Natural Deduction and the Isabelle Proof Assistant

Jørgen Villadsen

`jovi@dtu.dk`

Andreas Halkjær From　　　　　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

We describe our Natural Deduction Assistant (NaDeA) and the interfaces between the Isabelle proof assistant and NaDeA. In particular, we explain how NaDeA, using a generated prover that has been verified in Isabelle, provides feedback to the student, and also how NaDeA, for each formula proved by the student, provides a generated theorem that can be verified in Isabelle.

1 Introduction

A textbook on logic in computer science like Huth & Ryan [12] focuses on natural deduction.

> Natural deduction is a common name for the class of proof systems composed of simple and self-evident inference rules based upon methods of proof and traditional ways of reasoning that have been applied since antiquity in deductive practice.

– Andrzej Indrzejczak, Internet Encyclopedia of Philosophy, `http://www.iep.utm.edu/nat-ded/`

We have developed a website for teaching first-order logic and natural deduction to computer science bachelor students:

<div align="center">

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

</div>

Our motivation is that our students should obtain a logical background that give them the prerequisites to build and formally verify dependent software. Building and verifying such software systems can be done in proof assistant computer programs like Isabelle [16] in which natural deduction reasoning is a central concept. Our website therefore forms part of a course which, among other subjects, teaches natural deduction. The website consists of a web application, NaDeA, which implements a natural deduction proof system. The website also contains explanations of the system. Lastly, the website contains the ProofJudge component which is used for assessing exercises done by students in NaDeA and providing them feedback. The syntax, semantics and a sound and complete proof system has been formalized in the proof assistant Isabelle such that all definitions are very precise. The present paper describes the developments since our previous paper on the system [19], in particular the interfaces to the Isabelle proof assistant. We therefore start out by giving some background on the Isabelle proof assistant.

1.1 The Isabelle Proof Assistant

Proof assistants are computer programs that assist computer scientists, mathematicians and logicians in defining concepts and proving properties about them. Furthermore, they ensure correctness of the proofs that are constructed.

Isabelle, or more precisely, Isabelle/HOL, is one such proof assistant. It implements higher-order logic (HOL) and we consider first-order logic (FOL) as a subset of HOL. For instance, a computer scientist can define a sorting algorithm in Isabelle/HOL since Isabelle/HOL contains functionality for defining functions in a similar style to a functional programming language like Haskell or Standard ML. Likewise, Isabelle contains functionality for defining datatypes and more. The computer scientists can use Isabelle/HOL to prove their algorithms correct. In Isabelle the preferred format of proofs uses the proof language Isar [20] which looks like a mixture of English, logic and a programming language, and is based on natural deduction. Each step of the proof is specified by the user and steps are chained together using proof methods, which implement rewriting or automatic theorem proving. In each step the user specifies which proof method to apply and which lemmas the method should use. This is done either by hand or using the Sledgehammer tool which filters out relevant lemmas from the current context and employs external top-of-the-line provers such as SPASS, E, Vampire, CVC4 and Z3 to find a proof that is then automatically reconstructed in Isabelle using an appropriate proof method.

Functions written in the functional programming style can be exported from Isabelle to the languages Standard ML, Haskell, OCaml and Scala. Thus the computer scientist can obtain a verified sorting program. We use this feature too.

User interaction with Isabelle/HOL can be done in the Prover Integrated Development Environments (PIDEs) Isabelle/jEdit and Visual Studio Code. This gives access to various kinds of feedback from the system such as information about the execution of proof methods, type inference, search for lemmas, lookup of definitions and much more. Isabelle contains a large library of definitions, functions, lemmas and theorems on which new developments can be built. Even more can be found in the Archive of Formal Proofs which is a collection of formal developments in Isabelle that anyone can submit to.

Another popular proof assistant is Coq, which also uses natural deduction.

1.2   Overview of the Paper

In Section 2 we explain NaDeA and the natural deduction system that it implements. In Section 3 we explain how we formalize the syntax and semantics of first-order logic in Isabelle. In Section 4 we explain the formalization of the natural deduction system in Isabelle. In Section 5 we present our new developments, which form interfaces between Isabelle and NaDeA. In particular, we explain how NaDeA, using verified code generated by Isabelle, can check if the students are on the right track in their proofs, and we explain how NaDeA can now export its proof to a format that can be checked by Isabelle. In Section 6 we discuss related work and in Section 7 we conclude.

2   The Natural Deduction Assistant (NaDeA)

The syntax of first-order logic is based on $\bot$ for falsehood, $\rightarrow$ for implication, $\vee$ for disjunction and $\wedge$ for conjunction. We use the following abbreviations:

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

These abbreviations are outside NaDeA in order to simplify the proof system. Also equality ($=$) is omitted from NaDeA. We use $\exists$ and $\forall$ for existential and universal quantification, respectively.

