Assertions

Introduction

This chapter describes how to use assertions in a Prolog IV program. Assertions are expressed with a new directive which is handled by a compiler preprocessor. They describe properties of predicates. The main function of assertions is to debug a program while compiling it. Here, the user can introduce assertions to describe properties that must be true when the rule is called and other properties that must be true on a success of the rule. In a pre-compilation phase, properties are to be proved, independently of the values taken by the arguments of the rule. If they are not proved, run-time checks can be added into the rule.

Syntax and meaning of assertions

An assertion is introduced by the way of a directive. For a given predicate, a unique assertion is allowed: it is composed of a set of preconditions and a set of postconditions. It is described in the following syntax:

:- $pred(Arg_1, Arg_2, ..., Arg_n) \quad : \quad (pre_1, pre_2, ..., pre_p) \quad => (post_1, post_2, ..., post_q)$.

where:

- $pred$ is the name of the predicate;
- $Arg_1, Arg_2, ..., Arg_n$ must be distinct variables representing the arguments of the predicate;
- $pre_1, pre_2, ..., pre_p$ represent the set of preconditions;
- $post_1, post_2, ..., post_q$ represent the set of postconditions.

The interpretation of such an assertion is:

- the set of preconditions defines the conjunction of the needed properties of an admissible model for the given definition of the predicate $pred$;
- the set of postconditions defines the conjunction of the properties of the set of solutions satisfying the predicate $pred$ in that model.

If no precondition is needed, the assertion must be written:

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

If no postcondition is needed, the assertion must be written:

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

The best way to use assertions is to declare them at the beginning of the program (like a ".h" file in a C program) in order to know all of them while performing the proof.

Conditions allowed in assertions

We define now the expected form of the conditions in the assertions. It is possible to mix two different kinds of conditions: predefined constraints of Prolog IV solvers, and more general "meta" conditions.

Concerning the first category, all Prolog IV constraints can be used: type constraints, interval constraints, linear constraints...

The second category of conditions contains "meta" constraints or properties. These conditions are:

- $bound(X)$ which is true if X is not a free variable;
- $ground(X)$ which is true if X is a ground term (fully known);
- $listnum(L)$ which is true if L is a list of numbers (not mandatory known);
- $listnum(L, A, B)$ which is true if L is a list of numbers belonging to the interval $[A, B](notmandatoryknown)$.

Limitations

For the conditions which are Prolog IV predefined constraints, performing the proof is completely dependent of the solvers, and particularly to their completeness (or their incompleteness). The two Prolog IV solvers are used: the linear solver, which is complete, and the interval solver, which is incomplete. According to the solver used, a proof can or cannot be performed. Example:

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

In this example, the assertion of the rule $a/2$ can be proved. In fact, the constraints involved by the set of preconditions are first added to the current set of constraints. Thus the constraint $X\ gelin\ Y$ is added. When the body of the rule is handled, the constraint $dif(X,Y)$ is added. Then, to verify the set of postconditions, the goal is to check if each postcondition is deduced from the current set of constraints. To do that, the opposite constraint to the condition (here the constraint $X\ lelin\ Y$) is added to the current set, and a failure is expected. The completeness of the linear solver performs this failure. So the implication of the postcondition by the current set is proved.

Now, let us consider the example:

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

For rule $b/2$, when the opposite constraint to the postcondition $X \ gt \ Y$ ($X \ le \ Y$) is added to the current set (which already contains the constraint $dif(X, Y)$ added by the treatment of the body), the expected failure does not occur (because of the incompleteness of the interval solver). So the assertion is not proved.

A similar behaviour can appear when the two solvers are requested in one assertion. Example:

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

To reduce the effects of the incompleteness of the interval solver, it is recommended (else, a warning is raised):

- to limit non linear constraints to have a unique variable (and other constants) among their arguments;
- not to mix constraints handled by different solvers.

