

Vers la programmation logique

SAMUEL GALLAY

27 février 2021

1 Logique des prédicats

Malheureusement, la logique des prédicats, ou logique du premier ordre, qui constitue le bon cadre pour définir la programmation logique n'est pas au programme de CPGE (bien que la logique des proposition le soit).

On considère un alphabet \mathcal{A} composé :

- de variables (en fait de symboles représentant des variables, mais je les identifie), commençant par une majuscule.
- de prédicats, commençant par une minuscule, d'arité quelconque.
- de fonctions, commençant par une minuscule d'arité quelconque. Une fonction 0-aire est une constante.
- des symboles $\vee, \wedge, \rightarrow, \leftrightarrow, \neg$ représentant la disjonction, la conjonction, l'implication, l'équivalence et la négation.
- des quantificateurs existentiel et universel : \exists, \forall

On définit les termes \mathcal{T} d'un alphabet \mathcal{A} comme étant la clôture inductive des variables de \mathcal{A} par les fonctions de \mathcal{A} .

On définit l'ensemble \mathcal{F} des formules (bien définies) sur (\mathcal{A}) par induction :

- si $p \in \mathcal{A}$ est un prédicat d'arité n , et $t_1, \dots, t_n \in \mathcal{T}$, alors $p(t_1, \dots, t_n) \in \mathcal{F}$
- si $F, G \in \mathcal{F}$ alors $(\neg F), (F \wedge G), (F \vee G), (F \rightarrow G)$ et $(F \leftrightarrow G)$ sont aussi dans \mathcal{F}
- si $F \in \mathcal{F}$ et si $X \in \mathcal{A}$ est une variable, alors $(\forall X, F)$ et $(\exists X, F)$ sont aussi dans \mathcal{F}

On dit qu'une variable X est libre lorsqu'elle est contenue dans une sous formule précédée du quantificateur $\forall X$ ou $\exists X$. Sinon elle est liée. On dit qu'une formule est clause si elle ne contient pas de variables libres.

Une formule close F est satisfaisable (ou satisfiable) s'il existe un modèle dans lequel F est vraie. Sinon on dit que F est insatisfiable, ou réfutable. La définition se généralise à un ensemble de formules.

2 Introduction au 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 P enseigne M". Le symbole `:-` se lit *si*, et la virgule entre `apprend(E,M)` et `enseigne(P,M)` signifie *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 : [Flach \(1994\)](#). Un autre ouvrage plutôt plus complet est [Sterling et Shapiro \(1994\)](#).

3 La grammaire du langage Prolog

La première chose à faire est d'étudier la grammaire du langage, c'est à dire comment sont formés les mots du langage (en linguistique on parle de morphologie), mais aussi comment peuvent s'agencer les mots pour former un programme valide (c'est la syntaxe).

La grammaire du langage Prolog est non-contextuelle, ce qui signifie que toutes les règles ne contiennent à gauche qu'un seul symbole. Plus intuitivement, pour décrire un symbole non-terminal (entre chevrons) on a pas besoin de connaître ce qu'il y a avant ou après, d'où le "non-contextuel". J'en suis arrivé à cette définition de la grammaire de mon sous-ensemble du langage Prolog, ici présentée sous la forme de Backus-Naur :

```
<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> | <Prédicat> | <Prédicat> '(' <ListeTermes> ')' | <Tableau>
<Tableau> ::= '[]' | '[' <ListeTermes> ']'
| '[' <ListeTermes> '|' <Tableau> ']' | '[' <ListeTermes> '|' <Variable> ']'
```

Les grammaires non-contextuelles sont équivalentes aux automates à pile : ces derniers sont plus puissants que les automates finis (équivalents aux expressions régulières), mais moins puissants que les machines de Turing.

Une sous classe des grammaires non contextuelles est celle des grammaires non-contextuelles déterministes. Une grammaire non contextuelle est dit déterministe si elle est équivalente à un automate à pile déterministe. Cette sous catégorie est intéressante puisqu'il existe alors des algorithmes en temps linéaire pour effectuer l'analyse syntaxique... Je suppose que cette grammaire de Prolog est bien déterministe, mais je ne sais pas le montrer.

De toute façon la structure de l'analyseur syntaxique que je vais utiliser me permet en théorie d'utiliser n'importe quelle grammaire non-contextuelle.

4 La représentation des programmes Prolog

C'est probablement la partie la plus importante du design de l'interpréteur Prolog, puisque la manière de stocker les programmes Prolog en Caml impactera toute la suite.

```
type var = Id of string * int
type term = Var of var | Predicate of string * (term list)
type clause = Clause of term * (term list)
```

Cette première structure permet de représenter le plus petit Prolog possible. Chaque variable est représentée par une chaîne de caractères, son nom, et un numéro. Le numéro permet un renommage facile des variables, qui est nécessaire pour éviter des conflits noms dans l'algorithme d'unification. Le type term est le plus important, un terme peut être une variable, où une structure composée d'une chaîne, le prédicat et une liste de termes. Ainsi `append(eve, mathématiques)` est un terme. Finalement une clause est un terme, qui est à gauche du symbole `:-`, et la liste des termes qui sont à droite. Un programme sera alors une liste de clauses, séparées par des points.

