Vers la programmation logique

Samuel Gallay

2 mars 2020

1 L'écriture en OCaml d'un interpréteur du langage Prolog

Prolog, pour programmation en logique est un langage créé par A. Colmerauer, en 1972 à Marseille. La programmation logique est un paradigme différent de la programmation impérative ou de la programmation fonctionnelle. On définit des faits élémentaires et des règles de la manière suivante :

```
apprend(eve, mathematiques).
apprend(benjamin, informatique).
apprend(benjamin, physique).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).

etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
```

Ces informations décrivent l'ensemble des connaissances du programme. On lit ici "Benjamin apprend l'informatique", ou "Alice enseigne la physique". La dernière ligne est une règle : "L'élève E est étudiant du professeur P si E apprend la matière M et que P enseigne M". Le symbole `:-` se lit *si*, et la virgule entre `apprend(E,M)` et `enseigne(P,M)` sinifie *et*. Ensuite l'utilisateur peut effectuer des requêtes comme ceci :

```
?-etudiant_de(E, pierre)
```

L'interpréteur Prolog renvoie alors `E = benjamin` et `E = eve`.

Ce qui différencie la programmation logique des formes plus courantes d'informatique impérative, c'est qu'on n'explique pas au programme comment résoudre la requête, on se contente de décrire le problème.

De très nombreuses introductions au langage Prolog peuvent être trouvées sur internet, je n'en cite qu'une seule qui m'est apparue comme convenable pour une première approche : *Simply Logical : Intelligent Reasoning by Example*, P. Flach, 1994 [2].

1.1 La représentation des programmes Prolog

La représentation des programmes Prolog dans OCaml se fait de la manière suivante :

— `var` est une variable (commençant par une majuscule en Prolog), identifiée par une chaîne de caractères, et un numéro. Le numéro sera utilisé pour renommer facilement les variables, pour éviter des conflits de noms.

— `atom` est une chaîne de caractères représentant les symboles de prédicats : soit des constantes (arité 0 du prédicat), soit les foncteurs (arité strictement positive). L'arité de `etudiant_de` est 2.

— un `term` est soit une variable, soit sous la forme `prédicat(terme1, terme2, ...)`, qui est donc un atome, et une liste de termes.

— une `clause` est de la forme `etudiant_de(E,P):-apprend(E,M), enseigne(P,M).`, un terme à gauche et une liste de termes à droite. `eleve(samuel).` est une clause, il n'y rien à droite.

[ ]:
```
type var = Id of string * int;;
type atom = Atom of string;;
type term = Var of var | Term of atom * (term list);;
type clause = Clause of term * (term list);;
```

Ce n'est pas moi qui ai inventé cette structure particulière : je l'ai retrouvée dans un diapo de cours en Haskell sur le langage Prolog écrit par Alan Smaill en 2009 [6].

Une structure proche est utilisée en Lisp par Peter Norvig dans *Paradigms of Artificial Intelligence Programming*, 1992 [4]. Une citation qui permet de comprendre la structure d'un programme en Prolog : *"We will build a single uniform data base of clauses, without distinguishing rules from facts. The simplest representation of clauses is as a cons cell holding the head and the body. For facts, the body will be empty."*

J'explique plus bas le *pourquoi* de cette structure, qui permet de représenter les clauses de Horn.

1.2 L'analyse lexicale : le lexer

Il faut d'abord transformer le programme qui est une chaîne de caractères, en une liste de symboles. Les `token` sont les différents symboles présents dans un programme Prolog simple. Des variables et des atomes, et ces caractères spéciaux : `( | ) | , | . | :-`.

