MUSCADET version 4
Manuel de l'Utilisateur

http://www.normalesup.org/~pastre/muscadet/manuel-fr.pdf [1]

22/03/2018

Dominique PASTRE [2]
LIPADE - Université Paris Descartes
pastre@phare.normalesup.org

1. Introduction

Le démonstrateur de théorèmes MUSCADET est un système à base de connaissances basé sur la déduction naturelle (au sens de Bledsoe[71, 77]) et utilisant des méthodes proches de celles de l'être humain. Il comprend un moteur d'inférences qui interprète et exécute des règles et une ou plusieurs bases de faits qui sont les représentations internes de « théorèmes à démontrer ».

[1] Version anglaise dans http://www.normalesup.org/~pastre/muscadet/manual-en.pdf

[2] LAFORIA, Université Pierre et Marie Curie (Paris 6) puis CRIP5/LIPADE, Université Paris Descartes (Paris 5)

Les règles sont soit universelles et incorporées au système, soit construites par le système lui-même grâce à des métarègles, à partir des données (définitions et lemmes) fournies par l'utilisateur. Elles sont de la forme *si <liste de conditions> alors <liste d'actions>*. Les conditions sont en principe des propriétés rapidement vérifiées, les actions peuvent être soit des actions élémentaires rapidement exécutées, soit des "super-actions" définies comme des paquets de règles.

La représentation d'un « théorème à démontrer » (ou d'un sous-théorème) est une description de son état pendant la démonstration, constituée des objets qui ont été créés, d'hypothèses, d'une conclusion à démontrer, de règles dites actives, éventuellement de sous-théorèmes, etc. Au début, elle est constituée uniquement d'une conclusion qui est l'énoncé initial du théorème à démontrer et d'une liste de règles dites actives, c'est-à-dire de règles pertinentes pour ce théorème, qui a été construite automatiquement.

Les règles actives quand elles s'appliquent peuvent ajouter de nouvelles hypothèses, modifier la conclusion, créer de nouveaux éléments, créer des sous-théorèmes, construire de nouvelles règles locales à un (sous-)théorème. Quand la conclusion a été mise à *true*, par exemple si on a ajouté comme nouvelle hypothèse la conclusion à démontrer, ou si on a une conclusion existentielle $\exists X\, p(X)$ et une hypothèse $p(a)$, le théorème est démontré. S'il ne s'agit que d'un sous-théorème, cette information est remontée au théorème qui l'a créé.

2. Exemples

Dans tout ce texte, on utilisera les conventions PROLOG pour désigner des constantes (noms commençant par une minuscule) ou des variables (noms commençant par une majuscule ou par le symbole « _ »). De plus, dans les parties informelles, on étendra ces conventions aux prédicats (ce qui n'est pas possible en PROLOG), et d'une manière assez large, on écrira $P(X)$ pour désigner une expression quelconque dépendant de X.

2.1. Transitivité de l'inclusion

Soit à démontrer la transitivité de l'inclusion

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

avec la définition de l'inclusion

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

Recevant l'énoncé du théorème, MUSCADET crée des objets *a, b* et *c* en appliquant trois fois la règle

Règle $\forall$: *si on a la conclusion* $\forall X\, C(X)$
alors créer un nouvel objet x et la nouvelle conclusion est $C(x)$

et la nouvelle conclusion est

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

Puis la règle

Règle $\Rightarrow$: *si on a la conclusion* $H \Rightarrow C$
alors ajouter l'hypothèse H et la nouvelle conclusion est C

remplace la conclusion par $a \subset c$ et ajoute les deux hypothèses $a \subset b$ et $b \subset c$.
En effet, l'ajout de l'hypothèse H ne se fait pas telle quelle : une super-action *ajhyp(H)* comporte, entre autres, la règle

si H est une conjonction,
alors ajouter successivement tous les éléments de la conjonction

(Cette règle s'applique bien sûr récursivement si nécessaire).
La conclusion est ensuite remplacée par sa définition

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

par l'application de la règle

Règle def_concl_pred : *si on a la conclusion C*
il existe une définition de la forme $C \Leftrightarrow D$
alors la nouvelle conclusion est D

Avec les règles précédemment décrites $\forall$ et $\Rightarrow$, on a alors un nouvel objet x, une nouvelle hypothèse $x \in a$, et la conclusion est maintenant $x \in c$.

La règle suivante

Règle $\subset$: *si on a les hypothèses* $A \subset B$ *et* $X \in A$
alors ajouter l'hypothèse $X \in B$

est une règle qui a été construite automatiquement par MUSCADET à partir de la définition de l'inclusion.

Elle s'applique ici deux fois, ajoute les hypothèses $x \in b$ puis $x \in c$ qui est identique à la conclusion à démontrer.

La démonstration se termine par l'application de la règle

Règle stop_hyp_concl : *si la conclusion C est aussi une hypothèse*
alors mettre la conclusion à true

MUSCADET est capable de travailler dans le calcul des prédicats du 2ème ordre et l'exemple précédent peut lui être proposé sous la forme

$$transitive(\subset)$$

accompagné de la définition de la transitivité d'une relation R

$$transitive(R) \Leftrightarrow \forall A\, \forall B\, \forall C\, [\, R(A,B) \wedge R(B,C) \Rightarrow R(A,C)\,]$$ [3]

Après remplacement de la conclusion transitive($\subset$) par sa définition, on est ramené au cas précédent.

On peut aussi travailler sur l'ensemble des parties d'un ensemble

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

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

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

On peut aussi utiliser des quantificateurs relativisés[4] pour alléger l'écriture :

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

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

2.2. Ensemble des parties de l'intersection de deux ensembles

Soit à démontrer le théorème suivant

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

avec la définition de l'intersection

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

de l'ensemble des parties d'un ensemble

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

et de l'égalité d'ensembles

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

Après création des objets a et b comme dans l'exemple précédent, par un mécanisme assez complexe qui sera décrit en section 7, MUSCADET « élimine les symboles fonctionnels » $\cap$ et $\mathcal{P}$ en

[3] Les variables de prédicats n'étant pas possible en PROLOG, on verra en section 11 comment exprimer $R(A,B)$

[4] $\forall X \in A\ P(X)$ est une abréviation pour $\forall X\, (X \in A \Rightarrow P(X))$, $\exists X \in A\ P(X)$ pour $\exists X\, (X \in A \wedge P(X))$

créant et nommant, au moyen de l'opérateur « ; », les objets $a\cap b$:c, $\mathcal{P}(c)$:pc, $\mathcal{P}(a)$:pa, $\mathcal{P}(b)$:pb et $pa\cap pb$:pd. La nouvelle conclusion est alors

$$pc =_{ens} pd$$

Après remplacement de l'égalité de la conclusion par sa définition, soit

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

la règle

Règle concl_∧ : *si on a une conclusion conjonctive*
alors démontrer successivement tous les éléments de la conjonction

crée deux sous-théorèmes de numéros 1 et 2.
La conclusion du premier sous-théorème est $pc\subset pd$ et est remplacée par sa définition

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

un nouvel objet x est créé, une nouvelle hypothèse $x\in pc$ est ajoutée et la nouvelle conclusion est $x\in pd$.
La règle suivante, créée automatiquement à partir de la définition de l'ensemble des parties,

Règle $\mathcal{P}$: *si on a les hypothèses* $\mathcal{P}(A)$:B *et* $X\in B$
alors ajouter l'hypothèse $X\subset A$

donne l'hypothèse $x\subset c$.
La règle suivante [5]

Règle defconcl_elt : *si la conclusion est* $A\in B$
on a une hypothèse Term:B *et une définition* $X\in$Term $\Leftrightarrow$ $P(X)$
alors la nouvelle conclusion est $P(A)$

remplace la conclusion par $x\in pa \wedge x\in pb$

La règle concl_∧ provoque un nouveau découpage, le théorème 11 ayant comme conclusion $x\in pa$ remplacée par $x\subset a$ par la règle *defconcl_elt*.
Par les règles *def_concl_pred,* ∀ et ⇒ *,* t est créé, l'hypothèse $t\in x$ ajoutée et la nouvelle conclusion est $t\in a$, puis la règle ⊂ donne l'hypothèse $t\in c$.
Les règles suivantes ont été créées automatiquement à partir de la définition de l'intersection

Règle ∩1 *: si on a les hypothèses* $A\cap B$:C *et* $X\in C$
alors ajouter l'hypothèse $X\in A$

Règle ∩2 *: si on a les hypothèses* $A\cap B$:C *et* $X\in C$
alors ajouter l'hypothèse $X\in B$

Règle ∩3 *: si on a les hypothèses* $A\cap B$:C, $X\in A$ *et* $X\in B$
alors ajouter l'hypothèse $X\in C$

la règle ∩1 donne alors l'hypothèse $t\in a$ qui termine la démonstration du sous-théorème 11.
Les sous-théorèmes 12 puis 2, correspondant aux autres cas issus des découpages sont ensuite démontrés.