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, [Smaill \(2009\)](#). Une structure proche est utilisée en Lisp par Peter Norvig dans [Norvig \(1992\)](#).

```
type var = Id of string * int
type table = Empty | NonEmpty of term * table | TVar of var
and term = Var of var | Predicate of string * (term list) | Table of table
type clause = Clause of term * (term list)
```

L'ajout des listes Prolog (je les ai appelées tables ici pour les différencier des listes du langage Caml, mais ce n'est probablement pas le meilleur nom), rend la structure sensiblement plus complexe. J'ai choisi ce design pour pouvoir modéliser facilement une table comme `[a, b, c | X]`. Ici X est une variable qui ne peut que représenter une table, on ne veut pas que cela puisse être un terme quelconque (comme `eve` exemple). C'est une très bonne chose de rendre impossible la représentation de programmes invalides dans le type des objets, mais cela rend certaines choses plus compliquées. Les variables sont présentes à différents endroits, et certaines ne peuvent représenter que des tableaux alors que d'autres peuvent représenter n'importe quel terme.

J’ai mis un moment (plus ou moins les deux derniers mois) à me décider sur cette structure, et c’est parce que j’ai dû la changer un certain nombre de fois que j’ai réécrit beaucoup de choses pour qu’elles soient un peu plus modulables.

5 L’analyse syntaxique

Si j’ai bien appris une chose en écrivant un premier parser moi même sans reprendre de grandes idées existantes, c’est qu’il mieux vaut bien réfléchir avant si on veut être capable de modifier, ou même de relire son code après. Maintenant j’ai compris que le parsing est un art en soi, et j’ai décidé d’écrire un analyseur récursif descendant utilisant des combinateurs. J’ai appris les idées dans [Ljunglöf \(2002\)](#).

Je vois ici un parser comme une fonction qui à une chaîne de caractères à analyser renvoie un couple composé de la chaîne restante à analyser et une structure, un arbre produit par l’analyseur qui représente les caractères analysés. L’idée des combinateurs est de créer de nouveaux parsers plus complexes à partir de ceux existants. Voici la structure que produit le parser :

```
type structure = S of string | V of var | L of table | T of term | C of clause
               | W of clause list | P of string | TL of term list
```

Les parsers élémentaires que j’ai écrit directement permettent de reconnaître un caractère passé en paramètre et renvoient une structure `S chaîne` avec chaîne étant le caractère reconnu. J’en ai écrit d’autres permettant de reconnaître n’importe quelle lettre majuscule, ou minuscule, ou encore n’importe quel caractère valide.

Ensuite j’utilise deux combinateurs : `<+>` et `<*>`. Pour des raisons de priorité des opérateurs en OCaml, j’ai renommé le `<*>` en `*~*`. Le combinateur `<+>` exprime le *ou* | des règles de grammaire. Le combinateur `*~*` quant à lui permet de chaîner les parsers, il n’est pas représenté dans les règles de grammaire, les symboles sont juste accolés. Ainsi la règle `<Mot> ::= <Caractère> | <Caractère> <Mot>` se traduirait par :

```
let rec mot = caractère <+> caractère *~* mot
```

Malheureusement, on ne peut pas exprimer directement la récursion ainsi. . . Même si cela fonctionne très bien pour les règles non récursives, une limitation de OCaml —pour éviter de déclarer des structures comme `let rec a = a—` impose d’écrire quelques lignes en plus, rendant le code moins lisible.

Le dernier opérateur que je définis est `>~>`, d’une manière très similaire au `bind` `>=` monadique. Cet opérateur prend à gauche un parser, et à droite une fonction d’une liste de structures vers une structure (en simplifiant). La fonction peut par exemple décrire comment construire un nouveau mot à partir d’un caractère et d’un mot.

6 Les fonctions de base sur les programmes

- `string_of_term : term -> string` Transforme un terme en une chaîne de caractères.
- `var_in_term : var -> term -> bool` Teste si une variable est dans un terme.
- `var_in_eql : var -> (term * term) list -> bool` Teste si une variable est dans une liste de couples de termes.
- `replace_var_in_term : var -> term -> term -> term` Remplace une variable par un terme dans un terme. Attention, la fonction échoue si on essaie de remplacer une variable qui est censée représenter un tableau par un terme qui n’est pas un tableau.
- `replace_var_in_eql : var -> term -> (term * term) list -> (term * term) list` Remplace une variable par un terme dans une liste de couples de termes. Mêmes limitations.
- `find_vars_in_term_list : term list -> term list` Liste toutes les variables d’une liste de termes.
- `find_tvars_in_term : term -> var list` Liste toutes les variables représentant un tableau dans un terme.
- `find_tvars_in_term_list : term list -> var list` Liste toutes les variables représentant un tableau dans une liste de termes.
- `find_tvars_in_clause : clause -> var list` Liste toutes les variables représentant un tableau dans une clause.

