

INF6120

Programmation fonctionnelle et logique

Chapitre 10 - Interprétation d'un langage logique

Quentin Stiévenart

Été 2024

Termes :

- variables : `Foo`, `_`, ...
- constantes (atomes) : `foo`, `1`, ...
- structures : `foo(A, 1)`

Clause : conjonction de structures

`connected(X, Z, L), connected(Z, Y, L)`

Éléments syntaxiques en Prolog

Règles : une structure et une clause :

```
nearby(X, Y) :-  
    connected(X, Z, L),  
    connected(Z, Y, L).
```

Faits : une règle avec une clause vide

```
connected(bond_street, oxford_circus, central) :-  
    .
```

Termes Prolog en OCaml

On souhaitera distinguer différentes variables avec le même nom lors de l'évaluation, donc on leur attache un compteur.

```
type variable = string * int
```

```
type constant = string
```

```
type term =  
  | Variable of variable  
  | Constant of constant  
  | Structure of structure  
and structure = constant * term list
```

Fonctions utiles sur les termes

Une fonction de conversion vers string est toujours utile pour déboguer :

```
let rec term_to_string (term : term) : string =  
  match term with  
  | Variable (v, n) -> Printf.sprintf "%s%d" v n  
  | Constant c -> c  
  | Structure structure -> structure_to_string structure  
and structure_to_string ((name, terms) : structure) : string =  
  Printf.sprintf "%s(%s)"  
    name (String.concat "," (List.map term_to_string terms))
```

Clause : conjonction de structures

```
connected(X, Z, L), connected(Z, Y, L)
```

Clause : conjonction de structures

`connected(X, Z, L), connected(Z, Y, L)`

En OCaml :

```
type clause = structure list
```

```
let clause_to_string (clause : clause) : string =  
    String.concat "," (List.map structure_to_string clause)
```

Règles Prolog en OCaml

Règles : une structure et une clause :

```
nearby(X, Y) :-  
    connected(X, Z, L),  
    connected(Z, Y, L).
```

Règles Prolog en OCaml

Règles : une structure et une clause :

```
nearby(X, Y) :-  
    connected(X, Z, L),  
    connected(Z, Y, L).
```

En OCaml :

```
type rule = structure * clause
```

```
let rule_to_string ((structure, clause) : rule) : string =  
    Printf.sprintf "%s :- %s"  
        (structure_to_string structure) (clause_to_string clause)
```

Programme d'exemple

Soit le programme Prolog suivant :

```
member(X, cons(X, _)).  
member(X, cons(_, L)) :- member(X, L).
```

et la requête :

```
?- member(X, cons(a, cons(b, nil))).
```

Programme d'exemple

Quelques utilitaires pour faciliter la transcription :

```
let x = Variable ("X", 0) in
let any = Variable ("_", 0) in
let l = Variable ("L", 0) in
let member a b = ("member", [a; b]) in
let cons a b = Structure ("cons", [a; b]) in
```

Programme d'exemple

```
let program =  
  (member x (cons x l), []) ::  
  (member x (cons any l), [  
    (member x l)  
  ]) :: []
```

```
let query =  
  let a = Constant "a" in  
  let b = Constant "b" in  
  let nil = Constant "nil" in  
  [member x (cons a (cons b nil))]
```

Rappel :

- on a vu l'unification dans le chapitre sur l'inférence de type
- l'unification se base sur la notion de *substitution*

Exemple en Prolog :

```
?- f(X, Y) = f(Z, g(3)).
```

```
X = Z,
```

```
Y = g(3).
```

Substitution : rappel

Mathématiquement, on note $t[x \mapsto y]$ pour dire que x est substitué par y dans le terme t (un type par exemple).