3. De MUSCADET1 à MUSCADET4

Les détails historiques de cette section seront utiles, en particulier, aux utilisateurs d'anciennes versions et aux lecteurs d'articles anciens.

[5] Cette règle des premières versions de MUSCADET a été remplacée par une règle plus générale quand l'appartenance a cessé d'être connue du système (contrainte de la libraire TPTP, voir section 3.2).

3.1 MUSCADET1

Une première version de MUSCADET, qui est maintenant appelée MUSCADET1, a été décrite et analysée dans [Pastre 84, 89, 89a, 93, 95].
MUSCADET1 faisait suite à un premier programme (DATTE, écrit en Fortran !) [Pastre 76, 78] qui était déjà basé sur la déduction naturelle (au sens de Bledsoe [71, 77]) et dont les méthodes étaient proches de celles utilisées par l'être humain.

Le moteur d'inférence de MUSCADET1 était écrit en PASCAL et les connaissances (règles, métarègles et super-actions) dans un langage qui se voulait simple et déclaratif. MUSCADET1 a montré de bons résultats, a été expérimenté pendant plusieurs années mais a montré ses limites. En particulier, le langage n'était pas adapté à l'expression de stratégies procédurales dont l'écriture avait été complexe et qui étaient difficiles à lire et à comprendre.

3.2 MUSCADET2

La version suivante, appelée MUSCADET2 (versions 2.0 à 2.7) [Pastre 01a, 01b, 02, 06, 07] a été entièrement écrite en PROLOG. La raison de ce choix est d'utiliser le même langage pour exprimer des connaissances déclaratives comme les règles, les définitions, les hypothèses, etc., des stratégies de démonstration plus procédurales, et le moteur d'inférence, lui-même réduit à peu de prédicats puisque complété par l'interpréteur PROLOG. Cela a apporté beaucoup de souplesse, de facilité d'écriture, d'efficacité et même de possibilités d'expression pour de nombreuses améliorations et de nouvelles stratégies dont certaines n'auraient pas pu être réalisées dans la première version. Il a également été possible d'utiliser, sans qu'il ait été nécessaire de les réécrire, toutes les facilités d'expression de PROLOG comme le calcul numérique (absent dans MUSCADET1) ou des notations infixes partiellement parenthésées en définissant tout simplement les opérateurs et leur priorités (mais ce n'est pas obligatoire, c'est seulement plus agréable pour l'utilisateur). Pour désigner des variables mathématiques ou des constantes, les conventions de PROLOG sont utilisées (les noms de variables commencent par une majuscule, les noms de constantes par un minuscule), il n'est donc plus nécessaire de préciser si un symbole est une variable ou une constante (mais il faut impérativement respecter cette convention).

MUSCADET2 a pu travailler sur les problèmes de la bibliothèque TPTP (TPTP Problem Library (**T**housands of **P**roblems for **T**heorem **P**rovers), http://www.cs.miami.edu/~tptp). De nouvelles stratégies ont été ajoutées, plus adaptées au style et aux axiomatiques utilisés dans cette bibliothèque.
De plus deux facilités de MUSCADET ont du être abandonnées. La première est la possibilité de déclarer que des données sont soit des définitions, soit des lemmes (ou des théorèmes connus). Ces deux types d'énoncés ne sont pas traités de la même façon et MUSCADET doit maintenant les reconnaître (voir section 6).
La deuxième facilité est le fait que MUSCADET1 connaissait le symbole d'appartenance ensembliste, ce qui n'est pas possible dans le cadre de la bibliothèque TPTP. Les règles qui l'utilisaient ont dû être généralisées. Par exemple, la règle *defconcl_elt* vue en section 2.2 a été remplacée par la règle plus générale

Règle defconcl_rel : *si la conclusion est R*(*A,B*)
on a une hypothèse Term:*B et une définition X*∈Term ⇔ Def
alors la nouvelle conclusion est l'expression obtenue
en remplaçant X par A dans Def

MUSCADET a participé aux compétitions CASC (http://www.cs.miami.edu/~CASC) à partir de 1999. MUSCADET n'a, bien sûr, pu concourir que dans les divisions « premier ordre », c'est-à-dire FOF (FEQ et NEQ), puisqu'il ne travaille pas sur les clauses. Les résultats [Pastre 06, 07] montrent la complémentarité de MUSCADET par rapport aux démonstrateurs basés sur le Principe de résolution.

On trouvera dans [Pastre 99] une analyse de quelques insuffisances de MUSCADET1 améliorées dans MUSCADET2 ainsi que la description de quelques nouvelles stratégies conçues à l'occasion du travail sur la bibliothèque TPTP. Les utilisateurs de MUSCADET1 pourront aussi trouver dans [Pastre 98] une correspondance plus détaillée entre certaines des techniques des deux versions.

Outre les enrichissements continuels des bases de règles et les améliorations des stratégies de démonstration, dans les dernières versions de MUSCADET2, il a été possible, pour les problèmes de la bibliothèque TPTP d'appeler, sous Linux, le démonstrateur par un exécutable C qui appelait lui-même PROLOG et le démonstrateur. L'intérêt en est une plus grand simplicité d'utilisation et la possibilité d'écrire des scripts pour résoudre des listes de problèmes[6]. Par contre l'avantage de travailler sous PROLOG réside dans le fait de pouvoir consulter les bases de faits après exécution ou interruption, et même de tester une règle en la forçant à s'appliquer après interruption.

3.3 MUSCADET3

A partir de 2008, **la syntaxe de MUSCADET3** [Pastre 10a, 10b] **est celle de la bibliothèque TPTP**.
Bien que ce n'ait pas été indispensable, le symbole « : » utilisé dans l'expression « pour le seul Y:f(X) tel que p(Y) » a été remplacé par « :: » pour éviter la confusion avec le « : » de TPTP utilisé dans l'écriture des formules quantifiées.

`qqs(X,p(X))`		`! [X] : p(X)`
`ilexiste(X,p(X))`		`? [X] : p(X)`
`qqs(X,qqs(Y,p(X,Y)))`		`! [X,Y] : p(X,Y)`
`ilexiste(X,ilexiste(Y,p(X,Y)))`	s'écrivent	`? [X,Y] : p(X,Y)`
`A et B`		`A & B`
`A ou B`		`A | B`
`non A`		`~ A`
`seul(f(X):Y,PY)`		`seul(f(X)::Y,PY)`

Le deuxième changement important de la version 3 est l'extraction de la trace « utile » dont l'affichage peut être demandé. Pour cela, il était nécessaire de pouvoir remonter de l'étape finale aux étapes antécédentes. Les étapes ont donc été numérotées, et apparaissent comme nouveaux paramètres des faits, règles, conditions et super-actions. Une étape correspond à l'application effective d'une règle. Une étape peut donc comporter plusieurs actions. Les faits comme les hypothèses et les conclusions sont mémorisés avec le numéro d'étape où il sont été obtenus. Plusieurs faits peuvent donc avoir le même numéro d'étape. L'application, avec succès et effet, d'une règle est mémorisée. Pour permettre la remontée et aussi pouvoir écrire une justification détaillée, et pas seulement la succession des étapes, on mémorise aussi le nom de la règle, la nouvelle étape, les conditions instanciées, la liste des étapes des conditions, les actions instanciées et explicites et un texte donnant une justification qui est soit donné pour les règles générales (en général une explication logique), soit construit automatiquement avec la règle (par exemple, règle construite à partir de la définition de tel concept ou de tel axiome avec leur nom ou règle locale construite à partir de telle hypothèse universelle). Cette mémorisation se fait soit dans la règle soit dans la super-action, en particulier dans le cas d'une super-action récursive comme ajouter une hypothèse ou démontrer une conclusion conjonctive.