7 La correction des types

Le problème est qu’une même variable ne peut pas représenter n’importe quel terme si l’on sait que la variable ne peut représenter que des tableaux. La solution est de rechercher toutes les variables qui ne peuvent représenter que des tableaux, et forcer toutes leurs occurrences par des variables qui ne peuvent représenter que des tableaux.

Ensuite j’ai juste écrit des fonctions composant le parser et le vérificateur de types.

Dans toute la suite du code il ne faut jamais qu’une variable apparaisse à un endroit comme pouvant représenter n’importe quel terme et dans un autre comme pouvant ne représenter que des tableaux.

8 L'algorithme derrière Prolog

Ma référence en ce qui concerne l'algorithme utilisé par Prolog est [Nilsson et Małuszyński \(1990\)](#).

Ce qui semble être le premier article traitant du type de résolution utilisé par Prolog est [Robinson \(1965\)](#). 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 $P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q$, que l'on exprime souvent comme une grande disjonction de littéraux $\neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \vee Q$. Une clause de Horn est donc une disjonction de littéraux comportant au plus un littéral positif. Nous utilisons en Prolog que des clauses de Horn qui possèdent exactement un littéral positif ^[citation needed].

Les clauses de Horn sont contraignantes, par exemple $\neg P \Rightarrow Q$ n'est pas une clause de Horn. On peut par contre exprimer $(P_1 \vee P_2) \Rightarrow Q$ avec des clauses de Horn, il suffit d'écrire les deux clauses $P_1 \Rightarrow Q$ et $P_2 \Rightarrow Q$.

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 Robert Kowalski en 1974, la correction est montrée par Keith Clark en 1979, la complétude a été montrée premièrement par Robert Hill en 1974, mais quelque chose de plus fort a été montré par Clark en 1979.

9 L'unification

- On définit une substitution comme un ensemble de couples (variable, terme). Le terme est inséré à la place de la variable.
- L'unification se fait entre deux termes t_1 et t_2 . Deux termes s'unifient s'il existe une substitution θ des variables de t_1 et de t_2 telle que $\theta(t_1) = \theta(t_2)$. Une telle substitution est appelée un unifieur.
- On définit une loi de composition sur ces substitutions. $\theta_2\theta_1(t) = \theta_2(\theta_1(t))$ (J'ai l'impression que j'ai défini la composition dans l'ordre inverse de ce que l'on peut lire dans la littérature... à modifier)
- Si θ_1 et θ_2 sont deux unifieur, et s'il existe une substitution σ telle que $\theta_2 = \sigma\theta_1$, on dit que l'unifieur θ_1 est plus général que θ_2 , ce que l'on note $\theta_2 \preceq \theta_1$
- Il existe des unifieurs plus généraux que tous les autres. On appelle *Most General Unifier*, que j'abrège *MGU*, un tel unifieur. En fait la relation \preceq n'est pas antisymétrique, donc il peut exister deux unifieurs θ_1 et θ_2 tels que $\theta_1 \preceq \theta_2$ et $\theta_2 \preceq \theta_1$. Le *MGU* est en fait unique au renommage des variables près.

La description de l'algorithme d'unification, qui permet de trouver l'unifieur le plus général est très bien faite dans [Nilsson et Małuszyński \(1990\)](#). De plus, il en est donné une preuve sa terminaison et de sa correction, ce qui est très bon point.

L'algorithme manipule des systèmes d'équations. Un système d'équations $\{X_1 = t_1, \dots, X_n = t_n\}$ est dit sous forme résolue si les (X_n) sont des variables, les (t_n) sont des termes et aucune des variables X_n n'apparaît dans les t_n . Si $\{X_1 = t_1, \dots, X_n = t_n\}$ est sous forme résolue, alors $\{X_1/t_1, \dots, X_n/t_n\}$ est un *MGU*.

E est l'ensemble des équations. Au départ il n'y en a qu'une seule :

Par exemple $E = \{\text{etudiant_de}(E,P) = \text{etudiant_de}(E, \text{pierre})\}$.

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 :

$f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$ avec $n \geq 0$

Alors remplacer l'équation par $s_1=t_1 \dots s_n=t_n$

$f(s_1, \dots, s_m) = g(t_1, \dots, t_n)$ avec $f \neq g$

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 $X \neq t$ et X apparaît plus d'une fois dans E

Si X est un sous-terme de t Alors ÉCHEC

Sinon on remplace toutes les autres occurrences de X par t

- Il est prouvé que cet algorithme termine et renvoie soit échec, soit un ensemble équivalent des équations sous *forme résolue*.

