Programmation Logique

Samuel Gallay

10 juin 2021

1 Introduction

Pendant mon TIPE, je me suis intéressé à la programmation logique, et plus particulièrement au langage Prolog. Initialement je souhaitais comprendre comment fonctionnaient les langages de programmation, et j'ai été attiré par l'apparence un peu atypique du langage Prolog. J'ai décidé d'écrire en OCaml un interpréteur pour ce langage.

J'ai fourni à la fin de ce rapport l'intégralité du code OCaml de mon interpréteur. Puisque relire le code d'autrui est rarement une chose amusante, une version en ligne de cet interpréteur est accessible à cette adresse : https://samuelgallay.github.io/prolog/. Elle permet de l'essayer dans un navigateur. J'en discuterai un peu par la suite.

Ce document suppose que le lecteur a déjà entendu parler du langage Prolog, ou au moins que ce dernier a parcouru dans les grandes lignes la page Wikipédia consacrée au Prolog.

2 La grammaire du langage

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

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

Ci-dessus est présenté un exemple de programme Prolog. Les briques élémentaires d'un programme sont les *termes* : `eve`, `apprend(eve, mathématiques)` et `E` sont des termes. Une *clause* se termine par un point, et contient soit un unique terme, soit un terme et une liste de termes.

J'ai défini la grammaire du sous-ensemble du langage Prolog que je souhaitais interpréter. J'ai finalement choisi cette forme, qui ajoute des listes à l'exemple précédent, ainsi que des variables sans nom '_'.

```
<Caractère>    ::=  'a'..'z' | 'A'..'Z' | '_' | '0'..'9'
<Mot>          ::=  <Caractère> | <Caractère> <Mot>
<Prédicat>     ::=  'a'..'z' | 'a'..'z' <Mot>
<Variable>     ::=  'A'..'Z' | 'A'..'Z' <Mot> | '_'
<Programme>    ::=  <Clause> | <Clause> <Programme>
<Clause>       ::=  <Terme> '.' | <Terme> ':-' <ListeTermes> '.'
<ListeTermes>  ::=  <Terme> | <Terme> ',' <ListeTermes>
<Terme>        ::=  <Variable> | <Predicat> | <Predicat> '(' <ListeTermes> ')' | <ListeProlog>
<ListeProlog>  ::=  '[]' | '[' <ListeTermes> ']' | '[' <ListeTermes> '|' <ListeProlog> ']'
                    | '[' <ListeTermes> '|' <Variable> ']'
```

3 Représentation d'un programme Prolog

C'est une partie de mon travail qui m'a demandé plusieurs essais infructueux. Il est assez naturel de représenter un terme par un type somme récursif. Le souci est que de nombreuses fonctions dans le code n'ont de sens que si les termes qu'elles prennent en argument possèdent des propriétés particulières : certaines fonctions n'accèptent que des variables, d'autres que des listes, etc... En plus une variable peut représenter une liste, ce qui complique les choses.

L'utilisation des *variants polymorphes* d'OCaml, en particulier avec les mécanismes de *coercion* rendent le code plus clair. Ces variants permettent de forcer les queues des listes Prolog à être elles-même des listes. Pour préserver cet invariant, je distingue deux types de variables : celles qui peuvent représenter n'importe quel terme et celles qui ne peuvent représenter que des listes.

```
type id = Id of string * int

type term =
  [ `GeneralVar of id
  | `ListVar of id
  | `EmptyList
  | `List of term * prolog_list
  | `Predicate of string * term list ]

and prolog_list = [ `ListVar of id | `EmptyList | `List of term * prolog_list ]

type var = [ `GeneralVar of id | `ListVar of id ]

type clause = Clause of term * term list
```

4 L'analyse syntaxique

L'analyse syntaxique est la transformation d'une chaîne de caractères représentant un programme en un arbre de syntaxe abstraite. Il existe plusieurs bibliothèques en OCaml, par exemple Menhir, qui permettent d'écrire des grammaires dans une forme très proche de celle de Backus-Naur et qui génèrent le code effectuant cette transformation.

