The 2007 Federated Conference on

Rewriting, Deduction and Programming

Paris, France

June 25 – 29, 2007

PATE'07

International Workshop on Proof Assistants and Types in Education

June 25th, 2007

Proceedings

Editors:

Herman Geuvers (Nijmegen, NL)
Pierre Courtieu (CNAM, Paris, France)

Program Committee:

Herman Geuvers (co-chair)
Pierre Courtieu (co-chair)
Hugo Herbelin (INRIA Paris, France)
Adam Naumowicz (Białystok, Poland)
Claudio Sacerdoti Coen (Bologna, Italy)
Pawel Urzyczyn (Warsaw, Poland)

Preface

We would like to thank the institutional sponsors of RDP'07 without whom it would not have been possible to organize RDP'07: the Conservatoire des Arts et Métiers (CNAM), the Centre National de la Recherche Scientifique (CNRS), the École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE), the GDR Informatique Mathématique, the Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs, and the Région Île de France.

Contents

William Farmer (Invited Speaker)
The Use of Formal Reasoning Technology in Mathematics Education: Opportunities and Challenges 1

Claudio Sacerdoti Cohen, Enrico Zoli
A Note on Formalising Undefined Terms in Real Analysis 3

Agnieszka Kozubek and Paweł Urzyczyn
In the search of a naive type theory 17

Cezary Kaliszyk, Freek Wiedijk, Maxim Hendriks, Femke van Raamsdonk
Teaching logic using a state-of-the-art proof assistant 33

Serge Autexier and Marc Wagner
Status Report on the Tight Integration of a Scientific Text-Editor and a Proof Assistance System 49

Adam Naumowicz
How to Teach to Write a Proof 65

Jakub Sakowicz and Jacek Chrząszcz
Papuq: a Coq assistant 75

Jérémy Blanc and J.P. Giacometti and André Hirschowitz and Loïc Pottier
Proofs for freshmen with Coqweb 93

René David and Christophe Raffalli
Some considerations about proof assistants for education 108

The Use of Formal Reasoning Technology in Mathematics Education: Opportunities and Challenges

William Farmer

McMaster University

Abstract

Abstract: Mathematics education is a very attractive market for applications of formal languages and reasoning tools. The number of people who study mathematics in school and university is enormous, and there is strong support for improving mathematics education with the aid of computer technology. There are also excellent opportunities to use formal reasoning technology to enhance, if not transform, how students learn mathematics. For example, proof assistants offer a way to reinvigorate the teaching of proof and deductive reasoning in high school and university. Great as these opportunities may be, they are not easy to pursue. Many challenges stand in their way. Not the least of which is the wide-spread scepticism with which formal reasoning technology is regarded in the mathematics community. This talk will discuss these opportunities and challenges and will argue that bold applications, like an interactive mathematics laboratory based on formal reasoning technology, have the best chance of realizing the opportunities and overcoming the challenges.

Invited Talk

A Note on Formalising Undefined Terms in Real Analysis

Claudio Sacerdoti Coen[1,2] Enrico Zoli[1,3]

Department of Computer Science
University of Bologna
Bologna, Italy

Abstract

To adopt proof assistants based on logic of total functions in education, one major problem is that of representing partial functions. In particular, one wishes to capture undefinedness in a rigorous way, while staying as close as possible to traditional mathematical practice. In this paper, we propose to represent potentially undefined terms with partial setoids on lifted terms, and to understand book equalities occurring in equality chains as special directed relations that hold only under assumptions of definedness for some terms. We also employ a suitable notion of strict morphism to fully automate propagation of these directed relations in strict contexts.

Keywords: undefinedness, strictness, proof assistant, Real Analysis, education, Matita

1 Introduction

DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) is a recently started locally funded project of the University of Bologna aimed at the development of a self-assessment tool for students of a first Real Analysis course. The tool will be able to check the correctness of proofs and provide hints or counterexamples. It should also force the student to prove every necessary side condition in computational exercises.

The tool, still under development, will be based on the core of the Matita interactive theorem prover [1], also under development by the research group of Prof. Asperti at the University of Bologna. Matita is based on the Calculus of (Co)Induction Constructions, a logic of total functions. Since partial functions play a major role in Real Analysis, the first problem we have to address is that of the representation of partial functions.

[1] Partially supported by the strategic project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

[2] sacerdot@cs.unibo.it

[3] zoli@math.unifi.it

Several papers have already been devoted to the subject. Müller and Slind in [6] provide a good comparison of the techniques proposed and give many useful references. However, in didactics we cannot adopt those solutions that are far from the usual mathematical practice. The latter, called in [4] the *traditional approach to undefinedness*, can be summarized as: 1) atomic terms (variables and constants) are always defined; 2) a function application is undefined either when the argument or the function is undefined or when the function is undefined on the argument; 3) formulas are always defined, and are false when undefined terms occur within them.

Among the proposals close to the usual mathematical practice, Farmer's [4] is certainly very interesting, but it requires an ad-hoc logic with undefinedness. A similar remark applies to other proposals requiring ad-hoc logics, e.g. [7]. Since we do not plan to change the logic of Matita, we cannot adopt these solutions. In this paper, we therefore encode partial functions as total functions on lifted terms. To hide the details of this formalisation from the student, we exploit an automation tactic that deal with partial setoids [8]. Indeed, lifted terms naturally form a partial setoid, when equipped with the partial equivalence relation that coincides with equality on defined terms, and is false otherwise.

The main contribution of the paper is the observation that there is more than reasoning on partial setoids in the traditional handling of equality chains over potentially undefined terms. Indeed, in equality chains some steps also reduce the proof of definedness of one side to that of the other. Hence we need to extend automation to fully capture this form of reasoning.

We claim that in practice our approach is suitable for didactics. However, it is still to be completely automated and tested extensively on students.

To be more concrete, consider the following running example as it usually appears in textbooks.

Theorem 1.1 *Suppose that x is a real number with $|x| < 1$. Then $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$.*

Proof. $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$ by an easy induction over n. Thus

$$\begin{aligned}
\sum_{n=0}^{\infty} x^n &= \lim_{n\to\infty} \sum_{i=0}^{n} x^i \\
&= \lim_{n\to\infty} \frac{1-x^{n+1}}{1-x} \\
&= \frac{\lim_{n\to\infty}(1-x^{n+1})}{1-x} \\
&= \frac{1-\lim_{n\to\infty} x^{n+1}}{1-x} \\
&= \frac{1}{1-x}.
\end{aligned}$$

□

In our opinion, this exercise is interesting for it consists in two distinct parts: the first is a simple proof by induction that exercises proving skills; the second part is an easy computational proof where the main difficulty (completely hidden in the text) is to check that every expression is well-defined. Note that in the given proof

no rewriting step is explicitly justified at all. Therefore, in our judgement, it is too lax for an exercise solution, since we expect students to show at least where the hypotheses are employed.

The first question about this exercise we should pose ourselves concerns our teaching goals when proposing it to the student. We identify some alternatives in Sect. 2, and give a rigorous account of the justification of each proof step. Formalisation in a logic of total functions is discussed in Sect. 3. In the same section we also discuss how to provide automation to hide the intricacies of the formalization, allowing natural manipulation of formulas. Automation is achieved by coupling the ideas in [8] with the notion of *strict morphism*, i.e., a function that preserves equalities and undefinedness. Conclusions follow in Sect. 4.

2 Teaching goals

In principle, proof assistants can become a useful aid for self-assessment of proving skills. However, the gap between the usual pen&paper mathematical practice and the formalised mathematics these systems understand is significant and difficult to fill. Two phenomena yield this gap: the lack of absolute rigour in the usual manipulation of formulas; the limitations of formal systems, which render formalisation difficult and formalised proofs obscure. We postpone discussion on the second phenomenon to Sect. 3.

The first phenomenon is already evident in our running example where we manipulate limits of sequences before proving convergence of the sequences. Similarly, it often happens to compute derivatives of functions before proving their differentiability. Admittedly, an excess of rigour may lead to drown the essence of a proof in pedantic details. However, we argue that a first Real Analysis course for students in Mathematics or Computer Science should both frame rigorously their minds — our first teaching goal — and teach effective high-level formulas manipulation — our second teaching goal. The two teaching goals may require different sets of exercises (or different approaches to them).

For instance, our running example could be proposed as an exercise twice. The first time, just after the introduction of the "tends to" relation, where no partial operator is employed and each limit must explicitly be shown (or assumed) to exist. The expected solution to the exercise would be the following:

Theorem 2.1
Suppose that x is a real number with $|x| < 1$. Then $\sum_{i=0}^{n} x^i \xrightarrow[n\to\infty]{} \frac{1}{1-x}$.

Proof.
We prove $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$ by induction over n.
...
Moreover $x^{n+1} \xrightarrow[n\to\infty]{} 0$ since $|x| < 1$.
Hence $1 - x^{n+1} \xrightarrow[n\to\infty]{} 1$.
Hence $\frac{1-x^{n+1}}{1-x} \xrightarrow[n\to\infty]{} \frac{1}{1-x}$ as $1 - x \neq 0$.
Hence the thesis. □

Note that, with respect to Theorem 1.1, the equality chain has been replaced

with a chain of implications. Moreover, the two chains proceed in opposite order. The chosen order appears more natural in this context, because we need to rigorously prove the existence of limits before computing with them. The two proofs are different in an even more radical sense: in the original proof we can see formulas, such as $\frac{1-\lim_{n\to\infty} x^{n+1}}{1-x}$, that have no exact counterpart in the new proof. Note also the need to explicitly prove denominators to be different from 0, because of the lack of undefined terms (which is consistent with the choice of avoiding the partial limit operator).

It is possible to follow the original proof a little more closely by adopting the original order of the chain. However, the solution looks rather artificial:

Proof.
...
We have $1 - x \neq 0$ since $|x| < 1$.
We need to prove $\frac{1-x^{n+1}}{1-x} \xrightarrow[n\to\infty]{} \frac{1}{1-x}$.
It is sufficient to prove $1 - x^{n+1} \xrightarrow[n\to\infty]{} 1$.
It is sufficient to prove $x^{n+1} \xrightarrow[n\to\infty]{} 0$.
The thesis follows from $|x| < 1$. □

To reach our second teaching goal, we could propose the same exercise a second time just after the introduction of the limit operator. This time the emphasis is on exercising the students on effective limit computation, where (usually) as many side conditions as possible are avoided. Thus, we look for a solution quite adherent to that of Theorem 1.1. In particular, we admit the existence of undefined terms to be handled according to the traditional approach discussed in the introduction.

In Theorem 1.1, the definedness of each expression containing a partial operator (a limit or a division) is not justified. Consistently with our first teaching goal, we would like to avoid such a lax approach, but we look for lightweight justifications. We propose to make sense of the proof by assuming that, at each step in the equation chain, ground for the definedness of the left hand side is derived by the assumption that the right hand side will be proved to be defined later on. This property does not always hold in general, unless side conditions are required. E.g., if $\frac{ax}{a}$ is defined, then so is x and $x = \frac{ax}{a}$; but the definedness of x yields neither that of $\frac{ax}{a}$ nor the equality $x = \frac{ax}{a}$. The latter property holds when the additional hypothesis $a \neq 0$ is assumed. Thus, replacing $\frac{ax}{x}$ with x in an equality chain or the other way around yields different side conditions; this suggests that, when terms can possibly be undefined, rewriting chains are not symmetric. As far as we know, this phenomenon has not extensively been studied in the literature. We propose to understand it rigorously by introducing two oriented relations: $\overset{\triangleright}{=}$ and its dual $\overset{\triangleleft}{=}$.

Definition 2.2
$x \overset{\triangleright}{=} y$ when x is defined if y is, and in this case they are equal.
$x \overset{\triangleleft}{=} y$ when $y \overset{\triangleright}{=} x$ or equivalently when y is defined if x is, and in this case they are equal.

These relations are oriented variants of Farmer's *quasi equality* relation: x is quasi equal to y when x and y are either both undefined or equal. Quasi equality

can be reduced to our definitions: x is quasi equal to y if $x \stackrel{\rhd}{=} y$ and $x \stackrel{\lhd}{=} y$. Therefore we call our relations *directed quasi equalities*.

Our running example can now be rigorously justified using $\stackrel{\rhd}{=}$:

Proof.
...

$$\begin{aligned}
\sum_{n=0}^{\infty} x^n &\stackrel{\text{def}}{=} \lim_{n\to\infty} \sum_{i=0}^{n} x^i \\
&\stackrel{\rhd}{=} \lim_{n\to\infty} \frac{1-x^{n+1}}{1-x} \\
&\stackrel{\rhd}{=} \frac{\lim_{n\to\infty}(1-x^{n+1})}{1-x} && (1)\\
&\stackrel{\rhd}{=} \frac{1-\lim_{n\to\infty} x^{n+1}}{1-x} && (2)\\
&\stackrel{\rhd}{=} \frac{1}{1-x} \text{ since } |x|<1 && (3)\\
&\quad \text{which is defined since } |x|<1.
\end{aligned}$$

□

It is to be noted that, since the existence of each expression is reduced by $\stackrel{\rhd}{=}$ to the definedness of $\frac{1}{1-x}$, in the final step we have to show this term to be defined in order to conclude $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$. Without this justification, the final conclusion of the equality chain would become (the weaker) $\sum_{n=0}^{\infty} x^n \stackrel{\rhd}{=} \frac{1}{1-x}$.

In general, given an equality chain

$$\ldots \stackrel{\rhd}{=} M_1 \stackrel{\lhd}{=} E_1 \stackrel{\lhd}{=} \ldots \stackrel{\lhd}{=} E_n \stackrel{\lhd}{=} T_1 \stackrel{\rhd}{=} E_{n+1} \stackrel{\rhd}{=} \ldots \stackrel{\rhd}{=} E_m \stackrel{\rhd}{=} M_2 \stackrel{\lhd}{=} E_{m+1} \stackrel{\lhd}{=} \ldots,$$

to conclude that all terms in the chain are equal (and thus defined), we have to prove that every term M_i is defined. This done, we recursively obtain that any other term in the equality chain is defined and, ultimately, that all terms are equal.

More formally, an equality chain fragment $E_1 \ \mathcal{R} \ E_2 \ \mathcal{R}' \ E_3$ must be always understood as an application of an ad-hoc generalized transitivity principle $\forall x, y, z. x \ \mathcal{R} \ y \wedge y \ \mathcal{R}' \ z \Rightarrow x \ \mathcal{S} \ z$, where the relation S depends on R and R' and is usually the strongest of them. The following generalized transitivity principles hold for our directed quasi equalities:

Lemma 2.3 *For x, y, z potentially undefined terms*

$$
\begin{aligned}
&x \stackrel{\rhd}{=} y \wedge y \stackrel{\rhd}{=} z \Rightarrow x \stackrel{\rhd}{=} z\\
&x \stackrel{\lhd}{=} y \wedge y \stackrel{\lhd}{=} z \Rightarrow x \stackrel{\lhd}{=} z\\
&x = y \wedge y \stackrel{\lhd}{=} z \Rightarrow x = z\\
&x \stackrel{\rhd}{=} y \wedge y = z \Rightarrow x = z\\
&x \stackrel{\rhd}{=} y \wedge y \stackrel{\lhd}{=} z \Rightarrow x = z \quad \textit{when } y \textit{ is defined}\\
&x \stackrel{\lhd}{=} y \wedge y \stackrel{\rhd}{=} z \Rightarrow x = z \quad \textit{when } x \textit{ and } z \textit{ are defined}
\end{aligned}
$$

The proof of this lemma is trivial (just recall that in the traditional approach two terms are equal only when they are both defined and have the same value).

As another example that employs an equality chain, let us look at the corresponding proof obtained by reversing the direction of the chain:

Proof.

...

Since $|x| < 1$ the expression $\frac{1}{1-x}$ is defined. Hence

$$
\begin{aligned}
\frac{1}{1-x} &\stackrel{\lhd}{=} \frac{1 - \lim_{n\to\infty} x^{n+1}}{1-x} \text{ since } |x| < 1\\
&\stackrel{\lhd}{=} \frac{\lim_{n\to\infty}(1 - x^{n+1})}{1-x}\\
&\stackrel{\lhd}{=} \lim_{n\to\infty} \frac{1 - x^{n+1}}{1-x}\\
&\stackrel{\lhd}{=} \lim_{n\to\infty} \sum_{i=0}^{n} x^i\\
&\stackrel{\text{def}}{=} \sum_{n=0}^{\infty} x^n
\end{aligned}
$$

□

This time, the expression that is directly shown to be defined is the leftmost end of the chain.

Observe that the corresponding rewriting steps in the direct and backward proof are justified by exactly the same lemma, since $x \stackrel{\rhd}{=} y$ is equivalent to $y \stackrel{\lhd}{=} x$ (because of the symmetry of the equality). For instance, both (1) $\stackrel{\rhd}{=}$(2) and (2) $\stackrel{\lhd}{=}$(1) are justified by

Lemma 2.4 *For all c and all real sequences $f(n)$,*

$$\lim_{n\to\infty} (c - f(n)) \stackrel{\rhd}{=} c - \lim_{n\to\infty} f(n)$$

The lemma used to justify (2) $\stackrel{\rhd}{=}$(3) as well as (3) $\stackrel{\lhd}{=}$(2) is

Lemma 2.5 *If $|x| < 1$, then $\lim_{n\to\infty} x^n \stackrel{\rhd}{=} 0$.*

Note that the hypothesis $|x| < 1$ is to be explicitly justified in the equality chain by a side proof. Incidentally, the following corresponding "dual" lemma (that could justify (2) $\stackrel{\lhd}{=}$(3) or (3) $\stackrel{\rhd}{=}$(2)) has no extra hypotheses, as the definedness of $\lim_{n\to\infty} x^n$ already gives $|x| < 1$.

Lemma 2.6 $0 \stackrel{\rhd}{=} \lim_{n\to\infty} x^n$.

Finally, we would like to spend a few additional words on teaching goals. So far, we have recognized two teaching goals in adopting proof assistants and types in mathematical education. The first one is the emphasis on mathematical rigour, so as to augment both the comprehension of the role played by axioms and hypotheses in inferences and the awareness of common pitfalls in mathematical reasoning. The second one is the enhancement of skills in mathematical practice, where rigour is not forced. In both cases it is important for a didactic tool to hide all the difficulties arising from the formalisation process. In particular, we expect the system to provide enough automation to completely hide from the student the side proofs arising from technical side conditions. These are justifications we do not expect to see on paper, independently from the teaching goal.

We can also identify a third alternative goal, i.e., teaching (mathematical) formalisation techniques, which are largely independent from the mathematical subject being formalized, and definitely outside standard curricula in Mathematics and Computer Science. For this reason we will not address this subject any longer.

The exercise dealing with the "tends to" relation can be straightforwardly formalised in a logic of total functions, provided that division requires either a proof that the quotient is not 0 or a formalisation via alternative techniques such as, e.g., Σ-types. The one that deals with the limit operator requires a preliminary encoding of undefined terms and operators, and a substantial amount of automation to hide the intricacies of this encoding. This is the subject of the next section.

3 Formalisation

Plenty of functions and operators in elementary calculus are partial. The following are examples of undefined expressions: division by zero; supremum of an unbounded set; limit of an oscillating sequence; value of a non-converging series; derivative of a discontinuous function; definite integral of a non-integrable function.

The usual "lax" mathematical practice manipulates partial expressions in a way that is often difficult to understand rigorously. In Sect. 1 we have partially discussed the issue by sticking to what Farmer calls "the traditional approach to undefinedness", enriched by two new directed quasi equality relations $\stackrel{\rhd}{=}$ and $\stackrel{\lhd}{=}$ that replace book equality in chains of equations. However, we have kept the description at an informal level. We now address the problem of capturing these ideas in a formal system.

In the unlikely case that our system is based on a logic with undefinedness [4] or a three value logic [7], the formalisation proceeds very smoothly: the logic is already aware of undefined terms, equality is already the proposed relation, $\stackrel{\rhd}{=}$ and $\stackrel{\lhd}{=}$ can be directly defined and appropriate replacement principles are likely to be

derivable.

When the logic is a logic of totally defined terms, we need a way to represent undefinedness and we need to change the equality (and every other predicate) to behave correctly over undefined terms. Since equality is replaced by a coarser relation, it is no longer true that $x = y \Rightarrow C[x] \iff C[y]$ for each context C. In order to keep term manipulation simple, the system is now in charge of automatically proving the side conditions that make the previous implication true for a particular C. In [8], the first author has given the Coq proof assistant this explicit support in the case of generalized setoids, including partial setoids. Partial setoids can be exploited in our context, since the well-known representation of potentially undefined terms of type T as elements of the lifted type $T_\perp \stackrel{\text{def}}{=} T \oplus \{\perp\}$ gives rise to a partial setoid.

Even when partial setoids automation is exploited, the user is still requested to prove many side conditions that are implicitly discharged in the mathematical practice, and that we plan to avoid basing our equality chains on $\stackrel{\rhd}{=}$ and $\stackrel{\lhd}{=}$. Hence, we now provide automation for $\stackrel{\rhd}{=}$ and $\stackrel{\lhd}{=}$. Let us recall the notion of partial setoid and the associated morphisms.

A *partial setoid* is a couple $(T, \simeq_T)$, where T is a type and $\simeq_T$ is a symmetric and transitive relation over T. An element x of T is said to be *proper* when $x \simeq_T x$. We will also write $x \downarrow^T$ for $x \simeq_T x$. We drop the subscript/superscript T when the partial equality relation is clear from the context.

A *morphism* of partial setoids $(S, \simeq_S), (T, \simeq_T)$ is a function $f : S \to T$ such that $f(x) \simeq_T f(y)$ whenever $x \simeq_S y$. We will write $f : (S, \simeq_S) \Rightarrow (T, \simeq_T)$ for f being a morphism between $(S, \simeq_S)$ and $(T, \simeq_T)$. With a little notational abuse, we will say that f has type $(S, \simeq_S) \Rightarrow (T, \simeq_T)$.

If $(S, \simeq_S)$ and $(T, \simeq_T)$ are partial setoids, so is $(S, \simeq_S) \times (T, \simeq_T) \stackrel{\text{def}}{=} (S \times T, \simeq_S \times \simeq_T)$ (the cartesian product equipped with the partial equivalence relation induced by $\simeq_S$ and $\simeq_T$ acting componentwise). The type **Prop** of propositions together with coimplication forms the (total) setoid (**Prop**, $\iff$).

Alternative formulations of the notion of partial setoids are explored in [3]. The one we adopt is that of Bailey [2].

For each type T, we represent potentially undefined terms of T with elements of the lifted type $T_\perp$. We equip $T_\perp$ with a partial setoid structure by defining the partial equivalence relation $\simeq$ as $x \simeq y$ iff $x, y \neq \perp$ and $x = y$ as elements of T. According to this definition, proper elements are defined elements and we can read $x \downarrow$ as "x is defined". Moreover, from $x \simeq y$ we immediately know $x \downarrow$ and $y \downarrow$ since for no x, not even $\perp$, we have $\perp \simeq x$.

The reader acquainted with the work of Farmer should note that our notation is not consistent with [4]: he denotes our relation $\simeq$ with $=$, whereas we reserve the symbol $=$ for either Leibniz's or book equality; moreover, he uses $\simeq$ for quasi equality, whereas we only have special symbols for directed quasi equalities. Finally, in Farmer's logic with undefinedness all atoms are implicitly defined, while our variables are always quantified over $T_\perp$ for some T. Thus, according to the terminology used by Feferman in [5], Farmer's logic is a logic of definedness, while we are encoding a logic of existence.

The following facts over morphisms are exploited in manual and automatic proofs over setoids. The identity function over S is always a morphism of type $(S, \simeq) \Rightarrow (S, \simeq')$ whenever $\simeq$ and $\simeq'$ are partial equivalence relations over S such that the graph of $\simeq$ is included in the graph of $\simeq'$ (i.e. $x \simeq x'$ implies $x \simeq' x'$ for each x, x'). The constant function $f(x) \stackrel{\text{def}}{=} c$ of type $S \to T$ is a morphism of type $(S, \simeq_S) \Rightarrow (T, \simeq_T)$ iff $c \downarrow^T$. The function $\mathcal{C}_f(x) \stackrel{\text{def}}{=} f(x, x)$ is a morphism of type $(S, \simeq_S) \Rightarrow (T, \simeq_T)$ whenever f is a morphism of type $(S, \simeq_S) \times (S, \simeq_S) \Rightarrow (T, \simeq_T)$. Composition of morphisms is a morphism. Finally, any partial equivalence relation $\simeq$ over S is a morphism of type $(S, \simeq) \times (S, \simeq) \Rightarrow (\mathbf{Prop}, \iff)$, since it is symmetric and transitive (proof given in [8]).

Let us now consider how to formalise our running example in this setting. Subtraction between real numbers is represented as a morphism of type $(\mathbb{R}_\perp, \simeq) \Rightarrow (\mathbb{R}_\perp, \simeq)$; exponentiation as a morphism of type $(\mathbb{R}_\perp, \simeq) \Rightarrow (\mathbb{N}, =) \Rightarrow (\mathbb{R}_\perp, \simeq)$; the limit operator as a morphism of type $(\mathbb{N} \to \mathbb{R}_\perp, \simeq_\to)$, where $f \simeq_\to g$ when $f(n) \simeq g(n)$ for each n; finally, division is represented as a morphism of type $(\mathbb{R}_\perp, \simeq) \times (\mathbb{R}_\perp, \simeq_0) \Rightarrow (\mathbb{R}_\perp, \simeq)$ where $x \simeq_0 y$ iff $x \simeq y$ and $x \neq 0$. Note that the identity function over real numbers is a morphism from $(\mathbb{R}, \simeq_0)$ to $(\mathbb{R}, \simeq)$; hence, we can compose a morphism of type $(\mathbb{R}, \simeq) \Rightarrow (S, \simeq_S)$ with a morphism of type $(T, \simeq_T) \Rightarrow (\mathbb{R}, \simeq_0)$. Note also that it is impossible to prove division to be a morphism of type $(\mathbb{R}_\perp, \simeq) \times (\mathbb{R}_\perp, \simeq) \Rightarrow (\mathbb{R}_\perp, \simeq)$, as $1 \simeq 1$ and $0 \simeq 0$ but $1/0$ is $\perp$ and it is not true[4] that $\perp \simeq \perp$.

Consider the third equality in our running example:

$$\frac{\lim_{n\to\infty}(1 - x^{n+1})}{1 - x} \stackrel{\trianglerighteq}{=} \frac{1 - \lim_{n\to\infty} x^{n+1}}{1 - x}$$

and suppose that Lemma 2.4 has already been proved. To justify the equality we need to contextualise (an instance of) the lemma to the context $C(w) \stackrel{\text{def}}{=} \frac{w}{1-x}$. In other words, we need to prove

$$\lim_{n\to\infty}(1 - x^{n+1}) \stackrel{\trianglerighteq}{=} 1 - \lim_{n\to\infty} x^{n+1} \Rightarrow \frac{\lim_{n\to\infty}(1 - x^{n+1})}{1 - x} \stackrel{\trianglerighteq}{=} \frac{1 - \lim_{n\to\infty} x^{n+1}}{1 - x}$$

or, more generally, that $x \stackrel{\trianglerighteq}{=} y \Rightarrow (C(x) \Rightarrow C(y))$. Adopting the terminology of [8], we need to prove that $C(\cdot)$ is a covariant morphism from the partial and asymmetric setoid $(\mathbb{R}, \stackrel{\trianglerighteq}{=})$ to the asymmetric setoid $(\mathbf{Prop}, \Rightarrow)$. This can be done, even automatically, exploiting all the lemmas in [8]. However, to our knowledge Coq is the only system that implements automation over "generalised" setoids (i.e., setoids whose relation is not required to be reflexive, symmetrical or transitive). Moreover, when asymmetric setoids are employed, user-provided proofs that basic morphisms are indeed morphisms become more involved. Therefore, we now propose a simpler alternative automation technique that does not require asymmetric setoids.

To avoid the latter, we can expand the definition of $\stackrel{\trianglerighteq}{=}$ both in the lemma and

[4] We do not write $\perp \not\simeq \perp$ since that latter symbol is better understood as *book diversity*, i.e., $x \not\simeq y$ iff x and y are both defined and different. Thus, it is not true that $\perp \not\simeq \perp$.

in the equality to be justified; they become, respectively,

$$\forall c, f.(c - \lim_{n\to\infty}) \downarrow \Rightarrow \lim_{n\to\infty}(c - f(n)) \simeq c - \lim_{n\to\infty} f(n)$$

$$\frac{1 - \lim_{n\to\infty} x^{n+1}}{1-x} \downarrow \Rightarrow \frac{\lim_{n\to\infty}(1 - x^{n+1})}{1-x} \simeq \frac{1 - \lim_{n\to\infty} x^{n+1}}{1-x}$$

The equality can now be justified, thanks to the lemma and the definition of a morphism, by assuming (H) $\frac{1-\lim_{n\to\infty} x^{n+1}}{1-x} \downarrow$ and showing:

(i) $(1 - x) \downarrow$ and $(1 - \lim_{n\to\infty} x^{n+1}) \downarrow$. Both are consequences of the assumption (H). The second proof is used to discharge the hypothesis of the lemma.

(ii) $C(w) \stackrel{\text{def}}{=} \frac{w}{1-x}$ is a morphism of type $(\mathbb{R}_\perp, \simeq) \Rightarrow (\mathbb{R}_\perp, \simeq)$.
This is of course the case, for $id(w) \stackrel{\text{def}}{=} w$ is an identity morphism, $c(w) \stackrel{\text{def}}{=} 1 - x$ is a constant morphism returning the element $1 - x$ (which is defined because of (i)), division is a morphism, and $C(w)$ is a composition of division, $c(\cdot)$ and the identity morphism.

When implementing automation over partial setoids as explained in [8], (ii) is proved automatically by the system. Hence we only need to provide automation for (i). Now, (i) holds since we can prove that $\forall n, d.\frac{n}{d} \downarrow \Rightarrow n \downarrow \wedge d \downarrow$. In other words, if the application of division to some arguments yields a defined term, then the arguments are defined as well. This is a general property of morphisms and predicates in the traditional approach to undefinedness, and it can be recognized to be just *strictness* (see, for instance, [5]):

Definition 3.1
A morphism $f : (S_\perp, \simeq_S) \Rightarrow (T_\perp, \simeq_T)$ is *strict* if $f(x) \downarrow^T$ implies $x \downarrow^S$.
A morphism $P : (S_\perp, \simeq) \Rightarrow (\mathbf{Prop}, \iff)$ is strict if $P(x)$ implies $x \downarrow$.

The partial equivalence relation $\simeq_S$ over $S_\perp$ is strict for each type S. Moreover, all morphisms in our running example are strict. For instance, if $x - y$ is defined then both x and y are.

