# Semaine 14: Programmation logique (2)

Nous avons vu qu'en programmation logique, on recherchait des solutions à des requêtes, étant donné un programme composé de faits et de règles.

Un interpréteur de programmes logiques essaie d'appliquer les règles pour résoudre une requête en utilisant l'unification comme étape élémentaire d'inférence.

Nous allons voir maintenant comment on peut réaliser cette recherche.



# Recherche par retour arrière

L'interpréteur Prolog va comparer de façon récursive une partie de la requête (le *but*) avec une clause du programme.

- Étant donnée une requête vide, l'interpréteur réussit.
- Étant donnée une requête non vide, l'interpréteur tente de faire correspondre le premier prédicat.
  - Correspondre signifie : s'unifier avec la partie gauche d'une clause.
  - Les clauses sont essayées dans l'ordre où elles sont écrites.
  - Si aucune clause ne correspond, on échoue.
  - Si une clause correspond, on invoque récursivement l'interpréteur sur la partie droite de la clause.
- Si l'invocation récursive échoue, on continue avec la clause suivante, sinon on réussit.
- Si l'interpréteur réussit, on continue avec le prédicat suivant de la requête.

# Remplacer l'échec par une liste de succès

Comment peut-on formaliser la stratégie décrite précédemment ?

De façon générale, comment peut-on exprimer une recherche par retour arrière dans un langage fonctionnel ?

Idée : au lieu de représenter l'*échec*, on fabrique la liste de tous les *succès* (les solutions possibles).

Si bien que :

- un échec est représenté par une liste vide de solutions.
- une recherche conjonctive devient une intersection de listes.
- une recherche disjonctive devient une concaténation de listes.

Pour assurer la terminaison et pour des raisons d'efficacité, il est obligatoire que les listes de solutions soient construites de façon paresseuse, à la demande de nouvelles solutions.

Ces listes seront donc modélisées par des flots (streams).



# Exemple de recherche

Supposons que nous ayons un graphe, dans lequel chaque noeud a un champ *successors* qui contient la liste de ses successeurs dans le graphe.

```
class Node {
    val successors: List[Node];
    ...
}
```

On fait l'hypothèse que le graphe est acyclique.

Le but est de rechercher un chemin entre deux noeuds donnés, ou d'échouer s'il n'en existe aucun.

L'idée est qu'au lieu de retourner un chemin ou d'échouer, on retourne toujours une liste, celle de tous les chemins possibles entre les deux noeuds donnés.

Cela est réalisé par la fonction :

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y) List(x)
  else
    for ( val z ← list2stream(x.successors);
          val p ← paths(z, y) ) yield x :: p;
```

ou bien, sans la notation **for** :

```
def paths(x: Node, y: Node): Stream[List[Node]] =
  if (x == y) List(x)
  else
    list2stream(x.successors) flatMap (z ⇒ paths(z, y) map (p ⇒ x :: p));
```

Notez la concaténation des différentes solutions, exprimée par l'application de la fonction *flatMap*.



Notez aussi qu'il nous faut convertir la liste des successeurs en un flot, de manière à assurer la génération paresseuse des solutions.

La fonction *list2stream* est définie comme suit.

```
def list2stream[a](xs: List[a]): Stream[a] = xs match {
  case List() ⇒ Stream.empty
  case x :: xs1 ⇒ Stream.cons(x, list2stream(xs1))
}
```

# La fonction d'interprétation

On revient maintenant à la fonction principale de l'interpréteur.

Sa tâche est de résoudre une requête *query* donnée, étant donné un programme consistant en une liste *clauses* de clauses.

Les solutions prennent la forme d'un flot de substitutions.

Interprétation : Pour chaque substitution $s$ du flot, l'instance de la requête obtenue par *query map s* peut être dérivée à partir des *clauses*.

```
def solve(query: List[Term], clauses: List[Clause]): Stream[Subst] = {
  def solve1(query: List[Term], s: Subst): : Stream[Subst] = ...
  solve1(query, List())
}
```



# Le coeur de l'interpréteur

L'implantation de la fonction *solve* est basée sur une fonction imbriquée *solve1*, qui prend deux paramètres :

- La liste des prédicats *query* qu'il reste à résoudre.
- La substitution courante, qui doit être appliquée à tous les termes, dans la requête et les clauses.

Voici son implantation.

