Cours de programmation fonctionnelle et logique
Cours 10 : Pro(grammation) log(ique)

Table des matières

1 Langage utilisé

SWI-Prolog version 3.2.9. Logiciel (compilateur/interpréteur) : `/usr/bin/pl` (sur les machines Linux). Documentation : `http://swi.psy.uva.nl/projects/SWI-Prolog/`.

2 Expression de faits

Un fait est une « vérité première » :

- Loïc est le père de Gérard.
- Gérard est le père de Maxime.
- Gérard est le père de Lise.

Expression de ces faits en Prolog. On définit un prédicat `père`, c'est-à-dire une fonction à valeurs booléennes. Par défaut, ce prédicat est faux là où il n'est pas défini.

```
père(loïc,gérard).
père(gérard,maxime).
père(gérard,lise).
```

Le prédicat `père` ainsi défini n'est donc vrai que pour les couples d'argument `(loïc,gérard)`, `père(gérard, maxime)` et `père(gérard,lise)`.

Attention : on peut utiliser des lettres accentuées dans les noms d'identificateurs, mais les noms de constantes doivent toujours commencer par une minuscule.

Les faits sont écrits dans un programme, appelé par exemple `exemples` (extension « .pl » facultative). On lance l'interpréteur, puis on charge le programme dans l'interpréteur :

```
bash-2.02$ pl
Welcome to SWI-Prolog (Version 3.2.9)
Copyright (c) 1993-1999 University of Amsterdam.  All rights reserved.

For help, use ?- help(Topic). or ?- apropos(Word).

?- consult(exemples).

exemples compiled, 0.01 sec, 764 bytes.

Yes
?- [exemples].

exemples compiled, 0.01 sec, 764 bytes.

Yes
?- halt.
bash-2.02$
```

Après avoir chargé le programme, l'interpréteur nous répond « Yes » pour indiquer que la compilation s'est passée sans erreurs.

3 Questions

Les faits étant déclarés, on peut poser des questions dans l'interpréteur. L'interpréteur nous renvoie alors la valeur de vérité de la formule.

```
?- père(loïc,gérard).

Yes

?- père(loïc,maxime).

No
```

La réponse à la deuxième question est négative, car le prédicat `père` n'ayant pas été défini sur le couple `(loïc,maxime)` prend la valeur *faux* sur ce couple.

On peut aussi poser des questions existencielles : « de qui Loïc est-il le père ? »

```
?- père(loïc,X).

X = gérard

Yes
```

Ici le deuxième argument n'est plus une constante mais une variable. Un nom de variable commence toujours par une majuscule. L'interpréteur ne renvoie pas *une* mais *toutes* les valeurs possibles.

```
?- père(gérard,X).

X = maxime ;

X = lise ;

No
```

Pour ce faire il faut taper un « ; » après chaque réponse pour que l'interpréteur affiche la réponse suivante. Quand l'interpréteur a affiché toutes les réponses possibles, il renvoie « No ».

Une question peut bien sûr porter sur plusieurs variables.

```
?- père(X,Y).

X = loïc
Y = gérard ;

X = gérard
Y = maxime ;

X = gérard
Y = lise ;

No
```

Les règles sont consultées suivant l'ordre de leur écriture, ce qui n'est pas sans conséquences sur le comporte,ent et le résultat des programmes.

4 Conjonction de questions

On cherche des valeurs `Y` et `Z` telles ques `père(loïc,X)` et `père(Y,Z)` soient vrais.

```
?- père(loïc,Y),père(Y,Z).

Y = gérard
Z = maxime ;

Y = gérard
Z = lise ;

No
```

L'ordre dans lequel la question est rédigée est important : la question est élaborée de la gauche vers la droite et est évaluée de la gauche vers la droite. Nous avons trois règles :

1. père(loïc,gérard).
2. père(gérard,maxime).
3. père(gérard,lise).

La figure 1 nous montre l'arbre de recherche des solutions pour cette question. Si nous réécriovons la question comme ci-dessous :

```
?- père(Y,Z),père(loïc,Y).

Y = gérard
Z = maxime ;

Y = gérard
Z = lise ;

No
```

nous obtenons la même réponse, mais l'arbre de recherche, présenté figure 2, devient tout de suite plus important. Il vaut toujours mieux mettre en tête les prédicats les plus contraignants.

5 Introduction de règles de déduction

```
grandpère(X,Y) :- père(X,Z),père(Z,Y).
```

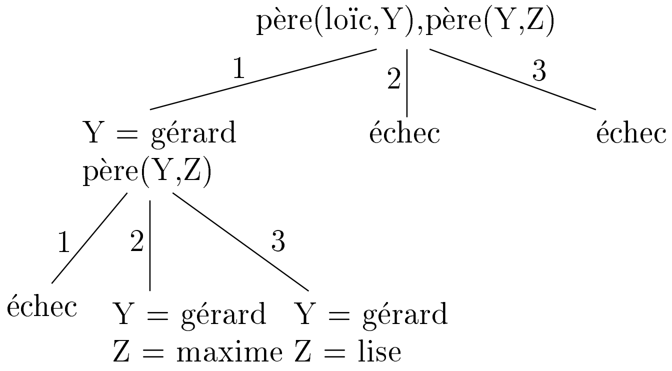

FIG. 1: Importance de l'ordre d'écriture des prédicats : première partie.