2.1 Natural Deduction — A Textbook Presentation

We now present the natural deduction rules as described in the literature, following Huth & Ryan [12]. 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 $\perp$-elimination rule (also known as the rule of explosion) [18]. The rules are as follows:

$$
\begin{array}{c}
\boxed{\begin{array}{c}\neg\phi \\ \vdots \\ \perp\end{array}} \\[2pt]
\hline
\phi
\end{array}\; PBC
\qquad
\frac{\phi \qquad \phi \to \psi}{\psi}\; \to E
\qquad
\begin{array}{c}
\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}} \\[2pt]
\hline
\phi \to \psi
\end{array}\; \to I
$$

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

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

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

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

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 [12, 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.

2.2   A Sample Natural Deduction Proof

Proofs in NaDeA consist of a number of lines and each line consists of the number of the line, the rule that was applied in this line and a list of assumptions (marked in square brackets) followed by a formula. If a line contains a ¤ symbol instead of the name of a rule, the student can click on it to specify which rule to use. Proofs are done using backward chaining. That is, the formula in each line is a goal which the student wants to prove. The student can prove it by clicking on ¤ and choosing a rule that has the goal as conclusion. This introduces a number of new lines, i.e. new subgoals – namely the premises and side-conditions of the rule we chose. This forms a tree which we represent by indenting subgoals appropriately. When there are no ¤ symbols left the proof is finished. The reason that proofs can be finished is that the rule *Assume* has zero premises, so this is the rule that is used in all leaves of a finished proof.

Let us look at a small example where we prove $(\forall x.R(x,x)) \rightarrow (\forall x.\exists y.R(x,y))$. First we need to choose a rule.

<div align="center">

1   ¤   [ ] $(\forall x.R(x, x)) \rightarrow (\forall x.\exists y.R(x, y))$

Boole	X
Imp_E	
Imp_I	
Dis_E	
Con_E1	
Con_E2	
Exi_E	

</div>

Let us say that we choose *Imp_I* corresponding to Huth & Ryan's $\rightarrow I$. This introduces a subgoal where, from the assumption $\forall x.R(x,x)$, we have to prove $\forall x.\exists y.R(x,y)$. Now we need to choose the rule to prove this subgoal with.

<div align="center">

1   Imp_I   [ ] $(\forall x.R(x, x)) \rightarrow (\forall x.\exists y.R(x, y))$
2   ¤ $[\forall x.R(x, x)]$ $\forall x.\exists y.R(x, y)$

Boole	X
Imp_E	
Dis_E	
Con_E1	
Con_E2	
Exi_E	
Uni_I	

</div>

We choose *Uni_I* corresponding to Huth & Ryan's $\forall I$. This introduces two subgoals – namely line 3 and line 4. Line 3 corresponds to the premise of the *Uni_I* rule – namely that we must prove from assumption $\forall x.R(x,x)$ that $\exists x.R(c',x)$. Line 4 corresponds to the side-condition which states that c' is new in the proof. The student can expand this line to see exactly what the side condition states. NaDeA automatically chooses c' such that it is new.

We now need to choose which rule to use to prove line 3. Since the outermost quantifier is existential, we choose the existential introduction rule *Exi_I*. Now we need to pick the witness and NaDeA presents us with a box for specifying a term. We choose the constant c' which then appears instead of the quantified variable in the new subgoal.

```
1        Imp_I    [] (∀x.R(x, x))→(∀x.∃y.R(x, y))
2        Uni_I      [∀x.R(x, x)]  ∀x.∃y.R(x, y)
3          ¤            [∀x.R(x, x)]  ∃x.R(c', x)
4          *            (c')
```

Next, we choose *Uni_E* corresponding to Huth & Ryan's ∀*E*. This leaves the subgoal where we from ∀*x.R*(*x,x*) need to prove ∀*x.R*(*x,x*). NaDeA immediately recognizes that this can be done using the *Assume* rule since the formula we want to prove appears exactly as one of the assumptions we have. This finishes the proof since there are no ¤-symbols left.

```
1        Imp_I    [] (∀x.R(x, x))→(∀x.∃y.R(x, y))
2        Uni_I      [∀x.R(x, x)]  ∀x.∃y.R(x, y)
3        Exi_I       [∀x.R(x, x)]  ∃x.R(c', x)
4        Uni_E        [∀x.R(x, x)]  R(c', c')
5        Assume          [∀x.R(x, x)]  ∀x.R(x, x)
6          *            (c')
```

The formula can be proved in Isabelle in a similar way but can also be proved automatically.

