NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle

Villadsen, Jørgen; Jensen, Alexander Birch; Schlichtkrull, Anders

Published in:
IfCoLog Journal of Logics and their Applications

Publication date:
2017

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

Link back to DTU Orbit

Citation (APA):
Villadsen, J., Jensen, A. B., & Schlichtkrull, A. (2017). NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle. IfCoLog Journal of Logics and their Applications, 4(1), 55-82.

NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle

Jørgen Villadsen
DTU Compute - Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Richard Petersens Plads, Building 324,
DK-2800 Kongens Lyngby, Denmark
`jovi@dtu.dk`

Alexander Birch Jensen
DTU Compute - Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Richard Petersens Plads, Building 324,
DK-2800 Kongens Lyngby, Denmark

Anders Schlichtkrull
DTU Compute - Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Richard Petersens Plads, Building 324,
DK-2800 Kongens Lyngby, Denmark

Abstract

We present a new software tool for teaching logic based on natural deduction. Its proof system is formalized in the proof assistant Isabelle such that its definition is very precise. Soundness of the formalization has been proved in Isabelle. The tool is open source software developed in TypeScript / JavaScript and can thus be used directly in a browser without any further installation. Although developed for computer science bachelor students who are used to study and program concrete computer code in a programming language we consider the approach relevant for a broader audience and for other proof systems as well.

Keywords: Natural Deduction, Formalization, Isabelle Proof Assistant, First-Order Logic, Higher-Order Logic.

The present article substantially extends our presentation and tool demonstration at TTL 2015 (Fourth International Conference on Tools for Teaching Logic, Rennes, France, 9–12 June 2015). In particular, a new section with elaboration on the formalization in Isabelle has been added. We would like to thank Stefan Berghofer, Jasmin Christian Blanchette, Mathias Fleury and Dmitriy Traytel for discussions about the formalization of logical inference systems in Isabelle. We would also like to thank Andreas Halkjær From, Andreas Viktor Hess, John Bruntse Larsen, Ashokaditya Mohanty and the referees for comments on the paper.

1 Introduction

In this paper we present the NaDeA software tool. First, we provide the motivation and a short description. We then present the natural deduction system as it is done in a popular textbook [15] and as is it done in NaDeA by looking at its formalization in Isabelle. This illustrates the differences between the two approaches. We also present the semantics of first-order logic as formalized in Isabelle. Thereafter we explain how NaDeA is used to construct a natural deduction proof. After that, we explain the soundness proof of the natural deduction proof system in Isabelle. Lastly, we compare NaDeA to other natural deduction assistants and consider how NaDeA could be improved.

1.1 Motivation

We have been teaching a bachelor logic course — with logic programming — for a decade using a textbook with emphasis on tableaux and resolution [1]. We have started to use the proof assistant Isabelle [2] and refutation proofs are less preferable here. The proof system of natural deduction [3, 4, 5, 15] with the introduction and elimination rules as well as a discharge mechanism seems more suitable. The natural deduction proof system is widely known, used and studied among logicians throughout the world. However, our experience shows that many of our computer science bachelor students struggle to understand the most difficult aspects.

This also goes for other proof systems. We find that teaching logic to computer science bachelor students can be hard because in our case they do not have a strong theoretical mathematical background. Instead, most students are good at understanding concrete computer code in a programming language. The syntax used in Isabelle is in many ways similar to a programming language. A clear and explicit formalization of first-order logic and a proof system may help the students in understanding important details.

We find it important to teach both the semantics of first-order logic and the soundness proof to bachelor students. In the present course the formal semantics as well as the soundness proof in Isabelle are presented to the students. The formalization is also available online in NaDeA and the entire Isabelle file is available in NaDeA too. However, in the present course the students are not expected to be able to construct such a formalization in Isabelle from scratch.

The proof assistant Isabelle is different from a programming language because the expressions are not necessarily computable. For instance, quantifications over infinite domains are possible.

1.2 The Tool

We present the natural deduction assistant NaDeA with a formalization of its proof system in the proof assistant Isabelle. It can be used directly in a browser without any further installation and is available here:

`http://nadea.compute.dtu.dk/`

NaDeA is open source software developed in TypeScript / JavaScript and stored on GitHub. The formalization of its proof system in Isabelle is available here:

`http://logic-tools.github.io/`

Once NaDeA is loaded in the browser — about 250 KB with the jQuery Core library — no internet connection is required. Therefore NaDeA can also be stored locally.

We present the proof in an explicit code format that is equivalent to the Isabelle syntax, but with a few syntactic differences to make it easier to understand for someone trying to learn Isabelle. In this format, we present the proof in a style similar to that of Fitch's diagram proofs. We avoid the seemingly popular Gentzen's tree style to focus less on a visually pleasing graphical representation that is presumably much more challenging to implement.

We find that the following requirements constitute the key ideals for any natural deduction assistant. It should be:

- Easy to use.
- Clear and explicit in every detail of the proof.
- Based on a formalization that can be proved at least sound, but preferably also complete.

Based on this, we saw an opportunity to develop NaDeA which offers help for new users, but also serves to present an approach that is relevant to the advanced users.

In a paper considering the tools developed for teaching logic over the last decade [14, p. 137], the following is said about assistants (not proof assistants like Isabelle but tools for learning/teaching logic):

> Assistants are characterized by a higher degree of interactivity with the user. They provide menus and dialogues to the user for interaction purposes. This kind of tool gives the students the feeling that they are being helped in building the solution. They provide error messages and hints

> in the guidance to the construction of the answer. Many of them usually offer construction of solution in natural deduction proofs. [...] They are usually free licensed and of open access.

We think that this characterization in many ways fits NaDeA. While NaDeA might not bring something new to the table in the form of delicate graphical features, we emphasize the fact that it has some rather unique features such as a formalization of its proof system in Isabelle.

2 Natural Deduction in a Textbook

We consider natural deduction as presented in a popular textbook on logic in computer science [15]. First, we take a look at substitution, which is central to the treatment of quantifiers in natural deduction.

2.1 On Substitution

The following definition for substitution is used in [15, p. 105 top]:

> Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

The usual side conditions that come with rules using this substitution seem to be omitted, but we are shortly after [15, p. 106 top] given the following definition of what it means that 't must be free for x in ϕ':

> Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

The following quote [15, p. 106 bottom] emphasizes the side conditions:

> It might be helpful to compare 't is free for x in ϕ' with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.

In our formalization such notions and their complications become easier to explain because all side conditions of the rules are very explicitly stated. We see it as one of the major advantages of presenting this formalization to students.

