MMode
A Mizar Mode for the proof assistant Coq

Mariusz Giero[2,1] and Freek Wiedijk[1]

[1] Department of Computer Science, University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
[2] Department of Logic, Informatics and Philosophy of Science,
University of Białystok, Plac Uniwersytecki 1, 15-420 Białystok, Poland

Abstract. We present a set of tactics for version 7.4 of the Coq proof assistant which makes it possible to write proofs for Coq in a language similar to the proof language of the Mizar system. These tactics can be used with any interface of Coq, and they can be freely mixed with the normal Coq tactics.
The system described in this report can be downloaded on the web at `<http://www.cs.kun.nl/~freek/mmode/mmode.tar.gz>`.

1 Introduction

1.1 The goal: a declarative proof language for Coq

Computers have made it practical to encode non-trivial mathematics in such a way that the correctness of the proofs can be automatically verified. However, this kind of encoded proof – the way the encoder works with it – currently does not resemble the mathematics that we know from journals and textbooks. Instead it looks like source code in a programming language.

The proof languages of the proof assistants (programs to verify the correctness of encoded mathematics) come in two flavors. On the one hand there are the *procedural* proof languages. These are the languages of the systems based on the architecture of the LCF system [7]. Examples of procedural provers are Coq [15], NuPRL [3], HOL [6, 8], Isabelle [12] and PVS [13]. In such a system a proof generally works its way backward, from the end of the proof to the beginning, and consists of a sequence of *tactics*, which are commands that transform a proof state. On the other hand there are the *declarative* proof languages. In a declarative system a proof is a sequence of statements, and proceeds from the beginning of the proof to the end. The steps in such a proof generally are bridged by some kind of automation built into the system.

The declarative provers differ according to how much automation the system offers. On the one hand there are the fully *automated theorem provers* like ACL2 [9] and Theorema [2]. Here a mathematical development is a linear list of lemmas, and the system tries to prove each lemma from the previous ones. On the other hand there are systems like Automath [11] and Agda [4]. These have no automation at all, but support *natural deduction* style declarative proof with

a hierarchical block structure. In the middle of this spectrum there is Mizar [10, 18]. Here a proof has a natural deduction style block structure, but the steps in the proof are also inferred from earlier ones using automation.

The notions of procedural and declarative prover are not very sharp.[3] Maybe the terms do not mean much more than 'the proofs look similar to Coq and HOL proof scripts' and 'the proofs look like Mizar proofs'. If one thinks in terms of natural deduction proofs, the difference is largely the amount of 'cuts' in proofs for the system: in a procedural prover a proof generally contains at most a few cuts, while in a declarative prover proofs contain a cut for almost every proof step. Also, the adjectives 'procedural' and 'declarative' really apply to the proof *language* used to write proofs for the system. Maybe we should not speak about procedural/declarative *provers*, but just of procedural/declarative *proofs*.

It is generally agreed that a declarative proof is more similar to the informal mathematical proofs from journals and textbooks than a tactics proof for a procedural system. For this reason some systems have put a declarative proof language on top of their procedural base. Generally this declarative proof language resembles the language of the Mizar system. The first to experiment with such a *Mizar mode* was John Harrison, who built one on top of his HOL Light system [8]. The main procedural system for which a Mizar-like proof language has become the accepted way of writing proofs is Isabelle, for which Markus Wenzel developed the Isar proof language [17].

We think that the Coq system should follow the example of Isabelle. This report describes a prototype declarative proof language for the Coq system. One goal for this language is that it should be closely integrated with the 'old style' Coq proof language. In particular all the automation of the Coq system should be available in the language in a natural way. Another goal is that it should be easy for people to experiment with our language.

Originally we called our prototype language 'Mizar mode', but then Henk Barendregt proposed the name 'mathematical mode', to stress the fact that the language should resemble the look of informal mathematics. Eventually both names were abbreviated to 'MMode' which is the current name of our system. (A third reading of MMode is 'Mariusz' mode'.)

Note that our system can *not* be used to automatically import Mizar proofs into Coq. We only imitated the *proof language* of Mizar for Coq, the logical foundations beneath the two systems still are completely different.

1.2 Approach

We decided to implement the MMode language in the spirit of Mizar Light [20]. This is another experimental Mizar mode for the HOL Light system. The main

[3] Once again, the differences are: (a) in a procedural prover proofs generally run backward from the goal to the assumptions, while in a declarative prover they run mostly forward from the assumptions to the goal, and (b) in a procedural prover the proof scripts generally do not contain the statements that occur in the proof states, while in a declarative prover they do.

difference with the Mizar mode by John Harrison is that there is no separate parser. Instead the system just consists of a collection of 'Mizar tactics'. These tactics can be freely interleaved with the already existing tactics of the system. If one just uses these Mizar tactics, the proof closely resembles a declarative proof in the Mizar language.

We had two models for the MMode language. On the one hand there was the Mizar language. The Mizar language has a large grammar, but the *proof fragment* (the part of the grammar that is directly related to proof steps, so what one gets if one leaves out the grammar rules that are related to other things like term/formula syntax or the type system) actually is rather small. On the other hand there were Henk Barendregt's ideas about what a declarative language should look like [1]. Since the Mizar language is much further developed than Henk's ideas, we decided to start by implementing the proof fragment of the Mizar language. Later we extended this to get closer to Henk's ideas as well (this will be described in Section 7).

The part of Mizar that we took as a model for the MMode language is given by the following context free grammar[4]:

typing := *var* { '**,**' *var* } [('**be**' | '**being**') *type*]
typings := *typing* { '**,**' *typing* }

proposition := [*label* '**:**'] (*formula* | '**thesis**')
propositions := *proposition* { '**and**' *proposition* }

simple-justification := ['**by**' *label* { '**,**' *label* }]
proof := '**proof**' { *step* } [*cases*] '**end**'
justification := *simple-justification* | *proof*

statement :=
  propositions justification
| [*label* '**:**'] *term* '**=**' *term justification* { '**.=**' *term justification* }

step :=
  '**let**' *typings* '**;**'
| '**assume**' *propositions* '**;**'
| ['**then**'] *statement* '**;**'
| ('**thus**' | '**hence**') *statement* '**;**'
| ['**then**'] '**consider**' *typings* '**such**' '**that**'
    propositions justification '**;**'
| '**take**' *term* ('**,**' *term*) '**;**'
| '**set**' *var* '**=**' *term* '**;**'

cases :=
  '**per**' '**cases**' *justification* '**;**'
  { '**suppose**' *propositions* '**;**' { *step* } }

[4] In this grammar we have used the following customary abbreviations: [*notion*] means zero or one occurrences of *notion*, {*notion*} means zero or more repetitions of *notion*, and ($notion_1$ | $notion_2$) means a choice between $notion_1$ and $notion_2$.

For an explanation of the meanings of the various constructions in this grammar, see [18].

This grammar is almost the full proof fragment of Mizar. However, there are a few differences;

- This grammar is more orthogonal than the real Mizar syntax. For instance in this grammar a **proof** can be used everywhere where a **by** can be used. In the real Mizar various steps (like **consider** and **per cases**) can only be proved with a **by** justification. Also in this grammar a **then** can be used at any point, while in the real Mizar a **then** is forbidden after a **consider** step.
- In this grammar some abbreviated steps from the real Mizar have been omitted. For instance there is no **given** (a combination of **assume** and **consider**) and there is no **let ... such that** (a combination of **let** and **assume**). These abbreviations can be trivially expanded, so their omission does not restrict the expressivity of the language.
- The Mizar steps related to the Mizar type system like **reconsider** have been omitted. They make no sense in a system like Coq in which types are unique.
- A few other Mizar constructions have not been included either. For instance **deffunc** and **defpred** are not in this grammar. (In a higher order system like Coq these are special cases of **set**.) The **case** variation of the **suppose** step in a **per cases** construction has also been omitted for simplicity.

All these differences are minor and therefore this language fragment should *not* be interpreted as being less powerful than the full proof language of the Mizar system.

We wanted the threshold for experimenting with the MMode to be as low as possible. For this reason we took the following design decisions:

1. *We did not want a separate parser for our language. All syntax was implemented through Coq's* `Grammar` *rules.*
   This choice makes it possible to use the MMode system in any Coq interface that accepts user-defined tactics with a syntax given by `Grammar` rules. By this choice we hoped to minimize the need for users to switch to a different interface to be able to use our system. In particular the MMode language works fine with the Proof General interface of Coq.
   This choice caused some complications. One was that the keywords **Let** and **Show** already were used by the original Coq language, and could not be used by MMode. Therefore MMode uses the keywords **Let_** and **Show_**, with an underscore at the end.
   Another problem was that we wanted the justification of a step to be an arbitrary Coq tactic. However in Coq 7.4 tactics cannot have tactics as an argument. Hugo Herbelin kindly showed us how to patch Coq 7.4 to allow this. However, this means that one needs to patch Coq to be able to try our system. We discuss this in more detail in Section 2.
2. *We did not want tactics implemented on the ML level. All tactics in the MMode language were implemented using David Delahaye's* $\mathcal{L}_{\mathrm{tac}}$ *language.*

In the C-CoRN project [5] we had one tactic (called `Rational`) that was implemented in ML. For this reason to be able to use the C-CoRN library with the native version of Coq, people had to build a C-CoRN specific Coq image. This should be trivial, but in practice this often was causing problems. For the MMode language we wanted to avoid this complication.
This choice slowed us down quite a bit. Often $\mathcal{L}_{\text{tac}}$ behaved erratically and even if it behaved properly the error messages were not very informative.

These two design choices might be changed for a later version of the system. For the moment we consider MMode to be a prototype that shows what is possible. For this reason we wanted it to be as easy as possible for people to try our system. If people happen to like it, a later version might depart more from the standard Coq environment.

We did not develop the MMode system systematically. Instead we took some existing Coq and Mizar proofs, and tried to fit them into a Coq version of the Mizar fragment that we showed earlier. We just implemented the MMode tactics that were needed to make these example proofs work. The example proofs are shown in Section 6 and fall into three categories:

Translations of Coq proofs that just use the basic Coq tactics, for examples that are easy to understand. Among these were proofs by Henk Barendregt from his development of the Fundamental Theorem of Arithmetic;

Translations of Coq proofs that use a complicated library. In our case this was the C-CoRN library. We included this kind of proof to show that our approach also works with non-trivial tactics;

Translations of Mizar proofs, to show the similarity of the MMode version to the original Mizar version.

1.3 Related Work

Our goal is to extend Coq to a system that can be used in a declarative way. Most proof assistants are procedural: only a few systems are declarative. The main declarative system is Mizar. Other declarative systems in the style of Mizar are Don Syme's DECLARE [14] and Vincent Zammit's SPL [21].

Mizar modes for procedural systems are not common either. The Mizar modes for HOL are currently just experiments. However, the Isar language for Isabelle [17] has become a success. It is now the official input language for the Isabelle system.

Our MMode language differs in an important respect from Isar (for a comparison of Mizar and Isar, see [16]). In Isar the declarative and procedural parts of the system both have a proof state of their own, called the static and dynamic proof states. However, in MMode we follow the approach from the Mizar modes for HOL, where the proof state that is used by the normal tactics of the system is also used for the declarative proofs. We think that this is more elegant and simple.

1.4 Contribution

We have created a system for writing declarative proofs with Coq. We have reimplemented most features of the proof language of the Mizar system.

The system is not a separate layer on top of Coq, but is written in such a way that proofs can freely mix old style Coq tactics and declarative proof steps. Also, the system can be used in any reasonable Coq interface.

1.5 Outline

This report is organized as follows. In Section 2 we describe the organization and installation of our system. In Section 3 we present the MMode proof language. In Section 4 we describe the inference automation (the **by** tactic) that we developed for the MMode system. In Section 5 we give a detailed account of each of the MMode tactics. In Section 6 we discuss the example proofs that we used to develop the MMode system. Finally, in Section 7 we describe the extension to the MMode syntax for Henk Barendregt's proofs.

In this report we assume familiarity with the Coq system. Basic knowledge of the Mizar proof language also helps. For an introduction to Mizar-style proof read the introduction to the MMode language in Section 3.2 on page 11.

2 The MMode system

We now describe the organization of the MMode distribution. If you do not want to experiment with MMode but just want to read about it, you can skip this section.

The MMode system has four variants:

MMode. This is the basic version of the system, which does not need anything besides the standard Coq library. To use it one needs to add one line:

```
Require MMode.
```

at the beginning of the Coq file.