[ ]:
```
type token = TLPar | TRPar | TDot | TIf | TAnd | TVar of string | TAtom of␣
↪string;;
```

[ ]:
```
(* Transforme une chaîne de caractères en liste et réciproquement *)
let to_list str = List.init (String.length str) (String.get str);;
let to_string lst = String.init (List.length lst) (List.nth lst);;


(* Enlève les espaces, tabulations de la liste de caractères *)
let rec remove_spaces = function
| (' ' | '\t' | '\n')::l -> remove_spaces l
| l -> l;;

(* Retire le premier mot (suite de lettres et de '_') de la liste *)
let rec read_word char_list mot = match char_list with
| ('A'..'Z' | 'a'..'z' | '_' as c)::l -> read_word l (c::mot)
| char_list -> char_list, to_string (List.rev mot);;

(* Pour séparer les variables et les atomes *)
```

```
let string_to_atomvar = function
| str -> (match String.get str 0 with
                | ('A'..'Z') -> TVar str
                | ('a'..'z') -> TAtom str
                | _ -> failwith "Wrong Word")

(* Transforme la liste de caractères en liste de symboles *)
let rec charlist_to_tokenlist liste_car = let lst = remove_spaces liste_car in␣
 ↪match lst with
| [] -> []
| '.'::l -> TDot::(charlist_to_tokenlist l)
| '('::l -> TLPar::(charlist_to_tokenlist l)
| ')'::l -> TRPar::(charlist_to_tokenlist l)
| ':'::'-'::l -> TIf::(charlist_to_tokenlist l)
| ','::l -> TAnd::(charlist_to_tokenlist l)
| ('A'..'Z' | 'a'..'z' | '_')::_ -> let l, m = read_word lst []
in (string_to_atomvar m)::(charlist_to_tokenlist l)
| l -> failwith ("Wrong Syntax in " ^ (to_string l));;

(* Transforme la chaîne de caractères en liste de symboles *)
let lexer str = charlist_to_tokenlist (to_list str);;
```

1.2.1 Un exemple d'utilisation de l'analyseur lexical :

[ ]:
```
let program = "apprend(eve, mathematiques).
apprend(benjamin, informatique).
apprend(benjamin, physique).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).
etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
egal(R,R).";;

lexer program;;
```

Quelques remarques sur le lexer : j'ai bien repris les idées du livre *Le Langage Caml*, Pierre Weis et Xavier Leroy, 1992 [7]. Une grande différence est l'utilisation des List à la place des Stream. L'utilisation des Stream semble plus intelligente, mais ils ont pour des raisons qui m'échappent ils ont presque disparu des dernières versions d'OCaml (le pattern-matching est beaucoup plus difficile). Avec les listes il y a bien plus de copies, mais nous n'avons pas là de véritable problème de performance.

1.3 L'analyse syntaxique : le parser

Il faut maintenant transformer la liste de `tokens` en une structure formée des types `clause`, `term`, `atom` et `var`.

1.3.1 Isoler les clauses séparées par des points

[ ]:
```
(* Sépare les différentes clauses *)
let extract_clauses token_list =
let rec f tk_lst tmp = match tk_lst with
| [] -> if tmp = [] then [] else failwith "No end of clause"
| TDot::l -> (List.rev tmp)::(f l [])
| t::l -> f l (t::tmp)
in f token_list [];;

extract_clauses (lexer program);;
```

1.3.2 Isoler les termes de droite et de gauche d'une clause séparés par :-

[ ]:
```
(* Sépare les membres de droite et de gauche d'une clause *)
let extract_rightleft cl =
    let rec f cl left = match cl with
    | [] -> (List.rev left), []
    | TIf::l -> (List.rev left), l
    | t::l -> f l (t::left)
in f cl [];;

extract_rightleft (lexer "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)");;
```

1.3.3 Isoler les différents termes d'une suite de termes séparés par des virgules

[ ]:
```
(* Sépare les termes d'une liste de termes *)
let extract_term_list tk_lst =
let rec f tk_lst term_tmp lvl = match tk_lst with
    | [] -> [term_tmp]
    | TLPar::l -> f l (term_tmp @ [TLPar]) (lvl + 1)
    | TRPar::l -> if lvl > 0 then f l (term_tmp @ [TRPar]) (lvl - 1)
                  else failwith "Mismatched Parenthesis"
    | TAnd::l when lvl = 0 -> term_tmp::(f l [] 0)
    | t::l -> f l (term_tmp @ [t]) lvl
in f tk_lst [] 0;;

extract_term_list (lexer "apprend(E,M), enseigne(P,M)");;
```