The following facts help showing that a predicate is strict and can be easily exploited in a tactic that proves syntactic composition of morphisms to be strict. The identity morphism is strict, whereas constant morphisms are not. Composition of strict morphisms yields a strict morphism. The morphism $f(x) \stackrel{\text{def}}{=} (g(x), h(x))$ is strict on x if (at least) one of $g(x)$ and $h(x)$ is. The morphism $P(x) \vee Q(x)$ on x is strict if $P(x)$ and $Q(x)$ are; $P(x) \wedge Q(x)$ is strict on x if $P(x)$ is strict or if $Q(x)$ is; $\exists y.P(x, y)$ is strict on x whenever $P(x, y)$ is strict on x for each y; moreover, $\forall y.P(x, y)$ is strict on x if there exists a y such that $P(x, y)$ is strict on x. There is no relation between the strictness of $P(x)$ and that of $\neg P(x)$. In particular, in the traditional approach to undefinedness all predicates over undefined terms are false, so that the negative form of a predicate $P(x)$ is not the logical negation of P, but the predicate "$x \downarrow \wedge \neg P(x)$". In turn, we obtain the following restriction over implication: the predicate $P(x) \Rightarrow Q(x)$ is strict on x if $Q(x)$ is strict on x and P is constant over x. The latter check is necessary, as the following counterexample shows: $x \simeq x \Rightarrow x \simeq x$ always holds, but it does not imply $x \downarrow$.

Similarly to what the first author did in [8], we can easily implement a semi-reflexive tactic to prove automatically when a morphism $P : (S, \simeq) \Rightarrow (\mathbf{Prop}, \iff)$, that is a syntactic composition of strict and non-strict morphisms, is strict in one argument. The tactic is based on the lemmas above.

Exploiting such a tactic, we can specialise [8] (or its restriction to symmetric and transitive setoids) to automatically prove, for a context P that is a suitable syntactic composition of strict and non strict morphisms,

$$P(x) \Rightarrow (y \stackrel{\rhd}{=} x) \Rightarrow P(y)$$

Coming back to our running example, we can now use automation to reduce, for instance, a proof of $\frac{\lim_{n\to\infty}(1-x^{n+1})}{1-x} \simeq \frac{1}{1-x}$ to a proof of $\frac{1-\lim_{n\to\infty} x^{n+1}}{1-x} \simeq \frac{1}{1-x}$ by means of a proof of $\lim_{n\to\infty}(1 - x^{n+1}) \stackrel{\rhd}{=} 1 - \lim_{n\to\infty} x^{n+1}$ and no other side condition.

For the sake of generality, we remark that strict morphisms are a special case of *co-guarded morphisms*.

Definition 3.2
A morphism $f : (S_\perp, \simeq_S) \Rightarrow (T_\perp, \simeq_T)$ is *Q-co-guarded* if $f(x) \downarrow^T \Rightarrow Q(x)$.

Strict morphisms are $\downarrow$-co-guarded morphisms. Note that for a strict morphism $P : (S, \simeq) \Rightarrow (\mathbf{Prop}, \iff)$ the condition $P(x) \downarrow \iff$ can be omitted since it is not informative.

For a Q-co-guarded strict morphism P over **Prop**, we can specialise [8] to automate proofs of

$$P(x) \Rightarrow (Q(x) \Rightarrow y \simeq x) \Rightarrow P(y)$$

So far, the only interesting class of co-guarded morphisms we currently know is that of strict morphisms.

To summarise, we suggest how to formalise chains of directed quasi equalities by automating "directed rewritings" justified by lemmas of the form $\forall \bar{x}, P(\bar{x}) \Rightarrow F(\bar{x}) \stackrel{\rhd}{=} F'(\bar{x})$, being $\bar{x}$ a vector of variables and the extra hypotheses $P(\cdot)$ justified as side conditions in the chain. Since $F(\bar{x}) \stackrel{\rhd}{=} F'(\bar{x})$ is equivalent to $F'(\bar{x}) \downarrow \Rightarrow F(\bar{x}) \simeq F'(\bar{x})$, a number of lemmas can exploit the hypothesis $F'(\bar{x}) \downarrow$ to drop many extra conditions such as the definedness of quantified variables occurring in $F'(\bar{x})$.

For instance, in our running examples all lemmas can be given in this form and only Lemma 2.5 requires an extra hypothesis. As a comparison, corresponding lemmas in Farmer's approach [4] would also miss the additional hypotheses, but for a different reason: since his logic is a logic of definedness, all quantified variables are implicitly assumed to be defined.

We finally add that we have not discussed two other simple and well-known approaches to undefined terms (not far away from the usual mathematical practice, but too lax according to our first teaching goal). Contrary to the traditional approach to undefinedness, in both approaches some dubious properties can hold even for undefined terms. The first approach extends every partial function and operation to the whole domain by assigning an arbitrary value outside the original

domain. For instance, $1/0$ becomes a perfectly valid real number, say 17, that enjoys the property $(1/0)/17 = 1$. The approach is often used in HOL, Isabelle and Mizar, and always in ACL2. The second approach extends axiomatically every partial function and operation to the whole domain, but this value remains unknown by prefixing each axiom of the theory with hypotheses that avoid characterising the partial functions outside their domain of definitions. For instance, it is true that $1/0$ is a perfectly valid number, but we can never get rid of the division since $(a/b) * b = a$ iff $b \neq 0$. However, all universal properties are enjoyed also by "undefined" terms: $0 * (1/0) = 0$ since $\forall x, 0 * x = 0$, $(1/0) = (1/0)$ as $\forall x, x = x$, etc. Even if these properties are unlikely to be noticed by the student, it may happen to unconsciously employ them, for instance omitting side conditions that are made unnecessary by the additional properties. Users of HOL, Isabelle and Mizar often mix the two approaches.

Coq, NuPRL and PVS, being based on dependent types, can use yet another approach: they can encode the domain of a partial function in its type, so that application to arguments outside the domain is not allowed by the type system. We have not considered this approach in the paper since it is not well suited for students for several reasons. For instance, applications of partial functions require a new argument that is the proof that the argument belongs to the domain of the function. First year students do not understand that derivations can be represented syntactically and used as arguments to functions; moreover, when a mathematical operator has a standard notation, there is no natural place where the new argument can be put. PVS avoids this problem by removing the extra arguments and replacing them with side proofs for domain conditions. Wiedijk and Zwanenburg in [9] apply the approach of PVS to obtain a first order logic with domain conditions, without dependent types. Looking at the examples in the paper it is pretty obvious that the user is faced with too many side conditions. We believe that their number can be highly reduced in equality chains exploiting directed quasi equalities.

4 Conclusions

The most important problem to be faced when adopting proof assistants and types in education is probably that of representating and manipulating undefined expressions. In the case that adopting a logic with undefinedness is not an option, we are led to represent partial functions and operators in logics of total functions. Many formalisations have been proposed in the literature, but most of them are of limited use in education. We suggest that types of potentially undefined terms can be suitably represented with partial setoids and that partial functions can be represented as morphisms over partial setoids. This suggestion is not new: support for term manipulation in the partial setoids setting has already been implemented for Coq in [8]. However, in this setting equality chains require too many side conditions to be proved.

While analysing an informal equality chain that deals with potentially undefined terms we have noticed that a number of side conditions can be dropped by reducing definedness of both equality arguments to that of just one of them (to be proved later on). We have captured this behavior by introducing two oriented quasi

equalities $\stackrel{\triangleright}{=}$ and $\stackrel{\triangleleft}{=}$ that can be exploited in equality chains. We have also proposed the automation of the propagation of these equalities by combining automation for partial setoids with the notion of a strict morphism. A strict morphism is a morphism P such that $P(x)$ implies x defined. Hence, for a given strict morphism P, the system can automatically prove $P(x) \Rightarrow (y \stackrel{\triangleright}{=} x) \Rightarrow P(y)$, which means that P is a contra-variant morphism over the partial asymmetric relation $\stackrel{\triangleright}{=}$ (compare with [8]).

Thanks to automation, we have shown how an informal proof that computes the limit of a convergent series could be formalised in a proof assistant like Matita, providing only the minimal amount of rigorous justifications we expect from clever students even in a pen&paper proof.

As a future work, we plan to complete the implementation in Matita of the proposed automation and to test the proposed approach on a significant set of exercises to be submitted to a good number of students. We also plan a deeper comparison with alternative approaches. In particular, directed quasi equalities and their combinations with strict morphisms are likely to be exploitable with similar benefits in other contexts.

References

[1] A. Asperti, C. Sacerdoti Coen, E. Tassi, S. Zacchiroli. "User Interaction with the Matita Proof Assistant", in Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving. To appear.

[2] A. Bailey. "Representing Algebra in LEGO", M. Phil. thesis, University of Edinburgh.

[3] G. Barthe, O. Pons, V. Capretta. "Setoids in type theory", Journal of Functional Programming, 13(2), pages 261-293, 2003.

[4] W.M. Farmer. "Formalizing Undefinedness Arising in Calculus", in Automated Reasoning, Lecture Notes in Computer Science (LNCS), 3097:475-489, 2004.

[5] S. Feferman, "Definedness", Erkenntnis 43 (1995) 295-320.

[6] O. Müller, K. Slind. "Treating partiality in a logic of total functions", the Computer Journal, 40:640–652, 1997.

[7] M. Kerber, M. Kohlhase. "A Mechanization of Strong Kleene Logic for Partial Functions", CADE 1994: 371-385.

[8] C. Sacerdoti Coen, "A Semi-reflexive Tactic for (Sub-)Equational Reasoning", in Types for Proofs and Programs International Workshop, TYPES 2004. Lecture Notes in Computer Science (LNCS), Vol. 3839, 99–115, 2006.

[9] F. Wiedijk, J. Zwanenburg, "First order logic with domain conditions", in Theorem Proving in Higher Order Logics, Proceedings of TPHOLs 2003, D. Basin and B. Wolff (eds.), Springer LNCS 2758, 221-237, 2003.

In the search of a naive type theory [1]

Agnieszka Kozubek and Paweł Urzyczyn[2]

Institute of Informatics,
Warsaw University
Poland

Abstract

This paper argues that an appropriate "naive type theory" should replace naive set theory (as understood in Halmos' book) in everyday mathematical practice, especially teaching mathematics to Computer Science students. As an initial formalism to represent the basics of such a theory we propose a certain pure type system. The consistency of this system is established by proving strong normalization.

Keywords: Naive type theory, pure type systems.

1 Why not set theory?

Set theory is an enormous success in the contemporary mathematics, including the mathematics relevant to Computer Science. Virtually all maths is developed within the framework of set theory, and virtually all books and papers are written under the silent assumption of ZF or ZFC axioms occurring "behind the back". We sometimes feel as if we actually lived in set theory, as if it was the only true and real world.

The set-theoretical background has made its way to education, from the university to the kindergarten level, and what once was a foundational subject on the border of logic and philosophy now has become a part of elementary mathematics.

And indeed, set theory deserves its pride. From an extremely modest background—the notion of "being an element" and the idea of equality—it develops complex notions and objects serving the needs of even most demanding researcher. Enjoying the paradise of sets we tend to forget about the price we pay for that.

Of course, we must avoid paradoxes, and thus the set formation patterns are severely restricted. We must give up Cantor's idea of "putting together" any collection of objects, resigning therefore, at least partly, from the very basic intuition that a set of objects can be selected by any criterion at all.

[1] Partly supported by the Polish Government Grant 3 T11C 002 27, and by the EU Coordination Action 510996 "Types for Proofs and Programs".

[2] Email: {kozubek,urzy}@mimuw.edu.pl

Universes vs predicates
In fact, there are *two* very basic intuitions that are glued together into the notion of a "set":

- Set as a domain or universe;
- Set as a "materialization" of a predicate.

We used to treat this identification as natural and obvious. But perhaps only because we were *taught* to do it. These two ideas are in fact different, and this very confusion is responsible for Russel's paradox. In addition, ordinary mathematical practice often makes an explicit difference between the two aspects. Mathematicians have been classifying objects according to their domain, kind, sort or *type* since the antiquity [2,19]. An empty set of numbers and an empty set of apples are intuitively *not* the same, as well as in most cases we do not need and do not want to treat a function in the same way as its arguments.

The difference between domains (types) and predicates is made explicit in type theory. This results in various simplifications. For instance, the difference between operations on universes (product, disjoint sum) and operations on predicates (intersection, set union) becomes immediately apparent and natural. A class-based example is that a union $\bigcup A$ of a family A of sets is typically of the same "type" as members of A rather than as A itself. In set theory, this argument is not sufficient to disprove e.g., $A \subseteq \bigcup A$, because classifying sets (a priori) into types is illegal.

Everyday maths vs foundations of mathematics
The purpose of set theory was to give a universal foundation for a consistent mathematics. That happened at the beginning of 20th century, when consistency of elementary notions was a serious issue, threatened by the danger of antinomies, and when modern formal mathematics was in its infancy. It was then important that the new foundation guarantees as much security as possible. Therefore, all the development had to be done from first principles. Using the Axiom of Foundation, one can actually prove in set theory that all the world is built from curly braces.

This foundational tool is now being widely used for a quite different purpose. We use set theory as a basic everyday framework for all kinds of mathematics, and we teach it to students, beginning at a very elementary level. But that puts us into an awkward situation. On the one hand, we want to use as much commonsense as possible, on the other hand we do not want paradoxes and inconsistency. So if we do not want to cheat, what we can do? One possibility is to hide the problem and pretend that everything is OK: *"Emmm... We assume that all sets under consideration are subsets of a certain large set."* This is what often happens in elementary and high-school textbooks. But is it really different than saying that the world is placed on the back of a giant turtle? An intelligent student must eventually ask on whose back the turtle is standing. And then all we can say is "Sit up straight!"

The other option is to take the skeleton out of the closet, put all axioms on the table, and pay a heavy overhead by spending a lot of effort on constructing ordered pairs in the style of Kuratowski, integers in the style of von Neumann, and so on. This approach is common at the university level and has been considerably

mastered. For half a century, the book [17] by Halmos has been giving guidance to lecturers how to achieve a balance between precision and simplicity. (Contrary to its title, the book is not about "naive" set theory. It is about axiomatic set theory taught in a "naive" style.) But even this didactic masterpiece is a certain compromise.

The idea vs the implementation

This is because the overhead is unavoidable. One reason is that very basic mathematical ideas must be encoded in set theory before they can be used, and a substantial part of student's attention is paid to the details of the encoding. To a large extent this is a wasted effort and it would be certainly more efficient to concentrate on "top-level" issues. Using an old comparison in a different context, it is quite like teaching the details of fuel injection in a driving school while we should rather let students practice driving.

To work with the set theoretical "implementation" of mathematics we must get accustomed to it, and this is painful to many students. But the problem does not end here. It begins here. The implementation is not "encapsulated" at all and we can smell the fuel in the passenger's cabin. One of the most fundamental God's creations is turned into a transitive set of von Neumann's numbers so we must live with phenomena like $1 \in 2 \subseteq 3 \in 4$ or $\mathbb{N} = \bigcup \mathbb{N}$.

We do not really need these phenomena. The actual use of various objects and notions in mathematics is based on their intensional "specification" rather than formal implementation. We still have to ask students to remember the rule

$$\langle\, a, b \,\rangle = \langle\, c, d \,\rangle \qquad \text{iff} \qquad a = c \wedge b = d,$$

rather than just the definition $\langle\, a, b \,\rangle = \{\{a\}, \{a, b\}\}$. But we must spend time on proving the above equivalence. A doubtful reward is the malicious homework "Prove that $\bigcup(\mathbb{N} \times \mathbb{N}) = \mathbb{N}$." We got used to such homeworks so much that we do not notice that they are nonsense. Perhaps it is time for re-thinking the curriculum.

2 Why type theory and what type theory?

We believe that an appropriate type theory should give a chance to build a framework for "naive" mathematics that would not exhibit many of the drawbacks mentioned above. In particular, it is reasonable to expect that a "naive type theory" can be more adequate than "naive set theory" from our point of view, in that it should

- be free from both paradoxes and from unnecessary and artificial formalization;
- distinguish between domains (universes) and sets understood as materialized predicates;
- begin with intensional specifications rather than from bare first principles;
- be closer to the everyday maths and computer science practice;
- be more appropriate for automatic verification.

We do not want to depart from ordinary mathematical practice, and thus our naive type theory should be adequate for classical reasoning, and extensional with respect

to functions and predicates. We find it however methodologically appropriate that these choices are not built in the design principles, but rather introduced explicitly by appropriate axioms. We also would like to include a Curry-Howard flavor, taking seriously De Bruijn's slogan [6]:

> *Treating propositions as types is definitely not in the way of thinking of the ordinary mathematician, yet it is very close to what he actually does.*

The basic idea is of course to divide the two roles played by sets, namely to put apart domains (i.e. types) and predicates, i.e. formulas depending on objects of a given type. Thus for any type A we need a powerset type $P(A)$. We see no reason to make a distinction between $P(A)$ and $A \to *$, where $*$ is the sort of propositions. That is, we would like to treat "$M \in \{a : A \mid \varphi(a)\}$" as syntactic sugar for "$\varphi(M)$".

Although our principal aim is a "naive" approach, we should be aware of the need for a formalization. Firstly, because we still need some justification for consistency, secondly, because it may be desirable that "naive" reasoning can be computer-assisted. We find it quite natural and straightforward to attempt such a formalization in the form of a PTS, to be extended with additional constructs and axioms.

Related systems

Simple type theory: In Church's simple type theory [8,19] there are two base types: the type **i** of individuals and the type **b** of truth values. Expressions have types and formulas are simply expressions of type **b**. There is no built-in notion of proof and formulas are not types. In addition to lambda-abstraction, there is another binding operator that can be used to build expressions, namely the *definite description* $\iota x.\, \varphi(x)$, meaning "the only object x that satisfies $\varphi(x)$". While various forms of definite description are often used in the informal language of mathematics, the construct does not occur in most contemporary logical systems. As argued by William Farmer in a series of papers [11,12,13,14], simple type theory could be efficiently used in mathematical practice and teaching. Also the textbook [2] by P.B. Andrews develops a version of simple type theory as a basis for everyday mathematics. This is very much in line with our way of thinking. We choose a slightly different approach though, mostly to avoid the inherently two-valued Boolean logic built in Church's type theory.

Weyl's predicative mathematics: In a recent paper [1] Robin Adams and Zhaohui Luo propose a logic-enriched predicative type theory based on Weyl's *Das Kontinuum* [21]. This framework is developed as a basis for classical mathematics with predicativity in mind, as a basic criterion. In particular, there is an explicit distinction made in this approach between "categories" and sets understood respectively as universes and predicates. However, eliminating "vicious circles" is a primary goal here, even entirely innocuous ones, and this certainly departs from the "naive" way.

Constable's computational naive type theory: We have to admit that R. Constable [9] was the first to rephrase the title of Halmos' book this way. But Constable's idea of a "naive type theory" is quite different than ours. It is inspired by Martin-Löf's way of thinking and based on the idea that a type is determined by a domain of objects plus an appropriate notion of equality. For instance, the field $\mathbb{Z}_3$ has the same domain as the set of integers $\mathbb{Z}$, only a different equality. And $\mathbb{Z}_6$ is defined

by taking an "intersection" of $\mathbb{Z}_2$ and $\mathbb{Z}_3$. All this makes a consistent exposition, and makes e.g. quotient formation easy. However (even putting aside the little counterintuitivity of the "contravariant" intersection) we still believe that a "naive" notion of equality should be more strict: two objects should not be considered the same in one context but different in another.

Coq and the calculus of inductive constructions: An almost adequate framework for a naive type theory is the Calculus of Constructions extended with inductive types. This is essentially the basic part of the type theory of the Coq proof assistant [5]. The paper [7] describes an attempt to use Coq in teaching rudiments of set theory. But in Coq, if A is a type ($A : \mathrm{Set}$ is provable) then the powerset $A \to \mathrm{Prop}$ of A is a kind ($A \to \mathrm{Prop} : \mathrm{Type}$ is provable). That is, a set and its powerset do not live in the same sort.

In this paper

We collect a few postulates concerning the possible exposition of a naive type theory, in the hope of starting a discussion in the community. We hope that this discussion will help establishing a new approach to both teaching and using mathematics in a way that will avoid the set-theoretic „overheads" and remain sufficiently precise and paradox-free. These postulates are discussed in Section 3.

We realize however that a naive approach to type theory can result in an inconsistency, as it happened to naive set theory and many other ideas. Therefore we consider it necessary to build the naive approach on top of a rigouristic formal system, to be developed in parallel. The relation between the formal language and the naive theory should be similar to the relation between the first-order ZFC formal theory, and Halmos' book [17].

In what follows, we address a very initial but important problem. A set X and its powerset $P(X)$ should be objects of the same sort, and we also assume that subsets of X should be identified with predicates on X. In the language of pure type systems that leads to the idea of a type assignment of the form $X \to * \;:\; *$, which turns out to imply inconsistency. In Section 4 we show that this inconsistency can be eliminated if the difference between propositions and types is made explicit.

3 Informal exposition

In this section we sketch some basic ideas of how a "naive" informal presentation of basic mathematics could look when set theory is replaced by type theory.

Types

Every object is assigned a *type*. Types themselves are not objects.[3] Certain types are postulated by axioms, and many of these should be special cases of a general scheme for introducing inductive (perhaps also co-inductive) types. In particular, the following should be assumed:

- A unit type with a single element *nil*.

[3] At least not yet. We may have to relax this restriction, if we want to deal with e.g. objects of type "semigroup". This may lead to an infinite hierarchy of universes.

- Product types $A \times B$ and co-product types $A + B$, for any types A, B.
- The type $\mathbb{N}$ of integers.
- The powerset type $P(A)$, for any type A.
- Function types $A \to B$, perhaps as a special case of a more general product type.

Types come together with their constructors, eliminators etc., the properties of which are postulated by axioms. For instance, the following should be an axiom:

$$\langle\, a, b \,\rangle = \langle\, c, d \,\rangle \qquad \text{iff} \qquad a = c \wedge b = d,$$

Equality
In the informal exposition of set theory equality is equality: two objects are equal iff they are the same object. One can do the same in the typed framework. There is however no reason to refrain from using Leibniz's equality for explanation. In the formal model one should probably assume Leibniz's equality as an axiom.

Sets
A predicate $\varphi(x)$, where $x : A$, is identified with a *subset* $\{x : A \mid \varphi(x)\}$ of type A. Subsets are assumed to be extensional, i.e.,

$$\varphi = \varphi' \qquad \text{iff} \qquad \forall x{:}A.\, \varphi(x) \leftrightarrow \varphi'(x).$$

Inclusion is defined as usual by $\varphi \subseteq \varphi'$ iff $\forall x{:}A.\, \varphi(x) \to \varphi'(x)$. Set union and intersection as well as the complement $-\varphi = \{x : A \mid \neg\varphi(x)\}$ are well-defined operations on sets.

An indexed family of sets is given by any 2-argument predicate, so that e.g. we can write the ordinary definition $\bigcap_{y:Y} A_y = \{x : X \mid \forall y{:}Y.\, A_y(x)\}$. Should we need an intersection indexed by elements of a *set* rather than a type we must explicitly include it in the definition by writing

$$\bigcap_{y \in \psi} A_y = \{x : X \mid \forall y{:}Y(\psi(y) \to A_y(x))\}.$$

At this stage, one can prove standard results about the properties of the algebra of sets. Subsets of a Cartesian product $A \times A$ are of course called relations, and we can discuss properties of relations and introduce constructions like transitive closure and so on.

Equivalences and quotients
While a definition of an equivalence relation over a type A presents no difficulty, the notion of a quotient type must be postulated separately. Clearly, for every $a : A$ we could consider a set $[a]_r = \{b : A \mid b\, r\, a\}$, and form a subset of $P(A)$ consisting of all such sets, but that would be inconsistent with our main idea: a domain of interpretation is always a *type* and not a set. The one-to-one correspondence between the quotient type and the set of equivalence classes should then be proven as a theorem ("the principle of equivalence"). Note that while this correspondence should of course be highlighted, there is no actual reason to *identify* members of the

quotient type, which are abstract objects (*classes of abstraction*), with the *equivalence sets*, as it is done in set theory. We must see a difference between abstraction and implementation. When we define a new notion by means of abstraction, we do not think in terms of sets. For instance when rationals are defined from integers, we do not really think of $1/2$ as of a set.

Functions: total or partial?
The notion of a function brings the first serious difficulty. In typed systems, once we assert $f : A \to B$ and $a : A$ we usually conclude $f(a) : B$. That means we treat $f(a)$ as a legitimate, well-defined object of type B. Everything works well as long as we can assume that all functions from A to B are total. However, it can happen that a function is defined only on a certain subset A' of a given domain. In set theory this is not a problem, because both the type of arguments and the actual domain are simply sets, and we can always take $f : A' \to B$ rather than $f : A \to B$. In the typed framework, we would like to still say that e.g. $\lambda x{:}\mathbb{R}.\, 1/x$ maps reals to reals, but the *domain* of the function is a proper subset of the type $\mathbb{R}$. There are several possible solutions of this problem, see [11] for an in-depth discussion.

Perhaps the most adequate solution for our needs is to pretend that all functions are total, but in some cases the values are simply unknown. One should make sure that all reasoning involving function values only applies to "known" values. For a more explicit treatment of partial functions, one can assign a domain predicate $\mathrm{dom}(f)$ to every function f.

This essentially implies a restriction on the use of the expression $f(a)$ to the cases when $a \in \mathrm{dom}(f)$, which seems to be quite consistent with the ordinary mathematical practice. In this respect it may turn out to be useful to return to the old idea of explicit description. A standard function definition should then have the form $f(x) = \iota y.\varphi(x, y)$, or equivalently $f = \lambda x\, \iota y.\varphi(x, y)$, and we would postulate an axiom of the form

$$x \in \mathrm{dom}(\lambda x\, \iota y.\varphi(x, y)) \qquad \text{iff} \qquad \exists! y\, \varphi(x, y).$$

Extensionality for functions would then be stated as

$$f = g \quad \leftrightarrow \quad (\mathrm{dom}(f) = \mathrm{dom}(g)) \wedge \forall x\, (x \in \mathrm{dom}(f) \to f(x) = g(x)).$$

The drawback of this solution is that $f(a)$ remains a well-formed expression of the appropriate type, no matter whether $a \in \mathrm{dom}(f)$ or not. Thus, we still have $f(a) = f(a)$ and this implies $\forall x \exists y\, (f(x) = y)$. Although very little can be actually concluded about the value $f(a)$, this seems confusing enough to suggest that we may have to adopt the idea of partial equality. Then we would be able to claim:

$$\exists z\, (f(x) = z) \to x \in \mathrm{dom}(f).$$

Note that this problem does not formally [4] occur in set theory, where $f(x) = y$ is syntactic sugar for $\langle x, y \rangle \in f$. In type theory, it is more natural to refrain

[4] But it does occur in practice: note e.g. that $f(x) \neq y$ can be understood differently than $\langle x, y \rangle \notin f$.

from entering this level of extensionality, and to assume function application as a primitive.

Subtypes
The difference between types and subsets becomes inconvenient in certain situations. One specific example is when we have an algebra with a domain represented as a type, and we need to consider a subalgebra based on a subset of that domain. Then we would prefer to have the "large" and the "small" domain living in the same sort. To overcome this difficulty, one may have to postulate a selection scheme: for every subset S of type A there exists a type $A|S$, such that objects of type $A|S$ are in a bijective correspondence with elements of S. This partially brings back the identification of domains and predicates, but it is happening in a controlled way.

4 Naive type theory as a pure type system

The assumption that a set and a powerset should live in the same sort leads naturally to the following idea: consider a pure type system with the usual axiom $* : \Box$ and with the rule $(*, \Box, *)$. This rule makes possible to build products of the form $\Pi x{:}A.\,\kappa$, where $A : *$ and $\kappa : \Box$, and the product itself is then a type (is assigned the sort $*$). In particular, the function space $A \to *$ is a type, and this is exactly the powerset of A. A subset of A is then represented by any abstraction $\lambda x{:}A.\,\varphi(x)$, where $\varphi(x)$ is a (dependent) proposition.

Unfortunately, this idea is too naive. As pointed out by A.J.C. Hurkens and H. Geuvers, this theory suffers from Girard's paradox, and thus it is inconsistent.

Theorem 4.1 (Geuvers, Hurkens [16]) *Let* VNTT *(Very Naive Type Theory) be an extension of* λP *by the additional rule* $(*, \Box, *)$. *Then every type is inhabited in* VNTT *(every proposition has a proof).*

Proof. The proof is essentially the same as Hurkens' proof in [18], (cf. the version given in [20, chapter 14]) for the system λU^-. There are two essential factors that imply that Russel's paradox can be implemented in a theory:

- A powerset construction $P(x)$ on any object x of a sort s lives in the same sort s.
- There is enough polymorphism available in s to implement a construction of an inductive object $\mu x{:}s.P(s)$.

In λU^- we have $s = \Box$ and polymorphism on the kind level is directly available. But almost the same can happen in VNTT, for $s = *$. Indeed, the powerset $A \to *$ of any type A is a type, and although type polymorphism as such is not present, it sneaks in easily by the back door. Instead of quantifying over types, one can quantify over object variables of type $T \to *$, where T is any type. Thus instead of using $\mu t : *.P(t) = \forall t(\forall u{:}*\,((u \to t) \to P(u) \to t) \to t)$ one takes $a : T$ and then defines

$$\mu t : *.P(t) = \forall x : T \to *\,(\forall y : T \to *\,((ya \to xa) \to P(ya) \to xa) \to xa),$$

with essentially the same effect. □

It follows that our naive type theory cannot be too naive, and must avoid the danger of Girard's paradox. The solution is to distinguish between propositions and sets, like in Coq.

Define a pure type system LNTT (Less Naive Type Theory) with four sorts

$$*^t, \quad *^p, \quad \Box^t, \quad \Box^p,$$

with axioms $(*^t : \Box^t)$ and $(*^p : \Box^p)$ and with the following rules:

$$(*^t, *^t, *^t), \quad (*^p, *^p, *^p), \quad (*^t, \Box^t, \Box^t), \quad (*^t, *^p, *^p), \quad (*^t, \Box^p, *^t).$$

The first and second rule represent, respectively, the formation of function types, and logical implication; the third rule is for dependent types and the fourth one introduces higher-order logic. The last rule is for the powerset type.

In the next section we prove the strong normalization property for LNTT. Of course, that is only the first step. We need a much richer consistent system to back up our "practical" exposition of sets, functions, and composite types, as sketched in the previous section. This will most likely require extending LNTT by various additional constructs, in particular a general scheme for inductive types, and additional axioms. All this is future work.

Strong normalization

First note that, as all PTSs with only β-reduction, the system LNTT has the Church-Rosser property on well-typed terms [15]. Moreover, LNTT is a singly sorted PTS [3], so the uniqueness of types and subject reduction property hold.

Definition 4.2 In a fixed context Γ we use the following terminology.

(i) A is a *term* if and only if there exists B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.

(ii) A is a *kind* if and only if $\Gamma \vdash A : \Box^t$.

(iii) A is a *constructor* if and only if there exists B such that $\Gamma \vdash A : B : \Box^t$.