CMode. This is the version of the system to be used with the C-CoRN library from Nijmegen [5]. It knows about the C-CoRN equality `[=]` and about the C-CoRN equational reasoning tactics `Algebra` and `Rational`. Also, it knows about the C-CoRN type for computational propositions which is called `CProp`.

To use the CMode variant of MMode, one adds the line:

```
Require CMode.
```

at the beginning of one's Coq file.

HMode. This is a version of the system to be used with Henk Barendregt's files in which he proved the Fundamental Theorem of Arithmetic. The reason that this needs a special version of MMode, is that Henk does not use the normal Leibniz equality of Coq, but defines his own equality. Also he has his own types for propositions called `prop` and `cprop`.

To use the HMode variant of MMode, one adds the line:

```
Require HMode.
```

at the beginning of one's Coq file.

HMode with synonyms. The HMode version of MMode has a variant where many synonyms have been added for the MMode steps, to make proofs look more like normal mathematical text. See Section 7 for a description of these synonyms.

To use the HMode variant of MMode with the synonyms, one adds the line:

```
Require HMode_synon.
```

at the beginning of one's Coq file.

Normally MMode will stop processing a proof at the first error. This is the normal behavior of Coq, but it differs from the behavior of Mizar which uses error recovery to continue checking after errors. If one uses the **Sketch** option of MMode, it will behave like Mizar in this respect: in that case justification errors will not be fatal. See Section 5.12 for a description of this feature. To use the Sketch option of MMode, one adds the line:

```
Require Sketch.
```

at the beginning of one's Coq file.

One does not need to have the MMode files next to the Coq files that make use of them. If the MMode files have been properly installed Coq will be able to find the relevant `.vo` files automatically in the Coq library directory. If the MMode files have not been installed, but are used from their source directory, one needs to add the flag

`-R` ⟨*directory that holds the* MMode `.vo` *files*⟩ `Coq.mmode`

to the Coq commands.

To install the MMode system, one needs to perform the following six steps:

1. Get the MMode distribution file from

   `http://www.cs.kun.nl/~freek/mmode/mmode.tar.gz`

2. Unpack the `mmode.tar.gz` file to a directory `mmode`:

   ```
   zcat mmode.tar.gz | tar xf -
   cd mmode
   ```

3. Patch the Coq source and recompile Coq (see below for an explanation of why this is necessary):

   ```
   patch -d ⟨your Coq source directory⟩/tactics < mmode.patch
   cd ⟨your Coq source directory⟩
   make world
   make install
   cd -
   ```

4. The next steps has three variants. Choose one of them:
   (a) Edit the `Makefile` according to whether or not you have C-CoRN (follow the instructions at the start of the file).
   (b) If you have C-CoRN but do not want to edit the `Makefile`, create a link called `algebra` in the `mmode` directory that points to your C-CoRN directory:

       `ln -s` ⟨*your* C-CoRN *directory*⟩ `algebra`

   (c) If you do not have C-CoRN, but want to try it, unpack the file `algebra.tar.gz` and compile it:

       ```
       zcat algebra.tar.gz | tar xf -
       cd algebra
       make
       cd ..
       ```

       The file `algebra.tar.gz` contains the part of C-CoRN that is used by the C-CoRN examples.

   In case you choose 4(a) or 4(b), be aware that you need to have a version of C-CoRN that works with Coq 7.4. (The most recent version of C-CoRN does not work with Coq 7.4 anymore.)
5. Type:

   ```
   make
   ```

   This will compile the MMode files and the examples.
6. Type:

   ```
   make install
   ```

   This will copy the MMode `.vo` files to the library directory of Coq. (It will use the command `coqc -where` to find out where this is.) It also will copy Henk's files that are needed for the HMode.

As step 3 of the installation, Coq 7.4 needs to be 'patched'. This is needed to be able to process the `Grammar` rules in the MMode files. Two lines in the Coq source file `tacinterp.ml` will be changed. This will allow the `Grammar` rules to define syntax for tactics that take tactics as arguments.[5] *The only change will be*

[5] To understand why MMode has tactics that take tactics as arguments, consider the MMode step:

```
Have (3)=(plus (3) (0)) [by plus_n_O].
```

The subgoal `(3)=(plus (3) (0))` will be solved by the tactic '`by plus_n_O`'. Both the full '`Have ...`' step itself, as well as the '`by plus_n_O`' argument to the `Have` tactic are Coq tactics.

This also means that the same step could have used a different Coq tactic for its justification. For instance it could have been:

```
Have (3)=(plus (3) (0)) [Omega].
```

that slightly more `Grammar` *rules will be accepted by Coq: anything that worked with Coq 7.4 will keep working in exactly the same way as it worked before.* So one can patch Coq without any fear for compatibility problems.

The MMode distribution contains the following files:

`mmode.patch`	patch to allow Coq 7.4 to process the MMode files
`src/`	source files of the MMode tactics
`other/henk/`	Coq files needed for the HMode
`algebra.tar.gz`	relevant part of the C-CoRN files for the CMode
`examples/`	examples of MMode proofs
`other/originals/`	the original versions from which the examples have been derived
`other/expanded_by/`	the examples where the **by** justifications have been expanded to more common Coq tactics
`README`	an ASCII file similar to this section
`paper/`	LaTeX source of this report

Apart from files for the four variants of MMode, the `src` directory also contains a file called '`BMode.v`' (the 'basic' part of MMode). This is the common part of MMode, CMode and HMode. It is not meant to be used by itself.

3 The MMode proof language

3.1 MMode syntax

In this section we will describe the proof language of the MMode system. The description will be from the point of view of a MMode user. For a more detailed description, which also discusses the implementation of the MMode tactics, see the next sections.

The MMode proof language is given by the following grammar. The MMode tactics do *not* implement this language completely. For a list of the actual steps that one can use in the current MMode system, see the Appendix on page 53. However, in this section we will describe the MMode language as if it were fully implemented:

typing := *var* { '**,**' *var* } '**be**' *type*
typings := *typing* { ('**,**' | '**and**') *typing* }

bound-formula :=
  '`[`' *var* '`:`' *type* '`]`' *formula*
| '`(`' *bound-formula* '`)`'

proposition := *formula* ['`(`' *label* '`)`']
propositions := *proposition* { '**and**' *proposition* }

bound-proposition := *bound-formula* ['`(`' *label* '`)`']
bound-propositions := *bound-proposition* { '**and**' *bound-proposition* }

justification :=
  ['`[`' '**by**' *ref* { '`,`' *ref* } ['**with**' *Coq-tactic*] '`]`']
| '`[`' *Coq-tactic* '`]`'

step :=
  '**Let_**' *typings* '`.`'
| '**Assume**' *propositions* '`.`'
| ('**Have**' | '**Then**') *proposition justification* '`.`'
  { ['**Thus**'] ('`_=`' | '`_[=]`' | '`=_`' | '`[=]_`') *term justification* '`.`' }
| '**Claim**' *proposition* '`.`'
    proof
  '**End_claim**' '`.`'
| ('**Thus**' | '**Hence**') (*formula* | '**thesis**') *justification* '`.`'
| ('**Consider**' | '**Then**' '**consider**') *var* '**such**' '**that**' *bound-propositions*
    justification '`.`'
| '**Take**' *term* '**and**' '**prove**' *formula* { '**and**' *formula* } '`.`'
| '**Show_**' *formula* '`.`'
| *Coq-tactic* '`.`'

cases :=
  '**First**' '**case**' *proposition* '`.`' *proof* { '**Next**' '**case**' *proposition* '`.`' *proof* }
  '**End_cases**' *justification* '`.`'

proof := { *step* } [*cases*]

In this grammar the notions *var*, *term*, *type*, *formula*, *label*, *ref* and *Coq-tactic* are taken to be primitive. These notions are inherited from Coq. The notions *var* and *label* are Coq identifiers, the notions *term*, *type*, *formula* and *ref* are Coq terms, and the notion *Coq-tactic* is for Coq tactics.

In comparison to the Mizar proof grammar on page 3, the most important differences in this grammar are:

- The MMode grammar has a special step called **Show_**, to state the current goal. This does not do anything (maybe this is the reason that Mizar does not have an equivalent step), it just documents the proof obligation in the proof.
- In the MMode grammar, justifications are only '**by**' justifications and not multi-step subproofs. To have subproofs one uses a special tactic called **Claim**. In contrast to this, in the grammar on page 3 a subproof can be used in exactly the same places as where a **by** justification can be used. (The real Mizar is in between these two extremes: in Mizar subproofs are not allowed in *all* justifications, but there is not a special keyword for them either.)
- In MMode one can have arbitrary Coq proof terms after the **by**. In Mizar one can only put labels there.

- There is no step in MMode that corresponds to Mizar's abbreviation step called '**set**'. In Coq there already exists the tactic **LetTac** for this.
- There are some other small differences between Mizar and MMode. In Mizar the keyword **thesis** can be used anywhere to represent the current goal, while in MMode it only can be used after **Thus** or **Hence**. In Mizar the formula after a **Thus** or **Hence** can have a label, in MMode it can not. In Mizar after a **take** step there is no statement of what the thesis/goal has become, in MMode there is (so the MMode '**Take**' is a combination of the Mizar **take** and the MMode **Show_**). In Mizar the justification in a proof by cases ('**per cases**') is at the start of the cases, in MMode it is at end (for only then it is known what the cases are).

All these differences are minor. The MMode grammar is very close to the Mizar proof grammar from page 3, and it therefore is justified to call MMode a 'Mizar mode'.

Both MMode steps and MMode justifications can be arbitrary Coq tactics. Therefore it is possible to freely mix 'traditional style' Coq tactics and MMode steps. For instance one might decide to do a proof by induction by just running Coq's **Induction** tactic, and then handle the cases that this tactic generates using MMode.

3.2 A brief introduction to MMode proofs

We now will go through a Coq session, to explain the basics of MMode proofs.

We start by typing:

```
Require Le.
Check le_trans_S.
```

Coq will answer:

```
le_trans_S
     : (n,m:nat)(le (S n) m)->(le n m)
```

Apparently the Coq lemma `le_trans_S` states that for all natural numbers n and m holds that if $n + 1 \leq m$ then also $n \leq m$.

We now will prove the equivalent statement for `lt` in various ways. This is of course completely trivial, but it will show the relation between the Coq way of doing proofs and MMode style proofs.

First, here is a low level Coq proof:

```
Lemma lt_trans_S_coq_simple : (n,m:nat)(lt (S n) m)->(lt n m).
Proof.
 Intros n m A1.
 Change (le (S n) m).
 Apply le_trans_S.
 Exact A1.
Qed.
```

Of course the proof can be done automatically as well:

```
Lemma lt_trans_S_coq_auto : (n,m:nat)(lt (S n) m)->(lt n m).
Proof.
 Auto with arith.
Qed.
```

(In the `Arith` library of Coq all lemmas like this are proved in this style.)

We will now prove the same statement using MMode. Again we start in a very basic way:

```
Lemma lt_trans_S_mmode_simple : (n,m:nat)(lt (S n) m)->(lt n m).
Proof.
 Let_ n,m be nat.
 Assume (lt (S n) m) (A1).
 Have (le (S (S n)) m) (A2) [by A1].
 Have (le (S n) m) (A3) [by A2,le_trans_S].
 Thus (lt n m) [by A3].
Qed.
```

Before we go through this line by line, note that this proof uses four kinds of steps: **Let_**, **Assume**, **Have** and **Thus**. These steps correspond in Coq to the tactics **Intro**, **Intro**, **Cut**/**Assert** and **Exact**. (Of course there are more MMode tactics than just these.[6] Below we will list them, and relate them to Coq tactics, and also to the natural deduction rules of first order logic.)

Note that there is no step that corresponds to **Apply**. The role of **Apply** is taken by the **by** justification.

Now let us look at this example proof in detail. Each line contains a *formula* that it will introduce to the context, it also contains a *label* between round brackets, and most of the lines contain a *justification* between square brackets.

We can execute the proof step by step. After all, each step is just a completely ordinary Coq tactic:

```
  ============================
   (n,m:nat)(lt (S n) m)->(lt n m)

lt_trans_S_mmode_simple < Let_ n,m be nat.
1 subgoal

  n : nat
  m : nat
  ============================
   (lt (S n) m)->(lt n m)

lt_trans_S_mmode_simple < Assume (lt (S n) m) (A1).
1 subgoal
```

[6] Also **Thus** does more than **Exact**, because it can split conjunctions: it corresponds to the natural deduction rule of $\wedge$-introduction.

```
  n : nat
  m : nat
  A1 : (lt (S n) m)
  ============================
   (lt n m)

lt_trans_S_mmode_simple < Have (le (S (S n)) m) (A2) [by A1].
1 subgoal

  n : nat
  m : nat
  A1 : (lt (S n) m)
  A2 : (le (S (S n)) m)
  ============================
   (lt n m)

lt_trans_S_mmode_simple < Have (le (S n) m) (A3) [by A2,le_trans_S].
1 subgoal

  n : nat
  m : nat
  A1 : (lt (S n) m)
  A2 : (le (S (S n)) m)
  A3 : (le (S n) m)
  ============================
   (lt n m)

lt_trans_S_mmode_simple < Thus (lt n m) [by A3].
Subtree proved!
```

This example session shows the relation between the formulas and labels in the MMode steps, and how these steps affect Coq's proof state.

