EPTCS 400

Proceedings of the

12th International Workshop on Theorem proving components for Educational software

Rome, Italy, 5th July 2023

Edited by: Julien Narboux, Walther Neuper and Pedro Quaresma

Published: 6th April 2024
DOI: 10.4204/EPTCS.400
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii
Julien Narboux, Walther Neuper and Pedro Quaresma

Invited Presentation: The challenges of using Type Theory to teach Mathematics 1
Yves Bertot

WebPie: A Tiny Slice of Dependent Typing 2
Christophe Scholliers

Using Large Language Models for (De-)Formalization and Natural Argumentation Exercises for Beginner's Students 28
Merlin Carl

Improving the Diproche CNL through Autoformalization via Large Language Models 44
Merlin Carl

Teaching Higher-Order Logic Using Isabelle 59
Simon Tobias Lund and Jørgen Villadsen

A Coq Library of Sets for Teaching Denotational Semantics 79
Qinxiang Cao, Xiwei Wu and Yalun Liang

Waterproof: Educational Software for Learning How to Write Mathematical Proofs 96
Jelle Wemmenhove, Dick Arends, Thijs Beurskens, Maitreyee Bhaid, Sean McCarren, Jan Moraal, Diego Rivera Garrido, David Tuin, Malcolm Vassallo, Pieter Wils and Jim Portegies

Interactive Formal Specification for Mathematical Problems of Engineers 120
Walther Neuper

Preface

Julien Narboux
University of Strasbourg
France
jnarboux@narboux.fr

Walther Neuper
JKU Johannes Kepler University
Linz, Austria
walther.neuper@jku.at

Pedro Quaresma
University of Coimbra
Portugal
pedro@mat.uc.pt

The ThEdu series pursues the smooth transition from an intuitive way of doing mathematics at secondary school to a more formal approach to the subject in STEM education, while favouring software support for this transition by exploiting the power of theorem-proving technologies. What follows is a brief description of how the present volume contributes to this enterprise.

The 12th International Workshop on Theorem Proving Components for Educational Software (ThEdu'23) was a satellite event of the 29th international Conference on Automated Deduction (CADE 2023), July 1–4, 2023, Rome, Italy. ThEdu'23 was very successful, with one invited talk, by Yves Bertot (Inria, France), "The challenges of using Type Theory to teach Mathematics", and seven regular contributions. An open call for papers was then issued, to which eight contributions were submitted. Seven submissions have been accepted by our reviewers, who jointly produced at least three careful reports on each of the contributions. The resulting revised papers are collected in the present volume.

The contributions in this volume describe the many systems/approaches that share the goal of reaching a wider audience. In doing so, they aim to promote the possibilities of using automatic deduction systems/approaches in education. In his invited talk, Yves Bertot addressed the challenges of using Type Theory to teach Mathematics, the abstract of which is here contained in HTML format only. Christophe Scholliers presents a small dependently typed programming language called WebPie; the author believes that his article can provide a step forward towards the understanding and systematic construction of dependently typed languages for researchers new to dependent types. Merlin Carl describes two systems currently being developed that use large language models for the automatized correction of (i) exercises in translating back and forth between natural language and the languages of propositional logic and first-order predicate logic and (ii) exercises in writing simple arguments in natural language in non-mathematical scenarios. In a second article Merlin Carl introduces the Diproche system. An automated proof checker for texts written in a controlled fragment of German, designed for didactical applications in classes introducing students to proofs for the first time. Simon Tobias Lund and Jørgen Villadsen present a formalization of higher-order logic in the Isabelle proof assistant built in such a way that it can serve as a good introduction for someone looking into learning about higher-order logic and proof assistants. Qinxiang Cao, Xiwei Wu and Yalun Liang present a small Coq library of sets and relations, that they have developed, in such a way that standard math notations can be used when teaching denotational semantics of simple imperative languages. Jelle Wemmenhove et al. introduce the system "Waterproof", an adaptation of the Coq proof assistant into an educational tool in order to help students learn how to write mathematical proofs. Walther Neuper presents the second part (see EPTCS 328) of a precise description of the prototype that has been developed in the course of the ISAC project. The ISAC project aims to develop software to support the most frequently encountered learning scenarios, the solving of mathematical problems. This second part of the description concerns the specification phase, which precedes the solving of engineering problems.

For their help in reviewing the papers, the editors would like to thank the other members of the Pro-

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. ii–iii, doi:10.4204/EPTCS.400.0

© J. Narboux & W. Neuper & P. Quaresma
This work is licensed under the
Creative Commons Attribution License.

gramme Committee of ThEdu'23: Francisco Botana (University of Vigo at Pontevedra, Spain), David Cerna (Johannes Kepler University, Austria) João Marcos (Federal University of Rio Grande do Norte, Brazil), Filip Marić (University of Belgrade, Serbia), Julien Narboux (University of Strasbourg, France), co-chair, Adolfo Neto (Federal University of Technology – Parana, Brazil), Walther Neuper (Johannes Kepler University Linz, Austria), co-chair, Pedro Quaresma (University of Coimbra, Portugal), co-chair, Vanda Santos (University of Aveiro, Portugal), Anders Schlichtkrull (Aalborg University, Denmark), M. Pilar Vélez (Nebrija University, Spain) and Jørgen Villadsen (Technical University of Denmark, Denmark) and the external reviewers Pierre Boutry (University of Strasbourg, France) and Frédéric Tran-Minh (ESISAR Grenoble INP, Valence, France).

We, the volume editors, hope that this collection of papers will further promote the development of theorem-proving based software, and that it will allow to improve the mutual understanding between computer scientists, mathematicians and stakeholders in education.

While this volume goes to press, the next edition of the ThEdu workshop is being prepared: ThEdu'24 will be a satellite event of the International Joint Conference on Automated Reasoning, Nancy, France, July 1–6, 2024.

PC Chairs:
Julien Narboux (University of Strasbourg, France)
Walther Neuper (JKU, Johannes Kepler University, Linz, Austria)
Pedro Quaresma (University of Coimbra, Portugal)

The challenges of using Type Theory to teach Mathematics (Invited Talk)

Yves Bertot
INRIA, France
yves.bertot@inria.fr

The last decade has seen tremendous progress in the adoption of Type Theory-based interactive theorem provers to describe mathematical results, as illustrated by formal verification in group theory (the Feit-Thompson theorem, using Coq) and recent work in algebraic geometry (Formalisation of perfectoid spaces, using Lean).

Much of this work relies on expertise in using the proof systems, so that the question of applying these advances to teach mathematics remains an open problem. In the context of education, one must take into account the ever-changing level of expertise of the audience, the need to exercise skills to acquire them, etc.

In this talk, I will review a variety of features of Type Theory-based interactive provers that actually make these systems hard to use for teaching mathematics and reflect on corrective actions. In particular, I will review the issues that come specifically from the distance between the language of Type Theory and the language used in introductory mathematics at the level of undergraduate mathematics.

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 1–1, doi:10.4204/EPTCS.400.1

© Yves Bertot
This work is licensed under the
Creative Commons Attribution License.

WebPie
A Tiny Slice of Dependent Typing

Christophe Scholliers
Department of Mathematics, Applied Statistics and Informatics
Ghent University
christophe.scholliers@ugent.be

Dependently typed programming languages have become increasingly relevant in recent years. They have been adopted in industrial strength programming languages and have been extremely successful as the basis for theorem provers. There are however, very few entry level introductions to the theory of language constructs for dependently typed languages, and even less sources on didactical implementations. In this paper, we present a small dependently typed programming language called WebPie. The main features of the language are inductive types, recursion and case matching. While none of these features are new, we believe this article can provide a step forward towards the understanding and systematic construction of dependently typed languages for researchers new to dependent types.

1 Introduction

Dependent types are an interesting and fascinating subject with many recent developments [1, 2, 3, 4, 5, 6, 7]. Many theorem provers such as Coq [3] and Agda [2] make use of some form of dependent types. Unfortunately, the theory and implementation of full fledged systems such as Coq and Agda are complex and consists of thousands of lines of code. On the other hand, entry level textbooks on dependent types only give an overview of a core calculus omitting crucial aspects such as inductive types, dependent case matching and recursion. This article aims to bridge the gap between entry level textbooks on types such as "Types and Programming Languages" [8] and the theory of full fledged theorem provers based on dependent typing. While this article provides an entry level explanation on dependent types readers are expected to be familiar with basic type theory and functional programming [1].

The explanation of the dependently typed language constructs and their implementation is done in two stages. In a first stage, we present WebLF, a very basic dependently typed programming language inspired by the logical framework [6]. In a second stage, we add inductive types and motivate how the programmer benefits from them. The resulting WebPie language is available online [2] with several proof examples showing that the language can be used for small proofs.

The WebPie language is conceived as a language laboratory to further investigate the language features that are present in dependent type theory. The current implementation does not feature support for advanced features such as tactics for interactive theorem proving. By making these simplifications we hope to hit a sweet spot where the reader can be inspired to understand dependent typing and how to implement their own dependently typed programming language without being overwhelmed with the difficulties.

---

[1] In particular, readers are expected to be familiar with basic type systems as described in the book "Types and Programming languages" [8] up to System-F-ω. Furthermore, knowledge of basic functional programming language constructs [9] such as algebraic datatypes is expected.

[2] https://users.ugent.be/∼chscholl/WebPie/

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 2–27, doi:10.4204/EPTCS.400.2

© Christophe Scholliers
This work is licensed under the
Creative Commons Attribution License.

2 Getting Familiar with Dependencies

Often dependent types are perceived as exotic and difficult. There is however, no reason why dependent types are any more difficult than other concepts in computer science. The idea of dependency is in fact pervasive in many elementary programming language constructs. When using these constructs most programmers do not consciously think about the dependencies they are making. As a stepping stone towards dependent types it is worthwhile to make these dependencies in basic language constructs explicit. Let us take a look at three examples which should be familiar to most programmers.

Pure functions define dependencies between their arguments and return values. For example, the `increment` function when called with the argument 3 will return the value 4, which of course depends on the given argument 3. Looking at the types and values involved, functions capture a *dependency from terms to terms*.

Generics A second (distinct) form of dependency is encountered when making use of generics in programming languages such as Java [10]. When making a new container class (for example `new Vector <Integer>()`), the programmer explicitly passes a type to the container to receive a constructor that creates instances of a new type of container.

Important to note, is the dependency between the constructor and the given type. Similarly to how pure functions represent terms which dependent on other terms, generics encode *terms which depend on types*. This also means that in order for a system like this to exist that suddenly the distinction between types and terms are starting to blur. In systems without generics there is no way for the programmer to compute with types. They only serve as annotations. In this system however, the programmer can use types as if they would be terms. The use of types in this example is however, limited to passing types as arguments to create terms.

Type constructors In a system with generics as described before, we can only make a `Vector< Integer>` constructor, but we still do not have the power to protect functions to differentiate between a vector of `Integers` or of `Strings`. Also, at the type level, we need to be able to create new types. It would be silly/impossible to create a table with all possible types we would ever need in our programs. Therefore, many languages introduce the concept of a type level abstraction. In such a system, we can create functions which create other types. For example,`Vector<Integer>` is a function which receives a type and returns the type “Vector which contains Integers”. This means that type constructors capture a dependency from types to types.

Name	Dependency
Function	Term $\mapsto$ Term
Generics	Type $\mapsto$ Term
Type Abstraction	Type $\mapsto$ Type

Table 1: Type dependency overview.

Table 1 shows the different kinds of dependencies observed in the three examples. Attentive readers may have noticed there is one dependency missing from the table (terms to types). The idea of dependent types is to create a system that captures a dependency from *terms to types*. In the next section, we give an example of why such a system could be useful. In the remainder of the paper, we give an overview of how to implementation a tiny dependently typed language in JavaScript.

3 Dependent Types by Example

As it might not be obvious why dependent types might be useful we give a small practical example. After this practical example we look into how dependent types can be used as a proof system.

In many languages there is a function which allows the programmer to print a number of arguments by means of a format string. This format string indicates what the type of the arguments of the rest of the arguments should be. For example in C the following piece of code shows how the `printf` function is applied to two arguments. The first argument is the format string in which, `%d`, indicates that there should be a second argument which is a number. As expected executing this piece of code will print `"The number is 10"`.

```
printf("The number is %d", 10);
```

Unfortunately, the type of the arguments after the format string are not type checked and thus can lead to erroneous behaviour at runtime. In the context of dependent types the crucial insight is that the type of the `printf` function *depends* on the format string (value) given as an argument. For example, the following two invocations to the printf functions have a different type.

```
printf :: String -> Int -> Void
printf("The number is %d",  ... );

printf :: String -> String -> Void
printf("The string is %s", ... )
```

In a dependently typed programming language it is possible to define `printf` as a dependently typed function whose range type depends on the format string it receives. The definition of such a function is shown below.

```
def printf(s : String) : (computeType s) {
  % create a function based on the format string
}
```

The syntax `(s: String)` makes the argument `s` accessible in the rest of the type signature. In our case the range of the type is computed by a function `computeType`. The type checker is able to verify that for any possible format string, the `computeType` function correctly computes the type of the function returned by the `printf` function. The type of the `printf` function is a *dependent type* which is typically defined using the Π symbol. The type of the `printf` function is thus `Πs:String.(computeType s)`. Note that, Π, generalises traditional function types when the arguments of the dependent type are not used in the rest of the function signature. For example, the type `Πa:Int.String` is equivalent to `Int->String`.

4 WebLF: Dependent Types for Proof Assistance

WebLF is an implementation of a logical framework [6]. In a logical framework the developer can define the syntax, rules and proofs of formal systems by means of a dependently typed lambda calculus. In order to prove propositions, a logical framework exploits the correspondence between computer programs and mathematical proofs. This correspondence known as the Curry-Howard correspondence[11], states that there is an isomorphism between *types and propositions* and *proofs and programs*. For example, logical implication $\supset$ corresponds to function type $\rightarrow$. Universal quantification $\forall$ corresponds to the dependent function type Π. Given the Curry-Howard correspondence stating propositions corresponds to defining a type while giving a proof consists of giving an implementation for the given type.

In WebLF, defining the syntax of the formal system consist of stating axioms to define what the objects of the formal system are and how to construct them. Defining the semantics of the formal system consist of defining a set of judgements over these objects. Once the syntax and semantics of the formal system are defined, the programmer can use WebLF to prove propositions over the system.

In WebLF, there are two essential constructs `Axiom` and `def`. As the name indicates, `Axiom` allows the programmer to postulate assumptions which are taken for granted without giving an explicit proof for them. For example, to state that there is a set of natural numbers the programmer can state: `Axiom Nat : Set`. Here `Set` denotes the type of proper types, i.e. `Set` corresponds to kind star in Haskell.

With the `def` construct the programmer can state and prove propositions following the Curry-Howard correspondence. The syntax of `def` follows the syntax of ordinary function definitions found in many programming languages. To make this more concrete let us look at two small examples of how to encode formal systems in WebLF. The first example consist of encoding first-order logic, the second example encodes the natural numbers (with induction) [3].

4.1 First-Order Logic

In order to model first-order logic in WebLF we first declare a new set to represent the objects (formulas) of the logic (as shown in figure 1). This is done by adding an axiom stating that there is a set `o`. The type of the set of objects itself is `Set`.

```
% Syntax
% o are propositions
Axiom o : Set;
Axiom ⊃ : o → o → o;

% Judgment
Axiom true : o → Set;
%       φ
%       .
%       ψ
%   ----------- [ Imp-I ]
%     φ ⊃ ψ
Axiom impI : Πp:o.Πq:o.
               ((true p) → (true q)) →
               (true (⊃ p q));
```

Figure 1: WebLF: Syntax and semantics of FOL.

Then axioms for each of the constructors of `o` are added. Here we only show the constructor for implication ⊃. Its type indicates that ⊃ expects two objects and that it returns a new object. If `p` and `q` are two objects, `(⊃ p q)` is also an object.

Subsequently, we define what it means for an object of the logic to be true. This is done by stating that there is a judgment `true` over objects. Encoding a judgement in WebLF is done by defining a set which is parametrised over the defined objects.

Then a single introduction rule, implication introduction, is defined. In order to express the type of this introduction rule we need a dependent type. In WebLF dependent types are written as `Πn:T.X` where `X` might refer to the variable `n`. If `X` does not refer to `n` the type can be written more traditionally as `T→X`.

---

[3] More elaborate examples are available online.

The type of the introduction rule `impI` thus denotes that the introduction rule is a function expecting two objects (`p` and `q`) and a function which can transform any program `(true p)` into a program `(true q)`. Given these arguments `impI` returns a program `(true (⊃ p q))`.

Given these definitions we have enough axioms to define and prove our first proposition. In Figure 2 a function `imp_a_b_a` is defined. Through the Curry-Howard correspondence between `Π` and $\forall$ its type signature `ΠA:o.ΠB:o.(true (⊃A (⊃B A)))` encodes the proposition $\forall A,B.A \supset B \supset A$. The term in the body of the function is the program/proof for this type/proposition. It is built by a nested application of the `impI` rule. The outer application of `impI` binds the variable `p` to `A` and the variable `q` to `(⊃B A)`. The third argument to `impI` thus must be a function with the following signature `(true A)→(true (⊃B A))`. This function is constructed on lines 4-5. The argument `a` of the function can be used to construct a proof for `(true (⊃ B A))` by a second application of the `impI` rule. In this second application the variable `p` is bound to `B` and the variable `q` to `A`. We therefore, need to supply a function with the following signature `(true B)→(true A)`. This is quite easy because the variable `a` is exactly `(true A)`. It is thus sufficient to return the variable a. The inner application of `impI` thus returns us a proof of `(true (⊃ B A))`. Therefore, the outer application of `impI` returns us a proof `(true (⊃ A (⊃ B A)))`, exactly what we needed.

```
1 %  |- A ⊃ B ⊃ A.
2 def imp_a_b_a(A:o,B:o) : (true(⊃ A (⊃ B A))) {
3   (impI A (⊃ B A)
4         (λa:(true A).
5             (impI B A (λb:(true B).a))))
6 }
```

Figure 2: WebLF: Proof example in the domain of FOL.

At his point it is worth noting that there are no guarantees provided by WebLF (and other proof assistants) that the encoded formal system corresponds to the actual system.

4.2 Natural Numbers and Induction

In most cases simply applying the set of constructor axioms will not be sufficient to prove more interesting propositions. For example, to prove interesting properties over the natural numbers we will often need induction as a reasoning principle. In WebLF such reasoning principles can be added as an axiom. Such an encoding is shown in Figure 3.

First the Set `Nat` is defined with two constructors `z,s` denoting zero and successor. Then a judgement `plus` is defined which relates three numbers where the third number is the sum of the first two. Two rules for this judgement are defined. The first, `plus_zero`, encodes that adding `z` to any number `x` is simply `x`. The second, `plus_x_y`, encodes that when given a proof `(plus x y z)` it can be deduced that `(plus x (s y)(s z))`.

Subsequently the induction principle over `Nat`, `nat_ind`, is defined. This encoding states that for all propositions `P` over `Nat`, when given a proof for that proposition for `z` and a function `p_succ` that when given any number `x` and a proof of `P x` gives a proof of `(P (s x))` we can derive that the proposition holds for all elements in `Nat`.

Having defined `Nat`, `plus` and `nat_ind` we can state and prove that for all `Nat` we can find a proof that `(plus z x x)`. In spite of being a bit verbose the proof `plus_zero_x` follows standard reasoning by induction.

```
Axiom Nat : Set;
Axiom z   : Nat;
Axiom s   : Nat->Nat;

% Adding two numbers
Axiom plus      : Nat -> Nat -> Nat ->Set;
Axiom plus_zero : Πx:Nat.(plus x z x);
Axiom plus_x_y  : Πx:Nat.Πy:Nat.Πz:Nat.
                 (plus x y z) ->
                 (plus x (s y) (s z));

% Induction principle for Nat
Axiom nat_ind   : ΠP:Nat->Set.
                  Πp_zero:(P z).
                  Πp_succ:(Πx:Nat.Πp:(P x).
                                   (P (s x))).
                  Πx:Nat.(P x);

def plus_zero_x( x:Nat ) : (plus z x x) {
  ((nat_ind (λx:Nat.(plus z x x))
    (plus_zero z)
    (λx:Nat.
     (λIH:(plus z x x).
      (plus_x_y z x x IH))))
   x)
}
```

Figure 3: WebLF: Induction and natural numbers.

5 A Formal Introduction to WebLF

Now that we have given an overview of how to work with WebLF, we will give an overview of the formalisation of WebLF.

5.1 Syntax

Figure 4 shows the syntax of WebLF. An expression is either a variable, a type universe, a lambda abstraction, a Π type or an application. Type universes are needed to denote the types of types, and the types of types of types and so on. The index *i* in a type universe is a number indicating the universe level.

$$
\begin{array}{lcll}
e & ::= & & \\
 & | & x & \textit{variable} \\
 & | & \mathtt{Type}_i & \textit{sorts} \\
 & | & \lambda x : e.e & \textit{abstraction} \\
 & | & \Pi x : e.e & \textit{abstraction} \\
 & | & e\ e & \textit{application} \\
 & | & \mathtt{Axiom}\ x : e & \textit{Axiom}
\end{array}
$$

Figure 4: WebLF Expressions.

5.2 Type Rules

The typing rules for our dependently typed lambda calculus are shown in Figure 5. The type of a variable is looked up in the context Γ. The type of an abstraction is a Π type but only when the expression denoting the type of the function e_t has type $\texttt{Type}_\texttt{i}$. The type of a Π type is a universe where the level is the maximum of the universe types of the argument and the body. The type of a universe is the universe type where the level is incremented by one. Finally, the type of an application is obtained by substituting the term t_2 in the return type e_r of the function type. This is only correct when the argument type is equal to the expected type $e_t = e_a$. This equivalence is defined by normalisation as shown in the next section.

$$\boxed{\Gamma \vdash x : \mathrm{T}}$$

$$\frac{x : \mathrm{T} \in \Gamma}{\Gamma \vdash x : \mathrm{T}} \ (\text{T-VAR}) \qquad \frac{\Gamma, x : e_t \vdash e_b : e_{tb} \qquad \Gamma \vdash e_t : \texttt{Type}_\texttt{i}}{\Gamma \vdash \lambda x : e_t . e_b : \Pi x : e_t . e_{tb}} \ (\text{T-ABS})$$

$$\frac{\Gamma \vdash e_t : \texttt{Type}_\texttt{i} \qquad \Gamma, x : e_t \vdash e_b : \texttt{Type}_\texttt{j}}{\Gamma \vdash \Pi x : e_t . e_b : \texttt{Type}_{\texttt{max(i,j)}}} \ (\text{T-PI})$$

$$\text{T-UNIV} \ \frac{}{\Gamma \vdash \texttt{Type}_\texttt{i} : \texttt{Type}_{\texttt{i+1}}}$$

$$\frac{\Gamma \vdash t_1 : \Pi x : e_t . e_r \qquad \Gamma \vdash t_2 : e_a \qquad e_t = e_a}{\Gamma \vdash t_1\, t_2 : [x \mapsto t_2] e_r} \ (\text{T-APP})$$

Figure 5: WebLF typing rules

5.3 Substitution

Substitution is different from usual substitution in the simply typed lambda calculus as shown in Figure 6. When substituting variables over an abstraction, substitution is performed both at the type level and in the function body. Just as in the simply typed lambda calculus, it is important to avoid variable capturing in the rule for abstraction and Π types. In an implementation it is necessary to perform α renaming in case variables would be captured, or make use of a nameless encoding such as De Bruijn index [12].

$$\begin{array}{lcl}
[x \mapsto s]\ x & = & s \\
[x \mapsto s]\ y & = & y \\
 & & y \neq x \\
[x \mapsto s]\ \texttt{Type}_i & = & \texttt{Type}_i \\
[x \mapsto s]\ \lambda y : e_t . e_b & = & \lambda y : [x \mapsto s] e_t . [x \mapsto s] e_b \\
 & & y \neq x \wedge y \notin Fv(s) \\
[x \mapsto s]\ \Pi y : e_t . e_b & = & \Pi y : [x \mapsto s] e_t . [x \mapsto s] e_b \\
 & & y \neq x \wedge y \notin Fv(s) \\
[x \mapsto s]\ (e_t\ e_b) & = & ([x \mapsto s] e_t\ [x \mapsto s] e_b)
\end{array}$$

Figure 6: WebΠ substitution.

5.4 Normalisation and Equivalence

In the T-App rule the type checker needs to be able to compare two expressions. Equivalence of expressions is defined here by evaluating both expressions and then check whether the evaluated values are syntactically equivalent. Traditional evaluation would leave certain expressions in a form where equivalent terms are not syntactically equivalent. In order to obtain a better approximation we define an alternative evaluation strategy which *normalises* expressions so that equivalent terms are also syntactically equivalent. Without normalisation two expressions could be equivalent but have a different syntactic form. Figure 17, in the appendix gives the big step normalisation rules for WebΠ. Cases for expression not shown in this figure are already normalised.

6 Implementing WebLF

In this section we give an overview of the implementation of the WebLF language. There are two major components to the implementation of WebLF, first the parser transforms the input text into a data structure. This data structure is then used by the type checker in order to verify that the provided program is type correct. While we try to be as complete as possible, we have deliberately omitted a number of details in our explanation.

First, as we try to focus on the implementation of the dependently typed programming language we do not explain any details about the parser.

Second, during type checking things might go wrong because the user has supplied the type checker with an erroneous program. WebLF has a number of places in the type checker to inform the programmer about what might be wrong with the program. In the exposition below we omitted the error handling code as we believe it would distract the reader from more important matters.

6.1 Representation of Expressions

In our implementation we make use of adt-simple, a JavaScript library for algebraic datatypes [4]. Datatypes are expressed with the keyword `union`, similar to `data` in Haskell. The `deriving` keyword allows certain operations over the algebraic datatypes to be generated by the adt-simple library. For example, deriving a function to determine the equality of expressions is done by deriving `adt.Eq`. Similarly, we derive the ability to pattern match over an algebraic data type with `adt.Extractor`. Finally, deriving a function to print an algebraic data type is done by deriving `adt.ToString`. Algebraic datatypes are defined using, JSON syntax.

In the figure below we define the `Expression` datatype. To keep the implementation simple, variables names are represented with strings. Note that in our definition we derive `Eq`, `Extractor` and `ToString`.

```
union Expression {
 Variable {value    : String},
 Universe {value    : Number},
 Lambda   {variable : String,
           type     : Expression,
           body     : Expression},
 Pi       {variable : String,
           type     : Expression,
           body     : Expression},
```

---

[4] More information on how to install and use the library can be found on: https://www.npmjs.org/package/adt-simple

```
 App       {fun      : Expression,
            arg      : Expression}
} deriving (adt.Eq,adt.Extractor,adt.ToString)
```

6.2 The Environment

In order to keep track of which variables have which types a type checker maintains an environment with variable-type bindings. The operations on this environment consist of extending the environment with a new binding and looking up the binding of a variable. In WebLF such an environment would theoretically be enough, but as WebLF also allows programmers to introduce definitions it not only keeps track of the variable-type bindings but also of variable-value bindings. Apart from this somewhat strange binding relation the environment supports the basic operations one would expect. Extending the environment with a variable-type binding: `extend_type(var, type, env)`, extend the environment with a type-value binding: `extend_type_value(var, value, env)` and operations to lookup values and types `lookup_val(var,env)`, `lookup_type(var,env)`.

6.3 Typing Expressions

The type checker is implemented as a function, `type_check`. This function receives two arguments, the context and the expression to type and returns the type of the expression. In the code snippet below we see that the function does a case match over the given expression. In case of a variable, the type of the variable is looked up in the context. When it is a universe the type is the universe level plus one. This corresponds to the T-VAR and T-UNIV rules. The other cases are bit longer and are split up in separate functions.

```
// Context -> Expression -> Type
function type_check(ctxt, expr) {
     match expr {
       // ctxt |- x : ?
       case: Variable { value : x } :
         return  lookup_type(x,ctxt);
       // ctxt |- (Type i) : ?
       case: Universe { value : i } :
         return  Universe(i+1);
       case: Lambda{variable:x,type:t,body:b} :
         return check_lam_type(x,t,b,ctxt);
       //...
     }
}
```

In order to typecheck a lambda expression we first determine the type of the body under the assumption that the variable binding has the given type. In the implementation this is done by extending the context with a variable type binding and recursively calling `type_check` (Line 2-3). Then we determine the type of the type declaration itself. In order to be properly typed the type declaration on lambda expression should be a universe type. The resulting type is a Π type.

The most interesting type checking rule is the rule for application. We first determine the type of the function `f` and the argument `a`. The type of the function should be a Π type and its argument type `typef.type` should be equal to the type of the argument. The resulting type is built by substituting the binding variable of the Π type by the expression `a` in the body of the Π type. In order to compare the two

```
// ctxt |- λx:t.b : ?
function check_lam_type(x,t,b,ctxt) {
   var t_body = type_check(extend_type(x,t,ctxt),b);
   var univ_t = type_check(ctxt,t);
   assertUniverse(univ_t,ctxt);
   return Pi(x,t,t_body, b);
}
```

Figure 7: Type checking lambda abstraction

expressions we make use of a function `check_equal` which first normalises the two expressions and then structurally verifies whether the two expressions are equal.

```
// ctxt |- (f a) : ?
function check_app_type(f,a,ctxt) {
   var typef  = infer_type(ctxt, f);
   var typea  = infer_type(ctxt, a);
   assertPiType(typef,ctxt);
   check_equal(typef.type,typea,ctxt);
   return subst(typef.variable,a,typef.body);
}
```

Figure 8: Type checking applications

The type of a Π type is determined by first type checking the declared type of the argument. Then the body is type checked under the assumption that the binding has the correct type. Both of these types `typet` and `typeb` should be universe types. The resulting type is a universe type at level max of the two universe types.

```
// ctxt |- Πx:t.b   : ?
function check_pi_type(x,t,b,ctxt) {
   var typet = type_check(ctxt, t);
   var ectxt = extend_type(x,t,ctxt);
   var typeb = type_check(ectxt, b);
   assertUniverse(typet,ctxt);
   assertUniverse(typeb,ctxt);
   return Universe(max(typet.value,typeb.value));
}
```

Figure 9: Type checking Π types

6.4 Normalising Expressions

Normalisation closely follows the formal definition. For example, to normalise a lambda abstraction we recursively normalise the type and the body of the lambda.

```
function normalise(e,ctxt) {
  match e  {
          //...
```

```
      case Lambda { variable:x, type:t, body:b } :
         return Lambda(x,normalise(t,ctxt), normalise(b,extend_type(x,t,ctxt
            )));
      //...
  }
}
```

6.5 Limitations

Although extremely concise, WebLF forms the basis of many early proof assistant. It can be of great help by warning the programmer in case there is a reasoning error in his/her proof. Once a proof is accepted by the theorem prover the programmer can be confident that all the reasoning steps are sound. One of the limitations of WebLF is that the programmer needs to provide a very large set of axioms. For example, induction over the natural numbers needs to be explicitly provided as an axiom as shown in section 4. More modern proof assistants have founds solutions to this limitation by providing the programmer with a structured way of defining inductive types. In the next section we show how these inductive types work in an extension to WebLF called WebPie. Subsequently, we show how their formalisation and how to implement them.

7 WebPie: Adding Inductive Types and Case Matching

Instead of letting the programmer define separate axioms one-by-one in WebPie the programmer can define a set of related axioms in one go.

Consider the example of natural numbers, in WebLF the programmer needs to define three axioms one for introducing the `Nat` set and two for the constructors. In WebPie these three related axioms can be defined as one inductive type, shown in Figure 10.

```
Inductive Nat : Set :=
  | Zero : Nat
  | Succ : Nat  → Nat;
```

Figure 10: WebPie: Inductive definition of natural numbers.

At first sight the programmer has not gained a lot by using a slightly different syntax. It is however, important to note that the programmer implicitly indicates that the definition of the set `Nat` is closed. Once the inductive type is defined there is no mechanism in place to extend it. Because inductive definitions are closed the only possible constructors for creating elements in `Nat` are `Zero` and `Succ`. Armed with this knowledge it becomes feasible to provide better support to the programmer. For example, it becomes possible to provide case matching and to actually prove the induction principle instead of providing it as an axiom. Without demanding that `Nat` is closed, the programmer could extend the constructors of the `Nat` set which would make type checking much more difficult.

Case Matching

Case matching allows the programmer to inspect a value of an inductive type and determine with which constructor the value was created. Important to realise is that for all of the different cases the case match should return a value of the same type. As an example of case matching consider the implementation of addition in Figure 11.

```
def add(x:Nat,y:Nat) : Nat {
  <(λn:Nat.Nat)>
  match x with {
      Zero   => y;
      Succ   => (λn:Nat.(Succ (add n y)))
  }
};
```

Figure 11: WebPie: Case matching and recursion.

In this example case match `<λn:Nat.Nat>` indicates that we case match over an element in `Nat` and that we will return a `Nat`. `match x with` indicates that we case match over the variable `x`. There are two constructors to consider: `Zero` and `Succ`. When `x = Zero` we simply return `y`. In case the natural number was constructed with `Succ` for example `(Succ Zero)` the function `(λn:Nat. (Succ (add n y)))` will be applied to the arguments of the constructors, i.e. `Zero`. The function returns a new number by adding its argument `n` to `y` and subsequently wraps it into a `Succ`. Note that from an abstract point of view case matching is very much like function application, you supply it with an argument and it returns a certain value.

Dependent case matching

Sometimes regular case matching is not powerful enough to capture the dependencies in our applications. In those cases we can make use of dependent case matching to make an explicit dependency between the return type of a case match and the value over which we case match. As an example, consider the proof of the induction principle over the natural numbers shown in Figure 12.

```
def nat_ind
( P   : Nat → Set,
  f   : (P Zero),
  fn  : (Πn:Nat.(P n) → (P (Succ n))),
   n  : Nat
) :  (P n)
{
   <(λn:Nat.(P n))>
   match n  with {
      Zero   => f;
      Succ   => (λn:Nat. (fn n (nat_ind P f fn n)))
   }
};
```

Figure 12: WebPie: Induction principle over natural numbers.

The encoding of the induction principle at the type level is the same as in section 4. The big difference here is that we do not state the induction principle as an axiom. The body of the function provides a proof that the reasoning principle is indeed correct.

Let us look at the proof term in detail. First of all, it is a lambda abstraction which expects one argument of type `Nat`, given this argument it produces a proof `(P n)` by case matching over n. Because we need to produce a different proof given the argument *n* we make use of dependent case matching. In the example, `<λn:Nat.(P n)>` indicates that we case match over an element in `nat` and that all the cases of the case match will return a proof `(P n)`.

Let us look into detail what this entails when `n` equals `Zero`. In this case the return type can be computed by substituting `n` for `Zero` in the body of the `λn:Nat.(P n)` , which equals `(P Zero)`. The implementation for the `Zero` case is easy because the argument `f` has exactly type `(P Zero)`.

In the second case the return type of the match should be `(P n)`. Implementing this case is a bit more difficult but we conveniently receive a function `fn` which when we provide a proof of `(P x)` will return us a proof of `(P (Succ x))` which is equal to `(P n)`. Obtaining the proof of `(P x)` is done by a recursively call to `nat_ind`.

Readers who are worried at this point about the last recursive call, can be assured that this recursion will stop eventually because the last argument `x` decreases in each recursive call. As this argument decreases in every recursive call, eventually the base case (`n` equals `Zero`) will be reached. In section 8.2, we explain how to the implementation deals with checking this (guard) condition.

Equality over natural numbers

We still need one ingredient to start making simple proofs over `Nat`, namely equality. There are many ways to encode equality in a proof assistant but here we define equality very specifically for natural numbers by defining an inductive type `Eq` (Figure 13). It has two constructors, `Eq_Rfl` and `Eq_Succ`. The first constructor encodes that all natural numbers are equal to itself. The second constructor encodes that when we have a proof that two numbers are equal to each other we can conclude that adding one to each of the numbers results in equal numbers.

```
Inductive Eq : Nat → Nat → Set :=
 | Eq_Rfl   : Πn:Nat.(Eq n n)
 | Eq_Succ : Πx:Nat.Πy:Nat.
              (Eq x y) →
              (Eq  (Succ x) (Succ y));
```

Figure 13: WebPie: Equality over numbers.

Having defined natural numbers, addition, the induction principle over natural numbers and equality of natural numbers it becomes possible to make some small proofs. A first proof `add_zero` shows that for any number `x` adding `Zero` to the left of that number equals `x`. The proof is constructed by applying the constructor `Eq_Rfl` which returns us a proof `(Eq x x)`. WebPie normalises the `(add Zero x)` to `x` and so the term is well typed.

A second proof `add_x_zero` shows that for any `Nat` adding `Zero` to the right equals that `Nat`. This proof is a bit more difficult because WebPie cannot normalise `(add x Zero)` to `x`. The reasons why WebPie cannot normalise this is because the function `add` case matches over the first argument, which could be any `Nat`. We can however, make use of our induction principle `nat_ind` defined earlier.

The function `nat_ind` expects as first argument a function which given a `Nat` gives us a proposition over this `Nat`. In our case we want to prove the proposition `(Eq (add n Zero)n)`.

The second argument is a proof for the case where the `Nat` is `Zero`. Substituting `n` for `Zero` in our proposition gives us `(Eq (add Zero Zero) Zero)`. We can provide a proof for this proposition by applying `Eq_Rfl` to `Zero`. This gives us a proof that `(Eq Zero Zero)`. This is not exactly the same as `(Eq (add Zero`

`Zero)Zero)` but by normalisation WebPie deduces that `(add Zero Zero)` equals to `Zero` and hence that the two proofs are equal.

The third argument is a function which when given a proof for `(P n)` returns us a proof for `(P (Succ n))`. The proof is created by making a new lambda abstraction which takes these arguments and produces the proof `(P (Succ n))` by applying `Eq_Succ`. Note that `Eq_Succ` will return a proof of `(Eq (Succ (add n Zero))(Succ n))`, this is not syntactically equal to `(P (Succ n))` which by simple substitution equals to `(Eq (add (Succ n) Zero) (Succ n))`. The trick here is that WebPie normalise the subterm `(add (Succ n) Zero)` to `(Succ (add n Zero))` making the two terms syntactically equal.

```
def add_zero(x:Nat) : (Eq (add Zero x) x) {
   (Eq_Rfl x)
};

def add_x_zero(x:Nat): (Eq (add x Zero) x) {
  (nat_ind
   (λn:Nat.(Eq (add n Zero) n))
   (Eq_Rfl Zero)
   (λn:Nat.(λIH:(Eq (add n Zero) n).
      (Eq_Succ (add n Zero) n IH)))
   x)
}
```

Figure 14: WebPie: Example proofs over natural numbers.

8 WebPie Formally

We define WebPie as an extension to WebLF, the formal exposition here is a simplification of the exposition as shown in [13]. The extension to the syntax of WebLF are shown in Figure 15. The extensions are inductive definitions `Ind`, constructors `Constr`, case matching `Match`, and recursive functions `Fix` . Inductive definitions are defined by specifying its name x and type e_t together with a list of constructor names x_i and their respective types e_i. A constructor is defined as a pair of the inductive type e_I and a number n indicating the index of the constructor in the inductive type. A dependent case match expects the type of the case match e_{ct} , the expression over which we case match and a list of deconstructors. Finally, recursive functions are defined by specifying a name x and the type of the recursive function e_t. Note that in the formalisation we also need to specify an index k. This index is used to help the type-checker in determining whether executing the recursive function will stop. In essence it indicates which argument of the recursive function is becoming smaller with each recursive call. The intuition is that if the type-checker can determine that an (inductive) argument of a recursive function becomes smaller in each recursive call then eventually the function needs to stop.

8.1 Type Checking Inductive Types

Type checking of inductive types is governed by three typing rules, one for defining the inductive type, one for type checking constructors and finally one for case matching.

$$\begin{array}{lcll} e & ::= & \dots & \\ & | & \texttt{Ind}(x:e_t)\{x_1:e_1|\dots|x_2:e_n\} & \textit{Ind. Type} \\ & | & \texttt{Constr}(e_I,n) & \textit{Constructor} \\ & | & \texttt{Match}(e_{ct},e_m)\{x_{c1}\Rightarrow e_1|\dots|x_{cn}\Rightarrow e_n\} & \textit{Case Match} \\ & | & \texttt{Fix}_{\texttt{k}}(x:e_t)\{e_b\} & \textit{Fix} \end{array}$$

Figure 15: WebPie Syntax Extensions.

Inductive definitions

In order to typecheck the definition of an inductive type we need to verify two things. First, all the constructors of the inductive type should return an object in the same universe. Which should also be the same universe as indicated in the type annotation when defining the inductive type. For example, in Figure 10 the type annotation indicates that objects of type `Nat` should live in universe `Set`. We should verify that under the assumption that `Nat:Set` all of the constructors also return something in type universe `Set`.

In order to get the universe of a Π type we define a function Υ which recursively traverses the type until it encounters a universe.

$$\begin{array}{rcl} \Upsilon(\Pi x:e_m.e) & ::= & \Upsilon(e) \\ \Upsilon(Type_i) & ::= & Type_i \end{array}$$

The second property we need to verify in order to typecheck an inductive definition is that the return type of all the constructors is an element of the inductive type which is being defined. It would of course be a type error when the constructor of a `Nat` returns a `Bool`. The set of all possible well formed constructors for an inductive type with name X is generated by the syntax Co. With the restriction that X does not occurs free in e_n and e_m. The intuition behind this syntax is that the type should end in an application of the inductive type being defined. We use shorthand notation $C(e,x)$ to indicate that the type represented by e is a valid constructor for the type with name x.

$$Co \quad ::= \quad (X\ \overline{e_n}) \mid P \to Co \mid \Pi x:e_m.Co$$

After forming an intuition about the two properties that need to be verified the typing rule for inductive definitions (T-IND) follows directly as shown in Figure 16.

Case Matching

The type checking rule for a match expression is complicated by the fact that there are multiple cases. Consider again the case match for the successor case in Figure 12.

```
<(λn:Nat.(P n))>
  match n  with {
     Zero => f;
     Succ => (λx:Nat.(fn x (nat_ind P f fn x)))
  }
```

In order to typecheck this expression the type checker first needs to ensure that the type of `n` is an inductive type. Once the type checker is sure that `n` is an inductive type it needs to ensure that:

- The list of cases provided by the programmer corresponds with the inductive definition.
- The return type of all the cases corresponds to the type annotation `(Πn:Nat.(P n))` instantiated with the particular case.

The rule T-Match encodes these two guards as shown in Figure 16. Checking that the list of cases is correct, is done by verifying that the list of constructor names in the cases corresponds to the definition i.e. $x_{c1}...x_{cn}$. Note that the typing rule enforces the programmer to write the cases in the same order as in the definition. It would not be that difficult to allow for permutations.

Checking that the type of all the cases is correct is a bit more difficult. In the example, there are two cases `Zero` and `(Succ x)`. To check that the right hand side of the case match is correct we first need to compute what the expected type for that case match should be.

This is done by apply the matched term to e_{ct} `(λn:Nat.(P n))` we respectively get `(P Zero)` and `(P ( Succ x))`. In the typing rule this is ensured by a meta predicate S. This meta predicate given the type of the constructor, the matching type and the constructor builds the appropriate type.

$$
\begin{aligned}
S(\Pi x : e_{tx}.e_r, e_{ct}, c) &= \Pi x : e_{tx}.S(e_r, e_{ct}, (c\ x)) \\
S((x_i\ \overline{e_a}), e_{ct}, c) &= (e_{ct}\ \overline{e_a}\ c)
\end{aligned}
$$

Finally, if all the cases are valid then the type of the entire case match can be derived by applying the type e_t to the parametric arguments of the inductive type over which we case match $\overline{x_m}$ together with the expression over which we case match e_m.

Constructors

Lastly, constructors are represented by an index i and an inductive definition I. Typechecking a constructor is done by verifying that the inductive definition is well typed and ensuring that the index is within bounds of the inductive definition (T-CONSTR).

8.2 Type Checking Recursive Functions

Because a dependently typed language encodes a logic, care needs to be taken in order to make sure that the programmer can only define pure terminating total functions. While some dependently typed programming languages allow the programmer to omit these totality checks [7] it is important to know that in such case the programming language does not ensure sound theorem proving.

Making sure that programs terminate is a well known problem in computer science for which no terminating algorithm exists. Therefore, any termination checking function is a conservative approximation of the set of all functions which terminate. When the termination check succeeds the programmer is sure that the function always terminates. On the other hand when the termination check fails the function might still terminate in practice. Even full fledged proof assistants such as Coq disallow the definition of certain obviously terminating functions. In this section we explain the basics of a simple termination checker. The basic idea of the termination checker is to make sure that at least one of the arguments of all recursive calls has an inductive argument which is decreasing. The guard condition is enforced by the meta predicate β.

$\boxed{\Gamma \vdash x : \mathrm{T}}$

$$\frac{\Upsilon(e_t) = u \qquad \forall i = 1..n\ \Gamma, x : e_t \vdash e_i : u \qquad C(e_i, x)}{\Gamma \vdash \texttt{Ind}(x : e_t)\{x_1 : e_1|...|x_2 : e_n\} : e_t}\ (\text{T-}\textsc{Ind})$$

$$\frac{1 < i <= n \qquad I = \texttt{Ind}(x : e_t)\{x_1 : e_1|...|x_2 : e_n\} \qquad \Gamma \vdash I : e_t}{\Gamma \vdash \texttt{Constr}(i, I) : [x \mapsto I]e_i}\ (\text{T-}\textsc{Constr})$$

$$\frac{\begin{array}{c} I = \texttt{Ind}(x : e_t)\{x_{c1} : e_{t1}|...|x_{cn} : e_{tn}\} \\ \Gamma \vdash e_{ct} : \Pi\overline{x_p : e_{pt}}.(I\ \overline{x_p}) \rightarrow \texttt{Type}_\texttt{i} \qquad \Gamma \vdash e_m : (I\ \overline{x_m}) \qquad \forall i = 1..n, \Gamma \vdash e_i : S(e_{ti}, e_t, x_{ci}) \end{array}}{\Gamma \vdash \texttt{Match}(e_{ct}, e_m)\{x_{c1} \Rightarrow e_1|...|x_{cn} \Rightarrow e_n\} : (e_t\ \overline{x_m}\ e_m)}\ (\text{T-}\textsc{Match})$$

$$\frac{\Gamma \vdash e_t : \texttt{Type}_\texttt{i} \qquad \Gamma, x : e_t \vdash e_b : e_t \qquad \beta\{f, k, arg_k(e_b), \emptyset, e_b\}}{\Gamma \vdash \texttt{Fix}_\texttt{k}(x : e_t)\{e_b\} : e_t}\ (\text{T-}\textsc{Fix})$$

Figure 16: WebPie Typing Extensions.

$$\begin{array}{rll}
\beta\{f, k, x_k, x_g, e\} = & true & f \notin fv(e) \\
\beta\{f, k, x_k, x_g, x_n\} = & x_n \neq f & \\
\beta\{f, k, x_k, x_g, \texttt{Match}(e_{ct}, e_m)\{x_{ci} \Rightarrow e_i\}\} = & \beta\{f, k, x_k, x_g \cup args(e_i), body(e_i)\} \wedge & e_m \in x_g \\
& \beta\{f, k, x_k, x_g, e_t\} \wedge \beta\{f, k, x_k, x_g, e_m\} & \\
\beta\{f, k, x_k, x_g, (x_n\ \overline{x_p})\} = & \forall x_p \in e.\beta\{f, k, x_k, x_g, x_p\}\ \wedge\ \overline{x_p}_k \in x_g \cup x_k & x_n = f \\
\beta\{f, k, x_k, x_g, e\} = & \forall e_s \subset e.\beta\{f, k, x_k, x_g, e_s\} &
\end{array}$$

The basic idea of this meta predicate is to verify whether the kth argument of each recursive call is guarded by a deconstruction. At the same time the meta predicate needs to make sure that aliases to the recursive function do not leak. In this case we simply do not allow the function to be passed on to other functions or to be returned in any form.