2.2 Natural Deduction Rules

We now present the natural deduction rules as described in the literature, again using [15]. The first 9 are rules for classical propositional logic and the last 4 are for first-order logic. Intuitionistic logic can be obtained by omitting the rule *PBC* (proof by contradiction, called "Boole" later) and adding the ⊥-elimination rule (also known as the rule of explosion) [16]. The rules are as follows:

$$\frac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\ PBC \qquad \frac{\phi \qquad \phi \to \psi}{\psi}\ \to E \qquad \frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \to \psi}\ \to I$$

$$\frac{\phi \vee \psi \qquad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \qquad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\ \vee E \qquad \frac{\phi}{\phi \vee \psi}\ \vee I_1 \qquad \frac{\psi}{\phi \vee \psi}\ \vee I_2$$

$$\frac{\phi \wedge \psi}{\phi}\ \wedge E_1 \qquad \frac{\phi \wedge \psi}{\psi}\ \wedge E_2 \qquad \frac{\phi \qquad \psi}{\phi \wedge \psi}\ \wedge I$$

$$\frac{\exists x\, \phi \qquad \boxed{\begin{array}{cc}x_0 & \phi\,[x_0/x]\\ & \vdots\\ & \chi\end{array}}}{\chi}\ \exists E \qquad \frac{\phi\,[t/x]}{\exists x\, \phi}\ \exists I$$

$$\frac{\forall x\, \phi}{\phi\,[t/x]}\ \forall E \qquad \frac{\boxed{\begin{array}{cc}x_0 & \\ & \vdots\\ & \phi\,[x_0/x]\end{array}}}{\forall x\, \phi}\ \forall I$$

Side conditions to rules for quantifiers:

$\exists E$: x_0 does not occur outside its box (and therefore not in χ).

$\exists I$: t must be free for x in ϕ.

$\forall E$: t must be free for x in ϕ.

$\forall I$: x_0 is a new variable which does not occur outside its box.

In addition there is a special copy rule [15, p. 20]:

> A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. [...] The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed.

The copy rule is not needed in our formalization due to the way it manages a list of assumptions.

As it can be seen, there are no rules for truth, negation or biimplication, but the following equivalences can be used:

$$\begin{aligned} \top &\equiv \bot \rightarrow \bot \\ \neg A &\equiv A \rightarrow \bot \\ A \leftrightarrow B &\equiv (A \rightarrow B) \wedge (B \rightarrow A) \end{aligned}$$

The symbols A and B are arbitrary formulas.

3 Natural Deduction in NaDeA

One of the unique features of NaDeA is that it comes with a formalization in Isabelle of the natural deduction proof system, including a proof in Isabelle of the soundness theorem for the proof system. In this section we present the definitions necessary for expressing the soundness theorem and the proof in Isabelle is presented in section 5.

3.1 Syntax for Terms and Formulas

The terms and formulas of the first-order logic language are defined as the datatypes term and formula (later abbreviated tm and fm, respectively). The type identifier represents predicate and function symbols (later abbreviated id).

```
identifier := string

term := Var nat | Fun identifier [term, ..., term]

formula := Falsity | Pre identifier [term, ..., term] | Imp formula formula |
           Dis formula formula | Con formula formula |
           Exi formula | Uni formula
```

Truth, negation and biimplication are abbreviations. In the syntax of our formalization, we refer to variables by use of the de Bruijn indices. That is, instead of identifying a variable by use of a name, usually x, y, z etc., each variable has an index that determines its scope. The use of de Bruijn indices instead of named variables allows for a simple definition of substitution. Furthermore, it also serves the purpose of teaching the students about de Bruijn indices. Note that we are not advocating that de Bruijn indices replace the standard treatment of variables in general. It arguably makes complex formulas harder to read, but the pedagogical advantage is that the notion of scope is practiced.

3.2 Natural Deduction Rules

Provability in NaDeA is defined inductively as follows (OK p z means that the formula p follows from the list of assumptions z and member p z means that p is a member of the list z):

$$\frac{\textsf{member p z}}{\textsf{OK p z}}\;\textsf{Assume} \qquad \frac{\textsf{OK Falsity ((Imp p Falsity) \# z)}}{\textsf{OK p z}}\;\textsf{Boole}$$

$$\frac{\textsf{OK (Imp p q) z} \qquad \textsf{OK p z}}{\textsf{OK q z}}\;\textsf{Imp\_E} \qquad \frac{\textsf{OK q (p \# z)}}{\textsf{OK (Imp p q) z}}\;\textsf{Imp\_I}$$

$$\frac{\textsf{OK (Dis p q) z} \qquad \textsf{OK r (p \# z)} \qquad \textsf{OK r (q \# z)}}{\textsf{OK r z}}\;\textsf{Dis\_E}$$

$$\frac{\textsf{OK p z}}{\textsf{OK (Dis p q) z}}\;\textsf{Dis\_I1} \qquad \frac{\textsf{OK q z}}{\textsf{OK (Dis p q) z}}\;\textsf{Dis\_I2}$$

$$\frac{\textsf{OK (Con p q) z}}{\textsf{OK p z}}\;\textsf{Con\_E1} \qquad \frac{\textsf{OK (Con p q) z}}{\textsf{OK q z}}\;\textsf{Con\_E2} \qquad \frac{\textsf{OK p z} \qquad \textsf{OK q z}}{\textsf{OK (Con p q) z}}\;\textsf{Con\_I}$$

$$\frac{\textsf{OK (Exi p) z} \quad \textsf{OK q ((sub 0 (Fun c []) p) \# z)} \quad \textsf{news c (p\#q\#z)}}{\textsf{OK q z}}\;\textsf{Exi\_E}$$

$$\frac{\textsf{OK (sub 0 t p) z}}{\textsf{OK (Exi p) z}}\;\textsf{Exi\_I}$$

$$\frac{\textsf{OK (Uni p) z}}{\textsf{OK (sub 0 t p) z}}\;\textsf{Uni\_E} \qquad \frac{\textsf{OK (sub 0 (Fun c []) p) z} \qquad \textsf{news c (p \# z))}}{\textsf{OK (Uni p) z}}\;\textsf{Uni\_I}$$

Instead of writing OK p z we could also use the syntax $z \vdash p$, even in Isabelle, but we prefer a more programming-like approach.

The operator # is between the head and the tail of a list. news c l checks if the identifier c does not occur in any of the formulas in the list l and sub n t p returns the formula p where the term t has been substituted for the variable with the de Bruijn index n.

