UPMC/master/info/4I503 APS
PROLOG

P. Manoury

Février 2019

Le langage PROLOG

Le langage PROLOG est un *langage déclaratif* basé sur la logique du premier ordre; d'où son nom *PROG[rammation] LOG[ique]*.

Définir et «calculer»

Un programme en PROLOG est une suite de *clauses de Horn*. Une telle clause est une formule de la forme $(F_1 \wedge \ldots \wedge F_n) \Rightarrow F$ (qui est équivalente à la relation de conséquence $F_1, \ldots, F_n \models F$). Les formules $F_1, \ldots, F_n$ et F sont des formules *atomiques*: application d'un symbole de *relation* (ou de prédicat) à des *termes*. La forme en est `r(`t_1`,...,`t_n`)` où `r` est un symbole de relation et les t_i sont soit des symboles de constantes, soit des symboles de variables soit des applications de la forme `f(`t_1`, ..., `t_k`)` où `f` est un symbole de fonction et t_1, ..., t_k des termes.

Dans la *syntaxe* PROLOG, les symboles de relation, de fonction ou de constantes commencent par une *minuscule* et les symboles de variables par une *Majuscule*. Le langage est du premier ordre: il n'y a ni variable de fonction, ni variable de relation (ou de prédicat).

Les clauses de Horn s'écrivent en PROLOG de la manière suivante: F `:-` F_1 `, ... ,` F_n `.` ou simplement F`.` lorsque l'ensemble d'hypothèses $F_1,\ldots,F_n$ est vide.

Par exemple considérons le célèbre syllogisme:

1. tous les hommes sont mortels
2. Socrate est un homme
3. donc Socrate est mortel

On traduit en PROLOG la première clause par

```
mortel(X) :- homme(X).
```

Notez l'utilisation de la majuscule `X` qui, en PROLOG, marque l'usage d'une *variable* qui pourra être *instanciée* – le «tous» de «tous les hommes sont mortels» est rendu par l'utilisation d'un symbole de variable. Une telle clause est appelée une *règle*. Elle dit ici comment le prédicat `mortel` peut être satisfait par un `X` indéterminé dès lors que le prédicat `homme` est satisfait par le même `X`.

On traduit la deuxième clause par

```
homme(socrate).
```

Notez l'utilisation du `s` minuscule de `socrate`. Ici, `socrate` est une constante qui pourra venir *instancier* une variable. Une telle clause est appelé un *fait*. Elle pose que `socrate` satisfait le prédcat `homme`.

Un *programme prolog* est donc un ensemble de faits et de règles – que l'on peut considérer comme un ensemble d'axiomes – dont on déduit d'autres faits – que l'on peut voir comme des conséquences des axiomes, c'est-à-dire, des théorèmes de la théorie définie par les axiomes.

Pour vérifier la conclusion de notre syllogisme (Socrate est mortel) on formule la *requête* suivante: `mortel(socrate)`. La réponse de PROLOG, qui sera «oui», est établie de la manière suivante:

- en remplaçant `X` par `socrate` dans la clause `mortel(X) :- homme(X).`, il faut établir que `homme(socrate)`;
- ce que nous donne exactement la clause `homme(socrate).`

On pourra aussi formuler la requête `mortel(platon)` et, dans ce cas, la réponse sera «non» car

- en remplaçant `X` par `platon` dans la clause `mortel(X) :- homme(X).`, il faut établir que `homme(platon)`;
- ce que ne nous donne aucune clause de notre programme.

On pourra enfin formuler une requête telle que `mortel(N)` pour laquelle la réponse sera l'ensemble des instances possibles pour la variable `N`. Ici, il n'y en aura qu'une: `socrate`.

Listes et couples, relations récursives

D'usage courant en programmation, les structures de *couple* et de *liste* ont une syntaxe dédiée en PROLOG. Couples et listes sont des *valeurs*: elles sont exprimées par des *termes*. Si t_1 et t_2 sont des termes alors `(`t_1`,`t_2`)` est le terme représentant la couples des valeurs de t_1 et t_2. La liste vide est dénotée par `[]`; la liste dont le premier élément est t_1 et dont la suite est t_s est exprimée par le terme `[`t_1`|`t_s`]`.