The meta predicate β first verifies whether a reference to the recursive function f is present in the expression e, if this is not the case all the recursive calls (none) will terminate. When we encounter a variable that is ok as long as it is not the recursive function f which is being defined. The reason we need to exclude all references to the recursive function is because otherwise we would need to include some form of alias tracking which would significantly increase the complexity of the termination check.

The two most interesting cases are the ones for a case match and application. When encountering a case match over a variable in our guarded variables we recursively apply β over the deconstructors. For each of these deconstructors the arguments of the deconstructors are added to the set of guarded variables. Because the recursive function can be used at the type level we also need to verify whether any recursive call in the type e_t is also guarded.

When verifying the guard condition over an application $(x_n\ \overline{x_p})$ where x_n is the recursive function f being defined the kth argument needs to be guarded and all the arguments need to pass the guard condition. Finally, for all other case the guard condition is recursively applied to all the subexpressions.

8.3 Normalisation

Next to extending the typing rules we also need to extend the normalisation rules to account for case matching. When case matching the term we first normalise the expression on which we case match. If it normalises to a constructor we can further normalise by substituting the arguments to the constructor in the right hand side of the corresponding case match E-Match1. Otherwise the whole term normalises to a new match where e_m has been replaced by its normal form e'_m.

$$\boxed{e \Downarrow n}$$

$$\frac{e_m \Downarrow (\texttt{Constr}(i,I)\overline{e_a}) \qquad [\overline{x_{ai}} \mapsto \overline{e_{ai}})] \Downarrow e'_i}{\texttt{Match}(e_{ct},e_m)\{(x_{c1}\ \overline{x_{a1}}) \Rightarrow e_1|...|(x_{cn}\ \overline{x_{an}}) \Rightarrow e_n\} \Downarrow e'_i}\ \textsc{E-Match-1}$$

$$\frac{e_m \Downarrow e'_m}{\begin{array}{ll}\texttt{Match}(e_{ct},e_m) & \{(x_{c1}\ \overline{x_{a1}}) \Rightarrow e_1|...|(x_{cn}\ \overline{x_{an}}) \Rightarrow e_n\} \Downarrow \\ \texttt{Match}(e_{ct},e'_m) & \{(x_{c1}\ \overline{x_{a1}}) \Rightarrow e_1|...|(x_{cn}\ \overline{x_{an}}) \Rightarrow e_n\}\end{array}}\ \textsc{E-Match-2}$$

Figure 17: WebPie Normalisation Extensions.

9 Implementing WebPie

The implementation of inductive types largely follow the extensions show in the formalisation. We first extend the union for Expressions with inductive definitions, constructors and case matching. Because the JavaScript extension for ADT's does not have native support for arrays the type of constructors and destructor is set to be * which means they could be anything. In our implementation constructors is a list of JSON objects with two fields `name` and `ctype`. Destructors are represented as an array of JSON objects with two fields `name` and `expr`.

```
union Expression {
   // .. extending WebLF
   Inductive { name         : String,
               arity        : Expression,
               constructors : * },
   Constr    { index        : Number,
               inductive    : Expression},
   Match     { carrier      : Expression,
               expr         : Expression,
               destructors  : * }
} deriving (adt.Eq,adt.Extractor,adt.ToString)
```

Figure 18: WebPie: Expression extension.

Extensions to the implementation of the type checker consists of adding cases for the constructors, inductive types and case matching. Because the programmer cannot type in a constructor himself typing the constructor is as simple as returning the field of the inductive definition.

```
function check_inductive(n,a,cst,ctxt) {
  var u = getUniverse(a);
  cst.forEach(function(cst) {
     var tCst = type_check(extend_type(n,a,ctxt),cst[1]);
     var eq   = equal_expr(tCst,u);
     positive(cst[1],n) );
  });
  return a;
}
```

Figure 19: Type Checking Inductive Definitions.

```
 function check_match(c,e,cases,ctxt) {
      var IndT    = normalise(type_check(ctxt,e),ctxt);
      var c_type  = normalise(type_check(ctxt,c),ctxt);
      var Ind     = findInductive(IndT);
      var pargs   = countArgs(Ind.arity);
      verify_c_type(c_type,Ind,pargs);
      var constructors = Ind.constructors;
      checkMatchLength(constructors,cases);
      cases.map( function(cs,i) {
           var cstT = constructors[i].ctype;
           var case_type  = normalise(makeTypeForCase(cstT,c,cs.name),ctxt);
           var e_it = type_check(ctxt,cs.expr);
           var x = check_equal(e_it,case_type,ctxt);
      });
      return normalise(App(makeRApp(c,IndT,pargs),e),ctxt);
 }
```

Figure 20: Type Checking a Match Expression.

Typing inductive definitions consist of checking that the universes correspond and that the constructors are well formed, as shown in Figure 19. In our implementation getUniverse and positive corresponds to Υ and $C(x,e)$ in the formalisation.

Finally type checking case matches (Figure 20) differs a little bit from the formalisation because the syntax of the formalisation and the implement differ slightly. In the implementation the programmer needs to supply an abstraction with the correct number of arguments for the constructor i.e. `Succ => ( λn:Nat. (Succ (add n y)))`instead of `(Succ n)=> (Succ (add n y)))`. The code first verifies whether the value over which we case match is an inductive type. Then it verifies whether the constructors match the cases. We then verify for each of the constructors that the right hand side of the cases match. In the implementation we do this by explicitly building up a Π type and apply normalisation with `makeTypeForCase`.

10 Safe Printf in WebPie

As a final example of using WebPie, we show how to encode the safe `printf` example of the introduction. As WebPie does not have any primitives for input output we merely simulate a safe `printf`.

The implementation of our safe `printf` (shown in Figure 21) starts by defining an inductive type `FormatString` to construct format strings. In our case the format string is either `End` or a type appended to the format string, i.e. a list of types. Given a list of types the function `computeType` gives us the type of a function that expects all the argument in the list and eventually returns the type `Void`. Finally, the function `printF` is a dependently typed function that given a format string `e` has the type `(computeType e)`. The implementation of `printF` makes use of dependent pattern matching on the format string. If the end of the format string has been reached it will simply return `Null`. In the other case it will create a new function that expects the head for the format string and recursively creates a new function based on the rest of the format string.

```
Inductive FormatString : Type :=
  | End   : FormatString
  | Cons : Set -> FormatString -> FormatString;

def computeType(e:FormatString) : Set  {
  <(Πf:FormatString.Set)>
  match e with  {
     End   => Void ;
     Cons  => (λhf:Set.
               (λrf:FormatString.
                  hf -> (computeType rf)))
  }
};

def printF( e:FormatString ) : (computeType e) {
  <(Πf:FormatString.(computeType f))>
  match e with  {
     End => Null;
     Cons => (λhf:Set.
               (λrf:FormatString.
                 (λx:hf.
                  (printF rf))))
  }
};
```

Figure 21: Implementation of printF.

11 Conclusion

It is our hope that this hands-on-approach towards the implementation of a small dependently typed programming language is a good addition to the more formal expositions of dependent typing. We gave a detailed explanation of the most important concepts of dependent typing and have illustrated with examples how these concepts can be used. We have avoided concerns such as efficiency and error handling to focus on the essence of the language. With this introduction to WebPie the reader can obtain a good understanding of how inductive datatypes, dependent case matching and guarded recursive functions work in dependently typed programming languages and can start reasoning about extending the language.

References

[1] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017, doi:10.1145/3110275.

[2] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, page 230–266, Berlin, Heidelberg, 2008. Springer-Verlag, doi:10.1145/1481861.1481862.

[3] coq team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. Available at `https://flint.cs.yale.edu/cs430/coq/pdf/Reference-Manual.pdf`.

[4] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983, doi:10.1007/978-3-642-81955-1_11.

[5] Arnon Avron, Furio Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *J. Autom. Reason.*, 9(3):309–354, December 1992, doi:10.1007/BF00245294.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993, doi:10.1145/138027.138060.

[7] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013, doi:10.1017/S095679681300018X.

[8] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[9] Graham Hutton. *Programming in Haskell*. Cambridge University Press, USA, 2nd edition, 2016.

[10] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[11] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[12] N.G. Bruijn, de. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972, doi:10.1016/1385-7258(72)90034-0.

[13] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '94, page 39–59, Berlin, Heidelberg, 1994. Springer-Verlag, doi:10.1007/3-540-60579-7_3.

[14] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018. Available at `https://softwarefoundations.cis.upenn.edu/lf-current/index.html`.

A WebLF Rules for Normalisation

$$\boxed{e \Downarrow n}$$

$$\frac{e_f \Downarrow \lambda x : e_t . e_b \qquad e_a \Downarrow e_{a'} \qquad [x \mapsto e_{a'}]e_b \Downarrow n}{e_f\ e_a \Downarrow n}\ (\text{N-}\textsc{App-Lam})$$

$$\frac{e_f \Downarrow e'_f \qquad e_a \Downarrow e_{a'}}{e_f\ e_a \Downarrow e'_f\ e'_a}\ (\text{N-}\textsc{App-e}) \qquad \frac{e_t \Downarrow n_t \qquad e_b \Downarrow n_b}{\lambda x : e_t . e_b \Downarrow \lambda x : n_t . n_b}\ (\text{N-}\textsc{Abs})$$

$$\frac{e_t \Downarrow n_t \qquad e_b \Downarrow n_b}{\Pi x : e_t . e_b \Downarrow \Pi x : n_t . n_b}\ (\text{N-}\textsc{Pi})$$

Figure 22: WebLF normalisation rules.

B Using Webpie

In this appendix we showcase some simple theorem proving and programming examples in Webpie. Concretely we show how to encode the first examples of the software foundations book volume 1 [14].

B.1 Data and Functions

In the example below we define a new inductive Set, the name of the new set is `day` and it has seven constructors, one for each day of the week.

```
Inductive day : Set :=
  | monday     : day
  | tuesday    : day
  | wednesday  : day
  | thursday   : day
  | friday     : day
  | saturday   : day
  | sunday     : day;
```

Figure 23: Inductive definition of day.

Once an inductive set has been defined we can write functions which take elements of `day` as arguments or return elements of the set `day`. For example, we can define a function which given a day returns the next weekday.

In the definition below we indicate that we will define a new function, the name of this function is `next_weekday` it takes one argument `d` which needs to be an element of day and we return an element of type `day`.

In order to inspect which particular day the function was called with, we make use of case matching. When matching over an inductively defined set the programmer needs to specify the type over which will be case matched and what the return type of each of the cases such be. In our case we match over a day and return a new day, in the code this is indicated by the annotation `<λx:day.day>`

```
def next_weekday(d:day) : day {
  <λx:day.day>
  match d with {
   monday    => tuesday;
   tuesday   => wednesday;
   wednesday => thursday;
   thursday  => friday;
   friday    => monday;
   saturday  => monday;
   sunday    => monday
  }
};
```

Figure 24: A function over the day type.

Next to inductively defined sets we can also define propositions. While it is tempting to think about propositions as things which are true this is certainly not the case. There are many propositions that can be stated but much fewer which can actually be proven to be true.

We start with a well known proposition namely equality. What does it mean for two things to be the same? One way of defining that two things are same is to say that two things are the same if they are exactly the same thing. Each object is the same as itself. To capture this idea we define an inductively defined relation `eq` which has only one constructor `eq_refl`. In order to avoid having to define equality propositions for each and every type we define we can make the equality proposition parametric in the type over which we define the equality. The type annotation `ΠT:Set.T->T->Set` indicates that the equality type expects a type `T` and two values of type T.

```
Inductive eq: ΠT:Set.T->T->Set :=
 | eq_refl : ΠT:Set.Πx:T.(eq T x x);
```

Figure 25: Parametric Equality Proposition.

B.2 Proof by normalisation

Armed with our equality proposition we can now start testing whether our definition of `next_weekday` is doing what we suspect it is doing. For example, we can now write a proof to show `(eq day (next_weekday monday)tuesday)`. Our hopes are that when we evaluate `(next_weekday monday)` it will give us `tuesday`. So in order to prove `(eq day (next_weekday monday)tuesday)`, we need to construct a proof `(eq day tuesday tuesday)`. Constructing such a proof is not that hard, we use the only available constructor for eq namely `eq_refl`, the first argument it expects is a set, and as we are trying to construct a proof over days we supply it with the type `day` as the first argument. Subsequently, we need to supply it with an element of day which is in our case is `tuesday`. The resulting term has type `(eq day tuesday tuesday)`. The typechecker will normalise `(next_weekday monday)` to `tuesday` during type checking and can conclude that what we wanted to prove is indeed correct.

```
def test_next_weekday(x : Void) : (eq (next_weekday monday) tuesday) {
  (eq_refl tuesday)
}
```

Figure 26: Testing next weekday.

B.3 Proof by rewriting

We have now shown how to use normalisation to prove some easy propositions. Unfortunately, we need some stronger proof techniques when we want to prove some more interesting propositions. As an example, imagine that we know that some number n is equal to some other number m. Knowing that they are equal we should be able to prove that $n+n=m+m$.

In theorem provers such as coq the programmer has the ability to simply rewrite the goals in case she has a proof that two terms are equal. In that case if the programmer has a proof that $n=m$ she can rewrite $n+n=m+m$ to $n+n=n+n$ and then use reflexivity.

If we want to be able to use such a rewriting technique in WebPie we first need to prove rewriting. We define a new function `rewrite` which given two elements `x,y` of type `T`, a proof `p` that the two elements are equal and a proof of a proposition `(P x)` gives us a proof of `(P y)`. Note that `P` itself is a function which returns a proposition when given a value with type T.

The actual proof for rewrite is the body of the function. In order to fully understand how this proof works it is recommended to keep the T-MATCH rule at hand. In essence the case match type indicates that when case matching over a proof `(eq T p q)` it will return a proof `(P q)`. The case match type `ΠT:Set .Πp:T.Πq:T.(eq T p q)-> (P q)` looks a bit more complicated because the equality type is parametric. To account for this the dependent case match needs to make these parameters of the equality type explicit, namely a set `T`, and two elements of that set. As the equality type only has one constructor (`eq_refl`) there is only one case to consider. This constructor expects two arguments, a type `T` and an element of that type. In the `eq_refl` case we can conveniently return `H` which has type `(P x)`.

The attentive reader might be confused at this point because the proof `H` is clearly not a proof of `(P y)`. The expected type of the `eq_refl` case is however constructed with the S helper function: `ΠT:Set.Πx:T.((λT:Set.λp:T.λq:T.(eq T p q)-> (P q))T x x (eq T x x))` which after normalisation reduces to: `ΠT:Set.Πx:T.(P x)`. Hence the proof `(P x)` is exactly what we needed in this case. Finally, because this is the only way we can construct a eq type we are finished.

```
def rewrite(T:Set,P:(T->Prop),x:T,y:T,p:(eq T x y),H:(P x)) : (P y) {
  <(λT:Set.(λp:T.(λq:T.(λeq:(eq T p q). (P q)))))>
  match p with {
   eq_refl => (λT:Set.(λx:T.H))
  }
};
```

Figure 27: Proving rewrite.

Now that we have a proof of rewrite we can use it to prove our proposition that when two numbers are equal to each other adding those numbers together also result in equal numbers as shown in Figure 28.

```
def plus_id_example(n:nat,m:nat,h1:(eq nat n m)) :
(eq nat (plus n n) (plus m m)) {
(rewrite nat
         (λx:(nat). (eq nat (plus n n) (plus x x)))
         n m h1
         (eq_refl nat  (plus n n)))
};
```

Figure 28: Using the rewrite function

C Proofs over natural numbers

```
Inductive Nat : Set :=
  | Zero : Nat
  | Succ : Nat  -> Nat;
Inductive Plus : Nat -> Nat -> Nat -> Set :=
  | Plus_Zero  : Πn:Nat.(Plus Zero n n)
  | Plus_Succ  : Πx:Nat.Πy:Nat.Πs:Nat.Πp:(Plus x y s).
                          (Plus (Succ x)  y (Succ s));

Inductive Eq : Nat -> Nat -> (Type 1) :=
 | Eq_Rfl  : Πn:Nat.(Eq n n)
 | Eq_Succ : Πx:Nat.Πy:Nat.(Eq x y) -> (Eq  (Succ x) (Succ y));

def nat_ind
( P  : Nat -> Prop,
  f  : (P Zero),
  fn : (Πn:Nat.(P n) -> (P (Succ n))),
   n : Nat
) :  (P n)
{
   <(λn:Nat.(P n))>
   match n  with {
      Zero  => f;
      Succ  => (λn:Nat. (fn n (nat_ind P f fn n)))
   }
};

def add(x:Nat,y:Nat) : Nat {
  <(λn:Nat.Nat)>
  match x with {
     Zero  => y;
     (Succ z)  => (λz:Nat. (Succ (add z y)))
  }
};

def add_zero(n:Nat) : (Eq (add Zero n) n) {
   (Eq_Rfl n)
};
```

```

def add_x_zero(x:Nat): (Eq (add x Zero) x) {
  ((nat_ind
   (λn:Nat.(Eq (add n Zero) n))
   (Eq_Rfl Zero)
   (λn:Nat.(λIH:(Eq (add n Zero) n).
      (Eq_Succ (add n Zero) n IH))))
   x)
};

def plus_x_zero(x:Nat) : (Plus x Zero x) {
  (nat_ind
         (λn:Nat.(Plus n Zero n))
   (Plus_Zero Zero)
   (λy:Nat.(λIH:(Plus y Zero y). (Plus_Succ y Zero y IH)))
   x)
}
```

Using Large Language Models for (De-)Formalization and Natural Argumentation Exercises for Beginner's Students

Merlin Carl
EUF
Flensburg, Germany
Institut für Mathematik
Europa-Universität Flensburg
Flensburg, Germany
merlin.carl@uni-flensburg.de

We describe two systems currently being developed that use large language models for the automatized correction of (i) exercises in translating back and forth between natural language and the languages of propositional logic and first-order predicate logic and (ii) exercises in writing simple arguments in natural language in non-mathematical scenarios.

1 Using Large Language Models for Autoformalization

Autoformalization, i.e., the automated translation from natural language to formal logic, is a natural language processing task that has been adressed in a number of ways; natural language proof checking systems such as Naproche (see, e.g., Cramer, [9]) and SAD (Verchinine et al., [15]) make use of grammar-based approaches, with natural-language parsers producing intermediate formats between natural language and formal logic. Recently, the interest in machine learning approaches has increased, including the use of neural networks (Azerbayev et al., [2], Wang et al., [14]) and large language models (Wu et al., [16]); autoformalization for mathematics at the undergraduate level was explored in Azerbayev et al. [2]. Two strong large language model that are publically available are OpenAI's GPT-3.5, with the associated text completion model text-davinci-003[1], and the more recent GPT-4-Turbo.[2] One advantage of pretrained models over training neural networks is that frequently, a few examples are sufficient to obtain a stable and usable model. It has been observed in [2] that GPT can successfully be used for autoformalization. Not surprisingly, the performance improves drastically when one merely demands autoformalization of expressions in a controlled natural language (CNL) rather than arbitrary sentences in natural (mathematical) language, which can be used, e.g., for automated proof checking in teaching contexts, see [7].

For the applications we describe in this article, however, we are interested not in translating mathematical sentences into formal logic, but rather statements in natural language on everyday matters. To this end, we wrote prompts for text-davinci-003 consisting of different numbers of examples, depending on the complexity of the task:

- 30 examples for the translation of natural language sentences into formulas in propositional logic with a given notation.

---

[1] See `https://platform.openai.com/docs/models/gpt-3-5`.

[2] See `https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo`.

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 28–43, doi:10.4204/EPTCS.400.3

© M. Carl
This work is licensed under the
Creative Commons Attribution License.

- 44 examples for the translation of natural language sentences into formulas in propositional logic, combined with a classification determining whether that sentences formulates a claim or an assumption.
- 56 examples for the translation of natural language sentences into formulas in first-order predicate logic.

Typical example prompts for the three tasks looked like this (original German; translated to English for the convenience of the reader):

- notation:{S:Fritz takes a boat;F:Fritz takes a plane;A:Fritz arrives in America;K:Fritz tries to swim}Fritz arrives in America if and only if he takes a boat or a plane, but not if he tries to swim.[3]#$((S \vee F) \leftrightarrow A) \wedge (K \rightarrow \neg A)$§
- notation:{W:This is supposed to be a joke;L:This is supposed to be funny;N:This is new}If this is supposed to be a joke, it is neither funny nor new.♯$[claim$[4]$, [W, \rightarrow, [neg, [L, or, N]]]]$§
- notation:{B(x,y):x is the brother of y;S(x,y):x is the sister of y}The sister of someone's brother is that someone's sister.♯$\forall x : \forall z : (\exists y : (B(x,y) \wedge S(y,z)) \rightarrow S(x,z))$§

As one can see, these examples are structured as follows: the first part of the form "notation:{...}" introduces a number of propositional letters, predicate letters and constant symbols, along with their intended semantics; this is followed by a sentence in natural language, a sharp serving as a separation symbol, a formalization of the natural language sentence in the given notation and a stop symbol. In each case, a few examples were added where either the natual language sentence was not a sentence at all, but rather some nonsense string, or could not be expressed in the given notation.

Requests to the model prompted in this way can then be made by expanding the prompt by a string of the form "notation:{...}ϕ♯", where ϕ is a natural language sentence. Experiments showed a satisfying performance on the intended kind of (simple) example sentences. An impressive feature was that formalizations worked well even when the precise formulation did not use the expressions given in the prompt. Thus, in the notation "R: It rains, S: There is a storm; P-the party will be cancelled", the sentence "If it pours or there is a strong wind, there will be no feast" was (correctly) formalized as $(R \vee S) \rightarrow P$.

The main drawbacks were the following:

- The model showed a tendency to report sentences with strange, wrong or absurd content, such as "If the moon is made of green cheese, then there is a giraffe on the moon" are as erroneous, even though they could easily be formalized within the given notation. Adding an explanation of the specific meaning of "error" in this context and several further examples did not solve this issue. Consequently, such examples, which are somewhat typical for logic classes, should at the moment be avoided when using the model.
- The model showed a certain tendency to use all pieces of the given notation in its formalization, so that, e.g., "Barking dogs don't bark"[5] was formalized as $\forall x((D(x) \wedge B(x)) \rightarrow S(x))$ in the notation where $D(x)$ stood for "x is a dog", $B(x)$ for "x barks" and $S(x)$ for "x bites". This was considerably improved by adding several examples with superfluous notation.

---

[3] Note that the system is to be used by students in Germany.

[4] Translated for the convenience of the reader; in the actual prompt, we used "beh" for "Behauptung", which represents claims in the internal Diproche format, see [5]. For assumptions, it would have been "vss" for "Voraussetzung".

[5] Note that this is not a typo. In order to evaluate how true the model formalizes the given sentence, rather than some more common substitute for it, we deliberately used this (absurd) sentence rather than the well-known proverb that barking dogs don't bite.

- The model showed a tendency to "project" expressions onto the given notation; for example, in the notation from the last bullet point, "Barking cats don't bite" was formalized as $\forall x((D(x) \wedge B(x)) \rightarrow S(x))$, even though no predicate letter for "cat" was contained in the notation ("meowing cats don't bite", however, led to an error message). Whether or not this presents a serious issue for the intended application remains to be seen.
- For sentences of high logical complexity (e.g., containing several quantifier alternations or junctors), the formalizations were frequently wrong. This, however, does not present much of an issue for the intended application.
- The disadvantages of using cloud-based LLMs include (i) their lack of stability (continued training can lead to a much worse performance, rendering working applications unusable), (ii) their lack of reliable availability (models may stop being available altogether, and temporarily become unavailable due to server capacity issues) and (iii) pricing. For these reasons, we expect our system to remain in an experimental state until workable local alternatives become available.

By now, text-davinci-003, which was state of the art when our system was developed early in 2023, is considered "legacy" by OpenAI and its use deprecated. Thus, additionally, we tried the same task with an AI-"assistant" based on the more recent model GPT-4-Turbo. Here, we achieved a surprisingly good performance using merely a prompt explaining the notation to be used (i.e., without offering any initial example cases); in contrast to the above examples, however, the notation – i.e., the abbreviations for the atomic propositions, predicate and constant symbols to be used in the formalization – was fixed and the same for all the examples. The precise prompts, along with the example sentences and the obtained output for 50 instances of first-order-formalization and 57 instances of formalization in propositional logic, can be found in the appendix. As can be seen from the examples (54)-(57) for propositional logic, "nonsense" sentences concerning strongly counterfactual scenarious did no longer pose an issue for GPT-4-Turbo. Concerning logically absurd sentences, such as (43), (44), (47), (48), one of these (44) was still (wrongly) labeled as "not expressable",[6] while the other two were processed correctly. In formalization in quantifier logic, all "absurd" examples, namely (24), (26), (33), were processed correctly. While a certain flexibility of expression was retained – for example, the sentence "In terms of size, Fritz surpasses himself" was formalized correctly given a notation that contained a two-place predicate for "larger than" – the issue of replacing non-expressible terms by terms in the vocabulary did no longer show up: Thus, the propositional examples (20), (21), (27), (57) were correctly identified as "not expressable".

For the task of formalization in propositional logic, we also evaluated several large language models that are available locally, i.e., they can be run on the user's local machine rather than remotely. The performance of general local LLMs and also of LLMs focusing on mathematics – such as WizardMath-70B[7] – turned out to be poor; the only reasonable results were obtained with models trained on the task of code-writing, in particular WizardCoder-34B.[8]

For propositional logic, GPT-4-Turbo formalized 55 of the 57 sample sentences correctly, or about 96.5 percent. The best performance of a local large language model for the same set of sentences was successful for 40 sentences, corresponding to a success rate of about 69 percent. As can be seen in lines (54)-(57), highly counterfactual content – which was a serious issue for davinci-003 – was consistently

---

[6] When asked for the reason why (44) would not be expressable, the model replied that "the statement is self-contradictory and cannot be expressed meaningfully using the given notation or any standard logical operators because it violates the law of non-contradiction. Thus, the proper response is "not expressable".", thus explicitly defending the (wrong) claim that logically inconsistent statements cannot be expressed in the formalism of propositional logic.

[7] See `https://huggingface.co/TheBloke/WizardMath-70B-V1.0-GGML`.

[8] See `https://huggingface.co/WizardLM/WizardCoder-Python-34B-V1.0`.

processed correctly by both models. Logically absurd statements, such as (43), (44), (48) were falsely labeled as “not expressible” once by both models; here, GPT-4 generated a lengthy explanation claiming that contradictory statements cannot be expressed in formal logic. Tautologous statements, such as (47) were unproblematic, as were factually wrong statements for both models. Both models could successfully handle a certain freedom of expression, as is demonstrated by the sentences (17), (18), (19), (31), (32), (34). Both models still had an issue with intricate logical structure, such as triple negation (38); as such examples are likely to arise rarely in the training data, this is to be expected. While GPT-4-Turbo clearly outperforms the local model, both models show usable results for the intended application, namely simple exercises of limited logical complexity.

For the task of autoformalizing natural language sentences in first-order logic, no local LLM showed a satisfying performance. The best results were obtained with GPT-4-Turbo, where 46 out of 50 provided example sentences were formalized correctly, i.e., 92 percent. The unexpressable sentences (18), (32), (36) were correctly identified. In cases where the content was counterfactual or absurd, the model stated this fact in a text comment accompanying the formalization, but still provided a correct formalization (see (24), (26), (27), (33)). For some of those sentences that were not formalized as expected, the model provided explanations: In the case of (13) (“If Hector barks, he is not a real dog”), the model explained that the notation did not allow for expressing “is not a real dog”; given common ways of using “real XYZ”, this is arguably correct; a similar point can be made for (46). Without these examples, the success rate would be about 96 percent. An actual mistake is (34), which was not expressible, as no 2-place-predicate “x bites y” was given in the notation; in (35), the formulation “baying canine” was apparently too “off” in order to be identified as “barking dog”. However, such formulations are not to be expected from someone seriously attempting to solve a deformalization exercise, so that the practical relevance of this mistake for the intended application appear limited.

We point out that the system is currently in an experimental stage. In particular, several practical challenges in terms of pricing and stability of the underlying LLMs will need to be overcome before the system can be actually employed (and tested) in teaching.

2 Automatized Correction of (De-)Formalization Exercises

Beginner students frequently have difficulties both with expressing statements in the language of logic and with interpreting statements written in the language of formal logic. Even after becoming acquainted to the meaning of the logical symbols, extracting the meaning remains a challenge: Frequently, a sentence such as $\exists x : \text{child}(x) \wedge \text{swims}(x)$ will be read out as “There is x such that x is a child and x swims”, remaining close to the surface structure of the formula, rather than something like “Some child swims”.

Indeed, the relation between “natural” ways of thinking of something and the way it is expressed in formal logic is quite intricate; one notorious example of this being the way of expressing dynamics via quantifier changes.[9]

Formalization and deformalization exercises are meant to practice this crucial skill in university education in mathematics. In a formalization exercise – also known as “math dictation”, see, e.g., [4][10] – a sentence in natural language is given, together with some formal vocabulary, and the student’s task is to produce a logical formula expressing this sentence. Thus, a typical formalization exercise could look

---

[9] For example, in the definition of continuity, the idea of “moving one point closer to another” is a dynamical conception expressed via quantifier changes; for a detailed discussion, see the considerations of the concept of continuity in Lakoff and Nunez [12], p. 309-315.

[10] We originally learned this term from Michael Junk.

like this:

- Let S stand for the statement “The sun shines”, W for the statement “I go out for a walk”. Formalize the statement “I won’t go for a walk unless the sun shines” in the language of propositional logic.
- Let $C(x)$ stand for “x is a child”, $S(x)$ for “x swims”. Formalize the statement “Some children swim” in the language of first-order predicate logic.

It is not too hard to provide automated feedback for such formalization exercises: One can, for example, store the exercise in the form of a pair (η,ϕ) consisting of the natural language statement η to be displayed to the user and a correct formalization ϕ of it, take the user’s input, determine whether it is a well-formed formula at all (and, if not, provide according feedback), and, in case it is, pass the tasks $\phi \rightarrow \eta$ and $\eta \rightarrow \phi$ to an automated theorem prover, such as the PyProver, which can be imported into Python code as a package. Systems that work roughly in this way include the “mathematical logic tutor” of Moreno et al. [13] or the formalization exercises in Edukera [1].

We remark here[11] that the concept of formalization is rather intricate. Indeed, a formalization in the sense above is a special case of a translation, and so the difficult question about the adequacy of translations also applies to formalizations: When is a formula ϕ an adequate formalization of a natural language sentence S? A naive approach might be to say that ϕ and S must be in some sense “provably equivalent”; this, however, leads to several difficulties: First, one would have to make precise the informal notion of proof required to make sense of this criterion. Second, such a notion would necessarily be relative to the set of accepted background assumptions, which are usually left implicit and may be highly context-dependent. For example, when asked to formalize “If a, b are the lengths of the catheti of a right triangle and c is the length of its hypotenuse, then $c^3 > a^3 + b^3$”, it should be acceptable – if theory-laden – to respond with $\forall a,b,c \in \mathbb{R}^+((c > a+b \wedge a^2+b^2 = c^2) \rightarrow c^3 > a^3+b^3)$, thus presupposing Pythagoras’ theorem; but if the sentence would be to formalize “If a, b are the lengths of the catheti of a right triangle and c is the length of its hypotenuse, then $a^2 + b^2 = c^2$”, it would be most inadequate to formalize this in the same way, thus leading to a tautology. (One might even ask whether it can be formalized adequately in the language of real closed fields at all without circularity.) Third, mere provable equivalence is a poor criterion for the adequacy of a formalization; for example, formalizing either of the preceding statements as $1 = 1$ should certainly not be counted as correct.[12] Other great examples of formalization, such as Gödel’s arithmetization of the concept of first-order provability require a creative mastering of some background theory. This kind of formalizing is a task resembling programming more than translation, and it is not what the system sketched in this paper is meant to teach. The difference between “translation-like formalization” and “programming-like formalization”, although didactically quite obvious, is not easy to pin down logically. However, by considering everyday rather than mathematical contexts, this issue is mostly avoided: There is not much background theory that could be used in expressing a sentence such as “Barking dogs don’t bite” into a first-order language with a vocabulary for “barks”, “bites” and “is a dog”, let alone for interpreting the respective first-order formula in natural language. We will thus accept a formula ϕ as a correct formalization of a sentence S with a fixed given formalization ψ if and only if the equivalence $\phi \leftrightarrow \psi$ is a tautology in the respective logic (that is, propositional logic or first-order logic), i.e., using the empty background theory. This seems to yield a didactically acceptable notion of adequacy, as long as the sentences under consideration are neither tautological nor contradictory – although “weird” formalizations accepted by a system working with this premise are still possible (e.g. by writing something like $A \vee (A \wedge A)$ or $\neg\neg\neg\neg A$ rather than A, or by

---

[11] We thank one of our referees for pointing out that this point warrants a discussion.

[12] Note that the former statement is correct, as $a^3 + b^3 < a^2c + b^2c = c^2 \cdot c = c^3$.

forming a conjunction with a number of unrelated tautologies), they are unlikely to actually be proposed by students, except by those who have already mastered the subject at hand anyway.

Deformalization exercises, on the other hand, are a far more delicate matter. In a deformalization exercise, a student is given a formal vocabulary together with a logical formula using that vocabulary, and is asked to express it in natural language in the simplest possible terms. Thus, typical deformalization exercises look as follows:

- Let S stand for the statement "The sun shines", W for the statement "I go out for a walk". Express the statement $W \rightarrow S$ in natural language.
- Let $C(x)$ stand for "x is a child", $S(x)$ for "x swims". Express the statement $\exists x : C(x) \wedge S(x)$ in natural language.

The automatized correction of such exercises is challenging in at least to ways: First, one needs to automatically translate the user's natural language input into the appropriate formal language, using the specified vocabulary. Second, one needs to grade the "naturalness" of the user's input, so that expressions such as "There is x such that x is a child and x swims" mentioned above receive a different feedback than "Some children swim".

In our system, which is currently in an experimental stage but will eventually be integrated into the Diproche system ([8], [5], [7], [4], [6]), exercises are stored as triples (n, η, ϕ), where n is the available vocabulary, η is a natural language sentence and ϕ is the formalization thereof, which should be "optimal" in terms of naturalness. This kind of representation has the advantage that the same exercise can both be used as a formalization exercise (displaying n and η, then asking for ϕ) and as a deformalization exercise (displaying n and ϕ, then asking for η).

2.1 Checking Deformalizations

In this section, we will explain the architecture of the checking routine for deformalization exercises. As explained above, users are given a set of notations and a formula using these notations. They can then enter an arbitrary string in a text window in a web interface. This input string is passed on to the formalization routine, which uses the input and the training examples to generate a request for a large language model (such as text-davinci-003) and passes it on to this LLM; if this yields the output "error", an error message is displayed to the user and no further processing takes place; otherwise, the formalization of the users input and the formula given in the problem statement are passed on to the PyProver, which determines whether the implications between them are provable (i.e., propositional or first-order tautologies). More precisely, if ψ is the formal expression fixed in the problem statement and ϕ is the result of autoformalizing the user's input, the PyProver is asked to prove both $\phi \rightarrow \psi$ and $\psi \rightarrow \phi$. The result of this is then passed on to the feedback creation, which will report which of these implications could be verified.

However, a merely logically correct answer does not imply that the input string is a good deformalization of the given formula, or that the user has understood this formula. Typically, beginners will translate a formula such as $\forall x((D(x) \wedge B(x)) \rightarrow \neg S(x))$ – in the vocabulary given above – "word for word" into something like "For all x, if x is a dog and x barks, then x does not bite". Thus, in order to provide a meaningful assessment of the input, the mere logical correctness needs to be supplemented with an evaluation of the "naturalness" or "simplicity" of the input string. This is the task of the "Grader" module.

Our attempts to train language models to provide such an assessment were utterly unsuccessful so far: Even with a considerable number of examples, the model appeared to be unable to distinguish unnaturally formulated sentences from perfectly naturally formulated sentences with an unusual content. We thus resorted to a rather simple-minded solution, which, however, turned out to work quite well for our purpose: We measure the complexity of the input by relating its length to the length of the template solution entered as part of the problem statement (although not displayed to the user) and normalize the result. A "word for word translation" of logical syntax into natural language will usually be considerably longer than the shortest formulation in natural language, so that this can be expected to approximate the degree of naturalness satisfyingly well. More precisely, if $|s|$ denotes the length of a string, η denotes the template solution in the problem statement and ϕ is the user's input, the degree of simplicity is given by

$$10\sigma(10(\frac{|\eta|}{|\phi|}-0.7))$$

where σ is the usual sigmoid function, given by $\sigma(x) := \frac{1}{1+e^{-x}}$. This yields a measure of the input's simplicity on a scale from 0 (not natural at all) to 10 (very natural); feedback is provided depending on whether this value is less than or equal to 5, strictly between 5 and 8 or equal to or above 8. Thus, a good solution, which will have $|\eta| \approx |\phi|$, will have $\sigma(10(\frac{|\eta|}{|\phi|}-0.7))$ close to (but below) 1, while a overcomplicated solution with $|\phi|$ much larger than η will lead to a value close to 0. A score of at least 5 is awarded if the template solution has a length that is 70 percent (or less) of the length of the input string.

Finally, the feedback creation either reports that the input is logically incorrect (not equivalent to the formula to be expressed) and, in this case, whether it is necessary, but not sufficient, sufficient, but not necessary, or neither; in this case, no evaluation of the simplicity of the input is given. If, on the other hand, the input was logically correct, then the user is provided with feedback reporting this, accompanied by the system's evaluation of its simplicity, asking the user, if necessary, to try her hand at further simplifications of her expression.

There are at least two sources of potential mistakes in this approach: First, the LLM may provide a formalization not faithful to the user's input, either by an actual formalization mistake or by interpreting the natural language semantics in a subtly different way, and second, the ATP may fail to verify the equivalence even if the input is correct. Due to the decidability of propositional logic (and the fact that the expressions arising in the given contexts will use less than 10 propositional variables and be of surveyable length, so that no resource issues can occur), the latter kind of mistake can only occur for first-order logic. Since the formulas coming up in this context will be logically rather simple – typically restricted to a single quantifier change – this is not likely to be a frequent issue; however, a substantial evaluation of this point will have to wait until the system can be actually employed with student users.[13] Potentially, it cannot lead to wrong solutions being reported as correct, but it could lead to correct solutions being reported as incorrect. In order not to confuse users, the feedback should thus be formulated as a "failure to verify" rather than as a claim that the solution is wrong. The user may then attempt to reformulate her solution to make it easier processable. Another option would be to switch to an ATP that can generate countermodels to non-verifable statements and report these back to the user. The former type of mistake can lead both to correct solutions being reported as incorrect and to incorrect solutions being reported as correct. The latter issue may in particular come up for sentences that deviate from, but strongly resemble, very common natural language sentences that are likely to occur in the training data. Also, issues are

---

[13] One way to exclude such difficulties altogether would be to restrict exercises to decidable fragments of first-order logic, such as monadic first-order logic (see, e.g., [3]), formulas with at most two variables (see, e.g., [11]) etc.; these classes already lead to a rich supply of exercises. If this path is taken, one needs to ensure that the user's input will also fall into the relevant class.

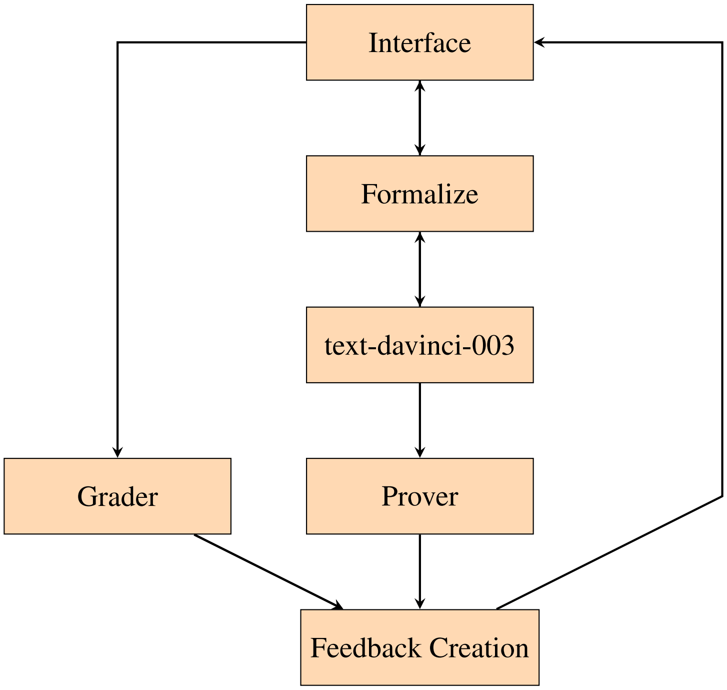


Figure 1: Flowchart for the correction routine for deformalization exercises. (Generated with the help of GPT-3.5.)

likely to occur when students write ungrammatical inputs. Again, an evaluation of the seriousness of such issues will have to wait until the system can actually be employed. To counter such difficulties, the formalization obtained by the system could be reported back to the user to check whether it faithfully captures what she intended to express. However, since the system is intended to teach basic skills in translating back and forth between formal logic and natural language, it is not clear that the intended users will be able to determine whether the formalization was done correctly.

3 Natural Language Argumentation

Besides formalization, learning how to prove is another challenge for beginner students. The Diproche system ([8], [5]) is designed to provide automated feedback on the several aspects – including logical correctness – for solutions to simple beginner proving exercises written in a controlled natural language (CNL). Recently, large language models have been integrated in the Diproche architecture to provide a more “liberal” CNL than the original parser-based one (see [7]).

Diproche is focused on mathematical argumentation in areas such as propositional logic, Boolean set theory or elementary number theory. As a preliminary exercise, it may also be helpful to practice logical argumentation in non-mathematical contexts, putting aside the difficulties students may have with the mathematical content and symbolism. Thus, a natural argumentation exercise might look like this:

- Suppose that the following is true: If the sun shines, Hans goes for a walk. When Hans goes for a walk, he takes his dog with him. When Hans takes his dog for a walk, the dog barks at the cat on the neighbour’s roof. When the dog barks at the cat on the roof, the cat runs away. However, the cat still sits on the roof. Show that the sun does not shine.

Using the same prompts discussed in the last section, such statements can automatically be translated into propositional logic[14]; the resulting formal representation can then be passed on to the checking components of the Diproche system, which will provide feedback on the logical correctness of the argument. Thus, an accepted solution to the above exercise could look like this:

- The cat still sits on the roof. Hence the dog did not bark. Consequently, Hans did not take his dog for a walk. So Hans did not go for a walk. Thus the sun does not shine.

For mathematical exercises, the Diproche system gives feedback on linguistical correctness, logical correctness, type mistakes (using variables without introducing them before, or in a wrong way, such as adding two propositions), success in achieving proof obligations and supposed logical or algebraic fallacies. For the "natural language argumentation" exercises, the feedback on type mistakes will be suppressed, since this should not be an issue for the intended kind of argumentation.

4 Further Work

Clearly, a system for didactical uses should be employed in teaching and evaluated in terms of usability and effectiveness. We plan to do so in the near future. On the technical side, the autoformalization with text-davinci-003 or GPT-4-Turbo, although impressive in many respects, still leaves some things to be desired.[15] We plan to gather a substantial amount of training data and use it to fine-tune pretrained language models. Whether this will lead to improved performance remains to be seen. Concerning the *formalization* exercises ("math dictations"), the LLM technology could be used for automatically deformalizing the user's formal input and reporting it back to her in the case the formalization is not correct, thus making it clearer where the mistake lies; here, more traditional NLP techniques, such as the "pretty printer" described in [10], may also be useful.[16]

Meanwhile, we believe that good use could be made of the "natural argumentation" framework in other subjects, in particular in philosophy: Here, it is a basic type of exercise to reconstruct plain text arguments in a semi-formal style where assumptions and consequences are explicitly labeled, and the approach discussed here for the verification of natural language argumentation could then be used to verify whether an argument written in this way is in fact logically cogent.

5 Acknowledgements

We thank our three anonymous referees for several comments that helped in improving the presentation of the paper, along with constructive criticism concerning its content.

References

[1] *Edukera Homepage*. `https://www.edukera.com/`.

[2] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir R. Radev & Jeremy Avigad (2023): *ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics*. ArXiv, doi:10.48550/arXiv.2302.12433. arXiv:arXiv:2302.12433v1.

---

[14] The same can, of course, easily be done with first-order logic.

[15] Recall in particular the issues with relying on remote services rather than local models.

[16] We thank one of our anonymous referees for pointing out this reference to us.

[3] Heinrich Behmann (1922): *Beiträge zur Algebra der Logik, insbesondere zum Entscheidungsproblem. Mathematische Annalen* 86, pp. 163–229, doi:10.1007/BF01457985.

[4] Merlin Carl (2020): *Automatized Evaluatoin of Formalization Exercises in Mathematics*, doi:10.48550/arXiv.2303.17513. ArXiv:2006.01800v2.

[5] Merlin Carl (2020): *Number Theory and Axiomatic Geometry in the Diproche System. Electronic Proceedings in Theoretical Computer Science* 328, pp. 56–78, doi:10.4204/EPTCS.328.4.

[6] Merlin Carl (2020): *Using Automated Theorem Provers for Mistake Diagnosis in the Didactics of Mathematics*, doi:10.48550/arXiv.2002.05083. ArXiv:2002.05083v1.

[7] Merlin Carl (2023): *Improving the Diproche CNL through Autoformalization via Large Language Models*. arXiv:arXiv:2303.17513.