Full example

Let us consider the famous program $SEND+MORE = MONEY$. This program can be entirely proved if we take into account the considerations listed above. That gives the following source code:

```
:- 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]).
```

Run-time checking

When verifying a program, if in the body of a rule some preconditions cannot be proved for a given goal, the goal is flagged as unchecked for these preconditions. Then, the proof continues normally as if the proof of the preconditions had succeeded. At compile-time, if the current mode allows dynamic checks, the code needed to verify the unchecked preconditions is inserted before the goal. The principle is the same for the postconditions. For example, the following program has a wrong assertion:

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

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

In this sample which computes the sum of the elements of a list, no groundness is assumed by the preconditions, so for the second rule the groundness cannot be proved in the postconditions. The rules actually inserted are:

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

Samples of executions:

```
 » sumlist([1, 2, 3], L).

L = 6.

 » sumlist([Q, 2, 3], L).
WARNING: Postcondition verification failed for sumlist/2 Rule Nb: 2
                Assertion No 1: ground(_967)
L ~real,
Q ~real.
```

For the assertion:
 :- $sumlist(L, S) : ground(S) => ground(L)$.

The groundness in the precondition cannot be proved for the goal $sumlist/2$ called in the body of the second rule, so the rules actually inserted will be:

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

Samples of executions:

```
sumlist([1, 2, 3], K).
WARNING: precondition verification failed for call of sumlist/2
           in rule sumlist/2 Rule Nb: 2 (Assertion No 1: ground(_950))
WARNING: precondition verification failed for call of sumlist/2
           in rule sumlist/2 Rule Nb: 2 (Assertion No 1: ground(_1002))
WARNING: precondition 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.
```

Samples of compilations

This program is correct:

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

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

This program is not correct, according to the assertion:

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

These compilations succeed without any warning:

```
:- 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), Sge0).
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).
```

This compilation raises warnings for groundness (Postcondition):

$:-\ slwrong(L, S) : (list(L)) => (ground(S), real(S)).$
$slwrong([], 0).$
$slwrong([X|L], S1)\ :-$
$\quad S1 = S + X,$
$\quad slwrong(L, S).$
WARNING: Postcondition verification failed for $slwrong/2$
(Assertion No 1: $ground(\_7618)$
Rule checked: $slwrong/2$ Rule nb: 2)

In this compilation, the proof fails because it needs an undefined assertion:

$:-\ slwrong(L, S) : (listnum(L), ground(L)) =>$
$\quad (ground(S), real(S)).$
$slwrong([], 0).$
$slwrong([X|L], S1)\ :-$
$\quad S1 = S + X,$
$\quad slmiss(L, S).$
WARNING: assertion is missing for $slmiss/2$
WARNING: Postcondition verification failed for $slwrong/2$
(Assertion No 1: $ground(\_7163)$
Rule checked: $slwrong/2$ Rule nb: 2)

This compilation raises warnings for groundness (Precondition):

$: -\ slBad(L, S) : (ground(S)) => (ground(L)).$
$slBad([], 0).$
$slBad([X|L], S1)\ :-$
$\quad S1 = S + X,$
$\quad slBad(L, S).$
WARNING: Precondition verification failed for $slBad/2$
(Assertion No 1: $ground(\_6952)$
Rule checked: $slBad/2$ Rule nb: 2)

This program has an inconsistent assertion:

$:-\ slerror(L, S) : (list(L), real(L)) => (ground(S), real(S)).$
$slerror([], 0).$
$slerror([X|L], S1)\ :-$
$\quad S1 = S + X,$
$\quad slerror(L, S).$
$slerror/2 : real(\_3237)$ Rule checked: $slerror/2$ Rule nb: 1
error : ['A clause or assertion is inconsistent']

Usage

To use the assertions, the first action to perform is to load the proof module (file $proof.p4o$) and to execute the predicate which initializes this module ($assertions\_\_init/0$). The corresponding parameters of the command line (to put in the file named $.p4prefs$) are:

$-l\ proof.p4o\ \ -x\ assertions\_\_init$

The assertion mode can be chosen in the *Prolog* menu of the text editor (item *Assertions*).