begin{aligned}(\forall\forall p(0,1,2))[2/0] &\rightsquigarrow \forall((\forall p(0,1,2))[3/1]) \rightsquigarrow \forall\forall(p(0,1,2)[4/2]) \rightsquigarrow \forall\forall p(0,1,4)\\ (\forall p(0,1,4))[1/0] &\rightsquigarrow \forall(p(0,1,4)[2/1]) \rightsquigarrow \forall p(0,2,3)\\ p(0,2,3)[0/0] &\rightsquigarrow p(0,1,2)\end{aligned}$$

To avoid having to reason about these shifts, we instead eliminate the universal quantifiers with fresh constants. The function *subc c s p* replaces occurrences of c with s in p, adjusting the variables in s when substituting under a quantifier. It is admissible to perform such a substitution uniformly across the formula and assumptions of a derivation:

lemma *OK-subc*: ‹*OK p z* ⟹ *OK* (*subc c s p*) (*subcs c s z*)›
(proof omitted)

The proof by induction on the derivation is mechanical, except for the quantifier rules where care must be taken to treat the involved constants correctly. The following renaming result is useful. Here *psubst* applies a function to every constant/function symbol:

lemma *OK-psubst*: ‹*OK p z* ⟹ *OK* (*psubst f p*) (*map* (*psubst f*) *z*)›
(proof omitted)

When we compose closure elimination (*sub*) with constant substitution (*subc*) in the right order we get the following telescoping sequence of substitutions:

$$\textit{subc}\ c_0\ \textit{(m-1)}\ (\textit{subc}\ c_1\ \textit{(m-2)}\ (\ldots\ (\textit{subc}\ c_{m-1}\ \textit{0}\ (\textit{sub}\ \textit{0}\ c_{m-1}\ \ldots))))$$

Each introduced constant is immediately substituted with the correct variable and since subsequent substitutions are of constants, they do not adjust previously inserted variables.

We can then prove our desired result:

lemma *remove-unis-sentence*:
assumes ‹*sentence* (*put-unis m p*)› ‹*OK* (*put-unis m p*) []›
shows ‹*OK p* []›
(proof omitted)

3.2 Shifting implications

Step 5 of our technique is to derive *OK p z* from *OK (put-imps p (rev z)) []*. We do so with the following lemma that shifts just one formula from an implication to the assumptions:

lemma *shift-imp-assum*:
assumes ‹*OK* (*Imp p q*) *z*›
shows ‹*OK q* (*p* # *z*)›
(proof omitted)

It is proved by weakening *z* with *p* and applying modus ponens, *Imp-E*. The weakening is shown by induction over the inference rules:

lemma *weaken-assumptions*: ‹*OK p z* ⟹ *set z* ⊆ *set z'* ⟹ *OK p z'*›
(proof omitted)

The proof is trivial except for the cases for *Exi-E* and *Uni-I*, where the Skolem constant fixed by the induction hypothesis is only new to the smaller set of premises, not necessarily the larger ones. Again, it is necessary to perform renaming using *psubst*.

We can now shift a list of implications by induction on the list:

lemma *remove-imps*: ‹*OK* (*put-imps p z*) *z'* ⟹ *OK p* (*rev z* @ *z'*)›
using *shift-imp-assum* **by** (*induct z arbitrary*: *z'*) *simp-all*

3.3 Completeness

In conclusion, we have completeness for any valid formula, open or closed:

theorem *completeness*:
assumes ‹∀ (*e* :: - ⇒ 'a) *f g*. *semantics e f g p*›
and ‹*denumerable* (*UNIV* :: 'a *set*)›
shows ‹*OK p* []›
using *assms* **by** (*simp add*: *completeness'*)

3.4 Tableau

We have applied the same technique to the tableau calculus from which we derive the completeness of our sequent calculus but with one modification: Instead of closing the formula with universal quantifiers, we close it by substituting fresh constants for the free variables. At that point we are still operating semantically, so we only need to show that this preserves validity and doing so is simple. We still make use of the telescoping sequence of substitutions but save the introduction of the universal quantifiers.