(iv) A is a *type* if and only if $\Gamma \vdash A : *^t$.

(v) A is a *formula* if and only if $\Gamma \vdash A : *^p$.

(vi) A is an *object* if and only if there exists B such that $\Gamma \vdash A : B : *^t$.

(vii) A is a *proof* if and only if there exists B such that $\Gamma \vdash A : B : *^p$.

We use the notation $Term_\Gamma$, $Kind_\Gamma$, $Constr_\Gamma$, $Type_\Gamma$, $Prop_\Gamma$, Obj_Γ, and $Proof_\Gamma$, to denote respectively terms, kinds, constructors, types, formulas, objects, and proofs of the context Γ. We may describe the structure of various categories of terms.

Lemma 4.3 *Assume a fixed context* Γ.

- *If* A *is a term such that* $\Gamma \vdash A : \Box^p$ *then* $A = *^p$.
- *If* A *is a kind then* A *is of the following form*
  - $A = *^t$ *or*
  - $A = \Pi x : \tau.B$ *where* τ *is a type and* B *is a kind.*

- *If A is a constructor then*
  - *A is a type, or*
  - *A is a variable, or*
  - *A is of the form $\lambda x:\tau.\kappa$ where τ is a type and κ is a constructor, or*
  - *A is of the form κM where M is an object and κ is a constructor.*
- *If A is a type then*
  - *A is a type variable, or*
  - *A is of the form $\Pi x:\tau.\sigma$ where τ and σ are types, or*
  - *A is of the form $\Pi x:\tau.*^p$ where τ is a type, or*
  - *A is of the form κM where M is an object and κ is a constructor.*
- *If A is a formula then*
  - *A is a propositional variable, or*
  - *A is of the form $\Pi x:\varphi.\psi$ where φ and ψ are formulas, or*
  - *A is of the form $\Pi x:\tau.\varphi$ where τ is a type and φ is a formula, or*
  - *A is of the form MN where M and N are objects.*
- *If A is an object then*
  - *A is an object variable, or*
  - *A is of the form $\lambda x:\tau.N$ where τ is a type and N is an object, or*
  - *A is of the form $\lambda x:\tau.\varphi$ where τ is a type and φ is a formula, or*
  - *A is of the form MN where M and N are objects.*
- *If A is a proof then*
  - *A is a proof variable, or*
  - *A is of the form $\lambda x:\tau.D$ where τ is a type and D is a proof, or*
  - *A is of the form $\lambda x:\varphi.D$ where φ is a formula and D is a proof, or*
  - *A is of the form D_1D_2 where D_1 and D_2 are proofs, or*
  - *A is of the form DN where D is a proof and N is an object.*

Lemma 4.4 *If A is a term which is not a proof and B is a subterm of A then B is not a proof.*

Proof. This is an immediate consequence of Lemma 4.3. □

Note that it follows from Lemma 4.4 that all formulas of the form $\Pi x:\varphi.\,\psi$, where φ and ψ are formulas, are actually implications (can be written as $\varphi\to\psi$) because the proof variable x cannot occur in ψ.

The first part of our strong normalization proof applies to all terms but proofs. For a fixed context Γ we define the translation $T_\Gamma : Term_\Gamma - Proof_\Gamma \to Term(\lambda P2)$ from terms of LNTT into the system $\lambda P2$. Special variables $Bool$, $Forall$ and $Impl$ will be used in the definition of T. Types for these variables are given by the following context:

$$\Gamma_0 = \{Bool:*, Forall:\Pi\tau:*.(\tau\to Bool)\to Bool, Impl: Bool\to Bool\to Bool\}$$

Definition of the translation T_Γ follows:

- $T_\Gamma(\Box^t)=\Box$;
- $T_\Gamma(\Box^p)=*$;

- $T_\Gamma(*^t) = *$;
- $T_\Gamma(*^p) = Bool$;
- $T_\Gamma(x) = x$, when x is a variable;
- $T_\Gamma(\Pi x : A.B) = \Pi x : T_\Gamma(A).T_{\Gamma,x:A}(B)$ for products created with the rules $(*^t, *^t, *^t)$, $(*^t, \Box^p, *^t)$, $(*^t, \Box^t, \Box^t)$;
- $T_\Gamma(\Pi x : \tau.\varphi) = Forall\ T_\Gamma(\tau)(\lambda x : T_\Gamma(\tau).T_{\Gamma,x:\tau}(\varphi))$ for products created with the rule $(*^t, *^p, *^p)$;
- $T_\Gamma(\Pi x : \varphi.\psi) = Impl\ T_\Gamma(\varphi)T_\Gamma(\psi)$, for products created with the rule $(*^p, *^p, *^p)$;
- $T_\Gamma(\lambda x : A.B) = \lambda x : T_\Gamma(A).T_{\Gamma,x:A}(B)$;
- $T_\Gamma(AB) = T_\Gamma(A)T_\Gamma(B)$.

We extend the translation T to contexts as follows:

- $T(\langle\rangle) = \Gamma_0$,
- $T(\Gamma, x : A) = T(\Gamma), x : T_\Gamma(A)$ if A is a kind, a type, or $*^p$,
- $T(\Gamma, x : A) = T(\Gamma)$ if A is a formula.

For the sake of simplicity we omit the subscript Γ if it is clear which context we are using. [5]

We now state some technical lemmas which are used in the proof of soundness of the translation T.

Definition 4.5 We say that contexts Γ and Γ' are *equivalent* with respect to the set of variables $X = \{x_1, \ldots, x_n\}$ if and only if Γ and Γ' are legal contexts and for all $x \in X$ we have $\Gamma(x) =_\beta \Gamma'(x)$.

Lemma 4.6 *If Γ and Γ' are equivalent with respect to X, and $N \in Term_\Gamma$ is such that $FV(N) \subseteq X$ and $\Gamma \vdash N : A$, then $\Gamma' \vdash N : A'$ where $A =_\beta A'$. In particular, if $N \in Term_\Gamma$ then $N \in Term_{\Gamma'}$.*

Proof. Induction with respect to the structure of the derivation $\Gamma \vdash N : A$. □

Lemma 4.7 *If Γ and Γ' are equivalent with respect to $FV(M)$ and $M \in Term_\Gamma$ then $T_\Gamma(M) = T_{\Gamma'}(M)$.*

Proof. Induction with respect to the structure of M. □

Now we prove that the translation T preserves beta-reduction.

Lemma 4.8 *If $\Gamma \vdash a : A$ and $\Gamma, x : A \vdash B : C$ for some C and a, B are not proofs then $T_\Gamma(B[x := a]) = T_{\Gamma,x:A}(B)[x := T_\Gamma(a)]$.*

Proof. Induction with respect to the structure of B, using Lemma 4.7. □

Lemma 4.9 *If B and B' are not proofs and $B \to_\beta B'$ then $T_\Gamma(B) \twoheadrightarrow_\beta^+ T_\Gamma(B')$.*

Proof. The proof is by a routine induction with respect to $B \to_\beta B'$. If B is a redex then, by Lemma 4.4, it must be of one of the following forms:

$$(\lambda x : \tau.\varphi)N, \qquad (\lambda x : \tau.M)N, \qquad (\lambda x : \tau.\kappa)N,$$

[5] I.e., when the context is clear from the context ;)

where τ is a type, φ is a formula, and M, N are objects. In each of these cases we apply Lemma 4.8. If B is not a redex, we apply the induction hypothesis, using Lemma 4.7. □

Lemma 4.10 *If $B =_\beta B'$ and B, B' are kinds, types or objects then $T_\Gamma(B) =_\beta T_\Gamma(B')$.*

Proof. By Church-Rosser property there exists a well-typed term C such that $B \twoheadrightarrow_\beta C$ and $B' \twoheadrightarrow_\beta C$. We have $T_\Gamma(B) \twoheadrightarrow_\beta T_\Gamma(C)$ and $T_\Gamma(B') \twoheadrightarrow_\beta T_\Gamma(C)$, by Lemma 4.9, whence $T_\Gamma(B) =_\beta T_\Gamma(B')$. □

Lemma 4.11 (Soundness of the translation T) *If $\Gamma \vdash A : B$ and A is not a proof then $T(\Gamma) \vdash T_\Gamma(A) : T_\Gamma(B)$ in $\lambda P2$.*

Proof. Induction with respect to the structure of the derivation of $\Gamma \vdash A : B$ using Lemmas 4.7, 4.8 and 4.10. □

Corollary 4.12 *If M is a term which is not a proof then M is strongly normalizing.*

Proof. Assume that M is not strongly normalizing. Then there is an infinite reduction

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots$$

By Lemma 4.9 then

$$T(M) \twoheadrightarrow_\beta^+ T(M_1) \twoheadrightarrow_\beta^+ T(M_2) \twoheadrightarrow_\beta^+ \cdots$$

But $T(M)$ is a valid term of $\lambda P2$, by Lemma 4.11, thus it is strongly normalizing. The contradiction shows that also M is strongly normalizing. □

To show strong normalization for proofs we use another translation t from LNTT to the calculus of constructions λC. This translation depends on a given context Γ.

- $t_\Gamma(*^t) = *$;
- $t_\Gamma(*^p) = *$;
- $t_\Gamma(x) = x$, if x is a variable.
- $t_\Gamma(\Pi x : \tau.B) = t_{\Gamma,x:\tau}(B)$, for products constructed using the rule $(*^t, \Box^t, \Box^t)$;
- $t_\Gamma(\Pi x : A.B) = \Pi x : t_\Gamma(A).t_{\Gamma,x:A}(B)$ for all other products;
- $t_\Gamma(\lambda x : \tau.\kappa) = t_{\Gamma,x:\tau}(\kappa)$, if κ is a constructor and τ is a type;
- $t_\Gamma(\lambda x : A.B) = \lambda x : t_\Gamma(A).t_{\Gamma,x:A}(B)$ for all other abstractions;
- $t_\Gamma(\kappa N) = t_\Gamma(\kappa)$ if κ is a constructor;
- $t_\Gamma(AB) = t_\Gamma(A)t_\Gamma(B)$ for all other applications.

We extend the translation t to contexts by taking

$$t(\langle\rangle) = \langle\rangle \quad \text{and} \quad t(\Gamma, x : A) = t(\Gamma), x : t_\Gamma(A).$$

Lemma 4.13 *If Γ and Γ' are equivalent with respect to $FV(M)$ and M is a term in Γ then $t_\Gamma(M) = t_{\Gamma'}(M)$.*

Proof. Induction with respect to the structure of M. □

Lemma 4.14 *Assume that $\Gamma, x : A \vdash B : C$ and $\Gamma \vdash N : A$ and N is an object or a proof.*

- *If N is an object and B is a type or a constructor then* $t_\Gamma(B[x := N]) = t_{\Gamma,x:A}(B)$.
- *If B is neither a type nor a constructor then*
$$t_\Gamma(B[x := N]) = t_{\Gamma,x:A}(B)[x := t_\Gamma(N)].$$

Proof. Induction with respect to the structure of B, using Lemma 4.7. □

Definition 4.15 A reduction step $A \to_\beta A'$ is *silent* if

- $A = (\lambda x : \tau.\kappa)N \to_\beta \kappa[x := N] = A'$, where κ is a constructor and N is an object, or
- $A = \Pi x : \tau.B \to_\beta \Pi x : \tau'.B = A'$ where $\tau \to_\beta \tau'$ and B is a kind, or
- $A = \kappa N \to_\beta \kappa N' = A'$, where $N \to_\beta N'$ and κ is a constructor, or
- $A = \lambda x : \tau.\kappa \to_\beta \lambda x : \tau'.\kappa = A'$, where κ is a constructor, or
- $A = C[B] \to_\beta C[B'] = A'$, where $C[\ ]$ is any context and $B \to_\beta B'$ is a silent reduction.

Lemma 4.16 *If $A \to_\beta B$ then $T_\Gamma(A) \twoheadrightarrow_\beta T_\Gamma(B)$. In addition, if the reduction $A \to_\beta B$ is not silent then $T_\Gamma(A) \twoheadrightarrow_\beta^+ T_\Gamma(B)$.*

Proof. Induction with respect to $A \to_\beta B$, using Lemma 4.14 when A is a redex, and Lemma 4.13 in the other cases. □

Corollary 4.17 *If $B =_\beta B'$ then $t_\Gamma(B) =_\beta t_\Gamma(B')$.*

Lemma 4.18 *Assume a fixed environment Γ.*

(i) *If M is a proof, an object, or a formula and $\Gamma \vdash M : T$ then $t(\Gamma) \vdash t_\Gamma(M) : t_\Gamma(T)$.*

(ii) *If M is a type or a constructor then $t(\Gamma) \vdash t_\Gamma(M) : *$ or $t(\Gamma) \vdash t_\Gamma(M) : \Box$.*

(iii) *If M is a kind then $t(\Gamma) \vdash t_\Gamma(M) : \Box$.*

Proof. Simultaneous induction with respect to the structure of the appropriate derivation, using Lemma 4.13. □

Theorem 4.19 *System* LNTT *has the strong normalization property.*

Proof. We already know that all expressions except proofs are strongly normalizing. Arguing as in the proof of Corollary 4.12, and using Lemma 4.16, we conclude that almost all steps in an infinite reduction sequence must be silent. Thus it suffices to prove that if D is a proof than there is no infinite silent reduction of D. This goes by induction with respect to the size of D, by cases depending on its shape. □

No conclusion

The above is by no means a complete proposal of either theoretical or didactic character. It is essentially a collection of questions and partial suggestions of how such a proposal should be eventually designed. These questions are of double nature, and we would like to pursue the two directions. The first one is to find means to talk about basic mathematics without referring to set theory in either a naive (i.e., inconsistent) or axiomatic way, using instead an appropriate type-based language.

That should happen in a possibly non-invasive way, keeping as much linguistic compatibility with the "standard" style as possible.

The second problem is to give a formal foundation to this informal type-based language. This formalization is to be used for two purposes: to guarantee logical consistency of the naive exposition and to facilitate computer assisted verification and teaching. That requires building a complex system, of which our PTS-style Less Naive Type Theory is just a very basic core. This system must involve various extensions in the style of [4], perhaps include a hierarchy of sorts, etc.

Acknowledgement

Thanks to Herman Geuvers for helpful discussions, e.g. about Girard's paradox.

References

[1] R. Adams and Z. Luo. Weyl's predicative classical mathematics as a logic-enriched type theory. Manuscript, 2007.

[2] P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, volume 27 of *Applied Logic Series*. Kluwer, second edition, 2002.

[3] H.P. Barendregt. Lambda calculi with types. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[4] G. Barthe. Extensions of pure type systems. In Dezani-Ciancaglini and Plotkin [10], pages 16–31.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[6] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[7] J. Chrząszcz and J. Sakowicz. Papuq: A Coq assistant. Manuscript, 2007.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[9] R.L Constable. Naive computational type theory. In H. Schwichtenberg and R. Steinbruggen, editors, *Proof and System-Reliability*, pages 213–259. Kluwer Academic Press, 2002.

[10] M. Dezani-Ciancaglini and G. Plotkin, editors. *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[11] W.M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–1291, 1990.

[12] W.M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[13] W.M. Farmer. A basic extended simple type theory. Technical Report 14, McMaster University, 2003.

[14] W.M. Farmer. The seven virtues of simple type theory. Technical Report 18, McMaster University, 2003.

[15] H. Geuvers. The Church-Rosser property for beta-eta-reduction in typed lambda calculi. In *Logic in Computer Science*, pages 453–460, 1992.

[16] H. Geuvers. Private communication, 2006.

[17] P.R. Halmos. *Naive Set Theory*. Van Nostrand, 1960. Reprinted by Springer-Verlag in 1998.

[18] A.J.C. Hurkens. A simplification of Girard's paradox. In Dezani-Ciancaglini and Plotkin [10], pages 266–278.

[19] F. Kamareddine, T. Laan, and R. Nederpelt. Types in logic and mathematics before 1940. *Bulletin of Symbolic Logic*, 8(2):185–245, 2002.

[20] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[21] H. Weyl. *The Continuum*. Dover, 1994.

Teaching logic using a state-of-the-art proof assistant

Cezary Kaliszyk Freek Wiedijk

Radboud Universiteit Nijmegen, The Netherlands

Maxim Hendriks Femke van Raamsdonk

Vrije Universiteit Amsterdam, The Netherlands

Abstract

This article describes the system ProofWeb that is currently being developed in Nijmegen and Amsterdam for teaching logic to undergraduate computer science students. This system is based on the higher order proof assistant Coq, and is made available to the students through an interactive web interface. Part of this system will be a large database of logic problems. This database will also hold the solutions of the students. This means that the students do not need to install anything to be able to use the system (not even a browser plug-in), and that the teachers will be able to centrally track progress of the students.
The system makes the full power of Coq available to the students, but simultaneously presents the logic problems in a way that is customary in undergraduate logic courses. Both styles of presenting natural deduction proofs (Gentzen-style 'tree view' and Fitch-style 'box view') are supported. Part of the system is a parser that indicates whether the students used the automation of Coq to solve their problems or that they solved it themselves using only the inference rules of the logic. For these inference rules dedicated tactics for Coq have been developed.
The system has already been used in a type theory course, and is currently being further developed in the first year logic course of computer science in Nijmegen.

Keywords: Logic Education, Proof Assistants, Coq, Web Interface, AJAX, DOM, Natural Deduction, Gentzen, Fitch

1 Introduction

1.1 Motivation

At every university, part of the undergraduate computer science curriculum is an introductory course that teaches the rules of propositional and predicate logic. At the Radboud Universiteit (RU) in Nijmegen this course is taught in the first year and is called 'Beweren en Bewijzen' (Dutch for 'Stating and Proving'). At the Vrije Universiteit (VU) in Amsterdam this course is taught in the second year and

[1] Email: {cek,freek}@cs.ru.nl {mhendri,femke}@few.vu.nl

[2] This research was funded by SURF project 'Web-deductie voor het onderwijs in formeel denken'.

is called 'Inleiding Logica' ('Introduction to Logic'). Almost all computer science curricula will have similar undergraduate courses.

For learning this kind of elementary mathematical logic it is crucial to make many exercises. Those exercises can of course be made in the traditional way, using pen and paper. The student is completely on his own, and in practice it often happens that proofs that are almost-but-not-completely-right are produced. Alternatively, they can be made using some computer program, which guides the student through the development of a completely correct proof. A disadvantage of the computerized way of practicing mathematical logic is that a student often will be able to finish proofs by random experimentation with the commands of the system (accidentally hitting a solution), without really having understood how the proof works. Of course, a combination of the two styles of practicing formal proofs seems to be the best option. So computer assistance for learning to construct derivations in mathematical logic is desirable. Currently the most popular program that is used for this kind of 'computer-assisted logic teaching' is a system called Jape [2], developed at the university of Oxford.

Besides exercises there is also the issue of examination. It would be good if the student has the opportunity to do at any moment a (part of the) logic exam by logging in to the system and be presented with a set of exercises from a database that have to be solved within a certain time. This may require human supervision to prevent cheating. We did not yet work on this, but just mention it as a possible interesting application of computer-assisted logic teaching.

1.2 Our contribution

This paper describes a system, currently named ProofWeb, that is in development at the RU in Nijmegen and at the VU in Amsterdam. This system is much like Jape (it might be considered to be an 'improved Jape-clone').

The two main innovations that our system offers over other similar systems are:

- The system makes the students work on a centralized server that has to be accessed through a web interface. The proof assistant that the students use will not run on their computer, but instead will run on the server.

  A first advantage is flexibility. The web interface is extremely light: the student will not need to install anything to be able to use it, not even a plug-in. When designing our system we tried to make it as low-threshold and non-threatening as possible. The student can work from any internet-connection at any time.

  A second advantage is that the student does not need to worry about version problems with the software or the exercises. Since everything is on the same centralized server, the students have at any time the right version of the software, exercises, and possibly solutions to exercises available, and moreover the teachers know at any time the current status of the work of the students.

- The system makes use of a state-of-the-art proof assistant, namely Coq [3], and not of a 'toy' system.

  Coq has been in development since 1984 at the INRIA institute in France. It is based on a type theoretical system called *the Calculus of Inductive Constructions*. It has been implemented in the dialect of the ML programming language called

Objective Caml, and has been used for the formal verification of many proofs, both from mathematics and from computer science. The most impressive verification using Coq is the verification of the proof of the Four Colour Theorem by Georges Gonthier [5]. Another important verification has been the development of a verified C compiler by Xavier Leroy and others [9].

The choice for a state-of-the-art proof assistant fell on Coq because both at the RU and at the VU it is already used in research and teaching.

An advantage of using a state-of-the-art proof assistant is again flexibility. The same interface can be used (possibly adapted) for teaching more advanced courses in logic or concerning the use of the proof assistant.

The system ProofWeb comes equipped with two more products.

- A large collection of logic exercises. The exercises range from very easy to very difficult, and will be graded for their difficulty. The exercise set is sufficiently large (presently over 200 exercises) so the student will not soon run out of practice material. More about the exercise set can be found in Section 6.
- Course notes, with a basic presentation of propositional and predicate logic, and a description of how to use the system ProofWeb. We want the presentation of the proofs in the system to be identical to the presentation of the proofs in the textbook. Therefore we develop both the 'Gentzen-style' and the 'Fitch-style' natural deduction variants. The course notes are still under development.

1.3 Related work

There are already numerous systems for doing logic by computer, of which Jape is the best known. A relatively comprehensive list is maintained by Hans van Ditmarsch [10]. Of course many of these system are quite similar to our system (as well as to each other.) For instance, quite a number of these systems are already web-based.

The distinctive features of our system are the use of a serious proof assistant, together with a *centralized* 'web application' architecture. The work of the students remains on the web server, can be saved and loaded back in, and the progress of the student is at all times available both to the student, the teacher and the system (i.e., the system has at all times an accurate 'user model' of the abilities of the student).

1.4 Contents

In the rest of the paper we present both our project and the current state of the system that we are building. We start with a short description of our project in Section 2, and discuss our experiences so far in Section 3. Next, in Section 4 we present the architecture of the interface. Section 5 is concerned with the supporting infrastructure of tactics and exercises, and Section 6 with the presentation of the collection of exercises. Finally, in Section 7 we give an outlook on future work and work that is currently in progress.

2 Structure of the project

The project of developing ProofWeb is financed by the SURF foundation [12] (the Dutch organization for computers in academic teaching) and runs in the period fall 2006 till fall 2007 (three semesters). Cezary Kaliszyk is employed for a full year at half time to program the system, while Maxim Hendriks is employed for half a year at full time to develop the educational materials (the database of problems and the course notes), as well as to evaluate the educational success of the project.

We identified the following nine sub-tasks, called 'work packages':

(i) the database of the system,

(ii) Coq tactics that exactly correspond to the rules the logic,

(iii) graphical representations for the proofs,

(iv) checking a Coq file against an exercise,

(v) a large set of logic problems,

(vi) course notes that explain the system,

(vii) using the system in actual courses,

(viii) dissemination of the results of the project,

(ix) evaluation of the project.

3 Experience so far

The system ProofWeb is used in the following advanced courses:

(i) In fall 2006: the course 'Logical Verification' [11] at the VU, taught by Femke van Raamsdonk. This is a computer science master's course about the type theory of the Coq system. The course is meant for more mature students but also recapitulates some undergraduate logic. It is therefore suitable for testing a first version of ProofWeb. Natural deduction is taught in Gentzen style, that is, proofs have a tree-like structure, and grow upward from the conclusion of the proof.

(ii) In spring 2007: the course 'Type Theory' at the RU, taught by Freek Wiedijk and Milad Niqui. This course is also a master's level course about the type theory of the Coq system, and corresponds to the Logical Verification course at the VU.

(iii) In spring 2007: the course 'Type Theory and Proof Assistants' in the 'Master Class Logic 2006-2007', taught by Herman Geuvers and Bas Spitters. This course is similar to the previous one, but is not exclusively aimed at students of the RU but at master's students from all over the Netherlands.

Moreover, ProofWeb is or will be used in the following introductory courses:

(i) In spring 2007: the course 'Beweren en Bewijzen' [1] at the RU, taught by Hanno Wupper and Erik Barendsen. This is a computer science undergraduate course in logic, with natural deduction in Gentzen style.

(ii) In fall 2007: the course 'Inleiding Logica' [6] at the VU, taught by Roel de

Vrijer. This is a computer science undergraduate course in logic, with natural deduction in Fitch style (cf. Section 7), that is, proofs have a structure of nested boxes, which structure a sequential list of proof steps. Another name for this kind of proofs is 'flag-style proofs', because often the assumptions of a subproof are written in the shape of 'flags'.

The course 'Logical Verification' at the VU in fall 2006 was a first opportunity to test the system. About 25 students followed and completed the course. They were all mature (graduate) students, very well able to deal with a system that was still in beta. A part of the course consists of learning type theory and Coq via basic (undergraduate) logic exercises, which were done using the system ProofWeb. We learned the following from the use of the system ProofWeb in this course.

Initially we did not have a dedicated server, so it was running on one of the group servers of the research group in Nijmegen on a non-standard port. One of the issues was, that the web-proxy at the VU did not allow the students to access pages running on non-standard ports, so they were required to turn the proxy off.

One of the assignments involves program extraction. Of course we did not allow running the extracted programs on the server, and therefore a mechanism allowing the students to obtain the extracted program was implemented.

The efficiency of the server turned out not to be a problem. At peak times the twenty-five students were able to use about 400Mb memory and a fraction of a CPU. This might be thanks to the fact that the students were not using tactics that involve automation.

During this course there was not yet support for visualizing proofs. Instead the students had to do their proofs using the customary Coq proof style, which consists of building a tactic script using the standard Coq tactics. This was not problematic, since one of the aims of the course is to learn Coq.

The second course in which ProofWeb is used is the course 'Type Theory' in spring 2007 at the RU. The first half of this course is basically an accelerated clone of the 'Logical Verification' course. As it turned out that initially there were only very few students who wanted to follow this course, it was decided that there would be no lectures, and that the students just would be given the course notes of 'Logical Verification' together with access to the server. They then would work on their own, with an opportunity to call for help if needed. It turns out that this worked unexpectedly well. The students just studied the lecture notes and did the exercises of the course. And even without much pressure on them in the form of requiring them to meet deadlines, they managed to keep on schedule reasonably well. The only thing that at some point confused them (after which a lecture was organized to make things clear) was the part of the course that did not correspond to Coq work: derivations in Pure Type Systems.

All in all our experience so far is that the system ProofWeb seems to work very well in teaching. Indeed, hardly any students used more traditional Coq interfaces like Proof General or CoqIDE. The courses so far are more advanced ones, so it remains to be seen whether ProofWeb also works well for larger numbers of undergraduate students, but we are optimistic about that. In addition, as of May 2007, the progress on all of the nine work packages seems to be well on target.

4 Architecture of the interface

In this section we shortly describe the architecture of the interface to Coq used in ProofWeb. The interface is an implementation of an architecture for creating responsive web interfaces for proof assistants [7]. It combines the current web development technologies with the functionality of local interfaces for proof assistants to create an interface that behaves like a local one, but is available completely with just a web browser (no Java, Flash or plugins are required).

To obtain this it uses the *asynchronous DOM modification* technology (sometimes referred to as *AJAX* or *Web Application*). This technique is a combination of three available web technologies:

- JavaScript — a scripting programming language interpreted by web browsers;
- *Document Object Model* (*DOM*) — a way of referring to subelements of a web page that allows modification of the page on the fly, creating dynamic elements;
- XmlHttp — an API available to client side scripts, that allows requesting information from the web server without reloading the page.

The asynchronous DOM modification consists in creating a web page that captures events on the client side and processes them without reloading the page. Events that require information from the server send the data in asynchronous XmlHttp requests and modify the web page in place. Other events are processed only locally.

ProofWeb uses an implementation of this architecture that is used to create a web interface for proof assistants. The server stores sessions for all users, and the clients are presented with an interface that is completely contained in a web browser, but resembles and is comparably responsive to a local interface like Proof General or CoqIDE (see Figure 1).

The architecture described in [7] was designed as a publicly available web service. Using it for teaching required the creation of groups of logins for particular courses. The students are allowed to access only their own files via the web interface, and teachers of particular courses have access to the directories of the students of these courses.

An example of the use of the interface in the 'Logical Verification' course can be seen in Figure 2.

5 Natural deduction for first-order logic

This section is concerned with natural deduction proofs for first-order logic in 'Gentzen style', where a proof is a tree.

5.1 Tactics

A first aim is to enable students of logic courses to construct derivations that correspond exactly to the derivations in the presentation of natural deduction that they use. Because in principle the full power of Coq is available, this means that we had to write tactics (in effect, to dumb Coq down) to match the traditional logic rules. What then arose was that, whereas in a Coq proof one can look at a hypothesis and

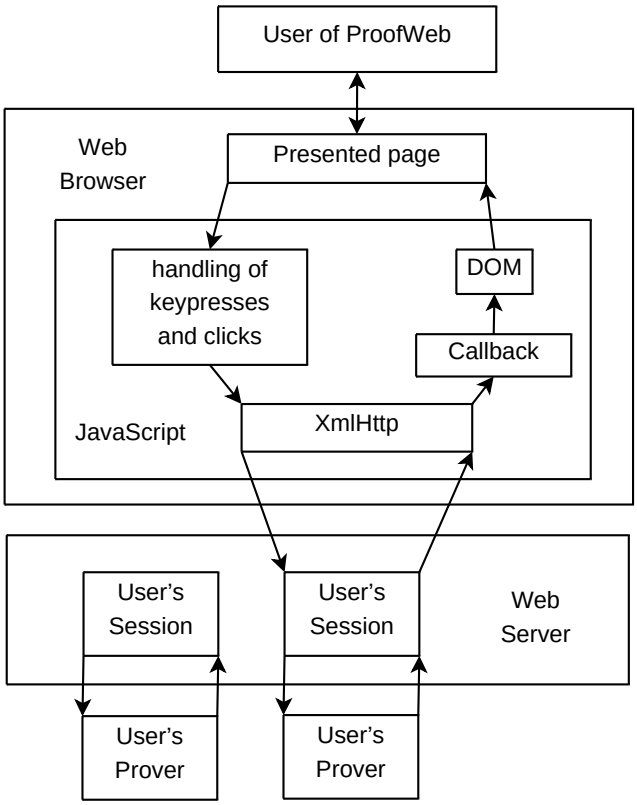


Fig. 1. ProofWeb architecture.

