13 UTILISATION DES ASSERTIONS DANS LES PROGRAMMES PROLOG IV

(version 1.0beta - September 1998)

DANS CE CHAPITRE est décrit comment les assertions peuvent être utilisés pour la mise au point de programmes Prolog IV.

13.1 INTRODUCTION

Les assertions sont exprimées avec une nouvelle directive prise en compte par un pré-processeur pendant la compilation. Elles décrivent des propriétés de prédicats.

La fonction première des assertions est de déboguer un programme pendant la phase de compilation. Ici, le programmeur peut introduire des assertions pour décrire les propriétés qui doivent être vraies lorsque la règle est appelée, et d'autres propriétés qui doivent être vraies si la règle est appliquée avec succès. Dans la phase de pré-compilation, les propriétés doivent être prouvées, indépendamment des valeurs prises par les arguments d'une règle. Si elles ne sont pas prouvées, des tests effectuant les vérifications pendant l'exécution de la règle peuvent être ajoutés automatiquement dans celle-ci.

13.2 SYNTAXE ET SIGNIFICATION DES ASSERTIONS

Une assertion est introduite au moyen d'une directive. Pour un prédicat donné, une seule assertion est permise : elle se décompose en un ensemble de pré-conditions et un ensemble de post-conditions. La syntaxe en est la suivante :

$$:- \ pred(Arg_1,\ Arg_2,\ \ldots,\ Arg_n) : (pre_1,\ pre_2,\ \ldots,\ pre_p) \Rightarrow (post_1,\ post_2,\ \ldots,\ post_q).$$

Dans laquelle :

- *pred* est le nom du prédicat;
- Arg_1, Arg_2, ..., Arg_n sont des variables distinctes représentant les arguments du prédicat;
- pre_1, pre_2, ..., pre_p est l'ensemble des pré-conditions;
- $post_1$, $post_2$, ..., $post_q$ est l'ensemble des post-conditions.

Les Assertions dans Prolog IV sont partiellement financée par le projet ESPRIT « DiSCiPl » numéro 22532.

L'interprétation d'une telle assertion est la suivante :

- l'ensemble des pré-conditions définit la conjonction des propriétés nécessaires que doit vérifier un modèle admissible pour la définition donnée du prédicat *pred*;
- l'ensemble des post-conditions définit la conjonction des propriétés de l'ensemble des solutions satisfaisant le prédicat *pred* dans ce modèle.

Si aucune pré-condition n'est nécessaire, l'assertion peut être écrite :

:- $pred(Arg_1, Arg_2, ..., Arg_n)$ => $(post_1, post_2, ..., post_q)$.

Si aucune post-condition n'est à préciser, on peut écrire l'assertion :

:- $pred(Arg_1, Arg_2, ..., Arg_n)$: $(pre_1, pre_2, ..., pre_p)$.

Le plus sûr moyen d'utiliser les assertions est de les déclarer au début du programme (comme les déclarations dans un programme C) pour être certain qu'elles soient connues du compilateur au moment de la recherche de preuve, lorsqu'il examine chacun des paquets.

13.3 CONDITIONS PERMISES DANS LES ASSERTIONS

Nous allons maintenant définir comment sont exprimées les conditions au sein des assertions.

Il est possible de mixer deux sortes de conditions : les contraintes (relations prédéfinies) de Prolog IV, plus quelques “méta” conditions d'ordre plus général.

En ce qui concerne la première catégorie, toute contrainte Prolog IV peut être utilisée, qu'elle soit de type intervalle, linéaire, arbre ou liste.

La seconde catégorie de conditions comporte les “méta-contraintes”. Ces conditions sont :

- `bound(X)` qui est vraie si `X` n'est pas une variable libre;
- `ground(X)` qui est vraie si `X` est un terme complètement connu (sans variables libres);
- `listnum(L)` qui est vraie si `L` est une liste de nombres (pas nécessairement connus);
- `listnum(L, A, B)` qui est vraie si `L` est une liste de nombres appartenant à l'intervalle `[A,B]` (pas nécessairement connus).

13.4 LIMITATIONS

Pour les conditions décrites par des contraintes de Prolog IV, la réalisation de la preuve est complètement dépendante des solveurs de celui-ci, et en particulier de leur complétude (ou leur incomplétude).

Les deux solveurs numériques sont utilisés ; le solveur linéaire, qui est complet, et le solveur par intervalles, qui est lui incomplet. Selon le solveur utilisé, une preuve peut être ou non réalisée. Exemple :

```
:- a(X,Y) : (X gelin Y) => (X gtlin Y).
a(X,Y) :- dif(X,Y).
```

Dans cet exemple, l'assertion de la règle `a/2` peut être prouvée. En fait, les contraintes mise en jeu dans l'ensemble de pré-conditions est tout d'abord ajouté à l'ensemble courant de contraintes. Ainsi, la contrainte `X gelin Y` est ajoutée. Lorsque le corps de la règle est traité, la contrainte `dif(X,Y)` est ajoutée à son tour. Ensuite, pour vérifier l'ensemble des post-conditions, on essaie de vérifier que chacune des post-conditions est déduite de l'ensemble courant de contraintes. Pour ceci, la contrainte de signification opposée à celle de la condition (ici la contrainte `X lelin Y`) est ajoutée à l'ensemble courant de contraintes, et un échec est attendu. La complétude du solveur linéaire permet ici cet échec. L'implication de la post-condition par l'ensemble courant de contrainte est donc prouvée.