Il est aisé, en PROLOG, de retouver la définition de fonction dont le résultat est un booléen. Par exemple, la fonction booléenne d'appartenance à une liste est codée la relation binaire `mem` définie par les clauses:

```
mem(X, [X|_]).
mem(X, [_|XS]) :- mem(X,XS).
```

Ces clauses expriment que un `X` appartient à une liste (non vide si et seulement si il est le même que le premier élément de la liste ou il appartient à la suite de la liste. Tous les cas qui ne satisfont ni l'une ni l'autre de ces deux clauses donneront un résulat «faux» (rien n'appartient à la liste vide et `X` n'appartient à aucune liste non vide dont il différe de chaque élément).

Notez comment, dans la première clause, l'identité de l'élément cherché `X` et du premier élément de la liste est signifiée par l'identité du symbole de variable. Dans la *résolution* des requêtes portant sur `mem`, il faudra que la variable `X` puisse recevoir, en ses deux occurences, une mëme instance.

Pour ce qui est des fonctions dont le résulat n'est pas une valeur booléenne, il suffit de remplacer la définition de la fonction n-aire $f(x_1, \ldots, x_n) = y$ par la relation n+1-aire $f(x_1, \ldots; x_n, y)$ qui peut se lire: y est le résulat de f appliqué à $x_1, \ldots, x_n$.

Par exemple, la fonction qui donne la valeur associée à une clef dans une liste d'association est représentée par la relation ternaire `assoc(`t_1`,`t_2`,`t_3`)` qui est satisfaite si et seulement si t_3 est la valeur associée à la clef t_1 dans la liste d'association t_2. Elle est définie par les deux clauses:

```
assoc(X, [(X,V)|_], V).
assoc(X, [_|XS], V) :- assoc(X, XS, V).
```

Notez ici encore l'usage de l'identité de nom pour exiger l'identité d'instance (c'est-à-dire, de valeur). Notez également comment il suffit de donner les cas de satisfaction de la relation: inutile de dire qu'une clef n'a pas de valeur dans une liste vide.

SWI prolog

Pour réaliser nos programmes PROLOG, on pourra utiliser SWI prolog. La commande `swipl` nous donne une boucle d'interaction avec l'environnement de SWI prolog:

```
manoury@ssh:~$ swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

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

?-
```

L'invite `?-` nous invite à formuler une requête comme `fib(5,X)` pour obtenir la valeur de `X`.

Naturellement, cette requête n'a de chance d'aboutir que si nous avons informé le système de nos définitions. Supposons que celles-ci ont été enregistrées dans un fichier appelé `prog.pl` (extension obligatoire). Nous «chargeons» ce fichier avec la requête:

```
?- [prog].
true.

?-
```

La réponse du système est `true`, les définitions sont maintenant connues et nous pouvons demander:

```
?- fib(5,X).
X = 8.

?-
```

La réponse du système est ici la valeur qu'il a pu déterminer pour la variable présente dans notre requête.

Script

La commande `swipl` peut prendre des arguments comme le nom d'un fichier de définitions et le but initial à satisfaire. Par exemple

```
swipl -s Typage/typeChecker.pl -g main_stdin
```

est équivalente à

```
swipl
?- [Typage/typeChecker].
?- main_stdin.
```

Nous avons défini le prédicat `main_stdin` de cette manière:

```
main_stdin :-
    read(user_input,T),
    typeProg(T,R),
    print(R),
    nl,
    exitCode(R).
```

où

1. `read(user_input,T)` lit un terme PROLOG (sensé être la traduction d'un programme APS) sur l'entrée standard et le lie à `T`,
2. puis, `typeProg(T,R)` applique les règles de typage au terme `T` pour construire le résultat `R` (atome `ok` ou `ko`),
3. ce résultat est affiché puis le programme est stopé avec un code de retour (O ou 1) déterminé par la valeur de `R`.

Si `translate` est un programme qui produit sur la sortie standard la traduction en terme PROLOG d'un programme APS contenu dans un fichier est passé sur la ligne de commande, le script

```
#!/usr/bin/env bash
Syntaxe/translate $1 | swipl -s Typage/typeChecker.pl -g main_stdin
```

permet d'enchaîner la traduction d'un programme APS et sa vérification de type. On pourra utiliser le code de retour du typage pour enchaîner ou non sur l'évaluation.