Most of the references in this proof are to the label of the previous proof step. This is so common in Mizar-like proofs, that there is a special abbreviation for this. If one writes **Then** instead of **Have** or **Hence** instead of **Thus**, there is an implicit reference to the previous proof step. Another abbreviation is that the statement in the final step can be abbreviated as **thesis**. Together this leads to a low level MMode proof of our example in a for Mizar more customary style:

```
Lemma lt_trans_S_mmode_abbrev : (n,m:nat)(lt (S n) m)->(lt n m).
Proof.
 Let_ n,m be nat.
 Assume (lt (S n) m).
 Then (le (S (S n)) m).
 Then (le (S n) m) [by le_trans_S].
 Hence thesis.
Qed.
```

So in MMode labels are only necessary when the reference is to a step that is not the previous one.

In MMode the **by** tactic has a similar power to the **Auto** tactic of Coq. Therefore, we do not need to prove this statement in this low level manner, but instead can just write:

```
Lemma lt_trans_S_mmode_by : (n,m:nat)(lt (S n) m)->(lt n m).
Proof.
 Thus thesis [by le_trans_S].
Qed.
```

This completes our brief introduction to MMode proofs.

For less trivial MMode proofs see the examples in Sections 6 and 7. Realize that even in the example with synonyms starting on page 46 (which looks like a 'presentation' of a proof) the text still can be executed one tactic at a time. For instance after the line:

```
   Also ((x[x]pp)=n) [by A8 , times_com].
```

the goal state of Coq will look like:

```
  P := [n:nat]O<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
  IH : (before n P)
  A4 : O<n
  pp : nat
  A5 : (primediv pp n)
  x : nat
  A7 : O<pp/\x[x]pp=n
  A8 : pp[x]x=n
  A9 : n=pp[x]x
  A10 : x=O
  A11 : x[x]pp=O
  _ : x[x]pp=n
  ============================
   O=n

subgoal 2 is:
 FF
subgoal 3 is:
 O<x
subgoal 4 is:
 (EX L:ListN|(allprimes L)/\n=(Prod L))
subgoal 5 is:
 ((prime n)\/~(prime n))\/(P n)
```

Although these proofs do not *look* like Coq tactic scripts, they *are*.

3.3 Coq equivalents of MMode tactics

There are various ways to explain the meaning of the MMode steps. In Section 5 we will discuss them in detail, but first for brevity we will just *list* the MMode steps and list how the same effect is obtained using traditional Coq tactics:

by	tactics like **Auto**/**EAuto**, **Prolog** and **Intuition**
Let_	**Intro**
Assume	**Intro**
Have/**Then**	**Cut**/**Assert**
Claim	**Cut**/**Assert**
Thus/**Hence**	**Exact**, possibly preceded by a **Split** of a conjunction in the goal
Consider	**Elim**, of an existential statement
Take	**Exists**
cases	**Elim**, of a disjunction
Show_	this does nothing (it just uses **Change** to verify the current goal)

This table is not exact,[7] it just is meant to give a rough idea of what the various MMode constructions are for.

Note that this table is very similar to the list of Mizar Light tactics in [20]. The differences are that here there is a distinction between **Have** and **Claim** (both are equivalent to the tactic '`A`' in [20]), and that in [20] there is no equivalent of **Show**_.

3.4 MMode steps and natural deduction

In this report, the MMode language will only be described in terms of the Coq system. However it is possible to give a more system independent account of the MMode language. Most of the semantics of this language is system (and even logic) independent.

The only feature of MMode that does not have a 'natural' meaning is the **by** tactic. Apart from that the language is just natural deduction and makes sense for any logic that has the connectives $\neg$, $\rightarrow$, $\wedge$, $\vee$, $\forall$ and $\exists$.

In this report we will not describe the meaning of the MMode language in a system independent way (for such an account of a language similar to MMode, see [19]). We will just list the correspondence between the natural deduction rules for the logical connectives and the corresponding MMode constructions:

[7] In particular **by** is more general than the tactics mentioned here; see Section 4 below for a precise description.

$\bot$ elimination	(**by**)
$\neg$ introduction	**Assume**
$\neg$ elimination	(**by**)
$\rightarrow$ introduction	**Assume**
$\rightarrow$ elimination	(**by**)
$\wedge$ introduction	**Thus**/**Hence**
$\wedge$ elimination	(**by**)
$\vee$ introduction	(**by**)
$\vee$ elimination	*cases*
$\forall$ introduction	**Let**_
$\forall$ elimination	(**by**)
$\exists$ introduction	**Take**
$\exists$ elimination	**Consider**

An entry of '(**by**)' in this table means that the **by** justification is used to reason using this rule. For instance application of the rule of $\rightarrow$-elimination would be written like:

```
..............
Have P->Q (A1) [by ...].
Have P (A2) [by ...].
..............
Have Q [by A1,A2].
..............
```

The six entries in this table marked '(**by**)' are a lower bound on the power of the **by** tactic. The **by** of the Mizar system and the **by** of MMode satisfy this bound. Both of these **by** tactics have the property that they include these six natural deduction rules.

4 By

The MMode language (and other declarative proof languages like Mizar and Isar) combines two aspects:

- natural deduction proofs in a block structured language;
- *automation* of inferences for forward reasoning.

The set of tactics that are used for the first will be explained in detail in Section 5. In this section we will discuss the **by** tactic, which in the MMode system is the implementation of automated forward reasoning.

Most steps in both the Mizar and the MMode systems generate a *proof obligation* that has to be proved by a *justification*. For instance the MMode step

```
Consider m such that [m:nat] n=(plus m (1)) [tactic].
```

has as proof obligation

```
(EX m:nat | n=(plus m (1)))
```

This is the statement that *tactic* should completely prove.

In the Mizar system there is only one 'tactic' that is used to prove justifications like this, which is called **by**.[8] This prover has the following three properties:

1. It is *very fast*. Either it almost immediately succeeds in proving the proof obligation or it almost immediately fails. So in both cases it returns the answer in a very short time. On modern hardware Mizar can prove thousands of **by** justifications in seconds.
2. It is *complete*, in the sense that anything that can be proved in the logic of the underlying system can be proved using the combination of natural deduction steps and **by**. For Mizar this is easy to establish, cf. the discussion in Section 3.4. The MMode **by** preferably also should be complete for the Coq logic. We conjecture that for the current implementation this is the case, but we do not have a proof of this yet.
3. It is sufficiently *powerful*, in the sense that the **by** prover can establish inferences that a human user considers to be trivial. The justifications that the Mizar **by** proves are in the Mizar literature called *obvious* inferences.

Note that there is a tension between property 1 and property 3.

As MMode is a Mizar mode, we have been looking for a 'natural' implementation of a **by** prover for the Coq system. We have gone through three approaches:

- Coq already has automated proof search tactics: **Auto**, **EAuto**, **Prolog**, **Intuition** and **Jp** ('JProver'). As a first approximation, it seems natural to just use one of those tactics as the Coq version of **by**. In particular **Intuition** seems the natural choice for a Coq version of 'obviousness'.
  However, in practice this does not come close enough to the 'completeness' and 'powerful' requirements of **by**. Often a Coq proof will need tactics like **Induction** or **Inversion** to proceed naturally. This kind of reasoning will not be captured by one of the already automated search tactics of Coq.
- For a second attempt at a Coq **by** we reasoned as follows: Most Coq tactics can be run either with no arguments, or with a Coq term for an argument. Coq proofs mostly are a linear list of such zero- or one-argument tactic invocations. We want **by** to search for short proofs of this shape.
  This method generalizes the **Auto** tactic, which searches for short proofs that *only* use the **Intro** and **Apply** tactics. Therefore we called our attempt **GAuto** ('Generalized Auto').
  The **Auto** tactic tries to prove the goal by applying the **Apply** tactic to:
  1. items from *hints* databases;
  2. items from the context.

[8] This is not completely true. Mizar proof steps sometimes use the **from** justification as well, which is a very weak second order prover. Here we will ignore this, and just talk about '**by**'.

The idea of **GAuto** was to modify this, by dropping the hints sets (so this makes the tactic weaker), but to use other tactics apart from **Apply** (this makes the tactic stronger). So apart from **Apply**, the **GAuto** tactic will recursively try to apply tactics like **Absurd**, **Rewrite ->**, **Rewrite <-**, **Elim**, **Induction** and **Inversion** to variables from the context. Also **GAuto** will try tactics without arguments other than **Intro**, like **Split**, **Simpl** and **Red**.
One often observes that when people start learning Coq they manage to do proofs without any real understanding, by just trying tactics until the goal is solved. This approach to Coq proof finding is what **GAuto** is automating.

As an experiment we also put 'normal' Coq tactic scripts as justifications in the MMode examples. These variant proofs can be found in the directory `other/expanded_by/` of the MMode distribution. For instance, the proof of `nat_factorizes` will work with the following justifications:

```
Simpl; Apply conj; [ Apply A1 | Apply trivial]
Simpl; Rewrite -> times_com; Apply refl_equal
Apply conj; [Apply A2 | Apply A3]
Apply nat_HPD
Red in A5; Elim A5; Intro; Intro B1; Exact B1
Exact _
Elim A4; Intro; Intro B2; Rewrite <- B2; Apply times_com
Rewrite A9; Apply refl_equal
Red in A5; Elim A5; Intro B3; Intro; Exact B3
Red in _; Elim _; Intro B4; Intro; Exact B4
EApply propprod_propfact; [Apply A7 | Apply A6 | Exact _]
Rewrite A8; Simpl; Apply refl_equal
Rewrite <- A9; Apply times_com
Rewrite <- _ in A6; Exact A6
Elim not_lt_zero with 0; Exact _
Apply neq_zero_imp_gt_zero; Exact _
Apply A10; Elim A11; Intro B5; Intro; Exact B5
Elim A11; Intro; Intro B6; Exact B6
Apply A12; Exact _
Elim A5; Intro B7; Intro; Exact B7
Split; [Exact _ | Elim A13; Intro B8; Intro; Exact B8]
Elim A4; Intro; Intro B9; Apply eq_sym; Exact B9
Elim A13; Intro; Intro B10; Rewrite B10; Simpl; Apply times_com
Split; [Exact A14 | Exact A15]
Apply prime_dec
Apply cv_ind; Exact A16
```

(In these proof scripts '`_`' is the label used by MMode for an unlabeled formula that was put in the context by the previous proof step.)
The same justifications using the **by** tactic are:

```
by A1
by refl_equal, times_com
by A2, A3
```

```
by nat_HPD
by A5
by _
by times_com, A4
by A9
by A5
by _
by _, A7, A6, propprod_propfact
by A8
by A9, times_com
by _, A6
by _, not_lt_zero
by _, neq_zero_imp_gt_zero
by A11, A10
by A11
by _, A12
by A5
by _, A13
by A4, eq_sym
by A13, times_com
by A14, A15
by prime_dec
by A16, cv_ind
```

(Of course in MMode proofs the label '_' is not actually written. Instead the **Then** keyword is used for this.)

Coq's **Auto** tactic tries to use everything that is in the context. However, the **by** tactic only uses its arguments.[9] To accomplish this, **by** first replaces the context with a context that just contains variables corresponding to the arguments. It does this by calling **Generalize** for all its arguments, then clearing the context, and finally putting them in the context again by calling `Try Intros`. For instance, in the example proof from Section 3.2, when justifying the step:

```
Have (le (S n) m) (A3) [by A2,le_trans_S].
```

at the start of the justification the context looks like

```
n : nat
m : nat
A1 : (lt (S n) m)
A2 : (le (S (S n)) m)
============================
 (le (S n) m)
```

[9] There are two reasons for this: (1) efficiency and (2) to make the 'flow of reasoning' in the proof apparent. It might be interesting to experiment with a version of MMode where **by** is allowed to use everything from the context (instead of just its arguments). That way the proofs will probably check much slower, but also much less references to 'local' labels will be needed.