3   Syntax and Semantics of First-Order Logic in Isabelle

There are two main approaches to formalizing logics, namely shallow embeddings and deep embeddings.

Shallow Embedding
The act of representing one logic or language with another by providing a syntactic translation

– Wiktionary, `https://en.wiktionary.org/wiki/shallow_embedding`

This approach is trivial since first-order logic is a subset of Isabelle/HOL.

Deep Embedding
The act of representing one language, typically a logic or programming language, with another by modeling expressions in the former as data in the latter

– Wiktionary, `https://en.wiktionary.org/wiki/deep_embedding`

This approach is more involved, since we need to decide how to formalize first-order logic as data in Isabelle/HOL. The advantage, however, is that since formulas are represented as data, we can define a semantics on them and express and prove, in Isabelle, meta-theorems such as soundness and completeness of natural deduction. We therefore take this approach. Our formalization builds on a formalization by Berghofer [2] according to Fitting's [8] approach to first-order logic.

3.1 Syntax

The syntax of first-order logic can be formalized using a type *char list* for predicate symbols and function symbols together with datatypes for terms *tm* and formulas *fm*.

```
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
```

Notice that variables consist of natural numbers. This is because the formalization uses de Bruijn indices to represent variables. The idea is that an occurrence of *Var i* exists in the scopes of a set of quantifiers. The occurrence is then bound to the quantifier with the *i* th closest scope. Here are some examples:

$$(\forall x.\forall y.A(x,y)) \longrightarrow (\forall x.A(x,x))$$

Imp (Uni (Uni (Pre A [Var 1, Var 0]))) (Uni (Pre A [Var 0, Var 0]))

$$\forall x.\forall y.(\forall u.\forall z.A(z,u)) \longrightarrow A(x,y)$$

Uni (Uni (Imp (Uni (Uni (Pre A [Var 1, Var 0]))) (Pre A [Var 1, Var 0])))

3.2 Semantics

The semantics can be formalized as recursive functions on the syntax of first-order terms and formulas (*semantics_term* and *semantics*). Consider the first arguments to *semantics*. It represents a variable denotation and has type $nat \Rightarrow {}'a$, i.e. it takes a natural number (the variable) and returns an element of the universe. We represent the universe by a type variable ${}'a$.

```
primrec
  semantics_term :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a› and
  semantics_list :: ‹(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list› where
  ‹semantics_term e f (Var n) = e n› |
  ‹semantics_term e f (Fun i l) = f i (semantics_list e f l)› |
  ‹semantics_list e f [] = []› |
  ‹semantics_list e f (t # l) = semantics_term e f t # semantics_list e f l›

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)›
```

4   Natural Deduction Proof System in Isabelle

4.1   Membership

Using the Isabelle command primrec we define a primitive recursive function *member* that checks whether a formula (type fm) is a member of a list of formulas (type fm list).

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

This function is used in the Assume rule only.

4.2   Newness

The newness of a constant with respect to a term can be formalized as two mutually recursive functions which recurse on terms and lists of terms. The functions simply check if the variable appears anywhere in the term.

Hereafter the newness of a constant with respect to a formula can be defined by recursion over formulas.

```
primrec
  new_term :: ‹id ⇒ tm ⇒ bool› and
  new_list :: ‹id ⇒ tm list ⇒ bool› where
  ‹new_term c (Var n) = True› |
  ‹new_term c (Fun i l) = (if i = c then False else new_list c l)› |
  ‹new_list c [] = True› |
  ‹new_list c (t # l) = (if new_term c t then new_list c l else False)›

primrec new :: ‹id ⇒ fm ⇒ bool› where
  ‹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›

primrec news :: ‹id ⇒ fm list ⇒ bool› where
  ‹news c [] = True› |
  ‹news c (p # z) = (if new c p then news c z else False)›
```

The function *news* checks whether a constant is new with respect to a list of formulas.