Exemple :

$$(t1 \rightarrow t2)[t1 \mapsto \text{int}] = \text{int} \rightarrow t2$$

Substitution : représentation

On souhaite substituer des variables par des termes :

```
type subst = (variable * term) list

let subst_to_string (subst : subst) : string =
  let elems = List.map (fun ((v, n), t) ->
    Printf.sprintf "%s%d -> %s" v n (term_to_string t)) subst in
  "[" ^ (String.concat ", " elems) ^ "]"
```

Substitutions : substituer une variable

```
let subst_var (subst : subst) (v : variable) : term =  
  match List.assoc_opt v subst with  
  | Some term -> term  
  | None -> Variable v
```

Substitutions : appliquer une substitution

Une différence avec les substitutions du chapitre 9 :

- supposons que $\sigma = [X \mapsto \text{foo}(Y), Y \mapsto 1]$
- si on applique σ au terme X , on souhaite obtenir $\text{foo}(1)$
- il faut donc continuer la substitution jusqu'à ce qu'on arrive à un point fixe
 - point fixe : c'est quand $f(x) = x$, c'est-à-dire ici que la substitution ne change plus rien

Par exemple :

?- $f(g(Y), 1) = f(X, Y).$

$Y = 1,$

$X = g(1).$

Substitutions : appliquer une substitution

```
let rec apply_subst (subst : subst) (t : term) =  
  match t with  
  | Variable v ->  
    let t' = subst_var subst v in  
    if t = t' then  
      t  
    else  
      apply_subst subst t'  
  | Constant c ->  
    Constant c  
  | Structure (constant, terms) ->  
    Structure (constant, List.map (apply_subst subst) terms)
```

Unification : rappel

L'algorithme utilisé dans le chapitre 9 était le suivant :

Une approche est la suivante : on traite les contraintes une à la fois, et on construit une substitution. Pour chaque contrainte $t_1 = t_2$:

- *si on a $\text{int} = \text{int}$ ou $\text{bool} = \text{bool}$, on passe à la suite*
- *si on a $t_1 = \text{TFresh } x$, avec $\text{TFresh } x$ qui n'apparaît pas dans t_2 :*
 - *on ajoute $t_1 = \text{Fresh } x$ à la substitution et on continue*
- *similaire si $t_2 = \text{TFresh } x$ avec $\text{TFresh } x$ qui n'apparaît pas dans t_1*
- *si on a $a \rightarrow b = c \rightarrow d$:*
 - *on sépare la contrainte en deux : $a = c$ et $b = d$, et on continue*
- *sinon, les termes ne sont pas unifiables*

C'est l'algorithme de Martelli et Montanari (1982)

L'algorithme de Martelli et Montanari (1982) est linéaire.

Basé sur une analyse de cas, avec plusieurs règles :

- règle de suppression
- règle d'élimination
- règle d'échange
- règle de décomposition

Implémentation de l'unification

```
exception NoUnification
```

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  ...
```

Règle de suppression : si deux termes sont syntaxiquement égaux, ils s'unifient

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  | t1, t2 when t1 = t2 ->  
    (* Règle de suppression *)  
    subst (* déjà unifié par subst *)
```

On utilise =, l'équivalence syntaxique.

Algorithme de Martelli et Montanari

Règle d'élimination : si on unifie une variable $\text{var}(x)$ avec un autre terme t , on retient la substitution $\text{var}(x) \rightarrow t$

Règle d'échange : si on unifie quelque chose avec une variable, on échange nos paramètres pour revenir à la règle d'élimination ou de suppression.

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  ...  
  | (Variable x, t) | (t, Variable x) ->  
    (* Règle d'élimination (et règle d'échange) *)  
    (* on a  $x = t$  ou  $t = x$ , on l'ajoute à nos substitutions *)  
    (x, t) :: subst
```

Règle de décomposition : si on unifie deux structures de même nom et arité, il faut unifier chacun de leurs éléments

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  ...  
  | Structure (clause1, terms1), Structure (clause2, terms2)  
  when clause1 = clause2 ->  
    (* Règle de décomposition *)  
    unify_list subst terms1 terms2
```

Algorithme de Martelli et Montanari

`unify_list` applique `unify` à chaque élément d'une liste et propage les substitutions.