1.3.4 Transformer la liste de symboles en un terme

[ ]:
```
(* Renvoie l'intérieur d'une liste de symboles dont les extrêmes sont des␣
 ↪parenthèses *)
let inside_parenthesis = function
| [] -> failwith "Empty List"
| TLPar::l -> (let rec f = function
```

```
                | [] -> failwith "Mismatched Parenthesis"
                | [TRPar] -> []
                | h::t -> h::(f t)
                in f l)
| _ -> failwith "List is not beginning with TLPar";;

inside_parenthesis (lexer "(E,M)")
```

[ ]:
```
(* Prend une liste de tokens qui est censée ne représenter qu'un terme␣
↪'mouton', 'élève(samuel)', 'X',
'apprend(X, Y)' et renvoie ce terme *)
let rec tokenlist_to_term = function
| [] -> failwith "Error : [] is not a term"
| (TVar str)::[] -> Var (Id (str, 0))
| (TVar str)::_ -> failwith "No other arguments with Var"
| (TAtom str)::[] -> Term(Atom str, [])
| (TAtom str)::l -> Term(Atom str, List.map tokenlist_to_term␣
↪(extract_term_list (inside_parenthesis l)))
| _ -> failwith "Not a valid term";;

tokenlist_to_term (lexer "apprend(benjamin, informatique)");;
```

1.3.5 Transformer les symboles en clauses, le programme en liste de clauses

[ ]:
```
(* Transforme une liste de symboles représentant une clause en la clause elle␣
↪même *)
let tokenlist_to_clause tklcl =
let c = extract_rightleft tklcl in let l = fst c and r = snd c
in Clause (tokenlist_to_term l, if r = [] then [] else List.map␣
↪tokenlist_to_term (extract_term_list r));;

(* Transforme la chaîne de caractères du programme en la liste de clauses du␣
↪format voulu *)
let parser str = List.map tokenlist_to_clause (extract_clauses (lexer str)) ;;
```

1.3.6 Quelques petits parsers pratiques

[ ]:
```
(* Un parser sur une chaîne représentant un terme *)
let parser_term str = tokenlist_to_term (lexer str);;

(* Un parser sur une chaîne représentant une liste de termes *)
let parser_term_list str = List.map tokenlist_to_term (extract_term_list (lexer␣
↪str));;

(* Un parser sur une chaîne (sans le point) représentant une clause *)
let parser_clause str = tokenlist_to_clause (lexer str);;
```

```
parser_term "apprend(benjamin, informatique)";;

parser_term_list "apprend(E,M), enseigne(P,M)";;

parser_clause "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)"
```

1.3.7 Un exemple

[ ]:
```
let world = parser program;;
```

1.3.8 Remarques sur le parser

Je l'ai écrit moi-même, sans reprendre une structure existante… C'est très inefficace : le programme est parcouru au moins 3 fois par les 3 fonctions d'isolement, avant même d'entrer dans la fonctions récursive `tokenlist_to_term` qui sert réellement à créer l'arbre… De plus, le programme est copié plein de fois dans les fonctions de parcours. Néanmoins le coût total me parait de loin proportionnel à la longueur du code, et probablement proportionnel au niveau d'imbrication `(())` des parenthèses (ce qui pourrait certainement être évité). Je pense que ce n'est pas le plus important : les coûts d'exécution seront bien plus problématiques…

Je pense qu'il pourrait exister encore des codes Prolog dont la syntaxe est fausse et qui passent le parser. (En tous cas je ne suis pas assuré du contraire). Il faut se demander si l'on peut représenter des programmes faux dans les structures formées des types `clause`, `term`, `atom` et `var`.