J'ai quand même décidé d'écrire moi-même un analyseur syntaxique récursif descendant. Ma première source était Ljunglöf (2002). Je me suis rendu compte, en testant sur de longs programmes, que ma première implémentation était peu efficace. Quand plusieurs règles de production étaient disponnibles, j'effectuais un backtracking : la complexité était vraiment catastrophique. J'ai ensuite réalisé, en factorisant ma grammaire à gauche (donc en ajoutant des symboles non-terminaux), que celle-ci était $LL(1)$. J'ai pu écrire un parser sans backtracking en observant un symbole à l'avance à chaque fois. Pour simplifier un petit peu, j'ai d'abord écrit un lexer qui reconnait les symboles élémentaires.

Cette dernière implémentation est moins élégante, mais très rapide. Je peux analyser des fichiers d'un Mo en environ une seconde à environ 1GHz.

Enfin, puisque je différencie deux types de variables, je dois rechercher toutes les variables qui sont la queue d'une liste, et indiquer quelles sont des listes dans toutes leurs occurences.

Au final le code est peu lisible. J'ai utilisé de nombreux opérateurs infix, et j'ai encapsulé mes résultats dans un type `result`[1] pour transmettre des messages d'erreur.

5 L'unification

Les deux algorithmes à la base de Prolog sont l'unification et le principe de résolution, qui ont été introduits par Robinson (1965). Je ne vais pas rappeler son fonctionnement, mais l'unification prend en paramètres deux termes t_1 et

1. `type ('a, 'e) result = Ok of 'a | Error of 'e`

t_2 ne possédant pas de variables en commun : ceci justifie l'utilisation d'un entier dans mes identifiants, pour permettre un renommage facile. L'algorithme renvoie si elle existe la substitution σ (fonction des variables dans les termes) la plus générale telle que $\sigma(t_1) = \sigma(t_2)$.

L'algorithme d'unification complet est présenté dans Nilsson et Małuszyński (1990). Je l'ai adapté à ma représentation des termes, et je l'ai transformé en un algorithme récursif.

J'utilise en fait le même algorithme que PROLOG qui est une simplification de celui présenté dans Nilsson et Małuszyński (1990) : celui de PROLOG est plus rapide mais peut ne pas terminer. Par exemple en essayant d'unifier `X` et `f(X)`, l'algorithme, à la place d'échouer, va essayer de renvoyer `f(f(f(...)))`.

6 La recherche de solutions

Je ne vais pas non plus redécrire le principe de résolution de PROLOG. La résolution se ramène essentiellement à la construction et au parcours d'un arbre dont les noeuds portent des requêtes à résoudre, et les feuilles sont des substitutions solutions de la requête initiale. Un noeud possède plusieurs fils quand plusieurs clauses peuvent être utilisées pour résoudre la requête qu'il porte.

Puisque l'exécution d'un programme PROLOG peut ne pas terminer, l'arbre peut être infini. On ne peut donc pas le stocker entièrement en mémoire. C'est pour cela que j'utilise l'évaluation paresseuse, qui permet d'évaluer l'arbre à la demande :

```
type 'a tree = Leaf of 'a | Node of 'a tree Lazy.t list
```

Je trouve plus aisé d'écrire différents types de parcours sur un tel arbre que de modifier une grande fonction récursive. PROLOG utilise un parcours en profondeur de cet arbre. C'est ce que je fais avec la fonction `to_seq` de mon code qui transforme l'arbre en une séquence (liste possiblement infinie) de solutions qu'il suffit d'itérer.

7 Extensions

Je me suis intéressé à différents moyens d'augmenter cet interpréteur. Une variation intéressante de l'exécution est de procéder à un parcours en largeur de l'arbre, ce qui permet d'accéder en un temps fini à chacune des solutions.

Un point qui m'a intéressé est la mémoïsation des programmes PROLOG. On peut enregistrer les solutions des requêtes déjà traitées pour accélérer l'exécution. Ceci a été fait dans Warren (1992). On peut ainsi éviter des boucles infinies où l'on répèterait la même requête.

