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 $\phi : P(\mathbb{N}) \rightarrow (P(\mathbb{N}) \rightarrow P(\mathbb{N}))$ 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, anti-symmetry 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).

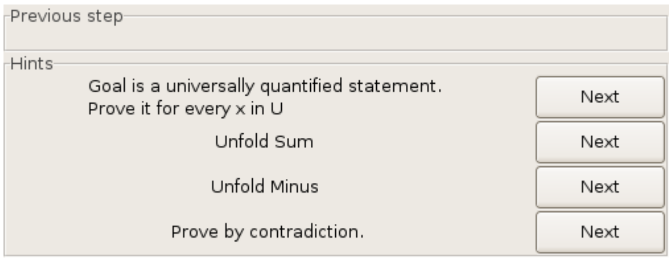
5.1 Hints: tactics for 1st order logic

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


Previous step

Hints

Goal is a universally quantified statement.
Prove it for every x in U

Unfold Sum

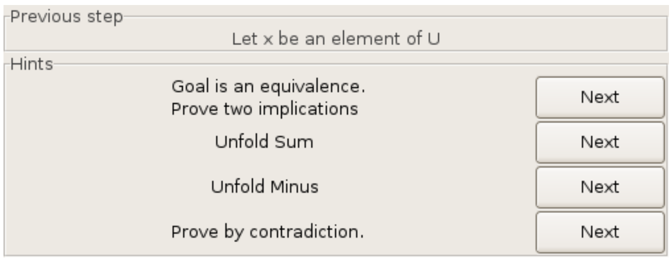
Unfold Minus

Prove by contradiction.

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:


Previous step

Let x be an element of U

Hints

Goal is an equivalence.
Prove two implications

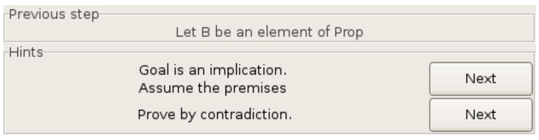
Unfold Sum

Unfold Minus

Prove by contradiction.

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

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


Previous step

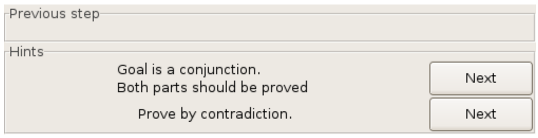
Let B be an element of Prop

Hints

Goal is an implication.
Assume the premises

Prove by contradiction.

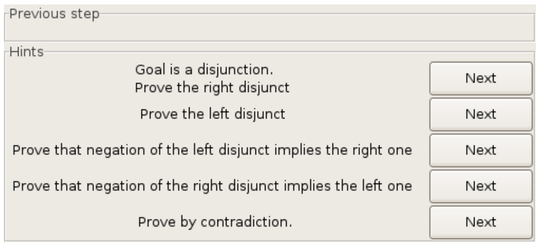
```
A : Prop
B : Prop
-------------------------(1/1)
A /\ B
```



Goal is a conjunction.
Both parts should be proved

Prove by contradiction.

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



Goal is a disjunction.
Prove the right disjunct

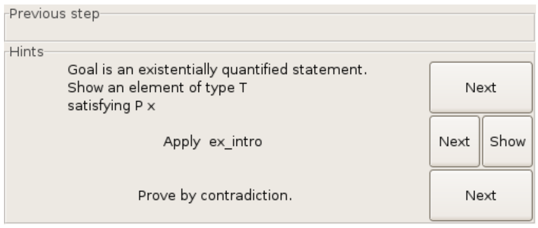
Prove the left disjunct

Prove that negation of the left disjunct implies the right one

Prove that negation of the right disjunct implies the left one

Prove by contradiction.

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



Goal is an existentially quantified statement.
Show an element of type T
satisfying P x

Apply ex_intro

Prove by contradiction.

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`).

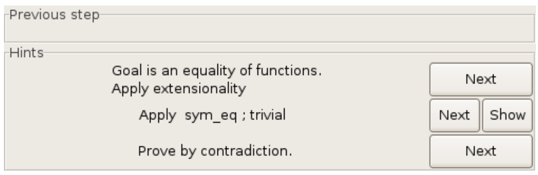
The hint for the existential quantifier needs a comment too. We use the tactic `eapply ex_intro`, which leaves the goal with a "hole". In fact, the user should rather use the `exists` tactic and give the right element explicitly.

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 `uncurry:(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
```



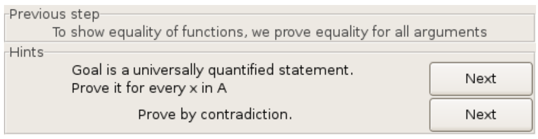
Goal is an equality of functions.
Apply extensionality

Apply sym_eq ; trivial

Prove by contradiction.

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
```



To show equality of functions, we prove equality for all arguments

Goal is a universally quantified statement.
Prove it for every x in A

Prove by contradiction.

