MUSCADET version 4.1
User's Manual

http://www.math-info.univ-paris5.fr/~pastre/muscadet/manuel-en.pdf [1]

2011

Dominique PASTRE
LIPADE – University Paris Descartes
pastre@math-info.univ-paris5.fr

1. Introduction

The MUSCADET theorem prover is a knowledge-based system. It is based on natural deduction, following the terminology of Bledsoe (Bledsoe[71, 77]), and uses methods which resemble those used by humans. It is composed of an inference engine, which interprets and executes rules, and of one or several bases of facts, which are the internal representations of "theorems to be proved".

[1] French version in http://www.math-info.univ-paris5.fr/~pastre/muscadet/manuel-fr.pdf

Rules are either universal and put into the system, or built by the system itself by metarules from data (definitions and lemmas) given by the user. They are in the form *if <list of conditions>, then <list of actions>*. Conditions are normally properties that are quickly verified. Actions may be either elementary actions which are quickly executed, or "super-actions" which are defined by packs of rules.

The representation of a "theorem to be proved" (or a sub-theorem) is a description of its state during the proof. It is composed of objects that were created, of hypotheses, of a conclusion to be proved, of rules called active rules, possibly of sub-theorems, etc. At the beginning, it is only composed of a conclusion, which is the initial statement of the theorem to be proved, and of a list of rules, which are called active, i.e. relevant for this theorem, and which were built automatically.

Active rules, when applied, may add new hypotheses, modify the conclusion, create new elements, create sub-theorems or build new rules which are local for a (sub-)theorem. If the conclusion was set to `true` - for example, if the conclusion to be proved was added as a new hypothesis or if there is an existential conclusion $\exists X\, p(X)$ and a hypothesis $p(a)$ - then the theorem is proved. If it is only a sub-theorem, this information is transmitted up to the theorem that created it.

2. Examples

In all this text, PROLOG conventions will be used to write constants (names starting with a lowercase letter) or variables (names starting with an uppercase letter or with the symbol "_"). Moreover this conventions will be extended to predicates (which is not possible in PROLOG), and by extension, $P(X)$ will be used to write whatever expression which depends on X.

2.1. Transitivity of inclusion

Prove the transitivity of inclusion

$$\forall A \forall B \forall C\ (A \subset B \wedge B \subset C \Rightarrow A \subset C)$$

with the definition of inclusion

$$A \subset B \Leftrightarrow \forall X\ (X \in A \Rightarrow X \in B)$$

To prove this theorem MUSCADET creates objects *a, b* and *c* by applying three times the rule

Rule $\forall$: *if the conclusion is* $\forall X\, C(X)$
then create a new object x and the new conclusion is $C(x)$

and the new conclusion is

$$a \subset b \wedge b \subset c \Rightarrow\ a \subset c$$

Then the rule

Rule $\Rightarrow$: *if the conclusion is* $H \Rightarrow C$
then add the hypothesis H and the new conclusion is C

replaces the conclusion by $a \subset c$ and adds the two hypotheses $a \subset b$ and $b \subset c$.
In effect, hypotheses *H* are analyzed before being added: a super-action *addhyp(H)* contains, among others, the rule

if H is a conjunction,
then successively add all the elements of the conjunction

(This rule is of course recursively applied if necessary).
The conclusion is then replaced by its definition

$$\forall X\ (X \in a \Rightarrow X \in c)$$

by applying the rule

Rule def_concl_pred : *if the conclusion is C*
there exists a definition of the form $C \Leftrightarrow D$
then the new conclusion is D

By the preceding rules $\forall$ and $\Rightarrow$, there is then a new object x, a new hypothesis $x \in a$, and the conclusion is now $x \in c$.
The following rule

Rule $\subset$: *if there are hypotheses* $A \subset B$ *and* $X \in A$
then add the hypothesis $X \in B$

is a rule that was automatically built by MUSCADET from the definition of inclusion.
Here it is applied twice, adds the hypotheses $x \in b$ then $x \in c$, which is the same as the conclusion to be proved. The proof ends by applying the rule

Rule stop_hyp_concl : *if the conclusion C is also a hypothesis*
then set the conclusion to true

MUSCADET is able to work in second order predicate calculus, and the preceding example may be written in the form

transitive($\subset$)

with the definition of the transitivity of a relation R

transitive(R) $\Leftrightarrow \forall A \forall B \forall C$ [$R(A,B) \wedge R(B,C) \Rightarrow R(A,C)$] [2]

After the conclusion transitive($\subset$) is replaced by its definition, the proof is the same as above.

One may also work with the power set of a set

$\forall E$ transitive($\subset$, $\mathcal{P}(E)$)

with transitive(R,E) $\Leftrightarrow \forall A \forall B \forall C$ ($A \in E \wedge B \in E \wedge C \in E \Rightarrow$ [$R(A,B) \wedge R(B,C) \Rightarrow R(A,C)$])

and $X \in \mathcal{P}(E) \Leftrightarrow \forall X\, (X \in E \Leftrightarrow X \subset E$

Relativized[3] quantifiers may be used in order to reduce writing:

transitive(R,E) $\Leftrightarrow \forall A \in E\ \forall B \in E\ \forall C \in E$ [$R(A,B) \wedge R(B,C) \Rightarrow R(A,C)$]

$X \in \mathcal{P}(E) \Leftrightarrow \forall X \in E\ X \subset E$

2.2. Power set of the intersection of two sets

Prove the following theorem

$$\forall A\, \forall B\, (\mathcal{P}(A \cap B) =_{set} \mathcal{P}(A) \cap \mathcal{P}(B))$$

with the definition of the intersection

$$X \in A \cap B \Leftrightarrow X \in A \wedge X \in B \quad \text{or} \quad A \cap B = \{X\ ;\ X \in A \wedge X \in B\}$$

of the power set of a set

$$X \in \mathcal{P}(E) \Leftrightarrow \forall X\, (X \in E \Leftrightarrow X \subset E) \quad \text{or} \quad \mathcal{P}(A) = \{X\ ;\ X \subset A\}$$

and of the equality of sets

$$A =_{set} B \Leftrightarrow A \subset B \wedge B \subset A$$

After the creation of objects a and b, as in the preceding example, by a rather complex mechanism which will be described in section 7, MUSCADET "eliminates functional symbols" $\cap$ and $\mathcal{P}$. To do this

[2] Predicate variables are not possible in PROLOG, we will see in section 11 how to express $R(A,B)$

[3] $\forall X \in A\ P(X)$ is short for $\forall X\ (X \in A \Rightarrow P(X))$, $\exists X \in A\ P(X)$ for $\exists X\ (X \in A \wedge P(X))$

it creates, by means of the operator ":", the objects $a\cap b$:c, $\mathcal{P}(a)$:pa, $\mathcal{P}(b)$:pb and $pa\cap pb$:pd. The new conclusion is then

$$pc =_{set} pd$$

After replacement of the equality of the conclusion by its definition, that is

$$pc\subset pd \wedge pd\subset pc$$

the rule

Rule concl_$\wedge$: *if the conclusion is a conjunction*
then successively prove all elements of the conjunction

creates two sub-theorems with numbers 1 and 2.
The conclusion of the first sub-theorem is $pc\subset pd$ and it is replaced by its definition

$$\forall X\,(X\in pc \Rightarrow X\in pd)$$

A new object x is created, a new hypothesis $x\in pc$ is added and the new conclusion is $x\in pd$.