Note that new constants instead of variables not occuring in the assumptions are used in the existential elimination rule and in the universal introduction rule.

In the types we use ⇒ for function spaces. We include the definitions of member, news and sub because they are necessary for the soundness theorem and also for the formalization in section 5:

```
member :: fm ⇒ fm list ⇒ bool

member p [] = False
member p (q # z) = (if p = q then True else member p z)


new_term :: id ⇒ tm ⇒ bool

new_term c (Var n) = True
new_term c (Fun i l) = (if i = c then False else new_list c l)

new_list :: id ⇒ tm list ⇒ bool

new_list c [] = True
new_list c (t # l) = (if new_term c t then new_list c l else False)


new :: id ⇒ fm ⇒ bool

new c Falsity = True
new c (Pre i l) = new_list c l
new c (Imp p q) = (if new c p then new c q else False)
new c (Dis  p q) = (if new c p then new c q else False)
new c (Con p q) = (if new c p then new c q else False)
new c (Exi  p) = new c p
new c (Uni p) = new c p

news :: id ⇒ fm list ⇒ bool

news c [] = True
news c (p # z) = (if new c p then news c z else False)
```

```
inc_term :: tm ⇒ tm

inc_term (Var n) = Var (n + 1)
inc_term (Fun i l) = Fun i (inc_list l)

inc_list :: tm list ⇒ tm list

inc_list [] = []
inc_list (t # l) = inc_term t # inc_list l
```

```
sub_term :: nat ⇒ tm ⇒ tm ⇒ tm

sub_term v s (Var n) = (if n < v then Var n else if n = v then s else Var (n − 1))
sub_term v s (Fun i l) = Fun i (sub_list v s l)

sub_list :: nat ⇒ tm ⇒ tm list ⇒ tm list

sub_list v s [] = []
sub_list v s (t # l) = sub_term v s t # sub_list v s l
```

```
sub :: nat ⇒ tm ⇒ fm ⇒ fm

sub v s Falsity = Falsity
sub v s (Pre i l) = Pre i (sub_list v s l)
sub v s (Imp p q) = Imp (sub v s p) (sub v s q)
sub v s (Dis  p q) = Dis  (sub v s p) (sub v s q)
sub v s (Con p q) = Con (sub v s p) (sub v s q)
sub v s (Exi  p) = Exi  (sub (v + 1) (inc_term s) p)
sub v s (Uni p) = Uni (sub (v + 1) (inc_term s) p)
```

3.3 Semantics for Terms and Formulas

To give meaning to formulas and to prove NaDeA sound we need a semantics of the first-order logic language. We present the semantics below. e is the environment, i.e. a mapping of variables to elements. f maps function symbols to the maps they represent. These maps are from lists of elements of the universe to elements of the universe. Likewise, g maps predicate symbols to the maps they represent. 'a is a type variable that represents the universe. It can be instantiated with any type. For instance, it can be instantiated with the natural numbers, the real numbers or strings.

```
semantics_term :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a

semantics_term e f (Var n) = e n
semantics_term e f (Fun i l) = f i (semantics_list e f l)

semantics_list :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list

semantics_list e f [] = []
semantics_list e f (t # l) = semantics_term e f t # semantics_list e f l

semantics :: (nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒
                                                              fm ⇒ bool

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 (% n. if n = 0 then x else e (n − 1)) f g p)
semantics e f g (Uni p) =
                  (! x. semantics (% n. if n = 0 then x else e (n − 1)) f g p)
```

Most of the cases of semantics should be self-explanatory, but the Uni case is complicated. The details are not important here, but in the case for Uni it uses the universal quantifier (!) of Isabelle's higher-order logic to consider all values of the universe. It also uses the lambda abstraction operator (%) to keep track of the indices of the variables. Likewise, the case for Exi uses the existential quantifier (?) of Isabelle's higher-order logic.

We have proved soundness of the formalization in Isabelle (shown here as a derived rule):

$$\frac{\textsf{OK p []}}{\textsf{semantics e f g p}}\ \text{Soundness}$$

This result makes NaDeA interesting to a broader audience since it gives confidence in the formulas proved using the tool. The proof in Isabelle of the soundness theorem is presented in section 5.

4 Construction of a Proof

We show here how to build and edit proofs in NaDeA. Furthermore, we describe the presentation of proofs in NaDeA.

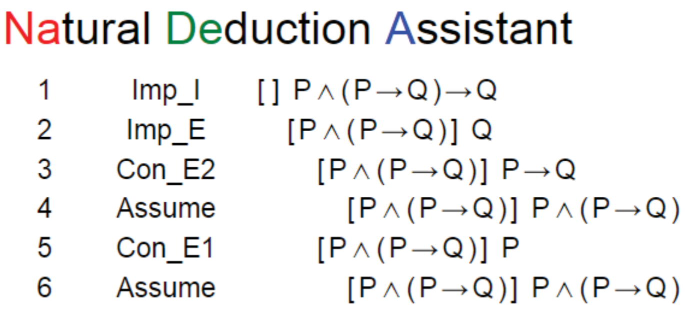
In order to start a proof, you have to start by specifying the goal formula, that is, the formula you wish to prove. To do so, you must enable editing mode by clicking the Edit button in the top menu bar. This will show the underlying proof code and you can build formulas by clicking the red ¤ symbol. Alternatively, you can load a number of tests by clicking the Load button.

At all times, once you have fully specified the conclusion of any given rule, you can continue the proof by selecting the next rule to apply. Again you can do this by clicking the red ¤ symbol. Furthermore, NaDeA allows for undoing and redoing editing steps with no limits.

All proofs are conducted in backward-chaining mode. That is, you must start by specifying the formula that you wish to prove. You then apply the rules inductively until you reach a proof — if you can find one. The proof is finished by automatic application of the Assume rule once the conclusion of a rule is found in the list of assumptions.

To start over on a new proof, you can load the blank proof by using the Load button, or you can refresh the page.