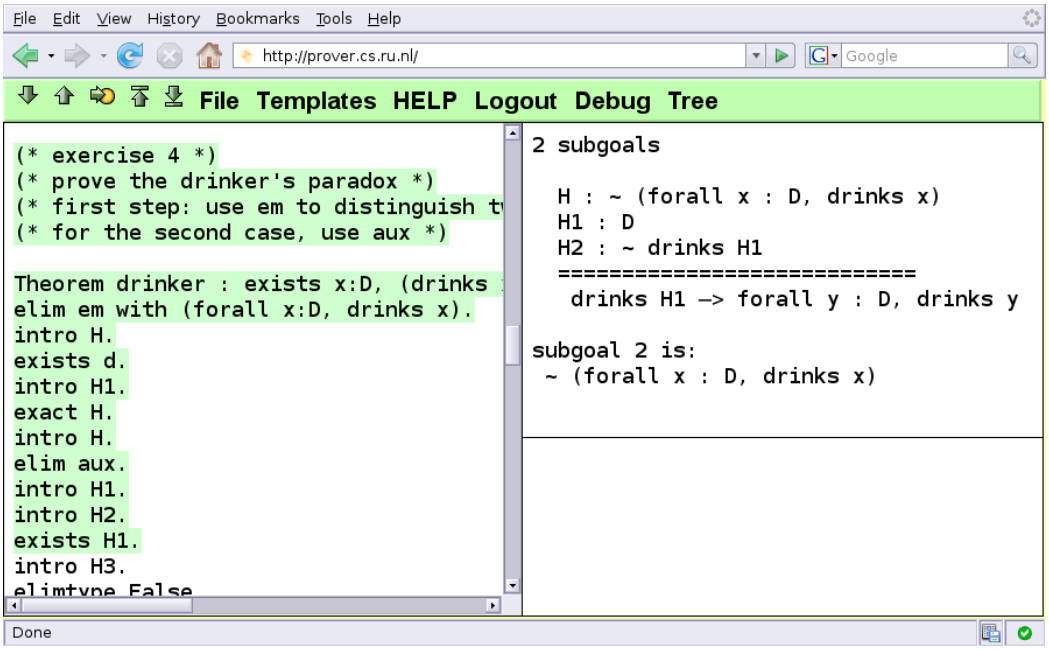


Fig. 2. ProofWeb in the 'Logical Verification' course.

eliminate it, ending up in a new proof state, traditional natural deduction offers no such jumps. So we naturally arrived at a set of backward working tactics: every proposition (the current goal) is deduced from another proposition (the new goal) using a deduction rule. The display style that fits most naturally to this kind of proof is a proof tree (for flag-style proofs see Section 7).

This imposes a relatively strict way of working. The proof trees have to be constructed from 'bottom to top'. On the one hand, this makes the construction of a deduction more difficult than on paper, because there is no possibility of building snippets of the proof in a forward way, using what is known from the hypotheses and their consequences. But on the other hand, the method forces the student to ponder the general structure of the proof before deciding by what step he will eventually end up with the current proposition. And the imposed rigidity is congenial with the aim of a logic course to encourage rigorous analytical thinking. Moreover, it becomes very clear where ingenuity comes in, such as with the disjunction elimination rule. The student is supposed to prove some proposition C. It is a creative step to find a disjunction $A \vee B$, prove this, and also prove that C follows from both A and B separately. The same goes for the introduction and elimination of negation.

As an example we present the tactic for disjunction elimination, which gives a good impression of the way additional tactics are implemented:

```
Ltac dis_el X H1 H2 :=
  match X with
  | ( _ \/ _ ) =>
   assert X;
   [ idtac |
     match goal with
       | x : X |- _ =>
         elim x; [intro H1 | intro H2]; clear x
     end
   ]
  | _ => fail "The first argument is not a disjunction"
end.
```

If the current goal is C, the tactic `dis_el (A \/ B) G H` will create the following three new goals:

(i) $A \vee B$;

(ii) C, but now with the extra assumption A with name (or proof, if viewed constructively) G;

(iii) C, but now with the extra assumption B with name (or proof, if viewed constructively) H.

Also, the tactic gives a nice and understandable error message. All the tactics have been given a name by using three letters of the connective's name and indicating whether the tactic implements an introduction rule or an elimination rule (and if necessary, if that is a left or a right variant). We give a small example of a proof with our set of tactics, and hope it speaks for itself:

```
Theorem example : ((A \/ B) /\ ~A) -> B.
Proof.
imp_in z.
dis_el (A \/ B).
con_ell (~A).
ass z.
imp_in y.
neg_el A.
con_elr (A \/ B).
ass z.
ass y.
imp_in x.
ass x.
Qed.
```

5.2 Visualization

A second aim is a visual presentation of proofs as in Jape. This meant requesting the proof information from Coq and converting it to a graph format. Coq internally keeps a proof state. This proof state is a recursive OCaml structure, that holds a goal, a rule which allows to obtain this goal from the subgoals, and the subgoals themselves. It is not just a tree structure, since a rule can be a compound rule that contains another proof state. Tactics and tacticals modify the proof state. Coq includes commands that allow inspecting the proof state. `Show` allows the user to see a non-current goal, `Show Tree` shows the succession of conclusions, hypotheses and tactics used to obtain the current goal and `Show Proof` displays the CIC term (possibly with holes).

The output of these commands was not sufficient to build a natural deduction tree for the proof. We added a new command `Dump Tree` to Coq that allows exporting the whole proof state in an XML format. An example of the output of the `Dump Tree` command for a very simple Coq proof:

```
<tree><goal><concl type="A -> A"/></goal>
  <cmpdrule><tactic cmd="intro x"/>
    <tree><goal><concl type="A -> A"/></goal>
      <cmpdrule><tactic cmd="intro x"/>
        <tree><goal><concl type="A -> A"/></goal>
          <rule text="intro x"/>
          <tree><goal><concl type="A"/><hyp id="x" type="A"/>
        </goal></tree></tree>
      </cmpdrule>
      <tree><goal><concl type="A"/><hyp id="x" type="A"/>
    </goal></tree></tree>
  </cmpdrule><tree><goal><concl type="A"/><hyp id="x" type="A"/>
</goal></tree></tree>
```

We modified ProofWeb to be able to parse XML trees dumped by Coq and generate natural deduction diagrams (see Figure 3). Those diagrams may be requested

by the user's browser in special query requests. The diagrams are displayed in a separate frame in the interface along with the usual Coq proof state. If the user switches on the display of the diagrams, the client side requests them when no text is being processed.

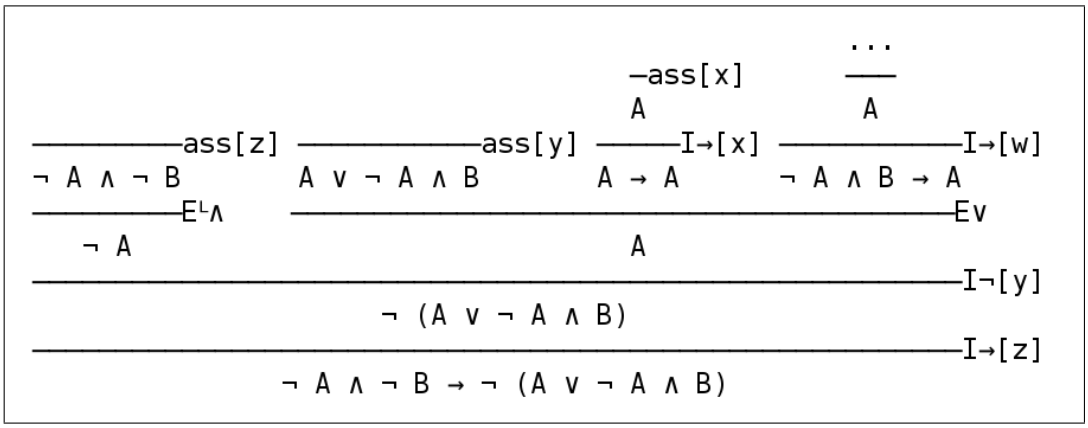


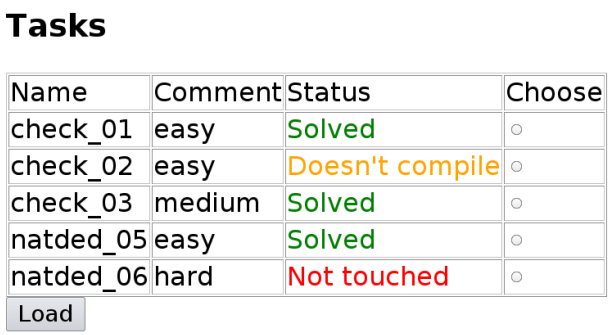
Fig. 3. A natural deduction tree as seen on the webpage (cropped screenshot).

6 The exercise set

Also part of the project is the development of a set of exercises for the students. For a particular course, a number of exercises assigned to the students. It is desirable that ProofWeb can be used as a complete course environment. So when a student logs in via the web interface as participant to a specific course, he is able to see the list of all the assigned tasks (see Figure 4). Every task has a certain status. The status can be one of the following:

- Not touched — When a particular exercise has not been opened, or has been opened but has not been saved.
- Does not compile — When the file has been edited and saved, but is not a correct Coq file. It can be because of real errors or because proofs are missing.
- Incorrect — The students are supposed to modify the given file only in designated places and to use only a set of allowed tactics. If the student uses a non-allowed too powerful tactic or just removes a task from the file it is marked as incorrect.
- Correct — Passed the verification by our tool.

The verification tool lexes the original task and the student's solution in parallel. The original solution includes placeholders that are valid Coq comments. Those placeholders mean that a particular place needs to contain a valid Coq term or a valid proof. For proofs the kind of proof determines the set of allowed tactics. For proofs and terms of given types the automatic verification is enough. However, there are tasks where students are required to give a definition of a particular object in type theory. For this kind of tasks manual verification by a teaching assistant of a course is required.



• **Tasks**

Name	Comment	Status	Choose
check_01	easy	Solved	○
check_02	easy	Doesn't compile	○
check_03	medium	Solved	○
natded_05	easy	Solved	○
natded_06	hard	Not touched	○

Load

Fig. 4. Tasks assigned to students and their status.

7 Outlook

At the moment of writing, the project ProofWeb is more or less half way, so this paper reports on work in progress. In this section we first discuss a main issue we are currently working on: adding the possibility of using the system for 'Fitch-style' natural deduction derivations. We then briefly comment on further points of current and future work.

7.1 Fitch-style deductions

The most important improvement is to add the possibility to use the system for so-called *Fitch-style* natural deduction proofs.[3] Fitch-style proofs have the graphical advantage over Gentzen-style proofs of being linear (as opposed to having a branching tree structure), which makes them more convenient to display for large proofs, like the ones constructed by the students for final assignments. Another name for these kind of proofs is *flag-style proofs*, because the assumptions of a subproof are often written in the shape of 'flags'. We are working on having the system display Fitch-style deductions. A basic version of this has already been implemented (see Figure 5), but needs further development. A number of issues arise. When a tactic creates a number of assumptions, should these be kept in one flag or should the system create multiple flags? Also, Fitch-style deductions may include repetitions of assertions assumed by flags. Most of these repetitions are redundant. However, not repeating assumptions immediately before they are used leads to very unreadable proofs. What should be done? We are currently looking at adapting the approach presented in [4] to Coq tactics.

7.2 Future work

Some of the other issues that currently are being worked on are:

- The course notes that are to accompany the system. These are still in a rudimentary stage.
- There is no separate web interface yet for the teacher to manage the student logins and the set of exercises for the course, nor to inspect the work of the students.

[3] This style of proof was initially developed by Stanisław Jaśkowski in 1934 and perfected by Frederic Brenton Fitch in 1952.

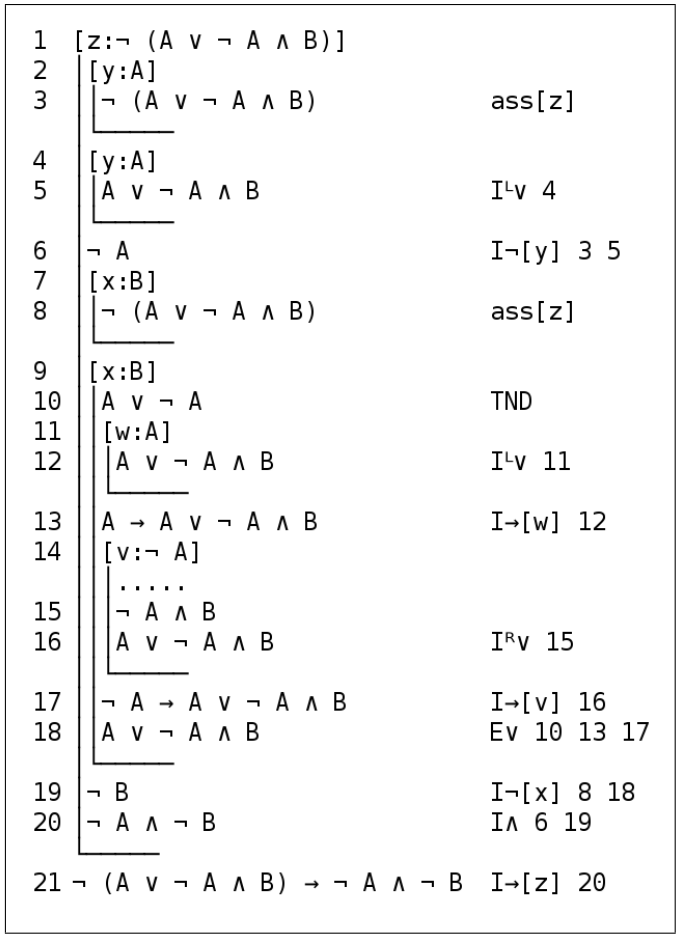


Fig. 5. A Fitch-style deduction as currently displayed by the system (implementation in progress).

At the moment this is only possible by logging on to the server through an ssh connection, and then listing and editing files manually. Clearly, a proper web interface for this is necessary.

- The deduction trees are currently rendered in an HTML IFrames, and can be optionally opened in a separate browser window to allow easy printing as PostScript or PDF. However students may need to use the trees in texts, and for that a dedicated TEX or image rendering of the trees could be implemented.
- The interface uses some web technologies that are not implemented in the same way in all browsers. It includes a small layer that is supposed to abstract over incompatible functionalities. Currently this works well with Mozilla compatible browsers (Firefox, Galeon, Epiphany, Netscape, ...). Also, some effort has been made to make the system work reasonably well with the most common versions of Internet Explorer. However, the compatibility of the system with most common web browsers is something that will need further attention.
- At the moment there is hardly any documentation of how to install and maintain the server. Our server currently is available to everyone who wants to experiment with our system, but there is no good guide available that explains how to install a server of his own. Because the server is still very much in a constant state of

flux, documenting the installation and maintenance processes is at this moment not yet reasonable. However, in the final phase of the project it will be important to also create this kind of documentation.

- With the current version of the system a log of each interaction of each student session is already stored on the server. Using these logs, it is possible to develop software for 'replaying' such a student session (possibly speeded up or slowed down). We are currently discussing whether it is useful to develop such an extension of the system.
- The system was designed in a way to be used in standard university courses. It might be useful to create a more complete online environment that would include introductory explanations and adaptive user profiles, therefore allowing students to learn logic without teacher interaction.

7.3 Beyond the project

If the development of ProofWeb is finished, a possibility is to integrate it with a system that supports the development of more serious proofs with the Coq system. One of the other projects that currently is being pursued in Nijmegen is the creation of a so-called 'math wiki' [8]. Here, traditional wiki technology is integrated with the same Coq front end that our system is based on.

7.4 Using the system

We think that it is important that our system is experimented with (and hopefully someday frequently used) by as many people as possible. For this reason, we currently offer the use of our system to anyone on the internet, even without any registration. The ProofWeb system can be tried at

http://prover.cs.ru.nl

References

[1] *Beweren en Bewijzen.*
URL http://www.cs.ru.nl/~wupper/B&B/index.html

[2] Bornat, R. and B. Sufrin, *Jape's quiet interface*, in: N. Merriam, editor, *User Interfaces for Theorem Provers (UITP '96)*, Technical Report (1996), pp. 25–34.

[3] Coq Development Team, "The Coq Proof Assistant Reference Manual Version 8.1," INRIA-Rocquencourt (2005).
URL http://coq.inria.fr/doc-eng.html

[4] Geuvers, H. and R. Nederpelt, *Rewriting for Fitch style natural deductions.*, in: V. van Oostrom, editor, *RTA*, Lecture Notes in Computer Science **3091** (2004), pp. 134–154.

[5] Gonthier, G., *A computer-checked proof of the Four Colour Theorem* (2006).
URL http://research.microsoft.com/~gonthier/4colproof.pdf

[6] *Inleiding Logica.*
URL http://www.cs.vu.nl/~tcs/il/

[7] Kaliszyk, C., *Web interfaces for proof assistants*, in: S. Autexier and C. Benzmüller, editors, *Proceedings of the FLoC Workshop on User Interfaces for Theorem Provers (UITP'06), Seattle*, 2006, pp. 53–64, to be published in ENTCS.

[8] Kaliszyk, C. and P. Corbineau, *Cooperative repositories for formal proofs* (2007), to be published in the proceedings of MKM 2007.

[9] Leroy, X., *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, in: *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2006), pp. 42–54.

[10] *Logic courseware.*
URL http://www.cs.otago.ac.nz/staffpriv/hans/

[11] *Logical Verification.*
URL http://www.cs.vu.nl/~tcs/lv/

[12] *SURF foundation.*
URL http://www.surf.nl/

Status Report on the Tight Integration of a Scientific Text-Editor and a Proof Assistance System

Serge Autexier

DFKI GmbH & Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.dfki.de/~serge)

Marc Wagner

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.ags.uni-sb.de/~mwagner)

Abstract

In order to foster the use of proof assistance systems, we integrated the proof assistance system ΩMEGA with the standard scientific text-editor TEXMACS. We aim at a document-centric approach to formalizing and verifying mathematics and software. Assisted by the proof assistance system, the author writes her document entirely inside the text-editor in a language she is used to, that is a mixture of natural language and formulas in LATEX style. In this paper we briefly describe the PLATΩ-system that realizes the integration and the mechanism that allows the author to define her own notation inside a document in a natural way, which is used to parse the formulas written by the author as well as to render the formulas generated by the proof assistance system. We illustrate the different facets of the support offered to the author by the proof assistance system by giving a worked example.

1 Introduction

The vision of a powerful mathematical assistance environment that provides computer-based support for most tasks of a mathematician has stimulated new projects and international research networks in recent years across disciplinary boundaries. Even though the functionalities and strengths of proof assistance systems are generally not sufficiently developed to attract mathematicians on the edge of research, their capabilities are often sufficient for applications in e-learning and engineering contexts. However, a mathematical assistance system that shall be of effective support has to be highly user-oriented. We believe that such a system will only be widely accepted by users if the communication between human and machine satisfies their needs, in particular only if the extra time spent on the machine is by far compensated by the system support. One aspect of the user-friendliness is to integrate formal modeling and reasoning tools with software that users routinely employ for typical tasks.

One standard activity in mathematics and the areas that are based on mathematics is the preparation of documents using some standard text preparation system like LATEX.

TEXMACS [20] is a scientific text-editor in the WYSIWYG paradigm that provides professional type-setting and supports authoring with powerful macro definition facilities like those in LATEX. As a first step towards assisting the authoring of mathematical documents, we integrated the proof assistance system ΩMEGA into TEXMACS using the generic mediator PLATΩ [22]. In this setting the formal content of a document must be amenable to machine processing, without imposing any restrictions on how the document is structured, on the language used in the document, or on the way the document can be changed. The PLATΩ system [21] transforms the representation of the formal content of a document into the representation used in a proof assistance system and maintains the consistency between both representations throughout the changes made on either side.

Such an integrated authoring environment should allow the user to write her mathematical documents in the language she is used to, that is a mixture of natural language and formulas in LATEX style with her own notation. To understand as far as possible the semantics of full natural language in a mathematical document we currently rely on annotations for the document structure that must be provided manually by the user. Although it might still be acceptable for an author to indicate the macro-structures like theories, definitions and theorems, writing annotated formulas (e.g. "`\F{in}{\V{x},\F{cup}{\V{A},\V{B}}}`" instead of "`x \in A \cup B`") is definitely not. We present a mechanism that allows authors to define their own notation and to use it when writing formulas within the same document. Furthermore, this mechanism enables the proof assistance system to access the formal content and use the same notation when presenting formulas to the author.

The paper is organized as follows: Section 2 gives a short introduction to the proof assistance system ΩMEGA, its main components MAYA and the TASK LAYER, as well as to the PLATΩ system that realizes the integration with the text-editor TEXMACS. Section 3 presents the annotation language for documents of the PLATΩ system, in particular for formulas. Inspired by notational definitions in text-books, we present the means the author has to define notations, from which an *abstraction* parser to read formulas and a corresponding *rendering* parser to render formulas according to the user-defined notation are generated. We sketch the basic mechanisms to accommodate efficiently modifications of the notations, to restrain ambiguities, to allow for the redefinition of notations and to use notations defined in other documents. Section 4 gives a worked example that illustrates the different facets of the support offered on that basis. We discuss related works in Section 5 before concluding in Section 6.

2 Preliminaries: ΩMEGA, MAYA, TASK LAYER and PLATΩ

The development of the proof assistance system ΩMEGA is one of the major attempts to build an all-encompassing assistance tool for the working mathematician or for the formal work of a software engineer. It is a representative of systems in the paradigm of *proof planning* and combines interactive and automated proof construction for domains with rich and well-structured mathematical knowledge. The ΩMEGA-system is currently under re-development where, among others, it has been augmented by the development graph manager MAYA, and the underlying natural deduction calculus has been replaced with the CORE-calculus [5].

The MAYA system [8] supports an evolutionary formal development by allowing users to specify and verify developments in a structured manner, it incorporates a uniform mech-

Figure 1. Architecture of integration of the text-editor and the ΩMEGA via the mediator PLATΩ

anism for verification in-the-large to exploit the structure of the specification, and it maintains the verification work already done when changing the specification. Proof assistance systems like ΩMEGA rely on mathematical knowledge formalized in structured theories of definitions, axioms and theorems. The MAYA system is the central component in the new ΩMEGA system that takes care about the management of change of these theories via its OMDOC-interface [13].

The CORE-calculus supports proof development directly at the *assertion level* [11], where proof steps are justified in terms of applications of definitions, axioms, theorems or hypotheses (collectively called *assertions*). It provides the logical basis for the so-called TASK LAYER [9], that is the central component for computer-based proof construction in ΩMEGA. The proof construction steps are: (1) the introduction of a proof sketch [23], (2) deep structural rules for weakening and decomposition of subformulas, (3) the application of a lemma that can be postulated on the fly, (4) the substitution of meta-variables, and (5) the application of an inference. Inferences are the basic reasoning steps of the TASK LAYER, and comprise assertion applications, proof planning methods or calls to external theorem provers or computer algebra systems (see [9,6] for more details about the TASK LAYER).

A formal proof requires to break down abstract proof steps to the CORE calculus level by replacing each abstract step by a sequence of calculus steps. This has usually the effect that a formal proof consists of many more steps than a corresponding informal proof of the same conjecture. Consequently, if we manually construct a formal proof many interaction steps are typically necessary. Formal proof sketches [23] in contrast allow the user to perform high-level reasoning steps without having to justify them immediately. The underlying idea is that the user writes down only the interesting parts of the proof and that the gaps between these steps are filled in later, ideally fully automatically (see also [19]). Proof sketches are thus a highly adequate means to realize the tight integration of a proof assistance system and a scientific text-editor.

The mediator PLATΩ [22] has been designed as a support system to realize the tight integration of a proof assistance system and a text-editor (see Figure 1). PLATΩ is connected with the text-editor by an informal representation language which flexibly supports the usual textual structure of mathematical documents. This semantic annotation language, called *proof language* (PL), allows for underspecification as well as alternative (sub)proof attempts. In order to generate the formal counterpart of a PL representation, PLATΩ separates theory knowledge like definitions, axioms and theorems from proofs. The theories are formalized in the *development graph language* (DL), which is close to the OMDOC the-

ory language supported by the MAYA system, whereas the proofs are transformed into the *tasklayer language* (TL) which are descriptions of TASK LAYER proofs. Hence, PLATΩ is connected with the proof assistance system ΩMEGA by a formal representation close to its internal datastructure.

Besides the transformation of complete documents, it is essential to be able to propagate small changes from an informal PL representation to the formal DL/TL one and the way back. If we always perform a global transformation, we would on the one hand rewrite the whole document in the text-editor which means to lose large parts of the natural language text written by the user. On the other hand we would reset the datastructure of the proof assistance system to the abstract level of proof sketches. For example, any already developed expansion towards calculus level or any computation result from external systems would be lost. Therefore, one of the most important aspects of PLATΩ's architecture is the propagation of changes.

The formal representation finally allows the underlying proof assistance system to support the user in various ways. PLATΩ provides the possibility to interact through context-sensitive service menus. If the user selects an object in the document, PLATΩ requests service actions from the proof assistance system regarding the formal counterparts of the selected object. Hence, the mediator needs to maintain the mapping between objects in the informal language PL and the formal languages DL and TL.

In particular, the proof assistance system supports the user by suggesting possible inference applications for a particular proof situation. Since the computation of all possible inferences may take a long time, a multi-level menu with the possibility of lazy evaluation is provided. PLATΩ supports the execution of nested actions inside a service menu which may result in a patch description for this menu.

Through service menus the user also gets access to automatic theorem provers and computer algebra systems which can be used to automatically verify conclusions and computations as well as suggest possible corrections. These and many more functionalities are supported by PLATΩ through its mechanism to propagate changes as well as the possibility of custom answers to the user of the text-editor.

A more detailed description of PLATΩ is given in [22]. Note that, generally we aim at an approach that is independent of the particular proof assistance system to be integrated. Therefore the proof language as well as the service menu language are parameterized over the sublanguages for definitions, formulas, references and menu argument content.

Presentational convention: The work presented in this paper has been realized in TEXMACS. Although the TEXMACS markup-language is analogous to LATEX-macros, one needs to get used to it: For instance a macro application like `\frac{A}{B}` in LATEX gets `<frac|A|B>` in TEXMACS-markup. Assuming that most readers are more familiar with LATEX than with TEXMACS, we will use a LATEX-syntax for the sake of readability.

3 Dynamic Notation

The PLATΩ system supports users to interact with a proof assistance system from inside the text-editor TEXMACS by offering service menus and by propagating changes of the document to the system and vice versa. Mediating between a text-editor and a proof assistance system requires to extract the formal content of a document, which is already a challenge in

Element	Arguments
\F	{*name*}{\B?, (\F |\V |\S)⋆ }
\B	{\V+ }
\V	{*name*, (\T |\TX |\TF)?}
\S	{*name*}
\T	{*name*}
\TX	{(\T |\TX |\TF), (\T |\TX |\TF) }
\TF	{(\T |\TX |\TF), (\T |\TX |\TF) }

Figure 2. Grammar of the annotation language for formulas.

itself if one wants to allow the author to write in natural language without any restrictions.

Therefore we currently use the semantic annotation language presented in [21] to semantically annotate different parts of a document. The annotations can be nested and subdivide the text into dependent theories that contain definitions, axioms, theorems and proofs, which themselves consist of proof steps like for example subgoal introduction, assumption or case split. The annotations are a set of macros predefined in a TEX$_{\text{MACS}}$ style file and must be provided manually by the author. We were particularly cautious that adding the annotations to a text does not impose any restrictions to the author about how to structure her text. Note that for the communication with the proof assistance system, also the formulas must be written in a fully annotated format whose grammar is shown in the table of Figure 2. `\F{name}{args}` represents the application of the function `name` to the given arguments `args`. `\B{vars}` specifies the variables `vars` that are bound by a quantifier. A variable `name` is denoted by `\V{name}` and may be optionally typed. A type `name` is represented by `\T{name}`. Complex types are composed using the function type constructor $\rightarrow$ represented by `\TF{type1, type2}`, or the pair type operator $\times$ represented by `\TX{type1,type2}`. Finally, a symbol `name` is denoted by `\S{name}`. For example, the formula $x \in A \cap (B \cup C)$ is represented by the fully annotated form `\F{in}{\V{x},\F{cap}{\V{A},F{cup}{\V{B},V{C}}}}`, the quantified formula $\forall x.\ x = x$ as `\F{forall}{\B{\V{x}},\F{=}{\V{x},\V{x}}}`. In many cases the type of a variable can be reconstructed using type inference or from bounded context variables, therefore typing variables is optional in our system.

The vision is on the one hand to combine our approach with the MATHLANG project [12] and on the other hand to use natural language analysis techniques for the semi-automatic annotation of the document structure, e.g. to automatically detect macro-structures like theories, definitions and theorems. Although these macro-structures might be annotated manually by the user at the moment, it is not acceptable to write formulas in fully annotated form. This motivates the need for an *abstraction* parser that converts formulas in LATEX syntax into their fully annotated form. Furthermore, we also need a *rendering* parser to convert fully annotated formulas obtained from the proof assistance system into LATEX-formulas according to the user-defined notation.

The usual software engineering approach would be to write grammars for both directions and integrate the generated parsers into the system. Of course, this method is highly efficient but the major drawback is obvious: the user has to maintain the grammar files together with her documents. In our document-centric philosophy, the only source of knowledge for the mediator and the proof assistance system should be the document in the text-editor. Therefore, instead of maintaining special grammar files for the parser, the

idea of dynamic notation is to synthesize these grammar informations from the definitions and notations occurring in the same document where they are used. This WYSIWYG style of defining notation starts from basic abstraction and rendering grammars for types and formulas, where only the base type *bool*, the complex type constructors $\rightarrow, \times$ [1] and the logic operators $\forall, \exists, \lambda, \top, \bot, \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ are predefined. This set of basic types and logic operators is a parameter of the PLATΩ system that has to be specified for each proof assistant system. A component of PLATΩ maps them into the input syntax of the particular proof assistant (e.g. Π, o) and the way back. Since mathematicians in general are not familiar with the type o, we decided to use the name *bool* instead. Based on that initial grammar the definitions and notations occurring in the document are analyzed in order to extend incrementally both grammars for dealing with new symbols, types and operators. The scope of a notation should thereby respect the visibility of its defining symbol or type, i.e. the transitive closure of dependent theories. Finally, all formulas are parsed using their theory specific *abstraction* parser.

Notations defined by authors are typically not specified as grammar rules. Therefore, we first need a user friendly WYSIWYG method to define notations and to automatically generate grammar rules from it. Looking at standard mathematical textbooks, one observes sentences like *"Let x be an element and A be a set, then we write $x \in A$, x is element of A, x is in A or A contains x."*. Supporting this format requires the ability to locally introduce the variables x and A in order to generate grammar rules from a notation pattern like $x \in A$. Without using a linguistic database, patterns like *x is in A* are only supported as pseudo natural language. Beside that, the author should be able to declare a symbol to be right- or left-associative as well as precedences of symbols.

We introduce the following annotation format to define the operator $\in$ and to introduce multiple alternative notations for $\in$ as closely as possible to the textbook style.

```
\begin{definition}{Function $\in$}
  The predicate \concept{\in}{elem \times set \rightarrow bool}
  takes an individual and a set and tells whether that
  individual belongs to this set.
\end{definition}

\begin{notation}{Function $\in$}
  Let \declare{x} be an individual and \declare{A} a set,
  then we write \denote{x \in A}, \denote{x is element of A},
  \denote{x is in A} or \denote{A contains x}.
\end{notation}
```

Figure 3 shows how the above example definition and notation appear in a TEXMACS document. A definition may introduce a new type by `\type{name}` or a new typed symbol by `\concept{name}{type}`. We allow to group symbols to simplify the definition of precedences and associativity. By writing `\group{name}` inside the definition of a symbol, this particular symbol is added to the group `name` which is automatically created if it does not exist. A notation may contain some variables declared by `\declare{name}` as well as the patterns written as `\denote{pattern}`. Furthermore, one can specify a

[1] Here $\times$ is not a pair type constructor, but only syntactic sugar that allows to write $\tau_1 \times \tau_2 \rightarrow \tau_0$ as short-hand for $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_0)$.