The following rule, which was automatically created from the definition of the power set,

Rule $\mathcal{P}$: *if there are hypotheses* $\mathcal{P}(A)$:B *and* $X\in B$
then add the hypothesis $X\subset A$

gives the hypothesis $x\subset c$.
The following rule [4]

Rule defconcl_elt : *if the conclusion is* $A\in B$
there is a hypothesis Term:B *and a definition* $X\in$ Term $\Leftrightarrow P(X)$
then the new conclusion is $P(A)$

replaces the conclusion by $x\in pa \wedge x\in pb$

The rule concl_$\wedge$ leads to a new splitting, the conclusion of sub-theorem 11 is $x\in pa$ which is replaced by $x\subset a$ by the rule *defconcl_elt*.
By the rules *def_concl_pred,* $\forall$ and $\Rightarrow$, t is created, the hypothesis $t\in x$ is added and the new conclusion is $t\in a$, then the rule $\subset$ gives the hypothesis $t\in c$.
The following rules were automatically created from the definition of the intersection

Rule $\cap$1: *if there are hypotheses* $A\cap B$:C *and* $X\in C$
then add the hypothesis $X\in A$
Rule $\cap$2: *if there are hypotheses* $A\cap B$:C *and* $X\in C$
then add the hypothesis $X\in B$
Rule $\cap$3: *if there are hypotheses* $A\cap B$:C, $X\in A$ *and* $X\in B$
then add the hypothesis $X\in C$

The rule $\cap$1 then gives the hypothesis $t\in a$ which ends the proof of sub-theorem 11.
Sub-theorems 12 then 2, corresponding to other cases coming from splitting, are then proved.

3. From MUSCADET1 to MUSCADET4

3.1 MUSCADET1

A first version of MUSCADET, which is now called MUSCADET1, was described and analyzed in [Pastre 89, 89a, 93, 95].

MUSCADET1 came after an initial program (DATTE, written in Fortran!) [Pastre 76; 78] which was already based on natural deduction (in the sense of Bledsoe[71, 77]), and used methods which resemble those used by humans.

[4] This was a rule in the first versions of MUSCADET. It has been replaced by a more general rule because "belonging to a set" could no longer be known by the system (this was a constraint of the TPTP Library, see section 3.2).

The inference engine of MUSCADET1 was written in PASCAL, and knowledge (rules, metarules and super-actions) was written in a language that was considered simple and declarative. MUSCADET1 produced good results; it was evaluated for several years but its use was limited. In particular, the language was not adapted to the expression of procedural strategies. Writing such strategies was complex and it was difficult to read and understand them.

3.2 MUSCADET2

The following version, called MUSCADET2 (versions 2.0 to 2.7)[Pastre 01a, 01b, 02, 06, 07], has been completely written in PROLOG. The reason for this is that it is possible to use the same language to express declarative knowledge such as rules, definitions, hypotheses, etc., more procedural knowledge such as proof strategies, and the inference engine itself. The inference engine contains only few predicates since it is completed by the PROLOG interpreter. This leads to more flexibility, more facilities for writing, and even more efficiency. Moreover it was possible to carry out many improvements and to write new strategies, which were not possible in the first version. It was also possible to use, without having to implement them, all the facilities of expression of PROLOG such as numerical calculus (missing in MUSCADET1) or infix and partially parenthesized notations by simply defining operators and precedences (but this is not compulsory, it is only more convenient for the user). To indicate mathematical variables or constants, PROLOG conventions are used (variables start with upper-case letters whereas constants start with low-case letters). So it is no longer necessary to precise if a symbol is a variable or a constant (but this convention must imperatively be used).

MUSCADET2 was able to work on problems of the TPTP Problem Library (**T**housands of **P**roblems for **T**heorem **P**rovers, http://www.cs.miami.edu/~tptp). New strategies were added, which were better adapted to the style and to the axiomatizations of this library.
Moreover, two convenient capabilities of MUSCADET had to be left out. The first one was the possibility to declare that some statements are either definitions, or lemmas (or know theorems). These two types of statements are not treated in the same manner and MUSCADET must now analyse them to recognize them (see section 6).
The second capability was the fact that MUSCADET1 knew the set belonging symbol, and that it is not possible in the TPTP library. Rules which used it had to be generalized. For example, the rule *defconcl_rel* seen in section 2.2 has been replaced by the more general rule

Rule defconcl_rel : *if* *the conclusion is R*(*A,B*)
there is a hypothesis Term:B *and a definition X*∈ Term ⇔ Def
then the new conclusion is the expression obtained
by replacing X by A in Def

MUSCADET has participated to CASC competitions (http://www.cs.miami.edu/~CASC) since 1999. MUSCADET, of course, could only compete in the "first order" divisions, that is FOF (FEQ and NEQ), since it does not work with clauses. The results [Pastre 06, 07] show the complementarity of MUSCADET with regard to provers based on the resolution principle.

[Pastre 99] gives an analysis of some insufficiencies of MUSCADET1, which were tackled in MUSCADET2, and the description of some new strategies which were conceived during the work on the TPTP library. The users of MUSCADET1 could also find in [Pastre 98] a more detailed correspondence between some of the techniques of both versions.

In addition to unceasing enlargement of the bases of rules and improvements of proof strategies, in the last versions of Muscadet2, for TPTP problems, the user could call, under Linux, an executable C file which itself called PROLOG and the prover. The interest is that it is easier to use and that it is possible to write scripts to solve lists of problems[5]. On the other hand working under PROLOG allows

[5] There was already such an executable in CASC versions but it gave only the result (prove or not proved) and not the proof or the search of the proof.

to look at the bases of facts after an execution or an interruption, and even to test a rule by forcing it to be applied.

3.3 MUSCADET3

Since 2008, the **syntax of MUSCADET3** [Pastre 10a, 10b] has been **that of the TPTP library**.
Although this was not absolutely necessary, the ":" symbol used in the expression "for the only Y:f(X) such that p(Y)" was replaced by "::" to avoid the confusion with the ":" of TPTP used in writing quantified formulas.

`for_all(X,p(X))`		`! [X] : p(X)`
`exists(X,p(X))`		`? [X] : p(X)`
`for_all(X,for_all(Y,p(X,Y)))`		`! [X,Y] : p(X,Y)`
`exists(X,exists(Y,p(X,Y)))`	are written	`? [X,Y] : p(X,Y)`
`A and B`		`A & B`
`A or B`		`A | B`
`not A`		`~ A`
`only(f(X):Y,PY)`		`only(f(X)::Y,PY)`

The second important modification of version 3 is the possibility of getting and displaying the "useful" trace. For this, it was necessary to be able to go back from the final step to antecedent steps. To this end, steps have been numbered and became new parameters for facts, rules, conditions and super-actions. A step corresponds to the effective application of a rule. So a step may involve several actions. Facts as hypotheses and conclusions are memorized with the number of the step where they have been obtained. So, several facts may have the same step number. The successful and effective application of a rule is memorized. To allow to go back and also to be able to write a detailed justification, and not only the sequence of steps, the system also memorizes the name of the rule, the new step, the instantiated conditions, the list of the steps of the conditions, the instantiated and explicit actions and a text giving a justification. This text is either given for general rules (generally a logical explication), or automatically built in the rule (for example, rule built from the definition of such concept or such axiom with their name, or local rule built from such universal hypothesis). This memorization is done either in the rule or in the super-action, in particular in the case of a recursive super-action such as adding a hypothesis or proving a conjunctive conclusion.