In NaDeA we present any given natural deduction proof (or an attempt at one) in two different types of syntax. One syntax follows the rules as defined in section 3 and is closely related to the formalization in Isabelle, but with a simplified syntax that makes it suitable for teaching purposes. The proof is not built as most often seen in the literature about natural deduction. Usually, for each rule the premises are placed above its conclusion separated by a line. We instead follow the procedure of placing each premise of the rule on separate lines below its conclusion with an additional level of indentation. Here is a screenshot followed by the proof tree:



Natural Deduction Assistant

```
1    Imp_I     [] P∧(P→Q)→Q
2    Imp_E       [P∧(P→Q)] Q
3    Con_E2        [P∧(P→Q)] P→Q
4    Assume          [P∧(P→Q)] P∧(P→Q)
5    Con_E1        [P∧(P→Q)] P
6    Assume          [P∧(P→Q)] P∧(P→Q)
```

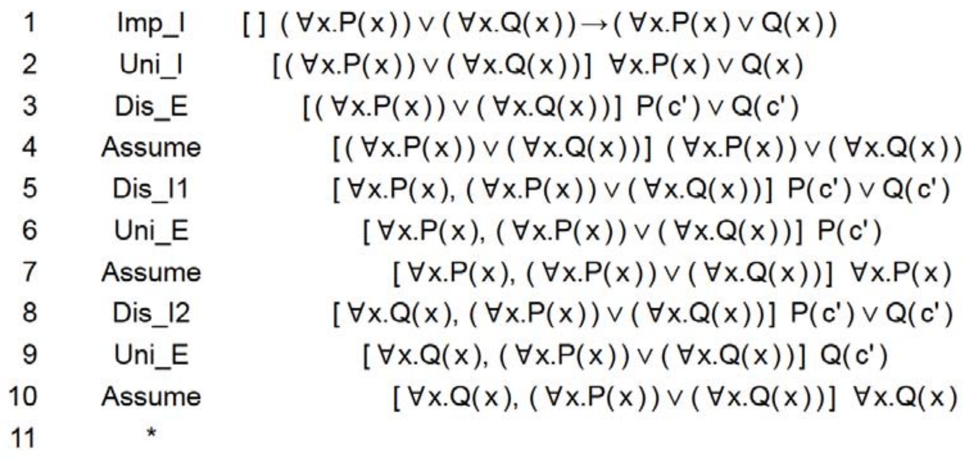
$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \land (p \to q)}\,(1)}{p \to q} \qquad \dfrac{\dfrac{}{p \land (p \to q)}\,(1)}{p}}{q}}{p \land (p \to q) \to q}\,(1)$$

The above proof can also be written in terms of the OK syntax as follows:

```
1  OK (Imp (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" []))) (Pre "Q" [])) []   Imp_I
2    OK (Pre "Q" []) [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]         Imp_E
3      OK (Imp (Pre "P" []) (Pre "Q" []))
         [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                     Con_E2
4        OK (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))
         [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                     Assume
5      OK (Pre "P" []) [Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" []))]         Con_E1
6        OK (Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))
         [(Con (Pre "P" []) (Imp (Pre "P" []) (Pre "Q" [])))]                     Assume
```

So in a way we have the two presentation styles. However, the standard form displayed in the screenshot is always presented and the programming style with the OK syntax is switched on and off with a single click in the browser. The programming style is mandatory when a formula must be entered. We find that the students in general prefer the standard form but also that the switch to the programming style when necessary is rather unproblematic.

For a small but quite interesting example of a proof of a first-order formula consider the following screenshot:



```
1    Imp_I    [ ] ( ∀x.P(x)) ∨ ( ∀x.Q(x)) → ( ∀x.P(x) ∨ Q(x))
2    Uni_I      [( ∀x.P(x)) ∨ ( ∀x.Q(x))]  ∀x.P(x) ∨ Q(x)
3    Dis_E        [( ∀x.P(x)) ∨ ( ∀x.Q(x))]  P(c') ∨ Q(c')
4    Assume         [( ∀x.P(x)) ∨ ( ∀x.Q(x))]  ( ∀x.P(x)) ∨ ( ∀x.Q(x))
5    Dis_I1         [ ∀x.P(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  P(c') ∨ Q(c')
6    Uni_E            [ ∀x.P(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  P(c')
7    Assume             [ ∀x.P(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  ∀x.P(x)
8    Dis_I2         [ ∀x.Q(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  P(c') ∨ Q(c')
9    Uni_E            [ ∀x.Q(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  Q(c')
10   Assume             [ ∀x.Q(x), ( ∀x.P(x)) ∨ ( ∀x.Q(x))]  ∀x.Q(x)
11   *
```

The line with the * in the proof is for the side condition that requires that the constant c' is new. By clicking on the proof the check is displayed in the OK syntax as follows:

```
news (Fun "c*" []) [(Uni (Dis (Pre "P" [Var 0]) (Pre "Q" [Var 0]))),
                    (Dis (Uni (Pre "P" [Var 0])) (Uni (Pre "Q" [Var 0])))]
```

The constant c' is written as "c*" here.

5 Formalization in Isabelle

Formalizations in Isabelle are written in a language that combines functional programming and logic. Our computer science bachelor students know programming from an introductory programming course and are introduced to logic in our course. This makes Isabelle a well suited way to present a sound proof system compared to a more abstract and mathematical approach. Furthermore, the language used in Isabelle is somewhat close to English, which also aids the intuitions of the students. Isabelle also allows the students to interactively inspect the different states of the proof and get an overview of the lemmas and theorems that are used in the steps – all in one screen. In this section we present the soundness proof using our formalization and show the concepts known from programming and logic.

5.1 An Overview

We first give an overview of the formalization in Isabelle. In the overview we see a number of datatypes *tm* and *fm*, that represent the objects that we want to reason about. We also see a primitive recursive function *member* which is used in the inductive definition *OK* of the proof system. Lastly, we see the *soundness* theorem of the proof system. We will explain these concepts as well as show and elaborate on the parts of the formalization that we did not put in the overview.

```
theory NaDeA imports Main begin

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
```

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

(* More primitive recursive functions as included in the previous sections *)

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" |

(* More rules as included in the previous sections *)