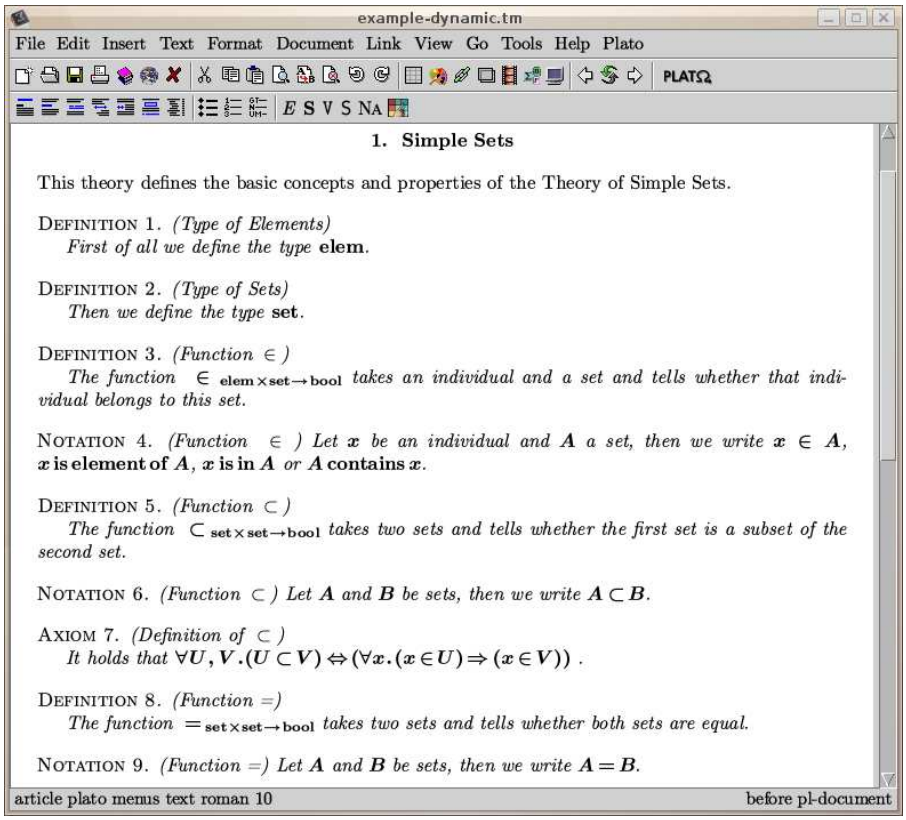


Figure 3. TEXMACS document with dynamic notation

symbol or group of symbols to be left- or right-associative by writing `\left{name}` or `\right{name}` inside the notation. Finally, precedences between symbols or groups are defined by `\prec{name1,...,nameN}`, which partially orders the precedence of these symbols and groups of symbols from low to high. Declarations are rejected if conflicting information is detected during the computation of the transitive closure. Information about association and precedence is considered to be visible in the whole theory whereas the visibility of notation starts at the position of its declaration. Please note that a notation is related to a specific definition by referring its name, in our example `Function $\in$`.

Starting the processing of a semantically annotated document, as for example the document shown in Figure 3, all surrounding natural language parts in the document are removed and the *abstraction* and *rendering* parsers and scanners are initialized with the initial grammars for types and formulas. The goal of processing the document is to produce a set of grammar rules for the respective grammars from the notational definitions given in the text. A top-down approach, that processes each definition or notation on its own, extending the grammars and recompiling the parsers before processing the next element, is far too inefficient for real time usage due to the expensive parser generation process [2] . The diagram shown in Figure 4 describes the procedure that we use to minimize the amount of parser generations as much as possible.

- Phase 1: All definitions are processed sequentially. For each definition the name of the

[2] The parser generator is implemented and compiled in Lisp to be part of PLATΩ. For instance, the parser generation for the example document takes $\approx 3sec$.

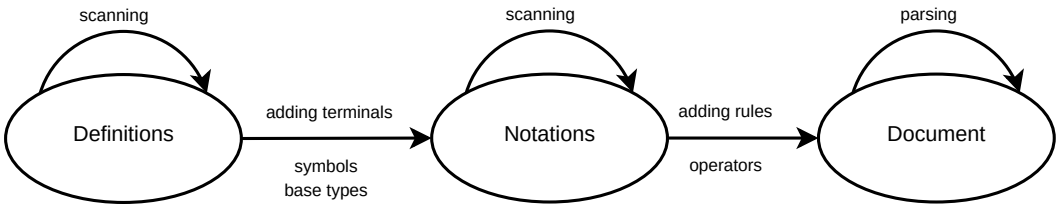


Figure 4. Procedure to minimize the amount of parser generations

introduced type or symbol is added to both grammars.

- Phase 2: All notations are processed sequentially. For each notation the introduced patterns are analyzed to generate rules for both grammars.
- Phase 3: *Abstraction* and *rendering* parsers are rebuilt and all formulas are processed.

The definitions and notations are analyzed to generate grammar rules for the *abstraction* grammar; in case different notations are defined for a symbol, this results in multiple grammar rules. For the *rendering* grammar we currently select the first given notation. The parser generator then creates the respective *abstraction* and *rendering* parsers. The details of the analysis and the parser creation can be found in [7].

3.1 Management of Change for Notations

The constructed *abstraction* parser is for one version of the document. When the author continues to edit the document, it may be modified in arbitrary ways, including the change of existing definitions and notations. Before the modified semantic content of the document is uploaded into the proof assistance system, we need to recompute the parsers and parse the formulas in the document. Always starting from scratch following the procedure described in the previous section is highly inefficient and may jeopardize the acceptance by the author of the system if that process takes too long. Therefore there is a need for management of change for the notational parts of a document and those parts that depend on them. The management of change task has three aspects:

- First, we must determine any modifications in the notational parts: we re-process all definitions and notations of the document and obtain a new set of grammar rules. By caching these sets of rules for each symbols, we can determine the grammar changes.
- Second, we must adjust only those parts of the grammar that are affected by the determined modifications and adjust the parsers accordingly. If the modification of the grammar is non-monotonic, i.e. some rules have been removed or changed, we currently have to recompute the whole parser from scratch. This is for instance the case if we change a notation for some symbol, e.g. replacing *"$A \subset B$"* by *"$B \supset A$"*, but not if we add an additional alternative notation for a symbol, like allowing *"A is subset of B"* in addition to *"$A \subset B$"*. If the grammar is simply extended, we can optimize the creation of the new parser (see [7] for details).
- Third, we must re-parse (resp. re-render) the formulas of the document that are affected by the changes. Once the parser has been adjusted we need to re-parse those formulas the author has written or changed manually, e.g. if the formula in the axiom of Figure 3 would have been replaced by $\forall U,V.\ (U\ is\ subset\ of\ V) \Leftrightarrow (\forall x.\ (x\ is\ in\ U) \Rightarrow (V\ contains\ x))$. Furthermore, we have to re-render those formulas that were generated

by the proof assistance system. To this end we store the following information on formulas in the document: for each formula we have a flag indicating if it was generated by the proof assistance system, the corresponding fully annotated formula, and the set of grammar rules that were used for parsing (resp. rendering) that formula:

- If the formula was written by the author, the associated fully annotated formula and the grammar rules are the result of the *abstraction* parser.
- If a formula was generated by the proof assistance system, that formula is the result of *rendering* the fully annotated formula obtained from the proof assistance system. The stored grammar rules are those returned by the *rendering* parser.

Note that we do not prevent the author from editing a generated formula. As soon as the author edits such a formula, the flag attached to the formula is toggled to "user" and the cached fully annotated version and grammar rules are replaced during the next *abstraction* parsing of the formula. The stored information is used to optimize the next parsing (resp. rendering) pass over the document: A formula is only parsed from scratch if at least one of the used grammar rules has been modified or deleted, or if either the user or the proof assistance system has changed the formula (resp. the fully annotated formula).

3.2 Ambiguities, Dependencies and Libraries

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered by the authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. We exploit the theory structure contained in a document to reduce the ambiguities the author would have to deal with and also support the redefinition of notations. With respect to added-values, we provide checks if a notation is used before it has been introduced in a document and, most importantly, we support an author to build on formalizations contained in other documents.

Ambiguities. The *abstraction* parser returns all possible readings and during the parsing process can try to make use of type-reconstruction provided by a so-called *refiner* [18] to eliminate possible readings that are not type-correct. This uses the logical context of a formula which is determined by the theory it occurs in: the different parts of a document must be assigned to specific theories. New theories can be defined inside a document and built on top of other theories. The notion of theory is that of OMDOC [13] respectively development graphs [15]. The other possibility consists of avoiding some ambiguities that would arise when sticking to have a single parser by having different parsers for different theories. As an example consider a theory of the integers with multiplication with the notation *"$x \times y$"* and a completely unrelated theory about sets and Cartesian products with the same notation. This typically is a source of ambiguities that would require the use of type information to resolve the issue. Using different parsers for different theories completely avoids that problem.

Redefining Notations. When importing a theory, we want to reuse the formal content, but possibly adapt the notation used to write formulas. This occurs less frequently inside a single document, but occurs very often when using a theory formalized in a different

document. Since we linked the parsing (and hence the rendering) to the individual theories, we allow to redefine notations for symbols inherited from other theories. The grammar rules for a parser are determined by including for each imported symbol that notation which is closest in the import hierarchy of theories. If there are two such theories [3], a conflict is raised and the author must interactively advise which notation to use.

Dependencies. A parser and the associated renderer are attached to a theory and each position in the document belongs to a theory. Therefore, it is possible that within a specific theory, a formula uses the notation of some symbol although the definition of that notation only occurs afterwards in the document. We notice such situations by comparing the position in the document where grammar rules are defined and where they have been used to parse a formula [4]. If we determine such a situation, we could try to re-arrange the document by inspecting the grammar rules used for *abstraction* and *rendering*. However, our impression is that most authors would be upset if their document is rearranged automatically. Therefore, we only notify the author in these situations.

Libraries. A library mechanism is the key prerequisite to support the development of large structured theories. We carry over that concept to the document-centric approach we are aiming at by extending the citation mechanism that is commonly used in documents. PLATΩ provides a macro to cite a document *semantically*, i.e. it will not only be included in the normal bibliography of the document, but the formalized content of the document is included. Aside from the extracted formalizations we extract the notations contained in that document. The structured theory approach for parsers turns out to be again particularly beneficial to support that process and to redefine the notations for imported symbols.

4 A Worked Example

In this section we will illustrate the integration by discussing a worked example in the theory of Simple Sets. The mediation between the informal representation in the text-editor and the formal representation in the proof assistance system will be described on an abstract level. All details on the communicated documents, patch descriptions and menus for this example can be found in [21].

Since the TEXMACS interface for proof assistance systems is under continuous development, a PLATΩ plugin for TEXMACS has been developed by the ΩMEGA group that maps the interface functions of PLATΩ to the current ones of TEXMACS and which defines a style file for PL macros in TEXMACS. In the following example, we use this plugin to establish a connection between TEXMACS and PLATΩ's XML-RPC server.

In the text-editor, we have written an example document with the semantic annotation language PL (defined in [21]). The theory *Simple Sets* in this document contains for example definitions and axioms for $\subset$, $=$, $\cup$ and $\cap$. Figure 3 (p. 55) and the top of Figure 5 (p. 59) show the theory as seen in TEXMACS. Furthermore, we have written a theory *Distributivity in Simple Sets* which imports all knowledge from the first theory *Simple Sets*. This second theory consists of a theorem about the *Distributivity of* $\cap$. The user has already started a proof for this theorem by introducing two subgoals. Figure 6 shows the theory as

[3] The theories can form an acyclic graph which may lead to the Nixon diamond scenario.
[4] A similar dependency is between a definition of a concept and the definition of its notation.

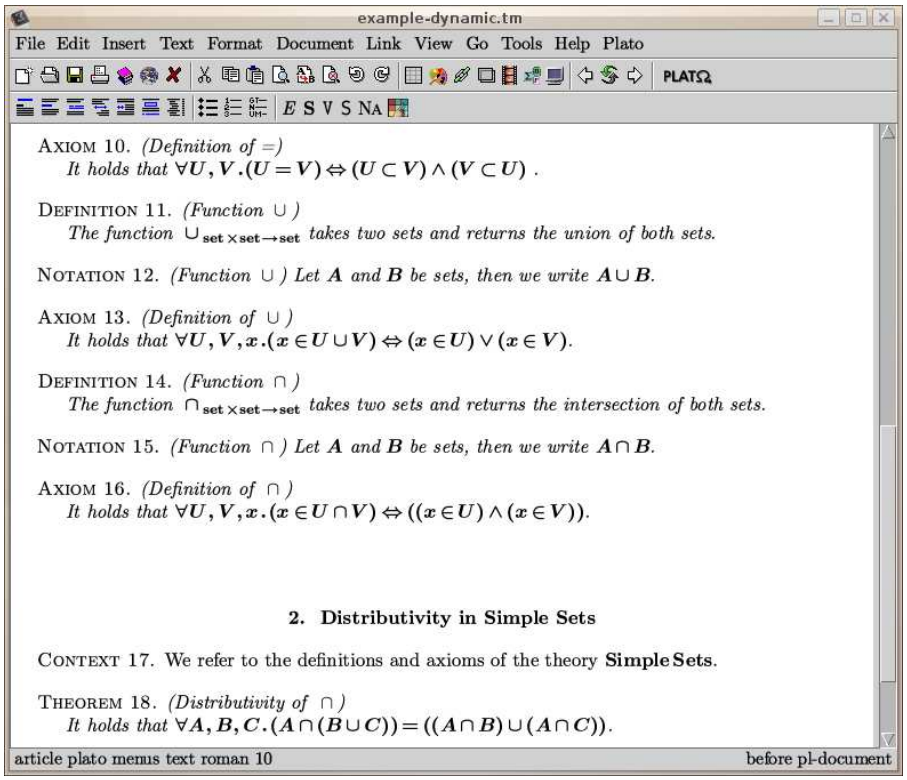


Figure 5. Theories *Simple Sets* and *Distributivity in Simple Sets* in TEXMACS

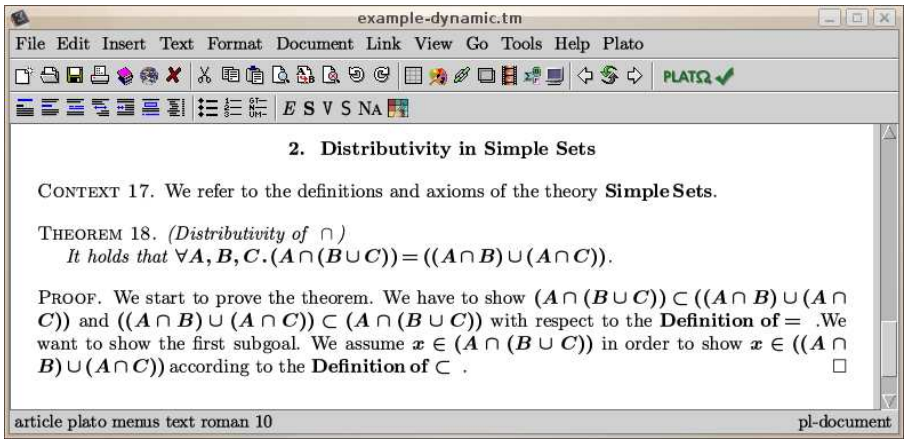


Figure 6. Manually written partial proof in TEXMACS

seen in TEXMACS.

The PL macros contained in the document must be provided by the user and are used to automatically extract the corresponding PL document, the informal representation of the document for PLATΩ. Uploading this PL document, PLATΩ creates a DL document containing definitions, axioms and theorems in a representation close to OMDOC that MAYA takes as input for the creation of a development graph. The partial proof is transformed into a TL document, an abstract representation for the TASK LAYER.

Further developing the document, the user has started to prove the first subgoal by deriving a new subgoal and introducing an assumption (see Figure 6). In general, the difference

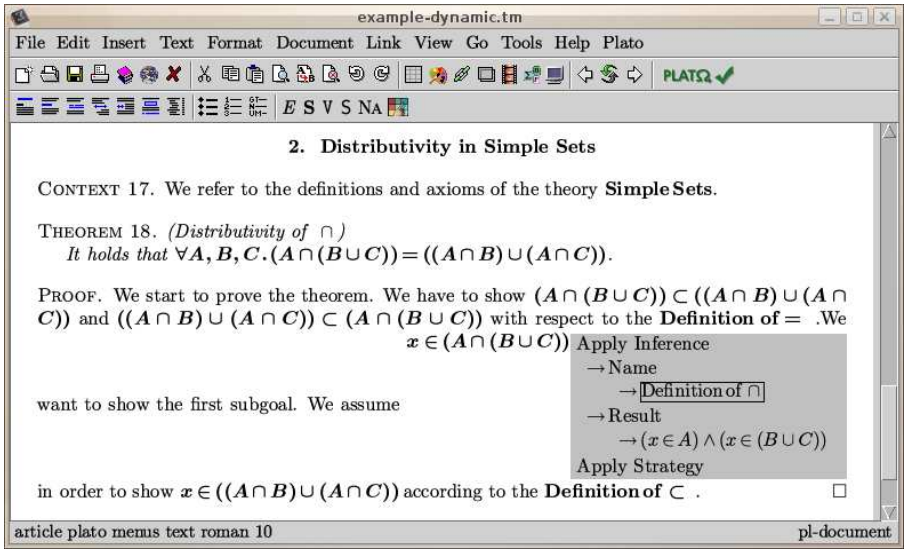


Figure 7. Interactive Proof Construction via Service Menus

with respect to the last synchronized version of the document should be computed and sent to PLATΩ. At the moment, TEXMACS is not able to compute these differences, therefore the whole document is uploaded again and PLATΩ computes the differences in the form of a patch. That patch for the informal PL document is then transformed by PLATΩ to a patch for the formal representations in DL and TL. In our example, the modifications do not affect the theory knowledge and the transformation only results in modifications for the representation of the TASK LAYER. Altogether, the user is able to synchronize her informal representation in the text-editor document with the formal representation in the proof assistance system.

The next interesting feature of PLATΩ is the possibility of getting system support from the underlying proof assistance system. Selecting the recently introduced formula in the assumption, the user requests a service menu from PLATΩ. Requesting services for the corresponding task in the TASK LAYER, a list of available inferences is returned to PLATΩ. In order to answer quickly to the text-editor, we generate nested actions that allow to incrementally compute the formulas resulting from the application of an inference rather than to precompute all possible resulting formulas for all available inferences. Note that the inferences are automatically generated from the axioms, lemmas and theorems contained in the document.

The menu is displayed to the user in TEXMACS, who can select which inference to apply. The selection triggers the computation of all resulting formulas, for instance for the inference **Definition of** ∩, defined by the corresponding axiom. PLATΩ tells the text-editor how to change the menu by sending a patch description for the menu. The user selects the desired formula (see Figure 7) which triggers the application of the inference. PLATΩ calls the TASK LAYER for the application of the selected inference in order to obtain the chosen formula. The TASK LAYER performs the requested operation which typically modifies the proof of TASK LAYER. This modification is transformed by PLATΩ into a patch description for the formal representation in TL and subsequently into a PL patch for the informal document in TEXMACS, which is then sent to the text-editor. Furthermore, the menu is closed by sending a patch description which removes it. Currently, the new proof frag-

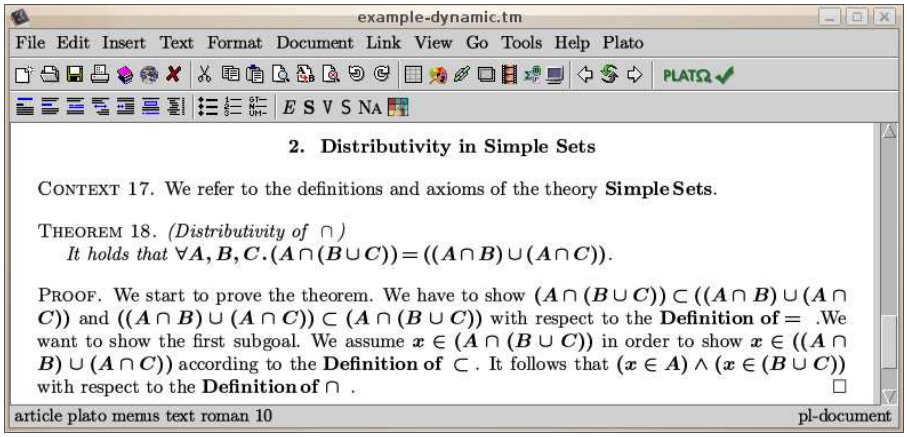


Figure 8. Modification of the Proof in TEXMACS by PLATΩ

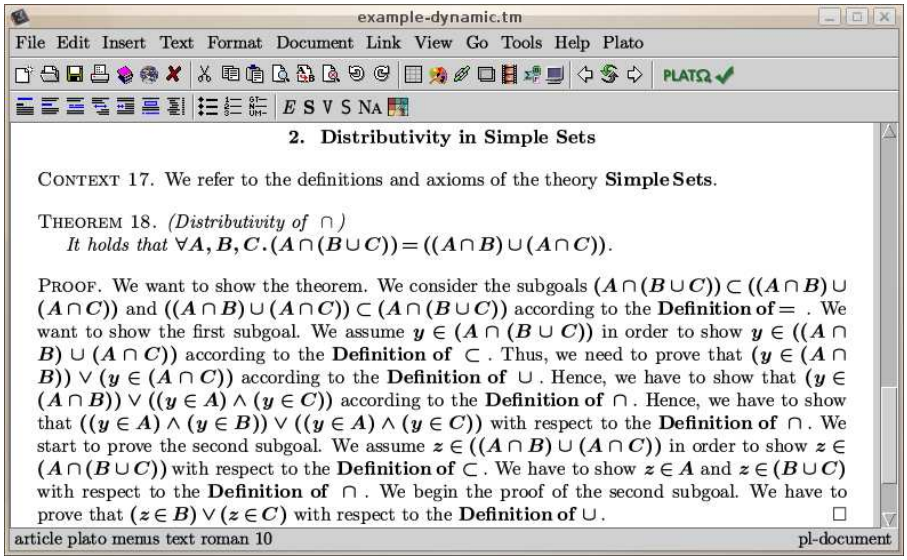


Figure 9. Automatically Constructed Proof in TEXMACS.

ments are inserted using predefined natural language fragments to describe the proof steps and the current rendering parser is used to render the formula in the notation defined by the author. Future work will consist of integrating the natural language proof presentation system P.REX [10] into PLATΩ, in order to generate true natural language output for the proof steps added by the proof assistance system. The text-editor finally patches the document according to this patch description. The resulting document is shown in Figure 8.

Aside from interactive proof construction, the author can call a proof procedure that searches for a proof on the TASK LAYER. In this case a sequence of proof steps is added to the TASK LAYER which are then transformed into patches for PLATΩ. Figure 9 shows such a proof in TEXMACS which uses the rendering parser to present the new formulas.

Note that the user can change any part of the document, including the parts generated by the proof assistance system. Due to the maintenance of consistent versions, the further development of the document can be a mix of manual authoring by the user and interactive authoring with the proof assistance system.

5 Related Work

To our knowledge there has not been any attempt to integrate a proof assistance system with text-editors in the tight and flexible way as done via PLATΩ. Approaches like AUTOMATH, MIZAR, ISAR, and THEOREMA do not consider the input document as an independent, first-class citizen with an internal state that has to be kept consistent with the formal representations in the proof assistance system while allowing arbitrary changes on each side. The first attempts in that direction have been carried out in the context of the MATITA prototype which used to have an integrated component to re-annotate computer generated representation of proofs. Since re-generation was a slow non-incremental operation based on complex XSLT style-sheets, re-annotating the text was an unpleasant experience for the user, which hampered user-acceptance of that part of the MATITA system. A similar attempt in that direction has been carried in the context of PROOF GENERAL [3]. In PROOF GENERAL the user edits a central document in a suitable editing environment, from which it can be evaluated by various tools, such as a proof assistance system, which checks whether the document contains valid proofs, or a renderer which typesets or renders the document into a human-oriented documentation readable outside the system. However, the system is only an interface to proof assistance systems that process their input incrementally. Hence, the documents edited in PROOF GENERAL are processed incrementally in a top-down manner, parts that have been processed by the proof assistance system are locked and cannot be edited by the user. Furthermore, the documents are in the input format of the proof assistance systems rather than in the format of some type-setting program. Though we have tried to design the functionalities and representation languages in PLATΩ's interface as general as possible, future work will have to show that PLATΩ can be easily adapted to different proof assistance systems (like PROOF GENERAL).

With respect to supporting the definition of new notations that are used for type-setting, the systems ISABELLE [16] and MATITA [2] are closest. ISABELLE comes with type-setting facilities of formulas and proofs for LaTeX and supports the declaration of the notation for symbols as prefix, infix, postfix and mix-fix. Furthermore, it allows the definition of *translations* which are close to our style of defining notations. The main differences are: ISABELLE processes the input document containing the declaration of the notation with a top-down mechanism and finally generates an output document in LaTeX. Since in our WYSIWYG setting, input and output document are physically the same document, the document is processed in multiple phases: first the definition of types and symbols are identified, second the parser rules are synthesized from the declarations of notation, third the document is processed with a family of local parsers. Due to the batch processing paradigm of ISABELLE, there are no mechanisms to efficiently deal with modifications of the notation, which is crucial in our interactive authoring environment.

In the context of MATITA Padovani and Zacchiroli also proposed a mechanism of *abstraction* and *rendering* parsers [17] that are created from notational equations which are comparable to the grammar rules we generate from the notational definitions. Their mechanism is mainly devoted to obtain MathML representations [1] where a major concern also is to maintain links between the objects in MathML to the internal objects. Similar to ISABELLE, the input document is processed top-down and is separated from the output document in MathML. Also, they do not consider the effect of changing the notations and to efficiently adjust the parsers. The main difference to the approaches in ISABELLE and

MATITA is that in our approach the parsers and renderers are part of the user interface and not of the proof assistance system.

Audebaud and Rideau presented in the context of the COQ proof assistant a command line interface connection with TEXMACS called TMCOQ [4]. Furthermore Mamane and Geuvers are developing a more document-centric proof script interface called TMEGG [14] which uses similar techniques as PROOF GENERAL with respect to the batch processing of the document.

6 Conclusion

In order to enable a document-centric approach for formalizing mathematics and software, the added-values offered in an assisted authoring environment must outweigh the additional burden imposed to the author compared to the amount of work for a non-assisted preparation of a document. We presented the mediator PLATΩ that allows the user to write her mathematical documents in the language she is used to, that is a mixture of natural language and formulas, and to define and use her own notation inside a document.

Building theory-specific parsers from these notation definitions, PLATΩ automatically builds up the corresponding formal representation from formulas written by the user in the LATEX-style she is used to and renders the formulas produced by the proof assistance system. It provides a sophisticated management of change for notations and takes further care of the maintenance of consistent versions in TEXMACS and the proof assistance system. All kinds of services of the underlying proof assistance system regarding the formal representation can be provided inside the text-editor through PLATΩ as context-sensitive service menus.

This paper illustrates the major aspects that are supported in the current version of the integration. Further work will consist of reducing the amount of annotations the user has to provide by employing natural language analysis techniques. Finally, we plan to increase the interoperability with other proof assistant systems as well as tutorial dialogue systems.

References

[1] Mathematical Markup Language (MathML) Version 2.0. W3c recommendation 21 february 2001. Technical report, `http://www.w3.org/TR/MathML2`, 2003.

[2] A.Asperti, C. Sacerdoti-Coen, E.Tassi, and S.Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving*, 2007. To appear.

[3] D. Aspinall, C. Lüth, and B. Wolff. Assisted proof document authoring. In M. Kohlhase, editor, *Mathematical Knowledge Management MKM 2005*, volume 3863 of *LNAI*, pages 65– 80. Springer, 2006.

[4] P. Audebaud and L. Rideau. TEXMACS as authoring tool for publication and dissemination of formal developments. In D. Aspinall and C. Lüth, editors, *5th Workshop on User Interfaces for Theorem Provers (UITP'03)*, ENTCS. Elsevier, September 2003.

[5] S. Autexier. The CORE calculus. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, July 2005. Springer.

[6] S. Autexier and D. Dietrich. Synthesizing proof planning methods and Ωants agents from mathematical knowledge. In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*, pages 94–109. Springer, 2006.

[7] S. Autexier, A. Fiedler, T. Neumann, and M. Wagner. Supporting user-defined notations when integrating scientific text-editors with proof assistance systems. In M. Kerber and R. Miner, editors, *Proceedings of MKM'07*. Springer, 2007.

[8] S. Autexier and D. Hutter. Formal software development in Maya. In D. Hutter and W. Stephan, editors, *Festschrift in Honor of J. Siekmann*, volume 2605 of *LNAI*. Springer, February 2005.

[9] D. Dietrich. The Task Layer of the ΩMEGA System. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[10] A. Fiedler. *User-adaptive Proof Explanation*. Phd thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[11] X. Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[12] F. Kamareddine, M. Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In M. Kohlhase, editor, *MKM 2005, Fourth International Conference on Mathematical Knowledge Management*, LNAI 3863, pages 217–233. Springer, 2006.

[13] M. Kohlhase. *OMDOC - An Open Markup Format for Mathematical Documents [Version 1.2]*, volume 4180 of *LNAI*. Springer, August 2006.

[14] L. E. Mamane and H. Geuvers. A document-oriented Coq plugin for TEX$_{\text{MACS}}$. In M. Kerber and R. Miner, editors, *Proceedings of MKM'07*. Springer, June 2007.

[15] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, April 2006.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2.

[17] L. Padovani and S. Zacchiroli. From notation to semantics: There and back again! In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*. Springer, August 2006.

[18] C. Sacerdoti-Coen and S. Zacchiroli. Efficient ambiguous parsing of mathematical formulae. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proceedings of the Third International Conference on Mathematical Knowledge Management*, volume 3119 of *LNCS*, pages 347–362, Bialystok, Poland, September 2004. Springer.

[19] J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof development with OMEGA: $\sqrt{2}$ is irrational. In M. Baaz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[20] J. van der Hoeven. Gnu TEX$_{\text{MACS}}$: A free, structured, WYSIWYG and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[21] M. Wagner. Mediation between text-editors and proof assistance systems. Diploma thesis, Saarland University, Saarbrücken, Germany, 2006.