- 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 [Norvig \(1992\)](#) "*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.*"
- **Attention !** Aucune variable ne doit apparaître en même temps dans les deux termes que l'on cherche à unifier : 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.
- L'algorithme n'est qu'une grande disjonction de cas, et le compléter pour prendre en compte des tableaux n'a pas été extrêmement difficile. . . Pour unifier deux tableaux il suffit d'unifier la tête, puis récursivement d'unifier la queue. L'algorithme devient par contre nettement plus long. Je l'ai écrit en utilisant des *sets*, dans un style impératif, ce qui après de longues fonctions récursives peut paraître rafraîchissant.

Mais à quoi sert l'unification ? Soit la clause du programme Prolog `etudiant_de(E,P):-apprend(E,M), enseigne(P,M)` . . Si l'on cherche à réaliser la requête `?-etudiant_de(E, pierre)`, on unifie `etudiant_de(X, pierre)` avec `etudiant_de(E,P)`, le membre de gauche de la clause. La substitution θ remplace P par `pierre` et E par X. Pour continuer la recherche, on applique la substitution au termes de droite de la clause, ce qui nous donne la nouvelle requête.

10 Le backtracking

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

- On cherche à satisfaire une requête (t_1, t_2, \dots, t_n) .
- On a t_1 qui s'unifie avec h^i , où $h^i \leftarrow t_1^i, \dots, t_m^i$ est une la i-ème clause du programme. On a auparavant renommé toutes les variables de la clause qui étaient présentes dans t_1 , sinon on ne peut pas appeler l'algorithme d'unification. On a donc un *MGU*, que l'on nomme θ_1 .
- On applique θ_1 au second membre de la clause, et à (t_2, \dots, t_n) . La nouvelle requête à satisfaire est donc $(\theta_1(t_1^i), \dots, \theta_1(t_m^i), \theta_1(t_2), \dots, \theta_1(t_n))$.
- On récurse, si on a à montrer $\theta_k(\dots \theta_2(\theta_1(\emptyset)) \dots)$ alors c'est gagné, puisque \emptyset signifie toujours vrai. La composée θ_{tot} des θ_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 t_1, \dots, t_n par θ_{tot} .
- Si t_1 ne s'unifie avec aucun des h^i , ça ne sert à rien d'essayer d'unifier t_2 , puisque de toute façon on garde t_1 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 initiale
- 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, etc. . . pas de conflits !

11 Discussion, histoire, etc. . .

J'ai beaucoup aimé la lecture de la préface de [Sterling et Shapiro \(1994\)](#). Robert Kowalski raconte son premier contact avec Prolog, le langage ayant été initié vers 1972 par Alain Colmerauer et Philippe Roussel à Marseille. Kowalski y est invité, mais les serveurs de calcul ne sont pas là, ils se connectent par telnet sur une machine IBM à Grenoble ! Pour la petite histoire, le premier message que Kowalski reçoit à l'exécution de son programme est "DEBORDEMENT DE PILE" !

Kowalski a introduit le SLD-résolution, entre autre nombreux travaux sur les algorithmes de Prolog. Il a eu deux étudiants en thèse. Keith Clark, qui a montré la correction et la complétude de la SLD-résolution et qui a introduit le principe de *negation as failure*. David H. D. Warren lui a écrit le premier compilateur pour Prolog, utilisant les *machines abstraites de Warren*.

J'ai un peu recherché qui travaillait encore sur Prolog et la programmation logique en France. Tous sont plus ou moins liés à l'équipe *PARTOUT* de l'Inria : <https://team.inria.fr/partout/>

12 Mémoïsation et évitement des boucles infinies

La méthode traditionnelle pour éviter les problèmes de boucles infinies est d'utiliser des *cuts*, un symbole ! à écrire dans le code qui coupe impérativement l'arbre de recherche si l'on en vient à montrer ce terme. . . *A priori* je ne suis pas fan, mais c'est ce que tout le monde utilise. À la place je me suis tourné vers la mémoïsation.

J'ai eu ma petite révélation quand j'ai compris qu'il existait deux David Warren qui ont travaillé sur Prolog, et qui ont même publié des papiers la même année, pouvant être référencés en même temps dans les mêmes articles ! Cette fois je m'intéresse

à cet article de David S. Warren : [Warren \(1992\)](#)

L'idée est de mémoriser les requêtes que l'on a déjà résolues, pour ensuite accélérer l'exécution. Un des avantages de cette méthode est de pouvoir éviter certaines boucles infinies...

Je n'ai pas repris la mémoïsation globale, (il faudrait que je réessaie, mais ce n'est pas aussi évident de l'écrire convenablement que de le dire). La seule chose que j'ai fait, c'est d'associer à chaque terme t_0 que l'on cherche à montrer une généalogie, c'est à dire les termes t_1, t_2, \dots, t_n , tels t_k appartienne au membre de droite de la clause dont on a unifié la tête avec t_{k+1} . —Gardons le terme généalogie, qui est bien plus clair que ce que je viens d'écrire.— La seule chose que je vérifie, c'est qu'il n'y ait pas deux termes identique (à une bijection des variables près...).