4.3   Substitution

Substitution is similarly defined by recursion over terms, lists of terms and formulas. In the proof system, whenever we do a substitution, we also remove a quantifier. The substitution function should take this into account. This is why the case for variables is rather involved – it needs to make sure that the variables point to the appropriate quantifiers. For the same reason the recursion on existential and universal quantifiers add one to the variable and apply an auxiliary function *inc_term* which also adds one to all variables in the term.

```
primrec
  inc_term :: ‹tm ⇒ tm› and
  inc_list :: ‹tm list ⇒ tm list› where
  ‹inc_term (Var n) = Var (n + 1)› |
  ‹inc_term (Fun i l) = Fun i (inc_list l)› |
  ‹inc_list [] = []› |
  ‹inc_list (t # l) = inc_term t # inc_list l›

primrec
  sub_term :: ‹nat ⇒ tm ⇒ tm ⇒ tm› and
  sub_list :: ‹nat ⇒ tm ⇒ tm list ⇒ tm list› where
  ‹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 v s [] = []› |
  ‹sub_list v s (t # l) = sub_term v s t # sub_list v s l›

primrec sub :: ‹nat ⇒ tm ⇒ fm ⇒ fm› where
  ‹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)›
```

4.4   The Proof System as an Inductive Predicate

With all these concepts in place we can define the proof system as an inductive predicate. Inductive predicates in Isabelle are specified by a number of implications that define for which elements the predicate holds. Furthermore, the predicate only holds for these elements and no others.

Each implication in our predicate corresponds to one of the rules by Huth & Ryan. However, compared to those rules we keep track of assumptions by explicitly representing them in lists.

```
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›
```

4.5   Soundness and Completeness

We define the validity of a formula as the formula evaluating to *True* in all variable denotations (e), function denotations (f) and predicate denotations (g) with the natural numbers as universe. This is different from the usual notion of validity which considers all universes – not only that of the natural
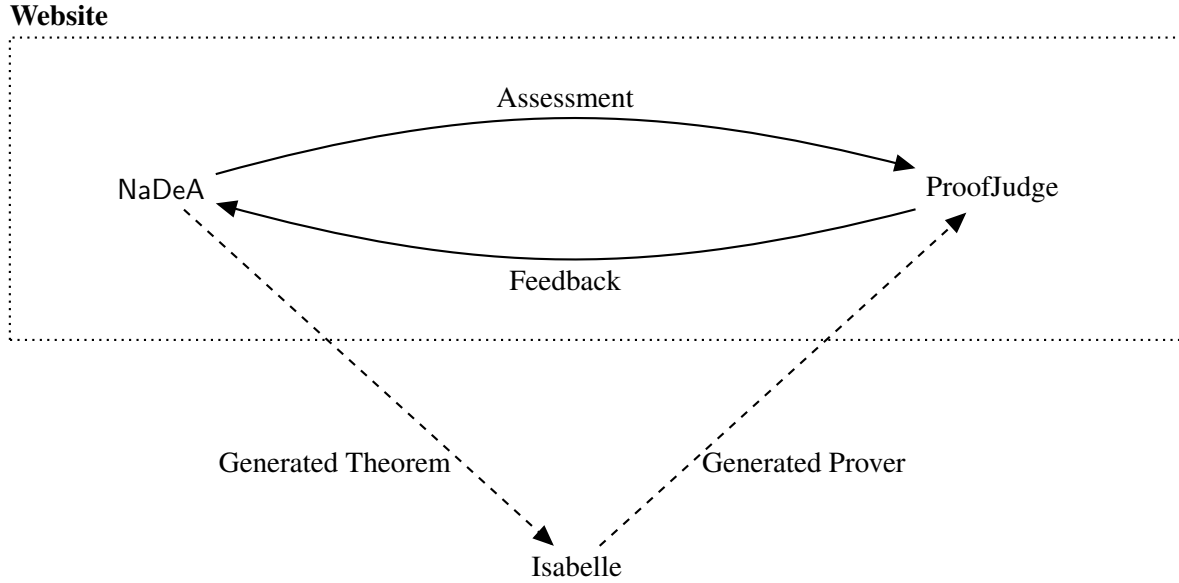
Website

Figure 1: NaDeA, ProofJudge and Isabelle

numbers. We can, however, prove in Isabelle that our notion of validity implies the truth of a formula in any variable denotation, function denotation and predicate denotation – and thus the formula must indeed be valid with respect to the usual notion of validity.

We prove that the valid formulas are exactly the same as those which can be proved using *OK*, hence the natural deduction proof system is sound and complete.

```
abbreviation ‹valid p ≡ ∀(e :: nat ⇒ nat) f g. semantics e f g p›

proposition ‹valid p ⟹ semantics e f g p›
  using soundness completeness by blast

proposition ‹OK p [] = valid p›
  using soundness completeness by blast
```

OK is preferred to longer words like *Provable* or symbols like ⊢ mainly because it is easier to pronounce in class. The complete Isabelle theory file with proofs of *soundness* and *completeness* is checked in a few seconds. The more than 4000 lines are available on the NaDeA website.

5   Verification

We now describe our two new additions to our website which connect it to Isabelle. This relationship is depicted in Figure 1. Our website consists of two subcomponents: NaDeA and ProofJudge. NaDeA is the application in which students do their natural deduction proofs, and ProofJudge is a system that gives teachers and teaching assistants the possibility to assess the proofs done by students. The ProofJudge system also has features for providing automatic feedback.