Maintenant, considérons cet exemple :

```
:- b(X,Y) : (X ge Y) => (X gt Y).
b(X,Y) :- dif(X,Y).
```

Pour la règle `b/2`, lorsque la contrainte opposée à la post-condition `X gt Y` (`X le Y`) est ajoutée à l'ensemble courant de contraintes (qui contient déjà la contrainte `dif(X,Y)`, précédemment ajoutée lors du traitement du corps de règle), l'échec attendu ne survient pas (à cause de l'incomplétude du solveur par intervalle). Dans ce cas, l'assertion n'est pas prouvée.

Un comportement similaire peut apparaître lorsque les deux solveurs numériques sont utilisés dans une assertion. Par exemple :

```
:- d(X,Y) : (X gtlin 3) => (X gt 5).
d(X,Y) :- Y = X + 2.
```

Pour réduire les effets de l'incomplétude du solveur par intervalle, il est recommandé (un avertissement est imprimé sinon) :

- de n'utiliser une contrainte non linéaire qu'avec au plus une variable libre parmi ses arguments (les autres étant des constantes);
- de ne pas mixer les contraintes gérées par différents solveurs.

13.5 EXEMPLE COMPLET

Considérons le célèbre programme `SEND + MORE = MONEY`.

Ce programme peut être complètement prouvé si nous prenons en compte les considérations citées plus haut. On obtient le source suivant :

```
:-  solution(Send,More,Money)  =>
         ( ground(Send) ,   cc(Send, 1000,9999) ,
         ground(More) ,  cc(More, 1000,9999) ,
         ground(Money) , cc(Money, 10000,99999) ).
:-   integers(L)  => (ground(L) , listnum(L)).
:-   allDifferentDigits(L) : listnum(L) => listnum(L,0,9).
:-   outsideOf(X,L) : (listnum(L) , real(X)) .
:-   mydif(X,Y) : (real(X), real(Y)) .

integers([]) .
integers([E|R]) :- enum(E), integers(R).

allDifferentDigits([]) .
allDifferentDigits([X|S]) :-
    cc(X,0,9),
    outsideOf(X, S),
    allDifferentDigits(S).

outsideOf(X, []) .
outsideOf(X, [Y|S]) :- mydif(X,Y), outsideOf(X, S).

solution(Send, More, Money) :-
    S ge 1 ,
    M ge 1 ,
    Send .+. More = Money ,
    Send  =            1000.*.S.+.100.*.E.+.10.*.N.+.D ,
    More  =            1000.*.M.+.100.*.O.+.10.*.R.+.E ,
    Money = 10000.*.M.+.1000.*.O.+.100.*.N.+.10.*.E.+.Y ,
    allDifferentDigits([M,S,O,E,N,R,D,Y]),
    integers([M,S,O,E,N,R,D,Y]) .
```

13.6 VÉRIFICATIONS PENDANT L'EXÉCUTION

Lors de la vérification d'un programme, si une pré-condition ne pouvait être prouvée pour un but donné du corps d'une règle, le but en question serait noté comme étant non vérifié pour ces pré-conditions. Le mécanisme de preuve se poursuit néanmoins comme si la preuve de la pré-condition avait été faite. Pendant la compilation, si le mode courant autorise la vérification dynamique, le code nécessaire pour vérifier la condition non-testée est inséré avant le but.

Le principe est le même pour les post-conditions. Par exemple, le programme suivant comporte une assertion erronée :

```
:- sumlist(L,S) : list(L)  => (ground(S) , real(S)).
sumlist([],0).
sumlist([X | L],S1) :-
    S1 = S + X ,
    sumlist(L,S).
```

Dans cet exemple qui calcule la somme des éléments d'une liste, aucune propriété “ground” n'est posée dans les pré-conditions; aussi, pour la seconde règle, cette propriété ne peut être prouvée dans les post-conditions. Les règles insérées en réalité sont les suivantes :

```
sumlist([],0).
sumlist([X | L],S1) :-
      S1 = S + X ,
      sumlist(L,S),
      dyn__check_post(sumlist([X | L],S1)).
```

Voici quelques exemple d'exécutions :

```
>> sumlist([1,2,3],L).

L = 6.

>> sumlist([Q,2,3],L).

WARNING: Post-condition verification failed for sumlist/2 Rule
Nb: 2
      Assertion No 1: ground(_967)
L ~ real,
Q ~ real.
```

Quant à l'assertion suivante :

```
:-  sumlist(L,S) : ground(S) => ground(L).
```