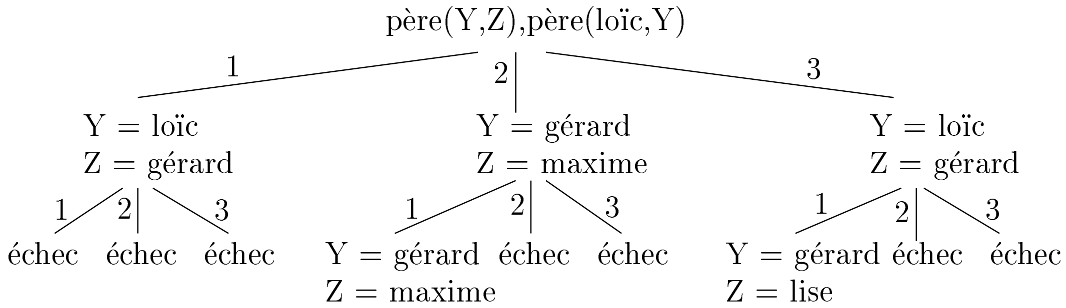

FIG. 2: Importance de l'ordre d'écriture des prédicats : deuxième partie.

Cette formule signifie que `grandpère(X,Y)` est vrai si `père(X,Z)` est vrai et si `père(Z,Y)` est vrai. Autrement dit, cette formule est équivalente à la formule :

$$\texttt{père(X,Z)} \wedge \texttt{père(Z,Y)} \rightarrow \texttt{grandpère(X,Y)}$$

Exemple d'utilisation :

```
?- grandpère(X,lise).

X = loïc ;

No
```

Règles récursives

```
ancêtre(X,Y) :- père(X,Y).
ancêtre(X,Y) :- père(X,Z),ancêtre(Z,Y).
```

Le prédicat `ancêtre` est ici défini par deux règles :

$$\begin{aligned} \texttt{père(X,Y)} &\rightarrow \texttt{ancêtre(X,Y)} \\ \texttt{père(X,Z)} \wedge \texttt{ancêtre(Z,Y)} &\rightarrow \texttt{ancêtre(X,Y)} \end{aligned}$$

Donc, le prédicat `ancêtre` est défii par la formule :

$$\texttt{père(X,Y)} \vee (\texttt{père(X,Z)} \wedge \texttt{ancêtre(Z,Y)}) \rightarrow \texttt{ancêtre(X,Y)}$$

La disjonction est obtenue pas l'introduction de plusieurs règles.

6 Principe d'effacement

Pour répondre à une question $q_1, ...q_n$, c'est-à-dire pour satisfaire une conjonction de relations, Prolog va tenter d'effacer ces « buts » un à un, dans lm'ordre où ils se présentent.

1. Soit B la suite initiale des buts. Soit C l'ensemble initial des contraintes. $C = \emptyset$.
2. Si le but est $B = b_1, b_2, ..., b_n$, on efface b_1.
3. Régle à appliquer. Soit $p : -q_1, ..., q_m$ la première des règles du programme telles que la contrainte $C \cup \{p = b_1\}$ ait une solution. Alors $B = q_1, ..., q_m, b_2, ..., b_n$ et $C = C \cup \{p = b_1\}$.
4. Itération : on recommence en 1 sauf si :
   - $B = \emptyset$, auquel cas on a obtenu une solution, la réponse étant C.
   - On bloque sur un but b_1 que l'on ne peut pas effacer, il y a alors échec.

   Dans tous les cas, il y a retour en arrière sur le dernier choix effectué, et une nouvelle tentative pour effacer, d'une autre manière, le dernier but effacer, afin de trouver une *autre* réponse possible.

   Lorsque l'on effectue un nouveau choix pour tenter d'effacer différemment un but `b`, on « défait » toutes les affectations de variables qui ont eu lieu depuis le précédent choix fait pour effacer `b` (*backtracking*).

7 Problème classique : représentation et manipulation des listes

- Liste vide : `[]`.
- Liste non vide : `[T|R]`, où `T` est un élément et `R` une liste.
- Liste à trois éléments : `[1, 2, 3]`.