In this section we will explain first the *generated prover* relationship between ProofJudge and Isabelle and thereafter the *generated theorem* relationship.

5.1 Generated Prover

ProofJudge can provide automatic live feedback to students by indicating whether they are on the right track in their proof attempt. It works by running a first-order prover on each subgoal to see if it can prove the subgoal. If it cannot, then this is a warning sign to the student that they may be going down a wrong path. This warning sign is indicated to the student by coloring the number of the subgoal in orange.

The prover we run is a tableau prover which is based on a logical kernel whose soundness we verified in Isabelle [13]. The kernel reimplements the kernel in Harrison's textbook [11] and the tableau prover is a port from OCaml to SML of the tableau prover in his book. More precisely, we define an axiomatic proof system in Isabelle/HOL and prove it sound. Hereafter Isabelle/HOL generates an SML program which forms a logical kernel. The tableau prover can be considered sound since all its logical operations are required to go through the kernel, and thus soundness is inherited from the kernel. Harrison, informally, argues that the tableau prover is also complete. In order to get the prover to run in the browser, we use a compiler to translate it from SML to JavaScript [7].

5.2 Generated Theorem

Formal verification of JavaScript applications is still at an infant stage – see e.g. the work by Park, Stefănescu and Roşu [17] for a promising first step. Therefore we do not attempt to verify NaDeA directly. Hence it is possible, though unlikely, that some proof rule in some context is applicable in NaDeA when, according to the formal proof system, it should not be. In turn this makes it possible that an invalid formula could be proved in NaDeA.

The formalization of NaDeA in Isabelle does not suffer from this problem. Every application of a rule is checked by Isabelle, ensuring that every side condition etc. is satisfied or subsequently discharged, assuming we trust Isabelle. Furthermore this formalization has been proved sound, meaning that if a formula can be derived then it is valid. We want to take advantage of this formalization to allow students to check that a formula they have proved in NaDeA is actually valid according to the semantics, as explained above.

5.2.1 Soundness

When a proof has been completed in NaDeA, the count of subgoals left turns into a smiley instead of a zero. Clicking this smiley reveals a window with several tabs and an explanation of each of them. The most basic tab is labeled *Verify natural deduction proof in Isabelle* and is automatically filled with two things: An Isabelle **proposition** with the proved formula in Isabelle syntax (shallow embedding) and a **theorem** stating the validity of the formula using the NaDeA syntax (deep embedding).

Consider the formula:

$$(\forall x. \forall y. A(x,y)) \longrightarrow (\forall x. A(x,x))$$

The generated **proposition** for this formula is:

proposition $(!\,x.\,(!\,y.\,A(x,y))) \,--> (!\,x.\,A(x,x))$
 by *blast*

We use ASCII characters for the connectives instead of Isabelle's full syntax: An exclamation mark for the universal quantifier and a three-piece arrow for the implication. This "typewriter" philosophy is to make it look more like programming and thus more familiar to the students.

Of more interest is the translation of the proof itself to the Isabelle-encoded proof rules. The *soundness* theorem is used to prove the validity of the formula given its (translated) derivation:

> **theorem** *semantics e f g* (*Imp* (*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))) (*Uni* (*Pre ''A'' [Var 0, Var 0]*))))
> **proof** (*rule soundness*)
> **show** *OK* (*Imp* (*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))) (*Uni* (*Pre ''A'' [Var 0, Var 0]*)))) []
> **proof** (*rule Imp-I*)
>   **show** *OK* (*Uni* (*Pre ''A'' [Var 0, Var 0]*))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **proof** (*rule Uni-I*)
>   **have** *OK* (*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** (*rule Assume*) *simp*
>   **then have** *OK* (*sub 0* (*Fun ''c∗'' []*) (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** (*rule Uni-E*)
>   **then have** *OK* (*Uni* (*Pre ''A'' [Fun ''c∗'' [], Var 0]*))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** *simp*
>   **then have** *OK* (*sub 0* (*Fun ''c∗'' []*) (*Pre ''A'' [Fun ''c∗'' [], Var 0]*))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** (*rule Uni-E*)
>   **then have** *OK* (*Pre ''A'' [Fun ''c∗'' [], Fun ''c∗'' []]*) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** *simp*
>   **then show** *OK* (*sub 0* (*Fun ''c∗'' []*) (*Pre ''A'' [Var 0, Var 0]*))) [*Uni* (*Uni* (*Pre ''A'' [Var 1, Var 0]*)))]
>   **by** *simp*
>   **qed** *simp*
> **qed**
> **qed**

The proof strongly resembles the one made in NaDeA visually and structurally with a few complications explained shortly. Importantly the application of rules is in the same order as on the website, allowing the student to recognize their original proof in its reformulation.