After

```
Generalize A2; Generalize le_trans_S
```

the goal becomes

```
n : nat
m : nat
A1 : (lt (S n) m)
A2 : (le (S (S n)) m)
============================
 ((n,m:nat)(le (S n) m)->(le n m))->(le (S (S n)) m)->(le (S n) m)
```

Then the context is cleared, so the goal becomes

```
n : nat
m : nat
============================
 ((n,m:nat)(le (S n) m)->(le n m))->(le (S (S n)) m)->(le (S n) m)
```

and then after

```
Try Intros.
```

the goal becomes

```
n : nat
m : nat
H : (n,m:nat)(le (S n) m)->(le n m)
A2 : (le (S (S n)) m)
============================
 (le (S n) m)
```

Only then **GAuto** is run.

– The actual implementation of **by** that is currently in MMode is *in between* the two previous approaches. Basically it implements the previous method, but it does not recursively look for arbitrary proof scripts up to a certain length. Instead it only tries all the relevant Coq tactics for *one* step. Before and after this one step it uses the automation of Coq. The two automated tactics that it uses there are

```
Intuition EAuto
Prolog [] 9
```

The tactics that it tries between these calls to **Intuition** or **Prolog** are

```
Elim term
Rewrite -> term
Rewrite <- term
Absurd term
Red in term
Rewrite term in term
Rewrite <- term in term
```

(This list works for the example proofs. Possibly it needs to be extended when more experience with MMode is gained.) Apart from the tactics in this list, the **by** tactic can be given an extra tactic after the keyword **with**. It will then try this tactic as well.

5 Implementation of the tactics

We will now describe the 'natural deduction' MMode tactics. For each tactic we will describe its effect on the proof state, when it is appropriate to use it, and how it has been implemented.

5.1 Let_/Assume

'**Let_**' *var* { '**,**' *var* } '**be**' *type* { ('**,**' | '**and**') *var* { '**,**' *var* } '**be**' *type* } '**.**'
'**Assume**' *formula* ['**(**' *label* '**)**'] '**.**'

The **Let_** tactic applies to a goal which is a dependent product '`(x:U)T`'.

```
Let_ x be U
```

puts '`x:U`' in the local context and the new subgoal becomes '`T`'.
The **Assume** tactic applies to a goal which is a non-dependent product '`T1 -> T2`'.

```
Assume T1 (H1)
```

puts '`H1:T1`' in the local context. If instead of '`Assume T1 (H1)`' we apply

```
Assume T1
```

then '`_:T1`' is put in the local context. In both cases the new subgoal becomes '`T2`'.
We can use both tactics with a list of arguments. For example, if a goal is of the shape

```
(x,y:U1;z:U2)T1 -> T2 -> T3
```

we can then apply

```
Let_ x,y be U1 and z be U2.
Assume T1 (H1) and T2.
```

to add `x`,`y`,`z`,`T1`,`T2` to the local context and we get `T3` as the new subgoal.
Let_ and **Assume** are also applicable to goals of which the head is a defined constant.
For example:

```
Coq < Lemma example: (Included U A B).
1 subgoal

  U : Type
  A : (Ensemble U)
  B : (Ensemble U)
  ============================
   (Included U A B)

example < Let_ x be U.
1 subgoal

  U : Type
  A : (Ensemble U)
  B : (Ensemble U)
  x : U
  ============================
   (In U A x)->(In U B x)
```

where the definition of `Included` is

```
Definition Included : Ensemble -> Ensemble -> Prop :=
   [B, C: Ensemble] (x: U) (In B x) -> (In C x).
```

The implementation:
The tactics **Let_** and **Assume** are defined with 3 tactics:

- **Match Context With** which enables the user apply **Let_** only to dependent products and **Assume** only to non-dependent products;
- **Intro** which adds a variable or an assumption to the local context;
- **Change** which checks the correctness of arguments, i.e., whether what the user typed corresponds to the goal. In the case of **Let_**, this is the type of a variable, in the case of **Assume**, this is the assumption.

5.2 Have

'**Have**' *formula* ['`(`' *label* '`)`'] ['`[`' *tactic* '`]`'] '`.`'

The tactic is applicable to any goal. It adds the hypothesis *formula* to the local context with a name *label*. The tactic *tactic* is to prove this hypothesis. For example:

```
1 subgoal

  ...................
```

```
  H : (In U A x)
  H0 : (Included U A B)
  ...................
  ============================
   True

example < Have (In U B x) (H2) [by H, H0].
1 subgoal

  ................
  H : (In U A x)
  H0 : (Included U A B)
  ................
  H2 : (In U B x)
  ============================
   True
```

The implementation:

Have applies **Cut** *formula*. Then it applies **Intro** to the first subgoal and *tactic* to the second one.

5.3 Iterated equality

['**Thus**'] ('`_=`' | '`_[=]`' | '`=_`' | '`[=]_`') *term* ['`[`' *tactic* '`]`'] '`.`'

In mathematics one often writes chains of equalities, like for example:[10]

$$n = (n - d) + d = dq + r + d = d(q + 1) + r$$

MMode (like Mizar) has the possibility to write this kind of 'iterated equalities'. In MMode this example becomes (the lines are taken from the HMode example called `euclid.v`):

```
Have   n = ((n-,d)[+]d)     [by ge_imp_mon_plus_eq, A5].
        _= (d[x]q[+]r[+]d)  [by A8].
        _= (d[x](S q)[+]r)  [by compute].
```

Note that this one iterated equality turns into three MMode tactics, of which the first is **Have** and the other two are '`_=`'.

In order to do an iterated equality first we apply the **Have** tactic to add an initial equality to the local context. Then the tactic `_=` or `=_` replaces the right

[10] In the second step this uses that $n - d = dq + r$.

hand side or left hand side of the equality, respectively with *term*. The *tactic* is to prove the equality of the *term* and the expression we want to replace.
We apply '**Thus** `_=`' or '**Thus** `=_`' if a goal and the last hypothesis in the local context are equalities. Suppose that a goal is the equality '`a = c`', and '`a = b`' is the last hypothesis in the local context. Then to prove the goal we apply '`Thus _= c [tactic]`' , where as the argument `tactic` we need to put a tactic that solves '`b = c`'.
The tactics '`_[=]`', '`[=]_`' were defined to do iterative equalities in C-CoRN. They work in an analogous way to '`_=`' and '`=_`', respectively.

The implementation:

We explain how the tactic '`_=`' works. The tactic '`=_`' is defined in an analogous way.
Suppose that the last hypothesis in the local context is the equality `a = b` and we apply `_= c [tactic]`. Then, the following steps are performed:

- `Cut (a = c)`
- The equality `a = c` is added to the local context of the first subgoal by **Intro**. The equality `a = b` is removed from that context.
- The theorem about transitivity of equality is applied to the second goal. This generates two subgoals: `a = b` and `b = c`. As the last hypothesis in the context is `a = b`, the first subgoal is solved by **Exact**. The second one is solved by the argument `tactic`.

5.4 Claim

'**Claim**' *formula* ['`(`' *label* '`)`'] '`.`'
proof
'**End_claim**' '`.`'

It adds, like **Have**, the hypothesis *formula* with the name *label* to the local context. The *proof* is a proof of this hypothesis.
Suppose that the goal is the formula `G` and we need the formula `A` to prove it. Then we can apply `Claim A (H1)`.

```
1 subgoal
...
=============
G

Claim A (H1).

3 subgoals
...
```

```
=============
A

A

G
```

It will generate `A` as a subgoal twice. After having proved the first subgoal, the formula `A` will be added to the local context. Then we need to apply **End_claim** tactic that will prove the second one[11]

```
2 subgoals
...
H1:A
=============
A

G

End_claim.

1 subgoal
...
H1:A
=============
G
```

Therefore we have `A` in the local context and we continue the proof of `G`.
The reason why the *formula* is generated as a subgoal twice is to improve the readability of a proof script. **End_claim** indicates the end of the proof of the *formula* in the proof script.

5.5 Thus thesis

'**Thus**' '**thesis**' '`[`' *tactic* '`]`' '`.`'

The tactic is applicable to any goal. We use this tactic to finish a proof, i.e., to do the last step. As the argument *tactic* we put a tactic that proves the goal.
Example:

```
  ..............
  A8 : x0 E X
  A10 : ~x0 E X
```

[11] After having proved `A`, it will be added to the local context with a label generated by Coq. After applying **End_claim** the right label (the label we put as an argument of **Claim**) will be assigned to `A`.

```
  ..............
  ============================
   x0 E A

Included_Add < Thus thesis [by A8,A10].

Subtree proved!
```

'**Thus**' *formula* '[' *tactic* ']' '.'

The tactic is applicable to a goal that is a conjunction. As the argument *formula* we put a formula that is a part of the conjunction and as the argument *tactic* a tactic to prove this formula. Then the goal is reduced to a conjunction without this formula.
Example:

```
  ............
  d : nat
  n : nat
  A2 : n<d
  ............
  ============================
   n<d/\n=d[x]zero[+]n

subgoal 2 is:
 n<d\/(P n)

Euclid < Thus (n<d) [by A2].
2 subgoals
  ............
  d : nat
  n : nat
  A2 : n<d
  ............
  ============================
   n=d[x]zero[+]n

subgoal 2 is:
 n<d\/(P n)

Euclid < Thus (n=d[x]zero[+]n) [by times_com].
1 subgoal
  ...........
  d : nat
  n : nat
  ...........
```

```
  ============================
   n<d\/(P n)
```

We can only prove one formula at a time. The formulas have to proved in the same order as they are in a conjunction statement.

5.6 Consider

'**Consider**' *var* '**such**' '**that**' *bound-formula* ['(' *label* ')']
{ '**and**' *bound-formula* ['(' *label* ')'] } ['[' *tactic* ']'] '.'

This tactic eliminates the existential quantifier. There are four kinds of existential statements with one predicate in Coq:

```
{x:A | (P x)}
{x:A & (P x)}
(EX x | (P x))
(EXT x | (P x))
```

depending on the types of `A` and `P`. As the argument *var* we put a witness, as the argument *bound-propositions* the lambda term `[x:A](P x)`, as the argument *label* the name we want to give for the hypothesis `(P x)`, and as the argument *justification* a tactic that proves the existential statement we want to eliminate. The tactic adds '*var*`:A`' and '*label*`: (P` *var*`)`' to the local context.
For example,

```
  ............
  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  A10: 0<x
  A12 : (P x)
  ............
  ============================
   (EX L:ListN|(allprimes L)/\n=(Prod L))

Coq < Consider LL such that
                [LL:ListN] (allprimes LL)/\x=(Prod LL) (A13) [by A12,A10].

  ............
  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  A10: 0<x
  A12 : (P x)
  ............
  LL : ListN
  A13 : (allprimes LL)/\x=(Prod LL)
  ============================
```

```
(EX L:ListN|(allprimes L)/\n=(Prod L))
```

We use **Consider** in an analogous way in order to eliminate the existential statements with two predicates. There are four kinds of these statements in Coq:

```
(EX x | P(x) & Q(x))
(EXT x | P(x) & Q(x))
{x:A | (P x) & (Q x)}
{x:A & (P x) & (Q x)}
```

To eliminate it, we apply:

```
Consider x such that [x:A](P x) (H1) and [x:A](Q x) (H2) [tactic]
```

It adds '`x:A`', '`H1 : (P x)`', '`H2 : (Q x)`' to the local context. The `tactic` justifies the existential statement we want to eliminate.
There are two more kinds existential statements in the C-CoRN library. We apply **Consider** to eliminate them in the same way.

The implementation:

First the right existential statement is formed from the argument *bound-propositions*. Then **Cut** is called with this statement as an argument. To the first subgoal the following tactics are applied:

- **Intro** in order to add an existential statement to the local context
- **Elim** in order to eliminate this existential statement
- **Intro** in order to add the variable *var* and a predicate to the local context.

To the second one, the tactic *tactic* is applied.
We tried to implement **Consider** as a tactic of grammar:

'**Consider**' *var* '**being**' *type* '**such**' '**that**' *formula label* ['`[`' *tactic* '`]`'] '`.`'

where as the *formula* we would put the application of the *var* to a predicate. For example, instead of typing:

```
Consider x such that [x:A](P x) [tactic].
```

we would type

```
Consider x being A such that (P x) [tactic].
```

This solution is more elegant. But we did not manage to implement it because we cannot use '`(P x)`' as an argument. The variable `x` is not yet in the context when we call **Consider** and Coq does not accept the formula `(P x)` (does not know what to apply to the predicate `P`).