3.4 MUSCADET4

Dans la version 4 du démonstrateur proprement dit il n'y a eu que des améliorations minimes.
Les améliorations ont surtout porté sur la rédaction de la trace utile (version 4.0 soumise à la compétition CASC) et sur l'interface qui a rendu son utilisation plus facile et plus complète, à la fois sous Linux ou sous PROLOG (version 4.1). Des options permettent de modifier directement le temps

[6] Un tel exécutable existait déjà dans les versions CASC mais ne donnait que le résultat, non la preuve ni la recherche de la preuve.

limite, le niveau d'affichage (trace complète / trace utile / résultat selon l'ontologie SZS) et le langage.
On trouvera en particulier dans les transparents de [Pastre 10] des extraits de traces utiles.
La version « th » a été rétablie et peut être utilisée sous Linux ou sous PROLOG. On peut alors démontrer un ou plusieurs théorèmes dont les énoncés et les énoncés des définitions et des lemmes se trouvent dans un ou plusieurs fichiers.

4. Représentations informatiques

Tout est exprimé en PROLOG, que ce soit les énoncés mathématiques qui sont des expressions PROLOG, les faits qui sont des clauses unitaires PROLOG, les règles qui sont des clauses PROLOG exprimant des connaissances déclaratives, les actions élémentaires et certaines stratégies qui sont des clauses PROLOG définissant des actions procédurales et les super-actions qui sont des clauses PROLOG regroupant des paquets de règles pour un même but.

Le moteur d'inférence est constitué de l'interpréteur PROLOG et de quelques clauses gérant l'application des règles (`appliregactiv` et `applireg`).

4.1. Expression des énoncés mathématiques

La syntaxe utilisée est celle de la bibliothèque TPTP.
Les connecteurs logiques `&` (et), `|` (ou), `~` (non), `=>`, `<=>` sont définis comme opérateurs infixes avec les priorités habituelles en mathématiques. Ils sont associatifs de gauche à droite.
Les formules quantifiées universellement s'écrivent ! [X,Y,...] : <énoncé dépendant de X, Y, ...>. Les formules existentielles s'écrivent ? [X,Y,...] : <énoncé dépendant de X, Y, ...>. [7]

On peut également utiliser les constantes `true` et `false`.

Le premier théorème donné en exemple en section 2.1 s'écrit

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

La démonstration d'un théorème `T` sera demandée par l'appel PROLOG `demontrer(T).`

La définition de l'inclusion est

```
inc(A,B) <=> ![X]:(app(X,A) => app(X,B))
```

Cette définition est donnée par

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

où `definition` est un prédicat PROLOG déclarant que l'énoncé en argument est une définition mathématique.

L'intersection et l'ensemble des parties sont définies par

```
![X]:(app(X,inter(A,B)) <=> app(X,A) & app(X,B))
![X]:(app(X,parties(A)) <=> inc(X,A))
```

On peut, comme le font les mathématiciens, utiliser des opérateurs infixes `app`, `inc`, `inter` en les déclarant, avec leurs priorités, par les directives PROLOG

```
op(200,xfy,app)
op(200,xfy,inc)
```

[7] Dans MUSCADET2 (et dans les publications correspondantes, elles s'écrivaient `qqs(X,` <énoncé dépendant de `X>)` et `ilexiste(X,` <énoncé dépendant de `X>` et les connecteurs s'écrivaient `et`, `ou`, `non`.

```
op(150,xfy,inter)
```
on écrira alors
```
![A,B,C]:(A inc B & B inc C => A inc C)
A inc B <=> ![X]:(X app A => X app B)
![X]:(X app A inter B <=> X app A & X app B)
![X]:(X app parties(A) <=> X inc A)
```
mais, depuis que MUSCADET est écrit en PROLOG, comme PROLOG affichera
```
A inc B&B inc C=>A inc C
```
on perd plus qu'on ne gagne en lisibilité à l'affichage !

Les mathématiciens écrivent couramment les définitions ensemblistes sous la forme $f(X,..) = \{X ; p(X,\ldots)\}$, par exemple $A \cap B = \{ X; X \in A \wedge X \in B \}$ ou $\mathcal{P}(A) = \{X; X \subset A \}$.
Cette possibilité [8] existe aussi dans MUSCADET sous la forme
```
A inter B = [X,X app A & X app B]
parties(A) = [X, X inc A]
```
mais on devra indiquer explicitement le symbole utilisé pour l'appartenance, par le prédicat « +++ », soit
```
+++(app).
```

Pour alléger l'écriture les mathématiciens utilisent aussi couramment les quantificateurs relativisés

$$\forall X \in A \ \forall Y \in B \ p(X,Y) \quad \text{et} \quad \exists X \in A \ \exists Y \in B \ p(X,Y)$$

qui sont des abréviations pour

$$\forall X \ \forall Y (X \in A \wedge Y \in B \Rightarrow p(X, Y)) \quad \text{et} \quad \exists X \ \exists Y \ (X \in A \wedge Y \in B \wedge p(X,Y)) .$$

Dans MUSCADET2, on pouvait écrire `qqs(X app A,qqs(Y app B,p(X,Y)))` et `ilexiste(X app E, ilexiste(Y app B, p(X,Y))), app` ayant été déclaré infixe,
Cette possibilité sera rétablie dans une prochaine version de MUSCADET sous la forme
`![X app A, Y app B]: p(X,Y)` et `?[X app A, Y app B]: p(X,Y)`
ou, plus généralement, avec n'importe quelle formule à la place de `X app A`, … .

Remarque : les énoncés des théorèmes doivent nécessairement être clos, ceux des définitions peuvent contenir des variables implicitement universelles.

4.2. Expression des faits

Le fait qu'un énoncé `C` est la *conclusion* du *(sous-)théorème à démontrer* de numéro `N` se représente par une clause unitaire PROLOG `concl(N,C,I)` où `I` est le numéro de l'étape où ce fait a été créé. On manipule de même quelques autres propriétés comme être une *hypothèse* (`hyp(N,H,I)`), un *objet* (`obj(N,O)`), un *sous-théorème* (`sousth(N,N1)`), etc.
Pour les *règles actives*, c'est la liste des règles actives pour un (sous-)théorème à démontrer qui est mémorisée par le fait `reglactiv(N,[R1,R2,…])`, `[R1,R2,…]` étant une liste de noms de règles ayant été activées automatiquement dans un ordre qui est important.

4.3. Expression des règles

Des règles (simplifiées) utilisées dans l'exemple de la section 2 sont les suivantes.
- règles données au système :
```
regle(N,=>) :- concl(N,A=>B,Etape),
               ajhyp(N,A,NouvelleEtape), nouvconcl(N,B,NouvelleEtape).
regle(N,!) :- concl(N,(!XX:C),Etape),
              creer_objets_et_remplacer(N,XX,C,C1,Objets),
              nouvconcl(N,C1,NouvelleEtape).
regle(N,def_concl_pred) :- concl(N,C,Etape),definition(Nom, C<=>D),
                           nouvconcl(N,D,NouvelleEtape).
regle(N,stop_hyp_concl):- concl(N,C,Etape1),ground(C), hyp(N,C,Etape2),
                          nouvconcl(N,true,NouvelleEtape).
```

[8] qui avait disparu de MUSCADET3 mais a été rétablie dans MUSCADET4.1

- règle construite par le système :

```
regle(N,inc) :- hyp(N, inc(A,B), Etape1), hyp(N, app(X,A), Etape2),
                not hyp(N, app(X,B),_),
                ajhyp(N, app(X,B), NouvelleEtape).
```

Le paramètre `N` sert à appliquer une règle au (sous-)théorème de numéro `N`.

On remarquera que le *si ... alors ...* est implicite. On aurait pu définir des symboles d'opérateurs `si … alors …` en PROLOG, mais cela n'était pas indispensable, puisque tout se traduit en PROLOG par des prédicats.
Les conditions sont en général des vérifications de l'existence (ou de l'absence) de faits qui sont des clauses PROLOG unitaires (`hyp`, `concl`, voir section précédente) ou d'une définition d'une certaine forme. Il peut y avoir aussi des conditions élémentaires.
Les actions sont soit des actions élémentaires exprimées sous forme de prédicats PROLOG traduisant des programmes élémentaires (`creer_objet_et_remplacer`), soit des super-actions définies par des paquets de règles (`ajhyp`, `nouvconcl`, etc, voir section suivante).

La condition `not hyp(N, app(X,B),_)` évite d'appliquer la règle si le théorème de numéro `N` comporte déjà l'hypothèse `app(X,B)`. (Dans l'exemple, cette condition évite seulement l'appel à ajhyp qui serait sans effet, mais dans d'autres cas, elle est essentielle, par exemple pour éviter une création infinie d'objets.)
En MUSCADET1, cette condition n'était pas nécessaire car une règle ne pouvait pas s'appliquer deux fois pour les mêmes instantiations. Cela avait d'autres inconvénients, comme par exemple ne pas pouvoir *forcer* une règle à s'appliquer de nouveau pour les mêmes instanciations.