This generated proof can be copied into the end of NaDeA.thy for Isabelle to verify, NaDeA.thy being the Isabelle formalization of NaDeA. The student can inspect the proof itself to be convinced that it verifies the correct thing, which can also serve as an introduction to the Isabelle syntax. We discuss some of the complications we faced when writing the program that generates the translated derivations.

Complications   The propositional part of the proof, where the rules have no side conditions, can be translated directly to applications of the corresponding rules encoded in Isabelle. For the four quantifier rules however, the substitutions and side conditions add some extra complexity.

The complications with substitutions can be seen in the proof above, in the lines proved with **by** *simp*. These lines rewrite the state of the proof slightly and are necessary when a proof rule either assumes or produces a formula with a substitution. Then for the (next) proof rule to apply, we need to ask Isabelle explicitly to (un)do the substitution using the simplifier. This adds a bit of noise to the proof but is a very simple transformation, that can be easily understood.

The second complication is, that while the website automatically checks and discharges the side conditions of proof rules, i.e. newness, this must be done explicitly when encoding them in Isabelle.

Luckily these side conditions are simple and can be discharged unobtrusively with the **qed** *simp*-syntax that runs the simplifier on the remaining goals.

5.2.2   Completeness

The NaDeA proof rules as formalized in Isabelle have been proved complete for both closed [9] and open formulas. Nevertheless students might be interested, having derived a closed formula, in verifying the validity of its corresponding "opened" versions, for instance to better understand the role of environments in the semantics. This functionality is available in the tab *Open formulas - Scratch.thy* whose content is explained below.

Consider the formula:

$$\forall x.\forall y.(\forall u.\forall z.A(z,u)) \longrightarrow A(x,y)$$

This formula is valid and thus derivable in NaDeA. It is the universal closure of:

$$(\forall u.\forall z.A(z,u)) \longrightarrow A(x,y)$$

The student might also be interested in the validity of this formula without having to do the proof over again. The goal is then to provide a proof of validity for the latter formula, given the derivation of the former.

We start off with a theory declaration importing the base theory:

>   **theory** *Scratch* **imports** *Natural-Deduction-Assistant* **begin**

Again we show the formula in Isabelle syntax (shallow embedding). This time we use the full Isabelle syntax, not just the "typewriter" subset, as the problem of open formulas is considered more advanced:

>   **proposition** ⟨∀x. (∀y. ((∀z. (∀u. A(z, u))) ⟶ A(x, y)))⟩
>   **by** *blast*

Again the derivation is translated to Isabelle (deep embedding), this time as its own **lemma** since we will need it more than once. The derivation itself is omitted for brevity and is similar to the one above:

>   **lemma** *as-given-by-the-user*:
>   ⟨OK (Uni (Uni (Imp (Uni (Uni (Pre ''A'' [Var 1, Var 0]))) (Pre ''A'' [Var 1, Var 0]))))) []⟩

We show that the formula is valid using this derivation and the soundness theorem as before:

>   **theorem**
>   ⟨semantics e f g (Uni (Uni (Imp (Uni (Uni (Pre ''A'' [Var 1, Var 0]))) (Pre ''A'' [Var 1, Var 0])))))⟩
>   **using** *as-given-by-the-user soundness* **by** *blast*

Our function *put-unis m* used below takes a formula and puts *m* universal quantifiers in front of it. Note that its argument here is the open version of the original formula:

>   **lemma** *as-given-by-the-user-any-unis*:
>   ⟨semantics e f g (put-unis m (Imp (Uni (Uni (Pre ''A'' [Var 1, Var 0]))) (Pre ''A'' [Var 1, Var 0])))⟩
>   **proof** −
>   **have** ⟨OK (put-unis 2 (Imp (Uni (Uni (Pre ''A'' [Var 1, Var 0]))) (Pre ''A'' [Var 1, Var 0]))) []⟩
>   **unfolding** *put-unis-def* **using** *as-given-by-the-user* **by** *simp*
>   **then show** *?thesis*
>   **using** *any-unis soundness* **by** *blast*
>   **qed**

The fact *any-unis* is explained below.