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  ...  
and unify_list (subst : subst) (terms1 : term list) (terms2 : term list)  
  : subst =  
  if List.length terms1 = List.length terms2 then  
    List.fold_left2 unify subst terms1 terms2  
  else  
    raise NoUnification
```

List.fold_left2 applique un repli à deux listes en même temps :

```
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
```

Par exemple :

```
# List.fold_left2 (fun acc x y -> (x, y) :: acc) [] [1;2;3] [4;5;6];;  
- : (int * int) list = [(3, 6); (2, 5); (1, 4)]
```

Unification : exemple

?- f(X, Y) = f(Z, g(3)).

X = Z,

Y = g(3).

```
# let term1 = Structure ("f", [Variable ("X", 0); Variable ("Y", 0)]);;
```

```
# let term2 = Structure ("f", [Variable ("Z", 0);  
                               Structure ("g", [Constant "3"])]);;
```

```
# subst_to_string (unify [] term1 term2);;
```

```
- : string = "[Y0 -> g(3), X0 -> Z0]"
```

On peut se retrouver avec des termes cycliques :

?- $X = f(X).$

$X = f(X).$

Une telle méthode d'unification rend Prolog **incorrect**¹, mais efficace.

¹On peut prouver des théorèmes qui sont faux, tels que $(\forall x, \exists y.p(x, y)) \rightarrow (\exists y, \forall x.p(x, y))$

Notre implémentation se comporte de la même façon :

```
# let term1 = Variable ("X", 0);;  
# let term2 = Structure ("f", [Variable ("X", 0)]);;  
# subst_to_string (unify [] term1 term2);;  
- : constant = "[X0 -> f(X0)]"
```

Éviter la cyclicité avec l'occurs-check

Pour éviter cela et avoir un langage logique entièrement correct, il faut s'assurer qu'unifier une variable X avec une structure S ne réussisse que si S ne contient pas X .

`unify_with_occurs_check/2` est fourni par Prolog, mais pas utilisé de base.

```
?- unify_with_occurs_check(X, f(X)).  
false.
```

Éviter la cyclicité avec l'occurs-check

On peut ajouter l'occurs-check à notre implémentation :

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  ...  
  | (Variable x, t) | (t, Variable x) ->  
    (* Règle d'élimination (et règle d'échange) *)  
    if appears_in x t then  
      (* Occurs-check *)  
      raise NoUnification  
    else  
      (* on a  $x = t$  ou  $t = x$ , on l'ajoute à nos substitution *)  
      (x, t) :: subst
```

Éviter la cyclicité avec l'occurs-check

```
# let term1 = Variable ("X", 0);;  
# let term2 = Structure ("f", [Variable ("X", 0)]);;  
# subst_to_string (unify [] term1 term2);;  
Exception: NoUnification.
```

Unification : au complet

```
let rec unify (subst : subst) (term1 : term) (term2 : term) : subst =  
  match apply_subst subst term1, apply_subst subst term2 with  
  | t1, t2 when t1 = t2 ->  
    subst  
  | (Variable x, t) | (t, Variable x) ->  
    if appears_in x t then  
      raise NoUnification  
    else  
      (x, t) :: subst  
  | Structure (clause1, terms1), Structure (clause2, terms2) when clause1 = clause2 ->  
    unify_list subst terms1 terms2  
  | _ -> raise NoUnification  
and unify_list (subst : subst) (terms1 : term list) (terms2 : term list) : subst =  
  if List.length terms1 = List.length terms2 then  
    List.fold_left2 unify subst terms1 terms2  
  else  
    raise NoUnification
```

Résolution : rappel

Avec le programme suivant :

```
member(X, cons(X, _)).  
member(X, cons(Y, L)) :- member(X, L).
```

On souhaite résoudre la requête :

```
?- member(X, cons(a, cons(b, nil)))
```

Solutions attendues : X=a et X=b.

Résolution : rappel

```
member(X, cons(X, _)).           % règle 1  
member(X, cons(_, L)) :- member(X, L). % règle 2
```

On a deux choix : utiliser la règle 1 ou la règle 2.