[8] Merlin Carl & Regula Krapf (2020): *Diproche - ein automatisierter Tutor für den Einstieg ins Beweisen*. In: *Digitale Kompetenzen und Curriculare Konsequenzen*, pp. 43–56.

[9] Marcos Cramer (2013): *Proof-checking mathematical texts in controlled natural language*. Ph.D. thesis, Rheinische Friedrich-Wilhelms-Universität Bonn.

[10] Salwa Tabet Gonzalez, Stéphane Graham-Lengrand, Julien Narboux & Natarajan Shankar (2021): *Semantic parsing of geometry statements using supervised machine learning on synthetic data*. In Jasmin Blanchette, James H. Davenport, Peter Koepke, Michael Kohlhase, Andrea Kohlhase, Adam Naumowicz, Dennis Müller, Yasmine Sharoda & Claudio Sacerdoti Coen, editors: *Joint Proceedings of the FMM, FVPS, MathUI,NatFoM, and OpenMath Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2021 co-located with the 14th Conference on Intelligent Computer Mathematics (CICM 2021), Virtual Event, Timisoara, Romania, July 26 - 31, 2021, CEUR Workshop Proceedings* 3377, CEUR-WS.org. Available at `https://ceur-ws.org/Vol-3377/natfom5.pdf`.

[11] Leon Henkin (1967): *Logical Systems Containing Only a Finite Number of Symbols*. Presses de l'Universite de Montreal, Montreal,.

[12] George Lakoff & Rafel E. Núñez (2001): *Where mathematics comes from : how the embodied mind brings mathematics into being*. Basic Books.

[13] A. Moreno & N. Budesca (2000): *Mathematical Logic Tutor-Propositional Calculus*. In: *First International Congress on Tools for Teaching Logic*, pp. 99–106.

[14] Wang Qingxiang, Chad Brown, Cezary Kaliszyk & Josef Urban (2019): *Exploration of Neural Machine Translation in Autoformalization of Mathematics in Mizar*. In: *CPP 2020: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 85–98, doi:10.1145/3372885.3373827.

[15] Konstantin Verchinine, Alexander V. Lyaletski & Andrei Paskevich (2007): *System for Automated Deduction (SAD): A Tool for Proof Verification*. In: *Automated Deduction – CADE-21. CADE 2007.*, 4603, Springer, Berlin, Heidelberg., doi:10.1007/978-3-540-73595-3_29. Available at `https://api.semanticscholar.org/CorpusID:6915907`.

[16] Yuhuai Wu, Albert Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik & Christian Szegedy (2022): *Autoformalization with Large Language Models*. In: *36th Conference on Neural Information Processing Systems (NeurIPS)*, doi:10.48550/arXiv.2205.12615.

6 Appendix

In this appendix, we give the detailed results of testing the performance of GPT-4-Turbo and WizardCoder-34B in autoformalization of natural language sentence in propositional logic and of GPT-4-Turbo in first-order logic. Although text-davinci-003 was used in the version of the system described above, it is now considered legacy and its use is deprecated; moreover, its performance in autoformalization tasks appears to have considerably dropped since spring 2023; therefore, no results for text-davinci-003 are listed. We first give the prompt given to the LLM, followed by a table listing the input sentences, the model's output and an evaluation of correctness. It should be noted that the sentences were tested in the order in which they are listed in the table and that other orders may lead to different results.

6.1 Propositional Logic

For propositional logic, we used the following prompt; in WizardCoder-34B, temperature was set to 0, repitition penalty to 1 (the minimum). Below, $\veebar$ is used for exclusive disjunction. In the case of missing brackets, formulas were read according to the usual priority rules. In (31) and (56), a missing closing bracket was added in the WizardCoder output, which was still counted as correct, as this can easily be done automatically.

> Express the sentence as a formula in propositional logic, using the given notation.
> Notation:
>
> - S:"The sun shines"
> - R:"It rains"
> - P:"There will be a party"
> - M:"It is Monday"
> - A:"Andreas will attend the party"
> - B:"Barbara will attend the party"
> - C:"The moon is made of green cheese"
> - G:"There is a giraffe on the moon"

For the GPT-4-Turbo assistant, the following was added:

> If the given sentence cannot be expressed with the given notation, return "not expressable".

Nr	Input	GPT-4	+/-	WizardCoder-34B	+/-
1	It does not rain and the sun shines.	$\neg R \wedge S$	+	example	
2	It's neither sunny nor rainy.	$\neg S \wedge \neg R$	+	$\neg(S \vee R)$	+
3	If there is a party on Monday, Andreas will attend it.	$(P \wedge M) \rightarrow A$	+	$M \rightarrow A$	-

4	If it rains on Monday, Andreas will stay away from the party.	$(R \wedge M) \to \neg A$	+	$(M \wedge R) \to \neg A)$	+
5	Andreas and Barbara will only attend the party together.	$(A \leftrightarrow B)$	+	$(A \leftrightarrow B)$	+
6	There will be a party if the sun shines on Monday.	$(S \wedge M) \to P$	+	$(S \to P)$	-
7	It is not true that Andreas will abstain from the party when it rains.	$\neg(R \to \neg A)$	+	$\neg((R \to \neg A) \wedge (M \wedge R))$	-
8	On rainy Mondays, Barbara does not go to parties.	$(R \wedge M) \to \neg B$	+	$((M \wedge R) \to \neg B)$	+
9	It is a sunny, rainy Monday.	$S \wedge R \wedge M$	+	$(S \wedge R \wedge M)$	+
10	On sunny Mondays, it rains.	$(S \wedge M) \to R$	+	$(M \wedge S) \to R)$	+
11	Andreas and Barbara attend the party.	$A \wedge B$	+	$(A \wedge B)$	+
12	At least one of Andreas and Barbara attends the party.	$A \vee B$	+	$(A \vee B)$	+
13	Exactly one of Andreas and Barbara attends the party.	$(A \wedge \neg B) \vee (\neg A \wedge B)$	+	$(A \veebar B)$	+
14	Andreas parties provided Barbara doesn't.	$\neg B \to A$	+	$(\to \neg A) \to P$	-
15	Andreas goes to the party provided Barbara doesn't.	$\neg B \to A$	+	$(B \to \neg A)$	+
16	Barbara and Andreas only visit Monday parties.	$(B \vee A) \to M$	+	$(M \to (A \vee B)) \wedge (\neg A \to M) \wedge (\neg B \to M)$	-
17	On Monday, there is good or bad weather.	$M \to (S \vee R)$	+	$(M \to (S \vee R))$	+
18	When it is wet, Andreas does not go to the party, but Barbara does.	$R \to (\neg A \wedge B)$	+	$((R \to \neg B) \wedge B)$	-
19	It is a bright Monday, and Andreas and Barbara jointly visit the celebration.	$(S \wedge M) \wedge (A \wedge B)$	+	$(M \wedge S \wedge (A \vee B))$	-
20	On Tuesday, there is a thunderstorm.	not expressable	+	not expressable	+
21	Christa exercises on Tuesday.	not expressable	+	not expressable	+

22	It's a bright Monday.	$S \wedge M$	+	$(M \wedge S)$	+
23	It rains if and only if the sun shines.	$R \leftrightarrow S$	+	$(S \to R) \wedge (R \to S)$	+
24	Sunshine and rain do not occur together	$\neg(S \wedge R)$	+	not expressible	-
25	Monday is no day for parties.	$M \to \neg P$	+	$\neg M$	-
26	Andreas is not a party person.	$\neg A$ / not expressable	+	$\neg A$	+
27	Andreas is not a Tyrannosaurus.	not expressable	+	not expressable	+
28	Andreas stays away from the party.	$\neg A$	+	$\neg A$	+
29	Andreas does not stay away from the party.	A	+	A	+
30	If Andreas stays away from the party, then so does Barbara.	$\neg A \to \neg B$	+	$(\neg A \to \neg B)$	+
31	On sunny days, Barbara and Andreas celebrate.	$S \to (A \wedge B)$	+	$(S \to (A \wedge B)$	+
32	Barbara feasts in the rain, but Andreas doesn't.	$R \to (B \wedge \neg A)$	+	$(R \to (B \wedge \neg A))$	+
33	It's not Monday, but Barbara still parties.	$\neg M \wedge B$	+	$\neg M$	-
34	In spite of the downpour, Andreas celebrates.	$R \wedge A$	+	$(R \to A)$	-
35	It's sun or rain, but not both.	$(S \vee R) \wedge \neg(S \wedge R)$	+	$(S \veebar R)$	+
36	On sunny and also on rainy days, no parties take place.	$(S \to \neg P) \wedge (R \to \neg P)$	+	not expressible	-
37	On sunny and also on rainy days, there are no parties.	$(S \to \neg P) \wedge (R \to \neg P)$	+	$((S \veebar R) \to \neg P)$	-
38	It is wrong that the party has not been canceled.	$\neg\neg P$ / P	-	$\neg\neg P$	-
39	Mondays are rainy.	$M \to R$	+	$M \wedge R$	-
40	Rainy days are Andreas' party days.	$R \to A$	+	$(R \to A)$	+
41	It is a lie that Andreas and Barbara attended the party together.	$\neg(A \wedge B)$	+	$\neg(A \wedge B)$	+
42	Barbara parties on rainy Mondays.	$(R \wedge M) \to B$	+	$(R \wedge M) \to B$	+
43	It's Monday and it's not.	$M \wedge \neg M$	+	$(M \wedge \neg M)$	+

44	Andreas goes to the party if he doesn't.	not expressable	-	$(\neg A \rightarrow A)$	+
45	Barbara goes to the party if and only if Andreas stays at home.	$B \leftrightarrow \neg A$	+	$(A \rightarrow B)$	-
46	On party Mondays, the sun shines and it rains.	$(P \wedge M) \rightarrow (S \wedge R)$	+	$(M \rightarrow (S \wedge R))$	-
47	It's Monday if and only if it's Monday.	$M \leftrightarrow M$	+	$(M \leftrightarrow M)$	+
48	It is neither Monday nor not Monday.	$\neg M \wedge \neg\neg M$ / $\neg M \wedge M$	+	not expressible	-
49	There is no party if Andreas isn't there.	$\neg A \rightarrow \neg P$	+	$(\neg A \rightarrow \neg P)$	+
50	Neither Andreas nor Barbara attend the party.	$\neg A \wedge \neg B$	+	$(\neg A \vee \neg B)$	+
51	One of Andreas and Barbara does not attend the party.	$(\neg A \vee \neg B)$	+	$(\neg A \vee \neg B)$	+
52	If it rains, then Andreas does not attend Monday parties.	$R \rightarrow (M \rightarrow \neg A)$	+	$(R \rightarrow (M \wedge \neg A))$	+
53	Given that it rains, the sun shines.	$R \rightarrow S$	+	$(R \rightarrow S)$	+
54	If the moon is made of green cheese, then there is a giraffe on it.	$C \rightarrow G$	+	$(C \rightarrow G)$	+
55	There are no moon giraffes.	$\neg G$	+	$\neg G$	+
56	On Mondays, the moon is made of green cheddar and populated by giraffes.	$M \rightarrow (C \wedge G)$	+	$(M \rightarrow (C \wedge G))$	+
57	There is an elephant on the moon.	not expressable	+	not expressible	+
	mistakes (total)	2		17	

6.2 First-Order Logic

For first-order logic, WizardCoder-34B showed no satisfying performance; for GPT-4-Turbo, we used the following prompt:

Express the given sentence as a formula in first-order logic, using the following notation:

- $D(x)$:x is a dog
- $B(b)$:b barks
- $S(a)$:a bites
- $L(a,b)$:a is larger than b
- fr:Fritz
- he:Hector

If the given sentence cannot be expressed with the given notation, return "not expressible". No comment or explanation; only return the formula.

Nr	Input	Output GPT-4	+/-?
1	Fritz is a dog.	$D(fr)$	+
2	Hector is taller than Fritz.	$L(he,fr)$	+
3	If Hector barks, then he bites.	$B(he) \rightarrow S(he)$	+
4	There is a dog.	$\exists x D(x)$	+
5	Every dog barks.	$\forall x(D(x) \rightarrow B(x))$	+
6	Some dogs bite.	$\exists x(D(x) \wedge S(x))$	+
7	Hector does not bite.	$\neg S(he)$	+
8	Barking dogs bite.	$\forall x((D(x) \wedge B(x)) \rightarrow S(x))$	+
9	Barking dogs don't bite.	$\forall x((D(x) \wedge B(x)) \rightarrow \neg S(x))$	+
10	All dogs bite.	$\forall x(D(x) \rightarrow S(x))$	+
11	Everything bites.	$\forall x S(x)$	+
12	Barking dogs don't bark.	$\forall x((D(x) \wedge B(x)) \rightarrow \neg B(x))$	+
13	If Hector barks, he is not a real dog.	not expressible	-
14	If Hector barks, he is not a dog.	$B(he) \rightarrow \neg D(he)$	+
15	Fritz and Hector are barking dogs.	$(D(fr) \wedge B(fr)) \wedge (D(he) \wedge B(he))$	+
16	If a dog barks, then Fritz barks.	$\forall x((D(x) \wedge B(x)) \rightarrow B(fr))$	+
17	No dog is bigger than Fritz.	$\forall x(D(x) \rightarrow \neg L(x,fr))$	+
18	There is a god.	not expressible.	+
19	Fritz is the tallest dog.	$D(fr) \wedge \neg \exists x(D(x) \wedge L(x,fr))$	+
20	No dog that is larger than Fritz will bite.	$\forall x(D(x) \wedge L(x,fr) \rightarrow \neg S(x))$	+
21	Fritz is smaller than Hector.	$L(he,fr)$	+
22	One of Hector and Fritz barks.	$B(he) \vee B(fr)$	+
23	Some dogs are larger than both Fritz and Hector.	$\exists x(D(x) \wedge L(x,fr) \wedge L(x,he))$	+
24	In terms of size, Fritz surpasses himself.	$L(fr,fr)$	+
25	Some dogs are larger than others.	$\exists x \exists y(D(x) \wedge D(y) \wedge L(x,y))$	+
26	Each dog is larger than each dog.	$\forall x \forall y(D(x) \wedge D(y) \rightarrow L(x,y))$	+
27	Fritz is a dog and he isn't.	$D(fr) \wedge \neg D(fr)$	+
28	Fritz is the only dog.	$D(fr) \wedge \forall x(D(x) \rightarrow x = fr)$	+
29	Every dog is larger than every other dog.	$\forall x \forall y((D(x) \wedge D(y) \wedge x \neq y) \rightarrow L(x,y))$	+

30	Exactly one of Hector and Fritz barks.	$B(he) \land \neg B(fr) \lor \neg B(he) \land B(fr)$	+
31	Hector is larger than any other dog, except Fritz.	$\forall x((D(x) \land x \neq fr \land x \neq he) \rightarrow L(he,x))$	+
32	Meowing cats don't scratch.	not expressible	+
33	Biting dogs are not dogs and they don't bite.	$\forall x((D(x) \land S(x)) \rightarrow (\neg D(x) \land \neg S(x)))$	+
34	Hector bites Fritz.	$S(he)$	-
35	Hector is a baying canine.	not expressible	-
36	Hugo is a biting dog.	not expressible	+
37	If it does not bark, it's not a dog.	$\forall x(\neg B(x) \rightarrow \neg D(x))$	+
38	It's not a dog if it does not bark.	$\forall x(\neg B(x) \rightarrow \neg D(x)$	+
39	There is more than one dog.	$\exists x \exists y(D(x) \land D(y) \land x \neq y)$	+
40	Hector barks, so Fritz does the same.	$B(he) \rightarrow B(fr)$	+
41	Fritz barks, and so does Hector.	$B(fr) \land B(he)$	+
42	Barking dogs and also biting dogs are smaller than Fritz.	$\forall x((D(x) \land (B(x) \lor S(x))) \rightarrow L(fr,x))$	+
43	Of each two dogs, one is larger.	$\forall x \forall y((D(x) \land D(y) \land x \neq y) \rightarrow (L(x,y) \lor L(y,x)))$	+
44	No two dogs are the same.	$\forall x \forall y((D(x) \land D(y) \land x \neq y) \rightarrow \neg(x = y))$	+
45	All dogs are the same: they bite!	$\forall x(D(x) \rightarrow S(x))$	+
46	All dogs are the same.	not expressible	-
47	Hector is a dog that neither barks nor bites.	$D(he) \land \neg B(he) \land \neg S(he)$	+
48	Hector and Fritz both bark, but only one of them bites.	$(B(he) \land B(fr)) \land ((S(he) \land \neg S(fr)) \lor (\neg S(he) \land S(fr)))$	+
49	All dogs bite, but only Fritz and Hector bark.	$\forall x(D(x) \rightarrow S(x)) \land \forall x(D(x) \land B(x) \rightarrow (x = fr \lor x = he))$	+
50	Hector is not the only barker.	$\exists x(B(x) \land x \neq he)$	+

Improving the Diproche CNL through Autoformalization via Large Language Models

Merlin Carl
EUF
Flensburg, Germany
Institut für Mathematik
Europa-Universität Flensburg
Flensburg, Germany
`merlin.carl@uni-flensburg.de`

The Diproche system is an automated proof checker for texts written in a controlled fragment of German, designed for didactical applications in classes introducing students to proofs for the first time. The first version of the system used a controlled natural language for which a Prolog formalization routine was written. In this paper, we explore the possibility of prompting large language models for autoformalization in the context of Diproche, with encouraging first results.

1 Introduction

The Diproche ("Didactical Proof Checking")[1] system is an automated proof checker for proofs in a controlled natural language (CNL) specifically adapted to elementary proving exercises in "introduction to proof" classes in first-year university education. Users can enter a proof in a controlled fragment of German and receive immediate feedback on logical correctness, type correctness, fulfillment of proof goals etc.; see [3], [4] for an introduction to the system and [5] for a report on the experiences with the first version of the system.

In spite of the impressive progress recently made with machine translation and the possibility of using this for the task of autoformalization (see, e.g., [11]), the Diproche CNL was implemented using classical techniques from computational linguistics, more specifically via a definite clause grammar written in Prolog. This was used to convert the natural language input into an internal list format, from which the proof obligations at each proof step are generated, to be passed on to an automated theorem provers (ATPs) in which those steps that should be accepted in the context of a specific exercise are hard-coded. The reason for this choice was to make the system as transparent as possible to the user: There should be no surprises concerning how the program interprets certain formulations, or which formulations it accepts. However, in practice, it quickly became apparent that this goal is in conflict with the other, similarly important, goal of providing a convenient CNL sufficiently rich for expressing proof texts in a way that resembles natural mathematical texts well enough to spare users the burden of learning a formalism on top of the difficulties they face when learning how to prove. Since the intended users are mathematical beginners with little to no experience with formal deduction, formal languages or proof calculi, it turned out that transparency is lost rather quickly. This leaves little reason to not take advantage of the possibilities of natural language processing based on machine learning techniques, such as pretrained language models. An obvious reason in favor of this approach is that developing, refining and changing such a

---

[1] Both the system's architecture and its name are inspired by the Naproche system, of which it is a kind of didactical "offspring", see `https://naproche-net.github.io/0`.

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 44–58, doi:10.4204/EPTCS.400.4

© M. Carl
This work is licensed under the
Creative Commons Attribution License.

prompt-based CNL is much easier and quicker than (re-)writing a formal grammar. In particular, making the model work in other natural languages is as easy as translating the natural language sentences in the prompt, no matter how (grammatically) different the new language is from the current one.[2]

In this paper, we begin to explore the possibilities of using the language model DaVinci-3 developed by OpenAI[3] for various tasks in didactical systems that aim at teaching how to prove, in particular transforming natural language input into a formal representation that can be further processed by formal proof checkers. For this purpose, we built a prototype in which a Python-based preprocessing routine using DaVinci-3 was combined with the logical components of Diproche written in Prolog. We also consider a very recent version of GPT-4, which shows an even stronger autoformalization performance.

At first, the experiences reported in Avigad et al. [1] with using large language models for autoformalization appear to be discouraging for this plan: Only about 11 percent of the natural language inputs were formalized correctly ([1], p. 3). Much better results were reported in [12], where more than 25 percent of the natural language inputs (which were problems for math competitions) were translated correctly into Isabelle (ibid., p. 1). Still, for reliably checking even a simple natural language argument consisting of typically more than 10 sentences with sufficient reliability to be of didactical use to beginner's students, anything considerably below a hundred percent is not good enough.

However, one needs to keep in mind that the approach in the works just mentioned is explicitly not to use a CNL, but to formalize sentences from everyday mathematical discourse. In contrast, our aim is by far more modest: Retaining the restriction to a small fragment of natural mathematical language, we want to use language models to (i) simplify the process of designing and converting to such languages and (ii) allow for more freedom in the specific choice of formulations compared with a CNL given by a formal syntax. The classical approach to translating between a CNL and a formal representation would be to implement a formal grammar for it, which is a quite cumbersome task. In contrast, our experience shows that, with certain qualifications, the Diproche CNL could be learned and improved with merely 71 lines of example formalization by text-davinci-003. Even more impressingly, if one merely requires translation to a more standard first-order format that is easily convertible to the internal representation format used by Diproche, the same effect could be achieved for GPT-4-Turbo with a prompt merely containing the relevant to notation, but no examples.[4]

The new system architecture, adapted from the one given in [3], is as follows, where Python components are marked with "Py" and Prolog components are marked with "Pr", while "LLM" denotes the large language model used for the autoformalization:[5]

2 Autoformalization and Proof-Checking

A naive approach to using autoformalization in automated proof-checking is the following: Each natural language sentence corresponds to some formula in first-order logic. By translating each sentence separately, one obtains a formalization of the whole text, which can then be given to an automated theorem prover for verification. This view, however, ignores a great deal of well-known features of natural language mathematics:[6]

---

[2] Not even this may be necessary: Prompted entirely with German sentences, our model was able to correctly process several (simple) sentences written in English, French and Chinese.

[3] See `https://openai.com/product`.

[4] Some qualifications apply here; see below.

[5] An experimental system has been implemented using text-davinci-003. The integration of GPT-4-Turbo is currently being developed.

[6] Cf., e.g., [8], p. 171; for a detailed treatments of the specifics of the language of mathematics, see Ganesalingam [9].

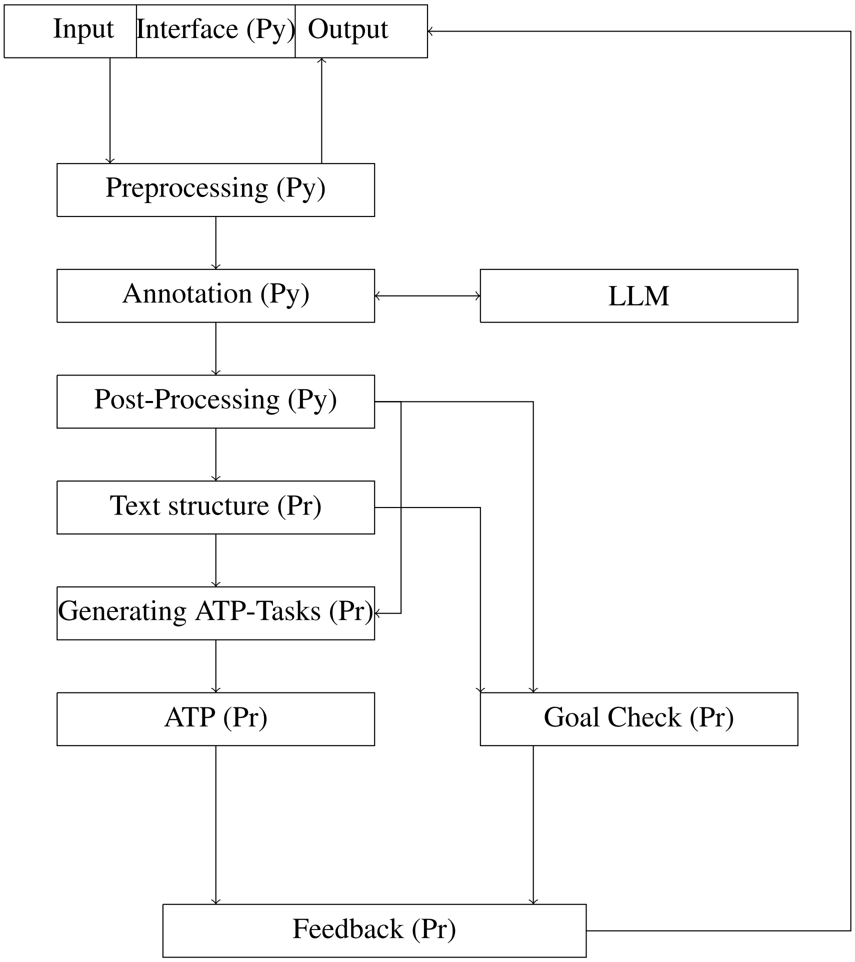


Figure 1: Flowchart of Diproche with integrated LLM

1. Sentences have different functions, such as goal announcements, deductions, assumptions, annotations. For a (logical) check of the text, these need to be identified, along with the content.
2. Sentences may have content that is not immediately expressible in first order logic. Consider "Since $a = b$, it follows that $a^2 = b^2$". The claim here is apparently that, at this point of the argument, it is established that $a = b$ and that, from this, $a^2 = b^2$ can be deduced. It would certainly be wrong to formalize this as $(a = b) \to (a^2 = b^2)$, since then, only the implication would be established, while it is its conclusion that the sentence is claiming. However, omitting the "Since $a = b$" would considerably distort the sentence's meaning: We do not want a sentence like "Since grass is green, it follows that $a^2 = b^2$" to be marked as correct. Another example would be a sentence containing multiple conclusions, such as "Now we have A, so we get $A \vee B$, and thus also $C \to (A \vee B)$". Thus, the formal representation of a sentence will need to use means beyond mere first-order logic.
3. The whole text has a structure. It may contain subproofs, variables and assumptions are introduced for certain parts of the argument and gone in others.[7] This overall structure needs to be taken into account when formally representing proofs.
4. Elliptic sentences that gain their meaning from context: "We show that there are infinitely many primes. Suppose otherwise."; "Thus, there is a line passing through P and Q. Call it l."; "Hence, n has at least one prime divisor. Pick one, and call it p."; "So we have $x \in A$. Consequently, it is even."[8]

Thus, a naive "sentence by sentence"-formalization is not enough as a basis for automated proof-checking. Along with a formalization of content, one needs to identify the function of the sentence, which is a task for automated text classification, one needs a formalism capable of representing figures such as justifications that do not appear in first-order logic (and the autoformalization routine needs to translate to this formalism), and one needs some kind of structural markers to identify the scope of declarations and assumptions, and, when formalizing a sentence, it must be possible to take into account earlier sentences as context.

3 Prompting and Training Large Language Models

The pre-trained language models offered by OpenAI can be adapted to a specific task in two different ways: Prompting and fine-tuning. While prompting can simply mean writing a description of the task at hand (such as "Write a birthday card for the person named in the input"), in the case of autoformalization, it is best done by offering a carefully chosen series of examples.[9] Prompting leaves the model internally unchanged; intuitively, one could regard it as writing a (long) question. Fine-tuning, in contrast, means actually modifying (training) the language model with an appropriate data set. This option is currently only available for language models considerably weaker than text-davinci-003; in order to achieve satisfying result, a considerably larger amount of examples is required.[10] Since such data is

---

[7] See, e.g., Cramer [7], p. 255.

[8] Cf, e.g., [10], p. 7-8.

[9] This approach is also taken in Avigad et al. [1].

[10] According to OpenAI `https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset` , one should "aim for at least ∼100 examples per class" for the classification task alone of associating sentences with their logical function, which would in our case amount to 700 sentences; the actual formalization task being by far more complex, one would likely need several thousand example sentences for reasonable results.

currently unavailable for Diproche, and also somewhat cumbersome to generate, we will only consider prompting in this paper, which seems to offer a quick and easy way to achieve an impressive level of autoformalization sufficient for basic didactical applications.[11]

3.1 The Diproche CNL and the Internal List Format

The Diproche CNL is explained in some detail in [3]. Here, we recall some basic features. The Diproche CNL is a fragment of natural mathematical German, comprising typical ways to express assumptions ("Suppose that x is even"), claims ("Hence, $x+1$ is odd"), variable declarations ("Let k be an integer"), goal announcements ("We will show that x is a square") and annotations ("Proof:", "qed", "Case 1:" etc.); further sub-types include declarations combined with assumptions, such as "Let k be an integer such that $n = 2k$" (which are existentially loaded and are in need of verification) and justified claims ("Since $x = 3(a+b)$, x is a multiple of 3"). The sentences[12] written in this CNL are converted into an internal list format whose crucial ingredients are a list of the of variables occuring in the sentence, its type (assumption, declaration, claim, annotation,...) and its actual content (which can be empty, as in the case of annotations). Thus, the sentence "Therefore, x is even" would be translated as [[x],beh,[even,[x]]].

When processing formulations such as "Suppose not" or resolving anaphors, prior sentences need to be taken into account as the context in which a certain sentences is to be translated. A typical line of our prompt looks like this:[13]

context:{We show that the intersection of A and B is a subset of the union of A and B.} Suppose not. #
[[A,B],ang,[not,[[A,cap,B],subseteq,[A,cup,B]]]]§

Here, the part "context:{}" contains the relevant context, the next part ("Suppose not") is the sentence to be translated, # serves to separate the natural language sentence from its formalization, then we have the formalization in the internal list format and finally § as the stop symbol, which prevents the language model from generating further text, such as more examples of natural language sentences and their formalizations in the same spirit. The examples also included ungrammatical and formally invalid sentences, for which the translation was "invalid" ("ungültig"). If the formalization routine leads to this result, the process is stopped and the sentence in question is reported to the user as not processable.

3.2 First Experiences

The first experiences with prompting DaVinci-3 for autoformalization in Diproche were encouraging: After only a few examples, the model had "grasped" the extraction of variables along with the classification and even offered (usually sensible) completions in the "spirit" of the given examples although no specific example for the case at hand was given; for example, after learning that "x is even" was to be formalized as [even,[x]], it drew the obvious analogy for "x is odd" or "x is prime". It correctly dealt with formulations that it had not seen in the examples – for example, after having seen that $x \in X$ was to be turned into [x,in,X], it correctly processed sentences like "x is an element of X" or "X contains x";[14] similarly, after being given one example of how to formalize an assumption, it correctly identified a variety of ways to formulate assumptions – including some that, such as "Gesetzt, es wäre der Fall,

---

[11] A word of warning is in order here: A prompt length currently cannot go beyond 4000 tokens, which is quite limited.

[12] In the relevant sense here, "sentence" includes annotations.

[13] Translated into English for the convenience of the reader; the actual prompt lines are German.

[14] Although our actual prompts were written in German, we offer English translations here for the sake of the reader. Although it would surprise us if it was otherwise, we therefore cannot guarantee that the results can be reproduced with prompts written in English.

dass" ("Let it be the case that"), were intentionally chosen to be somewhat uncommon. It even had a considerable success rate when, after a series of German prompts, it was given a sentence in English, French, Italian or Chinese; a system once developed in this way can thus be easily made available in other languages as well. Even without prior examples, the model exhibited the ability introduce variable names not given in the text and pick them in a sensible way; thus, for example, "Every natural number has a prime divisor" was formalized as [all,[n,in,nat],[exists,p,[[prime,p],and,[divides,[p,n]]]]]; in particular, in no instance was a variable name used for different variables. In most cases, the model was able to resolve anaphors, taking advantage not merely of grammatical categories, but even of content: For example, for the input "Hence x is an element of A. Thus, it is even. Consequently, it cannot be empty.", the first "it" was formalized as x, while the second was formalized as A. Turning formulas into the internal list format was also "learned" reliably along the way. Likewise, the model could correctly deal with elliptic formulations such as "We will show that $A = B$. Suppose not." of "Hence, there exists k such that $n = 2k$. Pick one.". We also observed that a rather common input format for automated theorem provers, namely THF[15] was already "known" to DaVinci-3; when merely asked to "formalize" various statements without given specific examples, it generated THF formulas.[16] This, however, is not of immediate relevance for our purposes, since Diproche uses its own internal format.

Still, there were issues, most of which, however, could be resolved to our satisfaction:

1. Along with the requested formalization, the model occasionally generated extra example pairs of natural language and formalization of its own. This could be prevented by introducing § as a stop symbol and putting this after each formalization.
2. Even so, the model sometimes added to the natural language input rather than merely formalizing the given expression. This was resolved by separating the natural language expression from the formalization by an # and making # a part of the input for each request.
3. When the input consisted of several sentences, the model frequently misrepresented later sentences, perhaps according to "expectations" of what these should have been rather than what was actually there. Only giving it one sentence per time was not an option, since this would have ruled out using the abilities of the model to refer to context, e.g., in the resolution of anaphors, or in processing such constructions as "We will show that p is prime. Suppose not." (see above). This was solved by explicitly labeling the preceding sentences as "context". The model was prompted with examples of sentences that could not be formalized without further context, in which case it should return the error message "missing context". In order to minimize such unwanted interference, the autoformalization routine works with the minimal amount of context that is required for the sentence at hand: Thus, given a list of sentences $[S_1, S_2, ..., S_n]$, it will, in the i-step, attempt to formalize S_i without invoking context (i.e., using the empty context). If this yields to a "missing context" error, S_i is tried again, this time with context $\{S_{i-1}\}$. If the error persists, the earlier sentences are added one by one; when all earlier sentences up to S_1 were added without success, the formalization attempt is stopped unsuccessfully.
4. Even when the correct formalization was given in several examples, the model occasionally choose to express it differently, i.e., express $A \cup B$ as [A,union,B], rather than [A,cup,B]. Since these cases were of a limited and surveyable number, this problem could be dealt with by a postprocessing routine.

---

[15] See, e.g., `https://www.tptp.org/Seminars/THF/Contents.html` or [2].

[16] A similar "surprise" is reported in [12], p. 1.

5. There were also difficulties to distinguish between implications ("If x is even, then $x+1$ is odd") and justifications ("Now $x+1$ is odd, because x is even"): rewriting the examples and adding the tag "justification" improved the performance, but attained nothing near perfect accuracy. We therefore decided to drop phrases containing justifications from the CNL in the first version.

Moreover, it soon became apparent that converting all types of formulas occuring in any of the subareas currently available in Diproche – in particular, propositional logic, Boolean set theory, axiomatic geometry and elementary number theory – was too much to ask from a model prompted with only 4000 tokens. We thus decided to further specify the task by writing separate prompts for each of these areas; when requested, the system would then use the model for the area to which the current proof text belongs. For the first experiments reported here, we concentrated on writing a prompt for Boolean set theory, an area for which several example Diproche texts are available and which also forms an important part of the beginner's lectures taught in Flensburg.

Besides text-davinci-003, we also had the opportunity to test an "assistant" based on a recent version of GPT-4. An assistant is a chatbot whose behaviour can be controlled by a prompt describing its intended functioning.[17] In order to keep the prompt length limited, we did not insist on a direct translation into the internal Diproche format, but were instead content with a standard first-order format that is easily automatically translatable into the required format. It turned out that, with a prompt explaining in detail the desired output format, the performance of the assistant was satisfying (see below) even without presenting a single example. However, it should be noted that the model's responses still depend on the previous course of the dialogue, so that answers to earlier requests play a role similar to the examples given to text-davinci-003. Thus, while the model was able to autoformalize the given statements with a rather high success rate, this might have been different had the statements been given in a different order. Thus, adding a set of examples representative of the task at hand seems advisable also for this option.

4 Performance on Typical Diproche Texts

To see whether the prompted language model would perform, we tested it against three solutions for set-theoretical exercises written in the Diproche CNL, and also modified versions of these solutions that contained mistakes. Not counting these variants, and only considering sentences that were actually passed on to the language model[18], these texts contained 33 sentences, all of which were processed correctly. Since these texts were typical texts that users of the system would be expected to write, this confirms that the prompted language model could serve as a replacement for the Diproche CNL in a practical setting.

In order to evaluate the performance of the GPT-4-based assistant, we used 50 example sentences from the area of Boolean set theory. The model was then asked to identify the type of the sentence (declaration, assumption, claim, declaration with additional assumption) and to provide a formalization. The results of this experiment can be found in a table in the appendix. In order to make it easier for the reader to evaluate the quality of the formalizations, we worked with English sentences, although the Diproche CNL is a fragment of German. As experiments suggest, the performance on German translations of the given sentences did not substantially deviate.

We also asked several mathematicians to write up clearly structured solutions using short and simple sentences to several sample exercises, but without mentioning any particular syntax. Our goal is to

---

[17] See, e.g., `https://platform.openai.com/docs/assistants/how-it-works/agents`.

[18] Some standard annotations are processed separately and not given to the language model.

evaluate how much of these solution texts will be processed correctly by the model in order to quantify the "naturalness" of the "learned CNL". The results of this will appear in future work.

These results show that, at least in the field of Boolean set theory, the "learned CNL", covers most of what the hard-coded Diproche-CNL for this purpose had to offer (we recall that "justified claims" were excluded in this investigation). On the other hand, it offers a much greater degree of freedom of expression; in particular, natural language variants of simple formal expressions, such as "x is an element of the intersection of A and B" are usually processed correctly, which would be somewhat cumbersome to obtain with a formal syntax.[19] Moreover, it turned out to be rather tolerant with respect to minor misspellings or typos, in contrast to the Diproche CNL, the orthographical strictness of which had apparently been a source of frustration for several students. This is a clear advantage of using large language models over using hand-crafted formalization routines.

To get a clearer picture of the prospects of this approach, it would certainly be desirable to have a systematic statistical evaluation of the system. This, however, would in fact of limited value due to the following reason: Since the models used above are continually modified, the performance measured at one point of time can be vastly different from the performance a few months later, even when evaluated using the exactly same requests.[20] We are planning such an evaluation after the system has been changed to work with local LLMs that can be kept stable over time (see also the next section). Moreover, since the preparation of the first version of this paper (in spring 2023) and the present version (in December 2024), new LLMs have become available that show a strongly improved performance for autoformalization tasks relevant to our purposes.

To give the reader at least some impression of the possibilities, we discuss here briefly the 50 above-mentioned example sentences processed with an OpenAI API-assistant based on the (currently experimental) model GPT-4-Turbo.[21] The precise prompt and the table of results can be found in the appendix. Note that the prompt does not contain any examples. Of these 50 examples, 49 were processed correctly, leading to a success rate of 98 percent. The resolvation of anaphorical expressions worked well in most cases, see, e.g., (13), (19), (23), (24). The exception was example (30) "From this, we get that, if A is not empty, then B is", where the anaphorical expression "then B is" was wrongly interpreted as referring to "not empty", while it would usually be understood as "empty". Phrases such as "exactly one" were correctly interpreted (46). The results also show that a variety of formulations was correctly processed, which considerably goes beyond the Diproche CNL, including in particular term description in natural language (13), (39), (41). Still, there are several points to be noted: First of all, the input format in this case did not take into account context. Due to this, sentences such as "Let x be an element of X" (sentence 11) become ambiguous in their role, being either variable declarations (if x appears for the first time in this sentence) or plain assumptions (if x was introduced earlier on). Similarly, sentence 16 ("Let A be a subset of B") was interpreted as a variable declaration of A (but not of B). While this is indeed a plausible reading, there are of course contexts where it would be wrong. In order to fix this, some kind of context needs to be provided, at least in the form of declared variables available at a certain point in the text. While sentence 6 was correctly formalized, the use of an existential quantifier for formalizing the phrase "non-empty intersection", rather than merely writing $G \cap H \neq \emptyset$, would in many contexts lead to difficulties in the further processing. In sentence (36), "whenever" could be read as an implicit universal quantifier, in which case the given formalization would be missing the quantifiers. However, since the

---

[19] We remark, though, that this did not always work reliably: For example, we observed one case where the model confused "the intersection of the complements of A and B" with "the complement of the intersection of A and B".

[20] See, e.g., [6], which indicates that the performance of GPT-4 and GPT-3.5 considerably *dropped* from March to June 2023 in several areas.

[21] Available under the name gpt-4-1106-preview.

formalization would be processed correctly in the context of Diproche, we regarded this as correct.

5 Conclusion and Further Work

The work reported in this paper indicates that prompting large language models such as DaVinci-3 is apparently a good “quick and dirty” way for developing and improving controlled natural languages for didactical systems such as Diproche which aim at automated proof verification at the beginner level; in particular, such languages tend to be more flexible and error-tolerant than those obtainable by classical methods with reasonable effort. Clearly, this merely scratches the surface of the possibilities that large language models open for such systems, and which will be the subject of further work. Moreover, the purpose of the present paper is merely to present the general concept and argue for its prospects; an accurate evaluation of the didactical advantages of using LLMs over formal grammars will be done once a system version suitable for the actual employment in teaching has been obtained.

On the one hand, the quality and reliability of autoformalization, which is currently mainly hindered by the bound on prompt length, could most likely be considerably improved by “specialization”, i.e., writing prompts for various sub-tasks, classifying the input accordingly and then handing them to the appropriate sub-module.

On the other hand, there is a number of other tasks relevant for such systems, which could conceivably be treated with large language models, including the following:

1. Immediate feedback on the proof text. Our experiments indicate that GPT is currently quite poor and unstable in evaluating the logical coherence of a proof text. However, for other kinds of feedback, such as stylistic remarks, it may be more useful.
2. Generating hints. While GPT is currently not able to reliably solve basic proving exercises, the texts it generates quite often tend to have the right overall structure. This could be used to provide hints for users how to approach a certain problem.
3. Mistake diagnosis. The current Diproche version includes an “Anti-ATP”, a logical (and algebraic) mal-rule library that codifies typical false deduction steps (such as deducing $\neg B$ from $\neg A$ and $A \rightarrow B$) and reports them to the user; the hope is that this can help becoming aware of such mistakes and eventually to avoid them. Using large language models for mistake diagnosis at least for algebraic manipulations may lead to a more “semantical” approach to this task, in contrast to the current one based on formal patterns.[22]

Concerning the practical use in (large) teaching situations, however, the following should be considered:

- Since DaVinci-3 and similar large language models cannot be run locally, each checking requires a considerable amount of web traffic between the user’s device and the servers on which the model is hosted. Compared with the good old-fashioned parsing approach, this takes noticably more time, in particular depending on the internet connection. In our experiments, the time from demanding feedback to receiving feedback typically went up from hardly noticable to at least 30 seconds.
- The servers to which the formalization requests are sent are of limited capacity and, if too many requests are sent in a certain amount of time, will refuse them. This already happened to the

---

[22] For example, both $(x+y)^2 = x^2 + y^2$ and $(x+1)^3 = x^3 + 1$ are currently identified as an instance of a distributive use of exponentiation, but $(x+y+z)^2 = x^2 + y^2 + z^2$ is not; in contrast, DaVinci-3 was able to recognize the third example as an instance of the same mistake, even though only examples with two summands were given to it in the prompt.

author frequently during development. With a considerable amount of students frequently using the system, it can be expected to happen regularly.

- Also, requests aren't free: One call to DaVinci-3 with the full amount of 4000 tokens costs about 8¢.[23] With a class of 250 students who are supposed to use the system regularly for their homework and make frequent intermediate checks when working on the exercises – which is the whole purpose of the system – this can quickly get expensive: Four such exercises with an average of five intermediate checkings would lead to a weekly price of 400$.
- Moreover, this approach has all the disadvantages of relying on external services: The underlying model may be changed, resulting in unexpected behaviour, or discontinued altogether; prices can go up; a certain amount of user data is sent to external servers, which may potentially lead to privacy issues etc.
- In particular, as [6] indicate, the continued modifications to GPT have led to a considerable *decrease* in certain mathematical abilities in three months. Similarly, we observed that certain requests that consistently worked fine at some point of time did so no longer some months later.
- The "learned CNL", although quite accurate most of the time, is still somewhat unreliable and occasionally shows surprising behaviour. Thus, for example, the sentence "Thus, x is an element of A or x is an element of A." was reported by text-davinci-003 as "invalid", while it worked perfectly well when replacing one of the A's by a different letter. This is likely due to the fact that such constructions are extremely uncommon in natural language. Moreover, there were instances where claims in the form conditional constructions were mis-identified as assumptions. In one case, a sentence was processed wrongly after changing the variable names. These difficulties can certainly be overcome by providing more examples, but it should still be kept in mind that, compared with hard-coded parsers, a certain amount of reliability is lost.
- A potential problem that may arise in use is that a "less controlled CNL" may make it more difficult for users to develop a feeling for what will be understood and what not. Whether this is an actual problem will have to be evaluated empirically.

To sum up, experiences indicate that large language models can be fruitfully and easily applied in developing CNLs for proof checkers for didactical applications. However, for practical applications, the aim should be to train models that can be run locally. This would at least solve the first five of the issues mentioned above. To this end, we have experimented with several large language models available on Hugging Face.[24] A general experience so far is that even the largest of the general LLMs perform poorly on autoformalization tasks and, perhaps surprisingly, large models specifically trained with mathematical content, such as WizardLM-70B[25] fare not much better. However, models trained for automated for code generation, such as WizardCoder-Python-34B[26] (apparently the largest local LLM for code generation currently available) show an autoformalization performance comparable to that reported above for text-davinci-003. We plan to systematically develop and evaluate this approach in the near future.

The ideal would be to create a language model specifically trained on large amounts of data for the task of autoformalization. Work towards such models is done, e.g., in [11]. However, until such pretrained models are available, the approach discussed in this paper seems to offer a workable alternative for certain applications.

---

[23] 2¢per 1000 tokens, see `https://openai.com/pricing` (accessed 12.03.2023).

[24] `https://huggingface.co/`

[25] `https://huggingface.co/WizardLM/WizardLM-70B-V1.0`

[26] `https://huggingface.co/WizardLM/WizardCoder-Python-34B-V1.0`

6 Acknowledgements

We thank our three anonymous referees for several comments that helped in improving the presentation of the paper, along with constructive criticism concerning its content.

References

[1] Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir R. Radev & Jeremy Avigad (2023): *ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics*. *ArXiv*, doi:10.48550/arXiv.2302.12433. arXiv:arXiv:2302.12433v1.

[2] Christoph Benzmüller, Florian Rabe & Geoff Sutcliffe (2008): *THF0 – The Core of the TPTP Language for Higher-Order Logic*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Automated Reasoning. IJCAR 2008*, 5195, pp. 491–506, doi:10.1007/978-3-540-71070-7_41.

[3] Merlin Carl (2020): *Number Theory and Axiomatic Geometry in the Diproche System*. *Electronic Proceedings in Theoretical Computer Science* 328, pp. 56–78, doi:10.4204/EPTCS.328.4.

[4] Merlin Carl & Regula Krapf (2020): *Diproche - ein automatisierter Tutor für den Einstieg ins Beweisen*. In: *Digitale Kompetenzen und Curriculare Konsequenzen*, pp. 43–56.

