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 :

```
etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
apprend(benjamin, informatique).
apprend(benjamin, physique).
apprend(eve, mathematiques).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).
```

Ces informations décrivent l'ensemble des connaissance du programme. Ensuite l'utilisateur peut effectuer des requêtes :

```
?-étudiant_de(E, pierre)
```

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

Ce qui différencie la programmation logique de l'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 (en majuscule en Prolog), identifiée par une chaine, et un numéro. Le numéro est nécessaire à la substitution, pour éviter les conflits entre les noms de variables il faut un renommage.
- `atom` est une chaine de caractères représentant les symboles de prédicats : soit des constantes (arité 0 du prédicat), soit les foncteurs (arité strictement positive).
- un `term` est soit une variable, soit sous la forme `expression(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, et qui est une modification de la structure en Lisp de *Paradigms of Artificial Intelligence Programming*, Peter Norvig, 1992 [4] : *"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 holdingthe 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 chaine de caractères, en une liste de symboles. Les `token` sont les différents symboles présents dans un programme Prolog simple. Des mots, et ces caractères spéciaux : `( | ) | , | . | :-` Parmi les mots, on distingue ensuite les variables et les atomes.

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

[ ]:
```
(* Transforme une chaine 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 enleve_blancs = function
| (' ' | '\t' | '\n')::l -> enleve_blancs l
| l -> l;;

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

(* Transforme la liste de caractères en liste de symboles *)
let rec lire_token liste_car = let lst = enleve_blancs liste_car in match lst␣
↪with
| [] -> []
| '.'::l -> TDot::(lire_token l)
| '('::l -> TLPar::(lire_token l)
| ')'::l -> TRPar::(lire_token l)
| ':'::'-'::l -> TIf::(lire_token l)
| ','::l -> TAnd::(lire_token l)
```

```
| ('A'..'Z' | 'a'..'z' | '_')::_ -> let l, m = lire_mot lst [] in (TWord m)::
 ↪(lire_token l)
| l -> failwith ("Wrong Syntax in " ^ (to_string l));;
```

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

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

let tkl = lire_token (to_list program);;
```

1.2.2 Une dernière étape : séparer les variables des atomes

[ ]:
```
(* Pour séparer les variables et les atomes *)
let tword_to_atomvar = function
| TWord str -> (match String.get str 0 with
                | ('A'..'Z') -> TVar str
                | ('a'..'z') -> TAtom str
                | _ -> failwith "Wrong Word")
| t -> t;;

let tkl_atomvar = List.map tword_to_atomvar tkl;;
```

Quelques remarques sur le lexer : j'ai bien repris les idées du livre *Le Langage Caml*, Pierre Weis et Xavier Leroy, 1992 [6]. 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 problèmes de performance. (Du coup mes fonctions sont sans effets de bord, purement fonctionnelles…)

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

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

```
let cl_lst = extract_clauses tkl_atomvar [];;
```

1.3.2 Isoler les termes de droite et de gauche d'une clause

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

let rl = extract_rl (List.hd cl_lst) [];;
```

1.3.3 Isoler les différents termes d'une suite de termes

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

let r = snd rl in extract_term_lst r [] 0;;
```

1.3.4 Transformer la liste de symboles en un terme

[ ]:
```
(* Renvoie l'intérieur d'une liste de symboles dont les extrèmes sont des␣
 ↪parenthèses *)
let in_par = 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";;

(* 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 tkl_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 tkl_to_term (extract_term_lst␣
 ↪(in_par l) [] 0))
| _ -> failwith "Not a valid term";;

in_par [TLPar; TVar "E"; TAnd; TVar "M"; TRPar];;

tkl_to_term (fst rl);;
```

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 tklcl_toclause tklcl =
let c = extract_rl tklcl [] in let l = fst c and r = snd c
in Clause (tkl_to_term l, if r = [] then [] else List.map tkl_to_term␣
 ↪(extract_term_lst r [] 0) );;

(* Transforme la chaine de caractères du programme en la liste de clauses du␣
 ↪format voulu *)
let str_to_tree str = List.map tklcl_toclause (extract_clauses (List.map␣
 ↪tword_to_atomvar (lire_token (to_list str))) []);;
```

1.3.6 Quelques exemples

[ ]:
```
tklcl_toclause (List.hd cl_lst);;

tklcl_toclause [TAtom "apprend"; TLPar; TAtom "benjamin"; TAnd; TAtom␣
 ↪"informatique"; TRPar];;

let world = str_to_tree "etudiant_de(E,P):-apprend(E,M), enseigne(P,M).
apprend(benjamin, informatique).
apprend(benjamin, physique).
apprend(eve, mathematiques).
enseigne(alice, physique).
enseigne(pierre, mathematiques).
enseigne(pierre, informatique).";;
```

1.3.7 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 `tkl_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…