Résolution : rappel

Utilisation de la règle 1 :

```
member(X, cons(X, _)).           % règle 1
```

1. On unifie notre requête et la tête de la règle :

```
?- member(X, cons(X, _)) = member(X, cons(a, cons(b, nil))).  
X = a.
```

2. On applique la substitution $X = a$ au corps de la règle. Il est vide, on ne fait donc rien.
3. On remplace notre requête par le corps de la règle substitué. On a donc fini cette résolution, avec comme solution $X = a$.

Résolution : rappel

Utilisation de la règle 2 :

```
member(X, cons(Y, L)) :- member(X, L). % règle 2
```

1. On unifie notre requête avec la tête de la règle :

```
?- member(X, cons(_, L)) = member(X, cons(a, cons(b, nil))).  
L = cons(b, nil).
```

2. On applique la substitution au corps de la règle, on obtient un nouveau but :

```
?- member(X, cons(b, nil))
```

3. On remplace notre requête par le corps de la règle substitué. On continue à partir de là.

Résolution : rappel

On doit maintenant résoudre :

```
?- member(X, cons(b, nil))
```

Utilisation de la règle 1 :

```
member(X, cons(X, _)).           % règle 1
```

1. On unifie notre requête et la tête de la règle :

```
?- member(X, cons(X, _)) = member(X, cons(b, nil)).
```

```
X = b.
```

2. On applique la substitution $X = a$ au corps de la règle. Il est vide, on ne fait donc rien.
3. On remplace notre requête par le corps de la règle substitué. On a donc fini cette résolution, avec $X = b$.

La procédure générale de résolution est donc :

- on prends le but à résoudre
- on l'unifie avec une règle de la base de donnée
 - il y a un choix à effectuer, plusieurs règles peuvent s'appliquer
 - Prolog commence par la première règle
- on applique l'unification entre la requête et la tête de la règle sélectionnée
- on remplace le but par le corps de la règle, après substitution
- on continue

Résolution : représentation des choix

Il faut donc maintenir:

- les buts à résoudre
- une version du programme où on a éliminé les règles utilisées
- la substitution courante
- la profondeur courante

On va représenter cela comme un *choix* :

```
type choice = {  
  goals : goal list;  
  rules : rule list;  
  subst : subst;  
  depth : int;  
}
```

Résolution : utilité de numéroté les variables

```
?- member(X, cons(a, cons(b, cons(c, nil))))
```

Dessiner l'arbre de résolution.

On se retrouve avec une branche où $Y = a$, $L = \text{cons}(b, \text{cons}(c, \text{nil}))$ ayant une sous-branche où $Y' = b$, $L' = \text{cons}(c, \text{nil})$.

Y et Y' sont des variables différentes !

Renumérotation des variables

À chaque *profondeur* de résolution, on va renuméroter les variables

```
let rec renumber (n : int) (term : term) : term =  
  match term with  
  | Variable (x, _) ->  
    Variable (x, n)  
  | Constant _ ->  
    term  
  | Structure (constant, terms) ->  
    Structure (constant, List.map (renumber n) terms)  
  
let renumber_structure (n : int) ((name, terms) : structure)  
  : structure =  
  (name, List.map (renumber n) terms)
```

Une solution est une substitution.

```
?- member(X, cons(a, cons(b, cons(c, nil))))
```

```
X = a ;
```

```
X = b ;
```

```
X = c.
```

Trois solutions, trois substitutions.