Il faut savoir que des compilateurs pour le langange ont déjà été écrits. Le premier a été celui de David H. D. Warren [2].

Pour créer une interface à mon interpréteur, et pouvoir le partager, j'ai compilé mon code OCAML en JAVASCRIPT [3]. Notons que l'on perd un facteur 10 dans la vitesse d'exécution par rapport à du code natif. C'est aussi pour simplifier cette compilation que j'ai réécrit un parser sans utiliser une bibliothèque existante. J'utilise quelques expressions rationnelles pour la coloration syntaxique de l'éditeur. Le programme sur la page de garde est une solution du célèbre *puzzle du zèbre* [4] !

Références

Peter LJUNGLÖF : Pure functional parsing, an advanced tutorial. 2002.

Ulf NILSSON et Jan MAŁUSZYŃSKI : *Logic, Programming and Prolog.* 1990.

John Alan ROBINSON : A machine-oriented logic based on the resolution principle. 1965.

David S. WARREN : Memoing for logic programs. 1992.

2. Ce ne sont pas les mêmes David Warren
3. J'ai utilisé MELANGE, un fork de BUCKLESCRIPT après que ce dernier se soit éloigné d'OCAML...
4. https://en.wikipedia.org/wiki/Zebra_Puzzle

Intégralité du code OCaml

```
(* ***********************************
            Type Declarations
   *********************************** *)

type id = Id of string * int

type term =
  [ `GeneralVar of id
  | `ListVar of id
  | `EmptyList
  | `List of term * prolog_list
  | `Predicate of string * term list ]

and prolog_list = [ `ListVar of id | `EmptyList | `List of term * prolog_list ]

type var = [ `GeneralVar of id | `ListVar of id ]

type clause = Clause of term * term list

type subst =
  | GeneralSubst of [ `GeneralVar of id ] * term
  | ListSubst of [ `ListVar of id ] * prolog_list

type substitution = subst list

type token =
  | PredicateString of string
  | VariableString of string
  | LeftParenthesis
  | RightParenthesis
  | LeftBracket
  | RightBracket
  | Vertical
  | Comma
  | Period
  | If
  | EndOfFile

(* ***********************************
            Pretty Printing Functions
   *********************************** *)

let rec string_of_term : term -> string = function
  | `GeneralVar (Id (s, _)) -> if s.[0] = '_' then "_" else s
  | `ListVar (Id (s, _)) -> s
  | (`EmptyList | `List _) as t -> "[" ^ string_of_listcontent t ^ "]"
  | `Predicate (s, []) -> s
  | `Predicate (s, l) -> s ^ "(" ^ (l |> List.map string_of_term |> String.concat ", ") ^ ")"

and string_of_listcontent : [ `EmptyList | `List of term * prolog_list ] -> string = function
  | `EmptyList -> ""
  | `List (t, tail) -> (
```

```
      match tail with
      | `EmptyList -> string_of_term t
      | `List _ as lst -> string_of_term t ^ ", " ^ string_of_listcontent lst
      | `ListVar (Id (s, _)) -> string_of_term t ^ " | " ^ s)

let string_of_clause (Clause (t, tl)) =
  string_of_term t
  ^ (if tl = [] then ""
    else " :-" ^ (tl |> List.map (fun t -> "\n  " ^ string_of_term t) |> String.concat ","))
  ^ "."

let string_of_program cl = cl |> List.map string_of_clause |> String.concat "\n\n"

let string_of_token = function
  | PredicateString s -> s
  | VariableString s -> s
  | LeftParenthesis -> "("
  | RightParenthesis -> ")"
  | LeftBracket -> "["
  | RightBracket -> "]"
  | Vertical -> "|"
  | Comma -> ","
  | Period -> "."
  | If -> ":-"
  | EndOfFile -> "EOF"

let string_of_tokenlist tl = tl |> List.map string_of_token |> String.concat "; "

(* ***********************************
            Tokenizer
   *********************************** *)

exception TokenNotFound of char

let rec parse_word = function
  | (('a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_') as c) :: t ->