[5] Merlin Carl, Hinrich Lorenzen & MIchael Schmitz (2022): *Natural Language Proof Checking in Introduction to Proof Classes – First Experiences with Diproche*. *Electronic Proceedings in Theoretical Computer Science* 354, pp. 59–70, doi:10.4204/EPTCS.354.5.

[6] Lingjiao Chen, Matei Zaharia & James Y. Zou (2023): *How is ChatGPT's behavior changing over time? ArXiv* abs/2307.09009, doi:10.48550/arXiv.2307.09009. Available at `https://api.semanticscholar.org/CorpusID:259951081`.

[7] Marcos Cramer (2013): *Proof-checking mathematical texts in controlled natural language*. Ph.D. thesis, Rheinische Friedrich-Wilhelms-Universität Bonn.

[8] Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder & Jip Veldman (2009): *The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts*. In: *Controlled Natural Language. CNL 2009*, 5972, pp. 170–186, doi:10.1007/978-3-642-14418-9_11.

[9] Mohan Ganesalingam (2013): *The language of mathematics*. Springer Berlin Heidelberg, doi:10.1007/978-3-642-37012-0.

[10] Peter Koepke & Bernhard Schröder (2003): *ProofML - eine Annotationssprache für natürliche Beweise*. *LDV Forum* 18, pp. 428–441, doi:10.21248/jlcl.18.2003.48. Available at `https://api.semanticscholar.org/CorpusID:30895733`.

[11] Wang Qingxiang, Chad Brown, Cezary Kaliszyk & Josef Urban (2019): *Exploration of Neural Machine Translation in Autoformalization of Mathematics in Mizar*. In: *CPP 2020: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 85–98, doi:10.1145/3372885.3373827.

[12] Yuhuai Wu, Albert Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik & Christian Szegedy (2022): *Autoformalization with Large Language Models*. In: *36th Conference on Neural Information Processing Systems (NeurIPS)*, doi:10.48550/arXiv.2205.12615.

7 Appendix

We give here the precise prompt for the assistant used in the autoformalization experiments with GPT-4-Turbo, along with the table listing the results.

We used the following prompt (typesetting adapted for the sake of the reader):

> Given a sentence, translate it into the format [type,subtype,formalization], where “type” is one of the following: “claim”, “assumption”, “variable declaration”, “goal declaration”.
> “variable declaration” has the subtypes “plain” and “with additional assumption”, for statements in which an assumption is made about the introduced variables beyond the type declaration.
> “formalization” works as follows:
>
> - For a plain claim, it is just a formalization of the statement in first-order logic.
> - For an assumption, it is a formalization in first-order logic of the assumed statement.
> - For a plain variable declaration, it is a list of pairs [var,type], where var is the variable to be declared and type is the type assigned to it.
> - For a variable declaration with additional assumption, it is a pair [declarations,assumption] consisting of the list of declarations as for a plain declaration and a formalization of the assumption. A mere existence claim is not a declaration; thus, "there is an integer x such that ..." is a claim, not a declaration. Declarations are indicated by words such as "Let", "pick", "Choose" etc.
> - If the given sentence contains free variables – that are not quantified over in the sentence – they should remain free in the formalization.
>
> Return only the triple [type,subtype,formalization], nothing else.
> Use $\emptyset$ for the empty set, $c(X)$ for the complement of X, $\cup$ for the union of sets, $\cap$ for the intersection of sets, $=$ for equality, $\subseteq$ for the subset relation, $\in$ for the element relation.

The results obtained with this prompt for 50 example sentences can be found in the below table.

Nr	Input	Output type	Output formalization	+/-
1	Let A be a set.	decl/plain	[[A,set]]	+
2	Let A be a non-empty set.	decl/assmpt	[[[A, set]], $A \neq \emptyset$]	+
3	Let A, B, C, be sets.	decl/plain	[[A, set], [B, set], [C, set]]	+
4	Let A, B, C be sets such that $A \cup B = c(C)$.	decl/assmpt	[[[A, set], [B, set], [C, set]], $A \cup B = c(C)$]]	+
5	Let Q, X and U as well as P be non-empty sets.	decl/assmpt	[[[Q, set], [X, set], [U, set], [P, set]], $(Q \neq \emptyset) \wedge (X \neq \emptyset)$ $\wedge (U \neq \emptyset) \wedge (P \neq \emptyset)$]	+
6	Let G, H be sets with non-empty intersection	decl/assmpt	[[[G, set], [H,set]], $\exists x(x \in G \wedge x \in H)$]	+
7	Let P, Q be disjoint sets.	decl/assmpt	[[[P, set], [Q, set]], $P \cap Q = \emptyset$]	+

8	Let X and Y be sets and let U be their union.	decl/assmpt	[[[X, set], [Y, set], [U, set]], $U = X \cup Y$]	+
9	Let A, B be sets with intersection K.	decl/assmpt	[[[A, set], [B,set], [K, set]], $A \cap B = K$]	+
10	Suppose that $A = B$.	assmpt	$A = B$	+
11	Let x be an element of X.	decl/assmpt	[[x,element],$x \in X$]	+
12	Assume that A and B are disjoint.	assmpt	$A \cap B = \emptyset$	+
13	Suppose that the union of A and B is equal to their intersection.	assmpt	$A \cup B = A \cap B$	+
14	Let us assume that the complement of Z contains u.	assmpt	$u \in c(Z)$	+
15	Suppose that m is not contained in M.	assmpt	$m \in M$	+
16	Let A be a subset of B.	decl/assmpt	[[[A,set]],$A \subseteq B$]]	+
17	Suppose that A is a subset of B.	assmpt	$A \subseteq B$	+
18	Suppose that A is not a subset of B.	assmpt	$\neg(A \subseteq B)$	+
19	Suppose that neither A is a subset of B nor vice versa.	assmpt	$\neg(A \subseteq B) \wedge \neg(B \subseteq A)$	+
20	Let us take it as given that x belongs to A, but not to B.	assmpt	$(x \in A) \wedge (x \notin B)$	+
21	It follows that A and B are equal.	claim	$A = B$	+
22	Consequently, A is not equal to A.	claim	$A \neq A$	+
23	Thus, A is an element of itself.	claim	$A \in A$	+
24	A is an element of itself and it isn't.	claim	$(A \in A) \wedge \neg(A \in A)$	+
25	If A is not empty, then B is a subset of Q.	claim	$(A \neq \emptyset) \rightarrow (B \subseteq Q)$	+
26	Each element of A is also an element of O.	claim	$\forall x(x \in A \rightarrow x \in O)$	+
27	No element of $c(X)$ is an element of $(A \cup B) \cap C$)	claim	$\forall x(x \in c(X) \rightarrow x \notin (A \cup B) \cap C)$	+
28	There is a set A such that $A \cap B = A \cap C$.	claim	$\exists A(A \cap B = A \cap C)$	+
29	There is no set A such that $A \cap B = A \cap C$	claim	$\neg \exists A(A \cap B = A \cap C)$	+
30	From this, we get that, if A is not empty, then B is.	claim	$(A \neq \emptyset) \rightarrow (B \neq \emptyset)$	-
31	From this, we get that, if A is not empty, then B is empty.	claim	$(A \neq \emptyset) \rightarrow (B = \emptyset)$	+
32	By reductio, every subset of A is also a subset of T.	claim	$\forall X(X \subseteq \rightarrow X \subseteq T)$	+

33	For all sets X, Y, Z it holds that: If it holds that Y is a subset of Z, then it also holds that the intersection of X and Y is a subset of the intersection of X and Z.	claim	$\forall X \forall Y \forall Z((Y \subseteq Z) \to (X \cap Y \subseteq X \cap Z))$	+
34	We will now show that A is in fact empty.	goal	$A = \emptyset$	+
35	We need to show that x belongs to K.	goal	$x \in K$	+
36	We need to demonstrate that, whenever $A \cap B = A \cup B$, then $A = B$.	goal	$(A \cap B = A \cup B) \to (A = B)$	+
37	Our goal is to see that x and y belong to Q	goal	$(x \in Q) \land (y \in Q)$	+
38	To see this, first suppose that A is empty while B is not.	assmpt	$(A = \emptyset) \land (B \neq \emptyset)$	+
39	Now it remains to establish the pairwise disjointness of U, V and W.	goal	$(U \cap V = \emptyset) \land (U \cap W = \emptyset)$ $\land (V \cap W = \emptyset)$	+
40	If $x \in A$, then $x \in (A \cup B)$	claim	$\forall x (x \in A \to x \in (A \cup B))$	+
41	As we will presently show, the complement of $A \cup B$ equals the intersection of the complements of A and B.	goal	$c(A \cup B) = c(A) \cap c(B)$	+
42	It is thus excluded that A is a subset of B.	claim	$\neg(A \subseteq B)$	+
43	Therefore, we get that, if A is empty, then B is not.	claim	$(A = \emptyset) \to (B \neq \emptyset)$	+
44	If X and Y are non-empty and disjoint then both are subsets of U.	claim	$(X \neq \emptyset \land Y \neq \emptyset \land X \cap Y = \emptyset) \to$ $(S \subseteq U \land Y \subseteq U)$	+
45	It now follows that at least one of A and B must be empty.	claim	$(A = \emptyset) \lor (B = \emptyset)$	+
46	From this, we get that exactly one of A and B is empty.	claim	$((A = \emptyset) \land (B \neq \emptyset))$ $\lor ((A \neq \emptyset) \land (B = \emptyset))$	+
47	Let A, B, C be sets and additionally pick D to be a non-empty set.	decl/assmpt	[[[A,set], [B,set], [C, set], [D, set]], $D \neq \emptyset$]	+
48	Consider sets A, B, C satisfying $A \cap B = c(C)$	decl/assmpt	[[[A,set], [B, set], [C,set]], $A \cap B = c(C)$]	+

49	If one of A, B is equal to C, then $C \subseteq (A \cup b)$	claim	$((A = C) \vee (B = C))$ $\rightarrow (C \subseteq (A \cup B))$	+
50	Thus, the empty set is disjoint from every set.	claim	$\forall X(\emptyset \cap X = \emptyset)$	+

Teaching Higher-Order Logic Using Isabelle

Simon Tobias Lund Jørgen Villadsen
Technical University of Denmark, Kongens Lyngby, Denmark
sitlu@dtu.dk jovi@dtu.dk

We present a formalization of higher-order logic in the Isabelle proof assistant, building directly on the foundational framework Isabelle/Pure and developed to be as small and readable as possible. It should therefore serve as a good introduction for someone looking into learning about higher-order logic and proof assistants, without having to study the much more complex Isabelle/HOL with heavier automation. To showcase our development and approach we explain a sample proof, describe the axioms and rules of our higher-order logic, and discuss our experience with teaching the subject in a classroom setting.

1 Introduction

Higher-order logic, also known as simple type theory [3], has been described as the combination of functional programming and logic [9], and has proved a very powerful tool for the formalization of mathematics and computer science. It is an expressive enough logic to cover a wide array of fields, while still being built on relatively simple principles, and a number of proof assistants based on higher-order logic are available.

We consider formal reasoning in the generic proof assistant Isabelle [10, 11]. In the present paper we are taking advantage of the genericity of Isabelle, but we also find that Isabelle is at least as user-friendly and intuitive as other proof assistants of comparable power. Although Isabelle is generic and comes with a number of object logics like first-order logic (FOL) and axiomatic set theory (ZF), the default object logic is higher-order logic, called Isabelle/HOL. One of the main aims of the present work is to gently introduce students to Isabelle/HOL and its tutorials [9, 10].

In [15], we gave an Isabelle formalization of intuitionistic and classical propositional logic as a natural deduction system. That formalization is here expanded to higher-order logic, and thereby also first-order logic. In contrast to some other Isabelle formalizations of proof systems [6, 14], which are developed in the complex Isabelle/HOL, we give a theory built directly on the fundamental Isabelle/Pure theory. Our formalization can therefore be viewed as an alternative (though very minimal) logical foundation to Isabelle/HOL. The main purpose of the formalization is as a teaching tool. It is small and transparent enough for someone with a working understanding of logic, but no experience with Isabelle, to study and understand thoroughly, something which is in practice very difficult with Isabelle/HOL.

The rest of the paper is structured as follows. We introduce our approach and describe our contributions in Section 2. We discuss related work in Section 3. In Section 4, we go through sample natural deduction proofs in detail, for propositional logic, first-order logic and higher-order logic, as an introduction to Isabelle code. We present formalizations of first-order logic in Section 5 as a stepping stone towards higher-order logic. In Section 6 and Section 7, we build up intuitionistic and classical higher-order logic, respectively. Finally, in Section 8, we discuss further developments and provide concluding remarks.

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 59–78, doi:10.4204/EPTCS.400.5

© S. T. Lund & J. Villadsen
This work is licensed under the
Creative Commons Attribution License.

2 Approach and Contributions

Higher-order logic is often considered fundamentally different from first-order logic. This point of view is called a dogma in a recent paper [1]:

> *But now there is a new generation of higher-order provers that aim to gracefully generalize their first-order counterparts. They give up the dogma that higher-order logic is fundamentally different from first-order logic. Instead, they regard first-order logic as a fragment of higher-order logic for which we have efficient methods, and they start from that position of strength.*

We find that teaching higher-order logic should be a graceful extension of teaching first-order logic, like the approach for provers [1]:

> *Accordingly, a strong higher-order prover should behave like a first-order prover on first-order problems, perform mostly like a first-order prover on mildly higher-order problems, and scale up to arbitrary higher-order problems.*

In the present paper we often restrict ourselves to the first-order features with respect to axioms, rules and theorems, but always with the power of the higher-order features when needed.

Our main contributions are as follows:

- Our approach provides a gentle introduction to Isabelle/HOL, with many simplifications and no automation, but acknowledging the importance of higher-order logic in automated reasoning in general and in proof assistants in particular, for mathematics and computer science.
- Our approach provides a natural deduction alternative to the foundation of mathematics, which is often based on axiomatic set theory, and the higher-order logic formalizes the axioms of choice, infinity and set comprehension.
- Our approach includes a formalization of implicational axiomatics, with classical implicational logic as a propositional logic fragment, as a common starting point for our teaching, with formal proofs of the soundness and completeness theorems in Isabelle/HOL.
- Our approach includes a succinct formalization of implication and universal quantification using the tools of Isabelle/Pure.
- Our approach has been used, successfully, four times in a computer science course on automated reasoning in the period 2020-2023 for about one hundred students in total.

There are three different “versions” of the Isabelle system we will consider in this paper:

1. Isabelle/Pure
2. Isabelle/HOL, and
3. Isabelle/HOL_Pure.

Isabelle/Pure contains the fundamentals of the Isabelle framework, like proof states, the proof environment, and the meta-logical operators like $\Longrightarrow$ (meta-logical implication) and $\bigwedge$ (meta-logical universal quantification). Isabelle/HOL is the main Isabelle system which contains a wide range of theories for many different fields of mathematics and computer science. Isabelle/HOL_Pure is our minimal formalization of higher-order logic. It contains exactly the same axioms as Isabelle/HOL, but only a minimum of extra definitions and derived rules. Both Isabelle/HOL and Isabelle/HOL_Pure import Isabelle/Pure.

We consider first-order logic systems FOL_Natural_Deduction and FOL_Implicational_Axiomatics, where the first is based on natural deduction while the other is axiomatic. Both systems are formalized as deep embeddings (that is, we define the formulas as an object-level datatype) in Isabelle/HOL, where we have proved them sound and complete. The simple higher order logic is HOL_Pure where we also have some example proofs and derived rules. Because we are axiomatizing this logic right on top of the Isabelle framework we cannot prove meta-properties like soundness or completeness of this system, like we could with the deep embeddings. If one tried to show soundness and completeness of the higher-order logic one would have to be able to define and reason about the semantics of the formulas, which is not possible because the structure of formulas is not accessible at the object level. The idea is to use both of these approaches to teach the students about Isabelle/HOL. The first-order logic systems are broad examples of somewhat complicated formalization efforts built on the full Isabelle system. They showcase patterns and techniques the students can later use in their own formalizations. The goal of the minimal HOL_Pure system is somewhat different. Here we aim to give the students a very thorough understanding of the most fundamental concepts in Isabelle, as opposed to a shallow understanding of a much larger portion.

These three formalizations are contained in the following Isabelle files:

FOL_Implicational_Axiomatics.thy A sound and complete deep embedding of an axiomatic proof system for first-order logic. The file contains 829 lines and is based on the entries Implicational_Logic and FOL_Axiomatic in Isabelle's Archive of Formal Proofs.

FOL_Natural_Deduction.thy A sound and complete deep embedding of a natural deduction proof system for first-order logic. The file contains 1386 lines and is based on the entry Synthetic_Completeness in Isabelle's Archive of Formal Proofs.

HOL_Pure.thy A minimal formalization of higher-order logic built directly on the Isabelle/Pure framework. The file contains 666 lines and is based on the Isabelle/Pure examples by Makarius Wenzel.

Our formalizations are available online:

`https://hol.compute.dtu.dk/Pure/`

We have used our approach in our automated reasoning course since 2020:

`https://kurser.dtu.dk/course/02256`

This is an advanced MSc course (about 40 students) with a focus on the natural deduction proof system, first-order logic, higher-order logic and type theory, in particular Isabelle/HOL [10].

The course is a 5 ECTS point MSc course and we have described various aspects of the course elsewhere [4, 7, 13, 15, 17] but we are not aware of other related work for teaching higher-order logic and natural deduction. In the course the students hand in 5 assignments and finally there is a 2-hour exam where the part using the approach described here usually constitutes 30% of the exam. All problems must be solved using the Isabelle proof assistant. The Isabelle files are handed in online.

In the course we are teaching we have some overarching goals. The students should gain a broad understanding of logic and reasoning, with formal proofs in higher-order logic as the end goal. Tied in with this is interactive theorem proving and foundations of mathematics. They should both learn how to use the Isabelle proof assistant in practice and the broader context of logic in computer science and the general project of formalizing mathematics.

For each of our aims we try to build up understanding in stages, starting with concepts that are easier or more familiar to the students. For example, we start by teaching propositional logic, then build up to first-order logic, and finally higher-order logic. In this paper we examine one such pipeline going from formalized first-order logic systems, to a formalized simple higher-order logic, and then finally the full Isabelle/HOL.

3 Related Work

In this paper, our main goal is to show how fundamental concepts of Isabelle can be taught using a simplified axiomatization of the same higher-order logic used in Isabelle/HOL.

In [2], a description of using Coq for teaching formal proofs to students is given. They make a connection to "textbook" proofs and how learning Coq first might make the students better at correct but more informal proofs later on. They give some example proofs similar in scope to example proofs and derived rules given in this paper.

In [8], a software tool for helping students construct proof trees in an interactive way is described. [12] describes a tool for verifying student proofs in tableaux. We view Isabelle as an alternative to such proof guides and checkers as it contains facilities both for guiding (and in some cases fully automating) the proof search and for checking the correctness of proofs.

In [16], a first-order logic natural deduction proof system is given. It is defined as a deep embedding in Isabelle/HOL and shown sound and complete. That paper focuses on using this system for teaching first-order logic to students. In addition to their Isabelle implementation they also give an interactive website for building and exporting proofs. The higher-order natural deduction developed in this paper is of course much more expressive but comes at the cost of verified meta properties like soundness and completeness.

We focus on proof systems based on implication, but it is worth noting that classical propositional logic can be presented, say, in a language containing only disjunction and negation as primitive connectives. For example, we have elsewhere [5] formalized the following axiom system with modus ponens as the only rule:

1. $p \lor p \to p$
2. $p \to p \lor q$
3. $(p \to q) \to (r \lor p) \to (q \lor r)$

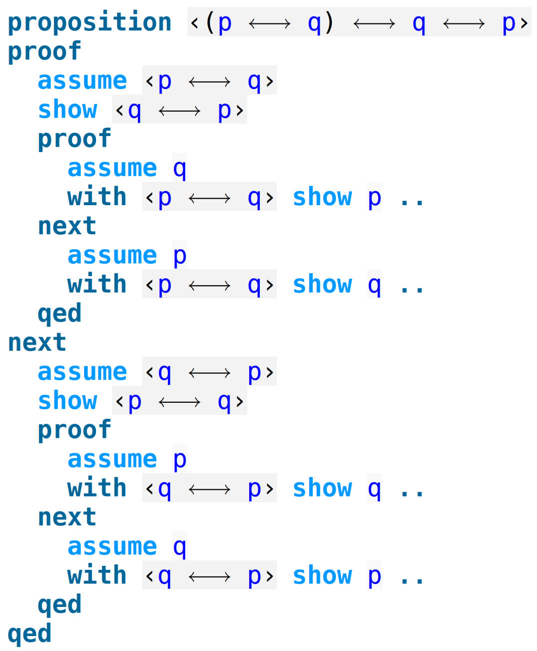
Here $p \to q$ is an abbreviation for $\neg p \lor q$. Without this abbreviation the axioms and the modus ponens rule would be difficult to grasp. The axiom system was first proved sound and complete by Rasiowa in 1947 [5], building on work by Russell and others. For natural deduction, in particular as in Isabelle/HOL, we find that the focus on implication is best, given the meta-logical operators like $\Longrightarrow$ (meta-logical implication) mentioned in the previous section.

4 Sample Natural Deduction Proofs in Isabelle

Our approach to teaching higher-order logic involves natural deduction proofs in propositional logic (see Subsection 4.1) and first-order logic (see Subsection 4.2), because a solid basis is needed. Finally, in Subsection 4.3, we consider natural deduction proofs of Cantor's theorem, a well-known result going beyond first-order logic.

4.1 Propositional Logic

We give a brief introduction to proving in Isabelle by the following sample natural deduction proof, which is an improved version of a proof given in our previous paper [15]. The proof works as-is in both our Isabelle/HOL_Pure and in standard Isabelle/HOL.



```
proposition ‹(p ⟷ q) ⟷ q ⟷ p›
proof
  assume ‹p ⟷ q›
  show ‹q ⟷ p›
  proof
    assume q
    with ‹p ⟷ q› show p ..
  next
    assume p
    with ‹p ⟷ q› show q ..
  qed
next
  assume ‹q ⟷ p›
  show ‹p ⟷ q›
  proof
    assume p
    with ‹q ⟷ p› show q ..
  next
    assume q
    with ‹q ⟷ p› show p ..
  qed
qed
```

The above figure shows a proof of the formula $(p \longleftrightarrow q) \longleftrightarrow (q \longleftrightarrow p)$. Note that the second pair of parentheses in the formula $(p \longleftrightarrow q) \longleftrightarrow (q \longleftrightarrow p)$ is dropped in the Isabelle proof, as all arrows are right-associative.

The structure of the proof should look familiar for logicians who know natural deduction. There are just two layers to the proof and four terminal branches. In the first layer we show the main bi-implication by assuming the left bi-implication then deriving the right bi-implication, then vice versa. The proofs of these two bi-implications make up the second layer. These proofs consist of assuming p then showing q, and vice versa.

The keywords of Isabelle and the Isar proof language [18–20] are similar to what one would use in a pen-and-paper natural deduction proof.

The starting "proposition" introduces a terminal fact which, once proved, becomes available for the remaining theory (and any other Isabelle documents that import this theory). There are several synonyms to "proposition", like "lemma", "theorem", and "corollary". After the "proposition" comes the formula which we want to prove.

We then use the "proof" keyword, which applies a single rule (in this case Isabelle has chosen the bi-implication introduction rule) and starts an Isar environment. The alternative "proof -" does the same without applying any rule first. The bi-implication introduction rule transforms the proof-state from the given formula to the two formulas: $p \longleftrightarrow q \Longrightarrow q \longleftrightarrow p$ and $q \longleftrightarrow p \Longrightarrow p \longleftrightarrow q$ (though the $\longleftrightarrow$ is displayed as $=$ in the Isabelle proof-state). $\Longrightarrow$ corresponds to a meta-implication in Isabelle, as opposed to the object-implication $\longrightarrow$. The meta-implications inform what the proof in the Isar environment should look like: the formula to the right of the rightmost arrow must be shown, while all the other formulas can be assumed.

The Isar environment consists of a list of facts, corresponding to proofs of these two formulas. In general, these facts will either be introduced by "assume", "have", or "show", though only "assume" and "show" are used in this proof. A fact introduced by "assume" requires no proof, while facts introduced by "have" and "show" must be shown either directly (using a possibly empty list of rules and proof methods) or by introducing new proof-blocks. This technique of introducing new proof-blocks for facts stated in another proof-block leads to the nested nature of Isabelle proofs. A fact introduced by “show” must finish off a goal of the proof-state, while a fact introduced by “have” can be any desired intermediate property. In each layer of the given proof we simply assume one formula and show another.

Since all the proofs introduced by the bi-implication introduction rule require showing two formulas, we need to clear the state after showing the first (thereby forgetting the previous assumptions) by using the “next” keyword.

There are four terminal facts in our proof: p, q, q, and p. We tell Isabelle to prove these directly by using “..”, which looks for an appropriate rule. In this case we need additional facts to match the bi-implication elimination rule. These facts are imported by the “with” keyword, which adds both explicitly given formulas and the previous fact to the context.

4.2 First-Order Logic

In our formalizations we have included 25 exercises covering mainly propositional and first-order logic. The exercises are of varying difficulty.

We include the solution to the final exercise below as a showcase for the power and elegance of our HOL_Pure formalization.

```
proposition 25: ‹(∀x. p x ⟶ (∃y. q y)) ⟶ (∀y. ¬ q y) ⟶ (∀x. ¬ p x)›
proof
  assume ‹∀x. p x ⟶ (∃y. q y)›
  show ‹(∀y. ¬ q y) ⟶ (∀x. ¬ p x)›
  proof
    assume ‹∀y. ¬ q y›
    show ‹∀x. ¬ p x›
    proof
      fix c
      show ‹¬ p c›
      proof
        assume ‹p c›
        from ‹∀x. p x ⟶ (∃y. q y)› have ‹p c ⟶ (∃y. q y)› ..
        from this and ‹p c› have ‹∃y. q y› ..
        then obtain c' where ‹q c'› ..
        from ‹∀y. ¬ q y› have ‹¬ q c'› ..
        from this and ‹q c'› show ⊥ ..
      qed
    qed
  qed
qed
```

The first two “proof” statements apply the implication introduction rule to break down the object-level implication to the meta implication, thereby allowing us to “assume” the precedent and “‘show” the somewhat simpler antecedent.

The third “proof” statement applies the universal introduction rule, so that we only have to show the statement for some specific but arbitrary c. The last proof statement applies the negation introduction rule, which allows us to assume the non-negated statement and try to derive a contradiction.

In the final proof block we use universal elimination, modus ponens, and existential elimination over the facts we have assumed so far, to obtain a c' for which q both does and does not hold. This finishes the proof.

4.3 Higher-Order Logic

Cantor's theorem is a famous result in set theory:

> There is no surjective function from a set to its power set.

Several formulations of Cantor's theorem are possible in higher-order logic. We have chosen to consider the following formulation in Isabelle/HOL:

```
theorem Cantor: ‹¬ (∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)›
```

The notation $'a$ is a stand in for an arbitrary type (non-empty). Note that sets can be represented as functions from the type of the elements to *bool*. We say that an element is in such a "set" if the function returns true for it.

It is illustrative for students to see that the proof of Cantor's theorem can be carried out in intuitionistic higher-order logic. Here is the full proof:

```
theorem Cantor: ‹¬ (∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)›
proof
  assume ‹∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x›
  then obtain f where ‹∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x› ..
  let ?D = ‹λx. ¬ f x x›
  from ‹∀s. ∃x. s = f x› have ‹∃x. ?D = f x› ..
  then obtain c where ‹?D = f c› ..
  let ?P = ‹f c c›
  have ‹?D c = ?D c› ..
  from this and ‹?D = f c› have ‹?D c = ?P› ..
  have ‹?D c›
  proof
    assume ?P
    with ‹?D c = ?P› have ‹?D c› ..
    from this and ‹?P› show ⊥ ..
  qed
  with ‹?D c = ?P› have ?P ..
  with ‹?D c› show ⊥ ..
qed
```

The proof above is written completely without appeal to advanced proof tactics, and each line represents at most one rule application. It thus works in both Isabelle/HOL_Pure and Isabelle/HOL (though in the latter one has to add $\bot$ as alternative notation for *False*). A detailed description of the proof follows.

```
theorem Cantor: ‹¬ (∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)›
proof
  assume ‹∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x›
  then obtain f where ‹∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x› ..
```

We use negation introduction (automatically picked by "proof") to transform the proof state from $\neg(\exists f.\ \forall s.\ \exists x.\ s = f\ x)$ to $\exists f.\ \forall s.\ \exists x.\ s = f\ x \Longrightarrow \bot$. Note that negation introduction actually just unfolds the definition of negation, and is therefore still intuitionistic. In the new proof state, we can assume the existence of a surjective function from the type $'a$ to $'a \Rightarrow bool$ (i.e. from a set to its power set). The "obtain" statement allows us to reference this function in the rest of the proof. The proof state now requires us to show $\bot$, i.e. find a contradiction.

```
  let ?D = ‹λx. ¬ f x x›
  from ‹∀s. ∃x. s = f x› have ‹∃x. ?D = f x› ..
  then obtain c where ‹?D = f c› ..
```

For some x of type $'a$, $f\ x$ will be a set where the elements also have type $'a$. We define a new set, $?D$, containing exactly those x not contained in $f\ x$ (recall that $f\ x\ x$ should be interpreted as the statement “x is in $f\ x$”).

We can show by our definition of f that there must be some c such that $?D = f\ c$.

```
  let ?P = ‹f c c›
  have ‹?D c = ?D c› ..
  from this and ‹?D = f c› have ‹?D c = ?P› ..
```

We let $?P$ be the statement that c is in $f\ c$. By reflexivity of equality and substitution we can prove that $?P$ must equal c being in D.

```
  have ‹?D c›
  proof
    assume ?P
    with ‹?D c = ?P› have ‹?D c› ..
    from this and ‹?P› show ⊥ ..
  qed
  with ‹?D c = ?P› have ?P ..
  with ‹?D c› show ⊥ ..
qed
```

This leads to a contradiction we can use to finish the proof. First, we can show that c must be in $?D$. Since this corresponds to c not being in $f\ c$, we can start by assuming $?P$ and try to show $\bot$. By the previous proof step we know that this must mean that c is in $?D\ c$. When unfolding the abbreviations we get that c both is and is not in $f\ c$, which shows $\bot$.

This shows that c must be in $?D$. By the same line of reasoning as above (though using the equality between c being in $?D$ and $?P$ holding in the opposite direction) we get that c both is and is not in $f\ c$. This finishes the proof.

We find that the full proof above is very useful for the students due to the lack of automation, but students should also work with the automation available in Isabelle/HOL, like in the following alternative proof.

```
theorem Cantor: ‹¬ (∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)›
proof -
  have ‹¬ (∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)› for f
  proof -
    {
      fix g
      have ‹∃p. ∀s :: 'a ⇒ bool. ∀x. p s x = (¬ s x)›
        by fast
      moreover have ‹∃s :: 'a ⇒ bool. ∀x. s x = f x x›
        by fast
      ultimately have ‹∃s. s ≠ f (g s)›
        by metis
    }
    then show ?thesis
      by metis
  qed
  then show ?thesis
    by fast
qed
```

The alternative proof above was found automatically by sledgehammer. We have rewritten the output of sledgehammer somewhat to be more legible, but not more than what we expect any student with a rudimentary understanding of Isabelle would be able to. Sledgehammer was only able to find the proof when the theorem is stated with f as a schematic variable, instead of a bound variable as in the original formulation. This is why we first prove an alternative formulation of the theorem, and then show that the quantified version follows from this.

We note that Cantor's theorem is right on the edge of what Isabelle's automation is able to handle. Sledgehammer could not return a single tactic application to solve the proof state, as is the common usage of this tool, but rather gave a large and quite illegible Isar proof. Furthermore, we were only able to find this proof using sledgehammer when using an unstable 2024 snapshot of Isabelle instead of the stable 2023 release.

In the stable 2023 release we obtain the following message from sledgehammer:

```
zipperposition found a proof...
zipperposition: One-line proof reconstruction failed: by metis
Warning: Isar proof construction failed
Done
```

In the snapshot available 10 January 2024 the Isar proof construction no longer fails. However, instead we obtained the illegible Isar proof mentioned above.

By a thorough examination of the proof given by sledgehammer, one can reduce it to the following two-step proof.

```
theorem Cantor: ‹¬ (∃f. ∀s :: 'a ⇒ bool. ∃x :: 'a. s = f x)›
proof -
  have ‹∀f g. (λs. s ≠ f (g s)) (λx. ¬ f x x)›
    by (metis (lifting))
  then show ?thesis
    by metis
qed
```

In the first step, we prove that for any function f from elements to sets and g from sets to elements, and set s that contains any x such that x is not in $f\ x$ (i.e. $s = ?D$), it must be the case that $s \neq f\ (g\ s)$. The reason for this is that s and $f\ (g\ s)$ must disagree on the element $g\ s$. By the definition of s, $g\ s$ must be in s if and only if it is not in $f\ (g\ s)$. This whole line of reasoning is found automatically by "metis" (the attribute "lifting" is needed).

In the second step, this is used to show the theorem. We need to show that the existence of a surjective function from a set to its power set implies a contradiction. If we assume that such a function exists, then there must also exist a function g which takes a set s and returns an element x such that $f\ x = s$. By the property shown in the first step we know that there is an s such that $s \neq f\ (g\ s)$. This, however, contradicts our definition of g. This line of reasoning was also found automatically by "metis" (the attribute "lifting" is not needed).

We note that the second step involves the use of the Axiom of Choice. From the fact that there for all s exists an x such that $f\ x = s$ we have to obtain a g that picks such an element when given an s. This proof is therefore not possible to perform at the stage in the development where we prove Cantor's theorem above; the Axiom of Choice is introduced later on.

```
AK: ⊢ p ⟶ q ⟶ p
AT: ⊢ (q ⟶ r) ⟶ (r ⟶ p) ⟶ q ⟶ p
AX: ⊢ ⊥ ⟶ p
AY: ⊢ ∀p ⟶ ⟨t⟩p
MP: ⊢ q ⟶ p ⟹ ⊢ q ⟹ ⊢ p
GR: ⊢ q ⟶ ⟨⋆a⟩p ⟹ new a p ⟹ new a q ⟹ ⊢ q ⟶ ∀p
PR: ⊢ (p ⟶ q) ⟶ p ⟹ ⊢ p

Assm: member p z ⟹ z ⇝ p
FlsE: z ⇝ ⊥ ⟹ z ⇝ p
ImpI: p # z ⇝ q ⟹ z ⇝ p ⟶ q
ImpE: z ⇝ p ⟶ q ⟹ z ⇝ p ⟹ z ⇝ q
UniI: z ⇝ ⟨⋆a⟩p ⟹ news a (p # z) ⟹ z ⇝ ∀p
UniE: z ⇝ ∀p ⟹ z ⇝ ⟨t⟩p
ImpC: (p ⟶ q) # z ⇝ p ⟹ z ⇝ p
```

Figure 1: The formalization of implicational axiomatics for first-order logic (the first 7 axioms and rules) and the formalization of natural deduction for first-order logic (the last 7 rules).

5 First-Order Logic — Soundness and Completeness

Our approach to teaching higher-order logic involves not only the formalization HOL_Pure, but also implicational axiomatics in the formalization FOL_Implicational_Axiomatics (see Subsection 5.1) and natural deduction in the formalization FOL_Natural_Deduction (see Subsection 5.2), both with formal proofs of the soundness and completeness theorems in Isabelle/HOL. Finally, in Subsection 5.3, we compare implicational axiomatics and natural deduction.

5.1 Implicational Axiomatics

Natural deduction is only one of multiple styles of proof systems. Most (with some notable exceptions like resolution) can be categorized as either Hilbert style or Gentzen style, where the difference, in broad strokes, is that Hilbert-style proof systems usually have few rules and many axioms while Gentzen-style systems have few axioms and many rules. Natural deduction is a subcategory of Gentzen systems. In the natural deduction system of the next section we only need a single axiom, namely Assm. In this section we present an alternative Hilbert style axiomatic proof system for first-order logic. The proof system is shown in the top of Figure 1. Both this and the system of the next section have been shown sound and complete in Isabelle/HOL.

In the axiomatic system there are four axioms and three rules. The modus ponens (MP) and generalization (GR) rules are quite standard, but the Peirce's Law (PR) rule is more commonly given as an axiom. It is notable that we can still show completeness of the system where this is given as a rule instead of an axiom. If we have the axiom $((p \longrightarrow q) \longrightarrow p) \longrightarrow p$ then we can show that $((p \longrightarrow q) \longrightarrow p) \Longrightarrow p$ is admissible by a single application of modus ponens. Showing that $((p \longrightarrow q) \longrightarrow p) \longrightarrow p$ is admissible from $((p \longrightarrow q) \longrightarrow p) \Longrightarrow p$, on the other hand, requires a lengthy derivation.

Despite the difference of approach, there is a correspondence between the individual axioms and rules of the axiomatic system and the natural deduction system of Figure 1. We will examine these in the following.

The AK axiom states that $p \longrightarrow q \longrightarrow p$ is valid for all p and q. This means that if we assume p (along with some passive q) then we can conclude p. This is similar to what the Assm axiom in the natural deduction system states.

The AT axiom states that implication is transitive: $(q \longrightarrow r) \longrightarrow (r \longrightarrow p) \longrightarrow q \longrightarrow p$. Such a rule is not necessary in natural deduction. The system would also be complete if we replaced this axiom with $(r \longrightarrow q \longrightarrow p) \longrightarrow (r \longrightarrow q) \longrightarrow r \longrightarrow p$ (AS), which works similarly to implication elimination under the context of a passive formula r. We have chosen to use AT instead of AS as we find it more intuitive and easier to explain to the students.

The MP rule represents modus ponens: if $p \longrightarrow q$ and p then q. In MP, modus ponens is given as a rule, which allows us to derive the validity of one formula from the validity of two other. This rule often forms the basis of reasoning within an axiomatic system. Modus ponens corresponds to implication elimination (ImpE) in natural deduction. A rule or axiom corresponding to implication introduction (ImpI) is not necessary in the axiomatic system, as all the implications necessary for building any valid formula can be introduced by the axioms.

The AX axiom allows us to conclude anything from falsity, like FlsE in natural deduction.

The AY axiom allows us to obtain $p\ c$ for a specific c from $\forall x.\ p\ x$, like UniE in natural deduction. The generalization rule (GR) allows us to conclude $\forall x.\ p\ x$ from the validity of $p\ c$ when c is fresh, similarly to UniI in natural deduction.

By adding the PR rule we extend to classical reasoning, like we do with ImpC in natural deduction.

5.2 Natural Deduction

In the following we describe the natural deduction system shown in the bottom-half of Figure 1. We also describe how the rules compare to the higher-order logic system given in Section 6 and Section 7. The judgements of the proof calculus are of the form $z \leadsto p$ where z contains a list of assumptions and p is the formula we try to prove.

The assumption rule (Assm) states that we can prove any formula we have assumed. This rule has no parallel in our higher-order logic, as it is implicit in the Isabelle framework; we can show $p \Longrightarrow p$ independently of our axioms.

The falsity elimination rule (FlsE) allows us to conclude anything if we have shown falsity (under a given set of assumptions). In higher-order logic it is possible to define falsity as $\forall p.\ p$, from which this property follows by the rules of the universal quantifier. In Lemma Falsity_E we derive the rule directly corresponding to FlsE using this technique.

The implication introduction rule (ImpI) states that if one has shown q while assuming p, then one has shown $p \longrightarrow q$.

The implication elimination rule (ImpE) states that if one has $p \longrightarrow q$ and p, then one has shown q.

The universal elimination rule (UniE) state that if one has $\forall x.\ p\ x$, then one can conclude $p\ c$ for any specific c. The three rules (ImpI, ImpE and UniE) are also given as axioms (that is, they are not derived) in the higher-order logic theory.

The universal introduction rule (UniI) here and the one for higher-order logic are defined slightly differently. Intuitively, these rules should represent that if one has shown $p\ c$ for some arbitrary c (i.e. a variable which one assumes nothing about) then one has shown $\forall x.\ p\ x$. This rule is sound because whatever reason you have for concluding $p\ c$ would work for any other terms as well. When defining the higher-order logic version of this rule we can rely on the Isabelle $\bigwedge$-binder and just define the rule as $(\bigwedge x.\ p\ x) \Longrightarrow \forall x.\ p\ x$. Isabelle will allow us to continue working on the object logic inside $\bigwedge x.\ p\ x$, while guaranteeing the freshness of x by binding it. In the deep-embedding approach used for first-order logic we have to do more work ourselves. First we define an Isabelle predicate for checking that a constant does not occur in a list of formulas. We can then define UniI as follows: if we can show $z \leadsto p$ after replacing x with a constant c in p and c does not occur in z or p, then we can conclude $z \leadsto \forall x.\ p$. In the Isabelle code this looks slightly different because we use de Bruijn indices.

The ImpC rule states that if we from $p \longrightarrow q$ can conclude p, then p must hold on its own. This rule is not intuitionistically valid and therefore has no parallel in the higher-order logic of Section 6. After extending the logic with the classical axiom of choice and the boolean and functional axioms of extensionality, which sometimes are and sometimes are not considered a part of intuitionistic logic, in Section 7 we are able to derive it.

5.3 Comparison of Implicational Axiomatics and Natural Deduction

As seen in the previous subsections, there is a substantial similarity between axiomatic first-order logic, first-order natural deduction, and higher-order natural deduction, at least when represented by the systems we have chosen.

By teaching all these systems to our students we aim to give them a broader understanding of logic and proofs and familiarize them with concepts they may encounter later. Vitally, by picking such similar systems we can really highlight the fundamental differences there are between these logics. For example that the real power of higher-order logic comes from what one allows the quantifiers to bind. Even without adding new rules this allows one to represent and prove properties that one could not in first-order logic.

One substantial difference between the systems we define is how they treat assumptions/context. In the higher-order natural deduction system we can use Isabelle's proof state environment to save and access context without having to mention it in the rules. In the first-order natural deduction system we must work with an explicit list of assumptions. When we apply implication introduction we can save the antecedent of the implication to the assumptions and when we apply universal introduction we must check that the arbitrary variable is actually fresh with regards to the context, for example. In the axiomatic system we have no context as such, but the passive formula in some of the rules allow us to save a kind of context when using the system in practice.

There is a further difference in what the implications in the two systems represent. In the implicational axiomatics formalization, implication is the fundamental reasoning tool used in derivations. This is why we use the same symbol, p, as the conclusion in all the axioms and rules. They represent different ways of deriving p, either by the assumptions in the same formula or by sound rules of reasoning. This focus on implication is also why we have called it implicational axiomatics. If one removed the axioms and rules containing other operators than implication (FlsE, UniI and UniE) one obtains a sound and complete proof system for classical implicational logic (i.e. the logic of formulas built only from implication).

In the natural deduction formalization implication is merely a logical operator to build formulas from. In this system the important logical relation is $\leadsto$, which separates the proof context (the assumptions) from the formula we are currently working to prove. The rules regarding implication in this system are either for obtaining a formula with implication (ImpI) or for using a formula with implication to show a subformula (ImpE).

6 Higher-Order Logic — Intuitionistic Logic

The main difference between the natural deduction system for first-order logic and our axiomatization of higher-order logic is which terms one allows quantification over. In first-order logic we only allow quantification over the fundamental domain elements, i.e. those given as arguments to functions and predicates. In higher-order logic, on the other hand, we allow quantification over all terms, including predicates and functions. This increases the expressiveness compared to the first-order logic substantially, even when the same axioms are used in both systems. As an example, consider again Cantor's Theorem, which one cannot state in first-order logic.

In the following we explain our formalization. We mostly show the definitions of our axioms and rules, and leave out the proofs of derived rules.

```
typedecl bool

judgment Trueprop :: ‹bool ⇒ prop› (‹_› 5)

axiomatization Imp (infixr ‹⟶› 25)
  where Imp_I [intro]: ‹(p ⟹ q) ⟹ p ⟶ q›
    and Imp_E [elim]: ‹p ⟶ q ⟹ p ⟹ q›

axiomatization Uni (binder ‹∀› 10)
  where Uni_I [intro]: ‹(⋀x. p x) ⟹ ∀x. p x›
    and Uni_E [elim]: ‹∀x. p x ⟹ p c› for c :: 'a
```

The formalization starts by introducing the boolean type, which becomes the type of formulas (or facts). We then give axiomatic definitions of the object-level implication ($\longrightarrow$) and universal quantifier ($\forall$). These are defined with regards to the meta-level implication ($\Longrightarrow$) and universal quantifier ($\bigwedge$), which are defined in Isabelle/Pure. Note that in higher-order logic, these four rules are all we need for intuitionistic logic; the other common operators can be derived as follows.

```
definition Falsity (‹⊥›) where ‹⊥ ≡ ∀p. p›

lemma Falsity_E [elim]: ‹⊥ ⟹ p›

definition Neg (‹¬ _› [40] 40) where ‹¬ p ≡ p ⟶ ⊥›

lemma Neg_I [intro]: ‹(p ⟹ ⊥) ⟹ ¬ p›

lemma Neg_E [elim]: ‹¬ p ⟹ p ⟹ q›

definition Truth (‹⊤›) where ‹⊤ ≡ ¬ ⊥›

lemma Truth_I [intro]: ⊤
```

We define falsity, negation, and truth. Falsity ($\bot$) is defined as $\forall p.\ p$, or "everything is true", which has obvious parallels to the principle of explosion. The negation of a formula p is defined as $p \longrightarrow \bot$, and truth ($\top$) is defined as the negation of falsity.