Qu'est-ce que ça change dans l'exécution des programmes :

```
natural(zero).
natural(s(N)) :- natural(N).
```

```
?- natural(X)
```

Ce programme renvoie un à un tous les entiers naturels quand on l'exécute sans vérification des boucles : **zero**, **s(zero)**, **s(s(zero))**... Avec la vérification, il renvoie juste **zero**.

```
natural(s(N)) :- natural(N).
natural(zero).
```

```
?- natural(X)
```

Ce programme ne renvoie rien sans vérification (boule infinie, puis dépassement de pile), mais renvoie juste **zero** avec vérification.

```
natural(zero).
natural(s(N)) :- natural(N).
```

```
?- natural(s(s(s(zero))))
```

Ce programme renvoie **vrai** pour les deux algorithmes.

Est-ce que c'est vraiment ça que je cherche comme comportement ? Mérite réflexion... D'un certain point de vue il semble plus agréable de ne pas avoir de différence dans le résultat en organisant les clauses dans un ordre différent...

Une application sympa est ma carte en Prolog du métro londonien (à la toute fin du code). On peut maintenant facilement définir la commutativité :

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

On peut demander `path(oxford_circus, charing_cross, R)`, tous les chemins entre deux stations, et le programme ne part pas dans une boucle infinie. Il renvoie tous les chemins qui ne passent pas deux fois par la même station intermédiaire.

13 La négation

Cette fois il me faut réfléchir à la négation en Prolog. C'est une sortie des clauses de Horn. C'est à faire, mais là il n'y a qu'une seule manière de s'y prendre, c'est d'utiliser la *negation as failure* bien décrite dans [Clark \(1978\)](#). C'est néanmoins très loin de ce que l'on pourrait espérer d'une négation logique.

La négation c'est fait ! Avec, on peut encore améliorer mon petit programme du métro londonien : maintenant il n'affiche que les chemins stricts d'*Oxford Circus* à *Charing Cross* !

14 Le Cluedo, une synthèse

L'article que j'avais trouvé traitant du Cluedo est toujours là : [Aartun \(2016\)](#). J'ai le sentiment qu'il manque plusieurs choses au minuscule Prolog que j'avais écrit pour pouvoir espérer un programme de Cluedo en Prolog :

- Des listes
- Des moyens de couper les arbres de recherche et d'éviter les boucles infinies
- Une sorte de négation

Le premier est fait, pour le second, mon implémentation est peut être suffisante, peut-être pas. Le dernière est faite aussi maintenant, mais elle donne une négation un peu limitée, et donc délicate à manipuler.

Ce que je n'ai pas envie d'implémenter :

- Des entiers naturels, ou pire des flottants (j'ai l'impression que je peux les éviter dans mon programme Cluedo)
- Tout plein de fonctions et d'opérateurs infixés qui sont définis dans les Prolog modernes
- En particulier des fonctions d'affichages, d'entrée-sortie...

Bon, mon implémentation de Prolog est déjà relativement (avec les listes) plus évoluée que la plupart des projets amateurs que j'ai trouvés sur Internet.

Quelques règles simplifiées du Cluedo :

- Il y a un certain nombre $N_{joueurs}$ de joueurs et N_{cartes} de cartes.
- On suppose que $N_{cartes} \equiv 1 \pmod{N_{joueurs}}$. Alors chaque joueur reçoit

$$c = \frac{N_{cartes} - 1}{N_{joueurs}}$$

- Le but du joueur est de deviner la carte qu'aucun joueur ne possède.
- Chaque joueur a le droit à son tour d'émettre une supposition, par exemple : "Le joueur A suppose que le joueur B possède une des trois cartes suivantes (a, b, c) ". Si le joueur B possède une des cartes (a, b, c) , il doit en montrer une —celle de son choix— au joueur A , sans la révéler aux autres joueurs de la partie.

It works !

Un exemple de mon programme de Cluedo (pas encore fini) !

```
joueurs([samuel, pierre, benjamin]).
cartes([cuisine, chambre, salle_de_bain, salon, garage, salle_a_manger, veranda]).
```

```
possede(samuel, cuisine).
possede(samuel, chambre).
```

```
possede_aucune_triplet(pierre, [salle_de_bain, salon, cuisine]).
possede_une_des(pierre, [garage, cuisine, salon]).
```

À la requête `possede(pierre, garage)`, le programme répond *Vrai* !

Références

Vemund Innvæer AARTUN : Reasoning about knowledge and action in cluedo using prolog. 2016.

Keith L. CLARK : Negation as failure. 1978.

Peter FLACH : *Simply Logical : Intelligent Reasoning by Example*. 1994.

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

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

Peter NORVIG : *Paradigms of Artificial Intelligence Programming*. 1992.

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

Alan SMAILL : Logic programming. 2009.