      let w, t' = parse_word t in
      (c :: w, t')
  | l -> ([], l)

let string_of_charlist cl = String.of_seq (List.to_seq cl)

let rec tokenlist_of_charlist : token list -> char list -> token list =
 fun acc -> function
  | (' ' | '\t' | '\n') :: t -> tokenlist_of_charlist acc t
  | '[' :: t -> tokenlist_of_charlist (LeftBracket :: acc) t
  | ']' :: t -> tokenlist_of_charlist (RightBracket :: acc) t
  | '(' :: t -> tokenlist_of_charlist (LeftParenthesis :: acc) t
  | ')' :: t -> tokenlist_of_charlist (RightParenthesis :: acc) t
  | '|' :: t -> tokenlist_of_charlist (Vertical :: acc) t
  | ',' :: t -> tokenlist_of_charlist (Comma :: acc) t
  | '.' :: t -> tokenlist_of_charlist (Period :: acc) t
  | ':' :: '-' :: t -> tokenlist_of_charlist (If :: acc) t
  | '_' :: t -> tokenlist_of_charlist (VariableString "_" :: acc) t
```

```
  | [] -> List.rev (EndOfFile :: acc)
  | ('a' .. 'z' as low) :: t ->
      let w, cl = parse_word t in
      tokenlist_of_charlist (PredicateString (string_of_charlist (low :: w)) :: acc) cl
  | ('A' .. 'Z' as up) :: t ->
      let w, cl = parse_word t in
      tokenlist_of_charlist (VariableString (string_of_charlist (up :: w)) :: acc) cl
  | tok :: _ -> raise (TokenNotFound tok)

let charlist_of_string s = List.init (String.length s) (String.get s)

let tokenlist_of_string s =
  try Ok (s |> charlist_of_string |> tokenlist_of_charlist [])
  with TokenNotFound tok -> Error ("this token is unknown : '" ^ String.make 1 tok ^ "'")

(* ***********************************
            Parsing
   *********************************** *)

type 'a parser = token list -> ('a * token list, string) result

let ( >>= ) = Result.bind

let ( *~ ) : 'a parser -> 'b parser -> ('a * 'b) parser =
 fun p q cl ->
  p cl >>= fun (a, cl') ->
  q cl' >>= fun (b, cl'') -> Ok ((a, b), cl'')

let ( *> ) : 'a parser -> ('a -> 'b) -> 'b parser =
 fun p f cl -> p cl >>= fun (a, cl') -> Ok (f a, cl')

let ignore_left (_, b) = b

let ignore_right (a, _) = a

let epsilon tl = Ok ((), tl)

let parse_token tk = function
  | t :: l when t = tk -> Ok (t, l)
  | _ -> Error ("Unable to parse token : " ^ string_of_token tk)

let look_ahead tl f =
  match tl with [] -> Error "Unexpected end of token stream." | tok :: toklist -> f (tok, toklist)

let rec prolog_list_of_termlist : prolog_list -> term list -> prolog_list =
 fun tail -> function [] -> tail | h :: t -> `List (h, prolog_list_of_termlist tail t)

let new_id =
  let counter_unnamed_ids = ref 0 in
  fun () ->
    counter_unnamed_ids := !counter_unnamed_ids + 1;
    Id ("_" ^ string_of_int !counter_unnamed_ids, 0)

let rec parse_term tl =
```

```
  look_ahead tl (function
    | VariableString s, l -> Ok (`GeneralVar (if s = "_" then new_id () else Id (s, 0)), l)
    | PredicateString s, l ->
        l |> parse_predicate_without_string *> fun term_list -> `Predicate (s, term_list)
    | LeftBracket, l -> l |> parse_prolog_list_without_left_bracket *> fun lst -> (lst :> term)
    | tok, _ -> Error ("Not a valid term : " ^ string_of_token tok))

and parse_predicate_without_string tl =
  look_ahead tl (function
    | LeftParenthesis, l -> l |> parse_term_list *~ parse_token RightParenthesis *> ignore_right
    | _, _ -> Ok ([], tl))

and parse_term_list tl =
  tl |> parse_term *~ parse_term_list_without_first_term *> fun (t, tl') -> t :: tl'

and parse_term_list_without_first_term tl =
  look_ahead tl (function
    | Comma, l -> l |> parse_term *~ parse_term_list_without_first_term *> fun (a, b) -> a :: b
    | _, _ -> Ok ([], tl))

and parse_prolog_list_without_left_bracket tl =