L'action élémentaire `creer_objets_et_remplacer(N,XX,C,C1,Objets)` renvoie dans `Objets` une liste de constantes `z`, `z1`, `z2`,… etc qui n'ont pas encore été utilisées et remplace dans `C` les variables de `XX` par ces constantes pour donner `C1`.

En réalité, les règles opérationnelles sont plus complexes, elles comportent des conditions et actions additionnelles, qui ont été ajoutées à la main[9] pour les règles données au système, mais qui sont ajoutées automatiquement pour les règles construites par le système. Les premières peuvent être vues dans le fichier `muscadet-fr`, les dernières dans les fichier des règles construites `reg_`...

*Parmi les condition*s :
- le numéro de l'`Etape` qui sera utilisé dans `traces` (voir plus loin)

Parmi les actions :
- la numérotation des étapes
- l'écriture de messages
- `traces(N,regle(Nom),`<condition ou liste de conditions>`,`
  <action ou liste d'actions>, <étape>, <explication>,
  <liste des étapes des conditions>`)`

mémorise les informations qui seront nécessaires pour extraire plus tard la trace utile.

4.4. Expression des super-actions

Les *super-actions* sont en général exprimées sous forme de paquets de règles *si ... alors ...* ou *si ... alors ... sinon ...*
Pour action(*X*): *si ... alors ...*
si ... alors ...

[9] car ajoutées progressivement dans les phases d'expérimentation mais elles auraient pu être ajoutées automatiquement à partir des règles simplifiées

sinon ...

s'exprime facilement en PROLOG

```
action(X) :- ( … -> …
              ; … -> …
              ; …       % par défaut (facultatif)
              ) .
```

La super-action `nouvconcl`, qui ne comporte qu'une règle, remplace la conclusion du (sous-)théorème de numéro `N` par `C` si celle-ci n'est pas déjà égal à `C`.

```
nouvconcl(N,C,E2) :- not concl(N,C,_),
                     etape_action(Etape),
                     affecter(concl(N,C,Etape)).
```

avec

```
etape_action(E1) :- ( var(E1),etape(E) -> E1 is E+1,affecter(etape(E1))
                    ; true).
```

Si `E1` n'a pas été instancié, ce qui est le cas dans la règle « `!` », le numéro d'étape est incrémenté de `1`.

Si `E1` a été instancié, ce qui est le cas dans la règle =>, où `E1` a été instancié par la première action `ajhyp` (décrite ci-après), il s'agit de la même étape.

`affecter` met à jour la conclusion et le numéro d'étape.

La super-action `ajhyp` traite les hypothèses que l'on veut « ajouter » pour n'ajouter finalement que des hypothèses élémentaires non universelles. Les conjonctions sont découpées. Les hypothèses universelles ne sont pas ajoutées, à la place on crée de nouvelles règles qui seront locales au (sous-)théorème en cours de démonstration.

Voici quelques-unes des règles définissant cette super-action

```
ajhyp(N,H,E) :- etape_action(E),
      ( % pour ajouter une conjonction, on ajoute successivement
        % (et récursivement) les éléments de la conjonction
        H = A & B -> ajhyp(N,A,E), ajhyp(N,B,E)
      ; % si H est déjà une hypothèse, on ne fait rien
        hyp(N,H,_) ->  true
      ; % de même que pour une égalité triviale
        H = (X = X) -> true
      ; % ordre lexicographique sauf pour les objets créés z<nombre>
        %       qui sont dans l'ordre de création (numérique)
        H = (Y=X), atom(X), atom(Y), avant(X,Y), ajhyp(N,(X=Y),E2)
      ; % si H est universelle ou est une implication, on crée
        % une (des) règles locale(s) au théorème de numéro N
        H = ( !_: _) -> creer_nom_regle(reghyp,Nom),
                                consreg(H,_,N,Nom,[])
      ; H = A => B -> creer_nom_regle(reghyp,Nom),
                              consreg( H,_,N,Nom,[])
        ...
      ; % sinon  on ajoute l'hypothèse H
        assert(hyp(N,H,E)),
        ecrire1([N, 'ajouter hypothese',H])
      ) .
```

5. Comment utiliser MUSCADET4

Le *paquet* est constitué d'un fichier source PROLOG `muscadet-fr`, d'un script `musca-fr` permettant de travailler sous PROLOG et de deux petits fichiers C `th-fr.c` et `tptp-fr.c` [10] que l'on peut

[10] `muscadet-en, musca-en, th-en.c et tptp-en.c` pour la version anglaise

compiler et qui permettent de travailler sous Linux. On appellera dans la suite de ce texte `th` et `tptp` les exécutables obtenus.

On peut aussi travailler sous PROLOG, ce qui permet en particulier d'avoir accès, en fin de démonstration (ou surtout en cas d'échec ou de plantage !) à tous les faits représentant l'état du théorème à démontrer : `hyp`, `concl`, `sousth`, règles (en particulier les règles construites), etc ; et même de tester directement l'application d'une règle sur l'état à ce moment-là.

On peut travailler à partir d'un fichier contenant une liste de définitions et un ou plusieurs théorème(s) à démontrer.

On peut aussi travailler directement sur la base de problèmes TPTP, après avoir défini une variable d'environnement shell `TPTP` vers la base TPTP (`setenv TPTP` <répertoire de la base>).
Sous PROLOG on peut aussi appeler directement le prédicat `demontrer` avec comme paramètre l'énoncé du théorème à démontrer, après avoir donné les définitions des concepts mathématiques éventuellement utilisés.

Le PROLOG utilisé est SWI-Prolog, logiciel du domaine public téléchargeable à l'adresse suivante `http://www.swi-prolog.org/Download.html`, et que l'on appelle par la commande `swipl`.
Attention : à partir de la version `SWI-Prolog 7` il est nécessaire d'utiliser une version de `Muscadet` postérieure à `4.6.2`.[11]

Dans tous les cas, se placer dans un répertoire contenant le fichier PROLOG `muscadet-fr`[12] (ou un lien de même nom vers ce fichier)

5.1 Démonstration directe

(uniquement sous PROLOG)

Appeler le script Unix `musca-fr`[13]. qui appelle PROLOG et charge le fichier `muscadet-fr`.

Pour démontrer $p \wedge q \Rightarrow (p \Leftrightarrow q)$ taper simplement `demontrer(p & q => (p <=> q)).`
Ne pas oublier le point. Pas d'espace avant la parenthèse.

Pour l'exemple du paragraphe 2.1 rentrer la définition de l'inclusion

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

puis demander la démonstration du théorème

```
demontrer(![A,B,C]:(inc(A,B) & inc(B,C) => inc(A,C))).
```

Ne pas oublier les points.

La preuve sera affichée sur la sortie standard.

Si l'on préfère utiliser des opérateurs infixes, taper

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

puis

```
demontrer(![A,B,C] : A inc B & B inc C => A inc C).
```

[11] à cause de la modification du type de la liste vide [] (qui était un atom dns les versions précédentes)

[12] `muscadet-en` pour la version anglaise

[13] `musca-en` pour la version anglaise

5.2 A partir de fichiers contenant théorèmes et définitions

Les données doivent être rentrées sous la forme suivante :

`:- op(`<priorité>, <type>, <nom>`).` (éventuellement)
`theoreme(`<nom>`,`<théorème à démontrer>`).`
`definition(`<definition>`).`
`lemme(`<nom>`,`<lemme>`).`
`include(`<fichier de données>`).`
`%` <commentaire PROLOG>

Ne pas oublier les points ! (Ce sont des termes PROLOG.)
Toutes ces données peuvent être écrites dans n'importe quel ordre.
Il peut y avoir plusieurs théorèmes qui seront démontrés l'un après l'autre.
`include` permet de mettre des données dans un ou plusieurs fichier(s) séparé(s).