Remarque : vous êtes priés de ne rentrer que des codes justes dans le parser, les messages d'erreur sont plus que lacunaires…

1.3.9 Quelques fonctions d'affichage pour le débogage

[ ]:
```
let rec term_to_string = function
| Var (Id (str,n)) -> str ^ (string_of_int n)
| Term (Atom a, []) -> a
| Term (Atom a, l) -> a ^ "(" ^ (String.concat ", " (List.map term_to_string␣
 ↪l)) ^ ")";;

let term_list_to_string l = String.concat ", " (List.map term_to_string l);;

let clause_to_string (Clause (h, l)) = (term_to_string h) ^ ":-" ^␣
 ↪(term_list_to_string l);;
```

1.4 L'algorithme derrière Prolog

Ma référence en ce qui concerne l'algorithme utilisé par Prolog est *Logic Programming and Prolog*, 1990, Nilsson et Małuszyński [3].

Ce qui semble être le premier article traitant du type de résolution utilisé par Prolog est *A Machine-Oriented Logic Based on the Resolution Principle*, Robinson, 1965 [5]. Cet article introduit l'unification, et le *principe de résolution* (Resolution Principle) dont dérive la SLD-resolution (Linear resolution for Definite clauses with Selection function) utilisée par Prolog.

Une *clause définie*, aussi nommée clause de Horn est de la forme `(P1 et P2 et ... et Pn) => Q`. En particulier, `(non P) => Q` n'est pas une clause de Horn. On peut par contre exprimer`(P1 ou P2) => Q` avec des clauses de Horn, il suffit d'écrire les deux clauses `P1 => Q` et `P2 => Q`. Les versions premières de Prolog se limitent aux clauses de Horn, pour une principale raison : la *correction* et la *complétude* (*soundness* and *completeness*) ont été montrée pour la SLD-resolution, qui n'est valide que sur des clauses de Horn. Pour les citations (un peu compliquées) : la SLD-resolution est introduite par Kowalski en 1974, la correction est montrée par Clark en 1979, la complétude a été montrée premièrement par Hill en 1974, mais quelque chose de plus fort a été montré par Clark en 1979.

1.4.1 L'unification

L'unification se fait entre deux termes A et B. Deux termes s'unifient s'il existe une substitution `thêta` des variables de A telle que `B = thêta(A)`.

Soit la clause du programme Prolog `etudiant_de(E,P):-apprend(E,M), enseigne(P,M).`. Si l'on cherche à réaliser la requête `?-étudiant_de(E, pierre)`, on unifie `étudiant_de(E, pierre)` avec `etudiant_de(E,P)`, le membre de gauche de la clause. La substitution thêta remplace `P` par `pierre` et ne modifie pas les autres variables. Pour continuer la recherche, on applique la substitution au termes de droite de la clause.

Voici l'algorithme décrit dans [3] :

```
E est l'ensemble des équations. Au départ il n'y en a qu'une seule :
Par exemple E = {etudiant_de(E,P) = étudiant_de(E, pierre)}.

Répéter tant que E change
(jusqu'à ce que l'on ne puisse plus rien appliquer aux équations)
    Sélectionner une équation  s = t dans E;
    Si s = t est de la forme :
        f(s1, ..., sn) = f(t1, ..., tn) avec n >= 0
            Alors remplacer l'équation par s1=t1 ... sn=tn
        f(s1, ..., sm) = g(t1, ..., tn) avec f != g
            Alors ÉCHEC
        X = X
            Alors supprimer l'équation
        t = X où t n'est pas une variable
            Alors remplacer l'équation par X = t
        X = t où X != t et X apparait plus d'une fois dans E
            Si X est un sous-terme de t Alors ÉCHEC
            Sinon on remplace toutes les autres occurences de X par t
```

Il est prouvé que cet algorithme termine et renvoie soit échec, soit un ensemble équivalent des équations sous *forme résolue*. Un ensemble d'équations `{X1 = t1, ..., Xn = tn}` est dit sous forme résolue si les `(Xn)` sont des variables, les `(tn)` sont des termes et aucune des variables `Xn`

n'apparait dans les `tn`. Cela permet ensuite de déterminer un unifieur.