  look_ahead tl (function
    | RightBracket, l -> Ok (`EmptyList, l)
    | _, _ ->
        tl
        |> parse_term_list *~ parse_prolog_list_without_term_list *> fun (termlist, plist) ->
           prolog_list_of_termlist plist termlist)

and parse_prolog_list_without_term_list tl =
  look_ahead tl (function
    | RightBracket, l -> Ok (`EmptyList, l)
    | Vertical, l -> l |> parse_prolog_list_queue
    | tok, _ -> Error ("Invalid right part of the list : " ^ string_of_token tok))

and parse_prolog_list_queue tl =
  look_ahead tl (function
    | LeftBracket, l ->
        l |> parse_prolog_list_without_left_bracket *~ parse_token RightBracket *> ignore_right
    | VariableString s, l -> l |> parse_token RightBracket *> fun _ -> `ListVar (Id (s, 0))
    | _ -> Error "Invalid list queue")

let parse_right_clause tl =
  look_ahead tl (function
    | Period, l -> Ok ([], l)
    | _, _ ->
        tl |> parse_token If *~ parse_term_list *> ignore_left *~ parse_token Period *> ignore_right)

let parse_clause tl = tl |> parse_term *~ parse_right_clause *> fun (t, rc) -> Clause (t, rc)

let rec parse_program tl =
  look_ahead tl (function
    | EndOfFile, _ -> Ok ([], [])
    | _, _ -> tl |> parse_clause *~ parse_program *> fun (a, b) -> a :: b)
```

```
let parse_request tl = tl |> parse_term_list *~ parse_token EndOfFile *> ignore_right

(* ***********************************
            Substitution Functions
   *********************************** *)

let map_vars f g =
  let rec mapterm : term -> term = function
    | #prolog_list as lst -> (map_prolog_list lst :> term)
    | `GeneralVar _ as v -> f v
    | `Predicate (s, l) -> `Predicate (s, List.map mapterm l)
  and map_prolog_list : prolog_list -> prolog_list = function
    | `EmptyList -> `EmptyList
    | `ListVar _ as v -> g v
    | `List (h, t) -> `List (mapterm h, map_prolog_list t)
  in
  mapterm

let apply_subst : subst -> term -> term =
  let id x = x in
  function
  | GeneralSubst (var, t) -> map_vars (function v -> if var = v then t else (v :> term)) id
  | ListSubst (var, t) -> map_vars id (function v -> if var = v then t else (v :> prolog_list))

(* De droite à gauche *)
let apply (s : substitution) term = List.fold_right apply_subst s term

(* ***********************************
            Unification Algorithm
   *********************************** *)

let rec unify : term -> term -> substitution option =
 fun t1 t2 ->
  let f ta tb s1 = Option.bind (unify (apply s1 ta) (apply s1 tb)) (fun s2 -> Some (s2 @ s1)) in
  match (t1, t2) with
  | (`ListVar _ as v), ((`ListVar _ | `EmptyList | `List _) as t) -> Some [ ListSubst (v, t) ]
  | (`EmptyList | `List _), `ListVar _ -> unify t2 t1
  | `EmptyList, `EmptyList -> Some []
  | `List (h1, t1), `List (h2, t2) -> Option.bind (unify h1 h2) (f (t1 :> term) (t2 :> term))
  | `List _, `EmptyList | `EmptyList, `List _ -> None
  | (`GeneralVar _ as v), _ -> Some [ GeneralSubst (v, t2) ]
  | _, `GeneralVar _ -> unify t2 t1
  | `Predicate (sa, la), `Predicate (sb, lb) ->
      if sa <> sb then None
      else if List.length la <> List.length lb then None
      else List.fold_left2 (fun opt_s ta tb -> Option.bind opt_s (f ta tb)) (Some []) la lb
  | (`ListVar _ | `EmptyList | `List _), `Predicate _
  | `Predicate _, (`ListVar _ | `EmptyList | `List _) ->
      None

(* ***********************************
            Type Inference
   *********************************** *)
```

```
let rec variables_in_term = function
  | `GeneralVar id -> [ (`GeneralVar id : var) ]
  | `ListVar id -> [ `ListVar id ]
  | `EmptyList -> []