5.7 Take

'**Take**' *term* '**and**' '**prove**' *formula* { '**and**' *formula* } '.'

This tactic is applicable to a goal that is an existential statement or a disjunction statement that includes an existential statement. As the argument *term* we put an object that satisfies the predicate of the existential statement. As the argument *formula* we put the application of the object to that predicate. The goal is reduced to the *formula*.
Example 1

```
1 subgoal

  .........
  n : nat
  .........
  ============================
   (EX r:nat|r<d/\n=d[x]zero[+]r)

Euclid < Take n and prove (n<d/\n=d[x]zero[+]n).
1 subgoal

  ...........
  n : nat
  ...........
  ============================
   n<d/\n=d[x]zero[+]n
```

Example 2

```
1 subgoal
  ................
  ============================
   X =c A\/(EXT A'|X==A' u {x}/\A' =c A)

Included_Add < Take (X-{x})
               and prove (X==(X-{x}) u {x}/\(X-{x}) =c A).

1 subgoal
  ................
  ============================
   X==(X-{x}) u {x}/\(X-{x}) =c A
```

The order that we put two *formula*-arguments, must be the same as the order of the corresponding predicates in the existential statement.

The implementation:

Take performs 3 steps.

- First it tries to apply **Red**. Therefore the user can also apply **Take** to a goal which is not an explicit existential statement.
- Then the tactic '**Apply** *term1* **with** *term2*' is applied. As the *term1* the constructors of the existential statements are used. The tactic is applied repeatedly until it succeeds. Every time another constructor is put. As *term2* the *formula* is put.
- At the end '**Change** *formula*' is applied. It is to check whether the user put the right argument for *formula*

It is possible to apply **Take** to a disjunction statement that includes an existential statement. In that case **Take** is applied one after another to every formula in this disjunction until it comes to this existential formula. If there is more than one existential statement in a goal then it depends on the parameter *formula* to which existential statement **Take** will be applied successfully.
For example, the goal is a disjunction of the form:

```
A1 \/ A2 \/ ... \/ (EX x:T | (P x)) \/ (EX x:T | (Q x)) \/ ... \/ An
```

Then '`Take` *var* `and prove (P` *var* `)`' will succeed on applying to `(EX x:T | (P x))` and '`Take` *var* `and prove (Q` *var* `)`' will succeed on applying to `(EX x:T | (Q x))`

5.8 proving by cases

'**First**' '**case**' *formula* ['`(`' *label* '`)`'] '`.`'
proof
{ '**Next**' '**case**' *formula* ['`(`' *label* '`)`'] '`.`' *proof* }
'**End_cases**' ['`[`' *tactic* '`]`'] '`.`'

We start to prove by cases with the tactic **First case**. This tactic adds *formula* to the local context with the label *label*. It also generates another subgoal that is a disjunction statement of the *formula* and the first subgoal. For example:

```
  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
  ============================
   (P n)

nat_factorizes < First case (prime n) (A1).

  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
```

```
  A1 : (prime n)
  ============================
   (P n)

 subgoal 2 is:
 (prime n)\/(P n)
```

What kind of disjunction is generated depends on the types of the first subgoal and the *formula*.
After having proved the first case, we have as a goal the disjunction statement of the first case formula and the goal formula. In our example it will be:

```
 (prime n)\/(P n)
```

We consider the next case with the tactic **Next case**. It adds the argument *formula* to the local context. The first goal is again the goal which we prove by cases and the second one the disjunction statement formed with the first case formula, the next case formula and the goal formula. For example:

```
  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
  ..............
  ============================
   (prime n)\/(P n)

nat_factorizes < Next case ~(prime n) (A1).

  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
  ..............
  A1 : ~(prime n)
  ============================
   (P n)

 subgoal 2 is:
 ((prime n)\/~(prime n))\/(P n)
```

To consider next cases, we apply **Next case**. Every time it adds *formula*. The first subgoal becomes the goal that we prove by cases. The second one is a disjunction that is enlarged by *formula*. For example, if we considered four cases `A`, `B`, `C`, `D` then the second subgoal would be '`(A \/ B \/ C \/ D) \/ goal`'. Note that the outermost disjunction is associated to the left.
After all the cases are done, the goal is a disjunction statement generated by the tactics **First case** and **Next case**. We apply **End_cases** to prove it. First **End_cases** applies **Left**. Since the disjunction is associated to the left, the current goal becomes a disjunction of all cases that we considered. Then the tactic

tactic is applied. In other words the *tactic* is to justify that we could split the proof into cases.
In our example there are two cases. After the second one is done, we have:

```
  P := [n:nat]0<n->(EX L:ListN|(allprimes L)/\n=(Prod L)) : nat->Set
  n : nat
  ..............
  ============================
 ((prime n)\/~(prime n))\/(P n)
```

We finish the proof by applying `End_cases [by prime_dec]`. First, the goal is reduced to

```
((prime n)\/~(prime n))
```

by the **Left** tactic and then it is proved by the tactic `by prime_dec`, where `prime_dec` states:

```
prime_dec
     : (n:nat)(prime n)\/~(prime n)
```

The implementation:

The solution that we justify the proof 'by cases' after considering all cases was implemented in order to be able to use the **by** tactic for it. One cannot use **by** at the beginning because there is nothing to be justified then, because the cases we want to consider are not listed yet.
Another solution can be to define a tactic that does not use **by**. For example:

```
Tactic Definition per_cases_by argument_lemma arg1 arg2 ... :=
    Elim argument_lemma with arg1 arg2 ....
```

where as the argument `argument_lemmma` we would put the identifier of the theorem that justifies the proof by cases and as `arg1`, `arg2`, etc. the values for dependent premises of this theorem. But then there is a problem with the number of these premises which varies and the order we should put them in. One more solution might be that instead of **Elim with** we use **Elim** and then as an argument for **Elim** we would put the application of the theorem to these values. We think that the implemented solution is more elegant and is also closer to the Mizar solution.
There are a few kinds of the disjunction statements in Coq depending what the types of the formulas in the given statement are. In order to form the right kind of disjunction the type of the argument *formula* in the tactics **First case** and **Next case** and the type of a goal is checked. We had to define separate tactics in CMode and HMode because in C-CoRN and Henk Barendregt's files new types are introduced: `CProp` and `cprop`.

5.9 Show_

'**Show_**' *formula* '.'

The tactic is a synonym of the tactic **Change**. It was defined to improve the readability of a proof script. We use it if we want to indicate in the proof script what the current goal is. As the argument *formula* we put the current goal. Example:

```
1 subgoal

  ============================
   (n:nat)~n=(S n)

n_Sn < Proof.
n_Sn < Induction n.

2 subgoals

  n : nat
  ============================
   ~(0)=(1)

subgoal 2 is:
 (n0:nat)~n0=(S n0)->~(S n0)=(S (S n0))

n_Sn < Show_ (~(0)=(1)).
2 subgoals

  n : nat
  ============================
   ~(0)=(1)

subgoal 2 is:
 (n0:nat)~n0=(S n0)->~(S n0)=(S (S n0))
```

The proof script looks like:

```
.............
Induction n.
Show_ ~(0)=(1).
.............
```

Therefore, we have information that after the application of **Induction** the goal is '`~(0)=(1)`'.

5.10 Then

The tactics: **Then**, **Then consider**, **Hence thesis** are variants of the tactics **Have**, **Consider**, **Thus thesis**, respectively. The difference is that if we put as the argument *tactic* the **by** tactic then the **by** takes as an argument also the last hypothesis in the local context. For details about **by** see section 4.
Example:

```
2 subgoals
  ..............
  A8 : x0 E X
  A11 : x0 E (A u {x})
  _ : ~x0 E X
  ============================
   x0 E A

 subgoal 2 is:
 (x0 E A\/x==x0)\/x0 E A

Included_Add < Hence thesis [by A8].
1 subgoal

  ..............
  A8 : x0 E X
  A11 : x0 E (A u {x})
  ============================
   (x0 E A\/x==x0)\/x0 E A
```

The **by** tactic took as the arguments the hypotheses '`A8`' and '`_`'.

The implementation:

The tactics with variant '**Then**' apply their mother tactics with the tactic **LINK** as an argument. The tactic **LINK** calls **Generalize** for the last hypothesis in the local context.
Suppose that we have the following context with `G` as a goal:

```
H1: P
...
_: P -> Q
============
G
```

When we apply '`Then Q [by H1]`', the formula `Q` will be generated as the second goal. Then **LINK** will be applied to this subgoal. It will call `Generalize _`. Next **by** will call `Generalize H1`, clear context and call `Try Intros`. Therefore in the context we will have the formula `P -> Q` (the last hypothesis in the context when we called **Then**) and the formula `P` (put as an argument for **by**).

5.11 Dealing with labels

The tactics: **Assume**, **Have**, **Claim**, **Consider**, **First case** and **Next case** have an optional argument called *label*. This is to give a name for the hypotheses which we can add to the context with these tactics.
Example:

```
  ...................
  ============================
   X =c (A u {x})->X =c A\/(EXT A'|X==A' u {x}/\A' =c A)

Included_Add < Assume (X =c (A u {x})) (A1).
1 subgoal
  ...................
  A1 : X =c (A u {x})
  ============================
   X =c A\/(EXT A'|X==A' u {x}/\A' =c A)
```

All the hypotheses in the context must have names. So if one does not specify what name we want to assign to a hypothesis, Coq generates it automatically. Then in the context we have a hypothesis that has a name that is not present in the proof script. While proving the user rather looks at the context and the goals generated by Coq than at the proof script. So it may happen that the user uses such a name as a reference for the **by** tactic and that makes the proof script unreadable.
To avoid this, the hypothesis is added to the local context with the name '_' if the user does not put the argument *label*.
Example:

```
  ...................
  ============================
   X =c (A u {x})->X =c A\/(EXT A'|X==A' u {x}/\A' =c A)

Included_Add < Assume (X =c (A u {x})).
1 subgoal
  ...................
  _ : X =c (A u {x})
  ============================
   X =c A\/(EXT A'|X==A' u {x}/\A' =c A)
```

and the user is aware that such a hypothesis is not labeled in their proof script. Coq does not allow to have the hypotheses with the same names. If the user does not give a name for another hypothesis, the previous one is removed and the new one gets a name '_'. If we want to check whether such and such hypothesis was introduced we can look into the proof script.

5.12 Sketch mode

When we formalize mathematics we often want to prove some facts later, in order to be able to first work on the main proof. In Coq we can use the **Cut** tactic or declare these facts as axioms. In Mizar mode we implemented another solution. It requires to declare `Require Sketch`. Then we can add to the local context a hypothesis without justification. Above this hypothesis an error message is generated.
Example:

```
1 subgoal

  H : P->Q
  ============================
   Q

example < Have P (H1).
1 subgoal

  H : P->Q
  error : inference_not_accepted
  H1 : P
  ============================
   Q
```

So we have `P` in the local context and we can continue our proof.
The error message is also generated when we do not put enough justification. This feature we can use with the tactics: **Have**, **Consider**, '`=_`', '`_=`', **Then**, **Then consider**.
We can only have one error message in the local context. If another unjustified hypothesis is added to the local context then the error message, referring to the previous one, will be removed and the error message will be generated above this hypothesis.
The continuation of the previous example:

```
1 subgoal

  H : P->Q
  error : inference_not_accepted
  H1 : P
  ============================
   Q

example < Have R (H2).
1 subgoal

  H : P->Q
```

```
  H1: P
  error : inference_not_accepted
  H2 : R
  ============================
   Q
```

The error message for the hypothesis `H1` has been removed and the new error message refers to the hypothesis `H2`.
To find the unjustified hypotheses in the local context we need to comment '`Require Sketch`' then Coq will stop checking at the first error.
One can also check whether a proof consists of the unjustified hypotheses by searching the proof term for `missing_proof_of`.
For example, in the proof below we added the formula `(In U X x)` without justification.

```
  x : U
  H : (In U (Intersection U X Y) x)
  ============================
   (In U (Union U X Y) x)

example < Have (In U X x).
1 subgoal

  x : U
  H : (In U (Intersection U X Y) x)
  error : inference_not_accepted
  _ : (In U X x)
  ============================
   (In U (Union U X Y) x)

example < Hence thesis [by Union_introl].
Subtree proved!
```

In the proof term, we can see that the 'axiom' `missing_proof_of` was applied to add the formula `(In U X x)` to the local context:

```
example =
[x:U; _:(In U (Intersection U X Y) x)]
 [H0:=(missing_proof_of (In U X x))]
  [H1:=[_:(In U X x)](Union_introl U X Y x _)](H1 H0)
     : (Included U (Intersection U X Y) (Union U X Y))
```

A little different situation is when we use tactics: **Thus thesis**, **Thus thesis**, **Hence thesis**, '**Thus** `_=`', '**Thus** `=_`'. If our justification is not enough or we do not put any justification, then the error message replaces the current goal. In order to continue the proof we need to apply the tactic **cp**.

```
  H : P
  H0 : P->Q
  ============================
   Q

subgoal 2 is:
 True

test < Thus thesis [by H].
2 subgoals

  H : P
  H0 : P->Q
  ============================
   inference_not_accepted

subgoal 2 is:
 True

test < cp.
1 subgoal

  H : P
  H0 : P->Q
  ============================
   True
```

The implementation:

To 'solve' a goal the following axiom is applied:

```
Axiom missing_proof_of : incomplete.
Definition incomplete := (A:Type)A.
```

Before 'solving' the goal the tactic **Intro** adds to the local context the 'hypothesis' `inference_not_accepted` with the label `error`. The type `inference_not_accepted` is defined as follows:

```
Definition inference_not_accepted := True.
```

6 Examples

We present four proofs in Mizar Mode language.