Exemples de fichiers :

```
exemple1 :
    definition(inc(A,B) <=> ! [X]:(app(X,A) => app(X,B))).
    theoreme(thI03,![A,B,C]:(inc(A,B) & inc(B,C) => inc(A,C))).

exemple1bis :
    :- op(200,xfy,app).
    :- op(200,xfy,inc).
    theoreme(thI03,![A,B,C]:(A inc B & B inc C => A inc C)).
    definition(A inc B <=> ! [X]: X app A => X app B)).

exemple2 :
     include(exemple2_definitions).
     % transitivite de l'inclusion
     theoreme(thI03,![A,B,C]:(inc(A,B) & inc(B,C) => inc(A,C))).
     % l'ensemble des parties d'une intersection est égale à
l'intersection des parties
     theoreme(thI21,![A,B]:(inc(parties(inter(A,B)),
                                inter(parties(A),parties(B))))).
avec exemple2_definitions :
     definition(inc(A,B) <=> ! [X]:(app(X,A) => app(X,B))).
     definition(app(X,parties(A))<=> inc(X,A)).
     definition(app(X,inter(A,B))<=> app(X,A) & app(X,B))).

exemple2bis :
     include(exemple2bis-definitions).
     :- op(200, xfy, app).
     :- op(200, xfy, inc).
     :- op(150,xfx,inter).
     % transitivite de l inclusion
     theoreme(thI03,![A,B,C]:(A inc B  &  B inc C  =>  A inc C)).
     %  l'ensemble des parties d'une intersection est égale à
l'intersection des parties
     theoreme(thI21,![A,B] : (parties(A inter B) inc parties(A) inter
parties(B))).
avec exemple2bis_definitions :
     :- op(200, xfy, app).
     :- op(200, xfy, inc).
     :- op(150,xfx,inter).
     definition(A inc B <=> ! [X]:(X app A => X app B)).
     definition((X app parties(A))<=> X inc A).
```

```
definition(X app  A inter B  <=>  X app A  &  X app B).
```
Remarque : la définition d'opérateur infixe doit être faite dans tous les fichiers où il apparaît.

Sous Linux (resp. PROLOG), on appellera l'exécutable (resp. prédicat) `th` avec comme argument un fichier (ou un chemin vers un fichier). Sous PROLOG cet argument doit être un atome donc entre quotes si nécessaire conformément aux conventions PROLOG .
Exemples
```
th exemple1
```
(resp. `th(exemple1).`)

Sous PROLOG on peut aussi appeler `th` avec un fichier ne contenant que des définitions qui seront ainsi lues et mémorisées. Dans ce cas il ne faudra pas mettre l'`include` correspondant dans le fichier des théorèmes, ce qui aurait pour effet de dupliquer les définitions (donc aussi les règles).[14]

La preuve sera enregistrée dans un fichier appelé `res_`<nom du théorème à démontrer> (par exemple `res_th1`). De plus quelques messages ont affichés à la console pour suivre le travail du démonstrateur.

5.3 A partir de la base de problèmes TPTP

(http://www.cs.miami.edu/~tptp**)**
Sous Linux (resp. PROLOG) on appellera l'exécutable (resp. prédicat) `tptp` avec comme argument un chemin vers un fichier de problème TPTP ou un nom de problème TPTP. Dans ce dernier cas, il est nécessaire d'avoir défini une variable d'environnement shell `TPTP` vers la base TPTP. (`setenv TPTP` <répertoire de la base>).
Exemples
```
tptp /home/dominique/$TPTP/Problems/SET/SET002+4.p
tptp SET027+4.p
```
Sous PROLOG, l'argument doit être un atome PROLOG donc les quotes sont nécessaires (présence de -, +, ., /, majuscules)
```
tptp('/home/dominique/$TPTP/Problems/SET/SET0 02+4.p').
tptp('SET027+4.p').
```

La preuve sera enregistrée dans un fichier appelé `res_`<nom du problème> (par exemple `res_SET002+4.p`). De plus quelques messages ont affichés à la console pour suivre le travail du démonstrateur.

5.4 Modification des options par défaut

5.4.1 Options par défaut
Les options par défaut (fichier `muscadet-fr`) sont :
- temps limite (`tempslimite`) : 10 secondes
- affichage de la trace complète de la recherche de la preuve (`tr`) : non
- affichage de la preuve utile (`pr`) : oui
- résultat selon l'ontologie SZS (`szs`) : non

Elles peuvent être modifiées.

[14] Jusqu'à la version 2.6, on pouvait lire des définitions seules, construire et mémoriser les régles construites dans un fichier qui pouvait être lu au cours d'une exécution ultérieure. En effet, la construction des règles était relativement longue par rapport à la démonstration des théorèmes. Cette possibilité a été supprimée dans les réécritures de la version 2.7. Etant donnés les progrès en temps d'exécution des machines modernes, la rétablir me semble superflu. Pour les très grosses bases de données (comme pour les très gros problèmes de TPTP), il serait nécessaire de sélectionner les définitions et axiomes en fonction des conjectures proposées, avant la construction des règles, non de seulement sélectionner les règles après la construction.

Pour les afficher, taper

`listing(tempslimite).` ou `l(tempslimite).` ou `tempslimite(T).`

`listing(afficher).` ou `l(afficher).` ou `afficher(A).`

5.4.2. Modifications à l'appel d'une démonstration

On peut donner comme arguments supplémentaires de `th` ou `tptp`, à la suite et dans n'importe quel ordre, un nouveau temps limite, et des nouvelles options d'affichage sous la forme ± <option>

\+ pour ajouter,

\- pour retirer

<option> ::= `tr` pour la trace complète
`pr` pour la preuve utile
`szs` pour le résultat selon l'ontologie SZS

Exemples

(Linux) `th exemple_th 50 +tr` ou (Prolog) `th(exemple_th,50,+tr).`
met le temps limite à 50 secondes et affiche la trace complète puis la preuve utile.

(Linux) `th exemple_th -pr` ou (Prolog) `th(exemple_th,-pr).`
n'affiche que le résultat (`theoreme demontre` ou `non demontre`) et le temps passé

(Linux) `tptp SET027+4.p -pr +szs` ou (Prolog) `tptp('SET027+4.p',-pr,+szs).`
n'affiche que le résultat selon l'ontologie SZS.

5.4.3 Modifications sous PROLOG

Les commandes suivantes modifient les options jusqu'à nouvel ordre

`affecter(tempslimite(`< temps>`)).` ou `limitertemps(`< temps>`)).`

`assert(afficher(`<option>`)).`

`retract(afficher(`<option>`)).`

5.4.4 Modifications du code PROLOG (fichier `muscadet-fr`)

Enfin, on peut modifier les valeurs par défaut dans le fichier `muscadet-fr` en modifiant la ligne `tempslimite(...).` ou en supprimant ou ajoutant des lignes `afficher(...).`.

5.4.5 Modes d'emploi en ligne

On peut retrouver les modes d'emploi en tapant `th` ou `tptp` sous Linux, ou `th.` ou `tptp.` sous PROLOG.

5.4.6 Modification du choix du langage.

On peut modifier le choix par défaut du langage pour la rédaction des traces (trace complète et/ou trace utile) et autres commentaires en donnant `en` (ou `fr`) comme argument supplémentaire dans la commande ou le prédicat `th` ou `tptp`.

Exemples : `th exemple1 en` ou `th(exemple1,en).`

Sous PROLOG on peut aussi le modifier directement par

```
affecter(lang(en)).
```

ou simplement `en.`

Cependant les noms des règles et des actions ne seront pas modifiés. Pour cela il faut utiliser la version anglaise (fichiers `*-en*`)

5.4.7 Effacement

Sous PROLOG, pour effacer toutes les données relatives au dernier problème et pouvoir ainsi en résoudre un nouveau problème sans devoir sortir et rentrer sous PROLOG, taper :

`effacertout.`

Pour effacer seulement les faits qui représentent l'état du dernier théorème démontré (ou non démontré ...) taper :

`effacerth.`

qui efface tous les faits qui représentent l'état de l'ancien théorème, mais les définitions et les règles construites ne sont pas effacées

6. Définitions et lemmes

Les définitions ne sont pas les seules connaissances mathématiques utilisées par le système. Il dispose aussi de lemmes déclarés au moyen d'un prédicat PROLOG `lemme`(<nom>, <énoncé>).
Ces lemmes peuvent être donnés au système (dans la version `th`).
D'autre part, à la lecture des données (hors conjectures) des problèmes de la bibliothèque TPTP MUSCADET décide s'il les déclare comme lemmes[15] ou comme définitions[16].
Par exemple,

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

sera une définition, mais

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

qui est un résultat (très) connu, éventuellement démontré par ailleurs par le système, sera un lemme.

Définitions et lemmes conduisent également à la construction de règles, mais pas exactement de la même façon. Par exemple, dans une conclusion, $A \subset C$ sera remplacé par sa définition $\forall X\ (X \in A \Rightarrow X \in C)$ mais non par la propriété suffisante $A \subset B \wedge B \subset C$ (lemme ci-dessus).