3.4 MUSCADET4

In version 4, in addition to other small improvements, the most important changes concern the writing of the useful trace (version 4.0 submitted to the CASC competition) and the user interface (version 4.1). The interface is more easily used and more complete, either under Linux or under PROLOG . Options allow to directly modify the time limit, the display level (entire trace / useful trace / result according to the SZS ontology) and the language.
In particular, slides of [Pastre 10] contain extracts of useful traces.
The version "th" is set up again and may be used under Linux or under PROLOG. The system can prove one or several theorems the statements of which are in one or several files, so as the statements of definitions and lemmas.

4. Machine representations

Everything is expressed in PROLOG. Mathematical statements are PROLOG expressions. Facts are unit PROLOG clauses. Rules are PROLOG clauses expressing declarative knowledge. Elementary actions and some strategies are PROLOG clauses defining procedural actions. And super-actions are PROLOG clauses grouping packs of rules for a given goal.

The inference engine is composed of the PROLOG interpreter and of some clauses which process the application of rules (`applyrulactiv` and `applyrul`).

4.1. Expression of mathematical statements

The syntax is that of the TPTP library.
The logical connectives & (and), | (or), ~ (not), `=>`, `<=>` are defined as infix operators with precedences in the order as the connectives are written down in mathematics. They are right associative.
The universally quantified formulas are written ! [X,Y,...] : <statement function of X, Y, ...>. The existentially quantified formulas are written ? [X,Y,...] : <statement function of X, Y, ...>. [6]

The `true` and `false` constants may also be used.

The example theorem introduced in section 2.1 is written

```
![A,B,C]:(subset(A,B) & subset(B,C) => subset(A,C))
```

The proof of the theorem `T` will be requested by the PROLOG call `prove(T).`

The definition of subset is

```
subset(A,B) <=> ![X]:(elt(X,A) => elt(X,B))
```

.
This definition is given by

```
definition(subset(A,B) <=> ![X]:(elt(X,A) => elt(X,B))).
```

where `definition` is a PROLOG predicate stating that the argument statement is a mathematical definition.

The definitions of intersection and of power set are

```
![X]:(elt(X,inter(A,B)) <=> elt(X,A) & elt(X,B))
![X]:(elt(X,power_set(A)) <=> subset(X,A))
```

It is possible, as mathematicians do, to use infix operators `elt`, `subset`, `inter` by defining them with their precedences by the PROLOG directives

```
op(200,xfy,elt)
op(200,xfy,subset)
op(150,xfy,inter)
```

then statements may be written

```
![A,B,C]:(A subset B & B subset C => A subset C)
A subset B <=> ![X]:(X elt A => X elt B)
![X]:(X elt A inter B <=> X elt A & X elt B)
![X]:(X elt power_set(A) <=> X subset A)
```

but PROLOG will display

```
A subset B&B subset C=>A subset C
```

we loose more than we gain in the readability of display !

Mathematicians usually write set definitions in the form $f(X,..) = \{X\,;\, p(X,\ldots)\}$, for example $A \cap B = \{\, X;\, X \in A \wedge X \in B \,\}$ or $\mathcal{P}(A) = \{X;\, X \subset A\,\}$.
This possibility [7] also exists in MUSCADET in the form

```
A inter B = [X,X elt A & X elt B]
power_set(A) = [X, X subset A]
```

[6] In MUSCADET2 (and in the corresponding publications, they were written `for_all(X,` <statement function of `X`>) and `exists(X,` <statement function of `X`> and the connectives were written `and`, `or`, `not`.

[7] which disappeared in MUSCADET3 but was restored in MUSCADET4.1

but the symbol used for set belonging must be explicitly indicated by the predicate "+++", that is
`+++(elt).`
In order to reduce writing mathematicians also currently use relativized quantifiers

$\forall X \in A \;\; \forall Y \in B \;\; p(X,Y)$ and $\exists X \in A \;\; \exists Y \in B \; p(X,Y)$

which are abbreviations for

$\forall X \, \forall Y (X \in A \wedge Y \in B \Rightarrow p(X, Y))$ and $\exists X \, \exists Y \, (X \in A \wedge Y \in B \Rightarrow p(X,Y))$.

In MUSCADET2, it was possible to write
`for_all(X elt A,for_all(Y elt B,p(X,Y)))` and
`exists(X elt E, exists(Y elt B, p(X,Y)))`, (`elt` having been defined infix),
This possibility will be restored in the next version of Muscadet in the form
`![X elt A, Y elt B]: p(X,Y)` and `?[X elt A, Y elt B]: p(X,Y)`
or, more generally, with any formula instead of `X elt A`, … .

Remark: the statements of theorems must be closed; the statements of definitions may contain variables which are implicitly universal.

4.2. Expression of facts

The fact that the statement C is the *conclusio*n of the *(sub-)theorem to be prove*d with number N is represented by the unit PROLOG clause `concl(N,C,I)`, where `I` est the number of the step where this fact was created. Some other properties are handled in the same manner, such as to be a *hypothesi*s (`hyp(N,H,I)`), an *objec*t (`obj(N,O)`), a *sub-theore*m (`subth(N,N1)`), etc..
For *active rules* exceptionally, the whole list of rules that are active for a (sub-)theorem to be proved is memorized by the fact `rulactiv(N,[R1,R2,…])`; `[R1,R2,…]` is a list of names of rules that were automatically activated in an order that is important.

4.3. Expression of rules

The rules (simplified) that were used in the example are written

```
rule(N, =>) :- concl(N,A=>B,Step),
               addhyp(N,A,NewStep), newconcl(N,B,NewStep).
rule(N, !) :- concl(N,(!XX:C),Step),
              create_objects_and_replace(N,XX,C,C1,Objects),
              newconcl(N,C1,NewStep).
rule(N, def_concl_pred) :- concl(N,C,Step),definition(Name, C<=>D),
                           newconcl(N,D,NewStep).
rule(N,stop_hyp_concl):- concl(N,C,Step1),ground(C), hyp(N,C,Step2),
                         not hyp(N,elt(X,B),_),
                         newconcl(N,true,NewStep).
```

(these three rules are given in the system)

```
rule(N, subset) :- hyp(N, subset(A,B,Step1), hyp(N, elt(X,A,Step2),
                   not hyp(N, elt(X,B),_),
                   addhyp(N, elt(X,B,NewStep).
```

(this rule was built by the system).

The parameter `N` helps to apply a rule to the (sub-)theorem of number `N`.

Notice that *if* ... *then* ... is implicit. It would have been possible to define PROLOG operator symbols `if … then …`, but this was not indispensable, since all is translated into PROLOG with predicates.
Conditions are generally the verification of the existence (or of the absence) of facts, which are unit PROLOG clauses (`hyp, concl`, see the preceding section), or of a definition which is in a certain form. There may also be elementary conditions.
Actions are either elementary actions, which are expressed as PROLOG predicates and express elementary programs (`create_objects_and_replace`), or super-actions which are defined by packs of rules (`addhyp, newconcl,` etc, see next section).

The condition `not hyp(N, elt(X,B),_)` avoids applying the rule if the theorem of number `N` already contains the hypothesis `elt(X,B)`. (In the example, this condition only avoids the call `addhyp`, which would have no effect here, but in other cases it is essential; for example to avoid an infinite creation of objects.)
For MUSCADET1, this condition was not necessary because a rule could not be applied twice for the same instantiations. This had other disadvantages; for example it was not possible to *force* a rule to be applied again for the same instantiations.

The elementary action `create_objects_and_replace(N,XX,C,C1,Objects)` returns in `Objects` a list of constants `z`, `z1`, `z2`,… etc which have not yet been used and replaces in `C` the variables of `XX` by these constants to give `C1`.