On veut écrire une fonction `concatène(X,Y,Z)` qui est vraie si la liste `Z` est égale au résultat de la concaténation des liastes `X` et `Y`. On veut donc pouvoir obtenir les réponses suivantes :

```
?- concatène([1,2],[3],[1,2,3]).

Yes
?- concatène([1,2],[3],Z).

Z = [1, 2, 3]

Yes
?- concatène(X,[3],[1,2,3]).

X = [1, 2]

Yes
?- concatène(X,Y,[1,2,3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```

Construction de la solution :

```
concatène([],Y,Z) :- vrai si Y = Z
concatène([T|R],Y,Z) :- vrai si Z = [U|S], T = U et  concatène(R,Y,S)
```

Solution

```
concatène([],Y,Y).
concatène([T|R],Y,[T|Z]) :- concatène(R,Y,Z).
```

7.1 Fonction `inv`

On veut une fonction qui nous renverse une liste.

```
?- inv([1,2],Y).

Y = [2, 1]

Yes
?- inv(X,[2,1]).

X = [1, 2]

Yes
```

On définit cette fonction au moyen de deux règles :

1. `inv([],[]).`
2. `inv([X|Y],Z) :- inv(Y,T),concatène(T,[X],Z).`

Le comportement de ce prédicat est illustré par l'exemple de la figure 3.

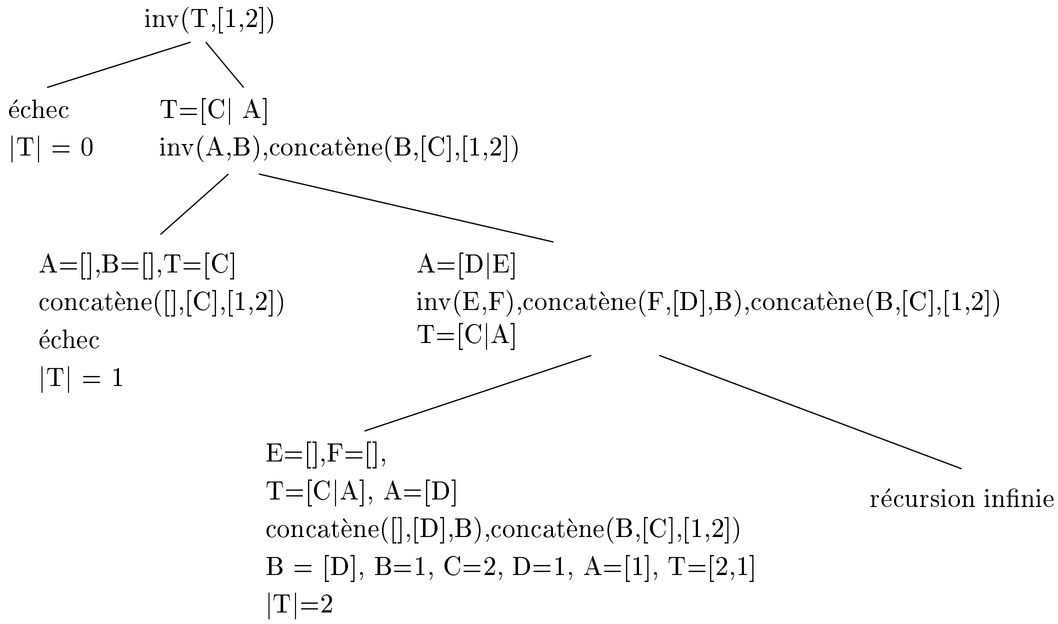

Fig. 3: Importance de l'ordre d'écriture des prédicats : récursion infinie.

On redéfinit cette fonction au moyen des deux règles :

1. `inv([],[]).`

2. `inv([X|Y],Z) :- concatène(T,[X],Z),inv(Y,T).`

Le comportement de ce prédicat est illustré par l'exemple de la figure 4.

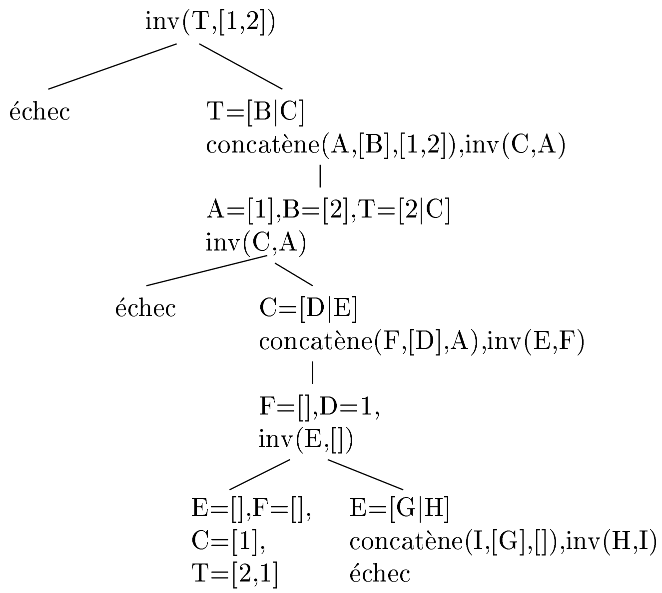

FIG. 4: Importance de l'ordre d'écriture des prédicats : récursion finie.