Leon STERLING et Ehud SHAPIRO : *The Art of Prolog*. 2nde édition, 1994.

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

A Code

A.1 types.ml

```
type var = Id of string * int

type table = Empty | NonEmpty of term * table | TVar of var
```

```

and term = Var of var | Predicate of string * term list | Table of table

type clause = Clause of term * term list
type 'a tree = Leaf of 'a | Node of 'a tree Lazy.t list

module TermSet = Set.Make (struct
  type t = term * term

  let compare = compare
end)

module VarMap = Map.Make (struct
  type t = var

  let compare = compare
end)

module TermMap = Map.Make (struct
  type t = term

  let compare = compare
end)

```

A.2 lexer.mll

```

{
  open Parser
  exception Error of string
}

let char = ['a'-'z' 'A'-'Z' '0'-'9' '_' ]

rule mylexer = parse
| [' ' '\t' '\n'] { mylexer lexbuf }
| '.' { STOP }
| ":-" { IF }
| ',' { AND }
| '(' { LPAR }
| ')' { RPAR }
| '[' { LBRA }
| ']' { RBRA }
| '|' { TAIL }
| '?' { ASK }
| '_' { NVAR }
| ['A'-'Z'] char* as s { VAR s }
| ['a'-'z'] char* as s { PRE s }
| eof { EOF }
| _ { raise (Error (Printf.sprintf "At offset %d: unexpected character.\n" (Lexing.lexeme_start lexbuf))) }

```

A.3 parser.mly

```

%token EOF
%token <string> VAR
%token <string> PRE
%token IF
%token AND
%token STOP
%token LPAR
%token RPAR
%token LBRA

```



```

%token RBRA
%token TAIL
%token ASK
%token NVAR

%start <Types.clause list> programme

%start <Types.term list> requete

%%

requete:
| r = liste_termes ASK EOF { r }

programme:
| c = clause EOF { [c] }
| c = clause p = programme { c :: p }

clause:
| t = terme STOP { Types.Clause (t, []) }
| t = terme IF l = liste_termes STOP { Types.Clause (t, l) }

var:
| v = VAR { Types.Id (v, 0) }
| NVAR { Utility.newvar () }

terme:
| v = var { Types.Var v }
| p = PRE { Types.Predicate (p, []) }
| p = PRE LPAR l = liste_termes RPAR { Types.Predicate (p, l) }
| t = table { Types.Table t }

liste_termes:
| t = terme { [t] }
| t = terme AND l = liste_termes { t :: l }

table:
| LBRA RBRA { Types.Empty }
| LBRA l = liste_termes RBRA { Utility.table_of_termlist Types.Empty l }
| LBRA l = liste_termes TAIL t = table RBRA { Utility.table_of_termlist t l }
| LBRA l = liste_termes TAIL v = var RBRA { Utility.table_of_termlist (Types.TVar v) l }

```

A.4 solver.ml

```

open Types
open Utility
open Unification

```

```

(* Recherche les variables ne pouvant désigner qu'un tableau, (comme C dans cet exemple : [A, B | C]), et remplace
let type_check_term term =
  List.fold_left
    (fun t v -> replace_var_in_term v (Table (TVar v)) t)
    term (find_tvars_in_term term)

let type_check_cl cl =
  let l = find_tvars_in_clause cl in
  List.fold_left
    (fun cl v ->
      let aux = replace_var_in_term v (Table (TVar v)) and (Clause (left, right)) = cl in
      Clause (aux left, List.map aux right) )

```

```

cl 1

(* Lit un programme et s'occupe des variables tableau *)
let read_program str =
  str |> Lexing.from_string |> Parser.programme Lexer.mylexer |> List.map type_check_cl

let read_request str =
  str |> Lexing.from_string |> Parser.requete Lexer.mylexer |> List.map type_check_term

(* Applique une substitution *)
let apply_subst_on_term = VarMap.fold replace_var_in_term

(* Applique une substitution sur une liste de termes *)
let apply_subst_on_termlist uni = apply_subst_on_term uni |> List.map

(* Prend un entier et une clause, renvoie la clause avec les variables renommées à l'entier *)
let rename n (Clause (t1, t1)) =
  let rec f = function
    | Var (Id (str, _)) -> Var (Id (str, n))
    | Predicate (atm, l) -> Predicate (atm, List.map f l)
    | Table t ->
      let rec aux = function
        | Empty -> Empty
        | TVar (Id (str, _)) -> TVar (Id (str, n))
        | NonEmpty (head, tail) -> NonEmpty (f head, aux tail) in
      Table (aux t) in
  Clause (f t1, List.map f t1)

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

(* Transforme un arbre (possiblement infini) en la séquence de ses feuilles. Parcours en profondeur. *)
let rec to_seq = function
  | Leaf str -> Seq.return str
  | Node t1 -> Seq.flat_map (fun par -> to_seq (Lazy.force par)) (list_to_seq t1)

(* La fonction de recherche, renvoie l'arbre des solutions *)
let rec sld_tree world req subs n =
  match req with
  | [] -> Leaf subs
  | head_request_term :: other_request_terms ->
    Node
      (List.filter_map
        (fun c ->
          let (Clause (left_member, right_member)) = rename n c in
          match mgu head_request_term left_member with
          | None -> None
          | Some unifier ->
            Some
              ( lazy
                (sld_tree world
                  (apply_subst_on_termlist unifier (right_member @ other_request_terms))
                  (unifier :: subs) (n + 1) ) ) )
        )
      world )

let solutions tree vars =
  Seq.map (fun l -> (vars, List.fold_right apply_subst_on_termlist l vars)) (to_seq tree)

let request world req =
  let termlist = read_request req in
  let vars = find_vars_in_termlist termlist in