Given this lemma we can take $m = 2$ for the validity of the original formula and $m = 1$ or $m = 0$ for the two possible open versions. The latter is what the two corollaries below do:

> **corollary** *without-one-Uni*:
> ⟨*semantics e f g* (*Uni* (*Imp* (*Uni* (*Uni* (*Pre* ″*A*″ [*Var 1*, *Var 0*]))) (*Pre* ″*A*″ [*Var 1*, *Var 0*]))))⟩
> **using** *as-given-by-the-user-any-unis put-unis.simps* **by** *metis*

> **corollary** *without-two-Uni*:
> ⟨*semantics e f g* (*Imp* (*Uni* (*Uni* (*Pre* ″*A*″ [*Var 1*, *Var 0*]))) (*Pre* ″*A*″ [*Var 1*, *Var 0*]))⟩
> **using** *as-given-by-the-user-any-unis put-unis.simps* **by** *metis*

One such corollary is produced for every outer universal quantifier in the given formula. Finally the theory is completed:

> **end**

Theorem *any-unis*   This theorem is a result of having completeness for open formulas and states that given a proof with some number *k* of outer universal quantifiers, a proof with *m* quantifiers, either fewer or more, can be derived.

> **theorem** *any-unis*: ⟨*OK* (*put-unis k p*) [] $\Longrightarrow$ *OK* (*put-unis m p*) []⟩
> **using** *put-unis remove-unis* **by** *blast*

The theorem follows from our two results *remove-unis*, that we can always derive a formula without its quantifiers, and *put-unis*, that we can put any number back in front. Their proofs are omitted for brevity.

5.2.3   The Theorem-Generating Program

Having seen examples of the theorems generated by our program we turn briefly to its implementation. The program is written in Standard ML and, like the generated prover, translated to JavaScript for integration with NaDeA. The program is meant to be used via NaDeA, but the students can also run the Standard ML code inside Isabelle's ML environment if they prefer to do so.

NaDeA supports exporting proofs in a textual format for sharing and archival purposes. The program parses this format into the following corresponding ML datatype:

datatype *proof = Proof **of** rule* ∗ *fm* ∗ *fm list* ∗ *tm option* ∗ *proof list*

Here *rule* is an enumeration type describing the rules and *fm* and *tm* are formulas and terms respectively. The single formula is the result of applying the rule, called the *goal*, while the list of formulas is the current assumptions. The optional term is the constant used for the *Exi_E* and *Uni_I* rules. Finally the list of (smaller) proofs corresponds to the premises of the rule.

This datatype is then translated to Isar by recursion on the list of premises until the *Assume* case is reached. As explained above, the applications of propositional rules are easily translated, as the way NaDeA handles these rules corresponds exactly to the Isabelle formalization.

In the *Exi_I* and *Uni_E* cases, we need to employ a minimal version of unification that determines what constant was used in the substitution. This amounts to traversing the current goal and the goal of the (single) premise in parallel to see what *Var 0* was substituted for. Note that after crossing a quantifier we need to consider variable 1 instead and so forth, symmetrically to how substitution works.

Knowing this allows us to (un)do the substitution before applying the rule, like in the previous example, to make the (premise) goal match the form of the premise or goal in the inductive definition.

Formally verifying the correctness of this program, as we have done for the generated prover, would require formalizing the input, the NaDeA format, as well as the output, a fragment of Isar. We note that even if this program wrongly translates some proofs — or they are not really proofs — Isabelle will in the worst case refuse to verify them.

6   Related Work

Our formalization builds on a formalization by Berghofer [2] but the formalizations and developments described in the present paper are new.

There are several other assistants for doing proofs in logic such as PANDA [10], Pandora [6], ProofWeb [1] and the Incredible Proof Machine [4]. Closest to our approach are ProofWeb and The Incredible Proof Machine since they also use aspects of proof assistants. ProofWeb does this by providing interaction between proof assistants (Coq, Isabelle and Lego) and a web interface. NaDeA focuses on natural deduction and so does the Incredible Proof Machine where proofs are performed in a novel graphical representation forming directed acyclic graphs. However, there is no direct interface with a proof assistant, but Breitner and Lohner [5] formalize in Isabelle/HOL the correspondence between their graphs and more classical proof trees, building on Blanchette, Popescu and Traytel's [3] abstract approach to formalizing soundness and completeness.

The idea of formalizing logics goes beyond the first-order case. Kumar, Arthan, Myreen and Owens [14] formalize higher-order logic with definitions and prove it sound. Likewise, the idea of using proof assistants in teaching is not limited to teaching logics. Nipkow and Klein's textbook [15] on programming language semantics teaches the topic on top of Isabelle.