We also give the introduction and elimination rules for these operators. Both the falsity elimination rule and the negation elimination rule correspond to "from a contradiction, anything follows". The introduction rules are as one would expect.

```
definition Con (infixr ‹∧› 35) where ‹p ∧ q ≡ ∀r. (p ⟶ q ⟶ r) ⟶ r›

lemma Con_I [intro]: ‹p ⟹ q ⟹ p ∧ q›

lemma Con_E1 [elim]: ‹p ∧ q ⟹ p›

lemma Con_E2 [elim]: ‹p ∧ q ⟹ q›

definition Dis (infixr ‹∨› 30) where ‹p ∨ q ≡ ∀r. (p ⟶ r) ⟶ (q ⟶ r) ⟶ r›

lemma Dis_I1 [intro]: ‹p ⟹ p ∨ q›

lemma Dis_I2 [intro]: ‹q ⟹ p ∨ q›

lemma Dis_E [elim]: ‹p ∨ q ⟹ (p ⟹ r) ⟹ (q ⟹ r) ⟹ r›
```

Two additional propositional operators, conjunction and disjunction, are defined by the universal quantifier and implication. The alternative definitions $p \land q \equiv \neg(p \longrightarrow \neg q)$ and $p \lor q \equiv \neg p \longrightarrow q$ would be simpler, but they don't work in an intuitionistic context. That is, without extending to classical logic it is impossible to derive the introduction and elimination rules given above if one used these definitions.

```
definition Exi (binder ‹∃› 10) where ‹∃x. p x ≡ ∀q. (∀x. p x ⟶ q) ⟶ q›

lemma Exi_I [intro]: ‹p c ⟹ ∃x. p x›

lemma Exi_E [elim]: ‹∃x. p x ⟹ (⋀x. p x ⟹ q) ⟹ q›
```

The existential quantifier is also defined by the universal quantifier and implication. The elimination rule (which is very closely related to the definition) can be rewritten to the following: $(\bigwedge x.\ p\ x \Longrightarrow q) \Longrightarrow (\exists x.\ p\ x) \Longrightarrow q$ (we note that some of the parentheses are superfluous). This means that we can obtain a fixed variable fulfilling p when trying to prove q from the assumption that such a variable exists.

```
definition Equality (infix ‹=› 50) where ‹x = y ≡ ∀p. p x ⟶ p y›

lemma Equality_I [intro]: ‹x = x›

lemma Equality_E1 [elim]: ‹x = y ⟹ p x ⟹ p y›

lemma Equality_E2 [elim]: ‹x = y ⟹ p y ⟹ p x›

lemma Equality_1E [elim]: ‹p x ⟹ x = y ⟹ p y›
  by (rule Equality_E1)

lemma Equality_2E [elim]: ‹p y ⟹ x = y ⟹ p x›
  by (rule Equality_E2)
```

We define two elements as equal if any property which holds for the first must also hold for the second. This gives rise to various substitution rules.

These axioms, definitions, and rules are enough to prove Cantor's Theorem.

7 Higher-Order Logic — Classical Logic

We will now extend the logic from the intuitionistic realm to the classical. This requires new axioms.

```
abbreviation (input) Iff (infixr ‹⟷› 25) where ‹p ⟷ q ≡ p = q› for p :: bool

axiomatization where Iff_I [intro]: ‹(p ⟹ q) ⟹ (q ⟹ p) ⟹ p ⟷ q›

lemma Iff_E1 [elim]: ‹p ⟷ q ⟹ p ⟹ q›
  by (rule Equality_E1)

lemma Iff_E2 [elim]: ‹p ⟷ q ⟹ q ⟹ p›
  by (rule Equality_E2)
```

The bi-implication introduction rule (Iff_I) above and the axiom of extensionality (Extension) below are sometimes considered part of intuitionistic logic. They are, however, not derivable from the four axioms we have defined so far, and must therefore also be given as axioms.

```
axiomatization where Extension: ‹(⋀x. f x = g x) ⟹ f = g› for f :: ‹'a ⇒ 'b›

axiomatization Epsilon (‹ε›) where Choice: ‹p c ⟹ p (ε p)› for c :: 'a

lemma Imp_C: ‹(p ⟶ q ⟹ p) ⟹ p›

lemma LEM: ‹p ∨ ¬ p›
proof (rule Imp_C)
  assume ‹p ∨ ¬ p ⟶ ⊥›
  have ‹¬ p›
  proof
    assume p
    then have ‹p ∨ ¬ p› ..
    with ‹p ∨ ¬ p ⟶ ⊥› show ⊥ ..
  qed
  then show ‹p ∨ ¬ p› ..
qed
```

Above we axiomatize Hilbert’s epsilon-operator, corresponding to the axiom of choice, which finally bridges the gap to classical logic. It is now possible to prove a version of Peirce’s Law (Imp_C, where C stands for classical) which is here given as a natural deduction rule rather than a logical formula. If one instantiates q with $\bot$, then Imp_C corresponds almost exactly to the Isabelle/HOL rule named classical: $(\neg p \Longrightarrow p) \Longrightarrow p$. The proof of Imp_C is very long, and is therefore not included. We instead show how one can use it to derive the Law of Excluded Middle (LEM). With LEM we have classical logic as in our previous paper [15], but now for higher-order logic and not just propositional logic.

In order to make the classical higher-order logic easier to use we can introduce the following rules known from Isabelle/HOL.

```
lemma classical: ‹(¬ p ⟹ p) ⟹ p›
proof -
  have ‹p ⟹ p› .
  assume ‹¬ p ⟹ p›
  with LEM and ‹p ⟹ p› show p ..
qed

lemma ccontr: ‹(¬ p ⟹ ⊥) ⟹ p›
proof -
  assume ‹¬ p ⟹ ⊥›
  then have ‹¬ p ⟹ p› ..
  with classical show p .
qed
```

Both rules are useful in the final assignment in the course where the students must prove the following formula, among others.

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

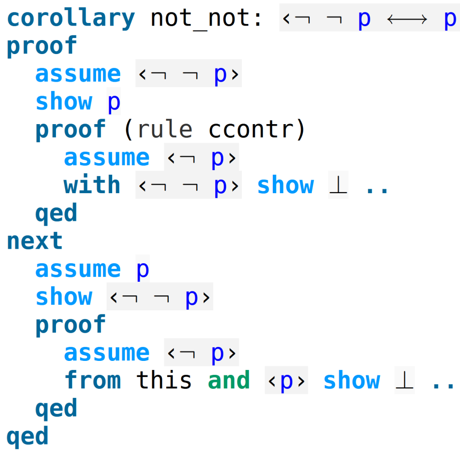
Formalization with r (*rich*) and f (*father*):

$$(\forall x.\ \neg\ r\ x \longrightarrow r\ (f\ x)) \longrightarrow (\exists x.\ r\ x\ \wedge\ r\ (f\ (f\ x)))$$

We prefer to keep the proof secret for students.

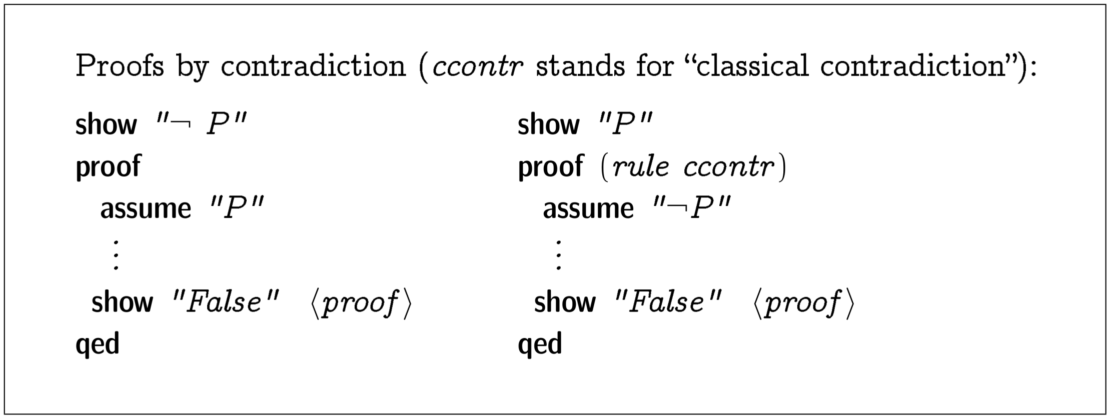
The rule classical is also useful for students of Isabelle/HOL in proofs of, for example, the well-known Clavius's Law and Peirce's Law [15].

The rule ccontr (for classical contradiction) is also useful in the following proof: rule (ccontr):



```
corollary not_not: ‹¬ ¬ p ⟷ p›
proof
  assume ‹¬ ¬ p›
  show p
  proof (rule ccontr)
    assume ‹¬ p›
    with ‹¬ ¬ p› show ⊥ ..
  qed
next
  assume p
  show ‹¬ ¬ p›
  proof
    assume ‹¬ p›
    from this and ‹p› show ⊥ ..
  qed
qed
```

We would like to emphasize that the rule ccontr (for classical contradiction) is explained in the Programming and Proving in Isabelle/HOL tutorial [9], on page 45:



Proofs by contradiction (*ccontr* stands for "classical contradiction"):

show *"¬ P"*	show *"P"*
proof	proof (*rule ccontr*)
assume *"P"*	assume *"¬P"*
⋮	⋮
show *"False"* ⟨*proof*⟩	show *"False"* ⟨*proof*⟩
qed	qed

Note that in the tutorial the negation introduction is also called a proof by contradiction. This might be rather confusing for students with a background in intuitionistic logic, but for most Isabelle novices it is actually a good way to view it.

8 Further Developments and Concluding Remarks

We now have most of the features in Isabelle/HOL but three features are missing.

The first is a simple axiomatization of the polymorphic constant for undefinedness.

```
axiomatization undefined :: 'a
```

Now the axiom of infinity given in two parts (Infinity_Base and Infinity_Step).

```
typedecl ind

axiomatization Zero (‹0›) and Suc :: ‹ind ⇒ ind›
  where Infinity_Base: ‹Suc x = 0 ⟹ p›
    and Infinity_Step: ‹Suc x = Suc y ⟹ x = y›

theorem No_Other_Zero: ‹¬ Suc x = 0›
proof
  assume ‹Suc x = 0›
  with Infinity_Base show ⊥ .
qed

corollary No_Other_Zero_Exists: ‹¬ (∃x. Suc x = 0)›
proof
  assume ‹∃x. Suc x = 0›
  then obtain c where ‹Suc c = 0› ..
  with No_Other_Zero show ⊥ ..
qed
```

Students can compare this to the similar axiomatization in the Isabelle/HOL theory Nat for natural numbers. The later introduction of the real numbers and also the complex numbers does not require additional axioms [3].

Finally we introduce facilities for sets, in particular the set membership predicate (∈) and the axiom of comprehension (Comprehension).

```
typedecl 'a set

axiomatization Collect and Member :: ‹'a ⇒ 'a set ⇒ bool› (infix ‹∈› 51)
  where Comprehension: ‹x ∈ Collect p ⟷ p x›
    and Set_Reduction: ‹Collect (λx. x ∈ s) = s›

theorem No_Member_Empty_Set: ‹¬ x ∈ Collect (λ_. ⊥)›
proof
  assume ‹x ∈ Collect (λ_. ⊥)›
  with Comprehension show ⊥ ..
qed

corollary No_Member_Empty_Set_Exists: ‹¬ (∃x. x ∈ Collect (λ_. ⊥))›
proof
  assume ‹∃x. x ∈ Collect (λ_ :: 'a. ⊥)›
  then obtain c where ‹c ∈ Collect (λ_ :: 'a. ⊥)› ..
  with No_Member_Empty_Set show ⊥ ..
qed
```

Except for some small differences in names, like *False* instead of ⊥, the above theorem and corollary have identical proofs in Isabelle/HOL as documented in the following stand-alone theory.

```
theory Scratch imports Main begin

theorem No_Member_Empty_Set: ‹¬ x ∈ Collect (λ_. False)›
proof
  assume ‹x ∈ Collect (λ_. False)›
  with mem_Collect_eq show False ..
qed

corollary No_Member_Empty_Set_Exists: ‹¬ (∃x. x ∈ Collect (λ_. False))›
proof
  assume ‹∃x. x ∈ Collect (λ_ :: 'a. False)›
  then obtain c where ‹c ∈ Collect (λ_ :: 'a. False)› ..
  with No_Member_Empty_Set show False ..
qed

end
```

The *Main* theory is the main theory of Isabelle/HOL. We note that the axiom of comprehension is called mem_Collect_eq in Isabelle/HOL.

We have found our formalization to be of great help when teaching both proving in Isabelle and in propositional logic, first-order logic and higher-order logic as such. When stuck on a proof students can quickly go through the entire repertoire of rules to find one that might help them. Future work includes elaboration on the overall approach, including proper teaching materials with even more explanations and examples.

As stated in the introduction, our approach has been used in a computer science course on automated reasoning in the period 2020-2023, and currently 84 students have registered in the spring 2024 course. We plan to discuss our teaching experiences thoroughly in a forthcoming paper. As always the Technical University of Denmark makes statistics about the course evaluations and student grades publicly available as soon as the course ends.

Acknowledgements Thanks to Marco Dessalvi, Frederik Krogsdal Jacobsen and Roberto Pettinau for help with the paper and the formalizations.

References

[1] Alexander Bentkamp, Jasmin Blanchette, Visa Nummelin, Sophie Tourret, Petar Vukmirović & Uwe Waldmann (2023): *Mechanical Mathematicians*. *Commun. ACM* 66(4), p. 80–90, doi:10.1145/3557998.

[2] Sebastian Böhne & Christoph Kreitz (2017): *Learning how to Prove: From the Coq Proof Assistant to Textbook Style*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017*, *EPTCS* 267, Open Publishing Association, pp. 1–18, doi:10.4204/EPTCS.267.1.

[3] William M. Farmer (2008): *The seven virtues of simple type theory*. *Journal of Applied Logic* 6(3), pp. 267–286, doi:10.1016/j.jal.2007.11.001.

[4] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France,*

29th June 2020, *Electronic Proceedings in Theoretical Computer Science* 328, Open Publishing Association, pp. 18–34, doi:10.4204/EPTCS.328.2.

[5] Asta Halkjær From, Agnes Moesgård Eschen & Jørgen Villadsen (2021): *Formalizing Axiomatic Systems for Propositional Logic in Isabelle/HOL.* In Fairouz Kamareddine & Claudio Sacerdoti Coen, editors: *Intelligent Computer Mathematics - 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings*, *Lecture Notes in Artificial Intelligence* 12833, Springer, pp. 32–46, doi:10.1007/978-3-030-81097-9.

[6] Asta Halkjær From, Frederik Krogsdal Jacobsen & Jørgen Villadsen (2021): *SeCaV: A Sequent Calculus Verifier in Isabelle/HOL.* In: *16th International Workshop on Logical and Semantic Frameworks with Applications (LSFA 2021)*, *Electronic Proceedings in Theoretical Computer Science* 357, Open Publishing Association, pp. 38–55, doi:10.4204/EPTCS.357.4.

[7] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2023): *On Exams with the Isabelle Proof Assistant.* In Pedro Quaresma, João Marcos & Walther Neuper, editors: Proceedings 11th International Workshop on *Theorem Proving Components for Educational Software*, Haifa, Israel, 11 August 2022, *Electronic Proceedings in Theoretical Computer Science* 375, Open Publishing Association, pp. 63–76, doi:10.4204/EPTCS.375.6.

[8] Joomy Korkut (2022): *A Proof Tree Builder for Sequent Calculus and Hoare Logic.* In Pedro Quaresma, João Marcos & Walther Neuper, editors: *Proceedings 11th International Workshop on Theorem Proving Components for Educational Software, ThEdu@FLoC 2022, Haifa, Israel, 11 August 2022*, *EPTCS* 375, Open Publishing Association, pp. 54–62, doi:10.4204/EPTCS.375.5.

[9] Tobias Nipkow (2023): *Programming and Proving in Isabelle/HOL (Tutorial).* `https://isabelle.in.tum.de/doc/prog-prove.pdf`.

[10] 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.

[11] Lawrence C. Paulson (2018): *Computational Logic: Its Origins and Applications.* *Proc. R. Soc. A. 474 20170872* 2210, doi:10.1098/rspa.2017.0872.

[12] Davi Romero de Vasconcelos (2022): *ANITA: Analytic Tableau Proof Assistant.* In Pedro Quaresma, João Marcos & Walther Neuper, editors: *Proceedings 11th International Workshop on Theorem Proving Components for Educational Software, ThEdu@FLoC 2022, Haifa, Israel, 11 August 2022*, *EPTCS* 375, Open Publishing Association, pp. 38–53, doi:10.4204/EPTCS.375.4.

[13] Jørgen Villadsen (2023): *A Formulation of Classical Higher-Order Logic in Isabelle/Pure.* In: *Proceedings of Logic & Artificial Intelligence*, Vladimir Andrunachievici Institute of Mathematics and Computer Science, pp. 223–238.

[14] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA).* In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *EPTCS* 290, pp. 14–29, doi:10.4204/EPTCS.290.2.

[15] Jørgen Villadsen, Asta Halkjær From & Patrick Blackburn (2022): *Teaching Intuitionistic and Classical Propositional Logic Using Isabelle.* In João Marcos, Walther Neuper & Pedro Quaresma, editors: Proceedings 10th International Workshop on *Theorem Proving Components for Educational Software*, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, *Electronic Proceedings in Theoretical Computer Science* 354, Open Publishing Association, pp. 71–85, doi:10.4204/EPTCS.354.6.

[16] Jørgen Villadsen, Asta Halkjær From & Anders Schlichtkrull (2017): *Natural Deduction and the Isabelle Proof Assistant.* In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 Aug 2017*, *EPTCS* 267, Open Publishing Association, pp. 140–155, doi:10.4204/EPTCS.267.9.

[17] Jørgen Villadsen & Frederik Krogsdal Jacobsen (2021): *Using Isabelle in Two Courses on Logic and Automated Reasoning.* In João F. Ferreira, Alexandra Mendes & Claudio Menghi, editors: *Formal Methods Teaching*, Springer International Publishing, Cham, pp. 117–132, doi:10.1007/978-3-030-91550-6_9.

[18] Makarius Wenzel (2007): *Isabelle/Isar - a generic framework for human-readable proof documents*. *From Insight to Proof - Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar, and Rhetoric. University of Białystok* 10(23), pp. 277–298.

[19] Makarius Wenzel (2023): *The Isabelle/Isar Reference Manual*. `https://isabelle.in.tum.de/doc/isar-ref.pdf`.

[20] Markus Wenzel (1999): *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents*. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring & Laurent Théry, editors: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, *Lecture Notes in Computer Science* 1690, Springer, pp. 167–184, doi:10.1007/3-540-48256-3_12.

A Coq Library of Sets for Defining Denotational Semantics

Qinxiang Cao
Shanghai Jiao Tong University, Shanghai, China
caoqinxiang@gmail.com

Xiwei Wu
Shanghai Jiao Tong University, Shanghai, China
yashen@sjtu.edu.cn

Yalun Liang
Shanghai Jiao Tong University, Shanghai, China
symb@olic.link

Sets and relations are very useful concepts for defining denotational semantics. In the Coq proof assistant, curried functions to Prop are used to represent sets and relations, e.g. `A -> Prop`, `A -> B -> Prop`, `A -> B -> C -> Prop`, etc. Further, the membership relation can be encoded by function applications, e.g. `X a` represents $a \in X$ if `X: A -> Prop`. This is very convenient for developing formal definitions and proofs for professional users, but it makes propositions more difficult to read for non-professional users, e.g. students of a program semantics course. We develop a small Coq library of sets and relations so that standard math notations can be used when teaching denotational semantics of simple imperative languages. This library is developed using Coq's type class system. It brings about zero proof-term overhead comparing with the existing formalization of sets.

1 Introduction

Theorem provers are very useful in teaching program semantics and program logics. Formal definitions can help students understand different concepts more precisely. Also, students can get immediate feedback from the theorem prover when writing a machine-checkable proof in their homework.

However, it is not easy to redesign existing courses by adding a theorem prover to it. To clarify, our objective here is to design course material that focuses on programming language theories. Interactive theorem proving is only used to help teaching. We cannot spent too much time on how to use a theorem prover and how to turn math concepts into formal definitions. Thus, it is especially critical to ensure that formal definitions in course material are intuitive to read, otherwise students need to pay significant efforts on learning and understanding the formal language used, in addition to learn their counterpart written in standard math symbols. When we prepared course material about denotational semantics, we found that the existing Coq formalization of sets and relations cannot be used directly in teaching.

Problems in sets' definitions. Typically, if a set `A` is formalized as a Coq type, then a subset of `A` can be formalized as a function of type `A -> Prop`, and set operators like union and intersection can be defined as higher-order functions. For example[1],

```
Definition union1 (X Y: A -> Prop) :=
  fun a => X a \/ Y a.
Notation "x ∪ y" := (union1 x y).
```

Moreover, the membership relation can be encoded by function applications, e.g. `X a` represents $a \in X$ if `X: A -> Prop`.

---

[1] Coq.Sets.Ensemble [11] defines subsets like this and its formalization of union is equivalent to our definition here.

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 79–95, doi:10.4204/EPTCS.400.6

© Cao, Wu and Liang
This work is licensed under the
Creative Commons Attribution License.

Similarly, relations from A to B (i.e. subsets of A × B) can be formalized as curried functions of type A -> B -> Prop and the union operator can be formalized as:

```
Definition union2 (X Y: A -> B -> Prop) :=
  fun a b => X a b \/ Y a b.
```

For membership, `X a b` represents $(a,b) \in X$ if `X: A -> B -> Prop`. Here, using curried functions instead of its uncurried counterparts of type A * B -> Prop allows us to avoid verbose construction and destruction of pairs in proofs.

Although curried functions are widely used for representing subsets and relations[2], it brings two problems.

(a) It is nontrivial to unify union1 and union2 above. We want to unify them because it is inconvenient to have two different Coq definitions for *union*. Using those two definitions means that, important properties like union's commutativity and associativity should also have two copies. It can be annoying that we need to choose a correct version every time we try to apply them. Especially in our teaching scenario, we should not require students to spend their attention on these subtleties, e.g. remembering the Coq theorem names of these different versions.

(b) Propositions about the membership relation are less intuitive to read. Students are more willing to use standard math notations like $a \in X$ and $(a,b) \in X$ instead of `X a` and `X a b` respectively. One potential solution is to define such membership relations using higher order functions:

```
Definition In1 (a: A) (X: A -> Prop) := X a.
Notation "x ∈ y" := (In1 x y).
Definition In2 (p: A * B) (X: A -> B -> Prop) :=
  X (fst p) (snd p).
```

Then, we need a unified definition of In1, In2 and potentially some In3 for ternary relations.

Remark. Giving up curried Coq types like A -> B -> Prop and choosing to use uncurried Coq types like A * B -> Prop seems a potential solution to these two problems above. However, this does not really work because we would be unable to use all standard library supports for relations, including those useful type classes (like Equivalence) and proof automation (like the morphism-based rewrite tactic).

Problems in set-related proof automation. When students learn set theory in their discrete math course, set equivalence and set inclusion can be reduced to propositions about the membership relation, i.e.

$$X = Y \quad \text{iff.} \quad \forall a.\, a \in X \iff a \in Y$$

$$X \subseteq Y \quad \text{iff.} \quad \forall a.\, a \in X \Rightarrow a \in Y$$

Such reduction is very useful because we can apply some proof automation like tauto and firstorder afterwards. If we define set equivalence and set inclusion accordingly (see below), it seems that these reduction can be simply implemented as an unfold tactic in Coq.

```
Definition included2 (X Y: A -> B -> Prop): Prop :=
  forall a b, X a b -> Y a b.
Notation "x ⊆ y" := (included2 x y).
```

---

[2] For example, Coq.Relations.Relation_Definitions [10] defines relations from A to A by functions of type A -> A -> Prop.

But in fact, unfolding does not work, because it cannot reduces X ⊆ Y to

```
forall a b, (a, b) ∈ X -> (a, b) ∈ Y.
```

As we just mentioned, we want to use standard math symbols which is more intuitive to read.

One may argue that included2 should be defined using the membership relation In2. But this cannot solve our problem either, since only changing included2's definition does not allow us to unfold X ⊆ Y ∪ Z to

```
forall a b, (a, b) ∈ X -> (a, b) ∈ Y \/ (a, b) ∈ Z.
```

Problems about composing binary relations. If we only consider binary relation composition, it can be easily defined by FOL propositions (see below). But we find that the composition operator between a binary relation and a unary relation (see below) shares many similar properties with the ordinary composition, e.g. associativity and distribution over set union.

```
Definition comp22 (R: A -> B -> Prop) (S: B -> C -> Prop) :=
  fun a c => exists b, R a b /\ S b c.
Definition comp21 (R: A -> B -> Prop) (S: B -> Prop) :=
  fun a => exists b, R a b /\ S b.
```

This similarity between comp22 and comp21 is natural, because a unary relation (i.e. S: B -> Prop) can be treated as a binary relation to the unit type (i.e. S: B -> unit -> Prop).

We want to have a unified definition of these two composition operators, and even some other composition operator about labeled relations (see below), so that our formalized material can help students understand this similarity.

```
Definition comp33
  (R: A -> list event -> B -> Prop)
  (S: B -> list event -> C -> Prop) :=
fun a l c => exists l1 l2 b,
  R a l1 b /\ S b l2 c /\ l = l1 ++ l2.
```

Our set library. In this paper, we present our Coq library of sets, which is initially designed for the purpose of teaching a programming language course. The source files can be found at: `https://bitbucket.org/qinxiang-SJTU/sets/src/`. During our development, we found that some of our requirement mentioned above like a unified definition of unions and a unified definition of relation compositions is also meaningful for professional users. Thus we carefully design our unified definitions and automated tactics so that using our library will cause no proof-term overhead comparing with using naive definitions like union1, union2, comp22, etc.

The rest of this paper. We will first give an overview of this library from the users' point of view in §2. We will then introduce its application in teaching in §3. For techniques used in developing this library, we introduce our unified definitions of set operators, relation compositions, and the membership relation in §4. We introduce our proof automation support in §5. Finally, we discuss related work in §6 and conclude in §7.

2 Overview of the library

2.1 Functions, predicates and constants

In this library, every Coq type T with form A -> B -> ... -> Prop is treated as sets, and we provide the following functions, predicates and constants for T.

```
Sets.equiv: T -> T -> Prop                                          (* == *)
Sets.included: T -> T -> Prop                                       (* ⊆ *)
Sets.empty: T                                                       (* ∅ *)
Sets.full: T
Sets.union: T -> T -> T                                             (* ∪ *)
Sets.intersect: T -> T -> T                                         (* ∩ *)
Sets.indexed_union: forall {I: Type}, (I -> T) -> T                 (* ⋃ *)
Sets.indexed_intersect: forall {I: Type}, (I -> T) -> T             (* ⋂ *)
Sets.general_union: (T -> Prop) -> T
Sets.general_intersect: (T -> Prop) -> T
```

Their meanings are mostly ordinary. Sets.indexed_union X and Sets.general_union P mean

$$\bigcup_{i\in I} X(i) \quad \text{and} \quad \bigcup_{X \text{ satisfies } P} X$$

respectively. Similarly, Sets.indexed_intersect X and Sets.general_intersect P mean

$$\bigcap_{i\in I} X(i) \quad \text{and} \quad \bigcap_{X \text{ satisfies } P} X$$

respectively. Thus, an instance of intersection's distribution law over a countably infinite union can be written in Coq as:

```
forall (A: Type) (X: A -> Prop) (Y: nat -> A -> Prop),
  X ∩ ⋃ Y == ⋃ (fun n => X ∩ Y n)
```

Besides, Sets.In is an uncurried Coq function representing the membership relation, i.e. from users' point of view, Sets.In "have" the following Coq types.

```
A -> (A -> Prop) -> Prop                            (* a ∈ X *)
A * B -> (A -> B -> Prop) -> Prop                   (* (a, b) ∈ X *)
A * B * C -> (A -> B -> C -> Prop) -> Prop          (* (a, b, c) ∈ X *)
...
```

The set library defines Rels.concat (R ∘ S) to represent different kinds of relation compositions. From users' point of view, the following variants are supported.

- If $R \subseteq A \times B$ and $S \subseteq B \times C$, then $R \circ S \subseteq A \times C$ (in Coq, R: A -> B -> Prop, S: B -> C -> Prop, and R ∘ S: A -> C -> Prop) and

```
(a, c) ∈ R ∘ S <->
  exists b: B, (a, b) ∈ R /\ (b, c) ∈ S.
```

- If $R \subseteq A \times B$ and $S \subseteq B$, then $R \circ S \subseteq A$ (in Coq, R: A -> B -> Prop, S: B -> Prop, and R ∘ S: A -> Prop) and

```
a ∈ R ∘ S <->
  exists b: B, (a, b) ∈ R /\ b ∈ S.
```

- If $R \subseteq A \times E^* \times B$ and $S \subseteq B \times E^* \times C$, then $R \circ S \subseteq A \times E^* \times C$ (in Coq, R: A -> list E -> B -> Prop, S: B -> list E -> C -> Prop, and R ∘ S: A -> list E -> C -> Prop) and

```
(a, l, c) ∈ R ∘ S <->
  exists (b: B) (l1 l2: list E),
    (a, l1, b) ∈ R /\ (b, l2, c) ∈ S /\ l = l1 ++ l2.
```

- If $R \subseteq A \times E^* \times B$ and $S \subseteq B \times E^*$, then $R \circ S \subseteq A \times E^*$ (in Coq, R: A -> list E -> B -> Prop, S: B -> list E -> Prop, and R ∘ S: A -> list E -> Prop) and

```
(a, l) ∈ R ∘ S <->
  exists (b: B) (l1 l2: list E),
    (a, b) ∈ R /\ b ∈ S /\ l = l1 ++ l2.
```

- If $R \subseteq A \times E^* \times B$ and $S \subseteq B \times E^\omega$, then $R \circ S \subseteq A \times E^\omega$ (in Coq, R: A -> list E -> B -> Prop, S: B -> Stream E -> Prop, and R ∘ S: A -> Stream E -> Prop) and

```
(a, l) ∈ R ∘ S <->
  exists (b: B) (l1: list E) (l2: Stream E),
    (a, b) ∈ R /\ b ∈ S /\ l = l1 +++ l2
```

where for any l: Stream E,

```
[e1; e2; ...; en] +++ l =
  Cons e1 (Cons e2 (... (Cons en l) ...)
```

Although we do provide supports for more complicated cases, they are not used in teaching.

The set library also defines Rels.id to represent “the identity relation”. The most useful variants are:

- For a b: A,

```
(a, b) ∈ Rels.id <-> a = b.
```

- For a b: A and l: list E,

```
(a, l, b) ∈ Rels.id <-> a = b /\ l = nil.
```

2.2 Useful theorems

We provide 61 useful lemmas about these set-related functions and predicates[3]. Here are the most important ones.

- Commutativity and associativity of unions and intersections.

```
Sets_union_comm:
  forall (x y: T), x ∪ y == y ∪ x;
Sets_union_assoc:
  forall (x y z: T), (x ∪ y) ∪ z == x ∪ (y ∪ z);
```

---

[3] Users usually cannot remember all of their names and thus need to seek help from Coq’s Search command.

```
Sets_intersect_comm:
  forall (x y: T), x ∩ y == y ∩ x;
Sets_intersect_assoc:
  forall (x y z: T), (x ∩ y) ∩ z == x ∩ (y ∩ z);
```

where T is a Coq type representing sets.

- Associativity of relation composition.

```
Rels_concat_assoc:
  forall x y z, (x ∘ y) ∘ z == x ∘ (y ∘ z).
```

We support different variants of this theorem based on different variants of relation compositions. Specifically, x, y and z above may have the following Coq types.

```
Case 1:
  x: A -> B -> Prop, y: B -> C -> Prop, z: C -> D -> Prop
Case 2:
  x: A -> B -> Prop, y: B -> C -> Prop, z: C -> Prop
Case 3:
  x: A -> list E -> B -> Prop, y: B -> list E -> C -> Prop,
  z: C -> list E -> D -> Prop
Case 4:
  x: A -> list E -> B -> Prop, y: B -> list E -> C -> Prop,
  z: C -> list E -> Prop
Case 5:
  x: A -> list E -> B -> Prop, y: B -> list E -> C -> Prop,
  z: C -> Stream E -> Prop
```

- The distribution law of relation composition over unions.

```
Rels_concat_union_distr_r:
  forall x1 x2 y,
    (x1 ∪ x2) ∘ y == x1 ∘ y ∪ x2 ∘ y;
Rels_concat_union_distr_l:
  forall x y1 y2,
    x ∘ (y1 ∪ y2) == x ∘ y1 ∪ x ∘ y2;
Rels_concat_indexed_union_distr_r:
  forall xs y,
    ⋃ xs ∘ y == ⋃ (fun i => (xs i) ∘ y);
Rels_concat_indexed_union_distr_l:
  forall x ys,
    x ∘ ⋃ ys == ⋃ (fun i => x ∘ (ys i)).
```

Similar to the associativity of relation composition, all different variants of relation composition have these distribution laws.

2.3 Tactic support

We provide “`Sets_unfold`” and “`Sets_unfold in ...`” for users to unfold set related definitions into a logical proposition about the membership relation. The lower-case version `sets_unfold` has a similar functionality, but treats sets as ordinary curried Coq functions and does not generate "∈" in propositions. For example, X ⊆ Y ∪ Z is transformed to the following two propositions by Sets_unfold and sets_unfold respectively.

```
forall a b, (a, b) ∈ X -> (a, b) ∈ Y \/ (a, b) ∈ Z
forall a b, X a b -> Y a b \/ Z a b
```

Mainly, Sets_unfold is designed for teaching scenarios and sets_unfold is designed for potential professional users. These two are both zero-cost in the sense of Coq proof term size.

We prove that Sets.union, Sets.intersect and Rels.concat preserve set equivalence and set inclusion. These statements are formalized using the morphism type classes in Coq's standard libraries. Thus, users can directly use Coq's built-in rewrite tactic to handle set equivalence and set inclusion in proofs.

Application examples of these tactics can be found in the next section.

3 Applications in teaching

This set library is designed and used in teaching three different undergrad courses (for three distinct classes of students) about programming language theory in Shanghai Jiao Tong University, one of the top universities in China.

Course ID	No. of students	Year and semester	Knowledge about set theory	Programming experiences
CS2612	about 40	3rd year, fall	Yes	2 years CS major courses
CS2205	about 25	2nd year, fall	No	1 year CS major courses
CS2206	about 10	2nd year, spring	No	more than 4 years

Student backgrounds are slightly different in these three courses. Students in CS2612 have learnt set theory seriously. Thus, they understand that a binary relation R from A to B is actually a subset of the Cartesian product $A \times B$, and notations aRb or $(a,b) \in R$ can be used to talk about R's elements. They have also learnt that binary relation composition has associativity and the distribution law over unions. Students in CS2205 and CS2206 have not learnt anything about binary relations but they have learnt the semantic theory of first order logic in a logic course. Thus, they have a brief understanding of relations. For all of these three courses, students have no prior knowledge about the Coq proof assistant.

The purpose of using Coq in these three courses are not making students Coq experts but helping students understand definitions and proofs.

For understanding binary relations. In CS2205 and CS2206, the instructor taught basic concepts about binary relations and the composition operation on the blackboard. Some diagrams like figure 1 were used for describing the intuition behind. Properties like composition's associativity and distribution law over unions were first introduced with the help of such diagrams, then formally proved using Coq. The instructor would type a Coq proof for one theorem like the following in class. This proof would only use binary relation composition's definition just taught on the blackboard, and all implementation of our set library is hidden.

```
Theorem Rels_concat_assoc:
  forall
```

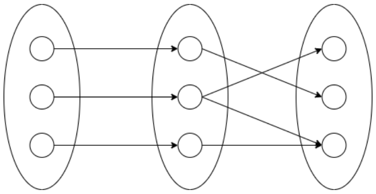


Figure 1: Diagrams for binary relation composition

```
    (A B C D: Type)
    (R1: A -> B -> Prop) (R2: B -> C -> Prop) (R3: C -> D -> Prop),
  (R1 ∘ R2) ∘ R3 == R1 ∘ (R2 ∘ R3).
Proof.
  intros. Sets_unfold. intros a d.
  (** Now, the proof goal is:
       (exists c : C,
         (exists b : B, (a, b) ∈ R1 /\ (b, c) ∈ R2) /\
         (c, d) ∈ R3) <->
       (exists b : B,
         (a, b) ∈ R1 /\
         (exists c : C, (b, c) ∈ R2 /\ (c, d) ∈ R3)) *)
  (** The rest is a FOL proof. *)
  split.
  + intros [c [[b [? ?]] ?]].
    exists b.
    split.
    - tauto.
    - exists c. tauto.
  + intros [b [? [c [? ?]]]].
    exists c.
    split.
    - exists b. tauto.
    - tauto.
Qed.
```

Students were then asked to formally prove another theorem by themselves. We find that Coq helps students understand how to really prove something using math definitions.

In CS2612, we use Coq to help students review basic concepts about binary relations that students had already learnt. The class would skim those important properties very quickly and the instructor would re-explain the intuition on the blackboard. Then, Coq was used to demonstrate a formal proof of binary relation composition's associativity so that students could establish a connection between intuitive understanding and strict math proofs during the review.

For defining denotational semantics. In a typical textbook denotational semantics, a program command's denotation is a binary relation from initial program states to ending states, i.e. $[\![c]\!] \subseteq \text{state} \times \text{state}$.

Then,

$$\begin{aligned}
\llbracket c_1; c_2 \rrbracket &= \llbracket c_1 \rrbracket \circ \llbracket c_2 \rrbracket \\
\llbracket \text{if } (e) \text{ then } c_1 \text{ else } c_2 \rrbracket &= \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c_1 \rrbracket \cup \text{test\_false}(\llbracket e \rrbracket) \circ \llbracket c_2 \rrbracket
\end{aligned}$$

where expressions are assumed to be side-condition-free and

$$\begin{aligned}
(s_1, s_2) \in \text{test\_true}(\llbracket e \rrbracket) \quad &\text{iff.} \quad s_1 = s_2 \text{ and } e \text{ is true on } s_1; \\
(s_1, s_2) \in \text{test\_false}(\llbracket e \rrbracket) \quad &\text{iff.} \quad s_1 = s_2 \text{ and } e \text{ is false on } s_1.
\end{aligned}$$

Then, $(c_1; c_2); c_3$ and $c_1; (c_2; c_3)$ are semantically equivalent because

$$\llbracket (c_1; c_2); c_3 \rrbracket = (\llbracket c_1 \rrbracket \circ \llbracket c_2 \rrbracket) \circ \llbracket c_3 \rrbracket = \llbracket c_1 \rrbracket \circ (\llbracket c_2 \rrbracket \circ \llbracket c_3 \rrbracket) = \llbracket c_1; (c_2; c_3) \rrbracket .$$

Also, (if (e) then c_1 else c_2); c_3 and if (e) then $(c_1; c_3)$ else $(c_2; c_3)$ are semantically equivalent because

$$\begin{aligned}
& \llbracket (\text{if } (e) \text{ then } c_1 \text{ else } c_2); c_3 \rrbracket \\
= \; & (\text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c_1 \rrbracket \cup \text{test\_false}(\llbracket e \rrbracket) \circ \llbracket c_2 \rrbracket) \circ \llbracket c_3 \rrbracket \\
= \; & \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c_1 \rrbracket \circ \llbracket c_3 \rrbracket \cup \text{test\_false}(\llbracket e \rrbracket) \circ \llbracket c_2 \rrbracket \circ \llbracket c_3 \rrbracket \\
= \; & \llbracket \text{if } (e) \text{ then } (c_1; c_3) \text{ else } (c_2; c_3) \rrbracket
\end{aligned}$$

In the third line of this proof above, the associativity of binary relation composition is implicitly applied! A Coq proof used in class (see below) reminds everyone this fact.

```
Lemma if_seq: forall e c1 c2 c3,
  ⟦ (if e then c1 else c2); c3 ⟧ ==
  ⟦ if e then (c1; c3) else (c2; c3) ⟧.
Proof.
  intros. simpl; unfold seq_sem, if_sem.
  (** Now, the proof goal is:
          (test_true ⟦ e ⟧ ∘ ⟦ c1 ⟧ ∪ test_false ⟦ e ⟧ ∘ ⟦ c2 ⟧) ∘ ⟦ c3 ⟧
          == test_true ⟦ e ⟧ ∘ (⟦ c1 ⟧ ∘ ⟦ c3 ⟧) ∪
             test_false ⟦ e ⟧ ∘ (⟦ c1 ⟧ ∘ ⟦ c3 ⟧) *)
  rewrite Rels_concat_union_distr_r.
  rewrite !Rels_concat_assoc.
  (** This line above uses the associativity. *)
  reflexivity.
Qed.
```

For denotational semantics of more practical languages. Using Coq also allows us to discuss more practical language features in these programming language courses. Specifically, when we present a realistic semantics with many subtleties involved, we can check line by line whether the original concise theory is broken or not. In these course, we provide extended and optional reading material about nondeterministic programming languages and program languages with observable behaviors (like IO events) for interested students.

For nondeterminism, we follow Back's approach [4], i.e.

- $\llbracket c \rrbracket.(\text{nrm})$ is a binary relations between program states; $(s_1, s_2) \in \llbracket c \rrbracket.(\text{nrm})$ iff. executing c from the initial state s_1 may terminate on s_2;
- $\llbracket c \rrbracket.(\text{inf})$ is a subset of program states; $s \in \llbracket c \rrbracket.(\text{inf})$ iff. executing c from the initial state s may not terminate.

Then,

$$\begin{aligned}
\llbracket c_1; c_2 \rrbracket.(\text{inf}) &= \llbracket c_1 \rrbracket.(\text{inf}) \ \cup\ \llbracket c_1 \rrbracket.(\text{nrm}) \circ \llbracket c_2 \rrbracket.(\text{inf}) \\
\llbracket \text{if } (e) \text{ then } c_1 \text{ else } c_2 \rrbracket.(\text{inf}) &= \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c_1 \rrbracket.(\text{inf}) \ \cup\ \text{test\_false}(\llbracket e \rrbracket) \circ \llbracket c_2 \rrbracket.(\text{inf})
\end{aligned}$$

Our set library reloads the "$\circ$" operator, which makes this definition above natural and intuitive.

For programs with observable behaviors, $\llbracket c \rrbracket \subseteq \text{state} \times \text{list\_of\_events} \times \text{state}$. Specifically, $(s_1, \tau, s_2) \in \llbracket c \rrbracket$ if and only if executing c from s_1 will generate event trace τ and terminate in state s_2. Again, we have $\llbracket c_1; c_2 \rrbracket = \llbracket c_1 \rrbracket \circ \llbracket c_2 \rrbracket$ according to the meaning of our reloaded "$\circ$".

Applying fixed point theorems. The fixed point theorem part is another place where students significantly benefit from using Coq. For example, in order to prove $f(x) = x$ on a partial order, it suffices to show that $f(x) \leq x$ and $x \leq f(x)$. But usually, students will not ask themselves why. It easily happen that they forgot they are working on a partial order not on real number's ordering. Coq forces students to answer such questions in every step in their homework proofs. For example, the answer to the question above is anti-symmetry.

The set library is used when proving $(P(\text{state} \times \text{state}), \subseteq)$ is a complete lattice, i.e. it is a partial order and every set of binary relations has a least upper bound (in a set theory sense). The instructor would check some of these properties in class, and others are left as homework — this is a traditional design in a programming language theory course but now it is done in Coq. For example, the following is the Coq proof of $\subseteq$'s anti-symmetry.

```
Theorem relation_inclusion_antisymm:
  forall (x y: state -> state -> Prop),
    x ⊆ y -> y ⊆ x -> x == y.
Proof.
  Sets_unfold. intros.
  (** Now, the proof goal is
         x, y : state -> state -> Prop
         H : forall a a0 : state, (a, a0) ∈ x -> (a, a0) ∈ y
         H0 : forall a a0 : state, (a, a0) ∈ y -> (a, a0) ∈ x
         a, a0 : state
         ____________________________________________
         (a, a0) ∈ x <-> (a, a0) ∈ y *)
  specialize (H a a0).
  specialize (H0 a a0).
  tauto.
Qed.
```

Summary and evaluation. Overall, using the Coq proof assistant helps us clarify potential ambiguity in natural language description, and emphasize important proof steps that may be ignored. Also, it allows us to extend typical toy languages to more realistic settings in theory courses — using the proof assistant gives students confidence that we do cover all subtle cases in proofs.

On the negative side, however, using Coq necessitate dedicating a few lectures to instructing students how to use the proof assistant itself. We try to minimize it in our course design. We teach students Coq inductive type when introducing programming language AST. We teach students structural recursive function and structure inductive proof when teaching denotational semantics and its theory. We teach students induction proofs over natural numbers in Coq when teaching Kleen fixed point theorem[4]. Coq proofs about logic (conjunction, disjunction, implication, negation and quantifiers) are taught separately. We choose not to teach students Curry-Howard correspondence, and not to teach students Coq's inductively defined propositions — we find that they are unnecessary for the courses and hard for students to understand.

Designing the set library is one of the effort for compressing the time spending on Coq itself. We use this library to hide how set-related operators are implemented and encoded as Coq definitions. Students find that this library is very easy to use and related Coq proofs that we demonstrated in classes are very easy to understand.

4 Implementation of the library: unified definitions

4.1 Set operators

We define union, intersection, empty set, full set, general union, general intersection, set equivalence and set inclusion using Coq's type class system. The main idea is, if `T` is a Coq type of form `A -> B -> ... -> Prop`, then we can construct a instance of type class `Sets.SETS T`.

```
Module Sets.
  Class SETS (T: Type): Type := {
    equiv: T -> T -> Prop;
    included: T -> T -> Prop;
    empty: T;
    full: T;
    union: T -> T -> T;
    indexed_union: forall {I: Type}, (I -> T) -> T;
    ...
  }.
End Sets.
```

Constructing instances of `Sets.SETS` only needs the following conclusions.

```
Instance Prop_SETS: Sets.SETS Prop.
Instance lift_SETS A B: Sets.SETS B -> Sets.SETS (A -> B).
```

We put some primary properties of these set operators in another type class beside `Sets.SETS` and derived other properties from primary properties. Overall, defining these operators and proving these properties is not hard. Researchers have achieved that when formalizing shallowly embedded assertion languages.

---

[4] We teach students general Coq inductive type quite early but the fact that natural numbers are defined as a specific inductive type in Coq are postponed to the first time that we need an induction proof over natural numbers.

4.2 Compositions

We formalize general composition using Coq type class Rels.RELS and an auxiliary type class Rels.RELS.

```
Module Rels.
  Class PRE_RELS (A T1 T2 T: Type): Type :=
  { concat_aux: T1 -> T2 -> A -> T }.
  Class RELS (T1 T2 T: Type): Type :=
  { concat: T1 -> T2 -> T }.
End Rels.
```

For example, the ordinary composition comp22 can be defined by an instance R2 of

```
Rels.RELS (A -> B -> Prop) (B -> C -> Prop) (A -> C -> Prop)
```

which is constructed by an instance R1 of

```
Rels.PRE_RELS B (B -> Prop) (C -> Prop) (C -> Prop).
```

To be more specific,

```
@concat_aux _ _ _ _ R1 :=
  fun (X: B -> Prop) (Y: C -> Prop) =>
    fun (b: B) (c: C) => X b /\ Y c.
@concat _ _ _ R2 :=
  fun (X: A -> B -> Prop) (Y: B -> C -> Prop)
    fun (a: A) =>
      Sets.indexed_union (fun b => concat_aux (X a) (Y b) b)
 (* which equals to
  fun (X: A -> B -> Prop) (Y: B -> C -> Prop)
    fun (a: A) (c: C) =>
      exists b: B, concat_aux (X a) (Y b) b *)
```

Similarly, comp21 can be defined by an instance R2' of

```
Rels.RELS (A -> B -> Prop) (B -> Prop) (A -> Prop)
```

which is constructed by an instance R1' of

```
Rels.PRE_RELS B (B -> Prop) Prop Prop.
```

We prove related properies based on this type class system.