Remarques :

— C'est dommage que l'algorithme se prête si bien à la programmation impérative quand on programme dans un langage fonctionnel.

— On trouve des algorithmes pour déterminer un unifieur qui ne manipulent pas exactement comme ça des systèmes d'équations, (et qui sont dans des styles plus fonctionnels), mais cet algorithme est le seul que j'arrive à comprendre convenablement, et en plus [3] prouve sa terminaison et sa correction, ce qui est très bon point.

— Le test `Si X est un sous-terme de t Alors ÉCHEC` est en pratique pas réalisé dans la plupart des implémentations Prolog (pas comme ça en tout cas), il est très lent et le cas n'arrive que très peu en pratique. On peut donc avoir des boucles infinies… Les versions "modernes" gèrent les unifications de structures infinies… Une citation de [4] "*This represents a circular, infinite unification. Some versions of Prolog, notably Prolog II (Giannesini et al. 1986), provide an interpretation for such structures, but it is tricky to define the semantics of infinite structures.*"

[ ]:
```
(* Teste si une variable est dans un terme *)
let rec var_in_term v = function
| Var w -> v = w
| Term (_, l) -> List.mem true (List.map (var_in_term v) l);;

var_in_term (Id ("P", 0)) (parser_term "etudiant_de(E,P)");;

(* Remplace une variable par un nouveau terme, récursivement à l'intérieur d'un␣
 ↪terme *)
let rec replace_var_in_term var new_t = function
| Var v when v = var -> new_t
| Var v -> Var v
| Term (a , l) -> Term (a, List.map (replace_var_in_term var new_t) l);;

replace_var_in_term (Id ("E",0)) (parser_term "pierre") (parser_term␣
 ↪"etudiant_de(E,P)");;
```

[ ]:
```
(* Teste si une variable est dans une liste de couples de termes *)
let var_in_eql v e = List.mem true (List.map (function t -> var_in_term v (fst␣
 ↪t) || var_in_term v (snd t)) e);;

var_in_eql (Id ("F", 12)) [Var (Id ("F", 12)), Var (Id ("A", 0)); Var (Id ("A",␣
 ↪0)), Var (Id ("E", 27))];;

(* Remplace une variable par un nouveau terme, à l'intérieur d'une liste de␣
 ↪couples de termes *)
let replace_var_in_eql var new_t eql =
List.map (function (a, b) -> (replace_var_in_term var new_t a),␣
 ↪(replace_var_in_term var new_t b)) eql;;
```

[ ]:
```
(* E est une liste de couples de termes *)
(* transforme E en un équivalent sous forme résolue (type option, None si␣
 ↪impossible) *)
let rec solve e =
    (* Parcourt une fois l'ensemble des équations *)
    let rec pass tmp = function
    | [] -> Some tmp

    | (Term (f, lf), Term (g, lg))::l when f = g && List.length lf = List.
 ↪length lg ->
        pass ((List.map2 (fun a b -> a,b) lf lg)@tmp) l

    | (Term (_, _), Term (_, _))::l -> None (* failwith "Unification failed" *)

    | (Var x, Var y)::l when x = y -> pass tmp l

    | (Term (a, la), Var x)::l -> pass ((Var x, Term (a, la))::tmp) l

    | (Var x, t)::l when var_in_eql x tmp || var_in_eql x l ->
        if var_in_term x t then None (* failwith "Unification failed, loop" *)
        else pass ((Var x, t)::(replace_var_in_eql x t tmp))␣
 ↪(replace_var_in_eql x t l)

    | (Var x, t)::l -> pass ((Var x, t)::tmp) l

in match (pass [] e) with
| None -> None
| Some eqlpp -> let b = List.sort_uniq compare eqlpp in if e = b then Some e␣
 ↪else solve b;;
```