4 The Sequent Calculus

We represent sequents as lists of formulas. The calculus is one-sided and as such has rules not just for each connective and quantifier but also each one negated. The proof system is given in Figure 1. It is important to note that *Neg* is not primitive but defined as *Neg p* ≡ *Imp p Falsity*. This is why we can make do without any rule for double negation; it is covered by the implication rules.

```
inductive sequent-calculus (⟨⊩ -⟩ 0) where
  Basic: ⟨⊩ p # z⟩ if ⟨member (Neg p) z⟩ |
  AlDis: ⟨⊩ Dis p q # z⟩ if ⟨⊩ p # q # z⟩ |
  AlImp: ⟨⊩ Imp p q # z⟩ if ⟨⊩ Neg p # q # z⟩ |
  AlCon: ⟨⊩ Neg (Con p q) # z⟩ if ⟨⊩ Neg p # Neg q # z⟩ |
  BeCon: ⟨⊩ Con p q # z⟩ if ⟨⊩ p # z⟩ and ⟨⊩ 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⟩ |
  GaExi: ⟨⊩ Exi p # z⟩ if ⟨⊩ sub 0 t p # z⟩ |
  GaUni: ⟨⊩ Neg (Uni p) # z⟩ if ⟨⊩ Neg (sub 0 t p) # z⟩ |
  DeUni: ⟨⊩ Uni p # z⟩ if ⟨⊩ sub 0 (Fun c []) p # z⟩ and ⟨news c (p # z)⟩ |
  DeExi: ⟨⊩ Neg (Exi p) # z⟩ if ⟨⊩ Neg (sub 0 (Fun c []) p) # z⟩ and ⟨news c (p # z)⟩ |
  Extra: ⟨⊩ z⟩ if ⟨⊩ p # z⟩ and ⟨member p z⟩
```

Figure 1: The sequent calculus.

The rules follow a particular naming scheme depending on their type. Propositional rules that do not branch are called α-rules and start with *Al*. Propositional rules that branch start with *Be* for β. The *Ga*-rules, for γ, operate on quantified formulas that can be built from arbitrary instances, i.e. existential statements and negated universals. Finally, to derive a formula with the *De*-rules, for δ, the constant used in the derived instance must be new. Note that all the rules work on the first formula in the sequent. This makes the rule easier to state and easier for the simplifier to work with which in turn makes formalizing derivations smoother.

The *Basic* axiom allows us to derive any sequent whose head occurs negated somewhere in the tail.

The derived *Neg* rule allows us to remove double negations:

```
theorem Neg: ⟨⊩ Neg (Neg p) # z⟩ if ⟨⊩ p # z⟩
  (proof omitted)
```

The *Extra* rule allows us to drop a head that already exists elsewhere. In practice it is more useful to use the admissible *Ext* rule allowing us to rearrange, contract and add formulas:

```
theorem Ext: ⟨⊩ y⟩ if ⟨⊩ z⟩ and ⟨ext y z⟩
  (proof omitted)
```

Here, *ext y z* expresses that y is an extension of z in that it contains all the formulas that z does and possibly more:

```
primrec ext where
  ⟨ext y [] = True⟩ |
  ⟨ext y (p # z) = (if member p y then ext y z else False)⟩
```

The function *member* is defined straightforwardly:

```
primrec member :: ⟨fm ⇒ fm list ⇒ bool⟩ where
  ⟨member p [] = False⟩ |
  ⟨member p (q # z) = (if p = q then True else member p z)⟩
```

The *ext* relation is equivalent to the subset relation:

lemma *member* [*simp*]: ‹*member p z* ⟷ *p* ∈ *set z*›
by (*induct z*) *simp-all*

lemma *ext* [*simp*]: ‹*ext y z* ⟷ *set z* ⊆ *set y*›
by (*induct z*) *simp-all*

We show soundness and completeness of the calculus by proving that our calculus can simulate our SeCaV calculus and vice versa. As mentioned, we know that the latter is sound and complete [4].

4.1 Relation to NaDeA

Figure 2 shows the NaDeA proof system where *OK p z* means that *p* can be derived from assumptions *z*.

inductive *OK* :: ‹*fm* ⇒ *fm list* ⇒ *bool*› **where**
Assume: ‹*member p z* ⟹ *OK p z*› |
Boole: ‹*OK Falsity* (*Imp p Falsity* # *z*) ⟹ *OK p z*› |
Imp-E: ‹*OK* (*Imp p q*) *z* ⟹ *OK p z* ⟹ *OK q z*› |
Imp-I: ‹*OK q* (*p* # *z*) ⟹ *OK* (*Imp p q*) *z*› |
Dis-E: ‹*OK* (*Dis p q*) *z* ⟹ *OK r* (*p* # *z*) ⟹ *OK r* (*q* # *z*) ⟹ *OK r z*› |
Dis-I1: ‹*OK p z* ⟹ *OK* (*Dis p q*) *z*› |
Dis-I2: ‹*OK q z* ⟹ *OK* (*Dis p q*) *z*› |
Con-E1: ‹*OK* (*Con p q*) *z* ⟹ *OK p z*› |
Con-E2: ‹*OK* (*Con p q*) *z* ⟹ *OK q z*› |
Con-I: ‹*OK p z* ⟹ *OK q z* ⟹ *OK* (*Con p q*) *z*› |
Exi-E: ‹*OK* (*Exi p*) *z* ⟹ *OK q* (*sub 0* (*Fun c* []) *p* # *z*) ⟹ *news c* (*p* # *q* # *z*) ⟹ *OK q z*› |
Exi-I: ‹*OK* (*sub 0 t p*) *z* ⟹ *OK* (*Exi p*) *z*› |
Uni-E: ‹*OK* (*Uni p*) *z* ⟹ *OK* (*sub 0 t p*) *z*› |
Uni-I: ‹*OK* (*sub 0* (*Fun c* []) *p*) *z* ⟹ *news c* (*p* # *z*) ⟹ *OK* (*Uni p*) *z*›