4.3 The membership relation

We define our general membership relation by two type classes.

```
Module SetsEle.
  Class PRE_SETS_ELE (T RES E: Type): Type := ...
  Class SETS_ELE (T E: Type): Type :=
  { In: E -> T -> Prop;
    set_transfer: T -> T; }.
End SetsEle.
```

As a result, In1 and In2 mentioned in §1 are equivalent to

```
@SetsEle.In (A -> Prop) A SE1
@SetsEle.In (A -> Prop) (A * B) SE2
```

respectively, where SE1 and SE2 are some instances. We omit details here since the main techniques are the same as defining Rels.concat — SetsEle.In in the type class SETS_ELE is the definition behind the notation of "∈" and instances of PRE_SETS_ELE are auxiliaries used for building instances of SETS_ELE. The function SetsEle.set_transfer has nothing to do with the implementation of SetsEle.In. It is used in the unfolding tactics (see §5).

5 Implementation of the library: the unfolding tactic

We have mentioned in §1 that simply unfolding Coq definitions of set equivalence and set inclusion cannot generate intuitive propositions. For example, if X Y Z: A -> B -> Prop, then unfolding X ⊆ Y ∪ Z results in

```
forall a b, X a b -> Y a b \/ Z a b.
```

Interestingly, unfolding is not too far way from our final solution! We only need to replace X, Y and Z with Coq terms like (fun a b => (a, b) ∈ X) first, then unfolding the definitions of ⊆ and ∪ in

```
(fun a b => (a, b) ∈ X) ⊆
  (fun a b => (a, b) ∈ Y) ∪ (fun a b => (a, b) ∈ Z)
```

will generate the following proposition.

```
forall a b, (a, b) ∈ X -> (a, b) ∈ Y \/ (a, b) ∈ Z.
```

The key in this transformation is turning Coq variables like X into (fun a b => (a, b) ∈ X). It is defined by SetsEle.set_transfer mentioned previously. The effect of SetsEle.set_transfer is:

- If X: A -> Prop, then SetsEle.set_transfer X can be unfolded to

  ```
  fun a => a ∈ X
  ```

- If X: A -> B -> Prop, then SetsEle.set_transfer X can be unfolded to

  ```
  fun a b => (a, b) ∈ X
  ```

- If X: A -> B -> C -> Prop, then SetsEle.set_transfer X can be unfolded to

  ```
  fun a b c => (a, b, c) ∈ X
  ```

- ...

It is worth mentioning, SetsEle.set_transfer X is $\beta\iota\delta$-equivalent to X for every instance of SetsEle.set_transfer. That means, we can always

```
change X with (SetsEle.set_transfer X)
```

in Coq to complete the transformation as long as X is a set. We omit the implementation of SetsEle.set_transfer in this paper, and it uses the same technique used in the definition of Rels.concat.

Our tactic Sets_unfold is an Ltac program in Coq. It first marks all Coq variables which SetsEle.set_transfer should apply on, then uses a change command to insert those SetsEle.set_transfer, and finally unfold the definition of SetsEle.set_transfer and other set-related definitions like unions and intersections. Sets_unfold is very carefully implemented so that it causes zero proof-term overhead, i.e. it only replaces a Coq expression with a $\beta\iota\delta$-equivalent one, and its implementation cannot even include one single rewrite. For example, consider the following proof in Coq.

```
Example Sets_union_comm_included:
  forall (A: Type) (X Y: A -> Prop),
    X ∪ Y ⊆ Y ∪ X.
Proof.
  intros. Sets_unfold.
  intros. tauto.
Qed.
```

Its proof term in Coq is:

```
fun (A : Type) (X Y : A -> Prop) (a : A) (H : a ∈ X \/ a ∈ Y) =>
  or_ind
    (fun H0 : a ∈ X => or_intror H0)
    (fun H0 : a ∈ Y => or_introl H0) H.
```

There is no redundancy compared to a manual proof.

6 Related work

Formalization of sets. In Coq standard library, Ensemble X is used to represent all subsets of X, and is defined as X -> Prop [11]. In Lean [13] standard library, set X is defined as X -> Prop [12]. In Isabelle/HOL, set X is isomorphic with X -> bool [16]. These three definitions are very similar and none of them can provide a unified definition of unions and intersections for sets and relations as we proposed in our set library. Besides, Coq standard library provides efficient implementations (based on lists and/or trees) of finite sets in the FSet module [8] and the MSet module [9]. Since sets used in denotational semantics are not necessarily finite, FSet and MSet are not useful for us.

As mentioned in §4.1, defining set intersection and set union is like defining conjunction and disjunction in shallowly embedded assertion languages. Also, defining indexed set intersection and indexed set union (Sets.indexed_union and Sets.indexed_intersect in our set library) is like defining existential quantifier and universal quantifier. Specifically, an assertion over program states can be treated a subset of program states. If a program state is a pair of stack and heap, then assertions can be defined as stack -> heap -> Prop in Coq. The existing works that provide unified formalization of conjunction, disjunction and quantifiers for assertion languages include the MSL library in VST [2] and the ModuRes library [19] used in the Iris project [6]. In comparison, our formalization of sets also provide definitions and proofs about relation compositions "$\circ$", and the membership relation "$\in$".

Formalization of denotational semantics. In many textbook about program semantics (like Winskel's *Formal Semantics for Programming Languages: An Introduction* [21]), an imperative program's denotation is defined as a binary relation between program states. Nipkow's formalization of denotational semantics follows this approach [14]. A typical extension of such textbook denotational semantics is to additionally consider possible errors like null-pointer dereference, so a program's denotation becomes a binary relation between state $\cup \{\perp\}$, where $\perp$ represents errors. Imperative-HOL [5] in Isabelle adopts this approach.

Our set library is used in teaching typical textbook denotations. For potential program errors, our formalization is more close to Park's denotational semantics [17], dividing a program's denotation into several fields (like $[\![c]\!]$.(nrm), $[\![c]\!]$.(err) and $[\![c]\!]$.(err)). Although Park's original work is to find a theory for defining nondeterministic programs behavior including possible nontermination which fails main stream power domain theory before that [20, 18], we choose not to go that far in our introductory courses.

Comparison between denotational semantics and big step operational semantics. The *Software Foundations* [1] book teaches big step operational semantics instead of denotational semantics. Leroy's Mechanized Semantics course [7] also chooses to teach big step operational semantics instead of denotational semantics. In comparison, denotational semantics and big step semantics are similar — we say a program's denotation is a binary relation between program states, or a program evaluates to a binary relation between program states. The main difference is: a program's denotation should be defined by a recursion over programs' syntax tree but a program's big step operational semantics is usually formalized by an inductive proposition, i.e. a rule-based definition in math. In our courses, we choose not to include Coq inductive propositions.

Using proof assistant in teaching semantics. Interactive theorem provers are useful for teaching basic logical concepts. Avigad shared his experience of teaching logics and set theories using Lean [3] in 2019. His course material directly uses Lean's internal formalization of sets. Students were taught to use Lean proving simple properties of sets.

Software Foundations [1] is a famous textbook for teaching students to use the Coq theorem prover and for teaching basic programming language theories based on Coq. Its material contains big step operational semantics, small step operational semantics and Hoare logic. However, using *Software Foundations* as a textbook means spending a lot of time teaching students to use Coq and then we have to cut down a decent portion of programming language theories from syllabus.

Nipkow shared his teaching experience in VMCAI 2012 [15]. He taught program semantics courses with the help of the Isabelle proof assistant. His major points include

- *"Teach Semantics, Not Proof Assistants."*
- *"Teach Proofs, Not Proof Scripts."*
- *"Do Not Let the Proof Assistant Dominate Your Presentation."*

We adopt a similar philosophy in the preparation of our course material.

7 Conclusion

This paper describes a Coq library of sets and relations for teaching denotational semantics. This library provides unified definitions of set operators, intuitive notation for the membership relation and automated tactics. At the same time, we keep our definitions compatible with the existing formal definitions in Coq standard libraries. With this set library, we can use the Coq proof assistant to help students understand denotational semantics, without spending too much time teaching how to use the theorem prover itself. Also, some components of this library (like the unified definitions of set operators) are also useful for professional users of Coq when building denotational semantics.

References

[1] Benjamin C. Pierce et. al: *Software foundations*. Available at `https://softwarefoundations.cis.upenn.edu/`.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy & Xavier Leroy (2014): *Program Logics - for Certified Compilers*. Cambridge University Press, doi:10.1017/CBO9781107256552. Available at `http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB`.

[3] Jeremy Avigad (2019): *Learning logic and proof with an interactive theorem prover*. In: Proof Technology in Mathematics Research and Teaching, Springer, pp. 277–290, doi:10.1007/978-3-030-28483-1_13.

[4] Ralph-Johan Back (1983): *A Continuous Semantics for Unbounded Nondeterminism*. Theor. Comput. Sci. 23, pp. 187–210. Available at `https://doi.org/10.1016/0304-3975(83)90055-5`.

[5] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök & John Matthews (2008): *Imperative Functional Programming with Isabelle/HOL*. In: TPHOLs 2008, Proceedings, Springer, pp. 134–149. Available at `https://doi.org/10.1007/978-3-540-71067-7_14`.

[6] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal & Derek Dreyer (2018): *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*. J. Funct. Program. 28, p. e20. Available at `https://doi.org/10.1017/S0956796818000151`.

[7] Xavier Leroy: *Mechanized Semantics Course, https://xavierleroy.org/CdF/2019-2020/*. Available at `https://xavierleroy.org/CdF/2019-2020/`.

[8] Coq standard library: *FSets*. Available at `https://coq.inria.fr/doc/V8.18.0/stdlib/Coq.FSets.FSets.html`.

[9] Coq standard library: *MSet*. Available at `https://coq.inria.fr/doc/V8.18.0/stdlib/Coq.MSets.MSets.html`.

[10] Coq standard library: *Relation definitions*. Available at `https://coq.inria.fr/doc/V8.18.0/stdlib/Coq.Relations.Relation_Definitions.html`.

[11] Coq standard library: *Sets.Ensemble in Coq*. Available at `https://coq.inria.fr/doc/V8.18.0/stdlib/Coq.Sets.Ensembles.html`.

[12] Lean standard library: *Sets in Lean*. Available at `https://leanprover.github.io/logic_and_proof/sets_in_lean.html`.

[13] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In: CADE 2015, Proceedings, Springer, pp. 378–388. Available at `https://doi.org/10.1007/978-3-319-21401-6_26`.

[14] Tobias Nipkow (1996): *Winskel is (Almost) Right: Towards a Mechanized Semantics Textbook*. In: Foundations of Software Technology and Theoretical Computer Science, 1996, Proceedings, Springer, pp. 180–192. Available at `https://doi.org/10.1007/3-540-62034-6_48`.

[15] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In: VMCAI 2012, Proceedings, Lecture Notes in Computer Science 7148, Springer, pp. 24–38. Available at `https://doi.org/10.1007/978-3-642-27940-9_3`.

[16] Tobias Nipkow, Lawrence C Paulson & Markus Wenzel: *Sets in Isabelle/HOL*. Available at `https://isabelle.in.tum.de/library/HOL/HOL/Set.html`.

[17] David Michael Ritchie Park (1979): *On the Semantics of Fair Parallelism*. In: Abstract Software Specifications, 1979, Proceedings, Springer, pp. 504–526. Available at `https://doi.org/10.1007/3-540-10007-5_47`.

[18] Gordon D. Plotkin (1976): *A Powerdomain Construction*. SIAM J. Comput. 5(3), pp. 452–487. Available at `https://doi.org/10.1137/0205035`.

[19] Filip Sieczkowski, Ales Bizjak & Lars Birkedal (2015): *ModuRes: A Coq Library for Modular Reasoning About Concurrent Higher-Order Imperative Programming Languages*. In: *ITP 2015*, *Proceedings*, Springer, pp. 375–390. Available at `https://doi.org/10.1007/978-3-319-22102-1_25`.

[20] Michael B. Smyth (1978): *Power Domains*. *J. Comput. Syst. Sci.* 16(1), pp. 23–36. Available at `https://doi.org/10.1016/0022-0000(78)90048-X`.

[21] Glynn Winskel (1993): *The formal semantics of programming languages - an introduction*. Foundation of computing series, MIT Press, doi:10.7551/mitpress/3054.001.0001.

Waterproof: Educational Software for Learning How to Write Mathematical Proofs

Jelle Wemmenhove* Dick Arends Thijs Beurskens Maitreyee Bhaid
Sean McCarren Jan Moraal Diego Rivera Garrido David Tuin
Malcolm Vassallo Pieter Wils Jim Portegies†
Eindhoven University of Technology
Eindhoven, The Netherlands
*`a.j.wemmenhove@tue.nl` †`j.w.portegies@tue.nl`

In order to help students learn how to write mathematical proofs, we adapt the COQ proof assistant into an educational tool we call *Waterproof*. Like with other interactive theorem provers, students write out their proofs inside the software using a specific syntax, and the software provides feedback on the logical validity of each step. Waterproof consists of two components: a custom proof language that allows formal, machine-verified proofs to be written in a style that closely resembles handwritten proofs, and a custom editor that allows these proofs to be combined with formatted text to improve readability. The editor can be used for COQ documents in general, but also offers special features designed for use in education. Student input, for example, can be limited to specific parts of the document to prevent exercises from being accidentally deleted. Waterproof has been used to supplement teaching the Analysis 1 course at Eindhoven University of Technology (TU/e) for the last four years. Students started using the specific formulations of proof steps from the custom proof language in their handwritten proofs; the explicit phrasing of these sentences helped to clarify the logical structure of their arguments.

1 Introduction

Many first-year mathematics undergraduate students struggle with learning how to write mathematical proofs. It is a new skill they often never encountered before, one they nonetheless need to master as one of the key mathematical competencies. Although students may have difficulty solving the puzzle at the core of many proofs, this is not the main issue [14, 21]. During various proof stages, students seem to be insufficiently aware what is required of them, and which steps they are, and are not, allowed to perform. Students will try to prove $\forall$-statements without introducing a variable; they are quick to assign extra properties to variables obtained from $\exists$-statements; or they swap the order between quantifiers, e.g. in ε-δ proofs. Struggling with new concepts is a natural part of the learning process, but too often the confusion about the mechanics underlying mathematical proofs persist beyond the training phase and hinders students' performance in later courses.

The learning process for mathematical proof writing can potentially be improved by the use of proof assistants [11, 22]. Such computer programs, like LEAN [15] or COQ [7], allow users to construct proofs step-by-step whilst the program continuously provides feedback on the logical validity of these steps. For example, if a user tries to show some statement involving a variable which has not been introduced, the system will throw an error. Thus, proof assistants could serve as a training environment for students to freely explore which actions at various stages of a proof are logically allowed. Additionally, they actively track the available variables, hypotheses, and proof objectives, which can help students that are unsure about what statements they can use or what needs to be shown. For some accounts from

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 96–119, doi:10.4204/EPTCS.400.7

© A.J. Wemmenhove et al.
This work is licensed under the
Creative Commons Attribution License.

teachers who have used proof assistants in their courses with the explicit aim to get students to produce better handwritten proofs, see the accounts by Nipkow [16], Patrick Massot [11, §2.2], and Frédéric Le Roux [11, §2.3]; Patrick Massot and Heather MacBeth also discuss their personal experiences with using LEAN to teach proof writing in the context of mathematical analysis on a panel *Teaching with proof assistants*[1] at the Lean Together 2021 meeting.

Despite their potential benefits, there are some issues with the current generation of proof assistants that hinder their integration into conventional proof writing courses.

(I) **Being able to write proofs in a proof assistant does not imply being able to write good-quality proofs by hand.** Although Thoma and Iannone [22] found that students who partook in a workshop on LEAN produced better handwritten proofs, a study by Knobelsdorf et al. [12] found that students in a small course involving COQ performed worse at writing proofs with pen and paper than with COQ itself. The members of the LEAN panel attest to a similar statement: students were able to master creating proofs with the proof assistant, but their handwritten proofs left much to be desired. Knobelsdorf et al. suggest that, in their case, the transfer of proof skills might have failed because COQ provides additional scaffolding, like the automated bookkeeping overview, which continuously displays the available assumptions and current proof goals, that was not carefully dismantled during the course.

(II) **Proof assistants have a steep learning curve.** It takes time to learn the specific proof language used by a proof assistant. Böhne and Kreitz [3], for example, remark that COQ's syntax is difficult to learn for students due to the unstructured naming convention of the keywords that indicate certain proof steps. Additionally, the foundational system used by most proof assistants, namely type theory, differs enough from the usual set theory to require some explanation. Propositions and subsets in particular are treated differently.

(III) **The feedback provided by proof assistants is reactive and limited to isolated proof steps.** Students need other kinds of feedback as well: students might get stuck and need a hint in order to continue with a proof; other students might produce technically correct proofs that are too complicated. Think of students neglecting to use lemmas and trying to derive everything from first principles.

(IV) **Installing a proof assistant is difficult.** The installation instructions can be involved, especially for non-LINUX platforms. They often require users to use the command line, but most mathematics freshmen have never used the command line before. At our institution, students use personal laptops instead of fixed computers in a computer lab, so pre-installing the proof assistants is not an option. Luckily, the developers behind most proof assistants are realizing the importance of having a simple installation procedure, see for example the development of COQ PLATFORM[2] which allows users to just install the binaries using a graphical installer.

(V) **The user interfaces for proof assistants can be uninviting.** Some editors, like the CoqIDE have a dated visual design, giving the impression that the program has not seen maintenance in a while. More modern editors exist, both COQ and LEAN provide extensions for VS CODE, but these interfaces strongly resemble coding environments. Although this might provide computer science students with a sense of familiarity, it can have the opposite effect on mathematics students.

---

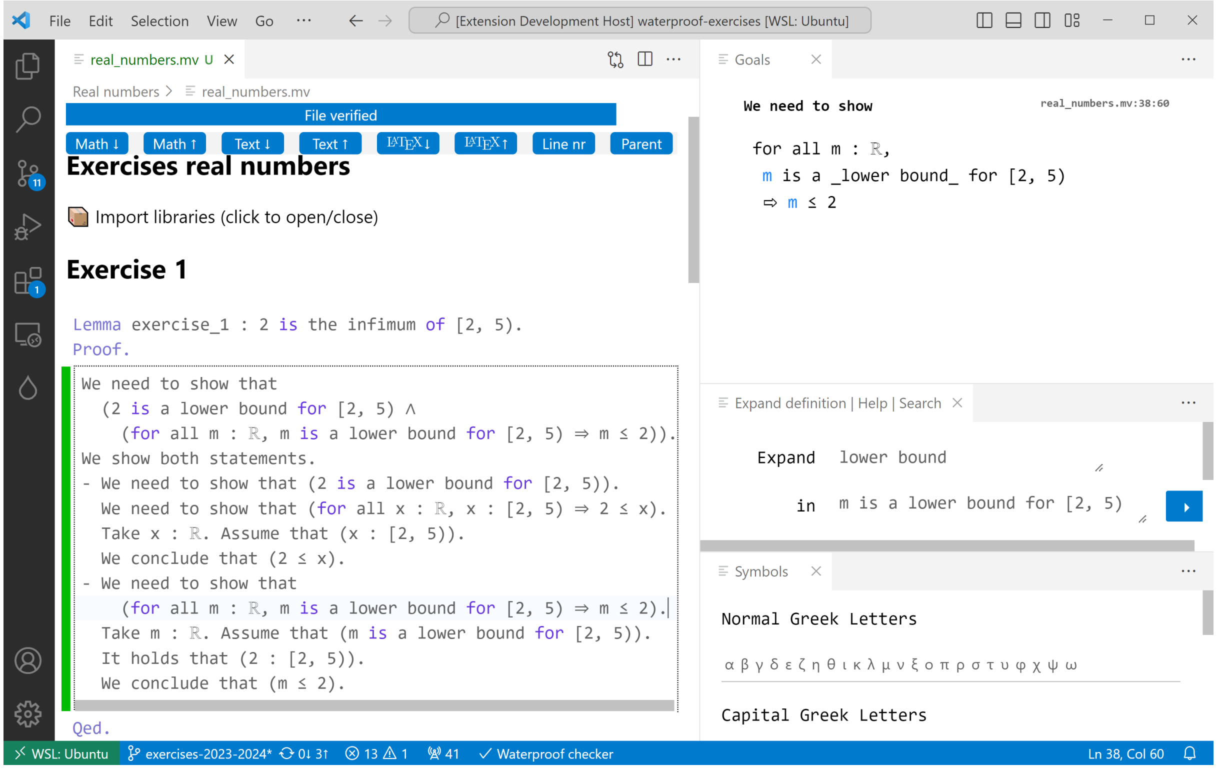
[1] Panel on teaching with proof assistants. Lean Together 2021. Panelists: Jasmin Blanchette, Jeremy Avigad, Julien Narboux, Heather Macbeth, Gihan Marasingha & Patrick Massot. Available at `https://leanprover-community.github.io/lt2021/schedule.html`.

[2] Available at https://github.com/coq/platform.

(VI) **Many proof assistants do not allow for LaTeX-formatted expressions.** Although proof assistants like COQ and LEAN allow for Unicode notation, this is a large downgrade from the beauty and versatility of LaTeX. This is not a superficial requirement either: good notation clarifies mathematical concepts and aids understanding.

To address the issues above, we created *Waterproof*, an adaptation of the COQ proof assistant that is designed specifically for teaching mathematical proof writing. A screenshot is shown in Figure 1. We felt that writing a proof in the existing proof assistants differs too much from the ordinary way of writing a proof, that this gap might explain the difficulty with transferring proof skills from proof assistants to pen-and-paper proofs (issue I), and that it and prevents student from tapping into prior knowledge when learning how to use a proof assistant (issue II). Hence, a key idea in Waterproof's design was that

> *Writing a proof in Waterproof should be as close as possible to writing a proof by hand, both in terms of the final product and the process of constructing the proof.*



Exercises real numbers

Import libraries (click to open/close)

Exercise 1

```
Lemma exercise_1 : 2 is the infimum of [2, 5).
Proof.
We need to show that
  (2 is a lower bound for [2, 5) ∧
    (for all m : ℝ, m is a lower bound for [2, 5) ⇒ m ≤ 2)).
We show both statements.
- We need to show that (2 is a lower bound for [2, 5)).
  We need to show that (for all x : ℝ, x : [2, 5) ⇒ 2 ≤ x).
  Take x : ℝ. Assume that (x : [2, 5)).
  We conclude that (2 ≤ x).
- We need to show that
    (for all m : ℝ, m is a lower bound for [2, 5) ⇒ m ≤ 2).
  Take m : ℝ. Assume that (m is a lower bound for [2, 5)).
  It holds that (2 : [2, 5)).
  We conclude that (m ≤ 2).
Qed.
```

```
We need to show                                real_numbers.mv:38:60

 for all m : ℝ,
  m is a _lower bound_ for [2, 5)
  ⇨ m ≤ 2
```

```
Expand  lower bound

    in  m is a lower bound for [2, 5)
```

Normal Greek Letters

α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω

Capital Greek Letters

Figure 1: Screenshot of the Waterproof editor showcasing Waterproof's custom proof language. The mixed document with formatted text and verified mathematics is shown on the left. The line prefixed by the box-emoji hides the document preamble that imports the required libraries. The vertical green bar indicates an input area where students can write their proof. The green color indicates that the proof is correct. On the right are multiple panels, from top to bottom they are: the automated bookkeeping overview, limited to only show the proof goal; a panel for expanding definitions, in other editors this is done within the main document; an overview of mathematical symbols, which can be inserted into the main document by clicking on them.

The Waterproof software[3] consists of two components: a custom proof language and a custom editor. The custom proof language allows COQ proofs to be written in a style that closely resembles the style of handwritten proofs. It is part of a custom COQ plugin that also contains subroutines that can give basic suggestions for the next proof step based on the main connector of the proof goal (first steps towards solving issue III). The custom editor provides a modern user interface (issue V) and uses *mixed documents* which beside verified COQ proofs also contain formatted text (including LaTeX-expressions (issue VI)). Both the custom editor and proof language are easy to install (issue IV). Some of the editor's features were designed around its use as an educational tool. Student input, for example, can be limited to specific parts of the document to prevent the accidental deletion of exercises. Features like an auto-complete function for mathematical symbols and proof steps also help to improve the usability of the editor in general. The amount of information shown in the automated bookkeeping panel is, by default, also restricted in order to make the proof construction process with Waterproof more closely resemble the process of constructing a proof with pen-and-paper.

Waterproof has been used to supplement the teaching of mathematical analysis at Eindhoven University of Technology (TU/e) for the last four years. Roughly 175 students register for the course, Analysis 1, every year; the majority are first-year undergraduate students. A selection of the weekly homework exercises have been made available as Waterproof exercise sheets, i.e. documents where the proofs are left to be filled out. Students can choose to hand in homework by hand or using Waterproof. The homework assignments written in Waterproof are graded automatically. Use of Waterproof has been optional as only a couple of instructors could answer questions about the program, naturally limiting its adoption by students. At the start of the 2022-2023 course, 25 student groups (≈ 100 students) handed in homework written in Waterproof; 19 groups (≈ 76 students) continued using it for the final assignment. We observed that students started using the specific formulations from Waterproof's proof language in their handwritten proofs; the explicit phrasing of these sentences helped to clarify the logical structure of their arguments.

Section 2 explores alternative solutions proposed by others to the problems I–VI outlined above. Waterproof's custom proof language and its implementation are discussed in Section 3, followed by a discussion of the custom editor in Section 4. Section 5 outlines how Waterproof was employed in the Analysis 1 course at the TU/e, and reports on both students' and teachers experience of using the software.

2 Related Work

As proof assistants have gained popularity among mathematicians, these programs have also found their way into mathematics education. The introduction already mentioned MacBeth and Massot who both used LEAN to teach mathematical analysis, as well as the others who spoke at the LEAN panel on teaching with proof assistants. The use of proof assistants in education is not a new phenomenon: communities who actively use these tools in their own research seem to have repeatedly attempted to harness their potential benefits for teaching purposes. The oldest example we found was the use of Mizar [1] to teach propositional logic at the University of Warsaw in the 70s, according to [19]. In computer science, proof

---

[3] Available at https://impermeable.github.io. The custom proof language is part of the *coq-waterproof* plugin, available at github.com/impermeable/coq-waterproof, this article pertains to version 2.1.0+8.17 . The custom editor is a VS CODE extension, available at marketplace.visualstudio.com/items?itemName=waterproof-tue.waterproof, this article pertains to version 1.0.0 . The editor can be installed directly from VS CODE's 'Extensions' panel, after which a walkthrough will guide the user on how to install the coq-waterproof plugin using a graphical installer.

assistants are often used to teach more advanced theoretical subjects like formal logic [10, 16] or type theory [13, §3]; students report this makes the courses feel more practical, as writing proofs in a proof assistants feels similar to coding. For some, proof assistants are merely a means to teach the theoretical contents of a course, whereas Nipkow, Massot, Le Roux, and MacBeth also used proof assistants to improve the quality of students' handwritten proofs.

Although some teachers have used existing proof assistants directly, many others saw the need to adapt these programs. Below is a list of tools and techniques that have been developed which would address (parts of) the problems I–VI discussed in the introduction.

Böhne and Kreitz (coauthors of the small COQ study [12] mentioned in Issue I) have developed a didactic method — opposed to a software solution — to explicitly guide the transition from COQ proofs to handwritten ones [3]. The gap between formal COQ proofs and informal textbook-style proofs is bridged by introducing three intermediate proof styles that gradually lower the level of formality. Students are first taught to write proofs in COQ, with custom, easier to learn tactics, and then to both translate between the different levels of formality, as well as develop proofs from scratch at each level. The authors mention that a solution like Waterproof, which modifies COQ itself to make the proofs be more like handwritten proofs, would be very interesting, but that such a development would require a large amount of preparatory work. The teaching method, on the other hand, is cheap and flexible: it is easy to adapt to different domains and can be adjusted as a course is being taught.

LURCH [6] markets itself as a word processor that offers proof verification as an additional service, like a spellchecker. It is a total solution with its own deduction engine. The system does not force a specific syntax on the users, instead semantic information is obtained by having users manually annotate mathematical expressions as 'claims', 'reasons' or 'premises'. The mathematical expressions can be written using LaTeX, which is dynamically formatted. As of now, proof checking is limited to logic and set theory; proofs have to be written out in full detail, as by design LURCH has no automation facilities. Earlier versions included a computer algebra system for verification as well, but this has been removed due to insufficient precision.

The DIPROCHE system [4, 5] is similar to LURCH: users can write out proofs in controlled natural language (German); the program parses the text and checks the proof using several of its own deduction engines. Unique to DIPROCHE is its ability to point out common mistakes and, in some cases, produce counterexamples. Currently, the DIPROCHE system supports exercises in propositional logic, set theory, elementary number theory, axiomatic geometry and elementary group theory; the parsers and deduction systems had to be adjusted to each domain. DIPROCHE runs on a remote web server, avoiding the problem of installation.

EDUKERA[4] [20] is a commercial web-application based on the COQ proof assistant that allows users to advance a proof using a click-based graphical interface. Pierre Guillot and Julien Narboux [11, §2.4] and Simon Modeste [11, §2.5] describe their experiences of using EDUKERA in their courses. They note that, due to its point-and-click interface, students are not required to memorize definitions and that some students managed to finish exercises without really understanding what they had to do [11, §2.4]. At the LEAN panel on teaching with proof assistants, Patrick Massot mentioned that he refrained from using EDUKERA in the past because even he himself started randomly pressing buttons out of frustration with the tool. Guillot and Narboux also mention that EDUKERA has not been maintained since 2020 and that teachers cannot add their create new exercises on their own.

---

[4] Available at https://edukera.com.

D∃∀DUCTION[5] [11, §2.3] is another click-based graphical interface for constructing proof terms step by step. It is built on top of the LEAN theorem prover and runs locally. Exercises can be created using LEAN itself, a custom parser allows teachers to specify per exercise which inference rules and definitions are available in the graphical interface. D∃∀DUCTION is not meant as a replacement for conventional exercises, it is a tool that students can use when writing proofs by hand, e.g. to check that the steps they wrote down are allowed or that they have the intended effects.

PROOFWEB [13] provides web access to a COQ proof assistant running on a central server and adds some additional features tailored to education. The user interface is similar to that of COQ itself, but the proof state can be displayed in different ways, like Gentzen's deduction trees or Fitch's flag-style proofs. The central server not only provides an easy way to access COQ, but also serves as a distribution point for exercises and allows teachers to keep track of students' progress. A parser checks whether students are cheating by using COQ's powerful automation procedures instead of the intended tactics; these tactics use a custom syntax that is easier to learn, like with Böhne and Kreitz in [3]. PROOFWEB use COQ version 8.2, which dates to 2009.

JSCOQ [9] is an adaptation of COQ that runs purely in the browser using JavaScript. Since JSCOQ requires no installation, it is often used in workshops to introduce people to COQ. JSCOQ uses mixed documents that combine executable COQ code with formatted text. The JSCOQ version of the Software Foundations series[6], for example, allows students to directly see the effects of executing the COQ code explained in these books.

LEAN VERBOSE[7] [11, §2.2] is a custom proof language for the LEAN theorem prover developed by Patrick Massot that, like Waterproof, mimics natural language formulations for tactics. Its goal is to make it easier for students to transition from proof assistants to pen-and-paper proofs. Like Waterproof (see Section 3.1), LEAN VERBOSE also implements a tactic that is able to provide students with suggestions on how to proceed if they are stuck. LEAN VERBOSE's help functionality is more advanced than the one in Waterproof: it is able to offer suggestions on how to use hypotheses and course-specific hints like how to expand certain definitions.

Comparing Waterproof to the solutions above, LURCH and DIPROCHE seem to be most similar: they provide their own editors and allow users to write proofs in a natural language. Both systems, however, use their own proof checkers, hence, they miss out on efforts by the larger community interested in formalizing mathematics. LEAN VERBOSE is similar to Waterproof's custom proof language, and was even developed with the same goal in mind, a comparison between the two systems should be interesting. Böhne and Kreitz provide the most radically different solution: instead of adapting proof assistants to fit education, they suggest a new method to incorporate the existing software into mathematics classes, one that is different from just having students write proofs in both the program and with pen-and-paper. D∃∀DUCTION is also interesting in this regard since it serves as a tool that students can use to check their reasoning when writing conventional proofs. Like Waterproof, JSCOQ allows for the combination of formatted text with COQ proofs in mixed documents; Waterproof was partially inspired by JSCOQ and we might try to use JSCOQ as a basis for a future version of Waterproof that runs completely in the browser.

---

[5] Developed by Frédéric Le Roux, Marguerite Bin, Florian Dupeyron & Antoine Leudière (2020). Available at https://perso.imj-prg.fr/frederic-leroux/d%E2%88%83%E2%88%80duction.

[6] Emilio Jesús Gallego Arias, Benoît Pin & Pierre Jouvelot (2017). Available at https://jscoq.github.io/ext/sf.

[7] Developed by Patrick Massot (2021). LEAN 3 version available at https://github.com/PatrickMassot/lean-verbose.

3 Custom Proof Language

Waterproof's custom proof language allows proofs to be written in a style that closely resembles handwritten proofs. To see how well the handwritten style is approximated, see the example proof shown in Figure 2. Figure 3 shows the same proof written in the default Coq language. By closing the gap between Waterproof's proof style and that of ordinary proofs, we expect that proof skills learned in Waterproof are transferred to pen-and-paper more easily (issue I). The Waterproof language is also easy to learn (issue II): for the Analysis 1 course, the entire language is explained in a single tutorial file, which takes the students only a couple of hours to complete. The custom proof language is part of the coq-waterproof plugin, which also contains the mathematical library used for the Analysis 1 course (a custom library based on the Coq standard library).

In this section, we often use the term *tactic*. This is proof-assistant terminology for a function that alters the state of the proof, like the subroutine underlying the proof step for introducing a new variable.

3.1 Features

The main features of Waterproof's custom proof language are listed below, most of these are exemplified by the proof in Figure 2. They are to be contrasted with the default Coq language in Figure 3.

- **Proof step formulations inspired by handwritten proofs.** The proof steps formulations in Waterproof's custom proof language take the form of full sentences that are used in ordinary mathematical proofs as well. Compare the formulations used by Waterproof's language (left) and Coq's default Coq syntax (right) for introducing the ε variable:

  ```
  Take ε : ℝ.
  ```
  vs.
  ```
  intro ε.
  ```

  The Waterproof formulation includes the mathematically relevant information that ε is a real number, whereas the default Coq tactic does not. The difference becomes even more apparent if we compare the formulations for introducing the assumption that $\varepsilon > 0$:

  ```
  Assume that (ε > 0).
  ```
  vs.
  ```
  intro ε_gt_0.
  ```

  The actual proposition itself is not even mentioned in the default Coq formulation. For Coq users, this is a non-issue, since the content of the assumption can easily be found in the automated bookkeeping overview. For educational purposes, however, we wish to limit the information shown there such that students learn to write proofs without it. To make the Coq proofs readable on their own, the relevant mathematical information has to be included back into the proof steps themselves. In fact, with Waterproof's proof step formulations, we found ourselves to pay less and less attention to the automated bookkeeping overview anyway.

  Like in ordinary proofs, Waterproof uses different formulations for introducing a variable and introducing an assumption. Default Coq uses the same tactic for both, because to its foundational system, type theory, there is no meaningful difference between the two. In Waterproof, an error is thrown if the wrong formulation is used, like for example in

  ```
  Take ε_gt_0 : (ε > 0).
  ```

- **Implicit use of automation to verify statements.** Basic statements do not need to be justified in ordinary proofs, but, by design, proof assistants require a proof for every claim that is made. To

```
Lemma example_coq_waterproof :
  for all ε : ℝ, ε > 0 ⇒ there exists a : ℝ, a : [0,4) ∧ 4 - ε < a.
Proof.
  Take ε : ℝ. Assume that (ε > 0).
  Either (ε < 2) or (ε ≥ 2).
  - Case (ε < 2).
    Choose a := (4 - ε/2).
    We show both (a : [0,4)) and (4 - ε < a).
    + We need to show that (0 ≤ a ∧ a < 4).
      We show both (0 ≤ a) and (a < 4).
      * We conclude that (& 0 < 4 - 1 < 4 - ε/2 = a).
      * We conclude that (a < 4).
    + We conclude that (4 - ε < a).
  - Case (ε ≥ 2).
    Choose a := 3.
    We show both (3 : [0,4)) and (4 - ε < 3).
    + We conclude that (3 : [0,4)).
    + We conclude that (& 4 - ε ≤ 4 - 2 = 2 < 3).
Qed.
```

Figure 2: Proof of $\forall \varepsilon > 0 \; \exists a \in [0,4), 4 - \varepsilon < a$ written using Waterproof's custom proof language.

```
Lemma example_coq :
  forall ε : R, ε > 0 -> exists a : R, ([0,4) a) /\ 4 - ε < a.
Proof.
  intro ε. intro ε_gt_0.
  assert (ε < 2 \/ 2 <= ε) as cases by lra.
  destruct cases as [ε_lt_two | two_le_ε].
  - (* Case ε < 2. *)
    exists (4 - ε/2).
    split.
    + split.
      * assert (0 <= 4 - ε/2) as h1 by lra; exact h1.
      * assert (4 - ε/2 < 4) as h2 by lra; exact h2.
    + assert (4 - ε < 4 - ε/2) as h3 by lra; exact h3.
  - (* Case ε ≥ 2. *)
    exists 3.
    split.
    + assert (0 <= 3 < 4) as h4 by lra; exact h4.
    + assert (4 - ε < 3) as h5 by lra; exact h5.
Qed.
```

Figure 3: Proof of the same statement written using Coq's default proof language (written in such a way that shows to what extent a pen-and-paper proof style can be replicated).

get Waterproof's proof writing style more in line with that of handwritten proofs, and to prevent students from being bogged down by having to show 'obvious' statements, Waterproof's tactics try to prove the user's claims automatically, like in the proof steps

```
It holds that (...).
We conclude that (...).
It suffices to show that (...).
```

Explicit justifications can still be provided by referring to specific lemmas, for example by writing

```
By lemma_1 it holds that (...).
```

Using an automation system brings multiple advantages. First, users are able to write proofs without them having to know all the details of the mathematical library. They do not need to know the specific names for basic properties like $0 < 1$. We also found that with automation it became much easier to use a forwards-reasoning style, which better matches the flow of regular math proofs than the backwards-reasoning style encouraged by most proof assistants. Finally, the automation system allows Waterproof's proof step formulations to de-emphasize the use of labels. In the default COQ language, every statement is labelled, because these labels are needed to construct the explicit justifications required by the proof assistant.

In the creation of mathematical libraries and exercises for their courses, teachers can tune the automation system such that it is able to show those (and only those) statements which they consider to be 'obvious'. The lemmas and solvers that the system is allowed to use in its proof search are managed using COQ's so-called *hint databases*, which can be customized by teachers themselves. For the Analysis 1 course, for example, we added custom tactics for verifying computations involving absolute values.

Some statements are actively shielded from the full power of the automation system. Only a selection of the available hint databases is used to try to prove such statements automatically. We chose to shield statements starting with a logical operator, e.g. $\forall x : \mathbb{R}, x^2 \geq 0$, since this forces students to actively engage with such operators for non-trivial statements.

- **Mandatory signposting.** Handwritten proofs contain many statements that indicate what needs to be shown or what case is being considered. Proofs written in proof assistants often neglect such *signposting* altogether, because the current proof state and goal are constantly displayed in the automated bookkeeping overview. Proofs can be signposted using comments, but these are optional. In contrast, Waterproof's proof language at certain points forces users to signpost their proofs, thus we communicate to students that we expect them to use these statements in their handwritten proofs as well.

  One example is the mandatory inclusion of a statement indicating what case is being considered after the case distinction $(\varepsilon < 2) \vee (\varepsilon \geq 2)$ in Figures 2 and 3. In Waterproof's custom proof language, the `Case (ε < 2).` statement *has* to be included before further progress can be made. The corresponding sentence in the default COQ proof, `(* Case ε < 2. *)`, is merely a comment. Use of the `Case (...).` statement in Waterproof is enforced by the `Either (...) or (...).`-tactic which performs the case distinction. It changes the first subgoal, which is visible to the user via the automated bookkeeping overview, to the text

```
Add the following line to the proof:
  Case (ε < 2).
```

After the correct `Case (...).` -statement is included, the text is changed back to show the original goal.

Besides forcing the users to indicate what case is being considered after a case distinction, Waterproof's proof language enforces users to indicate the base case and the induction step after the announcement of an induction argument, and to indicate what goal is being show after the proof of a $\wedge$-statement is split into separate subproofs.

Users can also insert optional signposting by using the statement

```
We need to show that (...).
```

whenever they feel that this helps to keep the proof readable. This sentence checks whether the goal specified by the user actually corresponds to what needs to be shown, thus acting as sanity-check.

- **More elaborate error messages.** Waterproof's custom tactics provide elaborate feedback to help students fix issues themselves. The errors thrown by Coq's default tactics can sometimes be difficult to parse. For example, if the goal is to show that $\exists x : \mathbb{R}, x \geq 0$, and the default Coq tactic `intro x.` is mistakenly used, a cryptic error message is returned:

  ```
  No product even after head-reduction.
  ```

  In comparison, the equivalent Waterproof tactic `Take x : ℝ.` returns the error message

  ```
  'Take ...' can only be used to prove a 'for all'-statement (∀)
  or to construct a map (→).
  ```

  Note that Waterproof's proof steps formulations also allow for more mistakes to be made since they require more data to be specified by the user. For example, if the user starts a $\forall x : \mathbb{R}, \ldots$ proof with `Take x : ℕ.`, they are presented with the error

  ```
  Expected a variable of type ℝ instead of ℕ.
  ```

  Such a mistake would not have been possible to make with Coq's default '`intro`'-tactic.

- **Conventional mathematical notation.** Waterproof's custom proof language provides basic notations for common mathematical objects like sets ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, etc.), logical operators ($\forall$, $\exists$, etc.), and intervals. For function application, the mathematical convention is used instead of the functional programming-notation used by default Coq, e.g.

  `f(1+1,2)` vs. `f (1+1) 2`

  We also provide an alternative way to denote subset membership than the one commonly used in default Coq (and other proof assistants based on type theory), namely

  `x : [0,1)` vs. `[0,1) x`

  Both expressions are definitionally equal to the proposition $0 \leq x < 1$. We wanted the :-symbol to just read to students as a replacement for $\in$-symbol, without having to explain the logical differences. Since some may object to the abuse of the type-theoretical :-symbol to denote subset membership, this notation has been made optional.

The custom language also provides a framework for expanding definitions whose notation consists of multiple words, like ‘`1 is the supremum of [0,1)`’. To expand this definition in default Coq, the user has to know that this term is represented internally as ‘`is_sup [0,1) 1`’, since it is the term ‘`is_sup`’ that needs to be expanded. With Waterproof’s custom proof language it is possible to write

```
Expand the definition of supremum in (1 is the supremum of [0,1)).
```

which returns the expanded definition. To allow users to unfold notation in this way does require some extra lines of code to be added to mathematical libraries in places where notations are defined, but this is within reach of teachers who are developing their own libraries.

- **Chains of (in)equalities.** Chaining simple (in)equalities to prove more complicated ones is key ingredient of mathematical proofs, especially in analysis. Some proof assistants support this style of reasoning, like Lean’s calculational proofs, but Coq, by default, does not. Waterproof’s custom proof language does provide the ability to use such chains, like in the sentence

```
We conclude that (& 0 < 4 - 1 < 4 - ε/2 = a).
```

- **Suggestion for next proof step.** In a first attempt to provide more than just reactive feedback, the custom proof language is equipped with a `Help.` -tactic that gives basic suggestions for what the next step in a proof should be. The suggestion is based on the current shape of the goal. For example, if the goal is to show that $\forall x : \mathbb{R}, x^2 \geq 0$, the suggestion will be to introduce the variable x using the tactic `Take x : ℝ.` .

3.2 Implementation

This section highlights some of the implementation details of Waterproof’s custom proof language, parts of the exposition can be quite technical. The custom language is mostly implemented using standard Coq scripts, the Ltac2 tactic language is used to specify the behavior of the custom tactics. The automation system is partially written in OCaml, as this allowed for better access to Coq’s internal structure like how the hint databases are called by the default ‘`auto`’-tactic.

Our coq-waterproof library provides many examples of custom Ltac2 tactics that are non-standard in the sense that they are not-necessarily aimed at manipulating the proof term. For example, large parts of the implementation of the `Take ...` -tactic discussed and shown below do not alter the proof state but instead serve to give detailed error messages. We hope that our library can provide inspiration to people learning the Ltac2 tactic language.

3.2.1 Example of a tactic implementation and the numerous amount of checks on user input

The `Take ...` -tactic is one of Waterproof’s natural language tactics implemented using Ltac2, it implementation is shown in Listing 1. The variables are introduced by calling the default ‘`intro`’-tactic (line 7), but before doing so, a numerous amount of checks are performed in order to provide detailed error messages. Thus, the implementation takes up an entire page, instead of a few lines.

The notation for the tactic is defined at the bottom (line 46). It allows multiple variables from different types to be introduced in a single step, e.g. by writing

```
Take n : ℕ and x, y : ℝ and z : ℂ.
```

Per variable and per type, it is checked whether the goal (still) requires a variable of this type to be introduced (lines 2–6, and lines 15–16 together with 22–23). For each type, it is also checked whether the type that needs to be introduced is not a proposition (lines 18–19 and 34–37). A type is considered to be a proposition if it belongs to the sort '`Prop`'. In general, mathematical libraries use '`Prop`' to encode propositions, but such a neat separation is not always guaranteed: the Homotopy Type Theory library [2], for example, places propositions in the same universe as one-element sets. Note that if the first check (lines 34–37) fails, a different error message is returned than if the repeated check (lines 18–19) fails.

3.2.2 Automation system written in OCAML

The implicit automation system uses a custom version of COQ's '`auto`'-tactic that tries to include a user-specified lemma in its proof finding algorithm. Like the original '`auto`'-tactic, the custom version is written in OCAML. The condition that the user-specified lemma has to be used, allows Waterproof to reject statements like

```
By IVT it holds that (1 + 1 = 2).
```

which claims that $1+1=2$ holds because of the intermediate value theorem (IVT). The custom '`auto`'-tactic will try to apply the '`IVT`'-lemma at every branch of the search tree, and fails if no successful path is found that involves the lemma.