6.1 Example from the Coq library

The first one is on a fact from the set theory. Standard Coq symbols were replaced by other ones to make the proof closer to mathematical text.

- 'E' denotes a predicate of being an element (`In`) of a set (`Ensemble`)
- '=c' denotes inclusion relation between two sets (`Included`)
- '{ }' denotes a set containing only one element (`Singleton`)
- 'u' denotes union of two sets (`Add`)
- '-' denotes subtraction of two sets (`Subtract`)

```
Lemma Included_Add:
  (X, A: (Ensemble U))(x:U)  (X =c (A u {x})) ->
   (X =c A) \/ (EXT A' | X == A' u {x}  /\ (A' =c A)).
Proof.
  Let_ X,A be (Ensemble U) and x be U.
  Assume (X =c (A u {x})) (A1).
  First case (x E X) (A3).
    Take (X-{x}) and prove (X==(X-{x}) u {x}/\(X-{x}) =c A).
    Have (X =c (X - {x} u {x})) (A4) [by add_soustr_2,A3].
    Have ((X - {x}) u {x} =c X) [by add_soustr_1, A3].
    Hence (X == ((X - {x}) u {x})) [by Dbl_Inc_Eq, A4].
    Let_  x0 be U.
    Assume (x0 E (X - {x})).
    Then ((x0 E X) /\ (~x == x0)) (A6) [by Subtract_inv].
    Then (x0 E (A u {x})) (A10) [by A1].
    First case (x0 E A).
    Hence thesis.
    Next case (x == x0).
    Hence thesis [by A6].
    End_cases [by Add_inv, A10].
  Next case (~(x E X)) (A7).
    Claim (X =c A).
      Let_ x0 be U.
      Assume (x0 E X) (A8).
      Then (x0 E (A u {x})) (A11) [by A1].
      First case (x0 E A).
      Hence thesis.
      Next case (x == x0).
        Then (~(x0 E X)) [by A7].
      Hence thesis [by A8].
      End_cases [by Add_inv,A11].
    End_claim.
  Hence thesis.
  End_cases [by classic].
Qed.
```

The original proof comes from the Coq standard library:

```
Lemma Included_Add:
```

```
  (X, A: (Ensemble U)) (x: U) (Included U X (Add U A x)) ->
   (Included U X A) \/
   (EXT A' | X == (Add U A' x) /\ (Included U A' A)).
Proof.
Intros X A x H'0; Try Assumption.
Elim (classic (In U X x)).
Intro H'1; Right; Try Assumption.
Exists (Subtract U X x).
Split; Auto with sets.
Red in H'0.
Red.
Intros x0 H'2; Try Assumption.
LApply (Subtract_inv U X x x0); Auto with sets.
Intro H'3; Elim H'3; Intros K K'; Clear H'3.
LApply (H'0 x0); Auto with sets.
Intro H'3; Try Assumption.
LApply (Add_inv U A x x0); Auto with sets.
Intro H'4; Elim H'4;
 [Intro H'5; Try Exact H'5; Clear H'4 | Intro H'5; Clear H'4].
Elim K'; Auto with sets.
Intro H'1; Left; Try Assumption.
Red in H'0.
Red.
Intros x0 H'2; Try Assumption.
LApply (H'0 x0); Auto with sets.
Intro H'3; Try Assumption.
LApply (Add_inv U A x x0); Auto with sets.
Intro H'4; Elim H'4;
 [Intro H'5; Try Exact H'5; Clear H'4 | Intro H'5; Clear H'4].
Absurd (In U X x0); Auto with sets.
Rewrite <- H'5; Auto with sets.
Qed.
```

6.2 Example of a HMode proof

The second one is on the fact from the number theory that every natural number can be factorized into a product of primes. The original proof was taken from Henk Barendregt's files.

```
Lemma nat_factorizes :
(n:nat)(O<n)->(EX L:ListN |((allprimes L)/\(n=(Prod L)))).
Proof.
  LetTac P :=[n:nat](O<n)->(EX L:ListN |((allprimes L)/\(n=(Prod L)))).
  Claim ((n:nat)(before n P) -> (P n)) (A16).
  Let_ n  be nat.
  Assume (before n P) (A10).
  First case (prime n) (A1).
    Assume ((O) < n).
    Have (allprimes (cons n nil)) (A2) [by A1].
```

```
    Have n=(Prod (cons n nil)) (A3) [by refl_equal, times_com].
    Take (cons n nil) and prove
          ((allprimes (cons n nil))/\n=(Prod (cons n nil))).
  Thus thesis [by A2, A3].
  Next case ~(prime n).
    Assume ((O) < n) (A6).
    Consider pp such that
      ([pp:nat](primediv pp n)) (A5) [by nat_HPD].
    Then (pp [|] n).
    Then consider x such that ([s:nat] (((O)<pp)/\(s[x]pp)=n)) (A4).
    Claim  (x<n)/\((O)<x) (A11).
      Have (pp[x]x)=n (A9) [by times_com, A4].
      Then n=(pp[x]x) (A7).
      Have (prime pp) [by A5].
      Then  one<pp.
    Hence (x<n) [by A7, A6, propprod_propfact].
      Claim (~x=O).
        Assume (x = O) (A8).
        Have (O = (x [x] pp)) [by A8].
               _= n [by A9, times_com].
        Then (O < O) [by A6].
        Hence thesis [by not_lt_zero].
      End_claim.
    Hence ((O) < x) [by neq_zero_imp_gt_zero].
    End_claim.
    Then (P x) (A12) [by A10].
    Have ((O)<x)  [by A11].
    Then consider LL such that
      [LL:ListN] (allprimes LL)/\x=(Prod LL) (A13) [by A12].
    Have (prime pp) [by A5].
    Then (allprimes (cons pp LL)) (A14) [by A13].
    Have (n = (x [x] pp)) (A15) [by A4, eq_sym].
           _= (Prod (cons pp LL))  [by A13, times_com].
    Take (cons pp LL) and prove
          ((allprimes (cons pp LL))/\n=(Prod (cons pp LL))).
  Thus thesis [by A14, A15].
  End_cases [by prime_dec].
End_claim.
Show_ (n:nat)(P n).
Thus thesis [by A16,cv_ind].
Qed.
```

6.3 Example of a CMode proof

The third one is a proof of the Fundamental Theorem of Algebra. The original proof comes from the C-CoRN library. In two places there is `Let_ f be (cs_crr (cpoly_cring CC))` instead of `Let_ f be (cpoly_cring CC)`. The reason was that `Change (cpoly_cring CC) in f`, which is called by **Let_** after `Intro f` (see 5.1), fails because it does not insert the relevant coercion.

Another thing is that in one of the last steps of the proof the label *Hrecn*, that is not present in the proof script, is used as a reference. This label refers to the induction hypothesis:

```
Hrecn : (f:(cpoly_cring CC))
         (degree_le n f)->(nonConst CC f)->{z:CC | f!z[=]Zero}
```

The problem is that `Induction n` adds this hypothesis to the context automatically and we can not make any indication of it in the proof script.

```
Lemma fta' :
  (n:nat)(f:(cpoly_cring CC))
    (degree_le n f) -> (nonConst ? f) -> z:CC | f!z [=] Zero.
Proof.
  Let_ n be nat.
  Induction n.
  Show_ (f:(cpoly_cring CC))
    (degree_le (O) f)->(nonConst CC f)->z:CC | f!z[=]Zero.
    Let_ f be (cs_crr (cpoly_cring CC)).
    Assume (degree_le O f) (A0) and (nonConst CC f).
    Then consider n such that
      ([n:nat](lt (O) n)) (A2) and
      ([n:nat]((nth_coeff n f)[#]Zero)) (A3).
    Have ((nth_coeff n f) [=] Zero) [by A2, A0].
    Then (Not (nth_coeff n f)[#]Zero) [by eq_imp_not_ap].
  Hence thesis [by A3].
  Show_ ((f:(cpoly_cring CC))
      (degree_le (S n) f)->(nonConst CC f)->z:CC | f!z[=]Zero).
    Let_ f be (cs_crr (cpoly_cring CC)).
    Assume (degree_le (S n) f) (A1) and  (nonConst CC f).
    Then consider m' such that
      ([m':nat] (lt (O) m')) (A4) and
      ([m':nat]((nth_coeff m' f)[#]Zero)) (A5).
    Then  f[#]Zero [by nth_coeff_ap_zero_imp].
    Then consider c such that
      ([c:CC](f!c[#]Zero)) [by poly_apzero_CC].
    Then consider a such that
      ([a:CC](b:CC & g:cpoly_cring CC | (degree_le n g) | (f[=]
                ((_C_ a)[*]_X_[+] (_C_ b))[*]g))) [by FTA_1', A1].
    Then consider b such that
      ([b:CC](g:cpoly_cring CC | (degree_le n g) | (f[=]
                ((_C_ a)[*]_X_[+](_C_ b))[*] g))).
    Then consider g such that
      ([g:(cpoly_cring CC) ](degree_le n g)) (A6) and
      ([g:(cpoly_cring CC)](f[=]((_C_ a)[*]_X_[+](_C_ b))[*]g)) (A7).
    First case (m:nat | (S m)=m').
      Then consider m such that
       ([m:nat](S m)=m') (A8).
      Have (nth_coeff (S m) f)
              [=] (nth_coeff (S m) ((_C_ a)[*]_X_[+](_C_ b))[*]g) (A9).
```

```
        _[=] a[*](nth_coeff m g)[+]b[*](nth_coeff (S m) g).
    Have  ((nth_coeff (S m) f)[#] Zero) [by A8,A5].
    Then (a[*](nth_coeff m g)[+]b[*](nth_coeff (S m) g) [#] Zero)
         (A10) [by A9,ap_well_def_lft_unfolded].
    First case (a[*](nth_coeff m g)[#]Zero).
      Then (a [#] Zero) (H6') [by cring_mult_ap_zero].
      Have f!([--]b[/]a[//]H6')
         [=] (((_C_ a)[*]_X_[+](_C_ b))[*]g)!([--]b[/]a[//]H6')  (A11).
        _[=] ((_C_ a)[*]_X_[+](_C_ b))!([--]b[/]a[//]H6')[*]g!([--]b[/]a[//]H6').
        _[=] (((_C_ a)[*]_X_)!([--]b[/]a[//]H6')[+]
              (_C_ b)!([--]b[/]a[//]H6'))[*] g!([--]b[/]a[//]H6').
        _[=] ((_C_ a)!([--]b[/]a[//]H6')[*]_X_!([--]b[/]a[//]H6')[+]b)[*]
                                  g!([--]b[/]a[//]H6').
        _[=] (a[*]([--]b[/]a[//]H6')[+]b)[*]g!([--]b[/]a[//]H6')
        _[=] (Zero::CC).
      Take ([--]b[/]a[//]H6') and prove (f!([--]b[/]a[//]H6')[=]Zero).
    Thus thesis [by A11].
    Next case (b[*](nth_coeff (S m) g)[#]Zero) (A12).
      Claim (nonConst CC g).
        Have (lt (0) (S m)) (A13) [by A4 , A8].
        Have (nth_coeff (S m) g)[#]Zero (A14)
                              [by cring_mult_ap_zero_op, A12].
        Take (S m) and prove (lt (0) (S m)) and ((nth_coeff (S m) g)[#]Zero).
        Thus thesis [by A13].
        Thus thesis [by A14].
      End_claim.
      Then consider z such that
        ([z:CC](g!z[=]Zero)) [by Hrecn, A6].
      Have f!z [=] (((_C_ a)[*]_X_[+](_C_ b))[*]g)!z (A15).
               _[=] ((_C_ a)[*]_X_[+](_C_ b))!z[*]g!z.
               _[=] (((_C_ a)[*]_X_[+](_C_ b))!z)[*]Zero.
               _[=] (Zero::CC) .
      Take z and prove (f!z[=]Zero).
    Thus thesis [by A15].
    End_cases [by cg_add_ap_zero, A10].
  Next case (0) = m'.
    Then (lt (0) (0)) [by A4].
  Hence thesis [by lt_n_n].
  End_cases [by 0_or_S].
Qed.
```