1.4 L'algorithme derrière Prolog

Ma référence en ce qui concerne l'algorithme utilisé par Prolog est *Logic Programming and Prolog*, 1990, Nilsson et Maluszynski [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 le programme Prolog suivant `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);;

let t = Term (Atom "etudiant_de", [Var (Id ("E", 0)); Var (Id ("P", 0))]);;

var_in_term (Id ("P", 0)) t;;

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

rpl_var_term (Id ("E",0)) (Var (Id ("E",27))) t;;
```

[ ]:
```
(* 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 rpl_var_eql var new_t eql =
List.map (function (a, b) -> (rpl_var_term var new_t a), (rpl_var_term var␣
↪new_t b)) eql;;

rpl_var_eql (Id ("A", 0)) (Var (Id ("Salut", 42))) [Var (Id ("F", 12)), Var (Id␣
↪("A", 0)); Var (Id ("A", 0)), Var (Id ("E", 27))];;
```

[ ]:
```
(* 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)::(rpl_var_eql x t tmp)) (rpl_var_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;;
```

[ ]:
```
let t1 = Term (Atom "etudiant_de", [Var (Id ("E", 0)); Term (Atom "pierre",␣
↪[])]);;
let t2 = Term (Atom "etudiant_de", [Var (Id ("S", 0)); Var (Id ("P", 0))]);;

solve [t1, t2];;
```

Une substitution est une liste de couples (var,term), le term est inséré à la place de la variable.

On sait : `etudiant_de(E1,P):-apprend(E1,M), enseigne(P,M).` (c) et on veut montrer `étudiant_de(E0, pierre).` (r). On veut unifier la tête de la clause (c) avec la requête (r).

Pour l'exemple plus haut, il faut : `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 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. On peut déduire facilement un MGU, (Most General Unifier) du système d'équations sous forme résolue de la fonction solve (c'est à ça qu'elle sert).

Le seul petit détail, c'est qu'il n'y a pas un unique MGU, on peut avoir thêta plus général que oméga, et oméga plus général que théta, la relation n'est pas antisymétrique. Le MGU est unique au renommage des variables près, et ça pose quelques petits soucis d'implémentation, si on ne veut pas avoir de conflits des variables…

En fait, il faut réorienter convenablement les équations du système sous forme résolue pour trouver le MGU qui nous intéresse. Si dans l'équation il y a une variable et un terme qui n'est pas une variable, alors `Var/terme` si il y a deux variables, normalement il y en a une qui vient de la requête et une de la tête de la clause, à ce moment là `Var(tête_clause)/Var(requête)`.

[ ]:
```
(* 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 *)
(* le mgu r c tend à conserver les variables de (r) plutôt que celles de (c) *)
let rec mgu r c = let f = function
    | Var x, Term (a, l) -> x, Term (a, l)
    | Var x, Var y -> if var_in_term x c then x, Var y else y, Var x
    | Term (_, _), _ -> failwith "I never thought this could happen, I'm sorry"
in match solve [r, c] with
| None -> None
| Some e -> Some (List.map f e);;

mgu t1 t2;;
```

[ ]:
```
(* La fonction bricolage a pour but de tester mgu *)
let bricolage str_r str_c =
let f str = tkl_to_term (List.map tword_to_atomvar (lire_token (to_list str)))
in mgu (f str_r) (f str_c);;
```

```
bricolage "etudiant_de(E, pierre)" "etudiant_de(F,P)";; (* F/E et P/pierre *)

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

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

bricolage "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.

[ ]:
```
let rec uni_on_term uni term = match uni with
| [] -> term
| (v,t)::l -> uni_on_term l (rpl_var_term v t term);;

[t1;t2];;
uni_on_term (Option.get (mgu t1 t2)) t2;;
```

[ ]:
```
let rec uni_on_term_lst uni lst = List.map (uni_on_term uni) lst;;

uni_on_term_lst (Option.get (mgu t1 t2)) [t1;t2];;
```

— 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.

On doit maintenant écrire une fonction pour le renommage, ce qui n'est pas la partie la plus agréable. La fonction prend un terme et une clause, et cherche dans la clause s'il y a des variables qui sont aussi dans le terme : ensuite elle renomme ces variables de la clause avec un nom qui est libre.

Une autre idée : utiliser le niveau de récursion comme identifiant, que l'on place dans l'entier transporté avec la variable, cela semble beaucoup plus efficace, un seul parcours de la clause ! (dans un style un peu impératif).

— 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 (Clause (t2, [t1;t2]));;
```

[ ]:
```
let rec sld (world : clause list) (req : term list) (subs : (var * term) list␣
 ↪list) n = 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 -> (
            match sld world (uni_on_term_lst unifier␣
 ↪(right_member@other_request_terms)) (unifier::subs) (n+1) with
            | None -> f other_clauses
            | Some subst -> Some subst)
        )
    in f world;;
```

[ ]:
```
let t1 = Term (Atom "etudiant_de", [Var (Id ("E", 0)); Term (Atom "pierre",␣
 ↪[])]);;
let t2 = Term (Atom "etudiant_de", [Var (Id ("S", 0)); Var (Id ("P", 0))]);;

sld world [t1] [] 1;;
```

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