Figure 2: The NaDeA proof system.

The following theorem relates NaDeA and the sequent calculus. The sequent calculus is one-sided, so the sequent corresponding to the NaDeA judgement has negated assumptions:

theorem *OK-sequent-calculus*: ‹*OK p z* ⟷ (⊩ *p* # *map Neg z*)›
(proof omitted)

As a corollary we can consider the case of no assumptions:

corollary ‹*OK p* [] ⟷ (⊩ [*p*])›
unfolding *OK-sequent-calculus* **by** *simp*

The relation is derived through the soundness and completeness of both systems instead of via translation between them.

5 Derivations

Let us look at some derivations in the calculus. Consider the following formula:

proposition ‹*p a* ⟶ *p a*› **by** *metis*

Converting to our abstract syntax we can start the derivation like so:

```
lemma ‹⊩
  [
    Imp (Pre ''p'' [Fun ''a'' []]) (Pre ''p'' [Fun ''a'' []])
  ]
  ›
proof −
```

We can neatly derive this formula by combining the **from**, **with**, **have** and **if** commands in Isabelle with the abbreviation for the goal, *?thesis*. Thus, we can apply rules breaking down the formula until we reach a sequent covered by *Basic*. First we apply the *AlphaImp* rule giving the new subgoal:

```
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Pre ''p'' [Fun ''a'' []]),
      Pre ''p'' [Fun ''a'' []]
    ]
    ›
    using that by simp
```

Next we apply the *Ext* rule to swap the order of the two resulting formulas:

```
  with Ext have ?thesis if ‹⊩
    [
      Pre ''p'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    ›
    using that by simp
```

And by doing so we have arrived at a *Basic* sequent, completing the derivation:

```
  with Basic show ?thesis
    by simp
qed
```

An interesting feature of our proof system is that the γ-rules are "destroyed" when we instantiate them in the sub-derivation. For some proofs, however, you need several instances of the same formula. The partial derivation in Figure 3 is such an example. In this case we can start off by using the *Ext* rule to duplicate the formula, allowing us effectively to instantiate it twice. Or if we view the derivation as going from the axioms towards the final formula, we end the derivation by contracting the two copies using *Ext*.

Another thing worth pointing out about the example in Figure 3 is the application of the *GammaExi* rule. The simplifier, as invoked by *simp*, is powerful enough to handle every rule application in our derivations except for some rule applications involving substitution. In those cases we need to explicitly instantiate the rule with the term used in the substitution by using the **where** attribute.

The full version of Figure 3 is given in the Appendix alongside two other examples. These also showcase the use of **and** to manage branching derivations.

```
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
  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
  ⋮
```

Figure 3: A γ-formula can be instantiated twice by duplicating it.

6 Conclusion

Our formalization has recently been used in a computer science course on automated reasoning with 27 students.

A really difficult challenge for manual proof in first-order logic is discussed on page 128 of the Handbook of Tableau Methods (Kluwer Academic Publishers 1999):

> *If every person that is not rich has a rich father, then some rich person must have a rich grandfather.*

Formalization with r (*rich*) and f (*father*):

$$\forall x(\neg r(x) \rightarrow r(f(x))) \rightarrow \exists x(r(x) \wedge r(f(f(x))))$$

Only one student managed to complete the proof for the challenge. Afterwards we reduced the proof to 19 steps in the Sequent Calculus Verifier (SeCaV) and these steps take up 466 lines in Isabelle/HOL in the format shown in the previous section and in the Appendix. We have a solution in the Natural Deduction Assistant (NaDeA) with 42 steps and 162 clicks in the web application [9].

