

Scheme - Projet

Université Paris I. Maîtrise de Logique. Année 2004-2005.
Nicolas Stroppa - stroppa@enst.fr

1 Introduction

1.1 Conseils

Les conseils donnés pour le projet Prolog sont bien sûr toujours d'actualité. Il s'agit donc d'écrire un code *lisible*, *commenté* et *argumenté* (vous devez être capable d'expliquer vos choix).

Une très bonne documentation est disponible en ligne pour Scheme. À consulter sans modération.

- La documentation de DrScheme: <http://www.drscheme.org/docs.html>
 - *The Scheme Programming Language* <http://www.scheme.com/tspl3/>
 - *How to Use Scheme* <http://www.htus.org/Book/2001-11-13/howto.html>
 - Des ressources: <http://www.schemers.org/>
- Des ouvrages généraux utilisant Scheme.
- L'incoutournable *Structure and Interpretation of Computer Programs*: <http://mitpress.mit.edu/sicp/full-text/book/book.html>
 - *How to Design Programs* <http://www.htdp.org/2003-09-26/Book/>

1.2 Le sujet

Le but du projet est d'étendre le mini-système expert étudié dans le TP 3, en essayant de s'affranchir de certaines de ses limitations. Deux directions permettent l'extension: (i) la nature des règles (e.g. autoriser des formules plus complexes qu'une simple conjonction), (ii) la nature du moteur d'inférence (e.g. utiliser une autre technique que le chaînage avant du TP 3).

Avant de commencer le projet, relisez la solution du TP 3 et assurez-vous de l'avoir bien assimilée.

2 Étendre les règles

2.1 Représentation des règles

Dans le TP 3, les conditions d'une règle correspondaient à une disjonction de condition. Ici, nous cherchons à modéliser des conditions un peu plus com-

plexes. Par exemple, on peut vouloir représenter des disjonctions de conjonctions de conditions (voire plus compliqué). Une fois que vous avez choisi le type de règles que vous voulez manipuler, convenez d'un mode de représentation pour ces règles, à l'aide de listes, ou de structures (`define-struct`, etc.). Songez à étendre au cas où les règles impliquent une liste de conséquences. Au même titre que les fonctions `si` et `alors` facilitent l'écriture des règles (notion de lisibilité), vous pourrez ajouter les connecteurs `et` et `ou`.

2.2 Modification des fonctions

Modifier les fonctions de votre programme de façon à prendre en compte cette nouvelle représentation. Pour cela, identifiez bien dès le départ le rôle de chaque fonction.

3 Étendre le moteur

Jusqu'à maintenant, nous avons procédé par *chaînage avant*, i.e. nous avons pris les règles une à une et testé si chaque règle étaient applicables étant donnée une base de faits.

3.1 Amélioration de la recherche des règles déjà appliquées

Pour voir si une règle est applicable, nous vérifions que la conséquence de la règle n'est pas déjà présente dans la base de faits. Une première amélioration consiste à associer un *identifiant* à chaque règle, et à construire un ensemble d'identifiants des règles déjà appliquées. Ainsi, à chaque application de règle, nous ajoutons l'identifiant de la règle à l'ensemble et avant d'appliquer une règle, nous vérifions que son identifiant n'est pas présent dans l'ensemble. Si l'identifiant est présent, la règle n'est pas appliquée. Vous pouvez utiliser la notion d'ensemble déjà étudiée pour modéliser l'ensemble des identifiants de règles appliquées. Une règle devra être désormais munie d'un identifiant (e.g. un entier): si la règle est représentée par une structure, ajouter un champ `id` à cette structure pour qu'une règle puisse contenir un identifiant.

Expliquer en quoi cela facilite la recherche des règles déjà appliquées.

3.2 Chaînage arrière

Le chaînage avant consiste à examiner une règle et à essayer de l'appliquer en regardant si ses conditions sont vérifiées. En *chaînage arrière*, on regarde la conséquence et on essaie de l'inférer en regardant les conditions qui permettent de la générer. On essaie alors de vérifier les conditions de la même façon, i.e. chaque condition est considérée comme un fait à vérifier. Si elle fait partie de la base de faits, alors elle est vérifiée. Sinon, on regarde si

elle se trouve comme conséquence de règles, auquel cas, on cherche à vérifier les conditions de ces nouvelles règles, etc., et ceci de manière récursive. Dans notre exemple, pour voir si l'on peut appliquer la règle

```
(make-regle (si '(cet animal est un carnivore)
  '(cet animal est de couleur fauve)
  '(cet animal a des taches noires))
  (alors '(cet animal est un guepard)))
```

nous cherchons à voir si les faits '(cet animal est un carnivore), '(cet animal est de couleur fauve), '(cet animal a des taches noires) peuvent être inférés. Pour cela, nous regardons les règles qui permettent de les engendrer. Par exemple, nous allons maintenant chercher à savoir si la règle

```
(make-regle (si '(cet animal est un mammifere)
  '(cet animal mange de la viande))
  (alors '(cet animal est un carnivore)))
```

(qui permet d'inférer '(cet animal est un carnivore)) peut être appliquée, etc.

Essayez d'implémenter un moteur d'inférence qui fonctionne en chaînage arrière. Attention, la version chaînage arrière est un peu plus compliquée. Pour l'implémentation en chaînage arrière, vous pouvez revenir à une description plus simple des règles (comme au TP 3 par exemple). Pour ne pas perdre ce que vous avez déjà fait jusqu'à maintenant, séparez bien les deux versions (en deux fichiers séparés par exemple).

Si vous n'arrivez pas à l'implémenter intégralement, essayez tout de même d'aller le plus loin possible, expliquez clairement où vous avez rencontré des difficultés, et suggérez des pistes pour résoudre les problèmes.

4 Le mot de la fin

Bon courage !