

Programmation Logique – TP noté

Une extension de Prolog

Narendra Jussien

2002

1 Introduction

Un des *légers inconvénients* de Prolog est le traitement particulier de l'arithmétique qui ne fonctionne qu'à *sens unique* et avec de fortes contraintes quant au degré d'instanciation des termes considérés.

Cette réflexion permanente à mener dès lors qu'on manipule des nombres conduit à du code redondant, peu clair et souvent source d'erreurs.

On aimerait donc, au contraire, pouvoir écrire du code simple tel que le suivant :

```
% factorielle
fact(0.0,1.0).      % on choisit ici de travailler sur les r{\e}els ...
fact(N,F) :-
    N1 eq N-1,
    F eq N*F1,
    fact(N1,F1).
% conversion Farenheit - Celsius
fc(C,F) :-
    C eq 5/9*(F-32).
```

eq est ici un nouvel opérateur dont la sémantique serait de maintenir une contrainte d'égalité entre les deux termes en permanence dans le calcul. fact/2 et f2C auraient alors les propriétés d'un prédicat Prolog classique : réversibilité, multiusage ...

```
Prolog {\e}tendu ?- fact(4,24).
yes
Prolog {\e}tendu ?- fact(5,X).
X=120.0 ?
yes
Prolog {\e}tendu ?- fact(X,720).
X=6.0 ?
yes
Prolog {\e}tendu ?- fc(15,F).
F=59.0 ?
yes
Prolog {\e}tendu ?- fc(X,88).
X=31.111111111111114 ?
yes
```

La déclaration de l'opérateur `eq` en Prolog se fait ainsi :

```
:- op(700,xfx,eq).
```

Le but de ce TP est d'étendre le langage Prolog pour qu'il ait ce comportement. Nous serons sûrement amenés à utiliser dans ce TP les prédicats prédéfinis¹ suivants : `=..`, `format`, `var`, `compound`, `ground`, ...

2 Quelques outils

Pour mettre en œuvre ce comportement, l'idée principale est de *mettre en attente* les égalités introduites dans le programme jusqu'à ce qu'elles puissent être calculées/vérfiées.

Nous appellerons une égalité déterminée, une égalité pour laquelle :

- soit tout est instancié et il n'y a plus qu'à vérifier l'égalité,
- soit tout est instancié sauf une variable et ainsi il n'y a plus qu'à calculer la valeur de cette variable.

Les égalités que nous traiterons obéiront à la grammaire suivante :

```
<egalite> --> <var> eq <terme>
<terme>   --> <var> | <var> <op> <var>
<op>     --> + | - | * | /
<var>    --> variable Prolog | ground-term Prolog
```

- Écrire le prédicat `egaliteDeterminee/1` qui réussit si l'égalité passée en paramètre est déterminée.

```
| ?- egaliteDeterminee(X eq 1 + 2).
```

```
yes
```

```
| ?- egaliteDeterminee(X eq 1 + Y).
```

```
no
```

```
| ?- egaliteDeterminee(X eq 3).
```

```
yes
```

```
| ?- egaliteDeterminee(4 eq 2 * Y).
```

```
yes
```

- Écrire le prédicat `evalEgalite/1` qui prend en paramètre une égalité déterminée et selon le cas, vérifie cette égalité ou instancie l'unique variable libre avec la valeur permettant de vérifier cette égalité.

```
| ?- evalEgalite(4 eq 2 * 2).
```

```
yes
```

```
| ?- evalEgalite(X eq 3 * 4).
```

```
X = 12 ?
```

```
yes
```

```
| ?- evalEgalite(4 eq 2 * Y).
```

```
Y = 2.0 ?
```

```
yes
```

```
| ?- evalEgalite(4 eq 0 * Y).
```

```
no
```

On veillera à traiter les cas d'erreurs possibles correctement (division par zéro par exemple).

- Écrire le prédicat `transforme/2` qui transforme une égalité quelconque passée en premier paramètre (tout de même de la forme `X eq terme`) en

¹Rappel : une doc en ligne est disponible à <http://www.cling.gu.se/datorinfo/sicstus/sicstus3/sicstus3.html>

une liste d'égalités obéissant à notre grammaire et dont la conjonction permet de retrouver l'égalité originale.

```
| ?- transforme(C eq 1/2 * (F -12), L).
L = [C eq _B*_A, _B eq 0.5, _A eq _D-_C, _D eq F, _C eq 12.0] ?
yes
```

On prendra bien garde à convertir² tous les nombres en réels (nous ne traiterons que les réels).

- Écrire le prédicat `reduitEgalites(Entree,Sortie)` qui évalue et supprime tant que c'est possible toutes les égalités déterminées de `Entree` et renvoie les égalités restantes dans `Sortie`.

