Semaine 13: Programmation logique (1/2)

La plupart des calculs sont dirigés, allant d'une entrée vers une sortie.

En programmation fonctionnelle, cela est rendu très explicite, car l'entrée est l'argument de la fonction et la sortie le résultat de la fonction.

Nous avons déjà vu une exception, la résolution de contraintes.

Nous avions défini un ensemble de relations, que l'ordinateur pouvait "résoudre" dans plusieurs directions.

La programmation logique ajoute deux idées à ce paradigme de programmation relationnel :

- L'idée qu'une solution est trouvée par une *recherche* qui peut essayer plusieurs alternatives.
- Une sorte de filtrage de motif symbolique appelé *unification*.

Prolog

Le langage de programmation logique le plus répandu est *Prolog*.

Prolog est un acronyme pour "Programmer avec la Logique".

Il a été développé dans les années 70 par Alain Colmerauer, au départ pour l'analyse syntaxique du langage naturel.

Prolog est utilisé dans des applications d'intelligence artificielle, telles que les systèmes experts, les bases de connaissances, l'analyse syntaxique du langage naturel.

Tout comme Lisp, Prolog est un petit langage avec une syntaxe simple et sans types statiques.

Les deux implémentations suivantes sont gratuites et disponibles sur différentes plateformes.

- GNU Prolog (http://gnu-prolog.inria.fr)
- SWI-Prolog (http://www.swi-prolog.org)

Exemple : *append*

La fonction *append* s'écrit ainsi en Scala.

```
def append[a](xs: List[a], ys: List[a]): List[a] = xs match {
    case Nil => ys
    case x :: xs1 => x :: append(xs1, ys)
}
```

Cette fonction peut être vue comme une traduction en Scala des deux règles suivantes.

1. Pour toute liste *ys*, concaténer la liste vide et *ys* donne *ys*.
2. Pour tous *x*, *xs1*, *ys*, *zs*, si concaténer *xs1* et *ys* donne *zs*, alors concaténer *x* :: *xs1* à *ys* renvoie *x* :: *zs*.

En Prolog ces deux règles peuvent s'écrire comme suit.

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```


Remarques :

- Les variables et les paramètres en Prolog commencent par une lettre majuscule, par ex. *X*, *Xs*, *Ys*.
- [...|...] est le "cons" des listes, par ex. [*X*|*Xs*] s'écrit *X* :: *Xs* en Scala.

Prédicats

En Prolog, *append* s'appelle un *prédicat*.

Un prédicat n'est rien d'autre qu'une procédure qui peut réussir ou échouer.

Notez que le "résultat de la fonction" en Scala devient maintenant un paramètre additionnel.

Clauses

Les prédicats sont définis par des *clauses*, qui peuvent être des *faits* (aussi appelés axiomes) ou des *règles*.

- $append([\,], Ys, Ys)$ est un fait; il établit que concaténer $[\,]$ et Ys donne Ys (pour tout Ys).
- $append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs)$ est une règle; elle stipule que concaténer $[X|Xs]$ et Ys donne $[X|Zs]$, *pourvu que* concaténer Xs et Ys donne Zs (pour tous X, Xs, Ys, Zs).

Donc, :– peut se lire comme une implication de droite à gauche $\Leftarrow$.

Toute clause se termine par un point ('.').

Requêtes

Une requête Prolog est un prédicat qui peut contenir des variables comme paramètres.

L'interpréteur Prolog va tenter de trouver une affectation des variables qui rend le prédicat vrai.

Par exemple, l'appel à $append(List(1), List(2, 3))$ en Scala serait modélisé par

$$append([1], [2, 3], X)$$

en Prolog. Cela conduirait à la réponse $X = [1, 2, 3]$.

Mais il est aussi possible de mettre des variables à d'autres endroits.

Par exemple,

- $append(X, [2, 3], [1, 2, 3])$ renvoie $X = [1]$.
- $append([1, 2], Y, [1, 2, 3])$ renvoie $Y = [3]$.

- *append*(X, Y, [1, 2, 3]) renvoie plusieurs solutions :

  $X = [\,], Y = [1, 2, 3]$, *or*
  $X = [1], Y = [2, 3]$, *or*
  $X = [1, 2], Y = [2]$, *or*
  $X = [1, 2, 3], Y = [\,]$.

- *append*([1], Y, Z] renvoie un schéma de solutions qui contient une variable : $Y = X, Z = [1|X]$.

Cette stratégie, quand elle fonctionne, peut être très flexible.

Cela ressemble beaucoup aux langages de requêtes pour les bases de données.

En fait, Prolog est souvent utilisé comme langage pour extraire de l'information d'une base de données en particulier quand la déduction est aussi nécessaire.