```
def solve1(query: List[Term], s: Subst): Stream[Subst] = query.match {
  case List() ⇒
    Stream.cons(s, Stream.empty)
  case q :: query1 ⇒
    for (val clause ← list2stream(clauses);
         val s1 ← tryClause(clause.newInstance, q, s);
         val s2 ← solve1(query1, s1)) yield s2
}
```

La compréhension ***for*** exprime que, pour résoudre une requête $q :: query1$ :

- On essaie de faire correspondre toutes les clauses (dans l'ordre).
- Pour chaque clause, on essaie de résoudre le premier prédicat $q$ de la requête en utilisant une nouvelle instance de la clause.
- Pour chaque succès dans l'étape précédente, on continue en résolvant le reste *query1* de la requête.

La fonction *tryClause* est définie comme suit.

```
def tryClause(c: Clause, q: Term, s: Subst): Stream[Subst] =
    unify(q, c.lhs, s) match {
        case Some(s1) ⇒ solve1(c.rhs, s1)
        case None ⇒ Stream.empty
    }
}
```

Autrement dit,

- si le prédicat but s'unifie avec la tête de la clause, on continue en résolvant la partie droite de la clause,
- sinon on échoue.



# Création de nouvelles instances

Une clause peut être utilisée plusieurs fois dans une dérivation.

Par exemple, voici un programme Prolog qui recherche les chemins dans un graphe.

```
successor(a, b).
successor(b, c).
path(X, X).
path(X, Y) :- successor(X, Z), path(Z, Y).
```

Pour construire le chemin entre a et $c$, on a besoin d'instancier l'axiome *successor* à *successor*$(a, b)$ et *successor*$(b, c)$.

C'est pourquoi *solve1* doit créer une *instance fraîche* d'une clause avant de l'utiliser dans une dérivation.

Cela conduit à l'implantation suivante de la classe *Clause*.

# Classe *Clause*

```
case class Clause(lhs: Term, rhs: List[Term]) {
  def tyvars =
    (lhs.tyvars ::: (rhs flatMap (t ⇒ t.tyvars))).removeDuplicates;
  def newInstance = {
    var s: Subst = List();
    for (val a ← tyvars) do { s = Binding(a, newVar(a)) :: s }
    Clause(lhs map s, rhs map (t ⇒ t map s))
  }
}
```



# Traitement de la négation

Le constructeur *not* doit être traité de façon spéciale.

*not*(*P*) réussit ssi *P* échoue.

Cela est exprimé par la clause suivante dans *solve1* :

```
def solve1(query: List[Term], s: Subst): Stream[Subst] = query.match {
  ...
  case Constr("not", qs) :: query1 ⇒
    if (solve1(qs, s).isEmpty) solve1(query1, s)
    else Stream.empty
  ...
}
```

# La programmation logique est-elle logique ?

On peut voir Prolog comme une approximation de la programmation avec la logique mathématique.

Autrement dit, étant donnée une requête $q$, nous aimerions que l'interpréteur retourne une substitution $s$ telle que $q\ map\ s$ soit une conséquence logique du programme.

Le programme lui-même peut être vu comme un ensemble d'axiomes et de règles, dans lesquelles :− dénote l'implication renversée $\Leftarrow$.

Quand cela marche, cela nous donne une notation bien fondée et très flexible dans laquelle formuler des programmes et des requêtes.

De plus, tout ingénieur connaît la logique (ou devrait la connaître), si bien que le paradigme de programmation est facile à apprendre.



# Non correction et incomplétude de la programmation logique

Cependant, l'exécution de l'interpréteur ne nous donne qu'une approximation de cet idéal.

Il y a plusieurs failles :

- Parfois l'interpréteur donne une réponse qui n'est pas une solution.
- Parfois l'interpréteur échoue à trouver une solution qui existe.

La première limitation est appelée *non correction*, la seconde *incomplétude*.

# Un problème lié à la non correction

La plupart des interpréteurs Prolog ne sont pas corrects, car ils omettent le test occurrence pendant l'unification, si bien que de fausses solutions sont trouvées.

**Exemple :** Pour tester si deux termes sont les mêmes, on utilise l'axiome :

$same(X, X).$

Maintenant, pour caractériser tous les nombres qui sont égaux à leur successeur :

$strangeNum(X) :- same(X, succ(X)).$

L'exécution du test *strangeNum* avec un nombre concret échoue toujours.



Cependant, si l'on veut découvrir s'il existe des nombres étranges dans un interpréteur Prolog standard, on obtient habituellement une solution :

$prolog> ?\ strangeNum(X).$
$[X = succ(X)].$

Ce problème peut être surmonté facilement en incluant un test d'occurrence dans la fonction d'unification (ce que nous avons fait dans l'interpréteur Prolog écrit en Scala).

# Incomplétude

Il peut arriver que des solutions à une requête existent, mais que l'interpréteur échoue cependant à les trouver.

**Exemple :** Reprenons notre programme de "recherche d'un chemin" et introduisons un cycle dans le graphe.

$successor(a,\ b).$
$successor(b,\ a).$
$successor(b,\ c).$
$path(X,\ X).$
$path(X,\ Y) :- successor(X,\ Z),\ path(Z,\ Y).$

Maintenant la requête

$prolog> path(a,\ c)$

conduit à un dépassement de pile au lieu d'une solution.

En examinant la fonction d'interprétation *solve1*, pourriez-vous voir ce qui se passe mal ?



# Problème avec la recherche en profondeur

L'interpréteur applique une recherche en "profondeur d'abord" pour trouver une solution.

Autrement dit, quand la tête d'une clause correspond, l'interpréteur va essayer immédiatement de prouver la précondition de la clause.

Dans notre cas, on obtient la chaîne de buts suivants à prouver :

$path(a,\ c)$
$successor(a,\ b),\ path(b,\ c)$
$path(b,\ c)$
$successor(b,\ a),\ path(a,\ c)$
$path(a,\ c)$
...

... et ainsi de suite jusqu'au dépassement de la pile.

# Problèmes avec la négation

Le traitement du *not* pose aussi des problèmes, en relation à la fois avec la correction et avec la complétude.

Un prédicat $not(P)$ est considéré comme vrai dans Prolog si $P$ ne peut être prouvé.

On appelle cela l'*hypothèse du monde fermé* : tout ce qui n'est pas dérivable est supposé faux.

Pour les requêtes sans variable, cela peut être une hypothèse raisonnable.

Mais pour les requêtes avec variables, le résultat est parfois étrange.



**Exemple :** Considérons le petit programme Prolog suivant concernant les aliments :