[22] M. Wagner, S. Autexier, and C. Benzmüller. PLATΩ: A mediator between text-editors and proof assistance systems. In C. Benzmüller S. Autexier, editor, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, ENTCS. Elsevier, August 2006.

[23] F. Wiedijk. Formal proof sketches. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs: Third International Workshop, TYPES 2003*, LNCS 3085, pages 378–393, Torino, Italy, 2004. Springer.

How to Teach to Write a Proof

Adam Naumowicz[1,2]

Institute of Computer Science
University of Białystok
Poland

Abstract

In this paper we present methodological foundations of courses employing the MIZAR proof-checking system, which are currently part of the obligatory curriculum for computer science students at the University of Białystok.

Keywords: MIZAR in education, the art of proving, natural deduction, automated proof checking.

1 Introduction

The title of this paper is intended to cause the reader's recollection of the widely-discussed and seminal Leslie Lamport's paper "How to Write a Proof" ([3]). In that paper Lamport proposed a method of writing proofs which "makes it much harder to prove things that are not true". Nowadays there are numerous systems that make it "almost completely impossible" to prove things that are not true, with the "almost" covering the always-non-zero probability of hardware and software faults. A recent comparison of leading proof systems compiled by Wiedijk ([11]) shows that today there is no monopoly for the best proof style – writing proofs considered as formal depends very much on the applied system's foundations and philosophy as well as the intended goal of creating the proof.

MIZAR is a proof-checking system aimed at developing formal mathematics in a rigorous way under the control of a computer, but without unnecessary departure from the standard mathematical practice (see e.g. [7], [10]). To this end, the system has been equipped with an underlying proof language quite close to the so-called "mathematical vernacular" based on a declarative style of natural deduction. Its distinctive features, which can be inferred from the aforementioned comparison (see

[1] This work has been partially supported by the FP6 IST project no. 510996 *Types for Proofs and Programs* (TYPES).

[2] Email: adamn@mizar.org

[11]), make it one of the most "mathematical in spirit" of all the systems and include inter alia:

- readable proof input files,
- ZFC set theory and classical logic forming the base,
- the use of (dependent) types,
- low but quite efficient automation,
- large mathematical standard library.

Considering these features, it is not surprising that the bibliography of this project [3] shows a long history of using its various versions in mathematics instruction on different levels – from secondary school courses to writing PhD theses (see e.g. [4],[8] and [6]).

Owing especially to the readability and writability of proofs, the Mizar system is now renowned for the biggest library of formalized mathematical data, Mizar Mathematical Library (MML) [4]. The system has been involved in several big formalization projects, attracted almost two hundred authors of serious contributions who represent more than a dozen countries, despite being initially developed in a much disadvantageous environment behind the 'Iron Curtain' in Poland through 1980s and 1990s.

2 Whom to teach writing proofs?

For over three decades now it has been tried to get working mathematicians more involved in the use and development of proof-assistants. Nowadays mathematicians utilize lots of scientific software to name just computer algebra systems, geometric presentation tools, etc. but for some reasons a widespread adoption and dissemination of proof-assistants in mathematical practice is still far ahead on the horizon. This dissemination process seems to be mainly impeded by the fact that the state-of-the-art proof-assistants are still not considered useful enough in the research aimed at obtaining new results. As Lamport puts it in [3], "Mathematicians tend to be conservative, and many are unwilling to consider that there might be a better way of writing proofs."

Worse still, the mathematical community presents a very sceptical attitude towards various initiatives, like the utopian but very inspiring QED project [5] for instance, directed at advancing research in this field. Despite the evidence collected recently that it is certainly possible to write fully formal mathematical expositions/formalizations for quite advanced mathematics, be it classical theories with well-established theorems ([2]), specialized monographs ([1]) or recent mathematical journal papers ([5]), the de Bruijn factor seems still too high to persuade working mathematicians to make the extra effort required by proving at least parts of their work rigorously under the control of a proof-assistant system.

Therefore, as mathematicians find the length of formal proofs and the unfamiliar

[3] The Bibliography of the Mizar Project, URL: http://mizar.org/project/bibliography.html.
[4] See the statistics presented at http://mmlquery.mizar.org.
[5] There is a QED archive at http://www-unix.mcs.anl.gov/qed/.

format rather intimidating, it has been observed that instead of trying to convince this community, a new approach worth trying is to appeal to the new generation. Arguably, today's students who do not yet carry any such bias of traditional mathematical practice may be even better users of proof-assistants if we manage to instill the idea into their minds early enough. When Lamport proposed his structured proof method presented in [3], he envisaged a growing interest in proving not only pure mathematics, but also various computer science applications. In particular he intended to use his system for proving the correctness of algorithms. Today this is done in practice, and it shows that apart from mathematicians an important and prospective group of users of various proof tools are also computer scientists.

3 How difficult is it to learn writing proofs with Mizar?

As Wiedijk aptly pointed out [6], *proof assistants tend to resemble their implementation language*, and so Mizar is about as complex as the Pascal programming language. If this is really the case, then the answer to the above question should definitely be "Not at all".

Let us remind that Pascal is an imperative computer programming language, developed around 1970 by Niklaus Wirth as a language particularly suitable for structured programming. A derivative known as Object Pascal was later designed for object oriented programming – this is the language that the current Mizar system is implemented in.

Initially, Pascal was a language intended to teach students structured programming, and generations of students have "cut their teeth" on Pascal as an introductory language in undergraduate courses. Owing to that, Pascal is far too often considered by many as suitable ***only*** for educational purposes. Once popular criticism in the spirit of Brian Kernighan's famous paper in defense of the C language "Why Pascal is Not My Favorite Programming Language" – although almost completely irrelevant to modern Pascal variants – is still a source of unjust prejudice in many computer science communities.

Nevertheless, variants of Pascal are still widely used today and all types of Pascal programs can be used for both education and "serious" software development. To name just a few notable examples, the original operating system of Apple Lisa computers was once coded in Pascal, and the primary high-level language used for development in the first couple of years of the Macintosh was also Pascal. Additionally, the popular typesetting system TEXwas written by Donald E. Knuth in WEB – the original literate programming system using Pascal.

Indeed, in many respects Mizar is similar to its implementation language. Even on the syntactic level there are certain similarities which allow to claim that learning Mizar should not in general be more complicated than learning programming in Pascal.

The lexis of Object Pascal consists of 29 special symbols, e.g. `#`, `[`, `@`, `<=`, etc. and 65 reserved words, e.g. `if`, `then`, `procedure`, etc. together with 39 so-

[6] See Wiedijk's slides to the "Formalization of Mathematics" course at the TYPES Summer School in Göteborg – August 2005 (http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/wiedijk_sl.pdf)

called "directives" – words with a reserved meaning only in certain contexts, e.g. `abstract`. In Mizar we have 27 special symbols and 110 reserved words (6 of them are not actually implemented yet), so the numbers are almost identical! What is more, there are 10 symbols shared by both languages and also 15 identical reserved words – it is quite surprising for two so much different projects – a proof language and a programming language. The most significant difference on the syntactic level, however, is that current Mizar, unlike Pascal, is case-sensitive.

But even more importantly, Mizar has semantic features which make it very suitable for teaching purposes, just like Pascal. An important similarity is surely due to the high level of both languages, as they tend to use as many as possible words from the natural English. On the one hand this is why there is this number of shared reserved words. On the other hand, this allows to create constructs and expressions close to natural language, and then relatively easy to acquiesce by new users. The same concerns several features we can describe as "syntactic sugar" – their sole purpose is to make the source text more readable for humans. However, these features are devised with the intention to preserve the source text's conciseness rather than make it much more verbose. This way the problems we may find learning the syntax of other languages which use elements of natural language excessively (like COBOL or SQL) are avoided.

Moreover, both the Pascal and Mizar languages are notably highly structured and also strongly typed. The latter is especially responsible for supporting the production of rigorous and semantically unambiguous expositions.

Pascal being an imperative programming language allows to manipulate the state (memory) of the machine by means of variables and other more complex expressions. Mizar's imperative part consists of steps of natural deduction, e.g. `assume`, `take`, `consider`, etc. – they allow to manipulate the current proof-goal, the so-called *thesis*. These imperative constructs, however, are used in a declarative way – stating what is to be proved rather than how to prove it.

Another feature of Mizar inherited directly from its implementation language is the way one works with the system. This is often called "lazy interaction" – not a full interaction, but rather running the system on a source text in order to reveal errors and then correct them using the step-wise refinement method. Mizar reports its errors in much the same way a Pascal compiler reports warnings and errors. As Mizar does not stop processing on encountering the first error, its error recovery mechanism allows to check correctness of incomplete texts similarly to compiling modularized programs. This is particularly useful when users want to develop various parts of a proof e.g. postponing a proof or its more complicated parts. Especially in the educational context this can be useful to first sketch a proof's structure, revise it when needed, and also develop parts independently by several people.

It should be noted that although Mizar possesses certain didactic qualities, its developers hope this will never make it subjected to so much undeserved prejudice as Pascal.

4 Basic framework of Mizar-aided courses at the University of Białystok

The Mizar system has been developed at the University of Białystok (formerly the Białystok branch of Warsaw University) since 1970s. The research involved the members of the mathematics department, so naturally all teaching experiments concerned local mathematics students. The teaching has never had a permanent position in the mathematics curriculum – Mizar-aided classes were organized mainly as voluntary monographic courses, e.g. "Lattice theory", "Category Theory", "Topology", etc. The situation changed when there emerged a new university unit – the department of Computer Science and then the core Mizar developers formally became its stuff and were assigned teaching duties concerning CS students. This gave the opportunity to instill more Mizar-based instruction into the curriculum.

At the same time, basic issues related to the use of computer tools that some mathematics students used to have – with CS students became far less problematic. The distribution of the system and teaching materials, as well as assessment can now be done easily via the Internet. And although the teaching is currently being done with the students gathered in class at a certain time, it would be possible to use the Internet medium to even more extent reducing the involvement on the side of the instructors.

Additionally, as the Mizar-oriented teaching is done by instructors who are also involved in teaching programming, some elements of the methodology of teaching programming languages can also be used. Currently the CS curriculum contains two one-semester Mizar-based courses for undergraduate and graduate students:

- Undergraduate level:
  (i) "Introduction to logic and set theory"
  (ii) "Applied logic"
- Graduate level:
  (i) "Software verification"
  (ii) "Proof verification"

4.1 Undergraduate level

The "Introduction to logic and set theory" course is devised to introduce the fundamental formal mathematical apparatus and form a basis for other strictly mathematical subjects like "Discrete mathematics" or "Mathematical analysis". The course is taken during the first semester of study, so there is virtually no knowledge that can be assumed, since the mathematics education in secondary schools often varies. Therefore the accompanying lecture must run in a way in parallel – introducing the standard mathematical symbolism and its Mizar counterparts together with hints to "the art of proving". In brief, the syllabus comprises:

- logical formulae and basic structures of conditional proofs,
- Boolean properties of sets (also Venn diagrams),
- families of sets and their properties,
- binary relations (composition, the converse relation, selected properties - e.g.

reflexivity, transitivity, etc.),

- functions (domain and codomain, image, etc.).
- equivalence relations, partitions and ordering relations.

The other undergraduate course, "Applied logic", heavily depends on the knowledge covered by the previous one. The students are supposed to know how to use all basic proof techniques. The Mizar symbolism is dominating as there is not much purely mathematical contents in the syllabus:

- Peano arithmetic,
- various forms of the induction principle,
- higher-order reasoning with Mizar schemes (statements with second order free variables),
- the axiomatics of set theory.

4.2 Graduate level

On the graduate level the main focus is on one of the most important applications of automated theorem proving – software verification. At the same time, students are supposed to become competent Mizar users and be able to individually produce formalizations in various domains.

The "Software verification" course covers the following theoretical and practical aspects:

- various semantics of software description (operational, denotational, axiomatic),
- program correctness criteria,
- mathematical models of computers,
- practical verification of exemplary algorithms (sequential instructions, loops, jumps, recurrence),
- using the "describer" technique to generate proof conditions.

The objective of the second Mizar-based course on the graduate level is to enable students to choose a domain in which they could carry out formalization. The formalization may form a basis of one's MSc thesis. On this level the students are supposed to be proficient Mizar users and be trained enough to develop themselves new contributions to the Mizar Mathematical Library. The syllabus topics of the classes comprise:

- the formal theory of mathematical proofs in connection with computer proof-checking systems,
- structuring and managing databases of formalized proofs,
- practical usage of discussed mechanisms based on selected fields of mathematics.

5 Getting to know how to write Mizar proofs

The methodology adopted for the realization of the above mentioned courses must reflect the gradual way of getting to know the system from its very basics till be-

coming an independent competent user. Therefore the methodology changes slightly with the advance of formalized material. There are, however, certain assumptions we consider crucial from the didactic point of view.

First of all, the amount of Mizar notions introduced at each stage should be reduced to minimum, i.e. the smallest possibly subset of Mizar which allows completing a certain task. For instance, the first session of the introductory course in logic is devoted solely to propositional and predicate calculus – we have to bear in mind that our students at the same time must also learn the standard mathematics notation of these notions and the way of installing, running, and interacting with the Mizar system. Therefore it is extremely convenient to prepare ready-made dedicated local environments which spare the students the intricacies of coping with much technical detail completely unnecessary at that stage. The first environment should therefore include only predicates with various arity and no particular denotation, as it is all one needs to practice the use of logical connectives, quantifiers and the rules of first-order logic. Similarly, before writing complete proofs of e.g. Boolean properties of sets, students should start with the so-called formal proof sketches (compare [9]), to have the syntax and proof structure correct first.

In general, we believe the whole instruction on the undergraduate level should be based on "incremented" local environments instead of using the full system with the standard mathematical library (which would require certain extra effort of searching the whole library). Teaching the interaction with the full system should be an integral part of the graduate-level courses.

The high-level features of the Mizar language and system should not be introduced too early – it seems much more proper to first show students how to prove things in an elementary way, and only when they master it, allow for a reflection when they are presented a more "elegant" high-level way to do the same. On the one hand, this can be done on the level of the language's "syntactic sugar" expressions like `then`, `hence`, `thesis` which should be introduced when students already know how to write proofs using "strict" Mizar. On the other hand, there are many system's capabilities normally available to users which allow to write shorter proofs, like the automatic definition expansion, the use of implicit general quantifiers, the use of semantic correlates rules for thesis elimination, the forward/backward proof distinction and so on – their introduction should be postponed too.

The acquisition of the formal Mizar language should also be split into two layers – passive and active. Only passive knowledge (the ability to read with general understanding) of certain constructs is enough at first. Taking as an example the whole part of Mizar's grammar which concerns definitions, we note that although the students should be able to read and refer to the definitions presented in Mizar abstracts (interface information only with all justifications removed), writing the student's own definitions (active knowledge required) is only the part of the second graduate-level course. In practice, this process resembles the way in which students master the use of programming languages – the major part of writing a Mizar article consists of using pre-existing theorems and definitions, just like the major part of writing a computer program consists of referencing available library procedures/functions.

6 Conclusions

Implementing the usage of the Mizar proof-assistant in the obligatory curriculum for CS students at the University of Białystok proved feasible. The simple methodology adopted with a combination of interdependent undergraduate- and graduate-level courses make the teaching process not harder than that of a popular programming language. In the perspective of few years we will be able to fully assess the results of this project by the number of MSc theses based on new contributions to the Mizar Mathematical Library. The choice to focus on the instruction of new CS adepts rather than mathematicians seems reasonable and prospective in order to facilitate the important use of formal methods is software verification and specification.

References

[1] Bancerek, G., *Development of the Theory of Continuous Lattices in* Mizar, In M. Kerber and M. Kohlhase (Eds.), "Symbolic Computation and Automated Reasoning", proceedings of the CALCULEMUS 2000 Symposium, 65–80.

[2] Geuvers, H. et al., *The Algebraic Hierarchy of the FTA Project*, In S. Linton and R. Sebastiani (Eds.) "Proceedings of the CALCULEMUS 2001 Symposium", Siena, 2001, 13–27.

[3] Lamport, L., *How to Write a Proof*, American Mathematical Monthly **102(7)** (1993), 600–608, URL: http://research.microsoft.com/users/lamport/pubs/lamport-how-to-write.pdf.

[4] Matuszewski, R. and P. Rudnicki, *MIZAR: the First 30 Years*, Mechanized Mathematics and Its Applications, **4(1)** (2005), 3–24.

[5] Naumowicz, A., *An Example of Formalizing Recent Mathematical Results in MIZAR*, In C. Benzmüller (Ed.) "Towards Computer Aided Mathematics", Journal of Applied Logic **4(4)** (2006), 396–413.

[6] Retel, K. and A. Zalewska, *MIZAR as a Tool for Teaching Mathematics*, Mechanized Mathematics and Its Applications **4(1)** (2005), 35–42.

[7] Rudnicki, P. and A. Trybulec, *Mathematical Knowledge Management in MIZAR*, In B. Buchberger and O. Caprotti (Eds.) "Proceedings of the First International Workshop on Mathematical Knowledge Management: MKM 2001", URL: http://www.emis.de/proceedings/MKM2001/rudnicki.ps.

[8] Trybulec, A. and P. Rudnicki, *Using Mizar in Computer Aided Instruction of Mathematics*, Norwegian-French Conference of CAI in Mathematics, Oslo, 1993, URL: http://mizar.org/project/oslo.ps.

[9] Wiedijk, F., *Formal Proof Sketches*, In W. Fokkink and J. van de Pol (Eds.) "7th Dutch Proof Tools Day, Program + Proceedings", CWI, Amsterdam, 2003, URL: http://www.cs.ru.nl/~freek/notes/sketches.pdf.

[10] Wiedijk, F., *Mizar: An Impression*, URL: http://www.cs.ru.nl/~freek/mizar/mizarintro.pdf.

[11] Wiedijk, F. (ed.), "The Seventeen Provers of the World" (foreword by Dana S. Scott), LNAI 3600, 2006.

Papuq: a Coq assistant ⋆

Jakub Sakowicz[1] and Jacek Chrząszcz[2]

Institute of Informatics
Warsaw University
Poland

Abstract

We describe an extension to CoqIDE called Papuq, targeted at students learning the basics of mathematical reasoning. The extension tries to bridge the gap between natural language used to teach proofs during the university course and the artificial language of Coq proofs. We believe it will give the students the possibility to practice writing proofs by themselves and at the same time to learn writing precise proofs in natural language.

Keywords: Coq, set theory, computer-aided proofs

1 Introduction

First year computer science students at Warsaw University have to complete a course named "Introduction to Set Theory". During this course, students learn the basic mathematical vocabulary and, what is more important, learn what a correct mathematical reasoning is. This latter part is notoriously difficult as the students are not used to apply rigor to the natural language they have to use in pen and paper proofs. Besides, it is very difficult for them to practice writing proofs as they are never sure whether a proof is correct or not and their tutor can assess about one proof per week, so this is not much.

Therefore we would like to propose to teach the very beginning of the set theory course using Coq. Unfortunately the existing Coq interface, CoqIDE, requires the knowledge of Coq commands and proof tactics. Of course learning those without the knowledge what a proof is, is simply impossible. For these reason the first author made an extension to CoqIDE, called Papuq, which in a separate window gives the user several hints how to continue the proof. The hints are written in natural language and accepting a given hint produces a real Coq tactic. At the

⋆ Partly supported by Polish Government Grant 3 T11C 002 27, and by the EU Coordination Action 510996 "Types for Proofs and Programs".

[1] Email: j.sakowicz@students.mimuw.edu.pl

[2] Email: chrzaszcz@mimuw.edu.pl

same time, another part of the window presents the last step of a proof, also in the natural language. This way the student sees a correct wording of every proof step and learns to write correct proofs himself.

2 Practical issues of set theory

In set theory we define everything from scratch (i.e. empty set and braces), but then the notions are used by mathematicians without thinking about the actual set theoretic definition. Moreover perfect equations from set theory point of view seem stupid. For example, since the standard encoding of a tuple $\langle a, b\rangle$ is the set $\{\{a\}, \{a, b\}\}$, we get:

$$\langle a, a\rangle = \{\{a\}\}$$

which "does not type check".

2.1 Implementation and abstraction

For a mathematician, a pair is something that can be constructed from two arbitrary mathematical objects, and once the pair is constructed, these things can be extracted from the pair in the correct order. Moreover, the construction is deterministic, i.e. two pairs are equal if and only if they are constructed from equal elements.

What mathematicians are interested in, are rather extensional properties of defined notions or their specification rather than the set theoretic encoding, which is merely an implementation.

2.2 Does everything need to be a set?

As the first-year course we started the paper with had once been called "Introduction to Mathematics", we think that resigning from teaching students how to encode a pair as a set is not a big problem for their mathematical education.

In this light, teaching students with Coq, where you cannot compare a pair with, say, a set (type) of natural numbers, is quite a viable choice.

2.3 Students' problems

Below we mention a couple of typical problems we encounter while correcting student's homework and exams.

- Confusing assumptions and conclusions.

  Since mathematical proofs can employ either a forward reasoning or a backward reasoning, it is "natural" to mix the two techniques leading to "interesting results". For example: Now we prove that f is injective. A function f is injective if for all x, y we have $f(x) = f(y)$ implies $x = y$. Let us consider arbitrary x, y such that $f(x) = f(y)$. Since this implies $x = y$, the function f is injective.
- Quantifier problems.

  Many students find it difficult to correctly handle quantifiers, especially the existential one. This comes from the confusion between the natural language

where the existence can be a predicate of its own and logic, where it only is a quantifier. When a students wants to create a formula saying that the set A has at most two elements, the first "logical" formula that comes to his mind is "if there exist three elements in the set A, two of them must be equal". It is really hard to translate this to a universally quantified formula.

It would be very good to insist on teaching the pattern of using and proving both the existentially and universally quantified formulae. Using an automated tool like Coq can be very useful.

- Unfolding problems.

  Sometimes students "forget" to unfold definitions, even if they have access to course-notes. For example given "$x \in r \subseteq A \times B$" they may not come to the next step of the reasoning which is "so x is of the form $\langle a, b \rangle$".

- Bad understanding of definitions.

  It is often the case, that students asked to prove injectivity of a function like $\phi : P(\mathbb{N}) \to (P(\mathbb{N}) \to P(\mathbb{N}))$ defined by $\phi(a)(b) = a \cap b$ give a solution like this: "Let a and a' be subsets of $\mathbb{N}$. Let us consider $\phi(a)$ and show that it is an injective function."

- Negating logical sentences.

  Although de Morgan laws are theoretically known to students, if the logical sentence is too complex, students easily get lost. For example, asked about a complement of a set of equivalence relations with finitely many equivalence classes in the set of all binary relations over natural numbers, students often answer that this is a set of equivalence relations with infinitely many classes.

3 Naive type theory

In order to teach mathematical reasoning to students using the Coq proof assistant, one must provide them with some working environment. In this section we describe the theoretical part of this environment. Of course set theory can be encoded in Coq (see [17] and [18]), but the encoding is far from being convenient for teaching. Instead, in accordance with [10], we propose a simple type theory that we find to be close to what mathematicians use in their everyday work. Precise description of this type theory and its (partial) realization as a pure type system can be found in [10]. Here, we give only an informal description and a straightforward encoding in Coq (see Section 4). Following [10] and [4], we use the term "Naive type theory".

Primitive notions of this theory are types and objects. All object have a type (exactly one). Types are not objects. The fact that the object a has type T will be written as usual as $a : T$. Equality is also a primitive notion. Two object are equal if they are indistinguishable, i.e. one can replace another in any context.

The basis of NTT is classical logic. In this paper first order logic is enough.

3.1 Types

There is a number of predefined types which are given together with certain axioms:

- `Prop` — the type of all formulae with constants `True`, `I:True` and `False` which

satisfies:
`forall P : Prop, False -> P`

- `empty` — empty type, satisfies:
  `forall x:empty, False`
- `unit` — singleton type, with element `tt:unit`, satisfies:
  `forall x:unit, x=tt`
- `bool` — two element type, with elements `true` and `false`, satisfies:
  `~ true = false`
  `forall x : bool, x=true \/ x=false`
- `nat` — the type of natural numbers, with constant `0` and successor function `S:nat->nat`, satisfying the following properties:
```
forall n:nat, n = 0 \/ exists m:nat, n = S(m)
forall n m:nat, S n = S m -> n = m
forall P:nat->Prop, P 0 -> (forall n:nat, P n->P (S n)) ->
                      forall n:nat, P n
forall n:nat, ~ S n = 0
```

Complex types can be built using type constructors: `->` (function type), `+` (disjoint sum of two types), `*` (Cartesian product of two types). Objects of these types can be made by lambda terms or constructors:

```
in_left : forall A B:Type, A -> A+B
in_right : forall A B:Type, B -> A+B
pair : forall A B:Type, A -> B -> A*B
```

We assume that equality of functions is extensional, i.e:
`forall f g :A->B, (forall a:A, f a = g a) -> f=g`
and that there are projections `fst:A*B->A` and `snd:A*B->B`, satisfying

```
forall a:A, forall b:B, fst (pair a b) = a
forall a:A, forall b:B, snd (pair a b) = b
```

3.2 Predicates

For all types `T`, *predicates* on `T` are all expressions of type `T -> Prop`, i.e. `Predicate T := T -> Prop`. They correspond to set theoretic subsets of `T`. In our theory we do not have the notion of subtype. In order to simulate reasoning about inclusion of sets, we define the notion of belonging as in "t belongs to a predicate A" (or "t is an element of A"), written `x IN A`, as follows:

```
forall T:Type, forall A:Predicate T, forall t:T, t IN A <-> A t
```

We define the notions of a sum, difference, intersection and inclusion of predicates in the standard way as well as the empty and the full predicate. Like for functions, equality for predicates is extensional, i.e.

```
forall T:Type, forall A B:Predicate T,
  (forall t:T, t IN A <-> t IN B) -> A = B
```

3.3 Functions

For functions (i.e. objects of type `X -> Y`) we define a number of standard notions:

- image
- injectivity
- surjectivity

In type theory, functions work for all elements of the type, and not for those belonging to a predicate. We are free, however, to consider properties of functions limited to a certain subset (predicate) of the domain:

```
MonomorphismPred (U V : Type) (f : U -> V) (A : Predicate U) :=
  forall x y : U, x IN A /\ y IN A -> f x = f y -> x = y
EpimorphismPred (U V : Type) (f : U -> V) (A : Predicate U) :=
  forall y : V, exists x : U, x in A /\ f x = y
```

Another property is the equality of functions on a given predicate. Note that a suitable replacement property is not expressible: indeed, we would have to quantify over formulas applying the function only to arguments from the given predicate. Of course for many particular predicates, the replacement property is provable, but in spite of that equality of functions over a given predicate is much less convenient than regular equality of functions.

3.4 Binary relations

For simplicity we limit ourselves to binary relations. Similarly to predicates, they are defined as:

```
Relation (A : Type) := A -> A -> Prop
```

Simple properties of binary relations, like reflexivity, transitivity, symmetry, antisymmetry can be defined in the standard way.

Note that in standard set theory, functions are just a special kind of relations. Here, they are a primitive notion. Still, we want to use the fact that if a relation is functional over the whole domain, there is also a corresponding function. In other words, we admit the principle of description:

```
(forall x, exists y, R x y) /\
(forall x y z, R x y -> R x z -> y = z) ->
  exists f : A->A, forall x, R x (f x)
```

3.5 Others

Apart from the basic notions of set theory, we consider also more advanced topics, such as cardinality theory. We define the notions of cardinality equality and inequality for predicates. Even though these notions become a bit complex in this setting, basic theorems from the cardinality theory can be proved easily.

4 Naive type theory in Coq

Almost all of the theory presented in the previous section can be encoded in Coq using standard inductive definitions and their properties and the `Type` hierarchy to encode predicate types (i.e. powersets). Only four axioms have to be added: excluded middle, extensionality of predicate equality, extensionality of functional equality and the principle of description.

By [9], when `Set` is predicative, these axioms are considered to be consistent. Further discussion of NTT and its consistency without the `Type` hierarchy can be found in [10].

Let us see a couple of example proofs in NTT in Coq and in the natural language. The natural language proofs are hand-written precise proofs that one would like to hear from first-year students proving the given propositions.

The first task is to show that $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$. Note that tactics `split`, `left`, `right`, `destruct` can be used to set operations without unfolding them. For example `split` applied to a goal $x\ IN\ A \cap B$ gives us two simple goals $x\ IN\ A$ and $x\ IN\ B$.

```
Variable U : Type.
Variable A B C : Predicate U.

Goal (A u B)\C = (A\C) u (B\C).
```

`apply ax_pred_ext.` `intro.`	In order to prove equality of two predicates, one must show for all x that belonging to one of them is equivalent to the other (the axiom `ax_pred_ext`).
`split.`	Equivalence can be showed by proving two implications.

`intro.`
`destruct H.`
`destruct H.`
	Let us assume that $x\ IN\ (A \cup B) \setminus C$. By definition of subtraction we get $x\ IN\ A \cup B$ and the negation of $x\ IN\ C$. By definition of sum, we have two cases.

`left.` `split; trivial.`	In the first case we have $x\ IN\ A$. The conclusion holds, because $x\ IN\ (A \setminus C)$ (the left part of the sum).
`right.` `split; trivial.`	In the second case we have $x\ IN\ B$. The conclusion holds, because $x\ IN\ (B \setminus C)$ (the right part of the sum).

`intro.`
`split.`
`destruct H.`
	In the opposite direction, let us assume that $x\ IN\ (A \setminus C) \cup (B \setminus C)$. By definition of subtraction, we must show $x\ IN\ A \cup B$ and the negation of $x\ IN\ C$. By assumption, we have two cases.

```
left.
destruct H.
trivial.
```

In the first case x *IN* A so x *IN* $A \cup B$.

```
right.
destruct H.
trivial.
```

In the second case x *IN* B so x *IN* $A \cup B$.

```
destruct H;
destruct H;
trivial.
Qed.
```

In both cases the negation of x *IN* C holds.

Now, let us show an example theorem about functions: a function f is an involution (i.e. $f \circ f = f$) if and only if f is an identity on its image. In this proof we used tactics `change` i `rewrite`.

```
Variable U : Type.
Variable f : U -> U.

Goal f o f = f
  <->  forall x : U, x IN (Image f (Whole U)) -> f x = x.
```

```
split.
intro.
intro x.
intro.
```

One has to prove two implications. Let us assume that f is an involution, i.e. $f \circ f = f$. Let us take x belonging to the image of f.

```
unfold Image in H0.
destruct H0.
destruct H0.
```

By definition there is y *IN* A such that $f(y) = x$.

```
rewrite H1.
change (f(f x)) with ((f o f) x).
rewrite H.
trivial.
```

Since $f(x) = f(f(y))$ we have $f(x) = f(f(y)) = f(y) = x$.

```
intro.
apply ax_fun_ext.
```

To prove the implication in the other direction, let us assume that f limited to its image is an identity. To prove equality of functions, we prove their equality for all arguments.

```
intro x.
unfold Comp.
apply H.
```

Let x *IN* A. The conclusion $f(f(x)) = f(x)$ comes from the assumption that f is an identity on its image.

```
unfold Image.
unfold Whole.
exists x.
split.
compute.
trivial.
trivial.
Qed.
```

Now we have to prove that $f(x)$ belongs to the image of f. It is the case, because there exists an element of type A, belonging to `Whole A`, such that its value is $f(x)$. Of course it is x.