Actually operational rules are more complex, they contain additionnal conditions and actions, which have been added by hand [8] for the rules given to the system, but which are automatically added for rules built by the system. The first rules may be seen in the file `muscadet-en`, the last ones in the files of built rules `rul_`...

*Among the condition*s :
- the `Step` number which will be used in `traces` (see below).

Among the actions :
- steps numbering,
- messages writing,
- `traces(N,rule(Name),`<condition or list of conditions>`,`
  <action or list of actions>, <step>, <explanation>,
  <list of the steps of the conditions>`)`

memorizes the informations which will be later necessary to extract the useful trace.

4.4. Expression of super-actions

Super-actions are generally expressed by packs of rules *if ... then ...* or *if ... then ... else ...*
To action(*X*) : *if ... then ...*
if ... then ...
else ...
is easily written in PROLOG

```
action(X) :-  ( … -> …
              ; … -> …
              ; …      % by default (optional)
              ) .
```

The super-action `newconcl`, which contains only one rule, replaces the conclusion of the (sub-)theorem of number `N` by `C` if the conclusion is not already equal to `C`.

```
newconcl(N,C,Step) :- not concl(N,C,_),
                      step_action(Step),
                      assign(concl(N,C,Step)).
```

with

```
step_action(E1) :- ( var(E1),step(E) -> E1 is E+1,assign(step(E1))
                   ; true).
```

If `E1` has not yet been instantiated, which is the case in the rule "`!`", the step number is incremented by `1`.

[8] because gradually added during experimentations, but they could be automatically added from simplified rules

If E1 has been instantiated, which is the case in the rule =>, where E1 has been instantiated by the first action addhyp (described hereafter), it is the same step.
assign updates the conclusion and the number step.

The super-action addhyp handles the hypotheses that have to be "added" in order to finally add only hypotheses that are elementary and not universal.
Conjunctions are split. Universal hypotheses are not added. Rather, in the place of universal hypotheses, new rules, which are local for the (sub-)theorem, are created.

Here are some of the rules that define this super-action

```
addhyp(N,H,E) :- step_action(E),
      ( % to add a conjunction, the elements of the conjunction
        % are successively (and recursively) added
        H = A & B -> addhyp(N,A,E), addhyp(N,B,E)
      ; if H is already a hypothesis, nothing is done
      ; hyp(N,H,_) ->  true
      ; % the same for a trivial equality
        H = (X = X) -> true
      ; % lexicographic order except for created objects z<number>
        %     which are in the order of their creation (numerical)
        H = (Y=X), atom(X), atom(Y), before(X,Y), addhyp(N,(X=Y),E2)
      ; % if H is universal or is a implication, rule(s) are
        % created, which are local for the theorem of number N
        H = ( !_: _) -> create_name_rule(rulehyp,Name),
                                      buildrules(H,_,N,Name,[])
      ; H = A => B -> create_name_rule(rulehyp,Name),
                                 buildrules( H,_,N,Name,[])
      ...
      ; % else H is added
        assert(hyp(N,H,E)),
      ) .
```

5. How to use MUSCADET4

The *package* contains a PROLOG source file muscadet-en, a script musca-en which allows to work under PROLOG and two small C files th-en.c and tptp-en.c [9] which can be compiled and which allow to work under Linux. The obtained executables will be named later on th and tptp.

Working under PROLOG allows in particular to have access, at the end of the proof (or more important in case of failure !) to all facts representing the state of the theorem to be proved : hyp, concl, sousth, rules (in particular built rules), etc ; and even to directly test the application of a rule on the current state.

It is possible to work from a file containing a list of definitions and one or several theorems to be proved.

It is also possible to directly work on the problems of the TPTP library, after having defined an environment shell variable TPTP to the TPTP library (setenv TPTP <directory of the library>).
Under PROLOG, it is also possible to call directly the predicate prove with as a parameter the statement of the theorem to be proved, after having given the definitions of the mathematical concepts eventually used.

[9] muscadet-fr, musca-fr, th-fr.c and tptp-fr.c for the French version

The PROLOG used is SWI-Prolog, which is freeware downloaded at the following address `http://www.swi.psy.uva.nl/projects/SWI-Prolog/download.html`, and which is called by the command `swipl`.

In all cases you have to be in a directory that contains the PROLOG file `muscadet-en`[10] (or a same-name link to this file).

5.1. Direct proof

(only under PROLOG)
Call the Unix command `musca-en`[11]. This invokes SWI-Prolog and loads the file `muscadet-en`.

To prove $p \rightarrow p$ simply type `prove(p => p).`
Do not forget the dot at the end. No space before the bracket.

Let us come back to the first example.
Introduce the definition of the inclusion by the PROLOG command

```
assert(definition(subset(A,B) <=> ![X]:(elt(X,A) => elt(X,B)))).
```

Then call for the proof

```
prove(![A,B,C]:(subset(A,B) & subset(B,C) => subset(A,C))).
```

Do not forget dots.

The proof will be displayed on the standard output.

If you prefer use infix operators, type

```
op(200,xfy,elt).
op(200,xfy,subset).
assert(definition(A subset B) <=> ![X]:(X elt A => X elt B))).
```

then

```
prove(![A,B,C] : A subset B & B subset C => A subset C).
```

5.2 From files containing theorems and definitions

Data must be written as below :

`:- op(`<precedence>, <type>, <name>`).` (eventually)
`theorem(`<name>,<theorem to be proved>`).`
`definition(`<definition>`).`
`lemme(`<name>,<lemma>`).`
`include(`<data file>`).`
`%` <PROLOG comment>

Do not forget the dots ! (They are PROLOG terms.)
All this data may be written in whatever order.
There may be several theorems which will be proved one after the other.
`include` allows to write data in one or several other file(s) .

Files examples :

```
example1 :
   definition(subset(A,B) <=> ! [X]:(elt(X,A) => elt(X,B))).
```

[10] `muscadet-fr` for the French version

[11] `musca-fr` for the French version

```
   theorem(thI03,![A,B,C]:(subset(A,B) & subset(B,C) => subset(A,C))).

example1bis :
     :- op(200,xfy,elt).
     :- op(200,xfy,subset).
     theorem(thI03,![A,B,C]:(A subset B & B subset C => A subset C)).
     definition(A subset B <=> ! [X]: X elt A => X elt B)).

example2 :
   include(example2_definitions).
   % transitivity of subset
   theorem(thI03,![A,B,C]:(subset(A,B) & subset(B,C) => subset(A,C))).
   % the power set of an intersection is equal to the intersection of the
power sets
   theorem(thI21,![A,B]:subset(powerset(inter(A,B),
                                inter(powerset(A),powerset(B))))).
```

with `exemple2_definitions :`

```
      definition(subset(A,B) <=> ! [X]:(elt(X,A) => elt(X,B))).
      definition(elt(X,powerset(A))<=> subset(X,A)).
      definition(elt(X,inter(A,B)) <=> elt(X,A) & elt(X,B)).

example2bis :
      include(example2bis-definitions).
      :- op(200, xfy, elt).
      :- op(200, xfy, subset).
      :- op(150,xfx,inter).
      % transitivity of subset
      theorem(thI03,![A,B,C]:(A subset B & B subset C => A subset C)).
      % the power set of an intersection is equal to the intersection of
the power sets
      theorem(thI21,![A,B]:(powerset(A inter B) subset powerset(A) inter
powerset(B))).
```

with `example2bis_definitions :`

```
      :- op(200, xfy, elt).
      :- op(200, xfy, subset).
      :- op(150,xfx,inter).
      definition(A subset B <=> ! [X]:(X elt A => X elt B)).
      definition((X elt powerset(A))<=> X subset A).
      definition(X elt  A inter B  <=>  X elt A  &  X elt B).
```

Remark :defining infix operators must be done in all files where they occur.