La propriété “ground” dans la pré-condition ne peut être prouvée pour le but `sumlist/2` appelé dans le corps de la seconde règle; aussi, les règles ajoutées en réalité sont complétées par des tests :

```
sumlist([],0).
sumlist([X | L],S1) :-
      S1 = S + X ,
      dyn__check_pre(sumlist(L,S), sumlist([X | L],S1)),
      sumlist(L,S).
```

Exemple d'exécutions :

```
sumlist([1,2,3],K).

WARNING: pre-condition verification failed for call of sumlist/2
    in rule sumlist/2 Rule Nb: 2 (Assertion No 1: ground(_950))
WARNING: pre-condition verification failed for call of sumlist/2
    in rule sumlist/2 Rule Nb: 2 (Assertion No 1: ground(_1002))
WARNING: pre-condition verification failed for call of sumlist/2
    in rule sumlist/2 Rule Nb: 2 (Assertion No 1: ground(_1054))
K = 6.

>> sumlist([1,2,3],6).

true.
```

13.7 EXEMPLE DE COMPILATIONS

Ce programme est correct :

```
:- fact(N,P) : (bound(N), real(N)) => (bound(P),real(P)).

fact(0,1) :- !.
fact(N,N*P) :-
   fact(N-1,P).
```

Celui-là ne l'est pas, si l'on en croit l'assertion donnée :

```
:-  badfact(N,P) : (bound(N), real(N)) => (bound(P),real(P)).

badfact(0,W) :- !.   /* <-- The error is here  */
badfact(N,N*P) :-
   !,
   badfact(N-1,P).

WARNING : Post-condition verification failed for badfact/2
    (Assertion No 1: bound(_3781)
    Rule checked: badfact/2 Rule nb: 1)
WARNING : Post-condition verification failed for badfact/2
    (Assertion No 2: real(_3998)
    Rule checked: badfact/2 Rule nb: 1)
```

Ces compilations réussissent sans donner d'avertissement :

```
:- sumlist(L,S) : ( listnum(L) , ground(L) ) =>
                ( ground(S) , real(S) ).

sumlist([],0).
sumlist([X | L],S1) :-
   S1 = S + X ,
```

```
   sumlist(L,S).

:-  sl(L,S) : ( listnum(L,0,100000) , ground(L) ) =>
       ( ground(S) , S ge 0 ).

sl([],0).
sl([X|L],X .+. S) :-
    sl(L,S).


:-  slOk(L,S) : (ground(L) ) => (ground(S) ).

slOk([],0).
slOk([X | L],S1) :-
    S1 = S + X ,
    slOk(L,S).
```

Cette compilation génère des avertissements pour la propriété "ground" (dans la post-condition) :

```
:-  slwrong(L,S) : (list(L) ) => (ground(S) , real(S) ).

slwrong([],0).
slwrong([X | L],S1) :-
    S1 = S + X ,
    slwrong(L,S).

WARNING: Post-condition verification failed for slwrong/2
    (Assertion No 1: ground(_7618)
    Rule checked: slwrong/2 Rule nb: 2)
```

Dans cette compilation, la preuve échoue car il manque une assertion dans le programme :

```
:-  slwrong(L,S) : ( listnum(L) , ground(L) ) =>
                ( ground(S) , real(S) ).

slwrong([],0).
slwrong([X | L],S1) :-
    S1 = S + X ,
    slmiss(L,S).

WARNING: assertion is missing  for slmiss/2
WARNING: Post-condition verification failed for slwrong/2
    (Assertion No 1: ground(__7163)
    Rule checked: slwrong/2 Rule nb: 2)
```

Cette compilation génère des avertissements pour la propriété "ground" (en pré-condition) :

```
:-  slBad(L,S) : (ground(S) ) => (ground(L) ).

slBad([],0).
slBad([X | L],S1) :-
    S1 = S + X ,
    slBad(L,S).

WARNING: Pre-condition verification failed for slBad/2
    (Assertion No 1: ground(_6952)
    Rule checked: slBad/2 Rule nb: 2)
```

Ce programme contient une assertion incohérente :

```
:- slerror(L,S) : (list(L), real(L)) => (ground(S), real(S)).

slerror([],0).
slerror([X | L],S1) :-
    S1 = S + X ,
    slerror(L,S).

slerror/2 : real(_3237)  Rule checked: slerror/2 Rule nb: 1
      error : ['A clause or assertion is inconsistent']
```

13.8 UTILISATION

Pour utiliser les assertions, la première chose à faire est de charger le module de preuve (fichier `proof.p4o`). les paramètres à fournir sur la ligne de commande prolog (ou dans le fichiers de préférences `.p4prefs` (sous UNIX) ou `p4prefs.tcl` (sous Windows)) sont :

```
-l   proof.p4o
```

Le mode de fonctionnement des assertions (vérification dynamique, etc) peut être choisi dans le menu **Prolog** des éditeurs de texte (item **Assertions**).