Besides this difficult challenge we ask the students to work on 24 formulas and their proofs. For the take-home exam from 7 May 2020 to 3 June 2020 the students are asked, among other things, to prove the following 9 formulas using the formalization in Isabelle/HOL:

proposition ⟨$(p \longrightarrow q) \longrightarrow p \longrightarrow q$⟩ **by** *metis*

proposition ⟨$p \longrightarrow (p \longrightarrow q) \longrightarrow q$⟩ **by** *metis*

proposition ‹$p \wedge (p \longrightarrow q) \longrightarrow q$› **by** *metis*

proposition ‹$p\ a \wedge (p\ a \longrightarrow (\forall x.\ p\ x)) \longrightarrow (\forall x.\ p\ x)$› **by** *metis*

proposition ‹$(\forall x.\ p\ x) \longrightarrow p\ a$› **by** *metis*

proposition ‹$p \longrightarrow q \longrightarrow p$› **by** *metis*

proposition ‹$(p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r$› **by** *metis*

proposition ‹$(\forall x.\ p\ x) \longrightarrow (\exists x.\ p\ x)$› **by** *metis*

proposition ‹$p \vee (p \longrightarrow q)$› **by** *metis*

Acknowledgements

We thank Alexander Birch Jensen and Anders Schlichtkrull for discussions.

References

[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.

[2] Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 1–13, 2019.

[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.

[5] Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull, and Jørgen Villadsen. Programming and verifying a declarative first-order prover in Isabelle/HOL. *AI Communications*, 31(3):281–299, 2018.

[6] Alexander Birch Jensen, Anders Schlichtkrull, and Jørgen Villadsen. First-Order Logic According to Harrison. *Archive of Formal Proofs*, January 2017. http://isa-afp.org/entries/FOL_Harrison.html, Formal proof development.

[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.

Appendix: Example Derivations

Full versions of 3 proofs discussed in the paper.

```
proposition ‹(∀ x. p x) ⟶ p a ∧ p b› by metis

lemma ‹⊩
  [
    Imp (Uni (Pre ''p'' [Var 0])) (Con (Pre ''p'' [Fun ''a'' []]) (Pre ''p'' [Fun ''b'' []]))
  ]
  ›
proof −
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Uni (Pre ''p'' [Var 0])),
      Con (Pre ''p'' [Fun ''a'' []]) (Pre ''p'' [Fun ''b'' []])
    ]
    ›
    using that by simp
  with Ext have ?thesis if ‹⊩
    [
      Con (Pre ''p'' [Fun ''a'' []]) (Pre ''p'' [Fun ''b'' []]),
      Neg (Uni (Pre ''p'' [Var 0]))
    ]
    ›
    using that by simp
  with BetaCon have ?thesis if ‹⊩
    [
      Pre ''p'' [Fun ''a'' []],
      Neg (Uni (Pre ''p'' [Var 0]))
    ]
    › and ‹⊩
    [
      Pre ''p'' [Fun ''b'' []],
      Neg (Uni (Pre ''p'' [Var 0]))
    ]
    ›
    using that by simp
  with Ext have ?thesis if ‹⊩
    [
      Neg (Uni (Pre ''p'' [Var 0])),
      Pre ''p'' [Fun ''a'' []]
    ]
    › and ‹⊩
    [
      Neg (Uni (Pre ''p'' [Var 0])),
      Pre ''p'' [Fun ''b'' []]
    ]
    ›
    using that by simp
```

```
  with GammaUni have ?thesis if ‹⊩
    [
      Neg (Pre ''p'' [Fun ''a'' []]),
      Pre ''p'' [Fun ''a'' []]
    ]
    › and ‹⊩
    [
      Neg (Pre ''p'' [Fun ''b'' []]),
      Pre ''p'' [Fun ''b'' []]
    ]
    ›
    using that by simp
  with Ext have ?thesis if ‹⊩
    [
      Pre ''p'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    › and ‹⊩
    [
      Pre ''p'' [Fun ''b'' []],
      Neg (Pre ''p'' [Fun ''b'' []])
    ]
    ›
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

```
proposition ‹(∀ x. p x ⟶ q x) ⟶ (∃ x. p x) ⟶ (∃ x. q x)› by metis

lemma ‹⊩
  [
    Imp
      (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0])))
      (Imp (Exi (Pre ''p'' [Var 0])) (Exi (Pre ''q'' [Var 0])))
  ]
  ›
proof −
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0]))),
      Imp (Exi (Pre ''p'' [Var 0])) (Exi (Pre ''q'' [Var 0]))
    ]
    ›
    using that by simp
  with Ext have ?thesis if ‹⊩
    [
      Imp (Exi (Pre ''p'' [Var 0])) (Exi (Pre ''q'' [Var 0])),
      Neg (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0])))
    ]
    ›
    using that by simp
```