Under Linux (resp. PROLOG), call the executable (resp. prédicat) `th` with as an argument a file (or a path to a file). Under PROLOG this argument must be an atom. It must be written between quotes if necessary according to PROLOG conventions.

Examples

`th example1` (resp. `th(example1).`)

Under PROLOG it is also possible to call `th` with a file containing only définitions which will be read and memorized. In this case do not put the corresponding `include` in the file of theorems, this would duplicate the definitions (then also the rules).[12]

[12] Until version 2.6, it was possible to read definitions alone, build and memorize the built rules in a file which could be read in a later execution. This was because building rules was relatively long comparatively to proving theorems This possibility was removed in version 2.7. Because of the improvements in execution time of modern machines, restoring it seems to me unnecessary. For very large date bases (as for the very large TPTP problems), it would be necessary to select definitions and axioms in function of the given problems, before building rules, not only select rules after having building them.

The proof will be saved in a file named `res_`<name of the theorem to be proved> (for example `res_th1`). Moreover some messages are displayed on the terminal to follow the prover's work.

5.3. From the TPTP library

(http://www.cs.miami.edu/~tptp**)**

Under Linux (resp. PROLOG) call the executable (resp. predicate) `tptp` with as an argument a path to a TPTP problem file or a TPTP problem name. In this last case, it is necessary to have defined an environment shell variable `TPTP` to the TPTP library (`setenv TPTP` <directory of the library>).

Examples

```
tptp /home/dominique/TPTP/TPTP-v4.0.1/Problems/SET/SET002+4.p
tptp SET027+4.p
```

Under PROLOG, the argument must be an PROLOG atom, then quotes are necessary (occurences of -, +, ., /, capital letters)

```
tptp('/home/dominique/TPTP/TPTP-v4.0.1/Problems/SET/SET0 02+4.p').
tptp('SET027+4.p').
```

The proof will be saved in a file named `res_`<problem name> (for example `res_SET002+4.p`). Moreover some messages are displayed on the terminal to follow the prover's work.

5.4 Modification of default options

5.4.1 Default options

Default options (file `muscadet-en`) are :

- time limit : 10 seconds
- display of the trace of the complete search of the proof : no
- display of the useful proof : yes
- result according to the SZS ontology : no

They may be modified.

To display them type

`listing(timelimit).` or `l(timelimit).` or `timelimit(T).`

`listing(display).` or `l(display).` or `display(A).`

5.4.2 Modifications in a proof call

Additional arguments may be given, in whatever order : a new time limit and new display options in the form ± <option>

+ to add,

- to remove

<option> ::= `tr` for the complete trace
`pr` for the useful proof
`szs` for the result according to the SZS ontology

Examples

`th example_th 50 +tr` or `th(example_th,50,+tr).`

sets the time limit to 50 seconds and displays the complete trace followed by the useful proof,

`th example_th -pr` or `th(example_th,-pr).`

displays only the result (`theorem proved` or `not proved`) and the time used,

`tptp SET027+4.p -pr +szs` or `tptp('SET027+4.p',-pr,+szs).`

displays only the result according to the SZS ontology.

5.4.3 Modifications under PROLOG

The following commands modify the options until next change

```
assign(timelimit(<time>)).  or  modifytimelimit(<time>)).
assert(display(<option>)).
retract(display(<option>)).
```

5.4.4 Modifications of PROLOG *code* (file `muscadet-en`)

Finally, default values can be modified in the file `muscadet-fr` by modifying the line `timelimit(...).` or by removing or adding lines `display(...)`.

5.4.5 Instructions for on line use

Instructions for use may be found again by typing `th` or `tptp` under Linux, or `th.` or `tptp.` under PROLOG.

5.4.6 Modification of the language choice.

It is possible to modify the default choice for the language used to display the traces (complete trace and/or useful trace), and other comments by giving it (`fr` or `en`) as an additional argument in the command or the prédicate `th` or `tptp`.

Examples : `th example1 en` or `th(example1,en).`

Under PROLOG it can also be directly modified par

```
assign(lang(en)).
```

or simply `en.`

Nevertheless the names of the rules and actions will not be modified. For this the english version must be used (files `*-en.*`)

5.4.7 Deletion

Under PROLOG, to delete all the data relative to the last problem and solve a new problem without restarting PROLOG, type :

```
deleteall.
```

To delete only the facts which represent the state of the last proved (or not proved ...) theorem, type :

```
deleteth.
```

which deletes all the facts which represent the state of the last theorem, but the definitions and the built rules are not deleted.

6. Definitions and lemmas

Definitions are not the only mathematical knowledge that is given to the system. Lemmas may also be given by means of a PROLOG predicate `lemma`(<name>, <statement>). For example, the transitivity of the inclusion may be given as a lemma to prove theorems of algebraic topology by

```
lemma(trans_subset, subset(A,B) & subset(B,C) => subset(A,C)).
```

On the other hand many statements, which are given in the problems of the TPTP library, are lemmas[13] and not definitions[14].

[13] for example in domain MGT (Management)

```
%---MP. If a time point belongs to the environment, then the end-point of
%---the environment cannot precede it.
input_formula(mp_environment_end_point,axiom,(
     ! [E,T] : ( ( environment(E) & in_environment(E,T) )
       => greater_or_equal(end_time(E),T) )   )).
```

[14] as

```
%----Definition of greater_or_equal (i.t.o. greater and equal).
input_formula(definition_greater_or_equal, axiom, (
! [X,Y] : ( greater_or_equal(X,Y) <=> ( greater(X,Y) | equal(X,Y) ) ) )).
```

Definitions and lemmas both lead to building rules, but not exactly in the same manner. For example, in a conclusion, $A \subset B$ will be replaced par its definition $\forall X\,(X \in A \Rightarrow X \in B)$ but $A \subset C$ will not be replaced by $A \subset B \wedge B \subset C$.

On the other hand, from a lemma containing the sub-formula $p(X) \Rightarrow q(f(X))$, the rule

if $p(X)$ *and* $f(X)$:Y *then* $q(Y)$

is created and, under some conditions, the rule

if $p(X)$ *then create* Y *and add the hypotheses* $f(X)$:Y *and* $q(Y)$

is also created, but from the definition of the power set

$X \in \mathcal{P}(A) \Leftrightarrow X \subset A$

the system only creates the rule

si $X \in A$ *and* $\mathcal{P}(A)$:PA *then add* $X \subset A$

it does not create the rule

if $X \subset A$ *then create the power set of* A

Lastly, MUSCADET sometimes builds news definitions, which are better adapted to its strategies. This is the case, for example, for the property *being disjoint* for two sets of. Two sets are *disjoint* if they have no common element. The mathematician uses the adjective *non-disjoint*: he/she does not say that two sets *are not disjoint*, but that these two sets are *non-disjoint*. This clearly shows that it is the property *non-disjoint* that is mentally the most important property. This is also the case for MUSCADET, which is better able to handle *positive* properties than *negative* properties. In the first MUSCADET version, moreover, the definition of *non-disjoint* was given, and the fact that two sets were disjoint was expressed by the property $\neg$ *non-disjoint*. MUSCADET2 was able to perform automatically the transformation from the definition of *disjoint*

disjoint$(A,B) \Leftrightarrow \neg\, \exists\, X\,(X \in A \wedge X \in B)$

It notices that the definition begins with a negation, it then replaces this definition by both of the following definitions

not-disjoint$(A,B) \Leftrightarrow \exists\, X\,(X \in A \wedge X \in B)$

and disjoint$(A,B) \Leftrightarrow \neg$ not-disjoint(A,B)

In the TPTP library, the statements are given together with there nature: `hypothesis`, `axiom` or `conjecture` [15]. Their name (even if it contains the word `definition` or `defn`) must not be used by the system. The conjectures in TPTP are the theorems to be proved in MUSCADET. In TPTP there is then a difference between *hypothesis* and *axiom,* which is useless for MUSCADET. On the contrary, in TPTP no distinction can be made between *definition* and *lemma*, although it is important for MUSCADET.