(* A proof of soundness' is included in the following sections *)

theorem soundness: "OK p [] ⟹ semantics e f g p"
  proof (simp add: soundness') qed

end
```

5.2 Terms and Formulas

Terms are defined by a datatype *tm*. Datatypes are a well-known concept from functional programming. A term is either a variable or a function application. Therefore, we have a constructor *Var* which constructs a variable from a *nat* representing its de Bruijn index. Likewise, we have a constructor *Fun* which constructs a function application from an *id* which is its function identifier and a *"tm list"* which represents its subterms.

When we introduce a datatype in Isabelle, we implicitly state that all terms can be constructed from its constructors. We also implicitly state that if two terms are equal then they must have been constructed from the same constructor and arguments. [18]

Formulas are also formalized as a datatype *fm*. It has a constructor for each operator and quantifier of our first-order logic.

5.3 Membership and Other Primitive Recursive Functions

List membership is defined as a primitive recursive function *member* over lists. The constructor for lists is # which separates the head of the list from the tail. The *member* function is primitive recursive because it removes a constructor from one of its arguments in every recursive call [2]. In Isabelle, primitive recursive functions are defined in much the same way as in functional programming, namely by stating cases for the different constructors.

The intuition of the function is that *member p z* returns true if the formula p is found in the list of formulas z and false otherwise. The function considers two cases: either the list is empty or it has a head and a tail. In the first case it is clear that the formula is not a member of the list. In the second case, we use the pattern *(q # z)* where q is the head of the list and z is the tail. If the head is equal to p it is true that p is a member of the list. Otherwise, we continue by looking in the tail of the list.

Other primitive recursive functions used in the theory are *semantics_term*, *semantics_list*, *semantics*, *new*, *news*, *inc_term*, *inc_list*, *sub_term*, *sub_list* and *sub*. These functions define the semantics, increasing the de Bruijn indices of a term, a constant being new to an expression, and substitution.

5.4 Proof System

Our proof system is defined by an inductive predicate. Each of the rules of the system is a case in the inductive predicate. For instance, consider the following rule:

$$\textit{Assume: "member p z} \Longrightarrow \textit{OK p z"}$$

The rule means that *OK p z* follows from *member p z*. Another case is the more complex rule:

$$\textit{Imp\_E: "OK (Imp p q) z} \Longrightarrow \textit{OK p z} \Longrightarrow \textit{OK q z"}$$

It states that *OK q z* follows from *OK (Imp p q) z* and *OK p z*. This corresponds to the usual notation for inference rules:

$$\frac{\mathsf{OK\ (Imp\ p\ q)\ z} \qquad \mathsf{OK\ p\ z}}{\mathsf{OK\ q\ z}}\ \mathsf{Imp\_E}$$

That a predicate is inductive means that it holds exactly when it can be derived using the given cases.

5.5 Proof of Soundness

We are now ready for the proof of soundness.

```
fun
  put :: "(nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a"
where
  "put e v x = (λn. if n < v then e n else if n = v then x else e (n − 1))"
```

The function *put* updates an environment by mapping variable *v* to value *x*. This is used in the definition of the quantifiers, but always for the outermost bound variable. Existing variables greater than *v* are pushed one position up, i.e. variable *i* now points to the value of variable *i − 1* in the old environment.

We use ***fun*** to declare many different functions without being restricted to the primitive recursive form. The operator λ is for lambda abstraction applied to occurrences of the parameter value and is known from functional programming. More informally, if *E* is some expression in Isabelle then *λx. E x* is the function that takes an input, for instance *y*, and returns *E y*.

```
lemma "put e 0 x = (λn. if n = 0 then x else e (n − 1))" proof simp qed
```

This lemma shows that *put* is a generalization of the expression

λn. ***if*** *n = 0* ***then*** *x* ***else*** *e (n − 1)*

which appears in the semantics. We use this generalization to prove properties of putting that we use in our soundness proof. The lemma is followed by a proof. In this case, the proof is performed automatically by the simplifier *simp*. The beginning of the proof is marked by ***proof*** and the end is marked by ***qed***. The proof method *simp* works by applying simplification rules [18]. It contains rules that are generated from definitions of functions, datatypes, etc., in addition to simplification rules from the Isabelle library.

```
lemma increment:
  "semantics_term (put e 0 x) f (inc_term t) = semantics_term e f t"
  "semantics_list (put e 0 x) f (inc_list l) = semantics_list e f l"
proof (induct t and l rule: semantics_term.induct semantics_list.induct)
qed simp_all
```

The lemma *increment* shows that we preserve the semantics of a term when we increment its de Bruijn indices while putting a value *x* at index 0. The reason is that putting pushes the values one index up in the environment. The proof is by induction on *t* and *l*, which is stated as

induct t ***and*** *l rule: semantics_term.induct semantics_list.induct* and it generates four proof goals; one for each of the cases in *semantics_term* and *semantics_list*. These goals can be inspected in the Isabelle editor by placing the cursor right after

```
(induct t and l rule: semantics_term.induct semantics_list.induct)
```

and looking in the so-called state panel. The proof method *simp_all* applies the simplifier to all available proof goals [18]. We place *simp_all* after ***qed*** in order to finish the proof and to allow inspection of the proof state interactively in Isabelle.

```
lemma commute: "put (put e v x) 0 y = put (put e 0 y) (v + 1) x"
proof force qed
```

The lemma *commute* shows that the function *put* commutes. More precisely, we want to put a value at position $v + 1$ in the environment and one at position 0, and the theorem shows that the order in which we do this does not matter, as long as we are careful with the indices.

The proof is automatic and uses the proof method *force*, which works by simplification and classical reasoning [2].

```
fun
  all :: "(fm ⇒ bool) ⇒ fm list ⇒ bool"
where
  "all b z = (∀p. if member p z then b p else True)"
```

The function *all* checks if the predicate *b* is true for all formulas in a list. The ∀ operator is for universal quantification.

```
lemma allhead: "all b (p # z) ⟹ b p" proof simp qed
```

```
lemma alltail: "all b (p # z) ⟹ all b z" proof simp qed
```

```
lemma allnew: "all (new c) z = news c z"
proof (induct z) qed (simp, simp, metis)
```

The lemma *allhead* states that if *b* holds for the entire list, then it holds for the head of the list in particular. The lemma *alltail* is similar, but for the tail of the list. Finally, the lemma *allnew* shows the equivalence between *news* and *all* combined with *new*. The proof uses the proof methods *simp* and *metis* in the order they are written, i.e. *simp* the first proof goal generated by the structural induction on *z*. Then *simp* simplifies the second proof goal which is afterwards proved by *metis*. The *metis* proof method is a resolution theorem prover [17].