Extraction d'information déductive

Voici une petite base de données représentant un arbre généalogique.

```
female(mary).          married(fred, mary).
female(ann).           married(peter, elaine).
female(elaine).        married(tom, sue).
female(jane).          married(alfred, ann).
female(sue).
female(jessica).

child(bob, fred).      child(sue, fred).      child(jessica, ann).
child(bob, mary).      child(sue, mary).      child(jessica, alfred).
child(peter, fred).    child(jane, sue).      child(paul, jerry).
child(peter, mary).    child(jane, tom).      child(paul, jane).
```

On peut accéder à l'information contenue dans la base de données en formulant une requête.

Une requête commence par un point d'interrogation suivi d'un prédicat.

Voici la retranscription d'une session avec un petit interpréteur Prolog écrit en Scala :

```
prolog> ?child(bob, fred).
yes
prolog> ?child(bob, bob).
no
prolog> ?child(bob, X).
[X = fred]
prolog> ?more.
[X = mary]
prolog> ?more.
no
prolog> ?child(X, bob).
no
```

La requête spéciale *?more.* demande des solutions supplémentaires pour la requête précédente.

On peut aussi définir des règles pour dériver des faits qui ne sont pas codés directement dans la base de données. Par exemple :


```
prolog> sibling(X, Y) :- child(X, Z), child(Y, Z).
```

Alors :

```
prolog> ?sibling(peter, bob).
yes
prolog> ?sibling(bob, jane).
no
prolog> ?sibling(bob, X).
[X = peter]
prolog> ?more.
[X = sue]
prolog> ?more.
[X = bob]
prolog> ?more.
[X = peter]
prolog> ?more.
[X = sue]
prolog> ?more.
no
```

Question : Pourquoi chaque frère (ou soeur) apparaît-t-il deux fois dans les solutions ?

La requête précédente ne correspond pas à ce que l'on veut, car *bob* apparaît comme son propre frère.

On peut corriger cela en définissant :

$$prolog\!> sibling(X, Y) :- child(X, Z), child(Y, Z), not(same(X, Y)).$$

Ici, le prédicat *same* est simplement défini par

$$same(X, X).$$

L'opérateur *not* est spécial (et quelque peu problématique!) en Prolog.

$not(P)$ réussit si le prédicat original P échoue.

Par exemple, pour définir qu'une personne est un homme, on peut utiliser :

$$male(X) :- not(female(X)).$$

Règles récursives

Les règles peuvent aussi être récursives.

Par exemple, pour définir que X est un ancêtre de Y :

$$parent(X, Y) :- child(Y, X).$$
$$ancestor(X, Y) :- parent(X, Y).$$
$$ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).$$

La possibilité de définir des règles récursives distingue la programmation logique des langages de requête de base de données.

Exercise: Définir le prédicat "X est un oncle de Y".

Exercise: Définir le prédicat "X et Y sont cousins".

Implantation de Prolog

L'interpréteur Prolog paraît avoir un fond d'"intelligence".

Nous allons maintenant découvrir comment cela est réalisé.

Il y a essentiellement deux ingrédients principaux :

- Un mécanisme de filtrage de motif qui est basé sur l'unification.
- Un mécanisme de recherche de dérivations.

Représentation des termes

On peut représenter les termes Prolog par une classe *Term* avec deux sous-classes, *Var* pour les variables et *Constr* pour les constructeurs.

```
trait Term {
    def tyvars: List[String] = ...
    def map(s: Subst): Term = ...
}
case class Var(a: String) extends Term;
case class Constr(a: String, ts: List[Term]) extends Term;
```

Par exemple, la variable *X* est représentée par

```
Var("X")
```

Et le terme *cons(X, nil)* est représenté par

```
Constr("cons", List(Var("X"), Constr("nil", List())))
```

Prolog a aussi du sucre syntaxique pour les termes représentant des listes, qu'on peut traduire de la manière suivante.

$$
\begin{array}{ll}
[\,] & = nil \\
[S|T] & = cons(S,\ T) \\
[S] & = cons(S,\ nil) \\
[T_1,\ ...,\ T_n] & = cons(T_1,\ ...\ cons(T_n,\ nil)\ ...\ )
\end{array}
$$

La classe *Term* définit deux méthodes.

- *tyvars* retourne la listes de tous les noms de variables de type du terme.
- *map* applique une substitution au terme (voir ci-dessous).

Filtrage de motif simple

Confronté à une requête telle que *?child*(*peter*, *X*), l'interpréteur tente de trouver un fait dans la base de donnée qui correspond (*match*) à la requête.