```

```

let sol = solutions (sld_tree world termlist [] 1) vars in
if sol () = Seq.Nil then Format.printf "This is false.\n%!"
else
  Seq.iter
    (fun (vars, tl) ->
      if vars = [] then Format.printf "This is true.\n%!"
      else
        Format.printf "There is : %s\n%!"
          (String.concat ", "
            (List.map2 (fun v t -> string_of_term v ^ " = " ^ string_of_term t) vars tl) ) )
    sol ;
Format.printf "\n%!"

```

A.5 utility.ml

open Types

```

let sprintf = Format.sprintf
let printf = Format.printf
let iter = ref 0

let newvar () =
  iter := !iter + 1 ;
  Id (" $" ^ string_of_int !iter, 0)

let rec table_of_termlist tail = function
| [] -> tail
| t :: tl -> NonEmpty (t, table_of_termlist tail tl)

(* Transforme un terme en une chaîne de caractères *)
let rec string_of_term = function
| Predicate (p, []) -> p
| Predicate (p, l) -> sprintf "%s(%s)" p (String.concat ", " (List.map string_of_term l))
| Var (Id (s, n)) -> sprintf "%s-%i" s n
| Table Empty -> "[]"
| Table (TVar (Id (s, n))) -> sprintf "%s-%i" s n
| Table (NonEmpty (t, l)) ->
  let rec aux = function
  | Empty -> ""
  | TVar (Id (s, n)) -> sprintf "|%s-%i" s n
  | NonEmpty (t, l) -> sprintf ", %s%s" (string_of_term t) (aux l) in
  sprintf "[%s%s]" (string_of_term t) (aux l)

(* DEBUG : affiche une liste de couple de termes *)
let print_eqs eqs =
  TermSet.iter (fun (t1, t2) -> printf "%s <-> %s\n%!" (string_of_term t1) (string_of_term t2)) eqs

(* Applique une substitution sur un terme *)
(* Attention, on ne peut remplacer une variable qui représente un tableau que par un tableau *)
let rec replace_var_in_term var new_term term =
  match term with
  | Var v -> if v = var then new_term else Var v
  | Predicate (p, l) -> Predicate (p, List.map (replace_var_in_term var new_term) l)
  | Table tbl ->
    let rec aux = function
    | Empty -> Empty
    | TVar v ->
      if v = var then
        match new_term with
        | Table nt -> nt

```

```

        | _ -> failwith "Try to replace a table by a term that is not a table"
      else TVar v
    | NonEmpty (head, tail) -> NonEmpty (replace_var_in_term var new_term head, aux tail) in
    Table (aux tbl)

let replace_var_in_eqs var new_term eqs =
  TermSet.map
    (fun (a, b) -> (replace_var_in_term var new_term a, replace_var_in_term var new_term b))
    eqs

let rec var_in_term var = function
| Var v -> if v = var then true else false
| Predicate (_, l) -> List.mem true (List.map (var_in_term var) l)
| Table t ->
  let rec aux = function
  | Empty -> false
  | NonEmpty (head, tail) -> var_in_term var head || aux tail
  | TVar v -> if v = var then true else false in
  aux t

let var_in_eqs var eqs =
  TermSet.exists (function a, b -> var_in_term var a || var_in_term var b) eqs

(* Recherche les termes qui sont des variables récursivement dans une liste de termes *)
let rec find_vars_in_termlist tl =
  let rec find_vars_in_term = function
  | Var v -> [Var v]
  | Predicate (_, l) -> find_vars_in_termlist l
  | Table t ->
    let rec aux = function
    | Empty -> []
    | TVar v -> [Table (TVar v)]
    | NonEmpty (head, tail) -> find_vars_in_term head @ aux tail in
    aux t in
  List.sort_uniq compare (List.concat (List.map find_vars_in_term tl))

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'un terme *)
let rec find_tvars_in_term = function
| Var _ -> []
| Predicate (_, l) -> List.concat (List.map find_tvars_in_term l)
| Table t ->
  let rec aux = function
  | Empty -> []
  | NonEmpty (head, tail) -> find_tvars_in_term head @ aux tail
  | TVar v -> [v] in
  aux t

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'une liste de termes *)
let find_tvars_in_termlist tl = List.concat (List.map find_tvars_in_term tl)

(* Renvoie la liste des variables représentant un tableau à l'intérieur d'une clause *)
let find_tvars_in_clause (Clause (t, tl)) = find_tvars_in_term t @ find_tvars_in_termlist tl

