Programmation logique et contrôle d'accès

Samuel Gallay

22 novembre 2020

Plan de l’exposé

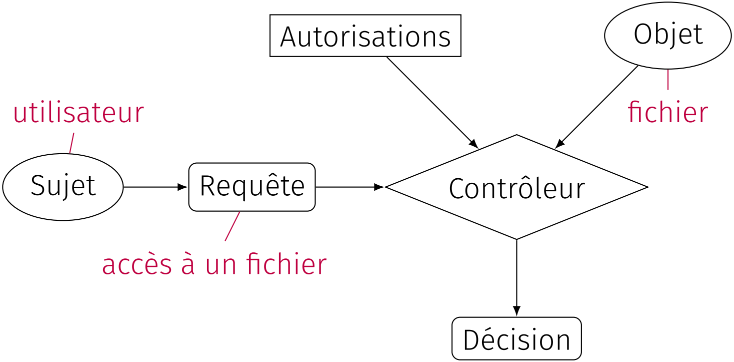
I. Contrôle d’accès

II. Logique et Prolog

III. Implémentation de l’interpréteur Prolog

IV. Tests de l’implémentation

Contrôle d’accès : motivation



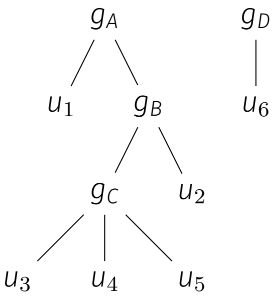
Matrice de contrôle d'accès : limitations

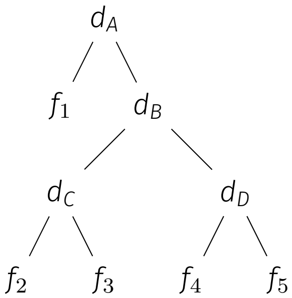
Fichier \ Utilisateur	Alice	Bob	Charlie	…
Fichier 1	1	1	0	…
Fichier 2	0	1	1	…
Fichier 3	1	0	0	…
…	…	…	…	

Inconvénients

- taille de la matrice
- suppression des autorisations

Arborescence de fichiers et groupes d’utilisateurs





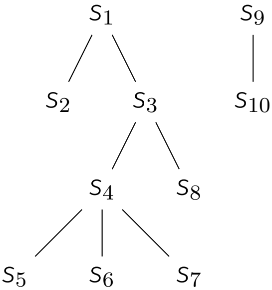
u_i : utilisateur
g_i : groupe

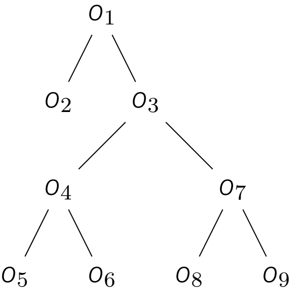
$appartient_{gr}(u_3, g_C)$
$inclus_{gr}(g_B, g_A)$

f_i : fichier
d_i : dossier

$appartient_{dos}(f_4, d_D)$
$inclus_{dos}(d_C, d_B)$

Arborescence de fichiers et groupes d'utilisateurs





s_i : sujet

$sous\_sujet(s_2, s_1)$

o_i : objet

$sous\_objet(o_2, o_1)$

Les règles d'autorisation

- Règles simples :

$$autorise(s_i, o_j), \text{ ou } sous\_sujet(s_i, s_j)$$

- Propagation des autorisations :

$$\forall U, \forall G, \forall D,$$
$$sous\_sujet(U, G) \wedge autorise(G, D) \Rightarrow autorise(U, D)$$

$$\forall F, \forall D, \forall U,$$
$$sous\_objet(F, D) \wedge autorise(U, D) \Rightarrow autorise(U, F)$$

Objectifs

Écrire un système permettant :

1. de représenter ce type de règles
2. de répondre à des questions comme :
   - *Alice peut-elle accéder au fichier A ?*
   - *Qui est-ce qui peut accéder au fichier A ?*
   - *À quels fichiers Alice peut-elle accéder ?*

Logique du premier ordre, ou calcul des prédicats

- Des prédicats, des variables, des fonctions, les connecteurs logiques $\vee$ et $\wedge$, et les quantificateurs $\exists$ et $\forall$.
- Les règles sont des *formules bien définies* dont toutes les variables sont liées (formules closes).
- Mise sous forme prénexe.
- Skolémisation.
- Mise sous forme normale conjonctive.

Logique du premier ordre, ou calcul des prédicats

Les règles sont maintenant toutes de la forme :

$$\forall X_1 \ldots X_s, \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k} L_{i,j} \right)$$

En ne notant plus les quantificateurs, et en séparant en n clauses, on se réduit à la forme :

$$\bigvee_{i=1}^{k} L_i$$

où les L_i sont des littéraux positifs ou négatifs.