Faire correspondre signifie : assigner des termes aux variables de la requête de telle manière que requête et fait soient identiques.

Dans notre exemple, les affectations [*X* = *fred*] ou [*X* =*mary*] joueraient, vu que *child*(*peter*, *fred*) et *child*(*peter*, *mary*) sont des faits de la base de données.

Les affectations de variables (ou : substitutions) sont modélisés par des listes de liaisons.

Chaque liaison associe un terme à un nom de variable :

```
type Subst = List[Binding];
case class Binding(name: String, term: Term);
```

On peut définir une fonction *lookup* qui recherche une liaison impliquant un nom donné dans une substitution :

```
def lookup(s: Subst, name: String): Option[Term] = s match {
  case List() ⇒ None
  case b :: s1 ⇒ if (name == b.name) Some(b.term)
                 else lookup(s1, name)
}
```


Substitutions comme fonctions

La fonction *map* applique une substitution à un terme.

On la définit comme suit.

```
class Term {
  def map(s: Subst): Term = this match {
    case Var(a) = lookup(s, a) match {
      case Some(b) ⇒ b map s
      case None ⇒ this;
    }
    case Constr(a, ts) ⇒ Constr(a, ts map (t ⇒ t map s))
  }
  ...
}
```

Autrement dit, on peut voir les substitutions comme des fonctions idempotentes ($\forall t,\ \sigma(\sigma(t)) = \sigma(t)$) sur les termes, qui se confondent avec l'identité sauf pour un nombre fini de variables.

Les fonctions de *filtrage*

Maintenant, on peut implanter l'algorithme de filtrage au moyen de deux fonctions ***match***.

Voici la première :

```
def match(pattern: Term, term: Term, s: Subst): Option[Subst] =
  Pair(pattern, term) match {
    case Pair(Var(a), _) => lookup(s, a) match {
      case Some(term1) => match(term1, term, s)
      case None => Some(Binding(a, term) :: s)
    }
    case Pair(Constr(a, ps), Constr(b, ts)) if a == b =>
      match(ps, ts, s)
    case _ => None
  }
```


Explications : La fonction ***match*** prend en arguments :

- un motif (c-à-d. un terme contenant des variables),
- un terme (supposé sans variable), et
- une substitution qui contient l'affectation des variables déjà déterminée.

Si le filtrage réussit, la fonction renvoie comme résultat *Some*(s) où s est une substitution.

Si le filtrage échoue, elle renvoie comme résultat la constante *None*.

L'algorithme de *filtrage* procède par un filtrage de motif sur les paires composées d'un motif et d'un terme.

- Si le motif est une variable, on doit d'abord vérifier si la variable n'a pas déjà été affectée.
- Si c'est le cas, on doit continuer le filtrage avec le terme qui était affecté à la variable.
- Sinon, on étend la substitution avec une liaison qui associe le nom de

la variable avec le terme.

- Si le motif et le terme ont tous deux le même constructeur en tête, le filtrage de motif procède récursivement avec leurs éléments, en utilisant la deuxième procédure de filtrage.

Exercise: Implanter la seconde procédure de filtrage de motif, dont la signature est :

$$\textbf{\textit{def match}}\,(patterns\colon List\,[Term],\ terms\colon List\,[Term],\ s\colon Subst\,)\ \colon Option\,[Subst\,]$$

Cette fonction doit retourner $Some\,(s1\,)$ où $s1$ est une substitution qui étend s et qui fait correspondre les motifs correspondants avec les termes.

Elle doit retourner *None* si aucune substitution semblable n'existe, ou si les deux listes n'ont pas la même taille.

Unification

L'algorithme de filtrage de motif marche bien pour extraire des faits, mais échoue pour les règles.

En effet, la partie gauche d'une règle (ou : tête) peut elle-même contenir des variables.

Pour filtrer une règle, il faut affecter des variables à la fois dans la tête d'une règle et dans la requête.

Par exemple, étant donnée la règle

$$sibling\,(X,\ Y\,) :-\ child\,(X,\ Z\,),\ child\,(Y,\ Z\,),\ not\,(same\,(X,\ Y\,)\,).$$

et la requête $?sibling\,(peter,\ Z\,)$, on doit faire correspondre $sibling\,(X,\ Y\,)$ à $sibling\,(peter,\ Z\,)$, ce qui conduit à l'affectation $[X = peter,\ Y = Z]$ (ou bien : $[X = peter,\ Z = Y]$).

L'algorithme de filtrage de motif doit être généralisé pour le rendre symétrique.

L'algorithme résultant est appelé *unification*.

Unifier deux termes x et y signifie trouver une substitution s telle que x *map* s and y *map* s soient égaux.