```

A.6 unification.ml

```

open Types
open Utilitary

```

```

(* E est un ensemble couples de termes *)
(* transforme E en un équivalent sous forme résolue (type option, None si impossible) *)

```

```

type equation_set = {mutable data: TermSet.t; mutable correct: bool}

let solve e =
  let eqs = {data= e; correct= true} in
  let aux = function
    | Predicate (f, lf), Predicate (g, lg) when f = g && List.length lf = List.length lg ->
      List.iter2 (fun a b -> eqs.data <- TermSet.add (a, b) eqs.data) lf lg ;
      eqs.data <- TermSet.remove (Predicate (f, lf), Predicate (g, lg)) eqs.data
    | Predicate (_, _), Predicate (_, _) -> eqs.correct <- false
    | Var x, Var y when x = y -> eqs.data <- TermSet.remove (Var x, Var y) eqs.data
    | Predicate (a, la), Var x ->
      eqs.data <- TermSet.remove (Predicate (a, la), Var x) eqs.data ;
      eqs.data <- TermSet.add (Var x, Predicate (a, la)) eqs.data
    | Table t, Var x ->
      eqs.data <- TermSet.remove (Table t, Var x) eqs.data ;
      eqs.data <- TermSet.add (Var x, Table t) eqs.data
    | Var w, t when var_in_eqs w (TermSet.remove (Var w, t) eqs.data) ->
      if var_in_term w t then eqs.correct <- false
      else (
        eqs.data <- TermSet.remove (Var w, t) eqs.data ;
        eqs.data <- replace_var_in_eqs w t eqs.data ;
        eqs.data <- TermSet.add (Var w, t) eqs.data )
    | Var _, _ -> ()
    | Predicate (_, _), Table _ -> eqs.correct <- false
    | Table _, Predicate (_, _) -> eqs.correct <- false
    | Table t1, Table t2 -> (
      match (t1, t2) with
      | Empty, Empty -> eqs.data <- TermSet.remove (Table Empty, Table Empty) eqs.data
      | Empty, NonEmpty _ -> eqs.correct <- false
      | NonEmpty _, Empty -> eqs.correct <- false
      | (NonEmpty (head1, tail1) as t1), (NonEmpty (head2, tail2) as t2) ->
        eqs.data <- TermSet.remove (Table t1, Table t2) eqs.data ;
        eqs.data <- TermSet.add (head1, head2) eqs.data ;
        eqs.data <- TermSet.add (Table tail1, Table tail2) eqs.data
      | ((NonEmpty (_, _) | Empty) as t), TVar a ->
        eqs.data <- TermSet.remove (Table t, Table (TVar a)) eqs.data ;
        eqs.data <- TermSet.add (Table (TVar a), Table t) eqs.data
      | TVar a, TVar b when a = b ->
        eqs.data <- TermSet.remove (Table (TVar a), Table (TVar b)) eqs.data
      | TVar w, table when var_in_eqs w (TermSet.remove (Table (TVar w), Table table) eqs.data) ->
        if var_in_term w (Table table) then eqs.correct <- false
        else eqs.data <- replace_var_in_eqs w (Table table) eqs.data
      | TVar _, _ -> () ) in
  let old = ref TermSet.empty in
  while eqs.data <> !old && eqs.correct do
    old := eqs.data ;
    TermSet.iter (fun c -> if eqs.data <> !old then () else aux c) eqs.data
  done ;
  if eqs.correct then Some eqs.data else None

```

(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 mgu r c =
  let f = function
    | Var x, t -> (x, t)
    | Predicate (_, _), _ -> failwith "Should not happen."
    | Table (TVar r), t -> (r, t)
    | Table (Empty | NonEmpty _), _ -> failwith "Should not happen." in

```

```

match solve (TermSet.singleton (r, c)) with
| None -> None
| Some e -> Some (e |> TermSet.to_seq |> Seq.map f |> VarMap.of_seq)

```

A.7 main.ml

```

let () =
  Format.printf "Bienvenue !\n\n" ;
  Tests.zebra () ;
  let world =
    Solver.read_program
    "
sous_sujet(alice, groupe1).
sous_sujet(groupe1, groupe2).
sous_sujet(bob, groupe2).

sous_objet(fichierA, dossier1).
sous_objet(fichierB, dossier2).
sous_objet(dossier1, dossier2).

autorise(groupe1, dossier2).

autorise(U, F) :- sous_objet(F, D), autorise(U, D).
autorise(U, F) :- sous_sujet(U, G), autorise(G, F).
"
  in
  let req = Solver.request world in
  req "autorise(U, F) ?"

```