7   Conclusion

This paper explains how we extended NaDeA with two features that connect it to Isabelle. The first one consists of a tableau prover verified in Isabelle which tries to prove the subgoals of the student's goals and reports back whether it succeeded or not. This is supposed to indicate to the students whether they are on the right track or not. The second one exports proofs done in NaDeA to proofs in Isabelle. The proofs are in a deep embedding of NaDeA's proof system. Because of the soundness of this system we can conclude in Isabelle that the proved theorem is valid.

We have successfully classroom tested NaDeA in a regular course with 70-80 bachelor students in computer science each year since 2015 (DTU 5 ECTS course 02156 Logical Systems and Logic Programming). The students spend around 15 hours on lectures and exercises plus around 3 hours on a hand-in assignment. In total more than 25 formulas are proved. A condensed course was offered for PhD students at the European Summer School in Logic, Language and Information, University of Toulouse, France, 17-28 July 2017.

The current version of NaDeA is 0.9.9 and after some polishing we plan to release 1.0.0 very soon. The source code is available on GitHub (MIT open source license). Future work includes investigation into other ways of integrating NaDeA and Isabelle.

We hypothesize that proof assistants will be an indispensable tool for tomorrow's computer scientists and engineers. Their use requires a solid understanding of logic, and we hope that NaDeA can serve as a safe playground where our students can gain such an understanding by studying natural deduction

in a controlled environment. By integrating NaDeA and Isabelle, we not only increase the trust in our system, but also practice what we preach – following our hypothesis, the Isabelle proof assistant is an indispensable part of NaDeA's design.

Acknowledgements

References

[1] *ProofWeb. (ProofWeb is both a system for teaching logic and for using proof assistants through the web).* Available at `http://proofweb.cs.ru.nl/login.php`. Accessed December 2017.

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

[3] Jasmin Christian Blanchette, Andrei Popescu & Dmitriy Traytel (2014): *Unified Classical Logic Completeness - A Coinductive Pearl*. Lecture Notes in Computer Science 8562, Springer, pp. 46–60, doi:10.1007/978-3-319-08587-6_4.

[4] Joachim Breitner (2016): *Visual Theorem Proving with the Incredible Proof Machine*. In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pp. 123–139, doi:10.1007/978-3-319-43144-4_8.

[5] Joachim Breitner & Denis Lohner (2016): *The meta theory of the Incredible Proof Machine*. Archive of Formal Proofs. `http://isa-afp.org/entries/Incredible_Proof_Machine.html`, Formal proof development.

[6] Krysia Broda, Jiefei Ma, Gabrielle Sinnadurai & Alexander J. Summers (2007): *Pandora: A Reasoning Toolbox using Natural Deduction Style*. Logic Journal of the IGPL 15(4), pp. 293–304, doi:10.1093/jigpal/jzm020.

[7] Martin Elsman (2011): *SMLtoJs: Hosting a Standard ML Compiler in a Web Browser*. In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, ACM, New York, NY, USA, pp. 39–48, doi:10.1145/2093328.2093336.

[8] Melvin Fitting (1996): *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science, Springer, doi:10.1007/978-1-4612-2360-3.

[9] Andreas Halkjær From (2017): *Formalized First-Order Logic*. BSc Thesis, Technical University of Denmark.

[10] Olivier Gasquet, François Schwarzentruber & Martin Strecker (2011): *Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students*. Lecture Notes in Computer Science 6680, Springer, pp. 85–92, doi:10.1007/978-3-642-21350-2_11.

[11] John Harrison (1998): *Formalizing Basic First Order Model Theory*. In: *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, Springer, pp. 153–170, doi:10.1007/BFb0055135.

[12] Michael Huth & Mark Ryan (2004): *Logic in Computer Science: Modelling and Reasoning about Systems. Second Edition*. Cambridge University Press, doi:10.1017/CBO9780511810275.

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

[14] Ramana Kumar, Rob Arthan, Magnus O. Myreen & Scott Owens (2014): *HOL with Definitions: Semantics, Soundness, and a Verified Implementation*. Lecture Notes in Computer Science 8858, Springer, pp. 308–324, doi:10.1007/978-3-319-08970-6_20.

[15] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics - With Isabelle/HOL*. Springer, doi:10.1007/978-3-319-10542-0.

[16] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283, Springer, doi:10.1007/3-540-45949-9.

[17] Daejun Park, Andrei Stefănescu & Grigore Roşu (2015): *KJS: A Complete Formal Semantics of JavaScript*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, ACM, New York, NY, USA, pp. 346–356, doi:10.1145/2737924.2737991.

[18] Jonathan P. Seldin (1989): *Normalization and excluded middle. I*. Studia Logica 48(2), pp. 193–217, doi:10.1007/BF02770512.

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

[20] Makarius Wenzel (2017): *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/dist/doc/isar-ref.pdf.