  | `List (h, t) -> variables_in_term h @ variables_in_term (t :> term)
  | `Predicate (_, l) -> l |> List.map variables_in_term |> List.concat

let variables_in_clause (Clause (t, l)) = List.concat_map variables_in_term (t :: l)

let variables_in_request r = List.concat_map variables_in_term r

let replace_tvars_in_term tvars =
  map_vars
    (fun (`GeneralVar id) -> if List.mem (`ListVar id) tvars then `ListVar id else `GeneralVar id)
    (fun x -> x)

let type_inference_clause (Clause (t, l) as c) =
  let f = replace_tvars_in_term (variables_in_clause c) in
  Clause (f t, List.map f l)

let type_inference_request r =
  let f = replace_tvars_in_term (variables_in_request r) in
  List.map f r

let type_inference_program = List.map type_inference_clause

(* ***********************************
            Solver
   *********************************** *)

let rename n (Clause (t, l)) =
  let f =
    map_vars
      (fun (`GeneralVar (Id (s, _))) -> `GeneralVar (Id (s, n)))
      (fun (`ListVar (Id (s, _))) -> `ListVar (Id (s, n)))
  in
  Clause (f t, List.map f l)

type 'a tree = Leaf of 'a | Node of 'a tree Lazy.t list

let rec sld_tree world request substitution n =
  match request with
  | [] -> Leaf substitution
  | head_request_term :: other_request_terms ->
      let filter_clause c =
        let (Clause (left_member, right_member)) = rename n c in
        let new_tree unifier =
          lazy
            (sld_tree world
               (List.map (apply unifier) (right_member @ other_request_terms))
               (unifier @ substitution) (n + 1))
        in
        Option.map new_tree (unify head_request_term left_member)
      in
```

```
      Node (List.filter_map filter_clause world)

let list_to_seq l = List.fold_right (fun x s () -> Seq.Cons (x, s)) l Seq.empty

let rec to_seq = function
  | Leaf str -> Seq.return str
  | Node tl -> Seq.flat_map (fun par -> to_seq (Lazy.force par)) (list_to_seq tl)

let solutions tree vars =
  Seq.map (fun substitution -> (vars, List.map (apply substitution) vars)) (to_seq tree)

(* ***********************************
            Interface Functions
   *********************************** *)

let program_of_string s =
  s |> tokenlist_of_string >>= parse_program >>= fun (p, _) -> Ok (type_inference_program p)

let request_of_string s =
  s |> tokenlist_of_string >>= parse_request >>= fun (r, _) -> Ok (type_inference_request r)

let string_sequence_of parsed_program parsed_request =
  let vars = List.concat_map variables_in_term parsed_request |> List.sort_uniq compare in
  let sol = solutions (sld_tree parsed_program parsed_request [] 1) (vars :> term list) in
  let f (vars_tl, tl) =
    if vars = [] then "This is true."
    else
      "A solution is : "
      ^ (List.map2 (fun v t -> string_of_term v ^ " = " ^ string_of_term t) vars_tl tl
        |> String.concat ", ")
  in
  Seq.map f sol

let basic_interpreter program_string request_string =
  match program_of_string program_string with
  | Error s ->
      print_endline "Unable to parse program :";
      print_endline s
  | Ok parsed_program ->
      (match request_of_string request_string with
      | Error s ->
          print_endline "Unable to parse request :";
          print_endline s
      | Ok parsed_request -> (
          let string_seq = string_sequence_of parsed_program parsed_request in
          match string_seq () with
          | Seq.Nil -> print_endline "No solution found."
          | Seq.Cons (sol, seq) ->
              print_endline sol;
              Seq.iter print_endline seq));
      print_newline ()
```