A partir d'un lemme comportant la propriété $p(X) \Rightarrow q(f(X))$, la règle suivante est créée

si on a $p(X)$ et $f(X):Y$ *alors ajouter* $q(Y)$

ainsi que, sous certaines conditions, la règle

si on a $p(X)$ *alors créer Y et ajouter les hypothèses* $f(X):Y$ *et* $q(Y)$

Par contre, de la définition de l'ensemble des parties, par exemple

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

seule la règle suivante est créée

si on a $X \in A$ *et* $\mathcal{P}(A)$*:PA alors ajouter* $X \subset A$

mais pas la règle suivante

si $X \subset A$ *alors créer l'ensemble des parties de A*

qui pourrait conduire à trop de créations d'objets.

Enfin, MUSCADET va quelquefois construire de nouvelles définitions plus adaptées à ses stratégies. C'est le cas par exemple pour la propriété d'être disjoints pour deux ensembles. Deux ensembles sont *disjoints* s'ils n'ont pas d'élément commun. Le mathématicien utilise l'adjectif *non-disjoint* : il ne dit pas que deux ensembles *ne sont pas disjoints*, mais il dit que ces deux ensembles sont *non-disjoints*. Cela montre bien que c'est la propriété *non-disjoint* qui est mentalement la propriété la plus importante. C'est également le cas pour MUSCADET qui est plus apte à traiter des propriétés *positives* que des propriétés *négatives*. Dans la première version de MUSCADET, on donnait d'ailleurs

---

15 par exemple dans le domaine 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) )   )).
```

16 comme `%----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) ) ) )).
```

la définition de *non-disjoint* et on exprimait le fait que deux ensembles étaient disjoints par la propriété ¬ *non-disjoint*. Puis MUSCADET a été capable de faire automatiquement la transformation à partir de la définition de *disjoint*

disjoint(A,B) ⇔ ¬ ∃ X (X∈A ∧ X∈B)

Il repère que la définition commence par une négation, il remplace alors cette définition par les deux définitions suivantes

non-disjoint(A,B) ⇔ ∃ X (X∈A ∧ X∈B)

et disjoint(A,B) ⇔ ¬ non-disjoint(A,B)

Dans la bibliothèque TPTP, les énoncés sont donnés avec leur nature : `hypothesis`, `axiom` ou `conjecture`.[17] Leur nom (même s'il contient le mot `definition` ou `defn`) ne doit pas être utilisé par les systèmes. Les conjectures de TPTP sont les théorèmes à démontrer de MUSCADET. Dans TPTP on fait donc une différence entre *hypothesis* et *axiom* qui est inutile pour MUSCADET, mais on ne peut pas faire de différence entre *définition* et *lemme* alors que c'est important pour MUSCADET qui ne les traite pas de la même façon.

C'est pourquoi tous les *axioms* et *hypotheses* de TPTP sont analysés dès leur lecture et sont classés, soit comme lemmes, soit comme définitions. Les définitions sont, en gros, les propriétés universelles qui sont des équivalences entre une expression prédicative simple et un énoncé plus complexe ou bien une égalité entre une expression fonctionnelle simple et une expression plus complexe. Le filtre n'a pas besoin d'être très fin, car si le classement comme lemme ou comme définition est crucial pour certains énoncés, il est indifférent pour la plupart.

7. Elimination des symboles fonctionnels

Les stratégies de MUSCADET sont conçues pour travailler avec des prédicats mathématiques ou logiques plutôt qu'avec des symboles fonctionnels. Néanmoins, MUSCADET accepte des énoncés écrits avec des fonctions, mais il les « élimine » en donnant un nom aux expressions fonctionnelles qui remplacera cette expression dans la formule prédicative. Ainsi $p(f(a))$ sera remplacé par $f(a):b \wedge p(b)$. Le symbole « : » sert à exprimer que b est l'objet $f(a)$ et la formule $f(a):b$ sera traitée comme une formule prédicative $p_f(a,b)$.

Pour des formules avec variables, c'est un peu plus compliqué. Un énoncé de la forme $p(f(X))$ où f est un symbole fonctionnel est équivalent aux deux énoncés suivants $\forall Y\,(f(X):Y \Rightarrow p(Y))$ et $\exists Y\,(f(X):Y \wedge p(Y))$. Selon le contexte, l'un ou l'autre de ces énoncés est préférable. Les raisons en sont développées dans [Pastre 84, 89].

Pour cela un nouveau quantificateur est utilisé appelé *pour le seul ... égal à ...* .

Pour le seul Y égal à f(X) *on a p*(Y) est noté (!Y:f(X) p(Y)) et représenté par `seul(f(X)::Y, p(Y))`

Le quantificateur sera alors, selon le contexte, traité comme un ∀, comme un ∃, éventuellement d'une manière plus simple ou un peu plus compliquée.

Par exemple on a vu le traitement des conclusions universelles par la règle ∀ . La règle ! fait à peu près le même traitement mais ne crée l'objet que s'il n'existe pas déjà.

Règle ! :- *si la conclusion est de la forme* !Y::F(X) P(Y)
alors si on a une hypothèse de la forme Z:F(X) *alors la nouvelle conclusion* est P(Z)
sinon créer un nouvel objet Z, ajouter l'hypothèse Z:F(X)
et la nouvelle conclusion est P(Z)

soit en machine

```
regle(N, concl_seul) :-
      concl(N,seul(A::X,B),I),
      ( hyp(N,A::X1,II) -> traces(...)
      ; creer_objet(N,z,X1), % donne noms z1, z2, etc
```

[17] Depuis la version 3 , TPTP a des types ou « roles » *définition* et *lemma* mais leur utilisation n'est pas celle décrite ici. Les définitions dont il est question dans ce texte sont déclarées comme *axiom*.

```
        ajobjet(N,X1),ajhyp(N,A::X1,I1),
        traces(...)
    ),
    remplacer(B,X,X1,B1), nouvconcl(N,B1,I1)
    .
```

où
`remplacer(B,X,X1,B1)` remplace `X` par `X1` dans `B` et renvoie `B1`
`creer_objet(N,z,X1)` crée et renvoie dans `X1` le premier objet `z1`, `z2`, etc. non encore créé.

Dans les anciennes versions de MUSCADET les noms des objets crées étaient formés à partir des termes correspondant par une « mise à plat », ce qui rendait la trace plus facile à lire, par exemple
`parties(a):parties_a, a inter b:inter_a_b,`
`parties(inter_a_b):parties_inter_a_b,`
`parties_a inter parties_b:inter_parties_a_parties_b`
etc.
Mais pour des termes un peu longs, la facilité de lecture était perdue. De plus il s'est avéré utile de garder trace de l'ordre dans lequel les objets avaient été créés (pour la recherche d'objets vérifiant certaines propriétés). La numérotation simple a donc été choisie (mais le prédicat `plat` existe toujours ainsi que la trace de son appel, on peut donc facilement revenir à cette option).

Dans la super-action *ajhyp*(*H*), une telle hypothèse !*Y*:*F*(*X*) *P*(*Y*) est traitée comme une hypothèse existentielle, mais plus simplement. En effet, dans la super-action *ajhyp*(*H*), si *H* est une hypothèse existentielle $\exists X\, p(X)$, elle est ajoutée telle quelle (action par défaut) car il pourrait être catastrophique de créer trop tôt les objets *X* tels que *p*(*X*). C'est seulement plus tard qu'elle sera traitée (voir section 10.5). Les hypothèses !*Y*:*F*(*X*) *P*(*Y*) sont, elles, traitées tout de suite.
Pour ajhyp(*H*) *:-* *si H est de la forme* !*Y*:*F*(*X*) *P*(*Y*)
alors *si on n'a pas d'hypothèse F*(*X*):*Z alors créer un nouvel objet Z*
et ajouter l'hypothèse F(*X*):*Z*
dans tous les cas, ajouter l'hypothèse ,
soit en machine

```
ajhyp(N,H,E) :-
    ( …
      …
    ; H = seul(A::X,Y)-> ( hyp(N,A::X1,_) → true
                         ; creer_objet(N,z,X1),ajhyp(N,A::X1,E)
                         ),
                         remplacer(Y,X,X1,Y1),ajhyp(N,Y1,E)
    ; …
    ) .
```

8. Construction des règles

Après l'« élimination » des symboles fonctionnels des définitions et des lemmes, des règles sont construites automatiquement à partir de ces énoncés. Ces règles sont plus opératoires que les définitions elles-même. Des exemples de telles règles construites ont été donnés en section 2.
Des règles sont aussi construites à partir des hypothèses universelles d'un théorème à démontrer.
Le nom des règles est construit à partir du nom du concept défini ou du nom du lemme.