À la place de manipuler des ensembles (`Set` existe en OCaml), j'ai juste utilisé des liste triées. La fonction `compare` est magique, elle définit une relation d'ordre pour n'importe quel type. J'ai ici utilisé la version complète de l'algorithme, qui n'est certainement pas la plus efficace. Comme l'a dit Donald Knuth : "*Premature optimization is the root of all evil (or at least most of it) in programming.*"

[ ]:
```
solve [parser_term "etudiant_de(E, P)", parser_term "etudiant_de(S, P)"];; (* E␣
 ↪= S *)
```

On définit une substitution comme une liste de couples (var,term), le terme est inséré à la place de la variable.

Si l'on sait que `etudiant_de(E1,P):-apprend(E1,M), enseigne(P,M).` (c) et que l'on veut montrer `étudiant_de(E0, pierre).` (r), alors il faut trouver une substitution qui unifie la tête de la clause (c) avec la requête (r).

Pour l'exemple plus haut, il faut la substitution `thêta = { P/pierre, E1/E0}`. Il faut comprendre "La variable P est remplacée par 'pierre', la variable E1 est remplacée par la variable E0. C'est bien

un unifieur, si on l'applique sur la requête et sur la tête de la clause, les deux deviennent égales.

Il peut exister plusieurs unifieurs, par exemple `thêta2 = { P/pierre, E0/samuel, E1/samuel}` en est aussi un. On voit que thêta2, c'est thêta composé avec `{E0/samuel}`, on dit alors que thêta est plus général que thêta2. Nous, nous cherchons le plus général de ces unifieurs, le MGU pour *Most General Unifier*.

Un détail : on peut avoir thêta plus général que oméga, et oméga plus général que théta, la relation n'est donc pas antisymétrique. En fait, le MGU est unique au renommage des variables près.

On peut déduire facilement un MGU du système d'équations sous forme résolue de la fonction `solve` (c'est à ça qu'elle sert). Si `{X1 = t1, ..., Xn = tn}` est sous forme résolue, alors `{X1/t1, ..., Xn/tn}` est un MGU.

Lorsque l'on veut unifier `etudiant_de(E,P)` et `etudiant_de(E,pierre)`, les deux variables `E` ne doivent pas être nommées de la même manière.

[ ]:
```
(* Prend 2 terme et renvoie l'unifieur le plus général s'il existe, None sinon.
Attention, aucune variable ne doit être présente dans (r) la requête et dans␣
 ↪(c) la tête de la clause *)

let rec mgu r c = let f = function
    | Var x, Term (a, l) -> x, Term (a, l)
    | Var x, Var y -> x, Var y
    | Term (_, _), _ -> failwith "Should not happen."
in match solve [r, c] with
| None -> None
| Some e -> Some (List.map f e);;
```

[ ]:
```
(* La fonction test_unification a pour but de tester mgu *)
let test_unification str_r str_c = mgu (parser_term str_r) (parser_term str_c);;

test_unification "etudiant_de(E, pierre)" "etudiant_de(F,P)";; (* F/E et P/
 ↪pierre *)

test_unification "etudiant_de(F,P)" "etudiant_de(E, pierre)";; (* E/F et P/
 ↪pierre *)

test_unification "f(X,g(Y))" "f(g(Z),Z)";; (* X/g(g(Y)) et Z/g(Y) *)

test_unification "f(g(Z),Z)" "f(X,g(Y))";; (* X/g(g(Y)) et Z/g(Y) *)
```

Quelques explications sur les derniers deux exemples :

— Je sais que pour tout `Z`, `f(g(Z),Z)` est vraie, et j'aimerais savoir s'il existe des couples `(X,Y)` tels que `f(X,g(Y))` soit vraie. Le programme me répond oui, il suffit de choisir `X = g(g(Y))`