6.4 Example from the Mizar library

The fourth one is a proof on the equality of the minimum of two real numbers and the absolute value of the subtraction of these numbers. Likewise in the above proof of FTA theorem the tactic `Let_ x,y be IR` cannot be used because of coercion. The original proof was taken from Mizar library and is presented below MMode proof.

```
Lemma min_abs:
```

```
 (x,y:IR) (Min x y) [=] (x [+] y [-] (AbsIR (x [-] y))) [/] Two
                          [//] (two_ap_zero IR).
Proof.
  Let_ x,y be (cs_crr IR).
  First case (x [<=] y) (H).
    Have (Min x y) [=]  x
                         [by eq_symmetric_unfolded, leEq_imp_Min_is_lft,H].
                  _[=] ((x [+] x)[/](Two::IR)[//](two_ap_zero IR)).
                  _[=] ((x[+]y)[-](y[-]x))[/](Two::IR)[//](two_ap_zero IR).
                  _[=] ((x[+]y)[-]((Max x y)[-]x))[/](Two::IR)[//](two_ap_zero IR)
                              [by H,leEq_imp_Max_is_rht;
                               Auto 6 with algebra_r algebra algebra_c algebra_s].
                   _[=] ((x[+]y)[-]((Max x y)[-](Min x y)))[/](Two::IR)[//]
                     (two_ap_zero IR)
                          [by H,leEq_imp_Min_is_lft;
                              Auto 6 with algebra_r algebra algebra_c algebra_s].
  Thus             _[=] ((x[+]y)[-](AbsIR (x[-]y)))[/](Two::IR)[//](two_ap_zero IR)
                        [by eq_symmetric_unfolded, Abs_Max;Algebra].
  Next case y[<=]x (H1).
    Have (Min x y) [=] (Min y x) [by Min_comm].
                  _[=] y [by eq_symmetric_unfolded, leEq_imp_Min_is_lft, H1;
                           Algebra].
                  _[=] ((Two [*] y))[/](Two::IR)[//](two_ap_zero IR) .
                  _[=] ((x[+]y)[-](x[-]y))[/](Two::IR)[//](two_ap_zero IR).
                  _[=] ((x[+]y)[-](x[-](Min y x)))[/](Two::IR)[//](two_ap_zero IR)
                           [by eq_symmetric_unfolded, leEq_imp_Min_is_lft, H1;
                              Auto 6 with algebra_r algebra algebra_c algebra_s].
                  _[=] ((x[+]y)[-]((Max y x)[-](Min y x)))[/](Two::IR)[//]
                       (two_ap_zero IR)
                          [by eq_symmetric_unfolded, leEq_imp_Max_is_rht, H1;
                             Auto 6 with algebra_r algebra algebra_c algebra_s].
                   _[=] ((x[+]y)[-]((Max x y)[-](Min y x)))[/](Two::IR)[//]
                       (two_ap_zero IR) [by Max_comm;Algebra].
                   _[=] ((x[+]y)[-]((Max x y)[-](Min x y)))[/](Two::IR)[//]
                       (two_ap_zero IR) [by Min_comm;Algebra].
  Thus             _[=] ((x[+]y)[-](AbsIR (x[-]y)))[/](Two::IR)[//](two_ap_zero IR)
                         [by eq_symmetric_unfolded, Abs_Max;Algebra].
  End_cases [by LeEq_dec].
Qed.
```

The Mizar proof:

```
theorem Th34:
min(x,y) = (x + y - abs(x - y)) / 2
proof
  now per cases;
  suppose
A1:   x <= y; then 0 <= y - x by Th12;
  then A2:   0 <= -(x-y) by XCMPLX_1:143;
  thus min(x,y) = x by A1,Def1
```

```
               .= (x+x)/2 by XCMPLX_1:65
               .= (((x+y)-y)+x)/2 by XCMPLX_1:26
               .= ((x+y)- (y - x))/2 by XCMPLX_1:37
               .= ((x+y)- -(x - y))/2 by XCMPLX_1:143
               .= ((x+y)-abs(-(x - y)))/2 by A2,ABSVALUE:def 1
               .= ((x+y)-abs(x - y))/2 by ABSVALUE:17;
  suppose
A3:   y <= x;
  then A4:   0 <= x - y by Th12;
  thus min(x,y) = y by A3,Def1
               .= (y+y)/2 by XCMPLX_1:65
               .= (x+y-x+y)/2 by XCMPLX_1:26
               .= ((x+y)- (x - y))/2 by XCMPLX_1:37
               .= ((x+y)-abs(x-y))/2 by A4,ABSVALUE:def 1;
  end;
  hence thesis;
end;
```

7 Synonyms

One of the main aims of our Mizar mode is to improve readability of Coq proofs. To make them look closer to mathematical text MMode was equipped with some synonyms of the basic tactics described in Section 5. They are available if we put the declaration `Require HMode_synon.` Here is the list of all synonyms. The *tactic* argument is either **by** or a Coq tactic and has to be put in square brackets, the *label* argument has to be put in round brackets. The only exception is the synonym **The ... is ...** for the **Assume** tactic, where *label* has to be put in square brackets. In this list square brackets mean an optional argument.

- **Assume** *formula* [*label*] *tactic*.
  Synonyms:
  - **Now assume** *formula*.
  - **The** *label* **is** *formula*.
  - **Indeed assume** *formula label*.
- **Have** *formula* [*label*] *tactic*.
  - **We have** *formula tactic*.
  - **Secondly we have** *formula tactic* .
  - **We have** *tactic formula label*.
  - **First we have** *formula label tactic*.
  - **Now** *formula label tactic*.
  - **Now** *tactic* **we have** *formula label*.
- **Then** *formula* [*label*] [*tactic*].
  - **Also** *formula* [*label*] *tactic*.
  - **So** *formula label* [*tactic*].
  - **Then** *tactic formula label*.
  - **Hence** *tactic* **again** *formula label*.

  - **Therefore** *tactic formula label*.
- **Thus thesis** *tactic*.
  - **Done** *tactic*.
  - **Then we are done** *tactic*.
- **Thus** *formula tactic*.
  - **Done** *formula tactic*.
  - **Firstly** *formula tactic*.
- **Hence thesis** *tactic*.
  - **Hence done** *tactic*.
  - **Hence we have our claim** *tactic*.
  - **Hence claim done** *tactic*.
  - **So we are done** *tactic*.
- **Show_** *formula*.
  - **We need to prove** *formula*.
  - **Finally we need to prove** *formula*.
- **Claim** *formula*.
  - **Secondly claim** *formula*.
- **Take** *variable* **and prove** *formula*.
  - **Finally take** *variable* **and prove** *formula*.
- **Then consider** *variable* **such that** *formula label*.
  - **Now choose** *variable* **such that** *formula label*.
- **First case** *formula* [*label*].
  - **Case 1** *formula* [*label*].
- **Next case** *formula* [*label*].
  - **Case 2** *formula* [*label*].
- **Apply** *term*.
  - **We apply** *term*.
- **Idtac**.
  - **So we have proved** *label*.
  - **Secondly**.

The definition of the tactic **Claim** in the `HMode_synon` is different from the definition of the **Claim** in the `MMode`. In the `MMode` the **Claim** requires to apply the tactic **End_claim** after we have finished the proof (see 5.4) while in the `HMode_synon` it does not. The **End_claim** is to indicate in the proof script the end of the 'claim' proof. In the `HMode_synon` we do not need such a tactic. We have tactics like **Hence claim done** which realize two things. We can do the last step in the 'claim' proof (in the MMode we use the **Thus thesis**) and the tactic indicates in the proof script the end of the 'claim' proof (the **Thus thesis** is not so meaningful).

The version of Mizar mode with synonyms is so far only an experiment. The above list was prepared to play with two proofs. The one is presented below, it is again the proof that every natural number can be factorized into a product of primes, and the second one, the proof of the Quotient-Remainder theorem, is available in the directory examples (`euclid_synon.v`). We are aware that the list should be enlarged and to the existing synonyms options like label should be added to make the version fully functional.

```
Lemma nat_factorizes :
(n:nat)(O<n)->(EX L:ListN |((allprimes L)/\(n=(Prod L)))).
Proof.
  LetTac  P :=[n:nat](O<n)->(EX L:ListN |((allprimes L)/\(n=(Prod L)))).
  We apply (cv_ind P).
  We need to prove ((n:nat)(before n P) -> (P n)).
   Let_ n be nat.
    The [IH] is

                (before n P).

    Case 1 (prime n) (A1).
      Now assume ((O)<n).
      We have [by A1]

                (allprimes (cons n nil)) (A2).

      Also n=(Prod (cons n nil)) (A3) [by refl_equal , times_com].
      Take (cons n nil) and prove
                ((allprimes (cons n nil))/\n=(Prod (cons n nil))).
      Then we are done [by A2 , A3].
    Case 2 ~(prime n).
      Assume ((O) < n) (A4).
      Have  (HPD n) [by nat_HPD].
      Now choose pp such that

              ([pp:nat] (primediv pp n)) (A5).

      Then (pp [|] n).
      Now  choose x such that

                ([s:nat](((O)<pp)/\(s[x]pp)=n)) (A7).

      Claim  (x<n)/\((O)<x) (A8).
        First we have (pp[x]x)=n (A8) [by times_com , A7].
        So n=(pp[x]x) (A9).
        We have (prime pp) [by A5].
        Then  one<pp.
        Hence (x<n) [by A9 ,  A4  , propprod_propfact].

        Secondly claim (~x=O).
          Indeed assume (x = O) (A10).

          Claim O=n.
            So ((x [x] pp)=O) (A11) [by A10].
            Also ((x[x]pp)=n) [by A8 , times_com].
          Hence we have our claim [by A11].
        Then (O < O) [by A4].

        Hence claim done [by not_lt_zero].
```

```
      Hence ((0) < x) [by neq_zero_imp_gt_zero].

      Now [by A8, IH] we have

               (P x) (A12).

      Then [by A8]

               (EX L:ListN |((allprimes L)/\(x=(Prod L)))) (A13).

      Now choose LL such that

              [LL:ListN] (allprimes LL)/\x=(Prod LL) (A14).

      We have (prime pp) [by A5].
      Also  (allprimes (cons pp LL)) (A15) [by A14].
      We have

               ((Prod LL)[x]pp)=(pp[x](Prod LL)) [by times_com].

      Hence [by A14] again (x[x]pp) = (pp[x](Prod LL)) (A17).
      We have (n = (x [x] pp)) [by A7 , eq_sym].
      Therefore [by A17]  (n=(Prod (cons pp LL))) (A16).
      Finally take (cons pp LL)
            and prove (allprimes (cons pp LL))/\n=(Prod (cons pp LL)).
    So we are done [by A15, A16].
  End_cases [by prime_dec].
Qed.
```

8 Conclusion

8.1 Discussion

The MMode system is a rather complete Mizar mode for Coq: it emulates all Mizar proof steps. However, currently it is just a prototype. It has just been developed enough to process the five example proofs. We would expect that for the next few proofs it will still need to be extended significantly. For instance, the **by** tactic only has been implemented for at most three references (in the real Mizar system there are **by** justifications with tens of references). This shows that at the moment the system is just a proof of concept.

We will now compare the efficiency of MMode proofs as compared to 'old style' Coq proofs:

– Our MMode proofs are approximately twice as long as the corresponding Coq proofs. There are two reasons for this. The Coq proof does not contain the statements of the proof states, while the MMode proof does. Also, the MMode proof text is indented so it contains much more whitespace.