The last proof that is worth showing here is a proof by induction.

```
Goal forall m k : nat, m + k = m -> k = 0.
```

```
induction m.
```

Proof by induction on m.

```
intros.
compute in H.
exact H.
```

Base case for $m = 0$. By definition of addition, $0 + k = k$, so $k = 0$.

```
intros.
simpl in H.
```

Now let us assume that $m+k = m$ implies $k = 0$ and show the same for $S(m)$. We have $S(m) + k = S(m)$. By definition of addition $S(m) + k = S(m + k)$ and hence our assumption is equivalent to $S(m+k) = S(m)$.

```
injection H.
apply IHm.
Qed.
```

Since constructors are injective, we get $m + k = m$ and $k = 0$ by induction hypothesis, which finishes the proof.

Of course just looking at the scripts does not tell us what the proofs are like. Especially for students, a more verbose way of using Coq is necessary.

5 Helping the student

The existing Coq interface, CoqIDE, provides much help in writing and correcting proof scripts, but still relies on the knowledge of tactics by the user. This section describes the facilities that we implemented in order to help users write simple proofs (almost) without knowing tactics. It is targeted at first-year students learning the basics of mathematics.

The implemented extension, called Papuq, presents the user with a choice of proof steps, described in natural language (i.e. in a language which is natural to mathematicians).

5.1 Hints: tactics for 1st order logic

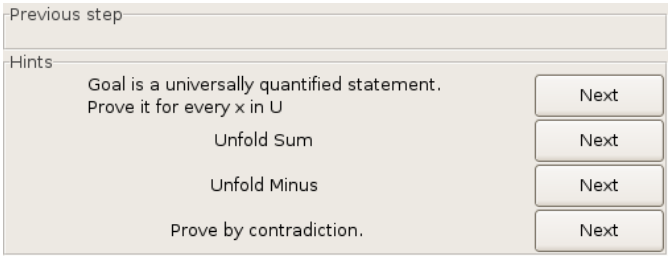
One of the students' problems described in section 2.3 was the use of first-order logic. Since proof steps made while proving first-order statements are largely routine, it

is natural to suggest the next step based on the syntactical structure of the goal.

Papuq offers this functionality in an additional window, divided into two parts. The lower part presents possible next steps in natural language and the upper part presents the step made recently. For example, when proving the distributivity of sum over a difference of predicates in the following state:

```
1 subgoal
U : Type
A : Predicate U
B : Predicate U
C : Predicate U
-----------------------------------(1/1)
forall x:U, x IN ((A u B)\C) <-> x IN ((A\C) u (B\C))
```

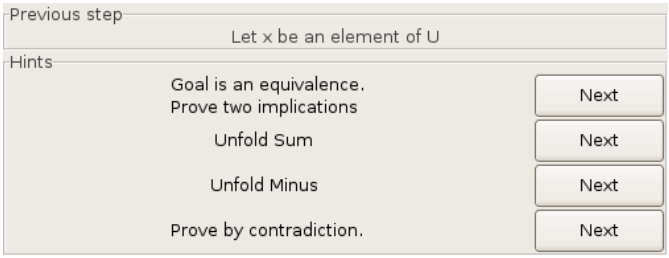
we will get the following suggestion



The first suggestion is generated by the first-order logic hint module. Pressing the "Next" button pastes the "`intro.`" string at the cursor position of the script window and executes the tactic. After this, the state is extended by an additional assumption:

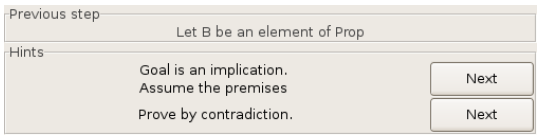
```
x : U
-----------------------------------(1/1)
x IN ((A u B)\C) <-> x IN ((A\C) u (B\C))
```

and the proof wizard window contains a natural language description of the last step and the next hint:

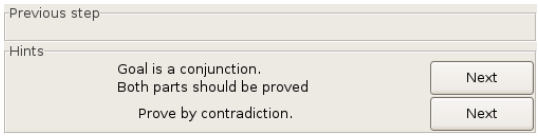


Papuq handles all first-order operators. Here are the generated suggestions for other operators:

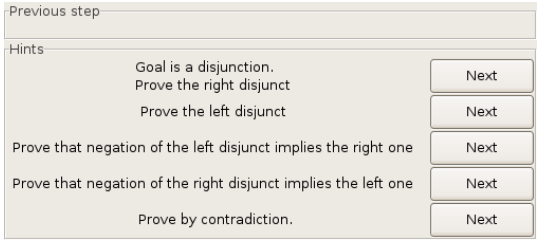
```
A : Prop
B : Prop
--------------------------(1/1)
A -> B
```



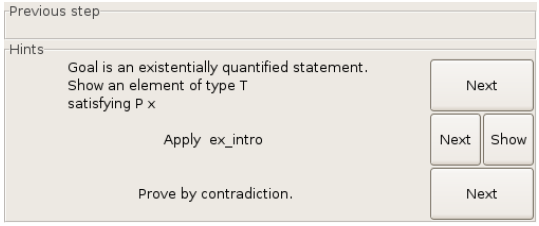
```
A : Prop
B : Prop
-------------------------(1/1)
A /\ B
```




```
A : Prop
B : Prop
-------------------------(1/1)
A \/ B
```




```
T : Type
P : T -> Prop
-------------------------(1/1)
exists x:T, P x
```



It is important to note the possibilities for disjunction. Apart from the intuitionistic proof by selecting and proving either the right or the left disjunct, there is also a possibility to use a classical way of reasoning by assuming the negation of one disjunct and proving the other one (because classically `(~A -> B) -> A \/ B`).

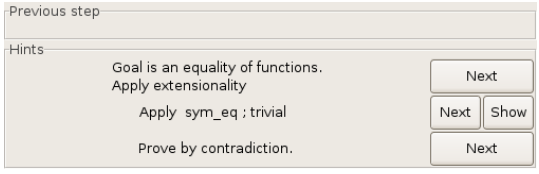
The hint for the existential quantifier needs a comment too. We use the tactic `eapply ex_intro`, which leaves the goal with a "hole". In fact, the user should rather use the `exists` tactic and give the right element explicitly.

5.2 Hints: axioms for equality of predicates and functions

Another way Papuq can help students is to advise them to use an axiom at the right moment. For example, in order to prove the equality of predicates or functions the canonical way to go (in our naive type theory) is to use the appropriate extensionality axiom.

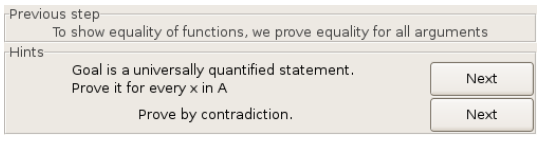
Let us show a small fragment of a proof that the uncurrying operation `uncury:(A->B->C)->(A*B->C)` is a bijection.

```
f : A -> B -> C
g : A -> B -> C
H : uncurry f = uncurry g
-------------------------(1/2)
f = g
```



After using extensionality, we are informed of the next step:

```
f : A -> B -> C
g : A -> B -> C
H : uncurry f = uncurry g
------------------------(1/2)
forall x : A, f x = g x
```



And when we agree to the introduction, we are advised to use extensionality again.

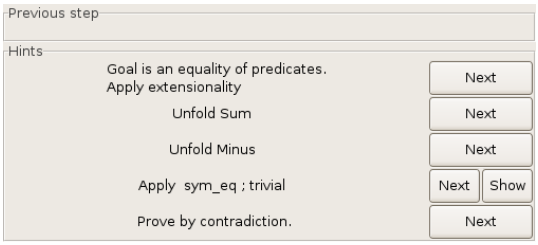
Showing this hint to students is important for at least two reasons. First of all, students are used to prove an equality by simplifying both sides of it until they reach identical expressions. Here they have to learn a completely different strategy. Second, as shown by our practice, students, even those who know that extensionality has to be used, sometimes confuse the conclusion of the problem with parts of extensionality axiom.

5.3 Hints from the auto database

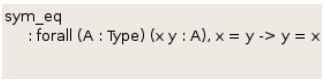
Since Coq already has a similar mechanism to help the user in remembering useful lemmas through their automatic application with the `auto` tactic, we included the list of applicable lemmas from the `auto` database in the list of suggestions presented in the Wizard Window. However, since we check that all listed lemmas are indeed applicable (by testing whether `progress tactic` would succeed) we list less lemmas than the command `Print Hint` would. Moreover, the interface gives the user the possibility to immediately see the lemma by clicking the "Show" button.

```
U : Type
A : Predicate U
B : Predicate U
C : Predicate U
------------------------------(1/1)
(A u B)\C = (A\C) u (B\C)
```

In the list below, the second element corresponds to the hint found in the auto database:



After clicking the "Show" button, we get the details of the lemma:



5.4 Simplified use of assumption

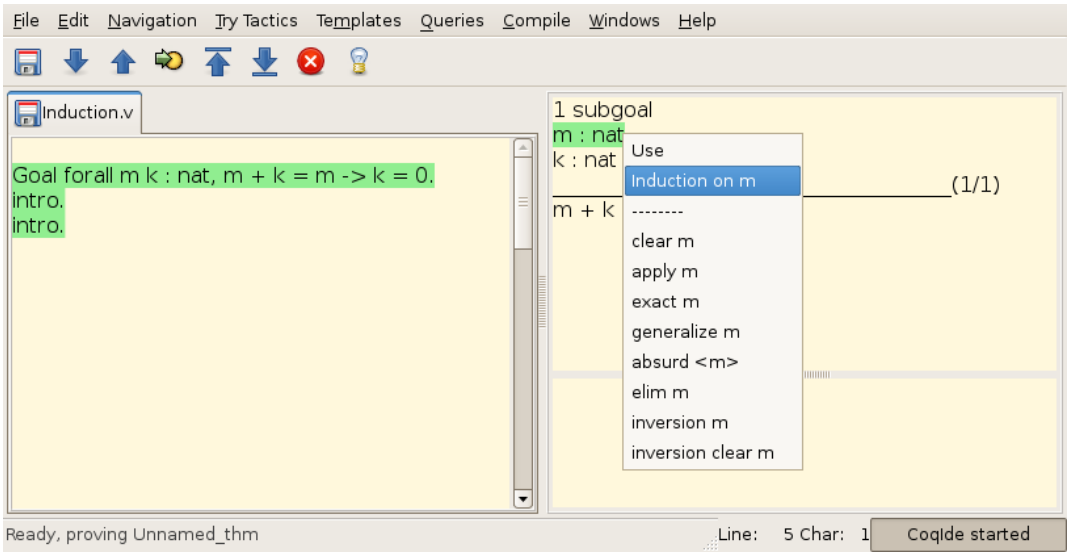
A common element of mathematical proofs are sentences "from the assumption we get ...". Usually one does not care how precisely the assumption is used. In Coq, however, there are many ways to do that. An equality can for example be used either to replace the left hand side by the right hand side (or vice versa — tactics `rewrite` and `rewrite <-`) or, if it is an equality of terms made from inductive constructors, to extract a simpler equality from the existing one (`injection`). A conjunction or other logical formula starting with an inductively defined connective

can be used through the appropriate elimination rule (`destruct`). An implication can be used only if its conclusion matches the goal and one can prove the premises (`apply`).

For a beginner, having to remember the names or at least meaning of all these tactics is quite problematic. Therefore we extended the context menu of the CoqIDE goal panel with the generic "Use" command, which selects the appropriate tactic. For equalities one can also choose "Rewrite" and "Rewrite backwards" commands and for elements of inductive types "Induction on H" to start an inductive proof. Also, the "Simplify" command, standing for "simpl in H" can appear in the context menu.

Since the appearance of a given command in the context menu is determined by the applicability of suitable tactics, one can have "Induction on H" entry in the context menu of an equality. Although this may seem rather counter-intuitive, in fact the `induction H` tactic can be successfully applied to an equality.

Let us see the example of a simple arithmetic formula, where the proof by induction is suggested by Papuq context menu:

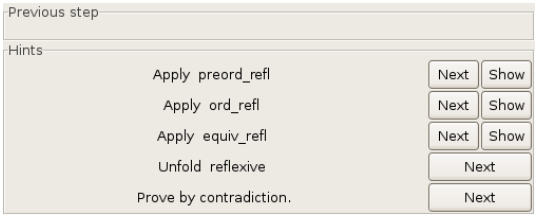


5.5 Other hints

Besides the hints mentioned above, Papuq offers:

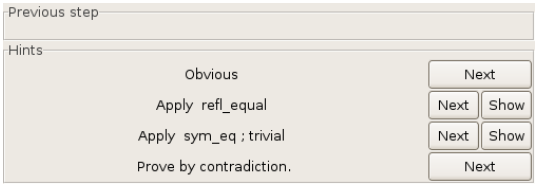
- unfolding of definitions — if the auto database suggests the `unfold` tactic, e.g:

```
Triangle : Type
Number : Type
R : relation Triangle
Area : Triangle -> Number
---------------------------(1/1)
reflexive R
```



- marking the "Obvious" goals (i.e. solvable by `trivial`)

```
Triangle : Type
Number : Type
R : relation Triangle
Area : Triangle -> Number
x : Triangle
---------------------------(1/1)
Area x = Area x
```



- proposing a proof by contradiction — this hint is alway available unless the goal is `False`; it uses the classical double negation proof: assumes the negation of the goal and the new goal becomes `False`.

It is important to note here that all examples given in Section 4 can be proved by clicking the hints from Papuq, apart from the second one where one tactic, `change (f (f x0)) with ((f o f) x0)`, has to be given by hand. Other slight disadvantage is the use of the tactic `eapply ex_intro` generated by Papuq in two of the examples, where the appearing existential variables are automatically inferred and the student can get a bit confused.

6 Papuq documentation

Papuq can be downloaded from http://www.mimuw.edu.pl/~chrzaszcz/Papuq. It is available as a patch to Coq sources, version 8.0pl4. After patching, compiling and installing, one has to run `coqide` and select *Windows → Show Wizard Window*. The window will react to the state of CoqIDE.

Apart from the patch, several Coq theory files are available: `CoqTypesTheory.v`, containing the encoding of NTT and a couple of files with exercises and their solutions.

The patch to CoqIDE contains slight modifications of existing code and the building procedure, and the implementation of the Papuq functionality. The extension is done in such a way that writing the set of hints in a new language is very simple. The whole patch has a little over 2000 lines.

The most important problem with writing the extension was of course very sparse documentation both of Coq and especially CoqIDE.

The modifications of CoqIDE itself were kept to the minimum. The only new implemented features are calls to event handlers, inserted in various places of the CoqIDE code.

6.1 CoqIDE modifications

We added references to side-effect functions in a few places in CoqIDE, together with registration functions. The references are empty in the beginning, but as the Papuq modules are implemented, they register proper handlers that get called every time a given event occurs.

The handlers we added are the following:

- `external_goal_handler : unit->unit` — called after the script fragment is

processed, before the result is displayed to the user,

- `external_undo_handler : unit->unit` — called when one or several steps of the script are undone, a scripting is moved to a new window or the processing is abandoned,
- `external_redo_handler : unit->unit` — called when one or many steps of the script are done without using Papuq
- `external_wizard_start : unit->unit` — action connected to the new menu command *Windows → Show Wizard Window*,
- `external_hyp_menu_handler : string->string->(string*string) list` — called when a context menu for a given hypothesis is generated; the result of this function is added to the context menu

Apart from introducing the above event handlers, we also exported several CoqIDE functions (by extending the `.mli` files).

Another change in CoqIDE, somewhat orthogonal to the implementation of Papuq functionalities, was creating a separate thread for CoqIDE. Because of that CoqIDE can be run on top of the Coq toplevel which, as it may embed OCaml toplevel as well, is especially useful for debugging purposes and probably was crucial to the success of Papuq.

To summarize, not more than 40 lines of CoqIDE code were changed.

6.2 Papuq functionality

The extension is implemented in six modules. These are:

- *Localization* — contains the *Resource* class used by the user interface. All strings are in this class or are constructed by its methods. No other modules contain or construct strings. Therefore a translator of Papuq to a new language has to provide only the implementation of the *Localization* module. Currently, the English and the Polish versions are available.
- *Teachcfg* — contains global configuration of Papuq. Currently this is a flag deciding whether hints should be tested for applicability, and constants setting the default window sizes.
- *Teachdebug* — this module contains functions to debug Papuq and test CoqIDE. They are not used during normal operation.
- *Teachutils* — implements data structures, manipulating functions and tools to support dialog with the Coq API such as printing functions, access to the current goal, number of subgoals, operations on the auto database, etc. The idea was to separate the algorithm to generate hints from the details of the access to the information kept by Coq.
- *Teachhint* — the heart of Papuq. Implements the algorithms generating hints. In case Papuq is further extended, new kinds of hints should be implemented here.
- *Teaching* — user interface of Papuq. It contains the class *Wizard* representing the Wizard Window. This module also registers event handlers from the class *Wizard* in CoqIDE.

7 Summary and Related Work

The implemented extension is a useful addition to CoqIDE, relieving the beginner of the difficult task of remembering Coq tactic names. It was made with the needs of first year students in mind, so the other principal task of the extension is to teach users the basics of mathematical reasoning, i.e. systematic use of definitions, logical rules and axioms. Of course the current version of the extension is by no means a finished general purpose tool. It is made for the naive Coq type theory that we prepared based on [10], but it can be extended to other theory files if needed.

The tool itself can be much improved, for example, the problematic introduction rule for existential quantifier. The natural language explanations of the "Previous step" could be given not only for steps made by clicking the Papuq window, but also for those made by clicking a hypothesis option or best for all ways of entering a tactic. Once this is accomplished, it would also be useful to present not only the latest step, but all steps done from the beginning of the proof with some indentations to show the proof structure. For now, we present the preliminary version of the tool and we hope to develop it in the future.

In general, even though the tactic language is very convenient for quick writing of proofs by an experienced user, it constitutes a big problem for beginners and the proofs written using the tactic language are completely unreadable. There is an ongoing work to come up with a novel, more declarative proof mode for Coq [8,5], where the user tells the machine the intermediate proof steps rather than the instruction what it should do. The experimental declarative proof language DPL by Pierre Corbineau has been included in the most recent version 8.1 of Coq.

There is also a number of works aiming at printing Coq proofs in natural language. They started from [7,6] and continued in the HELM and MoWGLI projects [1] as one of the results of the complex infrastructure to store, search and render large bodies of formalized mathematics. Other tools, like those based on the TeXmacs [15] editor, provide the possibility to nicely render mathematical formulae and interleave Coq proof script with human written proof [2,11]. But all these projects are targeted at experienced Coq users or experienced mathematicians.

More student oriented approach was taken by developers of other formal mathematics tools. For a few years now the Mizar system [13], which also has a proof rendering mechanism [12], is used to teach the foundations of mathematics to students at the University of Białystok. Similarly, PhoX [14], which also has its natural language presentation mechanism [16] is used for teaching at the University of Savoie. It also includes an easy user interface allowing a point-and-click proving.

The idea we implemented of a relatively independent "agent" suggesting next proof step to the user, have been thoroughly studied by the authors of the Ω-Ants system [3]. Unlike Papuq where the "agent" proposes almost only basic proof steps, the agents of the Ω-Ants system can be arbitrary complex proving tools. Indeed, the principal goal of Ω-Ants is not to teach students, but to integrate and parallelize various proof tools in order to build an efficient hybrid system.

Acknowledgements. We would like to thank Paweł Urzyczyn for encouraging us to do this work. We would also like to thank the anonymous referees for their helpful comments and suggestions.

References

[1] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. HELM and the semantic Math-Web. *Lecture Notes in Computer Science*, 2152:59–74, 2001.

[2] Philippe Audebaud and Laurence Rideau. TeXmacs as authoring tool for formal developments. *Electr. Notes Theor. Comput. Sci.*, 103:27–48, 2004.

[3] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[4] R.L Constable. Naive computational type theory. In H. Schwichtenberg and R. Steinbruggen, editors, *Proof and System-Reliability*, pages 213–259. Kluwer Academic Press, 2002.

[5] P. Corbineau. A declarative proof language for Coq, 2006. http://www.cs.ru.nl/~corbineau/dpl/index.html.

[6] Y. Coscoy. *Explication textuelles de preuves pour le calcul des constructions inductives*. Thèse d'université, Université de Nice-Sophia-Antipolis, September 2000.

[7] Y. Coscoy, G. Kahn, and L. Thery. Extracting text from proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proc. Second International Conference on Typed Lambda Calculi and Applications, Edinburgh, UK*, volume 902, pages 109–123, 1995.

[8] Mariusz Giero and Freek Wiedijk. MMode, a Mizar mode for the proof assistant Coq. Technical Report NIII-R0333, University of Nijmegen, 2003.

[9] Hugo Herbelin, Florent Kirchner, Benjamin Monate, and Julien Narboux. Coq version 8.0 for the clueless, sect. 5.2. http://coq.inria.fr/doc/faq.html#htoc37.

[10] Agnieszka Kozubek and Paweł Urzyczyn. In the search of a naive type theory. These proceedings, 2007.

[11] H. Geuvers L. Mamane. A document-oriented Coq plugin for TeXmacs. In *Mathematical User-Interfaces Workshop, St Anne's Manor, Workingham, United Kingdom*, Aug 2006.

[12] R. Matuszewski. On natural language presentation of formal mathematical texts. *Studies in Logic, Grammar and Rhetoric*, 3(16), 1999.

[13] The Mizar system. http://mizar.uwb.edu.pl/.

[14] Christophe Raffalli and Paul Rozière. Phox. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 67–71. Springer, 2006.

[15] The GNU TeXmacs system. http://www.texmacs.org/.

[16] Patrick Thevenon. Validation of proofs using PhoX. *Electr. Notes Theor. Comput. Sci.*, 140:55–66, 2005.

[17] Benjamin Werner. An encoding of ZFC set theory in Coq, 1997. http://coq.inria.fr/contribs/zermelo-fraenkel.html.

[18] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–346. Springer, 1997.

Proofs for freshmen with Coqweb

Jérémy Blanc

Laboratoire Jean-Alexandre Dieudonné, Université de Nice-Sophia Antipolis, CNRS

J.P. Giacometti

INRIA Sophia Antipolis

André Hirschowitz

Laboratoire Jean-Alexandre Dieudonné, Université de Nice-Sophia Antipolis, CNRS

Loïc Pottier

INRIA Sophia Antipolis

Abstract

Coqweb is a web interface for Coq, primarily designed for teaching. It allows teachers to propose statements in sufficiently familiar form. Students are invited to prove these statements essentially by clicking. Hints can be given in natural language, in which case the interface checks that the student somehow understands the hints. We present here the main features of Coqweb, and show through examples how it has been used since 2004.

Keywords: Proof Assistant, Interactive Theorem Prover, Education

1 Introduction

Proof assistants are now mature in some sense: they can be used to build formal proofs of highly respectable mathematical results (the fundamental theorem of algebra [FTA], or the four-color theorem [Gonthier Werner 04] for example). Unfortunately, only experts can use them, and learning to use a powerful assistant like Coq needs days of training. How could these proof assistants help teachers to teach what a proof is and how to search for a proof? The Coqweb environment [Coqweb] (a free software developed in ocaml and php by J.P. Giacometti, A. Hirschowitz and L. Pottier) is designed for teaching freshmen. Accordingly its main features are:

- The language of statements is fairly close to the standard mathematical language (thus full of ambiguities).

- The language of proof (tactics) is also fairly close to the standard mathematical practice.
- Proofs are essentially performed by clicking.
- It can be coupled with Wims, so that teachers can include Coqweb exercises into their Wims class sheets.

During the past three years, the regular freshmen (in mathematics and computer sciences) at the University of Nice-Sophia-Antipolis have been proposed a small amount of proving activity with Coqweb, essentially through the WIMS server [Xiao 98]. After several experiments, Coqweb is now a wiki where one can develop mathematics as we do in a course or a book : informal text using Latex, formal definitions of mathematical objects in a language close to the usual mathematical language, statements and proofs of theorems, exercises. As its name suggests, Coqweb heavily relies on the Coq assistant proof, which is used to performing the proofs and verifying them, even if this is quite transparent for the student.

The originality of Coqweb could be that the major concern of its design is to be accepted by typical students and teachers. Accordingly, the traditional approach initiated by the CtCoq pioneering system [Bertot 99] is slightly revised. In Coqweb, we have two separate modes: in the teacher mode, the expert teacher declares definitions and other resources, while in the student mode, the student solves an exercise, or "reads" the proof of a theorem by following the hints of an informal proof. We have focussed our attention onto the student mode, since, just as in the Wims server, non-expert teachers may use resources developed by expert teachers, and do not need to use the teacher mode. On the other hand, the student mode relies on the teacher mode where are generated the statements to be proven as well as the resources to be invoked.

This paper is organized as follows. In section 2, we briefly relate and discuss our experience in teaching proofs with Coqweb. In sections 3 and 4, we describe the student mode and its tactics. In section 5, we describe the teacher mode. In section 6, we show a complete example of a mathematical development in Coqweb for the freshmen at the University of Nice-Sophia-Antipolis. Finally in section 7, we discuss related and future work. The description and the discussion of the implementation of Coqweb will be done elsewhere.

Acknowledgments

We thank Valérie Moreau-Villeger, Joachim Yaméogo and Xiao Gang who took care in various ways of the connection between Coqweb and Wims. We thank Marco Maggesi and Carlos Simpson for stimulating conversations. We thank the organizers of the second Wims conference [WIMS07] who invited us to present our views on proof-teaching with Coqweb. We thank the referees whose criticism led to significant improvements in our presentation. Finally, the third-named author wishes to thank his chairman Michel Merle, his dean Raymond Negrel and his colleague Paul Silici who made possible the experience reported here, at least until fall 06. For what happened then, see [BlogDuB].

2 Teaching proofs

2.1 An impossible task?

Proofs are not really taught at our universities. Teachers just show and explain real proofs and then invite students to understand and reproduce them. From this experience, students are expected to be able to produce new proofs without having ever been explained what a proof is, and while their basic logic needs serious consolidation. Some teachers are satisfied with the way they teach proofs, considering that students who work enough should understand (and the accompanying conclusion is too often that most students do not work enough. For more on this controversial line, see [BlogDuB]). But many others consider teaching proofs as an almost impossible task. Some researchers in Education confirm the latter position. We quote from [Epp 03] (p.8): "A likely consequence for mathematics instruction is that in order to learn a complex process such as proof and disproof, effective integration of new modes of thought with pre-existing contradictory modes is a major undertaking. It is not surprising that easy solutions have not yet been discovered".

2.2 Proof plans versus complete proofs

One of the reasons why proofs are not understood by students is that the traditional informal style for proofs gives just a plan from which the reader is supposed to reconstruct a complete picture. Think of a game of chess as reported in a newspaper: they just give the sequence of moves, in some minimal notation. Grandmasters will understand the game from this minimal information without using a chessboard. But the typical reader of the newspaper will need a chessboard in order to see the corresponding sequence of positions and understand what is going on, if not necessarily why. The sequence of positions enlightens the corresponding sequence of moves.

Just the same happens for informal proofs as well as for proof scripts: they consist basically in an enumeration of actions (often called tactics) developing the proof, and this enumeration will be enlightened if we simultaneously show the corresponding enumeration of states of the proof.

Four or five years ago, one of us tried to teach proofs by showing and maintaining the state of the proof on a blackboard. This ended with the conclusion that he was not able to do it efficiently, even for freshmen proofs. On the contrary, to show the complete proof (including actions and states) on the screen through the Coqweb interface is pretty satisfactory. Indeed, we did experiment with this at the University of Nice during the first course in calculus (fall 06). Of course for this to be possible, it was necessary that the pretty-print of statements proposed by Coqweb be sufficiently close to what the typical freshman could accept and understand. In particular Coqweb had to manage ambiguous formulas.

2.3 Procedural versus declarative proofs

One of the reasons why proofs are not taught is that most teachers have not been taught themselves what a proof is. Some of them have been given a course in logic, but the notion of proof given there has essentially nothing to do with a real proof. As

a matter of fact, when you ask a logician, like J.Y. Girard for instance [Girard 07], what logic is doing in order to fill the gap between the current notion of logical proof and the current practice of mathematicians, he confirms that this task is taken into account mainly (only?) by the theorem provers community. Indeed each theorem prover, in particular our favorite one, Coq, has its operational notion of proof, which is sufficiently precise, at least for freshmen, and reflects a corresponding practice. We believe that a precise notion of proof is a crucial ingredient when trying to teach proofs for freshmen.

The Curry-Howard paradigm teaches us that a proof is a λ-term, but this point of view, while totally formalized and extremely appealing for researchers, is definitely too abstract for our students. Researchers of the theorem provers community have considered procedural (as in Coq [Coq 06]) as well as declarative (as in Mizar [MIZAR]) notions of proof scripts (see e. g. [Autexier Sacerdoti Coen 06,Corbineau,Sacerdoti 06]), and more recently have started mixing the two styles [Wiedijk 04]. Procedural as well as declarative scripts are again some form of incomplete proofs in the sense that either the successive states or the successive proof steps have to be inferred. The choice between the procedural and declarative styles is not too meaningful for Coqweb because students are not supposed to write proofs but only to generate them by clicking. Thus an informal notion of proof mentioning altogether states (lists of goals) and transitions (performed by tactics) is suitable. We have observed that such a notion of proof is well accepted by students.