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

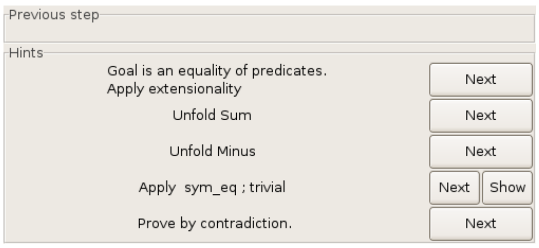
Showing this hint to students is important for at least two reasons. First of all, students are used to prove an equality by simplifying both sides of it until they reach identical expressions. Here they have to learn a completely different strategy. Second, as shown by our practice, students, even those 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

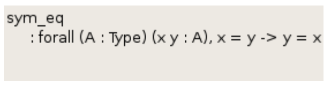
Since Coq already has a similar mechanism to help the user in remembering useful lemmas through their automatic application with the `auto` tactic, we included the list of applicable lemmas from the `auto` database in the list of suggestions presented in the Wizard Window. However, since we check that 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:



```
sym_eq
    : forall (A : Type) (x y : A), x = y -> y = x
```

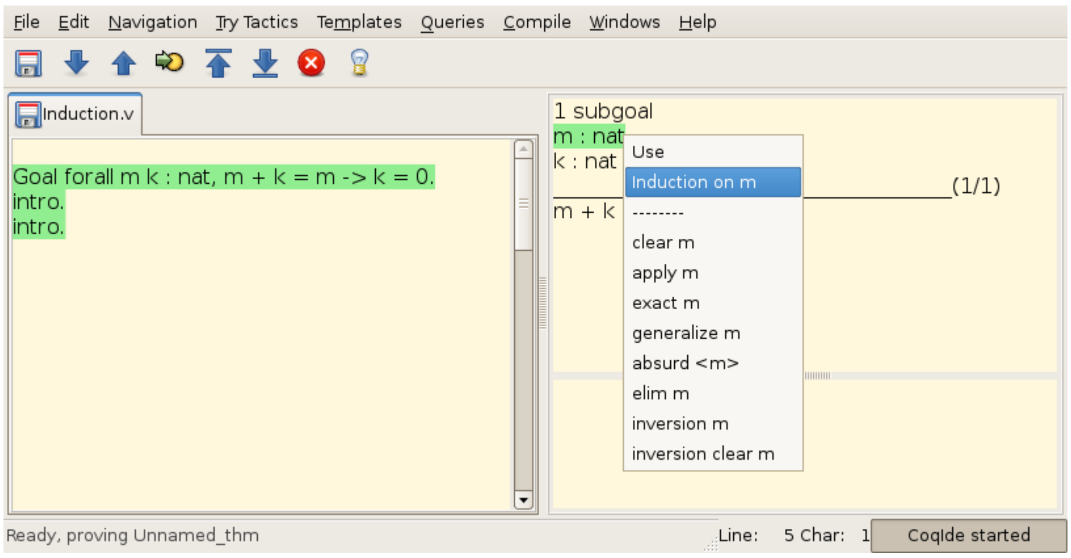
5.4 Simplified use of assumption

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:



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

```
1 subgoal
m : nat
k : nat
______________________________________(1/1)
m + k
```

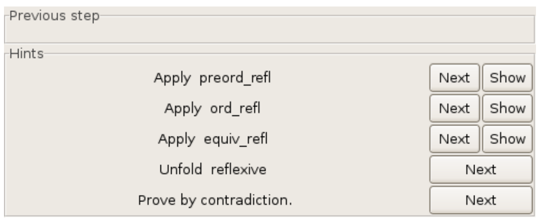
```
Ready, proving Unnamed_thm          Line:    5 Char:    1    CoqIde started
```

5.5 *Other hints*

Besides the hints mentioned above, Papuq offers:

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

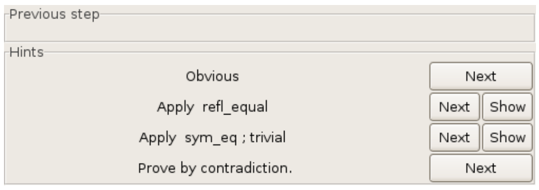
```
Triangle : Type
Number : Type
R : relation Triangle
Area : Triangle -> Number
---------------------------(1/1)
reflexive R
```



Apply preord_refl	Next	Show
Apply ord_refl	Next	Show
Apply equiv_refl	Next	Show
Unfold reflexive	Next	
Prove by contradiction.	Next	

- 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
```



Obvious
Apply refl_equal
Apply sym_eq ; trivial
Prove by contradiction.

- 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.