Using OCAML scripts, we are also able better control what hint databases are used by the automation system. In the default COQ language, the databases that are used are specified in the call to the '`auto`'-tactic directly, but for the Waterproof language to mimic the style of handwritten proofs, this is not possible. Our solution was to create a custom database management system. Based on the context, three different collections of hint databases can be used by the automation system. The 'weak' collection, for example, is used for proving shielded statements. We include custom vernacular commands that allow teachers to customize which hint databases are included in these collections when creating their exercises.

The OCAML scripts for the custom '`auto`'-tactic and the hint database management system were written by Balthazar Patiachvili during his 2023 internship, the technical details can be found in his M1 internship report [18].

3.2.3 Enforcing the use of signposting by wrapping the proof goal

The `Either (...) or (...).`-tactic forces users to indicate what case is being considered by secretly wrapping the proof goal in a type family that only the correct `Case (...).`-tactic can undo. The wrapped type is given a print-only notation that informs the user which tactic to use to unwrap the goal. This technique is used by all tactics that enforce the use of signposting. Listing 2 shows the definition of the 'case'-wrapper, its print-only definition, and how it is used by (subroutines of) the `Either (...) or (...).` and `Case (...).`-tactics to wrap and unwrap the goal. To make sure that the other tactics in Waterproof's custom proof language do not alter wrapped goals, they first call a '`panic_if_goal_wrapped`' function, which throws an error if the goal is wrapped. The `Take ...`-tactic can be seen to call this function in line 47 of Listing 1.

3.2.4 Flexible chains of (in)equalities using typeclasses

Waterproof's custom notation for chains of (in)equalities makes extensive use of typeclasses. Chains consists of a base-component, like '`x < y`', to which multiple link-components, like '$\leq$ `z`' or '`= w`'

```
Local Ltac2 intro_ident (id : ident) (type : constr) :=
  lazy_match! goal with
    | [ |- forall _ : ?u, _] =>
      let ct := get_coerced_type type in
      (* Check whether we need a variable of type [type], including coercions. *)
      match check_constr_equal u ct with
        | true  => intro $id
        | false => throw (too_many_of_type_message type)
      end
    | [ |- _] => throw (too_many_of_type_message type)
  end.

Local Ltac2 intro_per_type (pair : ident list * constr) :=
  let (ids, type) := pair in
  lazy_match! goal with
    | [ |- forall _ : ?u, _] =>
      (* Check whether [u] is not a proposition. *)
      let sort_u := get_value_of_hyp u in
      match check_constr_equal sort_u constr:(Prop) with
        | false =>
          (* Check whether we need variables of type [type], including coercions. *)
          let ct := get_coerced_type type in
          match check_constr_equal u ct with
            | true  => List.iter (fun id => intro_ident id type) ids
            | false => throw (expected_of_type_instead_of_message u type)
          end
        | true  => throw (of_string "Tried to introduce too many variables.")
      end
    | [ |- _ ] => throw (of_string "Tried to introduce too many variables.")
  end.

Local Ltac2 take (x : (ident list * constr) list) :=
  lazy_match! goal with
    | [ |- forall _ : ?u, _] =>
      (* Check whether [u] is not a proposition. *)
      let sort_u := get_value_of_hyp u in
      match check_constr_equal sort_u constr:(Prop) with
        | false => List.iter intro_per_type x
        | true  => throw (of_string
        "`Take ...` cannot be used to prove an implication. Use `Assume that ...` instead.")
      end
    | [ |- _ ] => throw (of_string
    "`Take ...` can only be used to prove a `for all`-statement or to construct a map.")
  end.

Ltac2 Notation "Take" vars(list1(seq(list1(ident, ","), ":", constr), "and")) :=
  panic_if_goal_wrapped ();
  take vars.
```

Listing 1: Implementation of the `Take ...`-tactic used for introducing variables.

```
1  Module Case.
2    Private Inductive Wrapper (A G : Type) : Type :=
3      | wrap : G -> Wrapper A G.
4    Definition unwrap (A G : Type) : Wrapper A G -> G :=
5      fun x => match x with wrap _ _ y => y end.
6  End Case.
7
8  Notation "'Add' 'the' 'following' 'line' 'to' 'the' 'proof:' 'Case' ( A )." :=
9    (Case.Wrapper A _) (at level 99, only printing,
10     format "'[ ' Add  the  following  line  to  the  proof: ']' '//'   Case  ( A ).").
```

```
11 Ltac2 either_or_prop (t1:constr) (t2:constr) :=
12   let h_id := Fresh.in_goal @_temp in
13   let attempt () := assert ($t1 \/ $t2) as $h_id by (* call to automation system *) in
14   match Control.case attempt with
15     | Val _ =>
16       let h_val := Control.hyp h_id in
17       destruct $h_val;
18       Control.focus 1 1 (fun () => apply (Case.unwrap $t1));
19       Control.focus 2 2 (fun () => apply (Case.unwrap $t2))
20     | Err exn => throw (of_string
21       "Could not find a proof that the first or the second statement holds.")
22   end.
```

```
23 Ltac2 case (t:constr) :=
24   lazy_match! goal with
25     | [|- Case.Wrapper ?v _] =>
26       match check_constr_equal v t with
27         | true => apply (Case.wrap $v)
28         | false => throw (of_string "Wrong case specified.")
29       end
30     | [|- _] => throw (of_string "No need to specify case.")
31   end.
```

Listing 2: Implementation of the 'case'-wrapper that forces users to state what case is being considered. Top: definition of the wrapper and its print-only notation. Middle: subroutine used by the `Either (...) or (...).`-tactic to wrap the goal by applying '`Case.unwrap`' in lines 18 and 19. Bottom: subroutine called by the `Case (...).`-tactic to unwrap the goal by applying '`Case.wrap`' in line 27.

can be added. The ordering-symbols in these components are purely formal, they are given a context specific interpretation at a later stage. The '`&`'-symbol at the start of an (in)equality chain is shorthand notation for a complicated expression that specifies how base- and link-components are combined using special '`chain_link`'-operators. These components, however, cannot be combined freely: the variables need to be of the same type, and the final combination is not allowed to contain both '<'- and '>'-symbols. This is where typeclasses come in: the '`chain_link`'-operators are typeclass properties which are only defined for certain combinations of chains and link-components. Typeclass resolution will fail for non-valid combinations. The '`&`'-symbol notation also adds a function called '`total_statement`', which turns a chain, with purely formal ordering symbols, into a large conjunction of individual, semantically meaningful (in)equalities. This function is again a typeclass property, such that the interpretation it provides to the formal ordering symbols can easily be adjusted to new contexts by the creators of mathematical libraries.

To function properly, the implementation of (in)equality chains requires some coupling with the rest of the proof language. In the `We conclude that (...).` -tactic, a piece of code is included to check whether the global statement of a chain, for example $0 < a$ for the chain $0 < 4 - 1 < 4 - \varepsilon/2 = a$, actually matches the current proof goal. Secondly, the automation system contains code that first splits chains into their individual components before proving them, so that, if the automation system fails, the user knows exactly which (in)equality is to blame.

4 Custom Editor

The Waterproof editor can be used for Coq documents in general, but it was also designed for use in education specifically. It uses mixed documents that combine formal Coq proofs with formatted text, including LaTeX-expressions. The editor offers general quality-of-life improvements, like an autocomplete function for mathematical symbols and common proof steps. For use of documents as exercise sheets, student input can be limited to designated areas. The amount of information shown in the automated bookkeeping panel is by default limited to only showing the proof goal. By removing this scaffolding, we expect that it is easier to transfer proof skills from Waterproof to pen-and-paper proofs (issue I).

4.1 Features

The main features of the custom editor are listed below.

- **Mixed documents.** The editor was built to support documents that combine verified Coq proofs with easily readable, formatted text. The styling of the text segments is specified using Markdown, which besides basic text formatting also supports bullet points, URLs, images, and animations. LaTeX-expressions are supported as well, both as inline and displayed equations.
- **Designated input areas.** In order to use Waterproof documents as *exercise sheets*, documents where the proofs have been left empty to be completed by students, it is important that students' input is limited to designated *input areas*. Otherwise, students might accidentally change or even delete part of the exercises. An example of such an input areas is shown in Figure 1, it is demarcated by the green bar to its left. The green color indicates that the proof provided there is correct; if the proof were to contain an error, the bar would be red.

  Editing the document outside of designated input areas can be toggled with a setting called 'Teacher Mode', it is disabled by default. In principle, students could enable Teacher Mode in the settings menu, but they do not know about its existence, nor would it give them any benefit in our course setup. Instead of allowing Teacher Mode to be set by all users, in a future version we would like to give control over this setting to teachers only.

For the creation of exercise sheets by teachers, it is convenient to first create a master file that tests the feasibility of exercises. This file can be explicitly saved as an exercise sheet, which removes all the content from input areas.

- **Hidden segments.** Parts of a document can be marked as *hidden segments*, their contents can be revealed and re-hidden by clicking on them. Originally this feature was meant for including optional hints for exercises, but it turned out to also be useful for hiding the preambles that import libraries at the start of a file (for an example, see Figure 1). If these preambles are not hidden, they distract and confuse students.
- **Limited automated bookkeeping.** Proof assistants alleviate the mental workload for mathematicians by keeping track of variables, assumptions, intermediate assertions, and proof goals in an automated bookkeeping overview. Students, however, might learn to become dependent on these features [12]. By default, the Waterproof editor only shows the proof goals, the remaining information can be found in a separate panel which needs to be opened on purpose. Whereas with the default COQ language, the automated bookkeeping panel may contain crucial information, Waterproof's custom language itself requires all the information needed to complete a proof.
- **Quick and easy input methods for mathematical symbols and proof steps.** The Waterproof editor provides an autocompletion functionality for mathematical symbols and proof steps from the custom proof language. To type the ∞-Unicode symbol, just start typing the string `\infty` and select the correct suggestion from the autocomplete menu. Proof steps are even easier thanks to the use of *code snippets*. Code snippets are templates with gaps for context-specific information, the `tab`-key can be used to move directly between these holes. Autocompletion for code snippets works the same way as for normal strings.

  Symbols and proof steps can also be inserted by selecting them from one of the additional side panels provided by the editor, see Figure 1. Although using the autocomplete function is faster, beginning users will not be familiar with the available proof steps and the (LaTeX) encodings used for the mathematical symbols. These panels provide an overview of which symbols and proof steps are available, they also include brief explanations and examples of how the proof steps are used.
- **Suggestion for next proof step.** Right-clicking in the proof makes a small 'Help' button appear, clicking this button calls the corresponding-tactic from Waterproof's custom proof language (Section 3.1, final bullet point), which provides the user with a suggestion what proof step to perform next.
- **Separate panel for expanding definitions.** The editor provides a way to expand definitions that is more similar to how this is done with pen-and-paper. In most proof assistants, expanding a definition is treated akin to other proof steps: a command in the proof text instructs the program to expand the definition and the result is shown in the information panel tracking the current proof state. When writing a proof by hand, looking up definitions in e.g. the lecture notes, is a separate activity that is spatially removed from the place where the proof is being written. The Waterproof editor provides a dedicated side-panel (see Figure 1), away from the main proof writing window, where users can instruct the proof assistant to expand definitions and inspect the results.

- **Modern, friendly design.** The Waterproof editor adheres to VS CODE's minimalist UI design and is compatible with its different color themes. Although VS CODE is often used for coding, thanks to the use of mixed documents the Waterproof editor does not resemble a coding environment.
- **Compatibility with the COQ ecosystem.** The Waterproof editor uses COQ's new `.mv`-files which support the mixing of formatted text and formal proofs. The editor does not yet support the older, more common `.v`-files.
- **Optional use of the custom proof language.** Although by default the editor is configured to use Waterproof's custom proof language, this can be changed in the settings menu. Doing so changes the autocomplete functionality and the proof step-overview panel to show a selection of COQ's default formulation of proof steps. Without the plugin implementing the custom proof language, the editor is no longer able to provide suggestions for the next proof step, but the separate panel for expanding definitions still works.

4.2 Implementation

The Waterproof editor uses COQ's new `.mv`-files as underlying documents. These files natively combine formatted text and formal COQ proofs. The formatting of the text is specified using the MARKDOWN language[8], HTML-like markers are added to the text parts to indicate the input areas and hidden segments. For example, an input area is marked by `<input-area> ... <\input-area>`. Editing and rendering of the text and code parts respectively is done using the *What You See Is What You Get (WYSIWYG)*-style editors PROSEMIRROR[9] and CODEMIRROR[10]. The PROSEMIRROR-MATH package[11] is used to render LaTeX-expressions.

The editor uses COQ LSP[12], developed by Emilio Jesús Gallego Arias, to communicate with a COQ instance running in the background. COQ LSP implements an extended version of the standard Language Server Protocol (LSP), the extension for example also specifies how the information in the automated bookkeeping panel is communicated between the editor and language server. COQ LSP succeeds COQ SERAPI [8], a COQ-specific communication protocol, which was used by previous versions of the Waterproof editor. Among other things, COQ LSP supports the new `.mv`-files and allows for continuous checking of documents. Before, the program had to be instructed manually to verify proof steps, which had to be explained to students. COQ LSP also provides custom ways to deal with errors. For example, incomplete proofs are automatically interpreted as being 'admitted', meaning that users can continue working on other proofs without having to explicitly note that they (temporarily) gave up on a preceding proof.

The editor is implemented as a custom VS CODE extension since this makes it easier for our small development team to maintain. Previous versions of the editor were stand-alone programs, which meant that we had to write and maintain code for all kinds of basic functionalities, like menu bars. Such features are now inherited from the VS CODE editor, and we can also reuse the standard VS CODE language client for communication with COQ LSP. Delegating the maintenance of these code bases allows us to focus on features specific to Waterproof.

---

[8] Developed by John Gruber (2004). Available at https://daringfireball.net/projects/markdown.

[9] Developed by Marijn Haverbeke. Available at https://prosemirror.net.

[10] Developed by Marijn Haverbeke. Available at https://codemirror.net.

[11] Developed by Benjamin Bray. Available at https://benrbray.com/prosemirror-math.

[12] Developed by Emilio Jesús Gallego Arias (2023). Available at https://github.com/ejgallego/coq-lsp.

We tried to keep the design of the editor as modular as possible to facilitate reuse of its code base in different experiments with editors for proof assistants. Similarly, we have also tried to reduce the coupling between the Waterproof editor and the custom proof language, but some dependency remains: the configuration of the autocompletion and the overview side-panel have to be manually adjusted to match the formulations in Waterproof's proof language. The feature that suggest the next proof step also relies on the custom proof language for its implementation.

5 Use in Education

For the last four years at Eindhoven University of Technology (TU/e), Waterproof has been used to supplement teaching the Analysis 1 course. The goal is to teach students how to rigorously prove theorems from calculus. This includes teaching them about mathematical concepts like metric spaces, sequences, series, convergence, limits, and continuity, but also teaching students how to write proper mathematical proofs. Students have encountered proofs and logic in an earlier course, but their proof writing skills are still in development. This section outlines how Waterproof is used in Analysis 1, how students and teachers experience working with the software, and a brief observation of the effects on students' handwritten proofs.

5.1 Course Details

Analysis 1 is a mandatory course for all first-year mathematics undergraduate students at the TU/e; each year approximately 175 students register for the course. Most are first-year students, some are taking the course for a second time. Almost all students are mathematics majors, some do a combined program with either physics or computer science. The course lasts for eight weeks (excluding the exam weeks) and has a study load of 5 ETCS. It consists of biweekly lectures and instruction classes, an exam, a midterm, and weekly homework exercises. The homework exercises are handed-in in groups of four and make up 10% of the final grade.

5.2 Waterproof in Analysis 1

Waterproof is used as an alternative for the regular homework exercises. A selection of exercises have been made available as Waterproof documents where the proofs are left incomplete. Instead of handing in handwritten solutions, students can choose to hand in completed versions of these documents.

Use of Waterproof is not mandatory, and only some of the instructors are able to offer support for Waterproof. Last year (2022-2023), three out of the six tutors were able to help students this way. Not all instructors were intimately familiar with the software: they had only used Waterproof to solve the exercises themselves, but this did allow them to answer many of the students' questions. In case these instructors could not figure out how to solve an issue, they could defer to some who had more experience with Waterproof. At the start of the course, students are informed which instruction classes will support the use of Waterproof, and that they should register for these classes if they wish to use the program; in the other instruction classes, students are expected to hand in their homework the regular way, i.e. handwritten or typed up in LaTeX.

No additional time is reserved to teach students how to use Waterproof: they are provided with a tutorial file (which takes a couple of hours to complete) that teaches them how to use Waterproof's custom proof language, and videos explaining how to use the editor. They can also ask questions during their scheduled instruction classes.

Homework solutions handed in as Waterproof files are graded automatically using MOMOTOR, an automated processing tool developed by the TU/e, that is integrated into the CANVAS Learning Management System (LMS). Grading is done on a binary scale: if a proof is deemed correct by an instance of the proof assistant running on the MOMOTOR server, full points are awarded; zero points are awarded if the proof contains a mistake or is incomplete. This strict grading scheme matches the feedback provided by the program to students when working on a proof; with human graders there is more room for nuance. To make sure that a student did not alter the exercise descriptions in their homework file, the student solutions are extracted and ran inside a fresh copy of the original Waterproof document.

5.3 Evaluation

We have evaluated Waterproof based on small student surveys, one-on-one conversations with students, and conversations with the tutors who were able to help students with Waterproof exercises. Note that this is an evaluation of a previous version of Waterproof which was used in last year (2022-2023)'s Analysis 1 course. We expect that the updated version of the software presented in this article will have improved the user experience.

5.3.1 Student Experience

Student opinions on Waterproof vary: some prefer to do their homework by hand, some are set on completing all exercises in Waterproof; both groups seem to be evenly represented. Both stronger and weaker students use the software. The retention rate is high: at the start of last year's course (2022-2023), 25 student groups (≈ 100 students) handed in their homework using Waterproof; 19 groups (≈ 76 students) continued using Waterproof for the final homework exercise. These approximate student numbers should be taken with a grain of salt, we did not track the number of individual students that used Waterproof. In some groups, a single group member works on all the Waterproof exercises, becoming this group's 'Waterproof expert'; in other groups, all four students work on the Waterproof exercises together.

Students are sometimes frustrated with Waterproof, relating to aspects of the program that still need to be improved. Syntax errors, for example, provide too little information on how to fix them: a single, hard-to-notice typo can cause an entire proof step to be rejected. Sometimes also correct proof steps are rejected by Waterproof: some statements are too difficult for the automated proving system to verify, whereas they are 'basic' for both students and instructors. Even in these cases, some students are determined to finish the exercise using Waterproof, even if it means rewriting their proof multiple times in different ways until it is accepted.

Students tend to stop using Waterproof when they feel they have little to gain from using the software, meaning that they think their handwritten solutions will get them enough points that working out their proofs in full using Waterproof is not worth the extra time and effort. Considering the high retention rate, this implies most students still find benefit to using Waterproof even at the end of the course.

5.3.2 Teacher Experience

The instructors who used Waterproof were impressed by how easy it was to use without prior experience with proof assistants. Nevertheless, they ran into the same issues as students, namely that some obvious statements would not be approved by the program and that some typos were hard to fix due to incomplete error messages.

Teachers also noticed that a majority of the questions they received during instruction classes were related to Waterproof instead of the exercises which were not converted into Waterproof documents. Students confirmed that they preferred to work on the Waterproof exercises during the instructions because this allowed them to ask questions directly when they encountered errors.

5.3.3 Observations regarding Waterproof's Effect on Handwritten Proofs

We observed that students started using Waterproof's specific formulations of proof steps in their handwritten homework as well as on the exam. The explicit use of phrases like *"We need to show that ..."*, *"By ... it holds that ..."*, or *"We claim that ..."* clearly communicated students' intentions and made it easier to grade their proofs, even if they turned out to be incorrect. Variables and hypotheses were properly introduced before being used, and quantifiers were not kept around unnecessarily.

6 Discussion

Preliminary observations suggest that using Waterproof helps to improve the quality of students' handwritten proofs, but this claim needs to be backed up by a proper study. Although students were separated into instruction classes that offered Waterproof support and ones that did not, we did not take the preparatory steps, e.g. getting students' consent, needed for a publishable evaluation. These steps have been taken for this year (2023-2024)'s Analysis 1 course; we will study student performance based on their grades and the structure of their proofs.

Although Waterproof's custom proof language can be used across mathematical disciplines, creating exercises for different courses remains a time-consuming process for teachers and/or developers. Part of the problem is finding a suitable library wherein all the required mathematics have been formalized. Often these libraries encode definitions and theorems in different ways than the precise formulations used in a course, so a bridge needs to be built to connect the two. Students will get confused if definitions in Waterproof exercises differ from those used in the regular exercises. Besides finding/building the right mathematics library, using Waterproof's custom proof language comes with an additional requirement: having to tune its automated proof finding system. The right lemmas have to be provided to make sure that statements which are 'obvious' to students and instructors are not rejected by the program.

The formulation of the proof steps in Waterproof's custom proof language are quite strict. Even small variations, like being able to write `Assume (...).` instead of `Assume that (...).`, have to be hard-coded. Deviations from these pre-defined formulations can result in syntax errors that are difficult to understand, this is especially a problem with typos. Currently, only English language proof step formulations are supported, other languages could be added without too much effort by developers.

Our solution for reasoning with chains of (in)equalities works very well. Still, considering how ubiquitous this reasoning technique is in mathematics, it would be nice if it were officially supported by COQ.

6.1 Future Work

We have iterated on Waterproof for the last four years and we plan to continue improving the software.

The automation system used by the custom proof language can still be improved further. Last year, some statements which students and teachers considered obvious were still rejected by the program. We have further tuned the system's hint databases to prevent such cases from happening. This year's iteration of the Analysis 1 course (2023–2024) will be another benchmark to see how successful we have been.

The feedback provided by the automation system when using the `By ...`-clause can also be improved further. Currently, we are able to check whether the justification provided by the user is indeed necessary to proof their claim for external lemmas. It is, however, also possible to insert the label of local assumptions into the `By ...`-clause, and these justifications cannot yet be checked. Moreover, if the specified lemma cannot be used, because, for example, one of its preconditions was not met, the user is only informed that the lemma failed to be used, and not about the reason why.

We also need better control over the automation system's ability to expand definitions. Currently, most of the tactics and solvers used by the automation system are able to do this automatically. As a result, some exercises can be solved almost directly by the automation system. This teaches wrong behavior to students since a proof might be accepted as complete without including some important intermediate steps. Ideally, the automatic expansion of some definitions should be shielded, such that the user has to explicitly mention the definition as part of a statement's justification, e.g. by writing

```
By definition_of_supremum it suffices to show that (...).
```

We have not yet found a satisfying way to encode subsets in proof assistants that allows them to be used like subsets in regular mathematics. Both the implementation as sigma (or record) types and as classifying predicates have their disadvantages that we do not want to bother students with: sigma types require coercions for elements from the subset to be used as elements from the underlying set, and quantification over the elements satisfying a classifying predicate has to be done via quantification over the underlying set. The mathematical libraries we considered also did not stick to one approach, they mixed them. For example, although most libraries tend towards using classifying predicates, the subset relation $\mathbb{N} \subset \mathbb{R}$ is almost always implemented using a coercion.

Currently the editor only handles `.mv` COQ files; we plan to support the older `.v`-format in the future, rendering the COQDOC comments as formatted text. We are also working to improve the modularity of the Waterproof editor, so others can reuse the parts that they like in their own projects. In the long term, we would like to have a version of Waterproof that runs completely in the browser like JSCOQ [9].

We have chosen to, by default, only have the Waterproof editor show a restricted version of the automated bookkeeping overview, but others might disagree with this choice. Although a restricted overview helps with our goal to make the process of writing a proof in Waterproof more similar to writing a proof with pen-and-paper, displaying all the available information also has its educational benefits. For example, showing all the variables, assumptions, and intermediate assertions allows students to focus on how to connect the logical arguments, or it might help students to better understand what the effects of certain proof steps are. It would be interesting to explore the impact of limiting the scaffolding provided by the bookkeeping overview on the students' ability to transition from writing proofs in a proof assistant to writing proofs with pen-and-paper.

7 Conclusion

Waterproof is an adaptation of the COQ proof assistant designed for teaching how to write mathematical proofs. Although regular proof assistants, with their direct feedback capabilities, have the potential to serve as a training areas for proof writing, they are currently lacking for use in education: there is a steep learning curve and proof writing skills obtained in these programs do not seem to transfer to handwritten proofs. We attempt to address both issues by making writing a proof in Waterproof more closely resemble writing a proof by hand.

Waterproof consists of two components, the first of which is a custom proof language that allows COQ proofs to be written in a style more similar to handwritten proofs. Proof steps take the form of sentences used in regular mathematics and an implicit automation system is used to prevent users from having to provide justifications for basic statements. At crucial steps, users are forced to signpost their proofs to keep them readable. We also add the option to write chains of (in)equalities, a feature missing in default COQ.

The second component is a custom editor, which uses mixed documents that contain both verified COQ proofs and formatted text. Some of its features are designed specifically for use in education, like the ability to limit student input to certain parts of the document, and the by-default restriction of the information shown by the automated bookkeeping overview and the scaffolding this provides. The editor also comes with some quality-of-life features, like autocompletion for proof steps and mathematical symbols.

The Waterproof software has been successfully used to supplement the teaching of mathematical proof writing in the Analysis 1 course at the Eindhoven University of Technology. Preliminary observations suggest that using Waterproof assists students in clarifying the logical structure of their proofs, including those written by hand. A future study will investigate these claims more thoroughly by systematically analyzing student grades and the quality of their proofs.

Acknowledgements. In 2019, the first version of Waterproof arose as the result of a Software Engineering Project (SEP) at Eindhoven University of Technology, which is part of the bachelor curriculum. Iterations by other SEP Teams followed in 2021 and 2023. We would like to thank the members of the SEP teams ChefCoq (2019), Waterfowl (2021), KroQED (2023) and their supervisors: Thijs Beurskens, Bas Gieling, Stijn Gunter, Menno Hofsté, Hugo Melchers, Anne Nijsten, Reinier Schmiermann, Erik Takke, David Tuin, Ruben Verhaegh, Geert van Wordragen, and their supervisor Tom Verhoeff; Adrien Castella, Adrian Cucoş, Cosmin Manea, Noah van der Meer, Lulof Pirée, Mihail Ţifrea, Tristan Trouwen, Tudor Voicu, Adrian Ș. Vrămuleț, Yuqing Zeng, and their supervisor Gerard Zwaan; Dick Arends, Franciska Asma, Casper Bloemhof, Daniel Chou Rainho, Milo Goolkate, Collin Harcarik, Banda Norimarna, Gijs Pennings, Lisa Verhoeven, Pieter Wils and their supervisor Roel Bloo. We thank Georgios Skantzaris for his feedback and help with testing when getting Waterproof ready for teaching for the first time. We would like to thank EdIn (Education Innovation) at Eindhoven University of Technology for financing the work of the students Thijs Beurskens, Sean McCarren, Jan Moraal and David Tuin on this project.

We would like to thank Balthazar Patiachvili for improving the automation system and converting part of the coq-waterproof library to an OCaml plugin during his 2023 internship. Finally, we are very grateful to Emilio Jesús Gallego Arias for his support, especially with COQ SERAPI and COQ LSP, and his quick answers to many questions.

References

[1] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk & Josef Urban (2015): *Mizar: State-of-the-art and Beyond*. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe & Volker Sorge, editors: *Intelligent Computer Mathematics*, Springer International Publishing, Cham, pp. 261–279, doi:10.1007/978-3-319-20615-8_17.

[2] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau & Bas Spitters (2017): *The HoTT Library: A Formalization of Homotopy Type Theory in Coq*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, Association for Computing Machinery, New York, NY, USA, p. 164–172, doi:10.1145/3018610.3018615. Available at `1610.04591`.

[3] Sebastian Böhne & Christoph Kreitz (2018): *Learning how to Prove: From the Coq Proof Assistant to Textbook Style*. In Pedro Quaresma & Walther Neuper, editors: Proceedings 6th International Work-

shop on *Theorem proving components for Educational software*, Gothenburg, Sweden, 6 Aug 2017, *Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 1–18, doi:10.4204/EPTCS.267.1.

[4] Merlin Carl (2020): *Number Theory and Axiomatic Geometry in the Diproche System*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: Proceedings 9th International Workshop on *Theorem Proving Components for Educational Software*, Paris, France, 29th June 2020, *Electronic Proceedings in Theoretical Computer Science* 328, Open Publishing Association, pp. 56–78, doi:10.4204/EPTCS.328.4.

[5] Merlin Carl, Hinrich Lorenzen & Michael Schmitz (2022): *Natural Language Proof Checking in Introduction to Proof Classes - First Experiences with Diproche*. In João Marcos, Walther Neuper & Pedro Quaresma, editors: Proceedings 10th International Workshop on *Theorem Proving Components for Educational Software*, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, *Electronic Proceedings in Theoretical Computer Science* 354, Open Publishing Association, pp. 59–70, doi:10.4204/EPTCS.354.5.

[6] Nathan Carter & Kenneth Monks (2016): *From Formal to Expository: Using the Proof-Checking Word Processor Lurch to Teach Proof Writing*. In Rachel Schwell, Aliza Steurer & Jennifer F. Vasques, editors: *Beyond Lecture: Resources and Pedagogical Techniques for Enhancing the Teaching of Proof-Writing Across the Curriculum*, Mathematical Association of America, Washington, DC 20090-1112, pp. 299–309. Available at `https://maa.org/sites/default/files/pdf/pubs/books/members/NTE85.pdf`.

[7] The Coq Development Team (2022): *The Coq Proof Assistant*, doi:10.5281/zenodo.1003420.

[8] Emilio Jesús Gallego Arias (2016): *SerAPI: Machine-Friendly, Data-Centric Serialization for Coq*. Technical Report, MINES ParisTech. Available at `https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408`.

[9] Emilio Jesús Gallego Arias, Benoît Pin & Pierre Jouvelot (2017): *jsCoq: Towards Hybrid Theorem Proving Interfaces*. In Serge Autexier & Pedro Quaresma, editors: *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016*, *Electronic Proceedings in Theoretical Computer Science* 239, Open Publishing Association, pp. 15–27, doi:10.4204/EPTCS.239.2.

[10] Martin Henz & Aquinas Hobor (2011): *Teaching Experience: Logic and Formal Methods with Coq*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 199–215, doi:10.1007/978-3-642-25379-9_16.

[11] Marie Kerjean, Frédéric Le Roux, Patrick Massot, Micaela Mayero, Zoé Mesnil, Simon Modeste, Julien Narboux & Pierre Rousselin (2022): *Utilisation des assistants de preuves pour l'enseignement en L1: Retours d'expériences*. *La Gazette de la Société Mathématique de France* 174, pp. 35–55. Available at `https://smf.emath.fr/publications/la-gazette-de-la-societe-mathematique-de-france-174-octobre-2022`.

[12] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne & Christoph Kreitz (2017): *Theorem Provers as a Learning Tool in Theory of Computation*. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, Association for Computing Machinery, New York, NY, USA, p. 83–92, doi:10.1145/3105726.3106184.

[13] Hendriks Maxim, Cezary Kaliszyk, Femke van Raamsdonk & Freek Wiedijk (2010): *Teaching logic using a state-of-art proof assistant*. *Acta Didactica Napocensia* 3(2), pp. 35–48. Available at `https://repository.ubn.ru.nl/handle/2066/249690`.

[14] Robert C. Moore (1994): *Making the transition to formal proof*. *Educational Studies in Mathematics* 27(3), pp. 249–266, doi:10.1007/BF01273731.

[15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn & Jakob von Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, Springer International Publishing, Cham, pp. 378–388, doi:10.1007/978-3-319-21401-6_26. Available at `https://lean-lang.org/papers/system.pdf`.

[16] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In Viktor Kuncak & Andrey Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation*, Springer

Berlin Heidelberg, Berlin, Heidelberg, pp. 24–38, doi:10.1007/978-3-642-27940-9_3. Available at `https://www21.in.tum.de/~nipkow/pubs/vmcai12.html`.

[17] Tobias Nipkow, Markus Wenzel & Lawrence C. Paulson (2002): *Isabelle/HOL*. *LNCS* 2283, Springer Berlin, Heidelberg, doi:10.1007/3-540-45949-9.

[18] Balthazar Patiachvili (2023): *Improvement of the automation in the coq-waterproof library*. Technical Report, École normale supérieure Paris-Saclay.

[19] Krzysztof Retel & Anna Zalewska (2005): *Mizar as a Tool for Teaching Mathematics*. *Mechanized Mathematics and Its Applications* 4, Special Issue on 30 Years of Mizar, pp. 35–42. Available at `http://mizar-jp.org/?page_id=21`.

[20] Benoit Rognier & Guillaume Duhamel (2016): *Présentation de la plateforme edukera*. In Julien Signoles, editor: *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Saint-Malo, France. Available at `https://hal.science/hal-01333606`.

[21] Annie Selden & John Selden (2008): *Overcoming Students' Difficulties in Learning to Understand and Construct Proofs*, p. 95–110. MAA Notes, Mathematical Association of America, doi:10.5948/UPO9780883859759.009.

[22] Athina Thoma & Paola Iannone (2021): *Learning about Proof with the Theorem Prover LEAN: the Abundant Numbers Task*. *International Journal of Research in Undergraduate Mathematics Education* 8, pp. 64–93, doi:10.1007/s40753-021-00140-1.

Interactive Formal Specification for Mathematical Problems of Engineers

Walther Neuper
Johannes Kepler University
Linz, Austria
`walther.neuper@jku.at`

The paper presents the second part of a precise description of the prototype that has been developed in the course of the *ISAC* project over the last two decades. This part describes the "specify-phase", while the first part describing the "solve-phase" is already published.

In the specify-phase a student interactively constructs a formal specification. The *ISAC* prototype implements formal specifications as established in theoretical computer science, however, the input language for the construction avoids requiring users to have knowledge of logic; this makes the system useful for various engineering faculties (and also for high school).

The paper discusses not only *ISAC*'s design of the specify-phase in detail, but also gives a brief introduction to implementation with the aim of advertising the re-use of formal frameworks (inclusive respective front-ends) with their generic tools for language definition and their rich pool of software components for formal mathematics.

1 Introduction

Since its beginnings more than two decades ago the *ISAC* project strives for a generally usable system for mathematics education, which might cover a major part of educational scenarios, posing and solving problems which can best be tackled by mathematical methods. This paper presents the second part of a concise description of the prototype that has been developed in the course of the project, the description of the so-called specify-phase, while the first part describing the solve-phase is already published [15].

The specify-phase first addresses the "what to solve", while the solve-phase addresses the "how to solve" later in the process of problem solving — an important conceptual separation in the education of engineers. In terms of computer mathematics the fist phase concerns a transition from facts in the physical world (captured by text or figures) to abstract notions of mathematics (mainly captured by formulas). And in terms of educational science and psychology this first phase concerns a transition from intuitive and associative human thinking to handling of abstract formal entities according to formal logic.

A successful transition results in a "formal specification" (to be exactly defined in §2.2 Pt.2 on p.123), so the subsequent solve-phase resides exclusively in the abstract area of computater mathematics, where Lucas-Interpretation [15] is able to automatically generate "next step guidance" [5] and reliable checks of logical correctness — this paper, however, addresses the preceding phase, more challenging in terms of computer mathematics and in terms of psychology.

The latter point will be touched on in this paper only by relating it here to recent advances in Artifical Intelligence, where ChatCPT for instance, achieved great public attention, even among computer mathematicians [3]. A software, which given a textual description of mathematical problems is able to

J. Narboux, W. Neuper and P. Quaresma (Eds.): Workshop on
Theorem proving components for Educational software 2023 (ThEdu'23)
EPTCS 400, 2024, pp. 120–138, doi:10.4204/EPTCS.400.8

© W.Neuper
This work is licensed under the
Creative Commons Attribution License.

convincingly reason about relevant facts, to react to supplementary remarks and even come up with a solution for the problem *and is able to justify the solution* — doesn't it outperform software of formal mathematics, in particular, might it outperform the *ISAC* prototype? In spite of this serious question, the present step of development in the *ISAC* project continues to build the prototype solely upon the proof assistant Isabelle [19]. Presently without systematic experiments and detailed examination, the *ISAC* project is aware, that integrations of AI technologies and formal mathematics are a vivid research topic (see, for instance, [2, 9]). And the description of our conservative approach contains hints at a "dialog guide" and a "user model", which will be necessary to adapt the power of the *ISAC* protoype (and system, sooner or later) to the needs of the user, the student – and in such future development the *ISAC* project will come back to AI.

The present description, of how the *ISAC* prototype handles the specify-phase, is partly a position paper in that it omits the justification for user requirements (already published in [16, 18, 17]) and not even evaluates them (which shall be more informative in a broader educational setting in the future). And partly the description reports technical details how design and implementation attempts to realise these requirements; a secondary objective of the description is to motivate the reuse of generic tools for language definition in proof assistants and their rich pool of software components for formal mathematics and also their front-ends — and *not* to re-invent the wheel again and again in educational math software development.

The paper has another special feature, the lack of related work — because there seems none to be to the best knowledge of the author. The ThEdu workshop series were founded to promote activities for the creation of software generally useful for mathematics education. There have been notable successes with geometry software and software to teach mechanical reasoning in academia, but software to support learning in the most general education settings, maths problem solving, still does not exist.

There is an actual reason for writing this paper now: It became apparent that the Java-based front-end of the *ISAC* prototype cannot catch up with recent development of front-ends in mathematics software: proof assistants make semantic information transparent to the user, they make formulas a door open for a great portion of mathematics knowledge just by mouse-click. So this lead to the hard decision to drop the Java-based front-end, i.e. about thirty valuable student projects [1] and to start shifting *ISAC*'s code in between Isabelle's proof machinery and Isabelle's upcoming new front-end Isabelle/VSCode. The first step in this shift is additionally motivated by a lucky technical coincidence: Isabelle's formulas (terms, in respective diction) are *linear* sequences of (very specific) characters – and so are formulas on the Braille display[2]. Thus *ISAC* engages in the "A-I-MAWEN" project aiming at an **A**ccessible [3] and **I**nclusive[4] **Ma**thematics **W**orking **En**vironment taking advantage from VSCode's accessibility features; and Isabelle enjoys collaboration on its upcoming new new front-end [11, 23].

The paper is organised as follows. After the introduction first come user requirements, as clarified in the *ISAC* project. The requirements are partitioned in general ones in §2.1 and ones addressing the specify-phase in §2.2.

After a definition of "formal specification" the design of interaction in the specify-phase is described by use of a running example given in §3.1. §3.2 describes the transition from text and figures into formal mathematics, §3.3 turns to the most essential features, freedom in input and freedom to pursuevariants

---

[1] `https://isac.miraheze.org/wiki/Credits`

[2] `https://en.wikipedia.org/wiki/Refreshable_braille_display`

[3] `https://en.wikipedia.org/wiki/Accessibility`

[4] `https://en.wikipedia.org/wiki/Inclusion_(education)`

in problem solving. §3.4 shows, what kinds of feedback can be given in the specify-phase. And §3.5 addresses problem-refinement, a particularly sketchy part in the *ISAC* prototype. The final section §3.6 concerns the transition from the specify-phase to the solve-phase.

Since the paper also wants to advertise the generic tools of proof assistants for defining new input languages, in particular the tools of Isabelle, the section on implementation is detailed as follows. §4.1 shows how elegantly *ISAC*'s input language for the specify-phase can be implemented in Isabelle/Isar, §4.2 goes into detail with handling semantics of the input. Analogously to the solve-phase also the specify-phase is organised in steps introduced in §4.3. A specific detail is refinement of equations discussed in §4.4. §4.5 addresses efficiency considerations and finally in §4.6 preconditions are described, which require evaluation by rewriting. §5 gives a brief summary and concludes the paper.

2 User Requirements

ISAC aims at an educational software and so design and development started with analysing the requirements, which a student places on the system when using it according to the introduction. The requirements capture and requirements analysis in the *ISAC* project [5] are already published in [16, 18, 17]. Here the requirements [6]. are kept as short as possible, are restricted to the specify-phase and serve to introduce the system features from the users point of view; the descriptions of respective implementation details in §3 justify the requirements more or less explicitly by backward references. However, first come some general considerations.

2.1 Foundational axioms about User Requirements

ISAC's decision to build software on top of reasoning technology – for maths education at high school and academic engineering courses, this decision is novel and opens up novel educational settings. The efficiency of these setting shall be researched thoroughly. As soon as tracing data[7] from the interaction with such tools become available, interesting discussions in educational research of mathematics education can be expected.

Until then the following requirements are considered as abstract and so far from concrete usability testing that these requirements are stated as axioms, axioms advocating software models to learn by interaction, *not* to learn by a teachers explanation primarily:

- **A complete model of mathematics** shall comprise all software tools required to solve a problem at hand, such that a student need not switch between different software applications: the problem statement, the textbook with background theory and explanation, the formulary, probably another textbook with a boilerplate for solving the problem at hand, and last not least an electronic notebook for interactive construction of a solution for the problem.
- **A transparent model of mathematics** shall be open for all kinds of inquiry of the underlying maths knowledge: What data is required to probably describe a given problem? What is required as input, what is expected as output? What are the types of the respective data types? What are the constraints (preconditions) on the input? What pattern does the problem at hand match, what are similar problems? And at the transition to the solve-phase there are further questions to be answered by the system: Which methods could be appropriate for finding a solution? Etc.

---

[5] `https://isac.miraheze.org/wiki/History`

[6] This list will go into the *ISAC* documentation and supersedes the old documentation [20]

[7] There are good experiences with logging user-interaction in *ISAC* [10]

- **An interactive model of mathematics** shall support step-wise construction of a solution to a given problem. Each step is checked for correctness and justified according to the above axiom of "transparency". So interaction follows the principle of "correct by construction".

ISAC's basic design is not embossed to mimic a human lecturer or teacher, rather the student shall learn independently from a software model, which clearly represents an essence of mathematics. In fact, if mathematics is the "science to mechanise thinking" (Bruno Buchberger [8]), then this science is most appropriate one to be modelled on computers, essentials can be best studied there and prover technologies provide the most powerful tools [14] for developing tools, which give justifications and explanations.

Experiences with *ISAC* indicate that the hint of "mechanises thinking" is helpful for students at any age, helpful to chase the omnipresent suspicion that mathematics is magic. Just watch a power user of a proof assistant, how she or he plays around with text snippets in order to find solutions (after a coarse idea of a subject matter or a proof has become clear).

2.2 User Requirements focusing Specify-Phase

Now, what follows are the requirements which serve to pass the specify-phase successfully, to come from human imagination of a problem (triggered by an informal text or a figure or whatever) to a formal representation, appropriate to be tackled within software.

1. As an educational software *ISAC* must be **intuitive** such that a newbie can easily learn to use the system by interacting with trial and error.
2. The system ensures "correctness by construction" when given a **formal specification** (definition): Two lists of terms are given, input *in* and output *out*. And given are two predicates: The precondition $Pre\ (in)$ constrains the elements of the input to reasonable values. The postcondition $Post\ (in, out)$ relates input and output (and thus characterises the kind of problem).

   This definition is according to [8] and restricts the kind of problems accepted by the system.
3. If a student gets stuck, the system can **propose a next step**, i.e. a single element of the specification, which is still missing (equivalent to Lucas-Interpretation [15]).
4. *ISAC* tries to raise students awareness of different operational power in different number ranges (e.g. in the complex numbers each polynomial has roots, in the real numbers it may have none). Thus it **restricts Isabelle's contexts to particular theories**, for instance to `Int.thy`, to `Real.thy` or to `Complex.thy`.
5. **No need for formal logic**: The formal specification of a problem must be comprehensible for students in an academic course in engineering disciplines (and also at high school) without any instruction in formal logic. So a *Specification* is given by *Given* (the input according to Pt.2), *Where* (the precondition) and *Find* (the output, i.e. the solution of the problem) and *Relate* (to be described in the next point).
6. ***Relate*** captures essential parts of the postcondition such that the description avoids logical operators (like $\exists$, $\forall$, $\wedge$, $\vee$ etc) according to requirement **UR.**5. This is a crucial design decision of *ISAC* which will be illustrated by the running example.
7. In general, mathematical problems can be solved in many ways. Thus a *Specification* must be **open for variants** which are handled within one and the same process of solving the problem. The running example (§3.3 below) illustrates three variants.

---

[8] `https://en.wikipedia.org/wiki/Bruno_Buchberger`

8. The transition from an informal problem description, given by text and by figures, to the formal specification must not be overstrained by **issues of formal notation**. The problem is well-known from educational use of algebra systems [7]. Thus *ISAC* will give the following templates for the respective types (examples are given in Fig.1 below):
    - [__=__, __=__] for lists of equalities, for example *Constants* $[a = 1, b = 2]$
    - [__, __] for lists of elements without =, for example *AdditionalValues* $[a, b, c]$
    - "__" for theories (as strings), for example *Theory_Ref* ”*Diff_App*”
    - "__/__" for string lists, for example *Problem_Ref* ”*univariate_calculus/optimisation*”
    - __ for arbitrary input (not involving the above cases), for example *Maximum* A.
9. Within the *Specification* the ***Model*** **serves two purposes**: for the *Problem_Ref* it shows the fields from *Given* to *Relate* according to Pt.5 and for *Method_Ref* it shows the guard of the program guiding the *Solution*; for details see [15]. A toggle switches between the two purposes (in the examples below indicated by $\otimes$ and $\odot$, the former for activation and the latter for idle).
10. The **specify-phase can be skipped** on predefined settings. This requirement is more important than it seems: *ISAC* may be not only be used for solving new problems, but also for exercising *known* problems, for instance differentiation. In that case *Specification* will be folded in such that the solve-phase will be started immediately by *Solution*.
11. A *Specification* **can be completed** automatically (by pushing a button or the like), in case a student wants to enter the solve-phase right now (and in case the course designer allows to do so by the predefined settings).
12. In equation solving the system can **assist in finding an appropriate type of equation**. That means, in this special case the system is capable of problem refinement (which is under ongoing research on more general problem types). The user requests this assistance by easy-going means (by pushing a button or the like).
13. There is the following **feedback on a specification's elements**, indicated at the appropriate location of the respective element on screen:
    - **Correct**: There is probably no specific indication.
    - **Syntax**: This feedback can be necessary, if the user switches the *Model* suddenly from *Problem_Ref* to *Method_Ref* or vice versa, see **UR.**9
    - **Incomplete**: Input is incomplete, in particular elements of type list.
    - **Superfluous**: An input can be syntactically correct, but have no evident relation to the problem at hand.

These user requirements evolved alongside two decades of development in the *ISAC* project, which was close to educational practice all the time due to the involved persons; and the requirements have been checked in several field tests [16, 18, 17].

The requirements raise a lot of questions with respect to detailed software design. Some of the questions will be clarified alongside presenting the running example in §3 below.