```
with AlphaImp have ?thesis if ⟨⊩
  [
    Neg (Exi (Pre ''p'' [Var 0])),
    Exi (Pre ''q'' [Var 0]),
    Neg (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0])))
  ]
  ⟩
  using that by simp
with DeltaExi have ?thesis if ⟨⊩
  [
    Neg (Pre ''p'' [Fun ''a'' []]),
    Exi (Pre ''q'' [Var 0]),
    Neg (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0])))
  ]
  ⟩
  using that by simp
with Ext have ?thesis if ⟨⊩
  [
    Neg (Uni (Imp (Pre ''p'' [Var 0]) (Pre ''q'' [Var 0]))),
    Neg (Pre ''p'' [Fun ''a'' []]),
    Exi (Pre ''q'' [Var 0])
  ]
  ⟩
  using that by simp
with GammaUni have ?thesis if ⟨⊩
  [
    Neg (Imp (Pre ''p'' [Fun ''a'' []]) (Pre ''q'' [Fun ''a'' []])),
    Neg (Pre ''p'' [Fun ''a'' []]),
    Exi (Pre ''q'' [Var 0])
  ]
  ⟩
  using that by simp
with Ext have ?thesis if ⟨⊩
  [
    Exi (Pre ''q'' [Var 0]),
    Neg (Imp (Pre ''p'' [Fun ''a'' []]) (Pre ''q'' [Fun ''a'' []])),
    Neg (Pre ''p'' [Fun ''a'' []])
  ]
  ⟩
  using that by simp
with GammaExi have ?thesis if ⟨⊩
  [
    Pre ''q'' [Fun ''a'' []],
    Neg (Imp (Pre ''p'' [Fun ''a'' []]) (Pre ''q'' [Fun ''a'' []])),
    Neg (Pre ''p'' [Fun ''a'' []])
  ]
  ⟩
  using that by simp
```

```
  with Ext have ?thesis if ⟨⊩
    [
      Neg (Imp (Pre ''p'' [Fun ''a'' []]) (Pre ''q'' [Fun ''a'' []])),
      Pre ''q'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    ⟩
    using that by simp
  with BetaImp have ?thesis if ⟨⊩
    [
      Pre ''p'' [Fun ''a'' []],
      Pre ''q'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    ⟩ and ⟨⊩
    [
      Neg (Pre ''q'' [Fun ''a'' []]),
      Pre ''q'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    ⟩
    using that by simp
  with Ext have ?thesis if ⟨⊩
    [
      Pre ''p'' [Fun ''a'' []],
      Neg (Pre ''p'' [Fun ''a'' []])
    ]
    ⟩ and ⟨⊩
    [
      Pre ''q'' [Fun ''a'' []],
      Neg (Pre ''q'' [Fun ''a'' []])
    ]
    ⟩
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

```
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
```

```
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
with DeltaUni have ?thesis if ⟨⊩
  [
    Dis (Pre ''p'' [Fun ''b'' []]) (Neg (Pre ''p'' [Fun ''a'' []])),
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
  ]
  ⟩
  using that by simp
with AlphaDis have ?thesis if ⟨⊩
  [
    Pre ''p'' [Fun ''b'' []],
    Neg (Pre ''p'' [Fun ''a'' []]),
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1]))))
  ]
  ⟩
  using that by simp
with Ext have ?thesis if ⟨⊩
  [
    Exi (Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Var 1])))),
    Pre ''p'' [Fun ''b'' []]
  ]
  ⟩
  using that by simp
with GammaExi[where t=⟨Fun ''b'' []⟩] have ?thesis if ⟨⊩
  [
    Uni (Dis (Pre ''p'' [Var 0]) (Neg (Pre ''p'' [Fun ''b'' []]))),
    Pre ''p'' [Fun ''b'' []]
  ]
  ⟩
  using that by simp
with DeltaUni have ?thesis if ⟨⊩
  [
    Dis (Pre ''p'' [Fun ''c'' []]) (Neg (Pre ''p'' [Fun ''b'' []])),
    Pre ''p'' [Fun ''b'' []]
  ]
  ⟩
  using that by simp
with AlphaDis have ?thesis if ⟨⊩
  [
    Pre ''p'' [Fun ''c'' []],
    Neg (Pre ''p'' [Fun ''b'' []]),
    Pre ''p'' [Fun ''b'' []]
  ]
  ⟩
  using that by simp
```

```
  with Ext have ?thesis if ⟨⊩⊢
    [
      Pre ''p'' [Fun ''b'' []],
      Neg (Pre ''p'' [Fun ''b'' []])
    ]
    ⟩
    using that by simp
  with Basic show ?thesis
    by simp
qed
```