— Je sais que pour tout `(X,Y)`, `f(X,g(Y))` est vraie, et j'aimerais savoir s'il existe des `Z` tels que `f(g(Z),Z)` soit vraie. Le programme me répond oui, il suffit de choisir `Z = g(Y)`

1.4.2 Le backtracking

Il me semble que nous nous rapprochons du but ! L'unification est une partie très importante de la SLD-resolution. L'autre point important est le backtracking. Encore quelques fonctions pour appliquer des substitutions sur des termes, listes de termes, pour composer des substitutions, pour vérifier que un terme et une clause n'ont pas de variables en commun, pour effectuer les renommages si nécessaire, et après je pense qu'il sera possible d'implémenter le backtracking.

[ ]:
```
(* Applique une substitution sur un terme *)
let rec apply_subst_on_term uni term = match uni with
| [] -> term
| (v,t)::l -> apply_subst_on_term l (replace_var_in_term v t term);;

(* Applique une substitution sur une liste de termes *)
let rec apply_subst_on_termlist uni lst = List.map (apply_subst_on_term uni)␣
↪lst;;
```

Voilà la manière dont j'ai compris l'algorithme :

- On cherche à satisfaire une requête `<- A1,...An`.
- On a `A1` qui unifie avec `Hi`, où `Hi <- Ci_1,..Ci_m` est une la i-ème clause du programme. On a auparavant renommé toutes les variables de `Hi <- Ci_1,..Ci_m` qui étaient présentes dans `A1`, sinon on ne peut pas appeler `mgu`. On a donc un MGU `thêta1`.
- La nouvelle requête à satisfaire est `<- thêta1(Ci_1,...Ci_m, A2,...An)`.
- On récure, si on a à montrer `thêta_k( ... thêta2( thêta1( _vide_) ) ...)` alors c'est gagné (`_vide_` signifie toujours vrai), la composée `thêta_tot` des `thêta_k` est une substitution des variables de la requête de départ. Ce qu'on veut renvoyer à l'utilisateur c'est l'image des variables de `A1,...An` par `thêta_tot`.
- Si `A1` ne s'unifie avec aucun `Hi`, ça ne sert à rien d'essayer d'unifier `A2`, puisque de toute façon on garde `A1` dans la nouvelle requête. Alors il faut remonter. C'est à dire qu'il faut essayer les autres unifications à l'étape d'avant.

Comment renommer les clauses pour avoir des noms libres à chaque unification ? On peut utiliser le niveau de récursion comme identifiant, que l'on place dans l'entier transporté avec la variable.

- On veut des variables numérotées à 0 dans la requête.
- On passe 1 lors du premier appel de sld : la première clause utilisée est renommée à 1, les variables de la requête sont toutes à 0.
- La deuxième clause utilisée est renommée à 2, dans la requête il y a des variables à 1 et à 0 : pas de conflit ! etc...

[ ]:
```
(* Prend un entier et une clause, renvoie la clause avec les variables␣
↪renommées à l'entier *)
let rename n (Clause (t1, tl)) =
let rec f = function
| Var (Id (str, _)) -> Var (Id (str, n))
| Term (atm, l) -> Term (atm, List.map f l)
```

```
in Clause (f t1, List.map f tl);;

rename 42 (parser_clause "etudiant_de(E,P):-apprend(E,M), enseigne(P,M)");;
```

[ ]:
```
(* La fonction de recherche, s'arrête à la première solution trouvée *)
let rec sld (world : clause list) (req : term list) (subs : (var * term) list␣
↪list) n =
(*Format.printf "%n %s\n%!" n (term_list_to_string req);*)
match req with
| [] -> Some subs
| head_request_term::other_request_terms ->
    let rec f = function
    | [] -> None
    | c::other_clauses -> let Clause (left_member, right_member) = rename n c in
        (match mgu head_request_term left_member with
        | None -> f other_clauses
        | Some unifier -> (*Format.printf "Used %s\n%!" (clause_to_string␣
↪(rename n c));*)
            (match sld world
            (apply_subst_on_termlist unifier (right_member@other_request_terms))
            (unifier::subs) (n+1) with
            | None -> (*Format.printf "Failed\n%!";*) f other_clauses
            | Some subst -> Some subst )
        )
    in f world;;
```