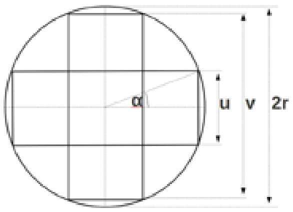
3 Design of the Specify-Phase

In the specify-phase a student interactively constructs a formal specification (definition in §2.2 Pt.2 on p.123), which then enables Lucas-Interpretation [15] to automatically generate user guidance. The concept of formal specification involves logic, which is not taught at many engineering faculties (and not at all at high school), however. Thus, according to §2.2 requirement no.2 on p.123, respective logic is hidden from the user; for explaining that approach we use a running example.

3.1 A Running Example

The running example is re-used from [11]: There it was used to illustrate what has been achieved in the old prototype and what is *planned* for developing a "mathematical working environment", whereas in this paper the example is used to explain the design and implementation of the part of the prototype supporting the specify-phase.

The efficiency of an electrical coil depends on the mass of the kernel. Given is a kernel with cross-sectional area, determined by two rectangles of the same shape as shown in the figure.
Given a radius $r = 7$, determine the length u and width v of the rectangles such that the cross-sectional area A (and thus the mass of the kernel) is a maximum.



In ancient times this kind of example belonged to a well-established class of examples at the end of the mathematics curriculum at German speaking Gymnasiums, "Extremwertaufgaben". Nowadays it can be found in text books for electrical engineering.

3.2 Formalisation of Problems and Authoring

In order to meet **UR.**3 it is necessary to prepare data for each example, where automated generation of user guidance is desired [13]. Classes of examples are collected to *Problem*s; this conforms with teaching experience: certain classes focus on certain problem patterns, which are explained and exercised by more or less different examples.

Most of the lecturers and teachers want to expose their *personal* collection of exercises to their students; so lecturers should be enabled to implement these independently in the system, without further expertise – so *authoring for lecturers* needs to be supported eventually for type `Formalise.T` below. However, the implementation of problem patterns `Model_Pattern.T` involves specific knowledge in computer mathematics – so *maths authoring* needs to be supported in addition, eventually. "Maths authors" might be lecturers of specific academic courses, book authors and others.

According to this distinction there are the following two data-types [9]:

```
01  type Formalise.T = model * refs;
```

```
01  type Model_Pattern.T =
02    (m_field *        (* field "Given", "Find", "Relate"   *)
03      (descriptor *   (* for term                          *)
04        term))        (* identifier for instantiating term *)
05    list;
06  (*does NOT contain preconditions "Where"; these have no descriptor*)
```

---

[9] We adapt ML [12] syntax for readability purposes: instead of "type T in structure Formalise" we write "Formalise.T". "(*" and "*)" enclose comments, which are outside ML syntax. The numbers on the left margin serve referencing and do not belong to the ML code.

The `type Formalise.T` consists of a (preparatory form of) model and `refs`; the latter are explained below in §3.6. `Model_Pattern.T` is designed according to **UR.**5. Each `term` is accompanied by a `descriptor`, which informs the user about what is requested from him to input. Pairs of `(descriptor * term)` are assigned to `m_fields` according to **UR.**5. The special values for the running example are as follows, where the `descriptors` are `Constants`, `Maximum`, `AdditionalValues`, `Extremum`, `SideConditions`, `FunctionVariable`, `Domain` and `ErrorBound`:

```
01  val demo_example = ("The efficiency of an electrical coil depends on the mass
                         of the kernel. Given is a kernel with cross-sectional area
                         determined by two rectangles of same shape as shown in the
                         figure. Given a radius r = 7, determine the length u and
                         width v of the rectangles such that the cross-sectional
                         area A (and thus the mass of the kernel) is a maximum.
                         + Figure",
02    ([
03    (*Problem model:*)
04      "Constants [r = (7::real)]", "Maximum A", "AdditionalValues [u, v]",
05      "Extremum (A = 2 * u * v - u \<up> 2)",
06      "SideConditions [((u::real) / 2) \<up> 2 + (v / 2) \<up> 2 = r \<up> 2]",
07      "SideConditions [(u::real) / 2 = r * sin \<alpha>, v / 2 = r * cos \<alpha>]",
08    (*MethodC model:*)
09      "FunctionVariable u", "FunctionVariable v",
10      "FunctionVariable \<alpha>",
11      "Domain {0 <..< r}",
12      "Domain {0 <..< \<pi> / 2}",
13      "ErrorBound (\<epsilon> = (0::real))"]: TermC.as_string list,
14     ("Diff_App", ["univariate_calculus", "Optimisation"],
15       ["Optimisation", "by_univariate_calculus"]): References.T)): Formalise.T
```

The text in line `01` exceeds the line limit, it has already been shown with the problem statement of the running example in §3.1. The reader may be surprised by the multiple entries for the descriptors `SideConditions`, `FunctionVariable` and `Domain`; this is due to *variants* introduced in §3.3 below — this representation of variants is considered intermediate, it was easy to implement and will be up to further refinement. The lines `14` and `15` represent the `refs` introduced above in `Formalise.T` and explained in §3.6 below. The attentive reader will also notice that the items of the example's `model` are not assigned to an `m_field` (line 02 on p.125), rather the `descriptor` is the means to relate items between different data structures, in this case between `Formalise.model` and the `problem` below defining the running example:

```
problem pbl_opti_univ : "univariate_calculus/Optimisation" =
  \<open>eval_rls\<close> (*for evaluation of 0 < r*)
  Method_Ref: "Optimisation/by_univariate_calculus"
  Given: "Constants fixes"
  Where: "0 < fixes"
  Find: "Maximum maxx" "AdditionalValues vals"
  Relate: "Extremum extr" "SideConditions sideconds"
```

The above `problem` assigns the elements of the model (in *ISAC* we call an element of a model an *item* in order not to confuse with *elements* of lists or sets) to a respective `m_field`. As already mentioned, the assignment is done via `descriptors` accompanied with place-holders (`fixes`, `maxx`, etc; see line 03 in `type Model_Pattern.T` on p.125) – these are instantiated on the fly with values of the concrete example selected by the user.

It is the task of a mathematics author to decide the structure of the model such that it covers an appropriate collection of examples, where a respective `Model_Pattern.T` of a `problem` is instantiated for several examples by values of a `Formalise.T`. For instantiation of the model for a `problem` an environment is required, for instance for the `demo_example` the environment

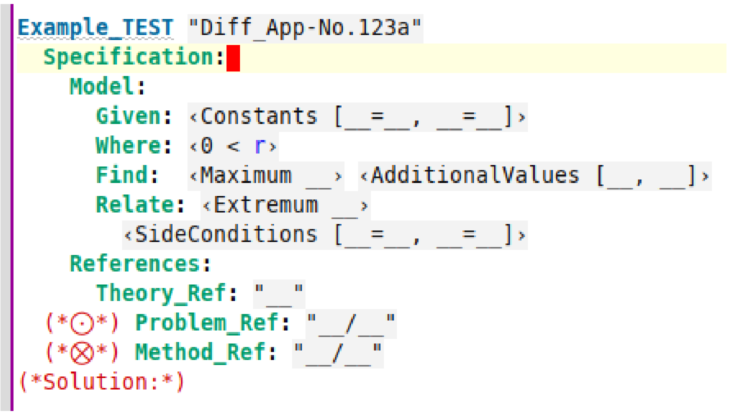
$$\begin{aligned}&[\ (\text{"}fixes\text{"},\ [r=7]),\ \ (\text{"}maxx\text{"},\ A),\ \ (\text{"}vals\text{"},\ [u,v]),\\ &\ \ (\text{"}extr\text{"},\ (A=2*u*v-u\uparrow 2)),\ \ (\text{"}sideconds\text{"},\ [(u/2)\uparrow 2+(v/2)\uparrow 2=r\uparrow 2])\\ &]\end{aligned}$$

is required.

Another reason for this late assignment is that one and the same item may belong to `m_field` `Relate` for a problem but to `Given` for a `method`, see §3.6. Finally it may be remarked that the identifier `"univariate_calculus/Optimisation"` can be found in a more convenient format already on p.126 on line 14).

3.3 Freedom of Input and Variants in Problem Solving

The model described above is presented [10] to students as shown in the screen-shot below. There the `descriptors` (`Constants`, etc) are followed by templates giving hints on the input format according to **UR.**8 in order to help students to cope with the troublesome transition to exact formal representation. The `Model` is embedded into the `Specification` which also contains `References` explained in §3.6 below. The above template is completed in Fig.2 on p.128 and the reader may refer to the `Relate` in the `Model` when reading the next paragraph.



```
Example_TEST "Diff_App-No.123a"
  Specification:
    Model:
      Given: ‹Constants [__=__, __=__]›
      Where: ‹0 < r›
      Find:  ‹Maximum __› ‹AdditionalValues [__, __]›
      Relate: ‹Extremum __›
        ‹SideConditions [__=__, __=__]›
    References:
      Theory_Ref: "__"
  (*⊙*) Problem_Ref: "__/__"
  (*⊗*) Method_Ref: "__/__"
(*Solution:*)
```

Figure 1: The template for starting a *Specification*.

Representation in field `Relate` is simplified such that a student needs not encounter formal logic according to **UR.**6. Fig.2 shows that for the `demo_example` `Extremum` and `SideConditions` actually are the essential parts of the postcondition, which characterises a `problem` by relating `Given` and `Find`:

$$\begin{aligned}&(A=2*u*v-u\uparrow 2)\ \wedge\ ((u/2)\uparrow 2+(v/2)\uparrow 2=r\uparrow 2)\ \wedge\\ \forall A'u'v'\ \ .\ \ &(A'=2*u'*v'-u'\uparrow 2)\wedge((u'/2)\uparrow 2+(v'/2)\uparrow 2=r\uparrow 2)\ \Rightarrow A'<A\end{aligned}$$

---

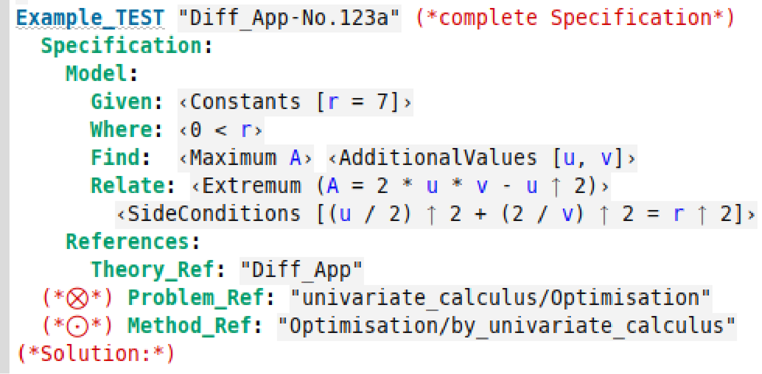
[10] §4.1 will show how easily the format presented in the screen-shot can be defined in Isabelle/Isar.

Students at engineering faculties usually are not educated in formal logic (computer science is an exception). A formula containing $\wedge$ or $\Rightarrow$ would distract them from actual problem solving. Thus *ISAC* decided to extract the essential parts of the postcondition:

$$\texttt{Extremum}\ (A = 2*u*v - u\uparrow 2),\ \texttt{SideConditions}\ (u/2)\uparrow 2 + (v/2)\uparrow 2 = r\uparrow 2$$

These parts also are sufficient to automatically generate user-guidance according to **UR.**3 (by enabling Lucas-Interpretation [13] to find a next step).

Input should not only be easygoing with respect to format and with bypassing formal logic. According to **UR.**7 the system should cope flexibly with semantics, it should also adapt to fresh ideas of a student and support various variants in solving a particular problem. Let us look at Fig.2 with the completed `Specification` and ask an important question: What could a student have in mind when encountering the problem statement on p.125? Looking at the figure on p.125, he or she might focus on the width u and the length v, and apply Pythagoras theorem $(u/2)\uparrow 2 + (v/2)\uparrow 2 = r\uparrow 2$. But students might focus as well on the angle α and formalise the `SideConditions` as $u/2 = r*\sin\alpha,\ v/2 = r*\cos\alpha$.



```
Example_TEST "Diff_App-No.123a" (*complete Specification*)
  Specification:
    Model:
      Given: ‹Constants [r = 7]›
      Where: ‹0 < r›
      Find:  ‹Maximum A› ‹AdditionalValues [u, v]›
      Relate: ‹Extremum (A = 2 * u * v - u ↑ 2)›
        ‹SideConditions [(u / 2) ↑ 2 + (2 / v) ↑ 2 = r ↑ 2]›
    References:
      Theory_Ref: "Diff_App"
  (*⊗*) Problem_Ref: "univariate_calculus/Optimisation"
  (*⊙*) Method_Ref: "Optimisation/by_univariate_calculus"
(*Solution:*)
```

Figure 2: The complete *Specification*.

And what if a student has an eye for simple solutions and takes as `SideConditions` either $u = \cos\alpha, v = \sin\alpha$ or $u\uparrow 2 + v\uparrow 2 = 4*r\uparrow 2$? The latter is semantically equivalent (equivalent in this specific example) to the variant shown in Fig.2. So, during the whole specify-phase, each formula input by a student must be checked for equivalence with the formulas prepared in the `Formalise.T`. Such checking is done via rewriting to a normal form. This might be quite tedious, for instance with equations involving fractions like in showing equivalence of the equations $u\uparrow 2 + v\uparrow 2 = 4*r\uparrow 2\ \equiv\ (\frac{u}{2})\uparrow 2 + (\frac{v}{2})\uparrow 2 = r\uparrow 2)$

An open design question, a "pattern-matching-language" ? Now we are ready to address the design decision related to the format given on p.126, which has been identified as preliminary and simplistic for the purpose of simple implementation. Much information is redundant there, apparently, and on the other hand interesting structural properties are hardly visible. For instance, the student's decision whether to take the `FunctionVariable u` or the `FunctionVariable v` for differentiation (see §3.6), this decision is fairly independent from the `problem`'s model – the `Model` just must no refer to α.

The need for a separated language for `Specification` becomes definitely clear when considering the precondition and the postcondition (the latter introduced above on p.128). The precondition must evaluate to true for a `Specification` to be valid, and the postcondition must be true for a `Solution` to be accepted by the system.

In order to evaluate precondition `Where` $0 < r$ to true, we need to create the environment $[(r,7)]$ from `Constants`. With the `demo_example` this is straight forward, but if we had `Constants [s = 1, t = 2]` we already need to employ some trickery with indexing. And what about the postcondition?

$$\begin{array}{ll} & extr\ \wedge\ sideconds\ \wedge \\ \forall\ maxx'\ \ funvar'. & extr' \wedge sideconds'\ \Rightarrow maxx' < maxx \end{array}$$

How could such a "pattern-matching" description be instantiated to the postcondition on p.128? Instantiated reliably, such that Isabelle could evaluate it to true with the values found by `Solution`? How would the variant with the α work out, which has two `SideConditions`?

Answering the open question above may go in parallel to implementation of a larger body of examples and respective `problems`. Field tests [18, 16, 17] in the *ISAC* project were based on limited content and thus did not gain much experience with a variety of problems with variants and respective flexibility of the system; one has to wait for implementation of more various examples and problems.

3.4 Interactivity and Feedback

The previous section describes how *ISAC* makes a `Specification` flexible and open for various variants in solving a particular problem. Flexibility in problem solving must be accompanied by flexible interactivity and appropriate feedback. Feedback for input to a `Model` is predefined by **UR.**13 and modelled by a respective datatype as follows (`values` are of type `term list`)

```
datatype I_Model.feedback =
    Correct of (descriptor * values)     (* wrt. syntax and semantics    *)
  | Incomplete of (descriptor * values)  (* for list types               *)
  | Superfluous of (descriptor * values) (* not found in Formalise.model *)
  | Syntax of TermC.as_string            (* in case of switching model   *)
```

The above code plus comments appear fairly self-explaining. `descriptor` has been introduced already. In case of empty input, we shall have `Incomplete (_, [])` and display templates for input-format according to **UR.**8. Syntax errors are handled very elegantly in Isabelle/PIDE (see §4.1); in case of switching from the `Model` of the `Problem_Ref` to the `Model` of the `Method_Ref` (see §3.6 below) it might be useful to keep a copy of an input with syntax errors. Items can be `Superfluous` either because they are syntactically correct but not related the `Model` under construction, or they belong to a `variant`, which has not yet been decided for. For instance, if the `Model` is as in Fig.2 on p.128 and the student adds `SideConditions` $[v = \sin\alpha]$, this item will be marked as `Superfluous` unless interactive completion to `SideConditions` $[v = \sin\alpha, v = \cos\alpha]$ and deletion of a competing item like `SideConditions` $[u \uparrow 2 + v \uparrow 2 = 4 * r \uparrow 2]$.

If a student gets stuck during input to a `Model`, then **UR.**3 allows him or her to request help. In the specify-phase such help is given best by presenting a (partial – predefined setting!) missing item. *But output of a `Model` with item does not conform to Isabelle's document model*: The content of an Isabelle theory is considered a document which has to be checked by the system — Isabelle is in principle a reactive system. There are few exceptions to the document model, for instance, when Sledgehammer [4] finds intermediate proof-steps on request and metis [11] displays the respective proof steps.

ISAC, in contrary, is designed as a tutoring system, where partners construct a `Specification` and a `Solution` in cooperation, where the user can propose a step while the system checks it and where the student can request help such that the system can propose a step as well. Since here psychology of

---

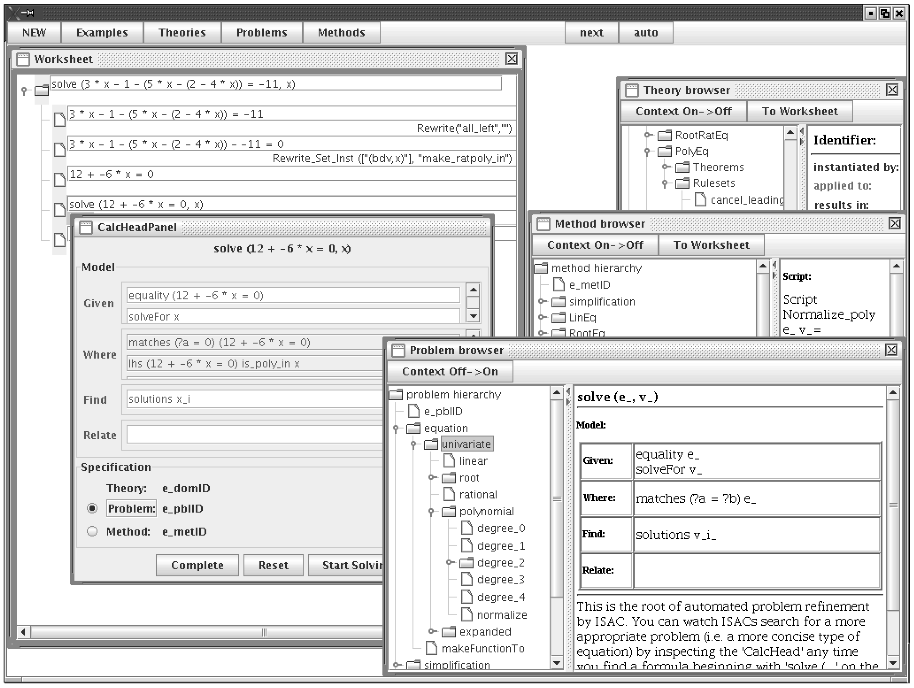
[11] https://isabelle.in.tum.de/library/HOL/HOL/Metis.html

learning is involved, a "user guide" component shall be involved in the future, and a "user model" shall provide personalisation. *ISAC*'s original architecture placed the components at the center of the system. But now *ISAC* is being embedded in Isabelle and Isabelle/PIDE is already highly elaborated — where will be the place for these components in Isabelle? This is one of the major open questions for §4.

3.5 Automated Specifications

Sometimes a user might not be interested in specification. For instance, when solving an equation, the type of the problem and the respective `Model_Pattern.T` are clear, and only the hierarchy of problem types would have to be searched for the appropriate type.

ISAC's old Java front-end had a so-called CAS-command for that situation, where one had minimal input, for instance $solve\ (12 - 6 \cdot x = 0,\ x)$. Then *ISAC* checked the most general precondition (an "=" in the input) and then started a breadth-first search in the tree of equations at the root "univariate", checking in sequence the type of the input term, `linear`, `root`, `polynomial`, rational, complex or transcendental. Fig.3 shows the currently available equations presented by the old Java fron-end. All these panels shall be redone by Isabelle/PIDE via Isabelle/VSCode.



```
solve (3 * x - 1 - (5 * x - (2 - 4 * x)) = -11, x)
3 * x - 1 - (5 * x - (2 - 4 * x)) = -11
                                         Rewrite("all_left","")
3 * x - 1 - (5 * x - (2 - 4 * x)) - -11 = 0
                  Rewrite_Set_Inst (["(bdv,x)"], "make_ratpoly_in")
12 + -6 * x = 0
solve (12 + -6 * x = 0, x)
```

Script
Normalize_poly
e_ v_=

solve (12 + -6 * x = 0, x)

Given: equality (12 + -6 * x = 0)
Given: solveFor x
Where: matches (?a = 0) (12 + -6 * x = 0)
Where: lhs (12 + -6 * x = 0) is_poly_in x
Find: solutions x_i
Relate:

Theory: e_domID
● Problem: e_pblID
○ Method: e_metID

solve (e_, v_)

Model:

Given:	equality e_ solveFor v_
Where:	matches (?a = ?b) e_
Find:	solutions v_i_
Relate:	

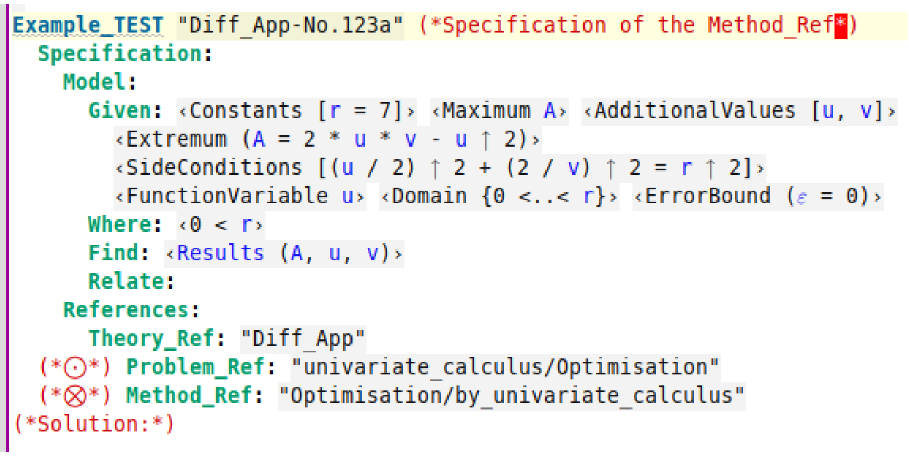
This is the root of automated problem refinement by ISAC. You can watch ISACs search for a more appropriate problem (i.e. a more concise type of equation) by inspecting the 'CalcHead' any time

Figure 3: The tree of equations (middle panel).

The more interesting question is, whether this coarse refinement via preconditions can be generalised to a kind of matching, which involves also the postcondition (see p.128); theoretical background would most likely be the refinement calculus [1].

3.6 Transition to the Solve-Phase

The solve-phase tackles the construction of a `Solution` for a given problem, where "next-step-guidance" is provided by Lucas-Interpretation [15]. For that purpose Lucas-Interpretation uses a program, and for such a program the values specified by a `Model` instantiate the formal arguments of the program to actual arguments. So items move from `Relate` to `Given` and new items need to be added to the program's guard in order to provide the program with all data required for automatically construct a `Solution`. Thus the `Model` for the guard of the `demo_example` might look as shown in Fig.4 below.



```
Example_TEST "Diff_App-No.123a" (*Specification of the Method_Ref*)
  Specification:
    Model:
      Given: ‹Constants [r = 7]› ‹Maximum A› ‹AdditionalValues [u, v]›
        ‹Extremum (A = 2 * u * v - u ↑ 2)›
        ‹SideConditions [(u / 2) ↑ 2 + (2 / v) ↑ 2 = r ↑ 2]›
        ‹FunctionVariable u› ‹Domain {0 <..< r}› ‹ErrorBound (ε = 0)›
      Where: ‹0 < r›
      Find: ‹Results (A, u, v)›
      Relate:
    References:
      Theory_Ref: "Diff_App"
  (*⊙*) Problem_Ref: "univariate_calculus/Optimisation"
  (*⊗*) Method_Ref: "Optimisation/by_univariate_calculus"
(*Solution:*)
```

Figure 4: The `Model` for the `Method_Ref`.

The reader might notice that the toggle ⊙⊗ has changed, indicating that the `Model` belongs to the `Method_Ref` (The toggle is not implemented presently and just reminded of by a comment).

So the `References` come into play, they raise lots of open design decisions: We encounter insightful *references* into huge data collections; their representation will heavily depend on specific features of VSCode (while the structure of theories, DAGs, has not even an interactive representation presently). Should they be collapsed by settings, if they would distract a certain class from problem solving? Should students be forced to input `References`, and when?

A reasonable intermediate step in development might be that a `Model` could be accepted as complete (e.g. Fig.2) by the system even with empty `References` (e.g. Fig.1) and a `Solution` could be started.

Toggling the `Model` between `Model_Ref` and `Method_Ref` (the former shown in Fig.2 on p.128, the latter in Fig.4) will be a challenge for implementation in Isabelle/VSCode.

4 Implementation in Isabelle/Isar

This paper also intends to demonstrate that it is waste of resources to start development of educational math software from scratch – nowadays users expectations on software are high such that an isolated development cannot keep pace with the still rapidly evolving state of the art. This is particularly true for front-end technology. *ISAC*'s long lasting experience with formula editors clearly demonstrates that unpleasant fact, which enforced *ISAC* to drop ten years of front-end development.

This section is going to demonstrate simplicity of implementing a specific input language as introduced in §3 above in one of the advanced proof assistants, in this case Isabelle. Isar [22] is the proof language of Isabelle and remarkably, this is defined in a generic manner such that almost arbitrary formal languages can be defined; for a particularly interesting example see [6].

4.1 Isabelle's Outer and Inner Syntax

Isabelle separates two kinds of syntax, an inner syntax for mathematical terms and an outer syntax for Isabelle/Isar's language elements. The latter is generic such that it allows for definition of various language layers. And it is a pleasure to show how easily an *ISAC* `Specification` is defined such that all inner syntax errors are indicated at the right location on screen, i.e. how easily all what is shown in the screen-shots on the previous pages are implemented such that syntax errors are shown appropriately:

```
keywords "Example" :: thy_decl
  and "Specification" "Model" "References" "Solution"
```

The above two lines prepare Isar to re-use available parsers [12]:

```
Outer_Syntax.command command_keyword\<open>Example_TEST\<close>
 "prepare ISAC problem type and register it to Knowledge Store"
 (Parse.name -- Parse.!!! (keyword\<open>Specification\<close> --keyword<:> --
   keyword<Model> -- keyword<:> |-- Problem.parse_pos_model_input --
    (keyword<References> -- keyword<:> |-- Parse.!!! References.parse_input_pos
     )) >>
  (fn (example_id, (model_input,
    ((thy_id, thy_pos), ((probl_id, probl_pos), (meth_id, meth_pos))))) =>
   Toplevel.theory (fn thy =>
     let
       val state = update_step example_id model_input
         ((thy_id, thy_pos), ((probl_id, probl_pos), (meth_id, meth_pos)))
     in set_data state thy end)));
```

With these two definitions done, one has in `update_step` only to provide appropriate calls of this function:

```
fun term_position ctxt (str, pos) =
  Syntax.read_term ctxt str
    handle ERROR msg => error (msg ^ Position.here pos)
    (*this exception is caught by PIDE to show "msg" at the proper location*)
```

The exception `ERROR` is caught by Isabelle/PIDE and, for instance, syntax errors detected by `Syntax.read_term` are displayed at the proper locations on screen together with `msg`. *That is all what Isar requires to check syntax of input as shown in the screen shots in the previous section and to indicate errors at the proper location on screen.*

The handling of *semantic* errors is accomplished with Isabelle/ML, Standard ML [12] enriched with an abundant collection of tools for formal logic as well as for connection to the front-end via Isabelle/PIDE [21]. This will be shown in the sequel.

4.2 Interaction and Feedback

Checking semantic appropriateness of user input to a Specification as described in §3 above, comprises a lot of questions: Does an item input to a `Model` belong to this particular example or is it just `Superfluous`? If there a list is to be input, are all the elements present or are some missing (items are `Incomplete`)? Is a `Model` complete with respect to a specified `Problem_Ref`? If `Theory_Ref` is input, are all the items of a `Model` still parsed correctly (or can, for instance, "$Re\,a + Im\,b$") not be parsed correctly, because the specified theory does not know the type "complex")? Etc.

---

[12] In spite of Isabelle's convenient latex extensions we still use verbatim, which displays "keyword<Specification>" as clumsily as "keyword\<open>Specification\<close>"

The central device for handling feedback to the above questions is the so-called `INTERACTION_MODEL` with the type `I_Model.T` [13] (where `variants` is a list of integers and `m_field` has been introduced by `type Model_Pattern.T` on p.125). `Position.T` establishes the reference to the location on screen:

```
type I_Model.single =
  variants *         (* pointers to variants given in Formalise.model *)
  m_field *          (* #Given | #Find | #Relate                      *)
  (feedback *        (* state of feedback for variables and values    *)
    Position.T);     (* for pushing feedback back to PIDE             *)

type I_Model.T = I_Model.single list;
```

`feedback` implements **UR.**13 and resembles the structure of `datatype feedback` in the presentation layer as shown on p.129 in §3.4. The first three characters suffice for internal naming:

```
datatype I_Model.feedback =
  Cor of (descriptor *         (* a term identifying an item          *)
     (values))                 (* of a particular example             *)
| Inc of (descriptor * values) (* incomplete lists/sets,
                                  if empty then output according to UR*)
| Sup of (descriptor * values) (* input not found Model               *)
| Syn of TermC.as_string       (* kept for P_Model.switch_pbl_met     *)
```

4.3 Step-wise Construction of Specifications

Step-wise construction is one of the axioms for *ISAC*'s design as captured on p.123. While step-wise construction appears self-evident for the solve-phase, where Lucas-Interpretation suggests or checks one input formula after the other, this appears artificial for the specify-phase: one can input any item to a `Specification` in any sequence, items of a `Model` as well as `References`. But there is also **UR.**3, which calls for the system's ability to propose a next step – and here we are:

```
datatype Tactic.input =
(* for specify-phase *)
  Add_Find of TermC.as_string                         (*add to the model*)
| Add_Given of TermC.as_string  | Add_Relation of TermC.as_string
| Model_Problem (*internal*)
| Refine_Problem of Problem.id                (*refine a Model_Pattern.T*)
| Refine_Tacitly of Problem.id (*internal*)
| Specify_Method of MethodC.id                      (*specify References*)
| Specify_Problem of Problem.id | Specify_Theory of ThyC.id
(* for solve-phase *)
| ...
```

The above `Tactics` are shown to the user, while some are used only internally; `Refine_*` will be introduced in §4.4 below. A variant stuffed with lots of data serve internal construction of a next step; the lists below shows only some examples:

```
datatype T = Add_Find' of TermC.as_string * I_Model.T
| Add_Given' of TermC.as_string * I_Model.T
| Add_Relation' of TermC.as_string * I_Model.T
| Model_Problem' of                   (*starts the specify-phase     *)
```

---

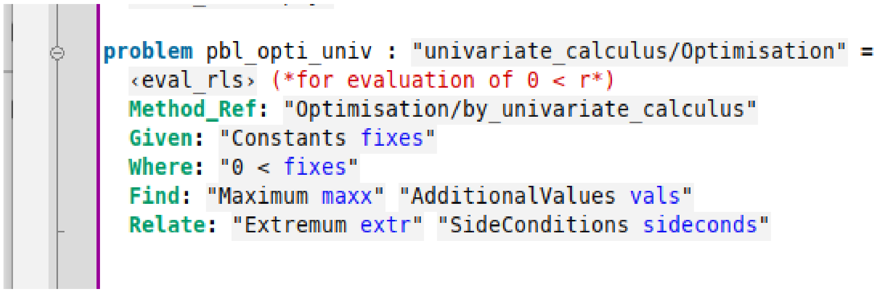
[13] In *ISAC*'s code this definition is shifted into a separate `structure Model_Def` *before* the definition of the store of a calculation, `Ctree`, which still lacks type polymorphism

```
      Problem.id *                      (*key to a Problem.T Store.node*)
      I_Model.T *                       (*model for the Problem       *)
      I_Model.T                         (*model for the MethoC        *)
  | Refine_Problem' ...
```

4.4 Refinement of Problems

UR.12 calls for refinement, a respective motivation was given in §3.5. Here implementation details are presented, because the subsequent section will address efficiency consideration raised by refinement [14].

A `Problem.T` is defined in an Isabelle theory; here the definition of the model-pattern of the running example is shown. This model-pattern covers almost all examples of a well-known text book, the items are identified by the `descriptors`; variables like `fixes` are to be instantiated from a `Formalise.model` as explained on p.127. The convenient representation in Fig.5 is stuffed with data required to solve a problem (see, for instance, the <`eval_rls`>) and expanded to the type `Problem.T`:



```
problem pbl_opti_univ : "univariate_calculus/Optimisation" =
  ‹eval_rls› (*for evaluation of 0 < r*)
  Method_Ref: "Optimisation/by_univariate_calculus"
  Given: "Constants fixes"
  Where: "0 < fixes"
  Find: "Maximum maxx" "AdditionalValues vals"
  Relate: "Extremum extr" "SideConditions sideconds"
```

Figure 5: The problem-pattern for the running example.

```
type Problem.T =
  {guh : Check_Unique.id,             (* unique within Isac_Knowledge   *)
  mathauthors : string list,          (* copyright etc                  *)
  start_refine : References_Def.id,   (* to start refinement with       *)
  thy  : theory,                      (* allows to compile model, where_*)
  cas : term option,                  (* CAS_Cmd                        *)
  solve_mets : References_Def.id list,(* methods solving the T          *)
  where_rls : Rule_Set.T,             (* for evaluation of preconditions*)
  where_ : Pre_Conds.unchecked,       (* preconditions as terms         *)
  model : Model_Pattern.T             (* "#Given", "#Find", "#Relate"   *)
}
```

By this type problems are stored in a tree borrowed from Isabelle; the key `["univariate_calculus", Optimisation""]` into this tree is created from the string `"univariate_calculus/Optimisation"` in Fig.5:

```
  type store = (T Store.node) list
```

4.5 Efficiency Considerations with Parsing

Two decades ago, at the time when *ISAC* started with development, parsing required substantial resources. Thus a specific component, `O_Model.T` was shifted in-between `Formalise.model` (p.126) and the interaction model `I_Model.T` (p.133).

---

[14] By the way, Lucas-Interpretation [15] is straightened considerably when equation solving in sub-problems automatically refines to the appropriate type of equation

```
type O_Model.single =
  variants *     (* pointers to variants given in Formalise.T *)
  m_field *      (* #Given | #Find | #Relate                  *)
  descriptor *   (* see Input_Descript.thy                    *)
  values         (* HOLlist_to_MLlist t | [t]                 *)
  );
type O_Model.T = O_Model.single list;
```

This structure is created at the beginning of the specify-phase (by `Tactic.Model_Problem`). The terms are already parsed, `descriptor` and `values` conveniently separated; also the variants are ready for convenient handling. At the occasion of parsing also an Isabelle `Proof_Context` is fed with the types of all variables in `Formalise.model`. This discharges the student of the necessity to explicitly input types.

Refinement of types of equations as introduced by **UR.**12 is particularly resources consuming: Fig.3 on p.130 shows that already the old prototype implemented quite a lot of types of equations. Refinement starts at the root `["univariate", "equation"]` of the branch, an environment with respect to the given equation has to be created from the `I_Model.T` and the current `Model_Pattern.T`, with this environment for each node in the tree the respective `"#Given"` and `"#Where"` needs to be instantiated, and last not least the latter requires evaluation by rewriting (see §4.6 below).

All that tasks act on terms, i.e. require parsing. If this is done during interactive problem solving, response time will get to high even with modern hardware. Thus not only `Problem.T` (see definition on p.126) contains terms alreay parsed, but also `Model_Pattern.T` and other structures. These structures (`Model_Pattern.T`, `Problem.T`, `MethodC.T` and `Error_Pattern.T`) hold terms, where the exact type is not known at compile time, and where the type needs to be adapted to the current `Problem.T`. Thus all these structures have a function

```
val adapt_to_type Proof.context -> T -> T
```

and these functions in turn call

```
val adapt_term_to_type: Proof.context -> term -> term
```

which is defined in structure `ParseC`.

4.6 Pre-Conditions in Problems and Methods

Pre-conditions are an essential part of a formal specification (definition in §2.2 Pt.2 on p.123). We compare the definition to *ISAC*'s representation of a `Model` (e.g. on p.127, p.128 or p.131):

formal specification	*ISAC*'s `Model`
in	`Given`
Pre (in)	`Where`
out	`Find`
Post (in, out)	`Relate`

However, `type Model_Pattern.T` (defintion on p.125) does *not* contain a precondition (*Pre* (in), `Where`) and so does the actual `Model` of a `Problem.T` (defintion on p.134; for specifics of `Relate` see p.127). The reason is that the role of items in a `Model` is to conform with a particular `Formalise.model` (made ready by parsing in `I_Model.T`) and whether it is present or not – whereas a precondition must evaluate to true in order to make a `Model` complete; this involves rewriting. The implementation details are as follows.

```
type Pre_Conds.T = (bool * term) list;
type Pre_Conds.unchecked_pos = (term * Position.T) list
type Pre_Conds.checked_pos = bool * ((bool * (term * Position.T)) list)
```

The role of preconditions is different from `Model`-items such that preconditions are stored only in a `Problem.T` (accompanied by a `Model`) and evaluated by use of environments, which are generated on the fly according to `I_Model` actually input:

```
val make_environments: Model_Pattern.T -> I_Model.T ->
  Env.T * (env_subst * env_eval)
val check_pos: Proof.context -> Rule_Set.T -> unchecked_pos ->
  Model_Pattern.T * I_Model.T -> checked_pos
```

5 Summary and Conclusions

This is the first concise description of how the *ISAC* prototype models the specify-phase. The specify-phase concerns the transition from a problem statement given in the form prose text and/or illustrations to a formal specification.

The description starts with user requirements in §2 and thus presents the perspective of students solving exercises in engineering studies (similar learning scenarios can be found in secondary schools). Here is also the definition of the notion "formal specification" as given by input and output as well as precondition and postcondition. §3 describes the design of the specify-phase with regard to the user requirements and identifies open design issues. The description of the implementation in §4 emphasises the generic tools of Isabelle/Isar and the powerful helper functions of Isabelle/ML to motivate readers to use existing proof assistants as a basis for the development of for the development of learning systems rather than starting from scratch again and again.

Another motivation for this description is to inform future collaborators of the Isac project about implementation details and its background.

Conclusions Over the decades, Isac's design of the specify-phase and the solve-phase was discussed long and wide and has proven useful in field tests; the time is ripe to move into development for widespread use. It seems helpful that the old Java-based front end has been abandoned and that the originally very ambitious specifications [20] are significantly reduced: *ISAC* will change from a multi-user system to a single-user system in line with the (current) architecture of Isabelle/PIDE; and in the next step of development the scope of application will be limited to inclusive learning situations (visually impaired students integrated) in secondary educational institutions to take advantage of the structural relationship between Isabelle's terms and the Braille display.

The sudden penetration of AI will pose the following new research tasks. Starting from a differentiation of problem solving through

1. intuitively and associatively thinking humans (who bear responsibility for their actions, etc)
2. AI with deep learning
3. formal mathematics

the interaction of (3) and (2) will be analysed. Design and implementation of a user guide and a user model (from the interaction of (3) and (1)) is already planned in the next development step.

References

[1] Ralph-Johan Back & Joakim von Wright (1998): *Refinement Calculus: A Systematic Introduction.* Springer-Verlag. Graduate Texts in Computer Science, doi:10.1007/978-1-4612-1674-2.

[2] Christoph Benzmüller & David Fuenmayor (2021): *Value-Oriented Legal Argumentation in Isabelle/HOL.* In Liron Cohen & Cezary Kaliszyk, editors: *12th International Conference on Interactive Theorem Proving (ITP 2021)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 193, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 7:1–7:20, doi:10.4230/LIPIcs.ITP.2021.7.

[3] Bruno Buchberger (2023): *Is ChatGPT Smarter Than Master's Applicants?* RISC Report Series 23-04, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria. Available at `https://www3.risc.jku.at/publications/download/risc_6684/23-04.pdf`.

[4] Jasmin C.Blanchette: *Hammering Away. A User's Guide to Sledgehammer for Isabelle/HOL.* contained in the Isabelle distribution. Available at `http://isabelle.in.tum.de/doc/sledgehammer.pdf`.

[5] Gabriella Daróczy & Walther Neuper (2013): *Error-Patterns within "Next-Step-Guidance" in TP-based Educational Systems.* In: *eJMT, the Electronic Journal of Mathematics & Technology*, 7, pp. 175–194. Available at `https://php.radford.edu/~ejmt/ContentIndex.php#v7n2`. Special Issue "TP-based Systems and Education".

[6] Adrian De Lon, Peter Koepke, Anton Lorenzen, Adrian Marti, Marcel Schütz & Makarius Wenzel (2021): *The Isabelle/Naproche natural language proof assistant (system description).* In: *28th International Conference on Automated Deduction (CADE 28)*, Lecture Notes in Computer Science, Springer-Verlag, doi:10.1007/978-3-030-79876-5_36

[7] Karl Josef Fuchs, Simon Plangg, Martin Stein & Gilbert Greefrath, editors (2020): *Computer Algebra Systeme in der Lehrer(innen)bildung (überarbeitete Neuauflage).* WTM Verlag, Münster.

[8] David Gries (1981): *The science of programming.* Texts and monographs in computer science, Springer-Verlag, doi:10.1007/978-1-4612-5983-1.

[9] Mateja Jamnik (2023): *How Can We Make Trustworthy AI?* In Marco Gaboardi & Femke van Raamsdonk, editors: *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 260, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 2:1–2:1, doi:10.4230/LIPIcs.FSCD.2023.2.

[10] Franz Kober (2012): *Logging of High-Level Steps in a Mechanized Math Assistant.* Master's thesis, IICM, Graz University of Technology. `https://static.miraheze.org/isacwiki/e/e6/Fkober_bakk.pdf`.

[11] Klaus Miesenberger, Walther Neuper, Bernhard Stöger & Makarius Wenzel (2023): *Towards an Accessible Mathematics Working Environment Based on Isabelle/VSCode.* In Pedro Quaresma, João Marcos & Walther Neuper, editors: Proceedings 11th International Workshop on *Theorem Proving Components for Educational Software*, Haifa, Israel, 11 August 2022, *Electronic Proceedings in Theoretical Computer Science* 375, Open Publishing Association, pp. 92–111, doi:10.4204/EPTCS.375.8.

[12] Robin Milner, Mads Tofte, Robert Harper & David MacQueen (1997): *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, London, doi:10.7551/mitpress/2319.001.0001.

[13] Walther Neuper (2012): *Automated Generation of User Guidance by Combining Computation and Deduction.* In Pedro Quaresma & Ralph-Johan Back, editors: *Electronic Proceedings in Theoretical Computer Science*, 79, Open Publishing Association, pp. 82–101, doi:10.4204/EPTCS.79.5.

[14] Walther Neuper (2019): *Technologies for "Complete, Transparent & Interactive Models of Math" in Education.* In Pedro Quaresma & Walther Neuper, editors: Proceedings 7th International Workshop on *Theorem proving components for Educational software*, Oxford, United Kingdom, 18 july 2018, *Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 76–95, doi:10.4204/EPTCS.290.6.

[15] Walther Neuper (2020): *Lucas-Interpretation on Isabelle's Functions.* In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software (ThEdu'20)*, Paris, France, 29th June 2020, Electronic Proceedings in Theoretical Computer Science, 328, pp. 79–95, doi:10.4204/EPTCS.328.5.

[16] Walther Neuper & Christian Dürnsteiner (2007): *Angewandte Mathematik und Fachtheorie mithilfe adaptierter Basis-Software.* Technical Report 683, IMST – Innovationen Machen Schulen Top!, University of

Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `https://www.imst.ac.at/imst-wiki/images/f/f9/683_Kurzfassung_Neuper.pdf`.

[17] Walther Neuper, Johannes Reitinger & Angelika Gründlinger (2008): *Begreifen und Mechanisieren beim Algebra Einstieg*. Technical Report 1063, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `https://www.imst.ac.at/imst-wiki/images/9/9d/1063_Langfassung_Reitinger.pdf`.

[18] Walther Neuper et al. (2006): *Angewandte Mathematik und Fachtheorie*. Technical Report 357, IMST – Innovationen Machen Schulen Top!, University of Klagenfurt, Institute of Instructional and School Development (IUS), 9010 Klagenfurt, Sterneckstrasse 15. Available at `http://imst.uni-klu.ac.at/imst-wiki/index.php/Angewandte_Mathematik_und_Fachtheorie`.

[19] Lawrence C. Paulson, Tobias Nipkow & Makarius Wenzel (2019): *From LCF to Isabelle/HOL*. *Formal Aspects of Computing* 31, pp. 675–698, Springer, London, doi:10.1007/s00165-019-00492-1.

[20] *ISAC* Team (2002): *ISAC – User Requirements Document, Software Requirements Document, Architectural Design Document, Software Design Document, Use Cases, Test Cases*. Technical Report, Institute for Softwaretechnology, University of Technology. Available at `https://static.miraheze.org/isacwiki/0/04/Isac-docu.pdf`.

[21] Makarius Wenzel & Burkhart Wolff (2011): *Isabelle/PIDE as Platform for Educational Tools*. In Pedro Quaresma & Ralph-Johan Back, editors: *Proceedings First Workshop on CTP Components for Educational Software, THedu'11, Wroclaw, Poland, 31th July 2011*, *EPTCS* 79, pp. 143–153, doi:10.4204/EPTCS.79.9.

[22] Markus Wenzel (1999): *Isar - a Generic Interpretative Approach to Readable Formal Proof Documents*. In G. Dowek, A. Hirschowitz, C. Paulin & L. Thery, editors: *Theorem Proving in Higher Order Logics*, LNCS 1690, 12th International Conference TPHOLs'99, Springer, doi:10.1007/3-540-48256-3_12.

[23] W.Neuper, B.Stöger & M.Wenzel (2022): *Towards Accessible Formal Mathematics with ISAC and Isabelle/VSCode*. Isabelle Workshop 2022 `https://sketis.net/isabelle/isabelle-workshop-2022`. Accessed: 202-10-13.