For this reason, all TPTP hypotheses and axioms are analyzed and are classified, either as lemmas or as definitions. Definitions are, roughly, universal properties that are equivalences between a predicative and simple expression and a more complex statement, or equalities between a functional and simple expression and a more complex expression. The matching does not have to be very precise, because if the classification as a lemma or as a definition is crucial for some statements, it is unimportant for most of them.

7. Elimination of functional symbols

Strategies of MUSCADET are designed to work with mathematical or logical predicates rather than with functional symbols. Nevertheless, MUSCADET accepts statements written with functions, but it « eliminates » them by giving a name to functional expressions which will replace this expression in the predicative formula. So, $p(f(a))$ will be replaced by $f(a)$:$b \wedge p(b)$. The symbol ":" is used to

[15] Since version 3, TPTP has types or "roles" *definition* and *lemma* but their use is not that which is described here. Definitions mentioned here are declared as axioms in TPTP.

express that b is the object $f(a)$, and the formula $f(a):b$ will be handled as if it was a predicative formula $p_f(a,b)$.
For formulas with variables it is a little more complicated. A statement of the form $p(f(X))$ where f is a functional symbol is equivalent to the two following statements $\forall Y\,(f(X):Y \Rightarrow p(Y))$ and $\exists Y\,(f(X):Y \wedge p(Y))$. Depending on the context, one or the other of these two statements is preferable. The reasons for this are developed in [Pastre 89].
For this a new quantifier is used, which is named *for the only ... equal to ...* .
For the only Y *equal to* $f(X)$, $p(Y)$ *is true* is noted $(!Y{:}f(X)\ p(Y))$ and represented by `only(f(X)::Y, p(Y))`.
Depending on the context, this quantifier will then be handled either as a $\forall$, or as an $\exists$, or perhaps in a simpler manner or in a somewhat more complicated manner.
For example we saw the treatment of universal conclusions by the rule $\forall$. The rule ! performs almost the same treatment but creates the object only if it does not yet exist.

Rule ! :- *if the conclusion is of the form* $!Y{::}F(X)\ P(Y)$
then if there is a hypothesis of the form $Z{:}F(X)$ *then the new conclusion is* $P(Z)$
else create a new object Z, add the hypothesis $Z{:}F(X)$
and the new conclusion is $P(Z)$

that is in the machine

```
rule(N, concl_only) :- concl(N,only(A::X,B),I),
                  ( hyp(N,A::X1,II) -> traces(...),
                  ; create_object(N,z,X1), % gives names z1, z2, etc.
                    addobject(N,X1), addhyp(N,A::X1,I1),
                    traces(...)
                  ),
                  replace(B,X,X1,B1), newconcl(N,B1,I1)
                  .
```

where
`replace(B,X,X1,B1)` replaces `X` by `X1` in `B` and returns `B1`
`create_object(N,z,X1)` creates and returns in `X1` the first object `z1`, `z2`, etc. not yet created.

In old versions of MUSCADET the names of created objetcts were built from corresponding terms by "flattening" them. This gave traces easier to read, for example

```
powerset(a):powerset_a, a inter b:inter_a_b,
powerset(inter_a_b):powerset_inter_a_b,
powerset_a inter powerset_b:inter_powerset_a_powerset_b
```

etc.
But for longer terms, the ease of reading was lost. Moreover, it became useful to know the order in which objects have been created (to search for object verifying some properties). Then the simple numbering has been chosen (but the predicate `flat` still exists as well as the trace of its call, so that it is easy to come back to this option).

In the super-action *addhyp*(H), such a hypothesis $!Y{:}F(X)\ P(Y)$ is handled as an existential hypothesis, but more simply. In effect, in the super-action *addhyp*(H), if H is an existential hypothesis $\exists X\,p(X)$, it is added as it is (default action) because creating too early the objects X such that $p(X)$ could be catastrophic. This hypothesis will be handled only later (see section 10.5). On the other hand, the hypotheses $!Y{:}F(X)\ P(Y)$ are immediately handled.

To addhyp(H):- *if H is of the form* $!Y{:}F(X)\ P(Y)$
then if there is no hypothesis $F(X){:}Z$ *then create a new object Z*
and add the hypothesis $F(X){:}Z$
in all cases, add the hypothesis $P(Z)$

that is in the machine

```
addhyp(N,H,E) :- ( …
```

```
 …
; H = only(A::X,Y)
   -> (hyp(N,A::X1,_) → true
      ; create_object(N,z,X1),addhyp(N,A::X1,E)
      ),
      replace(Y,X,X1,Y1),addhyp(N,Y1)
 …
) .
```

8. Building rules

After the "elimination" of functional symbols from definitions and lemmas, rules are automatically built from these statements. These rules are more operational than the definitions themselves. Examples of such built rules are given in section 2.

The names of the rules are built from the names of the concepts that are defined or from the names of the lemmas.

The super-action `buildrules` analyses the statements and calls the recursive PROLOG predicate `buildrules(E,N,Concept,Name,Body,Antecedents),` which is a super-action composed of metarules that perform the building of one or several rules in a procedural manner.

`Concept` is the concept that is defined in a definition, or the symbol `lemma` in the case of a lemma.

`Name` is the name of the rule that is currently being built. It is built from `Concept` and has a number if several rules are built from the same `Concept`.

`E` is the statement that is handled. At the beginning it is a piece of statement that is built by `buildrules` for each definition or lemma.

`Body` is the body of the rule, which is empty at the beginning and is filled little by little.

`N` may a number. In this case, this means that the rule is not built from a definition but that the rule is built from a universal hypothesis of the (sub-)theorem of number `N` and will be local only for this sub-theorem.

`Antecedents` is first empty. It then memorizes the antecedents little by little. This parameter was introduced to allow the extraction of the useful trace.

`buildrules(E,N,Concept,Name,Body,Antecedents)` analyses the expression `E` and, depending on its form, may extract sub-formulas for a recursive call, split it into several expressions, add conditions then actions. Adding conditions is also done by a recursive PROLOG predicate the metarules of which may lead to splitting.

9. Activation and order of rules

Activate a rule consists in putting it into the list of active rules, which is memorized by means of the predicate `rulactiv`. Rules will be tried in the listed order. If this order is important, it will have to be stated by metarules.

The super-action `activrul` begins by creating links (`acti_link`) that is it adds the facts `link(0,P)` for all *concepts P* which are in the initial statement of the theorem to be proved, for all those that are in the definitions of the preceding ones, and recursively. The symbols that are *concepts* are those that have a *definition*. They were memorized when rules were built.

Then the rules are activated (`acti_…`) in a order that depends on their type (given or built). Among the rules that were built from definitions, only those corresponding to links that were stated before are activated.