La super-action `consreg` analyse les énoncés et appelle le prédicat récursif PROLOG. `consreg(E,N,Nomfof,Concept,Nom,Cond,Antecedents)` qui est une super-action composée de métarègles effectuant la construction d'une ou plusieurs règles de manière procédurale.

`E` est l'énoncé traité, issu d'une définition ou un lemme ou d'une hypothèse universelle.
`N` est soit une variable non instanciée et les règles construites s'appliqueront à tous les (sous-) théorèmes, soit un numéro de sous-théorème et les règles ne s'appliqueront qu'à celui-ci et ses descendants.
`Nomfof` est soit le nom de l'énoncé initial (définition ou lemme), soit une chaine préfixée par `r_hyp_univ` (hypothèse universelle). Il ne sert qu'au confort du lecteur.
`Concept` est soit le nom du concept défini dans le cas d'une définition, le symbole `lemme` dans le cas d'un lemme ou `r_hyp_univ` dans le cas d'une hypothèse universelle.
`Nom` est le nom de la règle en cours de construction. Il a été construit à partir de `Concept` et est suivi d'un numéro si plusieurs règles sont construites à partir d'un même `Concept`.
`Cond` est la partie conditions de la regle déjà construite (vide au début).
`Antecedents` est la liste des numéros des étapes des conditions déjà construites (vide au début). Ce paramètre a été introduit pour permettre l'extraction de la trace utile.
`consreg(E,N,Nomfof,Concept,Nom,Cond,Antecedents)` examine l'expression `E` et selon sa forme, peut en extraire des sous-formules pour un appel récursif, découper en plusieurs morceaux, ajouter des conditions puis des actions, et mémoriser la ou les règles ainsi construites.

9. Activation et ordre des règles

Activer une règle consiste à la mettre dans la liste des règles actives, mémorisée par le prédicat `reglactiv`. Les règles seront essayées dans l'ordre où elles se trouvent dans la liste. Si cet ordre est important, il devra être établi par des métarègles.
La super-action `activreg` commence par créer des liens (`acti_lien`) c'est-à-dire ajoute les faits `lien(0,P)` pour tous les *concepts P* qui figurent dans l'énoncé initial du théorème à démontrer, ceux qui sont dans les définitions des précédents, et récursivement. Les symboles qui sont des *concepts* sont ceux qui ont une *définition*. Ils ont été mémorisés au moment de la construction des règles.
Puis les règles sont alors activées (`acti_…`) dans un ordre qui dépend de leur type (donné ou construit). Parmi les règles construites à partir des définitions, seules celles correspondant aux liens précédemment établis sont activées.

Jusqu'à la version 2.5, la liste de règles était examinée et réordonnée si nécessaire de façon à ce que si une règle R est plus générale qu'une autre règle R', R' soit essayée avant R. Les seuls cas considérés concernaient les règles R et R' telles que R est susceptible de créer un objet tel que P, R' est susceptible de créer un objet tel que P', et P est plus général que P'; alors R' devait être avant R. Malgré cette restriction, ce réarrangement était long et avec l'augmentation du nombre de règles des problèmes traités faisait perdre plus de temps qu'il n'en gagnait. Il a donc été supprimé. Il faudrait le rajouter car on gagnait aussi parfois en esthétique. Par exemple, pour montrer que la composée de deux applications injectives *f* et *g* est injective, on considère deux objets a_o et a_1 dont les images par g_of sont égales. Une règle R, construite à partir de la définition d'une application construit les objets $b_o=f(a_o)$ et $b_1=f(a_1)$. D'autre part une autre règle R', construite à partir de la définition de la composée, construit les objets $c_o=f(a_o)$ et $c_1=f(a_1)$ tels que $g(c_o)=g(c_1)=g_of(a_o)=g_of(a_1)$. On trouve alors que $b_o=c_o=c_1=b_1$ (unicité des images de a_o et a_1 et injectivité de g) puis que $a_o=a_1$ (injectivité de f). Si R' est appliquée avant R, on commence par créer c_o et c_1 et on ne crée pas b_o ni b_1 car a_o et a_1 ont déjà une image, on évite donc la création des deux objets b_o et b_1.

10. Quelques stratégies

Les stratégies suivantes sont assez classiques. Elles sont inspirées de la déduction naturelle. Il faut parfois ne pas effectuer les traitements trop tôt pour éviter d'éventuelles branches infinies ou de trop nombreux découpages.

10.1. Traitement des conclusions universelles et des implications

Leurs traitements sont simples et systématiquement effectués par les règle $\forall$ et $\Rightarrow$ vues en section 2 et qui sont de type *universel* donc prioritaire.

Règle $\forall$: *si on a la conclusion* $\forall X\, P(X)$
alors créer un nouvel objet x et la nouvelle conclusion est $P(x)$

Règle $\Rightarrow$: *si on a la conclusion* $H \Rightarrow C$
alors ajouter l'hypothèse H et la nouvelle conclusion est C

Leur expression en machine a été donnée en section 4.3

10.2. Traitement des conclusions conjonctives

La règle

Règle concl_$\wedge$: *si on a une conclusion conjonctive*
alors démontrer successivement tous les éléments de la conjonction

est exprimée par

```
regle(N,concl_et) :- concl(N,A & B,E), demconj(N,A & B,E,Efin).
```

avec

```
demconj(N,C,Econj, Efin) :-
 (C = (A & B) -> true ; (C=A,B=true) /* pour le dernier */),
  atom_concat(N,-,N0), gensym(N0,N1), % N1=N-1puis2puis...
  creersousth(N,N1,A,Ecreationsousth), % creation sousth (nouvelle etape)
  appliregactiv(N1), % demonstration du sous-theoreme
  concl(N1,true,Edemsousth),
  nouvconcl(N,B,Eretourth), % on retire A qui vient d'etre demontre
   (B = true -> Eretourth=Efin ; demconj(N,B,Eretourth,Efin)
 ).
```

`demconj` démontre récursivement toutes les conclusions de la conclusion conjonctive `A & B` en créant autant de sous-théorèmes qu'il y a de conclusions à démontrer. Les numéros des sous-théorèmes sont construits à partir du numéro `N` en ajoutant un tiret puis des numéros successifs (`0-1-1`, `0-1-2`, `0-1-3`, … sont les sous-théorèmes du (sous-)théorème de numéro `0-1`).

Le nouveau sous-théorème de numéro `N1` hérite des propriétés du théorème de numéro `N` (super-action `creersousth`) sauf pour la conclusion qui n'en est que le morceau `A` que l'on doit démontrer. On indique également que `N` a pour sous-théorèmes `N1`.

Quand un sous-théorème `N1` a été démontré (sa conclusion est égale à `true`), l'information est remontée au (sous-)théorème `N` et `A` est retirée de la conclusion initiale de `N`.

Quand c'est terminé, le (sous-)théorème `N` est démontré (sa conclusion est égale à `true`) si tous ses sous-théorèmes ont été démontrés.

10.3. Traitement des hypothèses universelles

Il n'y a pas d'hypothèses universelles puisque la super-action *ajhyp*, au lieu de les ajouter, les considère comme des lemmes et crée de nouvelles règles locales au (sous-)théorème en cours de démonstration.

10.4. Traitement des conclusions existentielles

Le traitement général (et assez sophistiqué) de MUSCADET1 n'a pas été réécrit dans MUSCADET2 ni dans les versions suivantes. Pour le moment, MUSCADET vérifie seulement qu'il existe des objets satisfaisant la propriété cherchée.

10.5. Traitement des hypothèses existentielles

Une hypothèse existentielle peut conduire à la création, s'il n'en existe pas encore, d'un objet satisfaisant la propriété indiquée. Mais cette création ne doit pas être faite systématiquement à

chaque fois que l'on ajoute une hypothèse existentielle car il pourrait conduire à une création infinie d'objets dans une seule direction en négligeant les autres. Des exemples de telles situations sont analysés dans [Pastre 84, 89, 93].
Pour cette raison, ces hypothèses existentielles sont dans un premier temps ajoutées sans traitement. Plus tard, une règle de type non prioritaire lance le traitement de la première hypothèse existentielle non encore traitée, puis on applique de nouveau les règles plus prioritaires avant de traiter l'hypothèse existentielle suivante. Cela permet de créer les nouveaux objets, un par un et successivement dans toutes les directions.