Restriction aux clauses définies

Attention

Perte de généralité par rapport à la logique du premier ordre : existence d'algorithmes efficaces sur les clauses de Horn.

Exactement un littéral positif :

$$\left(\bigvee_{i=1}^{n} \neg P_i\right) \vee Q \equiv \bigwedge_{i=1}^{n} P_i \Rightarrow Q$$

Si $n = 0$: un fait, Q toujours vrai.

Notation Prolog :

```
q(f1(X), ...) :- p1(f2(Y), ...), ..., pn(fk(Z), ...).
```

Exemple de programme Prolog

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

étudiant_de(E,P):-apprend(E,M), enseigne(P,M).

étudiant(E, pierre) ?
```

La SLD-Résolution

$$\frac{\neg(A_1 \wedge A_2 \wedge \ldots \wedge A_n) \qquad B_1 \leftarrow B_2 \wedge \ldots \wedge B_k}{\neg(B_2 \wedge \ldots \wedge B_k \wedge A_2 \wedge \ldots \wedge A_n)\theta_1}$$

Avec $\theta_1 = \text{MGU}(A_1, B_1)$.

Si $\neg(A_1 \wedge \ldots \wedge A_n) \Rightarrow$ *Faux*, alors $(A_1 \wedge \ldots \wedge A_n)\theta_1$ est *Vrai*.

Unification : substitution des variables de deux termes. Ex :
`étudiant(E, P) = étudiant(E, pierre)`

Il existe un unifieur plus général.

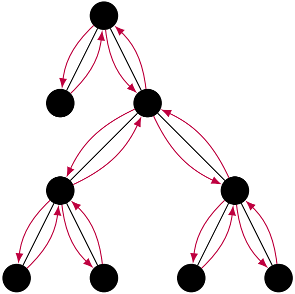
Algorithme d’unification

```
E = {étudiant_de(E,P) = étudiant_de(E, pierre)}.

Répéter tant que E change :
    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
```

Termine et est correct.

Le retour sur trace



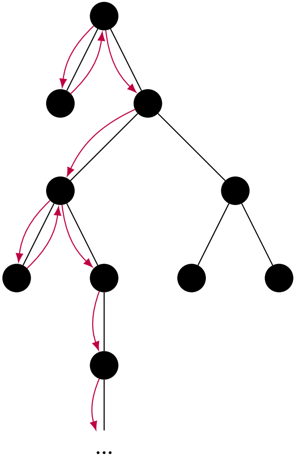
Il faut choisir dans quel ordre utiliser les clauses !

Solution utilisée en Prolog : parcours en profondeur.

Avantage

Faible cout en mémoire.

Le retour sur trace



Il faut choisir dans quel ordre utiliser les clauses !

Solution utilisée en Prolog : parcours en profondeur.

Avantage

Faible cout en mémoire.

Inconvénient

Risque manquer des solutions (boucles infinies).

L'analyse syntaxique

Syntaxe minimale de Prolog 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> '.'
```

Implémentation d'un analyseur syntaxique récursif descendant.

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

L’unification

L’ensemble d’équations est représenté grace à un `Set` en OCaml.

Les substitutions sont des tableaux associatifs des variables sur les termes.

Accès, insertion et suppression en $O(log(n))$.

L’arbre de recherche

Utilisation de l’évaluation paresseuse pour représenter l’arbre possiblement infini :

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

Je transforme l’arbre en une séquence possiblement infinie de ses feuilles (les solutions) :

```
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)
```

J’applique des filtres sur cette séquence, et il suffit d’itérer les solutions.

L'arbre de recherche

```
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)
```

Le problème du zèbre

```
There are five houses.
The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Old Gold smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house.
The man who smokes Chesterfields lives in the house
        next to the man with the fox.
Kools are smoked in the house next to the house
        where the horse is kept.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?
```

Le problème du zèbre

```
member(X, [X | Xs]).
member(X, [Y | Ys]) :- member(X, Ys).

isright(L, R, [L, R | T]).
isright(L, R, [H | T]) :- isright(L, R, T).

nextto(A, B, X) :- isright(A, B, X).
nextto(A, B, X) :- isright(B, A, X).

equal(X, X).
```

Le problème du zèbre

```
zebra(H, W, Z):-
equal(H, [[norwegian, _, _, _, _], _,
          [_, _, _, milk, _], _, _]),
member([englishman, _, _, _, red], H),
member([spaniard, dog, _, _, _], H),
...
nextto([norwegian, _, _, _, _], [_, _, _, _, blue], H),
isright([_, _, _, _, ivory], [_, _, _, _, green], H),
...
member([W, _, _, water, _], H),
member([Z, zebra, _, _, _], H).

zebra(Houses, WaterDrinker, ZebraOwner) ?
```

Retour au contrôle d'accès

```
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).

autorise(U, F) ?
```