Example: Voici quelques exemples d'unification.

```
unify(sibling(peter, Z), sibling(X, Y))  = [X = peter, Z = Y]
unify(same(X, X), same(mary, Y))         = [X = mary, Y = mary]
unify(cons(X, nil), cons(X, Y))          = [Y = nil]
unify(cons(X, nil), cons(X, a))          = <failure>
unify(X, cons(1, X))                     = <failure>
```

Le dernier cas est un peu subtil. Ici, l'unification échoue car il n'existe pas de terme fini T tel que $T = cons(1, T)$.

Cependant, il existe un terme *infini* qui satisfait l'équation, à savoir le terme qui représente une liste infinie de *1*.

Les interpréteurs Prolog normaux ne calculent que les termes finis composés de variables et de constructeurs (appelés *termes de Herbrand*, d'après le logicien Jacques Herbrand (1908-1931)).

Implantation de *unify*

```
def unify(x: Term, y: Term, s: Subst): Option[Subst] = Pair(x, y) match {
  case Pair(Var(a), Var(b)) if a == b =>
    Some(s)
  case Pair(Var(a), _) => lookup(s, a) match {
    case Some(x1) => unify(x1, y, s)
    case None => if ((y map s).tyvars contains a) None
                 else Some(Binding(a, y) :: s)
  }
  case Pair(_, Var(b)) =>
    unify(y, x, s)
  case Pair(Constr(a, xs), Constr(b, ys)) if a == b =>
    unify(xs, ys, s)
  case _ => None
}
```

Comme pour le filtrage de motif, on implante une version *incrémentale* de *unify*, où une substitution intermédiaire est passée comme troisième paramètre.

Les principaux changements par rapport au filtrage de motif sont :

- On doit maintenant tester le cas où les deux côtés sont la même variable. Dans ce cas, l'unification réussit avec la substitution donnée.
- Le cas où un côté est une variable a été dupliqué pour le rendre symétrique.
- On vérifie qu'une variable n'apparaît pas dans le terme auquel elle est liée, de façon à interdire les arbres infinis.

  (Ce qu'on appelle souvent le *test d'occurrence*).

Complexité de l'unification

Sans le test d'occurrence, la complexité de l'unification est linéaire en la taille des deux termes à unifier.

Cela peut paraître surprenant, car la taille du résultat de l'unification peut être exponentielle en la taille des termes !

Example: Unifier

```
seq(X1, b(X2, X2), X2, d(X3, X3))
seq(a(Y1, Y1), Y1, c(Y2, Y2), Y2)
```

donne

```
seq(a(b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3))),
      b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3)))),
    b(c(d(X3, X3), d(X3, X3)), c(d(X3, X3), d(X3, X3))),
    c(d(X3, X3), d(X3, X3)),
    d(X3, X3))
```

Il est facile de voir que, quand on étend la séquence, le premier terme de l'unificateur grandit exponentiellement avec la taille de la séquence.

Cependant, l'unification reste linéaire, car elle partage les arbres au lieu de les copier.

Complexité du test d'occurrence

Avec le test d'occurrence tel qu'il est implanté, l'unification devient dans le pire des cas exponentielle en la taille de son entrée, car les sous-arbres sont traversés de multiples fois.

En marquant les sous-arbres qui ont déjà été visités, on peut accélérer le test d'occurrence pour le rendre linéaire en la taille du graphe du terme.

L'unification devient alors un algorithme quadratique.

Il est possible de faire encore mieux, et de faire de l'unification un algorithme en $O(n\ log\ n)$.

...Et la triste réalité

Malheureusement, la plupart des interpréteurs Prolog laissent tout simplement de côté le test d'occurrence pour des raisons d'efficacité.

Cela serait cohérent si l'univers du discours était simplement étendu aux arbres infinis.

C'est l'approche retenue par un successeur du Prolog standard appelé Prolog 3 (également développé par Colmerauer).

Mais la plupart de interpréteurs Prolog se comportent de façon imprévisible quand on assigne à une variable un terme qui contient cette variable.

Par exemple, ils peuvent boucler indéfiniment.

De plus, des interpréteurs différents se comportent de manière différente.

Résumé

La programmation logique recherche des solutions à des requêtes, étant donné un programme composé de faits et de règles.

Le calcul n'a pas besoin d'être dirigé ; souvent une règle peut être utilisée de différentes manières avec des paramètres d'entrée et de sortie.

Un interpréteur pour la programmation logique tente d'appliquer des règles pour résoudre une requête, en utilisant l'unification comme étape d'inférence de base.

La semaine prochaine :

- Comment rechercher les solutions.
- Comment tout cela est (ou n'est pas ?) en relation avec la logique.