10.6. Traitement des conclusions disjonctives

On ne fait qu'appliquer des propriétés logiques simples.
Une conclusion disjonctive est vraie si un des éléments de la disjonction est vraie.
Si un des éléments d'une conclusion disjonctive est une négation $\neg A$, A est ajouté comme nouvelle hypothèse et $\neg A$ supprimé de la conclusion.
Certains traitements comme le remplacement de la conclusion par sa définition sont aussi effectués sur une des formules d'une conclusion disjonctive.
Si la disjonction a des sous-formules conjonctives, on fait remonter les conjonctions pour obtenir une conclusion conjonctive qui provoquera un découpage.

10.7. Traitement des hypothèses disjonctives

Des règles simples traitent des cas triviaux. Par exemple, une hypothèse $A \vee A$ est remplacée par A. Autre exemple, si un des éléments de la disjonction est déjà une hypothèse ou est de la forme $X=X$, cette hypothèse est supprimée [18].
Ensuite, un découpage peut être fait, mais, comme pour les hypothèses existentielles, les hypothèses disjonctives sont dans un premier temps ajoutées sans traitement. Plus tard, une règle de type non prioritaire considère la première hypothèse disjonctive $A \vee B$ non encore traitée et prépare le découpage en remplaçant la conclusion C par $(A \Rightarrow C) \wedge (B \Rightarrow C)$. Le découpage sera alors fait par la règle *concl_*$\wedge$.
Il est important de ne pas faire le découpage trop tôt pour ne pas multiplier inutilement les découpages dans le cas de nombreuses hypothèses disjonctives inutiles.

10.8. Connaissances spécifiques à certains domaines

Les connaissances spécifiques aux Espaces vectoriels topologiques [Pastre 84, 89] qui étaient données dans MUSCADET1 sous forme de règles opératoires n'ont pas été incorporées dans les versions suivantes. Plutôt que de les traduire directement par des clauses PROLOG, on projette de les écrire sous forme d'énoncés mathématiques, et d'écrire des métarègles capables de générer ces méthodes opératoires.
Par contre, le travail sur la géométrie discrète [Pastre 93] a été poursuivi en MUSCADET2 qui a permis d'obtenir des résultats plus satisfaisants en « aidant » moins le système.

11. Enoncés du 2ème ordre

On a vu en section 2.1. que MUSCADET est capable de travailler dans le calcul des prédicats du deuxième ordre. Ainsi, pour l'exemple de la section 2.1 (fichier d'exemple `exemple-ordre2`) on peut définir ainsi la propriété de transitivité d'une relation

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

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

et proposer le théorème

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

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

[18] Elle n'est pas matériellement supprimée, on mémorise seulement le fait qu'elle a été traitée.

soit, en machine

```
transitive(inc)
```

ou

```
transitive(inc,parties(E))
```

Pour la définition de `transitive`, à partir de la version MUSCADET2, il y eut une petite difficulté car la syntaxe PROLOG interdit d'écrire `R(X,Y)` si `R` est une variable. J'ai donc introduit le symbole fonctionnel «`..`» qui permet à `..[R,X,Y]` de désigner ce `R(X,Y)` interdit. La définition s'écrit alors

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

PROLOG n'unifie pas `r(a,b)` et `..[R,X,Y]`, j'ai donc écrit des prédicats permettant de le faire et de renvoyer `R=r, X=a et Y=b`.

Le choix de ce symbole «`..`» a été fait par analogie avec l'opérateur PROLOG « `=..` » . En effet, on a `r(a,b) =.. [r,a,b]` en PROLOG si `r` est une constante, et `r(a,b)` et `..[r,a,b]` s'unifient dans MUSCADET, sinon dans PROLOG. D'autre part, MUSCADET remplace `..[r,a,b]` par `r(a,b)` dès que `r` est une constante de façon à produire des traces plus agréables à lire.

Remarque : Il est possible à l'utilisateur d'écrire vraiment les formules comme en mathématiques et de traduire, par exemple par un script Unix, `R(X,Y)` en `..[R,X,Y]` avant de rentrer dans PROLOG.

Cette notation est également utilisée pour les fonctions et les applications mathématiques. On peut écrire `f(x)::y` qui s'unifie dans MUSCADET avec `..[F,X]::Y` pour les instantiations `F=f, X=x` et `Y=y`.

La super-action *ajhyp* traite ces expressions et ajoute les hypothèses sous la forme la plus agréable `r(a,b)` ou `f(x)::y` au lieu de `..[r,a,b]` ou `..[f,x]::y` par les règles

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

Attention à la place des blancs : « `=..  [...]` » et « `=  ..[...]` » ne désignent pas la même chose !

Cette notation n'est pas obligatoire, on peut choisir un symbole, `apply` par exemple, utilisé dans divers problèmes de la bibliothèque TPTP, ou tout autre symbole, et écrire partout `apply(R,X,Y)` (resp. `apply(F,X,Y)`) au lieu de `..[R,X,Y]` (resp. `..[F,X]::Y`). Mais, ce symbole `apply` ne pouvant être connu du démonstrateur on doit alors aussi donner l'équivalence pour chaque relation constante, par exemple

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

soit en machine

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

C'est ce qui a été fait dans quelques problèmes sur les relations (équivalence, (pré-)ordre, ordre total, bon ordre) qui ont été proposés pour être inclus dans la bibliothèque TPTP, par exemple le problème SET806+4.p affirmant que l'égalité d'ensembles définit une relation de pré-ordre. De plus, à la demande de Geoff Sutcliffe, des noms différents ont été donnés pour désigner respectivement les constantes et les relations d'arité supérieure à 0 (dans l'exemple précédent `subset_predicate` et `subset`) car d'autres démonstrateurs pourraient ne pas accepter le même symbole. Pour MUSCADET, comme pour l'être humain (il s'agit ici de théorie naïve des ensembles), utiliser le même symbole ne pose pas de problèmes. L'exemple du fichier `variante_set807+4.p` en est la version correspondante du problème SET806+4.p.

12. Disponibilité

MUSCADET4 est disponible à l'adresse

http://www.normalesup.org/~pastre/muscadet

`muscadet.html` est un mode d'emploi résumé
`muscadet-fr` est le fichier PROLOG complet[19]

`musca-fr`[20] est un script Unix qui permet de travailler sous PROLOG, il faut ensuite appeler `demontrer`[21], `th` ou `tptp`. Voir section 5, ou taper `th.` ou `tptp.` (avec un point, sans arguments) pour afficher les modes d'emploi.

`th-fr.c` et `tptp-fr.c`[22] sont des petits fichiers `C` qui permettent de travailler sous Linux. Les compiler. Soient par exemple `th` et `tptp` les exécutables obtenus[23]. `th` permet de travailler sur des données préalablement enregistrées dans des fichiers (voir section 5.2). `tptp` permet de travailler sur la bibliothèque TPTP (voir section 5.3). `th` et `tptp` sans arguments donnent les modes d'emploi.

`exemples`[24] est un répertoire d'exemples de données et d'exécutions.

13. Références

[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 76] - Pastre D., Démonstration Automatique de Théorèmes en Théorie des Ensembles, Thèse de 3ème cycle, Paris 6 (1976)

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

[Pastre 84] - Pastre D., MUSCADET : Un Système de Démonstration Automatique de Théorèmes utilisant Connaissances et Métaconnaissances en Mathématiques, Thèse d'état, Paris 6 (1984)

[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

[19] en anglais `muscadet-en`

[20] en anglais `musca-en`

[21] en anglais `prove`

[22] en anglais `th-en.c` et `tptp-en.c`

[23] commandes `cc th-fr.c -o th` et `cc tptp-fr.c -o tptp`

[24] en anglais `examples`

[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[25] : 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://ww.lip6.fr/reports/lip6.2000.002.html, 54-98, et http://www.normalesup.org/~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.normalesup.org/~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.normalesup.org/~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.normalesup.org/~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.normalesup.org/~pastre/mips.pdf, http://www.normalesup.org/~pastre/slides-mips.pdf

[Pastre 14] - Mathematical theorem proving, from Muscadet0 to Muscadet4, why and how, Workshop about Sets and Tools, Affiliated to ABZ 2014 Conference, Toulouse, France, 2014, http://sets2014.cnam.fr/papers/00010003.pdf, http://www.normalesup.org/~pastre/sets.pdf, http://www.normalesup.org/~pastre/slides-sets-par4.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

[25] Nom donné à l'époque à la première version PROLOG de MUSCADET