```
lemma map':
  "new_term c t ⟹ semantics_term e (f(c := m)) t = semantics_term e f t"
  "new_list c l ⟹ semantics_list e (f(c := m)) l = semantics_list e f l"
proof (induct t and l rule: semantics_term.induct semantics_list.induct)
qed (simp, simp, metis, simp, simp, metis)
```

```
lemma map:
  "new c p ⟹ semantics e (f(c := m)) g p = semantics e f g p"
proof (induct p arbitrary: e)
qed (simp, simp, metis map'(2), simp, metis, simp, metis, simp, metis, simp_all)
```

```
lemma allmap:
  "news c z ⟹ all (semantics e (f(c := m)) g) z = all (semantics e f g) z"
proof (induct z) qed (simp, simp, metis map)
```

The lemma *map'* shows that we preserve the semantics of a term if we map a constant that is new to the term to another value. Here, *f(c := m)* maps function identifier *c* to *m* in the function map *f*. Because the lemma is quite obvious it can be proved automatically. The first and third goals are proved by *simp*, and the second and fourth are simplified by *simp* and then proved by *metis*. The lemma *map* shows that the property of *map'* can be extended to also hold for formulas. This can also be proved automatically. There are seven proof goals of the induction corresponding to each of the formula constructors. We use *simp* to discharge of the first proof goal, then *simp* followed by *metis* for the next four. This time we use *metis map'(2)* to prove the case for predicates. This works by applying *metis* with the addition of the second part of *map'* as a fact with which it can reason. The last two proof goals are proved with the simplifier using *simp_all*. The lemma *allmap* further extends the property of the lemma *map'* to also hold for lists of formulas. We prove it using *simp* and *metis map*.

```
lemma substitute':
  "semantics_term e f (sub_term v s t) =
     semantics_term (put e v (semantics_term e f s)) f t"
  "semantics_list e f (sub_list v s l) =
     semantics_list (put e v (semantics_term e f s)) f l"
proof (induct t and l rule: semantics_term.induct semantics_list.induct)
qed simp_all
```

The lemma *substitute'* is the famous substitution lemma for terms. This lemma shows a relation between the world of syntax and the world of semantics. More specifically, the relation is between the syntactical operation of substitution and the semantic notion of variable environments. The two are related because a substitution instantiates a variable with a term, and this term represents a value. Thus we get the same semantics of the term if we instead of substitution put the value directly at the index of the variable in the environment. The proof is by induction and *simp_all*.

```
lemma substitute:
  "semantics e f g (sub v t p) = semantics (put e v (semantics_term e f t)) f g p"
proof (induct p arbitrary: e v t)
  fix i l e v t
  show "semantics e f g (sub v t (Pre i l)) =
      semantics (put e v (semantics_term e f t)) f g (Pre i l)"
  proof (simp add: substitute'(2)) qed
next
  fix p e v t assume *: "semantics e' f g (sub v' t' p) =
      semantics (put e' v' (semantics_term e' f t')) f g p" for e' v' t'
  have "semantics e f g (sub v t (Exi p)) =
      (∃x. semantics (put (put e 0 x) (v + 1)
          (semantics_term (put e 0 x) f (inc_term t))) f g p)"
    using * proof simp qed
  also have "... =
      (∃x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Exi p)) =
      semantics (put e v (semantics_term e f t)) f g (Exi p)" proof simp qed
  have "semantics e f g (sub v t (Uni p)) =
      (∀x. semantics (put (put e 0 x) (v + 1)
          (semantics_term (put e 0 x) f (inc_term t))) f g p)"
    using * proof simp qed
  also have "... =
      (∀x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Uni p)) =
      semantics (put e v (semantics_term e f t)) f g (Uni p)" proof simp qed
qed simp_all
```

The lemma *substitute* extends the substitution lemma to hold also for formulas. The proof is by induction on a formula p. In the proof we write *arbitrary: e v t* because then e, v and t are also arbitrary in the induction hypothesis. This more general induction hypothesis is necessary for the proof. Most cases can be proven by the simplifier without any instructions, but we prove the cases for predicates *Pre*, existential quantification *Exi* and universal quantification *Uni* more explicitly. For the predicates, we only need instruct the simplifier to use *substitute'(2)* as a simplification rule by writing *(simp add: substitute'(2))*. For the existential quantification we make an explicit proof. We fix the subformula p of an existential quantification for which we want to prove the property. As said, we want to prove it with an arbitrary variable environment e, an arbitrary variable v, and an arbitrary term t so we fix those as well. We then state the induction hypothesis $*$ which says that for the subformula p of our existential quantification we can put the value of the term t in the environment instead of doing substitution with t:

> ***assume*** **: "semantics e' f g (sub v' t' p) =*
>   *semantics (put e' v' (semantics_term e' f t')) f g p"* ***for*** *e' v' t'*

The ***for*** keyword ensures that *e'*, *v'*, and *t'* are arbitrary as we wished. We wish to prove the substitution lemma for the existential quantification *Exi p*, i.e. that

> *semantics e f g (sub v t (Exi p)) =*
>   *semantics (put e v (semantics_term e f t)) f g (Exi p)*

The keyword ***also*** together with ***finally*** is used to make a proof from left to right of the equality of two expressions. This is what we want to do, and thus we start from the left-hand side:

semantics e f g (sub v t (Exi p))

and realize that by the definition of substitution and the semantics of *Exi* we just need a single value x for which the semantics of *sub (v + 1) (inc_term t) p* is true under the environment *put e 0 x*. At the same time, we realize that we can now use the induction hypothesis. Therefore, instead of considering the semantics of *sub (v + 1) (inc_term t) p* under *put e 0 x*, we equivalently consider the semantics of p under the variable environment which is *put e 0 x* with the value of t put on index $v + 1$. We must thus continue our proof from

> *(∃x. semantics (put (put e 0 x) (v + 1)*
>   *(semantics_term (put e 0 x) f (inc_term t))) f g p)*

We can make this expression much simpler by using *commute* and *increment(1).*

(∃x. semantics (put (put e v (semantics_term e f t)) 0 x) f g p)

We finish our proof using the semantics of *Exi*, as well as the fact that *put* generalizes putting at index 0, and we get the right-hand side we were looking for:

semantics (put e v (semantics_term e f t)) f g (Exi p)

Then follows a proof of substitution for the universal quantification *Uni* since it has the same induction hypothesis. The proof is very similar. Finally we write ***qed*** *simp_all* to prove the remaining cases by simplification.