Untill version 2.5, the list of rules was analyzed and reordered if necessary in such a way that if a rule R is more general than another rule R', then R' will be tried before R. The cases only considered concern the rules R and R' such that R may create an object such that P, R' may create a object such that P', and P is more general than P'; then R' had to be before R. In spite of this restriction, this re-ordering was long and as the number of rules of the studied problems increased time was more lost than gained. So it has been removed. But it would nevertheless have to be

restored because the proof may also be more aesthetic. For example, to prove that the composition of two injective mappings f et g is injective, one considers two objects a_o and a_1 which have their images $g_o f(a_o)$ and $g_o f(a_1)$ equal. A rule R, built from the definition of mapping, builds the objects $b_o=f(a_o)$ and $b_1=f(a_1)$. On the other hand, another rule R', c built from the definition of composition, builds the objects $c_o=f(a_o)$ and $c_1=f(a_1)$ such that $g(c_o)=g(c_1)=g_o f(a_o)=g_o f(a_1)$. Then $b_o=c_o=c_1=b_1$ (uniqueness of images of a_o and a_1 and injectivity of g), then that $a_o=a_1$ (injectivity of f). If R' is applied before R, c_o and c_1 are first created and b_o and b_1 are not created because a_o and a_1 have already an image, both objets b_o and b_1 are then not created.

10. Some strategies

The following strategies are rather classic ones. They come from natural deduction. Sometimes it is necessary to avoid carrying out some treatments too early in order to avoid possible infinite branches or too much splitting.

10.1. Processing of universal conclusions and of implications

Their treatments are simple and systematically performed by the rule $\forall$ *and* $\Rightarrow$. These rules have been seen in section 2, their type is *universal* and consequently they have priority.

Rule $\forall$: *if the conclusion is* $\forall X\, P(X)$
then create a new object x and the new conclusion is $P(x)$

Rule $\Rightarrow$: *if the conclusion is* $H \Rightarrow C$
then add the hypothesis H and the new conclusion is C

Their machine expressions are given in section 4.3

10.2. Processing of conjunctive conclusions

The rule

Rule concl_$\wedge$: *if the conclusion is conjunctive*
then successively prove all the elements of the conjunction

is expressed by

```
rule(N, concl_and) :- concl(N, A & B, E),  proconj(N, A & B, E, Eend).
```

with

```
proconj(N,C,Econj, Eend) :-
 (C = (A & B) -> true ; (C=A,B=true) /* for the last one */),
  atom_concat(N,-,N0), gensym(N0,N1), % N1=N-1then2then...
  createsubth(N,N1,A,Ecreationsubth), % creation subth (new step)
  applyrulactiv(N1), % proof of the sub-theorem
  concl(N1,true,Edemsubth),
  newconcl(N,B,Eretourth), % remove A which has just been proved
  (B = true -> Ereturnth=Eend ; proconj(N,B,Ereturnth,Eend)
 ).
```

The conclusion is set to `true` if `proconj` succeeds, that is if all the elements of the conjunction were proved.

`proconj` recursively proves all the conclusions of the conjunctive conclusion `A & B` by creating as many sub-theorems as there are conclusions to be proved. The numbers of the sub-theorems are built from the number `N` by adding an hyphen then successive numbers (`0-1-1`, `0-1-2`, `0-1-3`, … are the sub-theorems of the (sub-)theorem numbered `0-1`).

The new sub-theorem of number `N1` inherits properties of the theorem of number `N` (super-action `createsubth`) except for the conclusion, which is only the sub-formula `A` that is to be proved. It is also pointed out that `N` has `N1` as a sub-theorem.

If the sub-theorem `N1` is proved (conclusion `true`), this information is sent to (sub-)theorem `N` and `A` is removed from the initial conclusion of `N`.

At the end, the conclusion of (sub-)theorem `N` is equal to `true` if and only if all its sub-theorems have been proved.

10.3. Processing of universal hypotheses

There are no universal hypotheses since the super-action *addhyp*, instead of adding them, considers them as lemmas and creates new rules, which are local for the current (sub-)theorem.

10.4. Processing of existential conclusions

The general (and rather sophisticated) treatment of MUSCADET1 has not yet been written in MUSCADET2 nor in the following versions. For the moment, MUSCADET only verifies that objects which satisfy the searched property actually exist.

10.5. Processing of existential hypotheses

An existential hypothesis may lead to the creation, if one does not already exists, of an object that satisfies the indicated property. But this object must not be created each time an existential hypothesis is added because this could lead to generate infinitely many objects in only one direction and to not taking into account other directions. Examples of such situations are analyzed in [Pastre 89, 93].
For this reason, these existential hypotheses are first added without treatment. Later, a rule which does not have high priority performs the treatment of the first existential hypothesis that has not yet been treated, then MUSCADET again applies rules that have higher priority before treating the following existential hypothesis. This allows MUSCADET to create the new objects one by one, and successively in all the directions.

10.6. Processing of disjunctive conclusions

Only simple and logical properties are applied.
A disjunctive conclusion is true if one of the elements of the disjunction is true.
If one of the elements of a disjunctive conclusion is a negation $\neg A$, A is added as a new hypothesis and $\neg A$ is removed from the conclusion.
Some treatments, such as replacing the conclusion by its definition, are also performed on one of the formulas of a disjunctive conclusion.
If the disjunction contains conjunctive sub-formulas, the conjunctions are pushed to the outside in order to obtain a conjunctive conclusion, which in turn will lead to splitting.

10.7. Processing of disjunctive hypotheses

Simple rules handle trivial cases. For example, a hypothesis $A \vee A$ is replaced by A.
Other example, if one of the elements of the disjunction is already a hypothesis or is of the form $X=X$, this hypothesis is removed [16].

Then, splitting may be done, but, as for existential hypotheses, the disjunctive hypotheses are first added without treatment. Later, a rule, which does not have high priority, performs the treatment of the first disjunctive hypothesis $A \vee B$ that has not yet been treated and prepares splitting by replacing the conclusion C par $(A \Rightarrow C) \wedge (B \Rightarrow C)$. Splitting will then be done by the rule *concl_∧* .
It is important not to split too early in order not to multiply splitting uselessly in the case of many useless disjunctive hypotheses.

10.8. Knowledge specific to certain domains

Knowledge specific to topological linear spaces [Pastre 89], which were given in MUSCADET1 in the form of operational rules, have not yet been put into the following versions. Rather than directly

[16] It is not physically removed, the fact that it has been treated is only memorized

translating them by PROLOG clauses, I intend to write them as mathematical statements and to write metarules, which will be able to generate these operational methods.
On the other hand, work on discrete geometry [Pastre 93] was continued in MUSCADET2, and this allowed us to obtain more satisfactory results while helping the system less.

11. Second order statements

We saw in section 2.1. that MUSCADET is able to work in second order predicate calculus. So, for the example in section 2.1 (example file `example-order2`), the property of transitivity for a relation may be defined by

$$\text{transitive}(R) \Leftrightarrow \forall A\ \forall B\ \forall C\ [R(A,B) \wedge R(B,C) \Rightarrow R(A,C)]$$

or $$\text{transitive}(R,E) \Leftrightarrow \forall A \in E\ \forall B \in E\ \forall C \in E\ [R(A,B) \wedge R(B,C) \Rightarrow R(A,C)]$$

and the theorem may be

$$\text{transitive}(\subset)$$

or $$\forall E\ \text{transitive}(\subset, \mathcal{P}(E))$$

that is, for the machine

```
transitive(inc)
```

or

```
transitive(inc,powerset(E))
```

For the definition of `transitive`, since version MUSCADET2, there was some difficulty because the PROLOG syntax does not allow the user to write `R(X,Y)` if `R` is a variable. I then introduced the functional symbol "`..`" which allows `..[R,X,Y]` to be written instead of this forbidden `R(X,Y)`.
The definition is then written

```
definition(transitive(R) <=>
           ![X,Y,Z]:(..[R,X,Y] & ..[R,Y,Z] => ..[R,X,Z]))
```

PROLOG does not unify `r(a,b)` and `..[R,X,Y]`; consequently, I wrote predicates to do it and which return `R=r, X=a and Y=b`.
This symbol "`..`" was chosen by analogy with the PROLOG operator "`=..`". In effect, `r(a,b) =.. [r,a,b]` is true in PROLOG if `r` is a constant. `r(a,b) and ..[r,a,b]` are unified in MUSCADET, although not in PROLOG. On the other hand, MUSCADET replaces `..[r,a,b]` by `r(a,b)` as soon as `r` is a constant in order to produce traces which are easier to read.