Surprisingly, the MMode version of a Mizar proof is longer too. The MMode version of `min_abs` is about three times as long as the Mizar original. The main reason for this is that the C-CoRN notation used in the Coq version of the Mizar statements is much clumsier than the original Mizar notation.
- Our MMode proofs take approximately three times as much time to check as the corresponding Coq proofs. This is all caused by the fact that we generate the justifications with the **by** tactic. The proofs where the justifications have been 'expanded' (in `other/expanded_by/`) are only slightly slower than the originals.
  The use of **by** slows checking for two reasons. First, it takes time for it to find the proof. Second, the proof it finds generally is less efficient than a proof found manually.
  Bold claim: We expect that there is room for improvement in the performance of **by**. The current version in MMode is really just a first experimental version.
- The MMode proof terms are less than twice as large as the corresponding Coq proof terms. (We looked at the size of the `.vo` files to judge the size of the proof terms.)
  This difference is even less when considering the β-normal forms of the proof terms. MMode proofs contain many more cuts than traditional Coq proofs. This is one of the main reasons for the increase in the size of the proof terms. Another reason, again, is that **by** will not always find as efficient a proof as a human will.

All in all we can summarize that using MMode probably imposes a performance penalty – both in space and time – of about a factor of two. We expect that when we develop MMode, there will be some space for improvement, mainly in the time dimension. However we do not expect that MMode proofs will ever be as small and fast as old style Coq proofs.

This can be compared to writing programs in assembly versus using a high level language like C. In assembly a program will be faster and will use less resources. However, this is not always a good reason to program in assembly.

8.2 Future work

Most importantly, the MMode system needs to be turned into more than a prototype. For this, two things are essential:

- MMode needs to be ported to the latest versions of Coq and C-CoRN (at the time of writing this report, the latest version of Coq is still version 7.4 but C-CoRN already has been modified to work with a more recent 'CVS version' of Coq).
- MMode needs to be used for a significant 'proof development', a formalization of a non-trivial piece of mathematics.

Then, there are some issues of a more theoretical character:

- The current version of MMode is very Coq specific. It would be nice to give a more system independent and more theoretically oriented description of the meaning of the MMode steps.
- It is an interesting question whether the given implementation of **by** in MMode is *complete*. The question then, is whether it is possible to prove any Coq theorem when only using the MMode tactics (where only variables are used for the references after the **by** tactic).

The syntax of MMode is currently rather restricted, mainly due to our decision only to use Coq's `Grammar` rules to implement it. There is room for improvement here:

- The tactics in the Appendix on page 53 only give a small sub-language of the syntax from Section 3.1 on page 9. Many more instances of the steps in this grammar should be implemented as tactics. For instance there should be **by** tactics with more than three arguments, **Let_** tactics that bind more than two variables, and so on.
  Alternatively we might look into having a MMode parser that is not just built using `Grammar` rules. If we do this, the MMode system will be harder to install and it will not work in all Coq environments anymore. But on the other hand, the full grammar from Section 3.1 might be available then. Also the keywords **Let_** and **Show_** would not need an underscore anymore, but could just be **Let** and **Show**.
- There should be more serious investigation of synonyms for MMode steps, as described in Section 7. It might be interesting to take an existing mathematical text (Henk Barendregt calls this 'best mathematical style'), and collect statistics on what wordings are actually used for the various kinds of MMode steps.
- Combinations of MMode tactics could be given syntax of their own. For instance Mizar's **Given** (= **Assume** + **Consider**) or **Let ... such that** (= **Let** + **Assume**) could be implemented. Another possibility might be to implement an **Otherwise** tactic: the step **Otherwise** P then would be an abbreviation of **Assume** ~*thesis*; **Then** P.

The tactics of MMode also can use improvements:

- When using the Sketch variant of MMode, the error messages for steps that have not been sufficiently justified just appear in the contexts. This means that when compiling a file with `coqc` there will be no feedback about those errors. Currently it is not possible to produce appropriate error messages without going to the ML level in the implementation of the tactics (we decided we did not want to do this). In the next version of Coq having proper error messages in the Sketch mode probably will be possible without programming on the ML level.
- The **by** tactic in its current form probably is (a) too weak and (b) too inefficient. It should be refined to find a proof faster, find more efficient proofs and find proofs in more cases. (It is not clear to us whether going to the ML level for the implementation of the **by** tactic would help with this.)

- It would be useful if **by** could print the proof it finds. (This is similar to the **Info** tactical of Coq.)
- The **by** tactic currently does two things. It builds a restricted context that just contains the arguments of the tactic. Then it searches for a proof in this context with **GAuto**. Separating these two parts might make it possible to substitute other automated proof tactics for **GAuto** but keep the property that only the arguments of the tactic can be used by it.
- The **by** tactic currently only takes references and tactics. It would be useful to extend this with **Hints** classes.
- The 'trick' of **End_claim** to have the same subgoal twice (to 'remember' what subproof one is working on) might be modified. For instance the second subgoal could be different, to make it more obvious that this is just a reminder of a subproof that needs to be closed, instead of a statement that really needs a proof of itself.

Finally an interesting topic is to investigate whether it is possible to automatically convert an existing 'old style' Coq proof script into an MMode proof. Of course we do not think that this would be a better way of getting MMode proofs than writing them directly. However, it might be interesting to see whether one could use MMode as a 'proof presentation language' for existing 'legacy' Coq proofs. This would be closely related to the work on generating proof presentations from Coq tactic proof scripts by Frédérique Guilhot, Hanane Naciri and Loïc Pottier (they do not have a publication about it yet).

Related to this is the question whether it is possible to support the writing of MMode proofs with some automation. For instance, most MMode proofs start with a couple of **Let_** and **Assume** steps that really are determined by the statement that one is proving. In traditional Coq proofs one only needs to write **Intros** for this. It should be possible to have a command that inserts these **Let_** and **Assume** steps automatically into the proof text. Henk Barendregt calls this kind of automatic support for the writing of MMode proofs *luxury* MMode.

A naive way to translate a traditional proof script into an MMode proof is to convert the Coq *proof term* that is constructed by the proof script to an MMode proof, but we do not think this is the best way. A better approach probably is to 'merge' all subgoals that the tactic script goes through into an MMode proof skeleton. Probably that would give an MMode proof of which not all the justifications can be done by **by**, but it would be a good starting point for a human to produce a working MMode proof.

Acknowledgments. We especially would like to thank Henk Barendregt for his enthusiasm for what he calls 'mathmode'. Thanks also to Dan Synek and Iris Loeb for their comments on draft versions of this document. Further thanks for all the help we got from Luís Cruz-Filipe, Herman Geuvers, Milad Niqui, Bas Spitters and Jasper Stein. Finally many thanks to Hugo Herbelin who showed us how to get Coq to do what we wanted it to do.

The first author of this report was financed as a young researcher under European Community contract nr. HPRN-CT-2000-00102, in the thematic network CALCULEMUS, Systems for Integrated Computation and Deduction.

References

1. Henk Barendregt. Towards an interactive mathematical proof language. Unpublished, <ftp://ftp.cs.kun.nl/pub/CompMath.Found/mathmode.pdf>.
2. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, and D. Vasaru. An Overview on the Theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*, Maui, Hawaii, 1997. ACM Press.
3. Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
4. C. Coquand. Agda home page, 2000. <http://www.cs.chalmers.se/~catarina/agda/>.
5. Luís Cruz-Filipe. A Constructive Formalization of the Fundamental Theorem of Calculus. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 108–126. Springer-Verlag, 2003.
6. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
7. M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 1979.
8. John Harrison. *The HOL Light manual (1.1)*, 2000. <http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz>.
9. Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, 2000.
10. M. Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993. <http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz>.
11. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Amsterdam, 1994.
12. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
13. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer-Verlag.
14. Don Syme. Three Tactic Theorem Proving. In *Theorem Proving in Higher Order Logics, TPHOLs '99, Nice, France*, volume 1690 of *LNCS*, pages 203–220. Springer, 1999.
15. The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2002. <ftp://ftp.inria.fr/INRIA/coq/current/doc/Reference-Manual-all.ps.gz>.
16. M. Wenzel and F. Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29:389–411, 2002.
17. Markus Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

18. F. Wiedijk. Mizar: An Impression. Unpublished, <http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz>, 1999.
19. F. Wiedijk. The Mathematical Vernacular. Unpublished, <http://www.cs.kun.nl/~freek/notes/mv.ps.gz>, 2000.
20. F. Wiedijk. Mizar light for HOL light. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *LNCS*, 2001.
21. Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In *Theorem Proving in Higher Order Logics, TPHOLs '99, Nice, France*, volume 1690 of *LNCS*, pages 185–202. Springer, 1999.

A The implemented MMode tactics

We now list all MMode tactics that have been implemented thus far. These tactics give a sub-language of the proof language of the grammar on page 9 in Section 3.1.

A.1 by

‘**by**’ *ref*
‘**by**’ *ref* ‘**,**’ *ref*
‘**by**’ *ref* ‘**,**’ *ref* ‘**,**’ *ref*

‘**by**’ *ref* ‘**with**’ *tactic*
‘**by**’ *ref* ‘**,**’ *ref* ‘**with**’ *tactic*
‘**by**’ *ref* ‘**,**’ *ref* ‘**,**’ *ref* ‘**with**’ *tactic*

A.2 Let_/Assume

‘**Let_**’ *var* ‘**be**’ *type*
‘**Let_**’ *var* ‘**,**’ *var* ‘**be**’ *type*
‘**Let_**’ *var* ‘**,**’ *var* ‘**be**’ *type* ‘**and**’ *var* ‘**be**’ *type*
‘**Let_**’ *var* ‘**be**’ *type* ‘**,**’ *var* ‘**be**’ *type*

‘**Assume**’ *formula*
‘**Assume**’ *formula* ‘**and**’ *formula*
‘**Assume**’ *formula* ‘(’ *label* ‘)’
‘**Assume**’ *formula* ‘(’ *label* ‘)’ ‘**and**’ *formula*

A.3 Have/Then

‘**Have**’ *formula* ‘[’ *tactic* ‘]’
‘**Have**’ *formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’
‘**Have**’ *formula*
‘**Have**’ *formula* ‘(’ *label* ‘)’

‘**Then**’ *formula* ‘[’ *tactic* ‘]’
‘**Then**’ *formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’
‘**Then**’ *formula*
‘**Then**’ *formula* ‘(’ *label* ‘)’

A.4 Iteration

‘_=’ *term* ‘[’ *tactic* ‘]’
‘=_’ *term* ‘[’ *tactic* ‘]’
‘_[=]’ *term* ‘[’ *tactic* ‘]’
‘[=]_’ *term* ‘[’ *tactic* ‘]’
‘_[=]’ *term*
‘[=]_’ *term*
‘**Thus**’ ‘_=’ *term* ‘[’ *tactic* ‘]’
‘**Thus**’ ‘=_’ *term* ‘[’ *tactic* ‘]’
‘**Thus**’ ‘_[=]’ *term* ‘[’ *tactic* ‘]’
‘**Thus**’ ‘[=]_’ *term* ‘[’ *tactic* ‘]’
‘**Thus**’ ‘_[=]’ *term*
‘**Thus**’ ‘[=]_’ *term*

A.5 Claim/Thus

‘**Claim**’ *formula*
‘**Claim**’ *formula* ‘(’ *label* ‘)’
‘**End_claim**’

‘**Thus**’ ‘**thesis**’ ‘[’ *tactic* ‘]’
‘**Hence**’ ‘**thesis**’ ‘[’ *tactic* ‘]’
‘**Hence**’ ‘**thesis**’

‘**Thus**’ *formula* ‘[’ *tactic* ‘]’
‘**Hence**’ *formula* ‘[’ *tactic* ‘]’

A.6 Consider

‘**Consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘[’ *tactic* ‘]’
‘**Consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula*
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘[’ *tactic* ‘]’
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’

‘**Consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula*
‘**and**’ *bound-formula* ‘[’ *tactic* ‘]’
‘**Consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’
‘**and**’ *bound-formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’
‘**and**’ *bound-formula* ‘(’ *label* ‘)’
‘**Then**’ ‘**consider**’ *var* ‘**such**’ ‘**that**’ *bound-formula* ‘(’ *label* ‘)’
‘**and**’ *bound-formula* ‘(’ *label* ‘)’ ‘[’ *tactic* ‘]’

A.7 Take

‘**Take**’ *term* ‘**and**’ ‘**prove**’ *formula*
‘**Take**’ *term* ‘**and**’ ‘**prove**’ *formula* ‘**and**’ *formula*

A.8 Per cases

‘**First**’ ‘**case**’ *formula* ‘`(`’ *label* ‘`)`’
‘**First**’ ‘**case**’ *formula*
‘**Next**’ ‘**case**’ *formula* ‘`(`’ *label* ‘`)`’
‘**Next**’ ‘**case**’ *formula*
‘**End_cases**’ ‘`[`’ *tactic* ‘`]`’

A.9 Show_

‘**Show_**’ *formula*