```
vegetable(bean).
vegetable(tomato).
food(hamburger).
junkFood(X) :- not(vegetable(X)).
healthy(X) :- not(junkFood(X)).
```

Clairement, cela implique que les hamburgers sont des aliments pauvres.

```
prolog> ?junkFood(hamburger).
yes
```

D'un autre côté, si on demande un aliment pauvre quelconque $X$, on obtient :

```
prolog> ?junkFood(X).
no
```

Et ceci, même si on restreint la recherche :

```
prolog> ?junkFood(X), same(X, hamburger).
no
```

Cet exemple met en évidence un problème vis-à-vis de la complétude : l'interpréteur échoue à trouver une solution qui est pourtant une conséquence logique du programme.



En utilisant un *not* additionnel, on peut transformer la perte de la complétude en un perte de correction.

D'un côté on obtient :

*prolog*> *?healthy* (*hamburger* ).
*no*

Mais en reformulant la requête, on obtient une "solution" :

*prolog*> *?healthy* (*X* ), *same* (*X*, *hamburger* ).
[*X* = *hamburger*]

Comme le fait que *hamburger* est bon pour la santé n'est pas une conséquence logique du programme, on a un cas de non correction.

# Résumé

Durant les 14 dernières semaines, nous avons abordé un large éventail d'approches de la programmation.

- Fonctionnelle – programmer en composant des fonctions.
- Impérative – programmer en affectant un état modifiable.
- Logique – programmer en recherchant les conséquences de faits et de règles.

Sont orthogonaux à ces trois paradigmes :

- Les systèmes de types
  - Typage statique (par ex. Scala) contre typage dynamique (par ex. Lisp, Prolog)
  - Formes de polymorphisme : généricité, sous-typage.



- L'Orientation objet
  - Classes et objets
  - Héritage
- L'Ordre d'évaluation
  - Appel par valeur et appel par nom.
  - Listes, flots et itérateurs comme abstractions de séquences.

C'est une boîte à outil plutôt riche de concepts et de techniques.

Cela offre la possibilité de programmer à un plus haut niveau, et en particulier de concevoir des bibliothèques de haut niveau.

De l'ensemble, deux concepts principaux ont émergés :

- l'abstraction : la capacité d'isoler un concept, de le nommer, de ne plus utiliser que son nom, en cachant les détails de l'implantation.
- la composition : se focaliser sur les combinateurs pour construire de nouvelles abstractions, plutôt que sur les primitives elles-mêmes. Cela donne un grand pouvoir (qui implique de grandes responsabilités) à l'utilisateur de vos abstractions.

Alors, en tant que programmeur compétent, recherchez des abstractions qui peuvent être composées !

J'espère que vous aurez trouvé le cours intéressant.