1.4.3 Déchiffrer les retours de la fonction sld

[ ]:
```
(* Compose correctement la liste de substitutions *)
let compose l =
let compose2 s1 s2 = s1@(List.map (function (v2, lt2) -> (v2,␣
↪apply_subst_on_term s1 lt2)) s2)
in List.fold_left compose2 [] l;;

(* Recherche les termes qui sont des variables récursivement dans une liste de␣
↪termes *)
let rec find_vars_in_termlist tl = List.sort_uniq compare (List.concat (List.map
(function
| Var v -> [Var v]
| Term (_,l) -> find_vars_in_termlist l)
tl))

(* Effectue un requête *)
let request world req = match (sld world (parser_term_list req) [] 1) with
| None -> None
```

```
| Some l -> Some (apply_subst_on_termlist (compose l) (find_vars_in_termlist␣
↪(parser_term_list req)));;
```

1.4.4 Enfin des essais !

[ ]:
```
request world "etudiant_de(E, pierre)";;

request world "etudiant_de(E, pierre), etudiant_de(E, alice)";;

request world "etudiant_de(A, B)";;

request world "etudiant_de(A, A)";;

request world "apprend(A, A)";;

request world "enseigne(A, A)";;

request world "enseigne(alice, physique)";;

request world "enseigne(alice, mathematiques)";;
```

1.5 Que faire maintenant ?

— Beaucoup de tests pour vérifier si tout fonctionne.

— Peut-être nettoyer un peu le code…

— Écrire quelques fonctions comme une ligne de commande interactive, pour une utilisation plus facile. Rendre les résultats plus clairs.

— Proposer une deuxième version de la recherche, qui renvoie tous les résultats possibles, ou qui propose de chercher le résultat suivant.

— Je pensais essayer d'écrire un programme pour "*résoudre*" le Cluedo. J'avais déjà essayé sans succès en C++. Le problème semble plus adapté au langage Prolog : "*tel joueur possède telle carte*", "*si un joueur montre une carte à un autre pour réfuter une supposition (triplet de cartes), c'est qu'il possède une des trois cartes*", sont des règles que l'on peut écrire en Prolog. Malheureusement j'ai trouvé un super mémoire de Master [1] qui traite bien le sujet.

— Je regarde un peu comment est-ce que l'on peut sortir des clauses de Horn, mais c'est un sujet compliqué.

— Je voulais voir s'il est possible d'étendre à des langages fonctionels des éléments de programmation logique. C'est aussi un thème intéressant, qui se nomme *functional logic programming*. On trouve plusieurs petits langages sur internet qui utilisent des approches différentes, mais aucun d'eux ne semble bien abouti. Ce serait presque un idéal d'associer des mécanismes efficaces des langages fonctionnels comme l'évaluation paresseuse, et les capacités de la programmation logique pour décrire les problèmes…

— Pour compiler Prolog, il faut se tourner vers les *Machines abstraites de Warren*.

Références

[1] V. I. Aartun. Reasoning about knowledge and action in cluedo using prolog. 2016.

[2] P. Flach. *Simply Logical : Intelligent Reasoning by Example*. 1994.

[3] U. Nilsson and J. Małuszyński. *Logic, Programming and Prolog*. 1990.

[4] P. Norvig. *Paradigms of Artificial Intelligence Programming*. 1992.

[5] J. A. Robinson. A machine-oriented logic based on the resolution principle. 1965.

[6] A. Smaill. Logic programming. 2009.

[7] P. Weis and X. Leroy. *Le langage Caml*. 1992.