On prendra bien garde à bien s'assurer que toutes les réductions possibles ont été faites dans la liste résultat `Sortie`.

```
| ?- reduitEgalites([Y eq X + 1, X eq 2],L).
L = [], X = 2, Y = 3
```

```
| ?- reduitEgalites([Y eq X + 1, Z eq Y + T, X eq 2], L).
L = [Z eq 3+T], X = 2, Y = 3
```

3 Un interprète pour notre Prolog étendu

Nous devons pouvoir conserver les égalités en attente tout au long de la démonstration. Il faut donc utiliser un interprète qui à tout moment connaît précisément la séquence de buts qu'il lui reste à prouver pouvant ainsi leur passer la liste courante des égalités en attente. Nous utiliserons donc un interprète avec continuation.

```
prouve_cont(true, []) :- !.
prouve_cont(true, [X | Xs]) :-
    !,
    prouve_cont(X,Xs).
prouve_cont( (A,B) , Cont) :-
    !,
    prouve_cont(A,[B | Cont]).
prouve_cont( But, Cont) :-
    predefini(But), !,
    But,
    prouve_cont(true,Cont).
prouve_cont( But, Cont) :-
    clause(But,Corps),
    prouve_cont(Corps,Cont).
```

Étendre cet interprète de telle sorte que :

- dès qu'il rencontre une égalité (utilisation de l'opérateur `eq`), cette égalité est transformée en une liste d'égalités en attente qui après traitement sera passée avec la continuation,
- dès qu'une nouvelle instanciation est réalisée (par unification avec une tête de clause par exemple), la liste courante des égalités en attente est traitée et le résultat est passé à la continuation,
- s'il arrive à prouver un but de donner avec une liste d'égalités en attente vide, il y a succès.

²Remarque : si X est un nombre, alors X + 0.0 est un réel de même valeur que X.

- si la liste d'égalités n'est pas vide à la fin de la démonstration, il y a quand même succès mais l'interprète doit prévenir l'utilisateur et afficher la liste des égalités en attente.

```
| ?- prouve_cont(fc(X,Y), [], []).
-> Il reste des contraintes en attente ... [_55 eq 0.56*_405,_405 eq _707-32.0,_707 eq _69]
true ?
yes
```

4 Un véritable Prolog étendu

Pour être complètement transparent vis à vis de l'utilisateur, on pourra utiliser le fichier `top_level.pl` fourni qui permet, après lancement du prédicat `main/0`, de fournir un lecteur et un résolveur de buts dans notre Prolog étendu.

```
| ?- main.
Prolog {\`e}tendu ?- fc(X,4).
X= -15.555555555555557 ?
yes
```

Pour compléter notre Prolog, nous allons introduire une autre *contrainte* à l'aide de l'opérateur `/=` qui exprimera la différence entre deux termes. La grammaire d'appel de cet opérateur sera la même (à l'opérateur prêt) que pour `eq`. Par contre, une telle contrainte ne sera déterminée que lorsque tout sera instancié.

Compléter en conséquence le programme écrit jusqu'à présent pour prendre en compte cette nouvelle contrainte.

5 Programmation par contraintes

Considérons le prédicat suivant :

```
:- dynamic tousDifférents/1.
tousDifférents([]).
tousDifférents([X|Xs]) :-
    différent(Xs,X),
    tousDifférents(Xs).

:- dynamic différent/2.
différent([],X).
différent([Y|Ys],X) :-
    X /= Y,
    différent(Ys,X).
```

Ce prédicat demande que tous les éléments de la liste passée en paramètre soient distincts les uns des autres. En Prolog classique un tel prédicat n'a d'intérêt que si la liste passée en paramètre est complètement instanciée. Dans notre extension, on peut appeler ce paramètre pour mettre en place ces contraintes dès le début de la résolution, permettant ainsi de les utiliser pour se sortir plus rapidement des branches conduisant à un échec.

```

Prolog {\e}tendu ?- tousDifferentes([X,Y,Z]), member(X,[1,2,3]),
                    member(Y,[1,2,3]), member(Z,[1,2,3]).
X=1,
Y=2,
Z=3 ?

```

Dans l'exemple précédent, la résolution en Prolog étendu provoque un échec dès que X et Y sont instanciées avec la valeur 1 changeant ainsi la valeur de Y en 2. Une version en Prolog classique aurait requis l'exploration successive des solutions 1,1,1, 1,1,2, 1,1,3 (toutes les possibilités pour Z) avant de revenir sur la valeur de Y.

Notre Prolog étendu utilise les contraintes *a priori* par opposition à Prolog standard qui ne peut les utiliser qu'*a posteriori*³.

D'autre part, comme nous l'avons déjà vu, si l'information passée au solveur n'est pas suffisante pour instancier toutes les variables, le système renvoie un ensemble de contraintes qui permet de caractériser les solutions réalisables sans provoquer un échec faute d'information ou en provoquant un succès sans condition (mauvais usage du prédicat \==).

Ce que nous avons fait pour l'égalité et la différence peut être fait pour d'autres types de relations (numériques, symboliques,...). L'extension de Prolog obtenue alors est appelée la Programmation Logique avec Contraintes (ou Programmation par Contraintes). C'est un outil particulièrement efficace pour résoudre les problèmes fortement combinatoires car il permet d'allier la déclarativité de Prolog à des techniques efficaces de résolution de contraintes.

³On appelle cette façon de résoudre un problème du *generate and test* (générer une solution instanciée avant de pouvoir la tester). Il s'agit d'une façon de procéder notoirement inefficace.