Remark: It is possible for the user to really write formulas as in mathematics and to translate, for example by a Unix script, `R(X,Y)` into `..[R,X,Y]` before starting PROLOG.

This notation is also used for mathematical functions and applications. `f(x)::y` may be written and is unified in MUSCADET with `..[F,X]::Y` for the instantiations `F=f, X=x and Y=y`.

The super-action *addhyp* handles these expressions and adds the hypotheses in the more pleasant form `r(a,b)` or `f(x)::y` instead of `..[r,a,b]` or `..[f,x]::y` by the rules

```
addhyp(N,H,E) :- ( …
                   …
                 ; H = ..[R,X,Y] -> H1 =..[R,X,Y], addhyp(N,H1,E)
                 ; H = ..[F,X]:Y -> Y1 =..[F,X], addhyp(N,Y1::Y,E)
                   …
                 ) .
```

Be careful with the spaces: "`=..[… ]`" and "`= ..[…]`" are not the same !
This notation is not compulsory; you may prefer to choose the `apply` symbol for example, which is used in several TPTP problems (or any other symbol), and write everywhere `apply(R,X,Y)` instead of `..[R,X,Y]` and `apply(F,X,Y)` instead of `..[F,X]::Y`. But , as this `apply` symbol may not be known by the prover, for each constant relation, the equivalence must be given, for example

$$\forall X \forall Y (\text{apply}(\subset, X, Y) \Leftrightarrow X \subset Y)$$

that is, for the machine

```
! [X,Y]:(apply(inc,X,Y)<=>inc(X,Y))
```

This has been done for some problems about relations (equivalence, (pre-)order, total order, well-order) which have been proposed for inclusion in the TPTP library, for example the problem SET806+4.p which states that set equality defines a pre-order relation. Moreover, upon request from Geoff Sutcliffe, different names were used for respectively the constants and the relations of arity greater than 0 (in the preceding example `subset_predicate` and `subset`) because other provers could not accept the same symbol for both. Nevertheless MUSCADET, as humans (here the matter is naive set theory), do not have difficulty in using the same symbol. The example in the file `variant_set806+4.p` is the corresponding version of problem `SET806+4.p`.

12. Distribution

MUSCADET4 is available at the address

`http://www.math-info.univ-paris5.fr/~pastre/muscadet`

`muscadet.html` contains short directions for use

`muscadet-en` is the complete PROLOG file [17]

`musca-en` [18] is a Unix script which allows you to work under PROLOG; you must then call `prove`[19], `th`, or `tptp`. See section 5, or type `th.` or `tptp.` (with a point, without any argument) to diplay the instructions for use.

`th-fr.c` and `tptp-fr.c`[20] are small `C` files which allow to work under Linux. Compile them. Let for example `th` and `tptp` be the obtained executables[21]. `th` allows to work with data previously saved in files (see section 5.2). `tptp` allows to work on the TPTP library (see section 5.3). `th` and `tptp` without any arguments diplay the instructions for use.

`examples`[22] gives a directory of examples with data files and executions.

[17] in French `muscadet-fr`

[18] in French `musca-fr`

[19] in French `demontrer`

[20] in French `th-fr.c` et `tptp-fr.c`

[21] commands `cc th-en.c -o th` and `cc tptp-en.c -o tptp`

[22] in French `exemples`

13. References

[Bledsoe 71] - Bledsoe, W.W. Splitting and reduction heuristics in automatic theorem proving, *Journal of Artificial Intelligence* 2 (1971), 55-77

[Bledsoe 77] - Bledsoe, W.W., Non-resolution theorem proving, *Journal of Artificial Intelligence* 9 (1977), 1-35

[Pastre 78] - Pastre D., Automatic theorem Proving in Set theory, *Journal of Artificial Intelligence* 10 (1978), 1-27

[Pastre 89] - Pastre D., MUSCADET: An Automatic theorem Proving System using Knowledge and Metaknowledge in Mathematics - *Journal of Artificial Intelligence* 38.3 (1989)

[Pastre 89a] - Pastre D., MUSCADET: Manuel de l'utilisateur, 1989, Rapport LAFORIA n°89/54, 62p

[Pastre 93] - Pastre D., Automated theorem Proving in Mathematics, *Annals on Artificial Intelligence and Mathematics* 8.3-4 (1993), 425-447

[Pastre 95] - Pastre D., Entre le déclaratif et le procédural: l'expression des connaissances dans le système MUSCADET, *Revue d'Intelligence Artificielle* 8.4 (1995), 361-381

[Pastre 98] - Pastre D., PUSCADET[23]: Manuel de l'utilisateur, 1998, Rapport interne, 62p

[Pastre 99] - Pastre D., Le nouveau MUSCADET et la TPTP Problem Library, colloque sur la Métaconnaissance, Berder (1999), rapport LIP6 2000/002 http://www.lip6.fr/reports/lip6.2000.002.html, 54-98, and http://www.math-info.univ-paris5.fr/~pastre/berder99.ps

[Pastre 01a] - Pastre D., MUSCADET2.3 : A Knowledge-based Theorem Prover based on Natural Deduction, International Joint Conference on Automated Reasoning IJCAR 2001 (Conference on Automated Deduction CADE-JC), 685-689

[Pastre 01b] - Pastre D., Implementation of Knowledge Bases for Natural Deduction, 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, 2nd International Workshop on Implementation of Logics, Cuba, 2001, 49-68

[Pastre 02] - Pastre D., Strong and weak points of the MUSCADET theorem prover, AI Communications, 15 (2002), 147-160, http://www.math-info.univ-paris5.fr/~pastre/AICom/AIC263.pdf

[Pastre 06] - Pastre D., Complementarity of natural deduction and resolution principle, in empirically automated theorem proving, rapport interne, 2006, http://www.math-info.univ-paris5.fr/~pastre/compl-ND-RP.pdf

[Pastre 07] - Pastre D., Complementarity of a natural deduction knowledge-based theorem prover and resolution-based provers in automated theorem proving, rapport interne, 2007, http://www.math-info.univ-paris5.fr/~pastre/compl-NDKB-RB.pdf

[Pastre 10] - Pastre D., Natural proof search and proof writing, Conferences on Intelligent computer mathematics, Workshop on Mathematically Intelligent Proof Search, Paris, 2010, http://www.math-info.univ-paris5.fr/~pastre/mips.pdf, http://www.math-info.univ-paris5.fr/~pastre/slides-mips.pdf

[Sutcliffe 09] - Sutcliffe, G., The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0, Journal of Automated Reasoning 43 (2009), 337-362

[23] PUSCADET was the name given at that time to the first PROLOG version de MUSCADET