Résolution d'un but

À partir de :

- un but à résoudre
- une liste de règles (version modifiée du programme)
- une substitution courante
- une profondeur

on peut résoudre le but en trouvant la première règle qui unifie avec le but

On retourne alors :

- les sous-buts à résoudre
- la substitution qui résulte de l'unification
- les règles restantes qui n'ont pas été essayées

Résolution d'un but

```
let rec solve_goal (goal : goal) (rules : rule list)
                  (subst : subst) (depth : int)
  : (goal list * subst * rule list ) option =
match rules with
| [] -> None
| (goal', clauses) :: rest ->
  try
    let subst' =
      unify subst (Structure goal)
        (Structure (renumber_structure depth goal')) in
    Some (List.map (renumber_structure depth) clauses, subst', rest)
  with NoUnification ->
    solve_goal goal rest subst depth
```

Résolution d'une requête complète

```
let solve (query : clause) (database : rule list) =  
  ...
```

- query est la requête de l'utilisateur, par exemple `member(X, cons(a, cons(b, nil)))`
- database est le programme, par exemple :

```
member(X, cons(X, _)) :- .  
member(X, cons(_, L)) :- member(X, L).
```

Résolution : point de départ

On démarre avec la requête initiale :

```
let solve (query : clause) (database : rule list) =  
  ...  
  let initial_choice = {  
    goals = query;  
    rules = database;  
    depth = 1;  
    subst = [];  
  } in  
  solve_aux initial_choice []
```

Résolution : boucle principale

Pour un choix à explorer, si la requête est vide :

- on a trouvé une solution, la substitution courante
- on cherche récursivement le reste des solutions

```
let rec continue (choices : choice list) : subst list =  
  match choices with  
  | [] -> []  
  | choice :: rest -> solve_aux choice rest  
and solve_aux (choice : choice) (choices : choice list) : subst list =  
  match choice.goals with  
  | [] -> [choice.subst] @ continue choices  
  ...
```

Si la requête n'est pas vide, on essaye de résoudre le premier but avec `solve_goal`

- si cela échoue, on continue la recherche avec les autres choix
- sinon, on a des sous-buts à résoudre et on continue avec une profondeur incrémentée

Résolution : boucle principale

```
and solve_aux (choice : choice) (choices : choice list) : subst list =  
  match choice.goals with  
  | [] -> ...  
  | goal :: rest_goals ->  
    begin match solve_goal goal choice.rules choice.subst choice.depth with  
    | None -> continue choices  
    | Some (subgoals, subst', rules') ->  
      let choices' = { choice with rules = rules' } :: choices in  
      let choice' = { rules = database;  
                      subst = subst';  
                      goals = subgoals @ rest_goals;  
                      depth = choice.depth + 1 } in  
      solve_aux choice' choices'  
    end  
end
```

Example

```
let program =  
  (member x (cons x l), []) ::  
  (member x (cons any l), [  
    (member x l)  
  ])) :: []  
let query =  
  let a = Constant "a" in  
  let b = Constant "b" in  
  let nil = Constant "nil" in  
  [member x (cons a (cons b nil))]
```

Example

```
# List.iter (fun subst -> Printf.printf "%s\n" (subst_to_string subst))  
  (solve query program)  
[L1 -> cons(b,nil), X1 -> a, X0 -> X1]  
[L2 -> nil, X2 -> b, X1 -> X2, L1 -> cons(b,nil), _1 -> a, X0 -> X1]
```

Restriction des solutions

Du point de vue utilisateur, on ne s'intéresse pas aux variables intermédiaires dans la solution

On ne va garder que les variables numérotées par 0 :

```
let to_user_solution (subst : subst) : subst =  
  List.filter_map (fun (v, t) ->  
    if snd v = 0 then  
      Some (v, apply_subst subst t)  
    else  
      None) subst
```

On adapte notre fonction solve

```
let solutions = solve_aux initial_choice [] in  
List.map to_user_solution solutions
```

Exemple

```
# List.iter (fun subst -> Printf.printf "%s\n" (subst_to_string subst))  
  (solve query program)  
[X0 -> a]  
[X0 -> b]
```

Concepts importants

- Un programme Prolog peut être vu comme une **base de donnée de règles** (et faits)
- La résolution utilise **l'unification** pour dériver des **substitutions**
- L'unification est très similaire à celle utilisée en typage
- Une solution est une **substitution**
- Le **retour en arrière** (*backtracking*) est implémenté au sein de l'interpréteur Prolog, ici de manière récursive