```
lemma soundness': "OK p z ⟹ all (semantics e f g) z ⟹ semantics e f g p"
proof (induct arbitrary: f rule: OK.induct)
  fix f p z assume "all (semantics e f g) z"
      "all (semantics e f' g) (Imp p Falsity # z) ⟹
          semantics e f' g Falsity" for f'
  then show "semantics e f g p" proof force qed
next
  fix f p q z r assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Dis p q)"
      "all (semantics e f' g) (p # z) ⟹ semantics e f' g r"
      "all (semantics e f' g) (q # z) ⟹ semantics e f' g r" for f'
  then show "semantics e f g r" proof (simp, metis) qed
next
  fix f p q z assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Con p q)" for f'
  then show "semantics e f g p" "semantics e f g q"
  proof (simp, metis, simp, metis) qed
next
  fix f p z q c assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Exi p)"
      "all (semantics e f' g) (sub 0 (Fun c []) p # z) ⟹ semantics e f' g q"
      "news c (p # q # z)" for f'
  obtain x where "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
    using *(1) *(2) proof force qed
  then have "semantics (put e 0 x) f g p" proof simp qed
  then have "semantics (put e 0 x) (f(c := λw. x)) g p"
    using *(4) allhead allnew map proof blast qed
  then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
    proof (simp add: substitute) qed
  moreover have "all (semantics e (f(c := λw. x)) g) z"
    using *(1) *(4) alltail allnew allmap proof blast qed
  ultimately have "semantics e (f(c := λw. x)) g q" using *(3) proof simp qed
  then show "semantics e f g q" using *(4) allhead alltail allnew map
  proof blast qed
next
```

```
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 t p)" for f'
  then have "semantics (put e 0 (semantics_term e f t)) f g p"
  proof (simp add: substitute) qed
  then show "semantics e f g (Exi p)" proof (simp, metis) qed
next
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Uni p)" for f'
  then show "semantics e f g (sub 0 t p)" proof (simp add: substitute) qed
next
  fix f c p z assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 (Fun c []) p)"
      "news c (p # z)" for f'
  have "semantics (λn. if n = 0 then x else e (n − 1)) f g p" for x
  proof −
    have "all (semantics e (f(c := λw. x)) g) z"
      using *(1) *(3) alltail allnew allmap proof blast qed
    then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
      using *(2) proof simp qed
    then have "semantics (λn. if n = 0 then x else e (n − 1))
        (f(c := λw. x)) g p"
      proof (simp add: substitute) qed
    then show "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
      using *(3) allhead alltail allnew map proof blast qed
  qed
  then show "semantics e f g (Uni p)" proof simp qed
qed simp_all
```

The lemma *soundness'* shows the soundness of the proof system. It is done by rule induction on the rules of the proof system. We have to prove that assuming that the derivations in the premises follow logically, then so does the derivation in the conclusion. For the rules *Boole*, *Dis_E*, *Con_E1*, *Con_E2* and *Uni_E* we state the induction hypothesis, and the assumption that the premises are satisfied. We then do the proof by automation. For *Uni_I*, *Exi_E* and *Exi_I* we write out the proofs explicitly because they are more complicated. We prove the remaining rules sound by automation with the substitution lemma as simplification rule. The keyword ***next*** is used to separate the different cases.

Let us look at how we proved *Uni_I* sound. The $*$ states our induction hypothesis which states that if our assumptions *z* are satisfied by any function map then so is

p with a constant *Fun c []* substituted for 0.

$$\textit{all (semantics e f' g) z} \Longrightarrow \textit{semantics e f' g (sub 0 (Fun c []) p)}$$

We additionally assume that the side condition that c is new to $p\#z$.

$$\textit{news c (p \# z)}$$

Since we want to prove the derivation from z to *Uni p* sound we also assume that the premises z are satisfied by a fixed f and a fixed g.

$$\textit{all (semantics e f g) z}$$

We then wish to prove that so is *Uni p*. Since the premises are satisfied by f and since c is new to them they must also be satisfied by $f(c := \lambda w.\ x)$.

$$\textit{all (semantics e (f(c := } \lambda w.\ x\textit{)) g) z}$$

In this step we used the proof method *blast* which is a tableau prover [17]. Then it follows by our induction hypothesis that also p with c substituted for 0 is satisfied.

$$\textit{semantics e (f(c := } \lambda w.\ x\textit{)) g (sub 0 (Fun c []) p)}$$

We then use the substitution lemma to add the value of t to the environment instead of doing the substitution.

$$\textit{semantics } (\lambda n.\ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ e\ (n - 1))\ (f(c := \lambda w.\ x))\ g\ p$$

Since c is new to p we might as well evaluate it in f instead of $f(c := \lambda w.\ x)$ and this concludes the proof.

$$\textit{semantics } (\lambda n.\ \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ x\ \textbf{\textit{else}}\ e\ (n - 1))\ f\ g\ p$$

5.6 A Consistency Corollary to the Soundness Theorem

Soundness is the main theorem about the formalization of the natural deduction proof system. As a corollary we immediately prove the following consistency result about the proof system:

Something, but not everything, can be proved.

In Isabelle we can prove it using the simplifier (*simp*), some simple rules and Isabelle's prover for intuitionistic logic (*iprover*), although a classical prover (say, *metis*) would work too, of course:

```
corollary "∃p. OK p []" "∃p. ¬ OK p []"
proof –
  have "OK (Imp p p) []" for p proof (rule Imp_I, rule Assume, simp) qed
  then show "∃p. OK p []" proof iprover qed
  have "¬ semantics (e :: nat ⇒ unit) f g Falsity" for e f g proof simp qed
  then show "∃p. ¬ OK p []" using soundness proof iprover qed
qed
```

Recall that ∃ is the existential quantifier in Isabelle. The symbol ¬ is negation in Isabelle. The first part (*∃p. OK p []* ***for*** *p*) follows from a simple proof of $p \to p$ (for an arbitrary formula p in first-order logic). The second part (*∃p. ¬ OK p []*) follows from the proof of soundness and from the fact that the semantics of *Falsity* is always false (for simplicity we consider universes with just one element, provided by the *unit* type).

5.7 Style of the Proof

When you do a proof in Isabelle, you need to choose how close you want the steps of the proof to be to each other. On one hand the proof should be understandable, but on the other hand you do not want the readers to get lost in small details. Larger steps also allow the reader to think for himself instead of having everything spelled out in detail. If a student wants to gain more insight, she can expand it, and let Isabelle check if the details she added were correct. Isabelle also has tools that allow its users to see which steps *simp* used to prove a result.

The notation we chose to use is close to that of programming rather than that of mathematics and set theory. Isabelle, however, also supports a more classical notation. Our motivation for the choice is our students' background from programming, as well as to show that a very well-defined structure lies beneath the logical symbols both at the object and the meta levels.

We use the formal semantics and soundness proof in our teaching. Among other things the students can make calculation using the formal semantics in Isabelle and also make changes to the formal semantics (for example, replacing the if-then-else with logical operators in Isabelle, or adding negation to the logic).

6 Related Work

Formalizations of model theory and proof theory of first-order logic are rare, for example [6, 7, 11, 20, 21].

Throughout the development of NaDeA we have considered some of the natural deduction assistants currently available. Several of the tools available share some common flaws. They can be hard to get started with, or depend on a specific platform. However, there are also many tools that each bring something useful and unique to the table. One of the most prominent is Panda, described in [13]. Panda includes a lot of graphical features that make it fast for the experienced user to conduct proofs, and it helps the beginners to tread safely. Another characteristic of Panda is the possibility to edit proofs partially before combining them into a whole. It definitely serves well to reduce the confusion and complexity involved in conducting large proofs. However, we still believe that the way of presenting the proof can be more explicit. In NaDeA, every detail is clearly stated as part of the proof code. In that sense, the students should become more aware of the side conditions to rules and how they work.

Another tool that deserves mention is ProofWeb [10] which is open source software for teaching natural deduction. It provides interaction between some proof assistants (Coq, Isabelle, Lego) and a web interface. The tool is highly advanced in its features and uses its own syntax. Also, it gives the user the possibility to display the proof in different formats. However, the advanced features come at the cost of being very complex for bachelor students and require that you learn a new syntax. It serves as a great tool for anyone familiar with natural deduction that wants to conduct complex proofs that can be verified by the system. It may, on the other hand, prove less useful for teaching natural deduction to beginners since there is no easy way to get started. In NaDeA, you are free to apply any (applicable) rule to a given formula, and thus, beginners have the freedom to play around with the proof system in a safe way. Furthermore, the formalized soundness result for the proof system of NaDeA makes it relevant for a broader audience, since this gives confidence in that the formulas proved with the tool are actually valid.

7 Further Work

In NaDeA there is support for proofs in propositional logic as well as first-order logic. We would also like to extend to more complex logic languages, the most natural step being higher-order logic. This could be achieved using the CakeML approach [8].

Other branches of logic would also be interesting. Apart from just extending the natural deduction proof system to support other branches of logic, another option is to implement other proof systems as well.

Because the NaDeA tool has a formalization in Isabelle of its proof system, we would like to provide features that allow for a more direct integration with Isabelle.

For instance, we would like to allow for proofs to be exported to a format suitable for Isabelle such that Isabelle could verify the correctness of the proofs. A formal verification of the implementation would require much effort, but perhaps it could be reimplemented on top of Isabelle (although probably not in TypeScript / JavaScript) or using Isabelle's code generation facility.

We would like to extend NaDeA with more features in order to help the user in conducting proofs and in understanding logic. For example, the tool could be extended with step-by-step execution of the auxiliary primitive recursive functions used in the side conditions of the natural deduction rules.

NaDeA has been successfully classroom tested in a regular course with around 70 bachelor students in computer science each year. The students find the formal semantics and the proof of the soundness theorem relevant and instructive. We have extended NaDeA with a so-called ProofJudge system [19] which allows students to submit solutions and get feedback. We are in the process of adding to NaDeA a simple automated theorem prover [20, 21], verified by the Isabelle proof assistant and developed using Isabelle's code generation facility, in order to make it possible to better guide the students if for example sub-proofs are started and there is in fact no possible proof.

References

[1] Mordechai Ben-Ari. Mathematical Logic for Computer Science. Third Edition. Springer 2012.

[2] Tobias Nipkow, Lawrence C. Paulson and Markus Wenzel. Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science 2283, Springer 2002.

[3] Dag Prawitz. Natural Deduction. A Proof-Theoretic Study. Stockholm: Almqvist & Wiksell 1965.

[4] Francis Jeffry Pelletier. A Brief History of Natural Deduction. History and Philosophy of Logic, 1-31, 1999.

[5] Melvin Fitting. First-Order Logic and Automated Theorem Proving. Second Edition Springer 1996.

[6] John Harrison. Formalizing Basic First Order Model Theory. Lecture Notes in Computer Science 1497, 153–170, Springer 1998.

[7] Stefan Berghofer. First-Order Logic According to Fitting. Formal Proof Development. Archive of Formal Proofs 2007.

[8] Ramana Kumar, Rob Arthan, Magnus O. Myreen and Scott Owens. HOL with Definitions: Semantics, Soundness, and a Verified Implementation. Lecture Notes in Computer Science 8558, 308–324, Springer 2014.

[9] Jørgen Villadsen, Anders Schlichtkrull and Andreas Viktor Hess. Meta-Logical Reasoning in Higher-Order Logic. Accepted at 29th International Symposium Logica, Hejnice Monastery, Czech Republic, 15-19 June 2015.

[10] ProofWeb. Online `http://proofweb.cs.ru.nl/login.php` (ProofWeb is both a system for teaching logic and for using proof assistants through the web). Accessed September 2016.

[11] Jasmin Christian Blanchette, Andrei Popescu and Dmitriy Traytel. Unified Classical Logic Completeness - A Coinductive Pearl. Lecture Notes in Computer Science 8562, 46–60, 2014.

[12] Krysia Broda, Jiefei Ma, Gabrielle Sinnadurai and Alexander Summers. Pandora: A Reasoning Toolbox Using Natural Deduction Style. Logic Journal of IGPL, 15(4):293–304, 2007.

[13] Olivier Gasquet, François Schwarzentruber and Martin Strecker. Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students. Lecture Notes in Computer Science 6680, 85–92. Springer 2011.

[14] Antonia Huertas. Ten Years of Computer-Based Tutors for Teaching Logic 2000-2010: Lessons learned. Lecture Notes in Computer Science 6680, 131–140, Springer 2011.

[15] Michael Huth and Mark Ryan. Logic in Computer Science: Modelling and Reasoning About Systems. Second Edition. Cambridge University Press 2004.

[16] Jonathan P. Seldin. Normalization and Excluded Middle. I. Studia Logica, 48(2):193–217, 1989.

[17] Jasmin Christian Blanchette, Lukas Bulwahn and Tobias Nipkow. Automatic Proof and Disproof in Isabelle/HOL. Lecture Notes in Computer Science 6989, 12-27, 2011.

[18] Tobias Nipkow and Gerwin Klein. Concrete Semantics with Isabelle/HOL. Springer 2014.

[19] Jørgen Villadsen. ProofJudge: Automated Proof Judging Tool for Learning Mathematical Logic. Exploring Teaching for Active Learning in Engineering Education Conference (ETALEE), Copenhagen, Denmark, 2015.

[20] Jørgen Villadsen, Anders Schlichtkrull and Andreas Halkjær. Code Generation for a Simple First-Order Prover. Isabelle Workshop, Nancy, France, 2016.

[21] Alexander Birch Jensen, Anders Schlichtkrull and Jørgen Villadsen. Verification of an LCF-Style First-Order Prover with Equality. Isabelle Workshop, Nancy, France, 2016.

Received 11 October 2016