2.4 Formal versus informal proofs

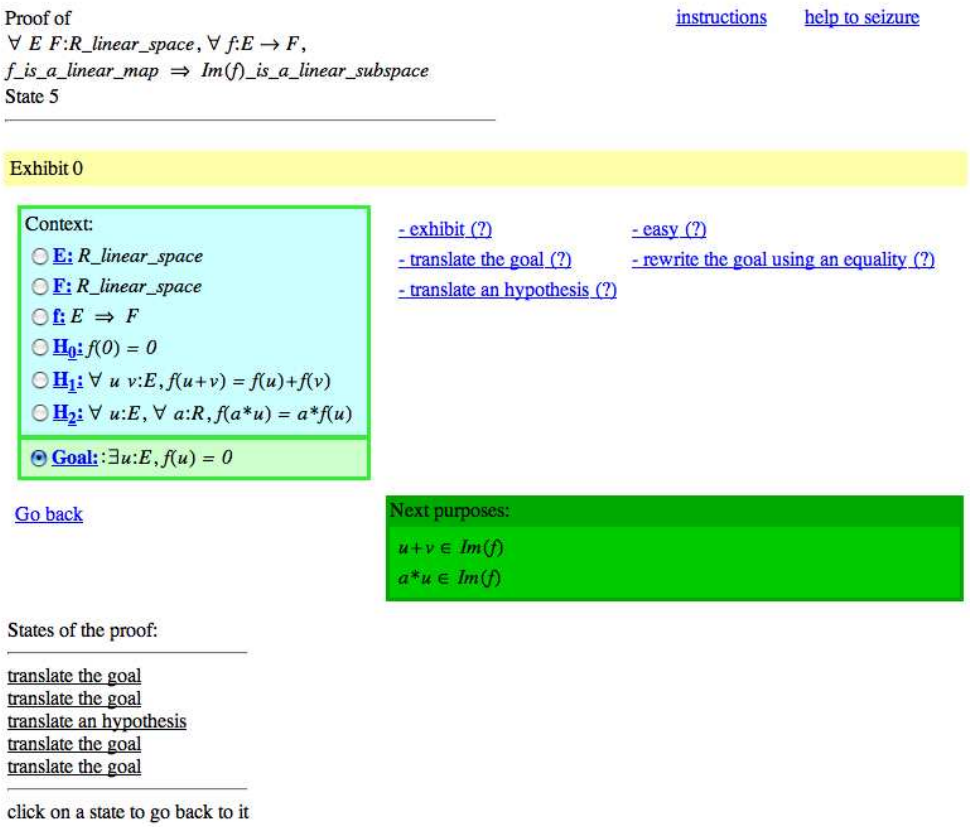
Teachers not acquainted with proof assistants insist that teaching formal proofs is relevant only if it is a way to teach informal proofs. For the moment, the only echo in Coqweb to this legitimate requirement is the guided mode. In this mode, the student is offered a hint which suggests more or less explicitly which tactic should be performed. In particular, the hint may take the form of the relevant sentence in a corresponding informal proof. In this case, the student has to translate the informal proof into a formal one. Of course, it is possible to propose exercises where the student has to perform the translation in the other way, but we did not try this yet.

2.5 Concepts versus images

One of the reasons why the proving activity is assimilated by so few students is that is is perceived as a kind of artistic activity, where very few helpful concepts and images are proposed to the beginner. Our attempt proposes several key concepts which are sufficiently simple and concrete, and to which Coqweb associates persistent images. The first main attribute of a proof is the corresponding sequence of *states* and the image for a state is the corresponding full student window. A state is itself a sequence of *purposes*, and the current purpose has its reserved place on the student window. A purpose is a pair of a *context* and a *goal* and both have their specific place in the student window. Similarly *variables* and *hypotheses* are the constituents of the context. Finally *tactics* have their own place in the student

window.

3 The student window



The student window is the window where the student is supposed to perform a proof. It can be activated either from the starting page of a Wims exercise (module coqweb_new), or through a button attached to an exercise in the teacher window. Note that students have access to the teacher window as everybody do.

The student window displays the original statement (overall goal), the current purpose, consisting of the current context and the current goal, the list of pending goals, the list of available tactics, and, optionally, the current hint. In order to perform a tactic, the student clicks on the corresponding button. In case the current tactic needs arguments, the student window opens a dialog area.

In the first version, Coqweb implemented "proof by pointing", as in CtCoq: to some extent, Coqweb was able to infer the tactic from the pointed area for instance in the goal; but we have discovered that it was too much an invitation to click without relating it to any concept.

3.1 Statements

Mathematical statements appear in the student window as goals or hypotheses. Our main concern has been that they appear in a form as close as possible to the one used by typical teachers. We give below a representative list of exercises which were solved by students during the past academic years, in the form where they appeared

in the student window (up to word by word translation from French to English). The reader will observe that it contains "immediate" consequences of definitions, as well as main results of the course. Note also that the disjunction and existential quantifier allow to ask for justified questions and calculations.

(i) Prove: $[2;3] \subset [1;4]$.

(ii) Prove: $(2,7,6)\_is\_a\_linear\_combination\_of\_(1,2,0) and (0,1,2)$.

(iii) Prove: $\exists a : R, \exists b : R, (40,62) = a * (1,14) + b * (6,1)$.

(iv) Prove:$\forall f : R \to R, (\forall x : R, f(x) = x + 1) \to f\_is\_injective$.

(v) Prove: $x : R \mapsto 2 * x + 3\_is\_injective$.

(vi) Prove: $\forall f\ g : R \to R, f\_is\_increasing\ and\ g\_is\_increasing \to f \circ g\_is\_increasing$.

(vii) Prove: $\forall f\ g : R \to R, f \circ g\_is\_injective \to f\_is\_injective\ or\ g\_is\_injective$.

(viii) Prove: $\forall f : R \to R, \forall u : sequence(R), (\forall n : N, u_{n+1} = f(u_n))\ and\ (u_0 < u_1 and\ f\_is\_increasing) \to u\_is\_increasing$.

(ix) Prove: $\forall a\ b : R, \forall f\ g : [a,b] \to R, f\_is\_increasing\ and\ g\_is\_increasing \to f + g\_is\_bounded\_above\_by\_f(b) + g(a)$.

(x) Prove: $\forall u\ v\ w : sequence(R), \forall a : R, limit(u) = a\ and\ (limit(v) = a\ and\ (u \leq w\ and\ w \leq v)) \to limit(w) = a$.

(xi) Prove: $\forall EF : R\_vector\_space, \forall f : E \to F, f\_is\_linear \to Im(f)\_is\_a\_linear\_subspace$.

3.2 Guided mode

The student window has a guided mode, where the next tactic is suggested by an informal hint. This hint is entered by the teacher directly through the student window (accessed with teacher's rights). In the guided mode, the student has to follow the suggested proof, otherwise an error message is emitted. One of the advantages of this mode is that only the necessary tactics are listed, and often they are very few, typically three or four.

3.3 Pending goals

As in other theorem provers, some tactics generate several goals. This is for instance the case of "apply" when the selected resource has several hypotheses. In this case, the pending goals are simply listed in a specific area on the student window, on the lifo mode. The corresponding contexts are not mentioned.

When using a proof assistant in order to prove a "serious" statement, the number of pending goals may easily grow up to ten or so. If in the same time these goals turn out te be difficult to read, when meeting a goal generated long ago, the user may wish to know "where this damned goal comes from". This problem is note handled at all in Coqweb. Indeed, for the statements proposed to students, pending goals are always simple and their origin is always clear.

3.4 Dialogs

Some tactics need arguments. This is the case for instance of the tactic "exists" which we call "exhibit". When this tactic is invoked, Coqweb opens a (one-line) area where the student is expected to enter the witness. The more important case is the case of the tactic "apply a resource" where Coqweb lists all relevant resources. The order in which they appear is programmed in such a way that, for our range of goals, the desired resource appears almost always among the very first ones. There is no alternative mechanism for browsing resources.

4 Tactics

In Coq as in many other theorem provers, the *tactics*, are the actions/transitions/moves which are available for jumping from one state of a proof to a new one, presumably easier. The list of Coq tactics contains around a hundred entries (see [Coq 06]). We cannot teach a hundred tactics at once, so we have to identify basic ones, and present them in such a way that the typical freshman will recognize an activity of proof which he already met earlier. We review below some of the most useful tactics and their relationship with freshmen.

4.1 The introduction tactic

The first Coq tactic, in the sense that almost every proof starts with it, is introduction. This is bad luck because, while introducing is certainly the right word (you definitely *introduce* a variable by telling its name), this cannot be understood by the typical freshman as a serious mathematical activity. If you try to teach introduction, students consider you just as mathematicians consider logicians: people concerned with meaningless mathematics. Thus introduction cannot be one of the basic freshman tactics. We will merge it with the "unfold" tactic, see below.

4.2 The apply tactics

While the typical freshman is not interested in introducing, he is perfectly interested in *applying*. Even too much. Indeed, teachers do not distinguish between three kinds of application.

The first one is what we call "apply a resource" or "apply a hypothesis". It is the major tool for a backward proof. The resource is a statement and the current goal should be the conclusion of an instance of this statement. Then the current goal is replaced by as many new goals as the instance has hypotheses. Of course, the resource should be available in the library.

The second one is what we call "deduce": again we apply a resource or a hypothesis, this time in order to prove not the current goal, but some other statement which we want to add to the current context. This tactic is what allows to proceed *forwards* (from the assumptions to the goal) while in a theorem prover like Coq, the default direction is *backwards* (from the goal to the assumptions). Note that teachers often prefer to think forwards and may reject the whole approach described here as "too backwards". While if you observe proofs from this point of view, you will

remark that the forward approach opens new room for the opacity of proofs since you may gather intermediate statements without saying why. Of course something similar happens in the backward approach. The difference is that in general the assumptions of a resource are considered easier to prove than the conclusion.

The third one is of a very different nature: we apply a resource which claims an equality in order to rewrite something in the current goal or in the current context. This is the "rewrite" tactic, which the typical freshman does not accept so easily, because he is so much used to apply and not at all to rewrite. More generally, the distinction between these three "apply" tactics is difficult to teach. At least we can stress their differences.

4.3 The unfold and simplify tactics

Just as "introduce", "unfold" is a word that mathematicians do not use. Indeed, since unfolding a definition is totally transparent for "Grandmasters", it is also completely neglected by teachers. Nevertheless, what emerges from our experience is that "unfolding" is a crucial proof activity for freshmen. This tactic shares with "introduction" the property that it has stricty no mathematical content. Thus we have merged these two tactics in a single one. For its name we have chosen the French verb "expliciter" (in the present paper, we will use "translate") which we can indeed use in the right places in our informal proofs. Also we have extended this tactic as much as possible in order to include all kinds of similar replacements. For instance, when applied to the goal $x \in A \cap B$, this tactic generates the two goals $x \in A$ and $x \in B$, although this is certainly not obtained by unfolding the definition neither of $\in$ nor of $\cap$. Similarly, from the goal $AandB$, it generates the two goals A and B.

In the same vein, we have a "simplify" tactic which we extend as far as possible (see 6.2 for an explicit use of this tactic).

4.4 Other tactics

First of all, we certainly need a tactic "easy", which handles "too" easy goals. Indeed, the typical freshman hates being asked to prove something too easy. But what looks easy for a freshman, is not necessarily so easy to prove in Coq, so that the "easy" tactic is not so easy to implement.

Next, the freshman uses the "exists" tactic, which we call "exhibit" since we consider as mandatory to find for each tactic a name which is a verb and reflects the activity. Of course this tactic is very important for understanding existence, which is a very delicate notion (not only) for freshmen. See 6.6 for an explicit use of this tactic.

Finally we have a tactic for contraposition and a tactic for induction.

5 The teacher mode

The teacher mode allows the expert teacher to declare definitions, axioms and exercises. The exercises may be activated, yielding a student window, from where

resources (definitions and axioms) will be invoked. The teacher mode runs on Wiki-Coqweb [WCW], a wiki based on Spikini, which is a variant of Wikini for Spip. For the moment, it is designed for expert teachers, more precisely for highly motivated teachers. Let us say briefly that the main task of Coqweb is to convert adequately the teacher's input into the corresponding Coq declarations, and to acknowledge it on a course page available on the wiki. More precisely, this task splits in two steps. First Coqweb checks syntactically the input before eventually writing a course page. While when a button is activated on a course page, Coqweb performs a semantic check before definitely acknowledging the definition, or before opening a student window. . We briefly describe the contents of course pages and the way they are edited.

5.1 Course pages

A course page is a page where the teacher can write a text including latex formulas, and insert Coqweb definitions and exercises. Here is for instance how appears the definition for vector space

```
Definition  cons_ev : qs E :Ens, (E->E->E) -> E->( R->E-> E)-> PREV
         (en coq: cons_ev)
Properties:

    * qs E :Ens, qs a : E->E->E, qs z :E, qs m : R->E ->E,
      E=( cons_ev a z m)
    * qs E :Ens, qs a : E->E->E, qs z :E, qs m : R->E ->E,
     ( (plus_ev (E := ( cons_ev a z m) ) ) = a )
    * qs E :Ens, qs a : E->E->E, qs z :E, qs m : R->E ->E,
     ( (zero_ev ( cons_ev
a z m) ) = z )
    * qs E :Ens, qs a : E->E->E, qs z :E, qs m : R->E ->E,
      ( (mult_ev (E := ( cons_ev a z m) ) ) = m )
```

Note that this is for the teacher, not for the student, even if it is freely available.

For each definition, there is a button for converting altogether the definition into a Coq declaration and the properties into axioms. For this conversion to work, the relevant other pages should have been loaded, and the previous definitions in the current page should have been already converted. If Coqweb does not succeed in this conversion, a (rough) error message appears. For each exercise, there is a button which is supposed (if the formulation is correct with respect to previously loaded material) to open a student window.If Coqweb does not succeed in converting the statement into a Coq statement, a (rough) error message appears.

Course pages are linked to one another in the standard hypertext way.

5.2 Editing course pages

Course pages may be edited within the wiki. The conception and edition of course pages is the main teacher's task, along with the optional task of writing hints. When trying to validate a definition together with its accompanying properties, the

teacher often requests the disambiguation ability of Coqweb. However we do not expect Coqweb to choose among various possible interpretations according to the context. Our idea is rather that teachers (and students) should write statements in a way offering only one reasonable interpretation. As an illustration of this position, our quantifiers are always explicitly typed.

Coqweb offers the possibility to declare two names for each definition: one name is for Coq and supports no overloading, while the second one is for Coqweb only and supports overloading. Of course the two names may coincide, which is the case in the above exemple..

6 Algebraic numbers in Coqweb

We present here some pages of Coqweb which have been used at the University of Nice-Sophia-Antipolis in 2007 along within a mini-course for motivated students on polynomial equations and their solutions. to present Coqweb to teachers.

6.1 Fields

Firstly, we will work with fields. It is convenient to code this in one page, called "fields", and then to specialize to some special fields like $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$. Here is what the page on the web looks like (we show only the compiled page, instead of the code page, as it is more readable, but the syntax is quite the same):

Definition *field: Set*
Definition *coercion:field→Set*
Definition *+: ∀K: field, K→K→K*
Properties:
- *∀K:field, ∀x y z:K, (x+y)+z=x+(y+z)*
- *∀K:field, ∀a b:K, a+b=b+a*

Definition **: ∀K: field, K→K→K*
Properties:
- *∀K:field, ∀x y z:K, (x*y)*z=x*(y*z)*
- *∀K:field, ∀a b:K, a*b=b*a*
- *∀K:field, ∀x y z t:K, (x+y)*(z+t)=x*z+x*t+y*z+y*t*

Definition *1: ∀K: field, K*
Properties:
- *∀K:field, ∀x:K, x*1=x*

Definition *0: ∀K: field, K*
Properties:
- *∀K:field, ∀x:K, x*0=0*
- *∀K:field, ∀x:K, x+0=x*
- *not(0=1)*

Definition */: ∀K: field, K→K→K*
Properties:
- *∀K:field, ∀x y z:K, (not (y=0)→(x/y)*y=x*

Definition *-: ∀K: field, K→K*
Simplifications:
- *∀K:field, ∀x:K, x+(-x)=0*
- *∀K:field, ∀x:K, (-x)+(x)=0*

Exercise: ∀K: field, ∀x y: K, x+y=0→y=-x

Definition *-: ∀K: field, K→K→K*
Properties:
- *∀K:field, ∀x y:K, (x-y)=x+(-y)*

Exercise: ∀K: field, ∀x: K, (-(-x)=x
Exercise: ∀K: field, ∀x y: K, -(x+y)=(-x)+(-y)
Definition *power: ∀K: field, K→N→K*
Properties:
- *∀K:field, ∀x:K, x^0=1*
- *∀K:field, ∀x:K, ∀a:N, x^(a+1)=x^a*x*

Each element of the set *field* corresponds to a field, and may be also viewed – using the coercion above – as a set. The symbols 0 and 1 represent the traditional constants. Observe the standard overloading for "-" .

Note that the function *power* may be used writing the symbol ^, and that the set $N = \mathbb{N}$ was introduced earlier, in a similar way.

The three exercises above are simple examples suited for the student who begins to use Coqweb. The student will see that although the assertions seem very simple, they are not so trivial to prove. He will need less than ten tactics in general , and

will see the so many steps that we usually skip in a classical proof.

6.2 A first proof in detail

We illustrate now in detail a proof of the first exercise of Section 6.1, describing what the student sees after each of his uses of tactics:

Goal: $\forall K$: *field*, $\forall x\ y$: K, $x+y=0 \to y=-x$

The student clicks on the tactic *"Translate the goal"*. This corresponds to the usual mathematical sentence "Let K be some field, and let x, y be two elements in K".

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **Goal:**$y=-x$

The student clicks on the tactic *"Prove an intermediate result"*. The computer asks him what the result is, and he chooses "`-x+(x+y)=-x`".

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **Goal:**$-x+(x+y)=-x$, **Next purpose:**$y=-x$

The student clicks on the tactic *"Rewrite the goal using an equality"*; the computer offers a list of available equalities, among which the student selects the hypothesis **H**. The computer replaces the expression $x + y$ by 0 in the goal.

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **Goal:**$-x+0=-x$, **Next purpose:**$y=-x$

The student chooses the tactic *"Simplify the goal"*. (Recall that the property *"*$\forall K$*:field,* $\forall x$*:*K*,* $x+0=x$*"* was declared as a simplification).

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **Goal:**$-x=-x$, **Next purpose:**$y=-x$

The student may now click on the tactic *"easy"* to achieve the proof of the intermediate goal, which becomes an additional assumption in the next context.

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **H$_0$:**$-x+(x+y)=-x$, **Goal:**$y=-x$

The student clicks on *"Rewrite the goal using an equality"* and selects the hypothesis **H$_0$**. The computer replaces the expression $-x$ by $-x + (x + y)$ in the goal.

Context: **K:***field*, **x:**K, **y:**K, **H:**$x+y=0$, **H$_0$:**$-x+(x+y)=-x$, **Goal:**$y=-x+(x+y)$

Three tactics are then needed to conclude. The first one rewrites $-x + (x + y)$ into $-x + x + y$, using the commutativity defined as a property of the addition. The second one simplifies the goal (the computer will replace $-x+x$ by 0 and then $0+y$ by y), to obtain $y = y$ instead of $y = -x + x + y$. The last one is *"easy"*.

6.3 The field $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$

We define some special elements of the set *field*, which correspond to the standard fields $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$, and add some specific properties to these. For $\mathbb{R}$ we add some order, and the n-th roots:

Definition *nroot: R→N→R (in coq: nroot)*
Properties:
- $\forall x$*:R,* $\forall n$*:N,* $x>=0 \to n>0 \to$ *(nroot x n)^n=x*

In a Coqweb's proof, the expression *(nroot x n)* is displayed as $\sqrt[n]{x}$.

For $\mathbb{C}$, we add the functions *Im* and *Re* which give the real and imaginary parts. We do not present these explicitly here.

6.4 Operations on functions over a field

Since our polynomials will be coded as functions, we need some operations on functions over a field. We define the addition, multiplication and subtraction of two functions and the opposite function of a function. The addition is for example coded like this:

Definition *+: ∀E:Set,∀K: field, (E→K)→(E→K)→E→K*
Properties:
• *∀E:Set, ∀K:field, ∀f g:E→K, ((f+g) x)=(f x)+(g x)*

The other operations are similarly defined.

6.5 Polynomials

In our implementation, a polynomial is a special function. We define the standard polynomial X (which is in fact the identity map), the constant polynomials, and continue to define other polynomials inductively. We work over the complex field $\mathbb{C}$.

Definition *X: C→C*
Properties:
• *∀a: C, (X a)=a*
Definition *coercion: C→(C→C) (coq name: cfcst)*
Properties:
• *∀a x: C, (cfcst a x)=a*
Definition *is_a_polynomial: (C→C)→Prop*
Properties:
• *∀a: C, (is_a_polynomial (a)*
• *is_a_polynomial (X)*
• *∀P Q :C→C, (is_a_polynomial P)→(is_a_polynomial Q)→is_a_polynomial(P+Q)*
• *∀P Q :C→C, (is_a_polynomial P)→(is_a_polynomial Q)→is_a_polynomial(P*Q)*
• *∀P :C→C, ∀a :C, (is_a_polynomial P)→is_a_polynomial(a*P)*
• *∀P :C→C, ∀n :N, (is_a_polynomial P)→is_a_polynomial(P^n)*
Exercise: is_a_polynomial (nroot 2 3)*X^3+1)
Exercise: ∀a b c:C, is_a_polynomial (a*X^2+b*X+c)

The proofs of the two exercises above are similar. Only two tactics are needed, i.e. *"apply a resource"* and *"easy"*. For example, in the first exercise, the student will apply the resource *"∀P Q :C→C, is_a_polynomial P→is_a_polynomial Q→is_a_polynomial (P+Q)"* and then will have to prove that $\sqrt[3]{2} \cdot X^3$ and 1 are polynomials. Applying the other resource *"∀P Q :C→C, is_a_polynomial P→is_a_polynomial Q→is_a_polynomial (P*Q)"*, he will have to prove that $\sqrt[3]{2}$, X^3 and 1 are polynomials. And so on, he will achieve the proof applying several times the properties of the definition *"is_a_polynomial"*.

To define the algebraic numbers, we have to highlight polynomials which have integer coefficients:

Definition *has_int_coeffs : (C→C)→Prop*
Properties:
- *∀a :Z, has_int_coeffs a*
- *has_int_coeffs X*
- *∀P Q : C→C, (has_int_coeffs P)→(has_int_coeffs Q)→has_int_coeffs(P+Q)*
- *∀P Q : C→C, (has_int_coeffs P)→(has_int_coeffs Q)→has_int_coeffs(P-Q)*
- *∀P Q : C→C, (has_int_coeffs P)→(has_int_coeffs Q)→has_int_coeffs(P*Q)*
- *∀P : C→C, ∀a :Z, (has_int_coeffs P)→has_int_coeffs (a*P)*
- *∀P : C→C, ∀n :N, (has_int_coeffs P)→has_int_coeffs (P^n)*
- *∀P : C→C, (has_int_coeffs P)→(is_a_polynomial P)*

Exercise: has_int_coeffs (5*X+1)
Exercise: has_int_coeffs (3*X^3+1)
Exercise: ∀n :N, (has_int_coeffs (X^n))
Exercise: ∀n :N, ∀a b :N, has_int_coeffs (a*X^n+b)

The exercises above are similar to those concerning *"is_a_polynomial"*.

6.6 Algebraic numbers

We may now define what an algebraic number is:

Definition *is_algebraic : C→Prop*
Properties:
- *∀a :C, (is_algebraic a)= (ex P : C→C, (has_int_coeffs P) and (ex b :C, not (P(b) =0)) and (P(a)=0))*

Exercise: ∀a : Z, (is_algebraic a)
Exercise: is_algebraic (nroot 2 2)
Exercise: is_algebraic (i)

We describe a proof of the first exercise. The student applies the tactic *"Translate the goal"* and obtains the following:

Context: **a:***Z*, **Goal:**∃*P:C⇒C*, (*has_int_coeffs P*) ***and*** (∃*b:C*, *P*(*b*)≠*0*, ***and*** *P*(*a*)=*0*)

Clicking on the "Exhibit" tactic, the student may choose the value of P to be $X - a$.

Context: **a:***Z*, **Goal:**(*has_int_coeffs X-a*) ***and*** (∃*b:C*, *X-a*(*b*)≠*0*, ***and*** *X-a*(*a*)=*0*)

The tactic *"Translate the goal"* may be used again, and the student has to prove the three subexpressions of the previous goal. The first one, *has_int_coeffs X-a*, is proved as above, using the properties of *"has_int_coeffs"*. The second one, ∃*b:C*, *X-a*(*b*)), is proved choosing $b = 1 + a$ and simplifying the expression obtained, thanks to the properties of the functions over a field and the operations on a field. The third one, *X-a*(*a*)=*0*, is proved similarly.

For first-year students, proving the following theorems

Theorem: ∀a b : C, (is_algebraic a)→(is_algebraic b)→(is_algebraic (a+b)
Theorem: ∀a b : C, (is_algebraic a)→(is_algebraic b)→(is_algebraic (a*b)

would be far too hard. However, the exercises below treat examples which may be handled directly, and could give a feeling on the theorem.

Exercise: is_algebraic (i+1)
Exercise: ∀a b : C, (a^2+a+1)=0→(b^2-b-1)=0→(is_algebraic (a+b))
Exercise: ∀a : C, a^5-a-1=0→(is_algebraic (a+1))

Exercise: ∀a : C, a^5-a-1=0→(is_algebraic (a^2+ (nroot 2 2)*a-2))
Exercise: is_algebraic (2/(1+ (nroot 2 2)
Exercise: is_algebraic (2 + 3*(nroot 3 2)

The student has to find the good polynomials, and replace these in the proofs. These are examples of proof exercises that test in fact calculation.

We now define which elements and subsets of ℂ are "solvable".

Definition *is_solvable : (partie C)→Prop*
Properties:
• *∀X : (partie C), (is_solvable X)= ((is_subfield X) and*
(∀a : C, ∀b : X, ∀n : N, (n>0)→(b=a^n)→(a _appartient_a_ X)))
Exercise: ∀X Y : partie(C), (is_solvable X)→(is_solvable Y)→(is_solvable (inter X Y))
Definition *is_solvable : C→Prop*
Properties:
• *∀a :C , (is_solvable a)= (∀X : partie(C),(is_solvable X)→(a _appartient_a_ X))*

The above exercise is quite long, but not difficult and may be interesting to do for the student. He has to use every particularity of the two definitions of *"is_subfield"* and *"is_solvable"*, and see that these are propagated to the intersection.

The set of all solvable elements was called "RESOL", and the last exercise of the file was to prove that this set is solvable:

Definition *RESOL : partie(C)*
Properties:
• *∀a :C , (a _appartient_a_ RESOL)=(is_solvable a)*
Exercise: is_solvable (RESOL)

This exercise is also quite long but not too difficult; the student has to show that RESOL satisfies each of the condition of *is_solvable*.

7 Related and future works

The pionneering interface for Coq has been CtCoq [Bertot 99], based on LeLisp, which is no more maintained. Later on, the main features of CtCoq, including "proof by pointing" have been recollected and extended in PCoq [PCOQ], which is implemented in Java. While CtCoq and PCoq were designed specifically for Coq, ProofGeneral [Proofgeneral] is designed for a generic theorem prover, but does not support proof by pointing. CoqIde is a variant of ProofGeneral dedicated and integrated to Coq. PhoX [PHoX] is a theorem prover aiming at a "minimal learning time". It has been used by third-year students in Chambery for several years. Matita [Asperti and al.07] seems to be the latest theorem prover: it is "document-centric" and its two main concerns are efficient interaction with a (presumably large) library and disambiguation of user input (see in particular [Sacerdoti Zacchiroli 04,Sacerdoti Zacchiroli 07]). Coqweb seems to be the first interface designed for teaching, with separate modes for students and teachers. It needs all kinds of improvements:

(i) the student window is far from perfect. In particular, the display of statements could still be much better.

(ii) at the end of a proof, it should offer a trace, which could be accepted by typical teachers as a correct informal proof.

(iii) there is no mechanism for searching resources, which is a problem when, form some unexpedted reason, Coqwed does not find the desired one.

(iv) the teacher window is far from user-friendly. In particular, the way overloading and coercions may be used is not sufficiently clearly specified and error messages are often frustrating.

(v) the library should be completed and the wiki should be better organized.

On the longer range, we also hope that Coqweb could be upgraded and become useful for more and more advanced mathematicians.

References

[Asperti and al.07] A.Asperti, C.Sacerdoti Coen, E.Tassi, S.Zacchiroli. "User Interaction with the Matita Proof Assistant." To appear in the Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving.

[Autexier Sacerdoti Coen 06] S. Autexier, C.Sacerdoti Coen: A Formal Correspondence between OMDoc with Alternative Proofs and the lambda-bar-mu-mu-tilde-Calculus MKM2006, LNAI, 4108, 67–81, 2006.

[Bertot 99] Yves Bertot, "The CtCoq System: Design and Architecture", Formal aspects of Computing, Vol. 11, pp. 225-243, 1999.

[BlogDuB] `http://pcmath165.unice.fr/annamath/spikini/?wiki=LeBlogDuB`

[Coq 06] The Coq proof assistant `http://coq.inria.fr/coq-eng.html`

[Coqweb] `http://pcmath165.unice.fr/wcw/spikini`

[Corbineau] Pierre Corbineau, Declarative Proof Language for Coq `http://www.cs.ru.nl/ corbinea/mmode.html`

[Epp 03] Suzanna S. Epp, The Role of Logic in Teaching Proof, American Mathematical Monthly (110)10, Dec. 2003, 886-899.

[FTA] Herman Geuvers, Freek Wiedijk, Jan Zwanenburg, Randy Pollack, Henk Barendregt, Fundamental Theorem of Algebra `http://www.cs.ru.nl/ freek/fta/`

[Girard 07] J.Y.Girard, Informal conversation, March 14, 2007

[Gonthier Werner 04] Georges Gonthier and Benjamin Werner, Coq Proof of the Four Color Theorem, `http://research.microsoft.com/ gonthier/`

[Luo 99] Z.Luo "Coercive Subtyping" Journal of Logic and Computation.

[MIZAR] The Mizar Home Page `http://mizar.uwb.edu.pl/`

[PCOQ] `http://www-sop.inria.fr/lemme/pcoq/pcoq-fra.html`

[PHoX] `http://www.lama.univ-savoie.fr/ RAFFALLI/phox.html`

[Proofgeneral] `http://proofgeneral.inf.ed.ac.uk/`

[Sacerdoti 06] C. Sacerdoti Coen: Explanation in Natural language of lambda-bar-mu-mu-tilde-terms In Fourth International Conference on Mathematical Knowledge Management (MKM2005), LNAI, Vol. 3863, 234–249, 2006.

[Sacerdoti Zacchiroli 04] Claudio Sacerdoti Coen, Stephano Zacchiroli: Efficient Ambiguous Parsing of Mathematical Formulae MKM04 , LNCS 3119, 347-362, 2004.

[Sacerdoti Zacchiroli 07] Claudio Sacerdoti Coen and Stefano Zacchiroli: Spurious Disambiguation Error Detection, in Proceedings of MKM 2007: . LNAI, to appear.

[WCW] `http://pcmath165.unice.fr/wcw/spikini/?wiki=AccueilWikiCoqWeb`

[Wiedijk 04] Freek Wiedijk, "Integrating procedural and declarative proof", small TYPES workshop, University of Nijmegen, 2004-11-01,

[WIMS07] Deuxime colloque international WIMS Enseigner, créer des ressources avec WIMS 9 - 10 - 11 Mai 2007 Nice Sophia Antipolis.

[Xiao 98] Gang Xiao, WWW Interactive Multipurpose Server, `http://wims.unice.fr/wims/`

Some considerations about proof assistants for education

René David and Christophe Raffalli

Université de Savoie

Abstract

PhoX is used for teaching mathematics, logic and computer sciences since almost ten years at the université de Savoie and Paris VII. Many design decisions and improvements to the proof assistant where decided for this specific application. In this talk, we will expose the ideas that helped the students, those that did not bring much and the new improvement that are planned for future versions.

Talk