Université
catholique de
Louvain

Syllabus du cours

LINGI1101: Logique et Structures Discrètes

Titulaire :
Peter Van Roy

2014

Table des matières

Remerciements

Je tiens à remercier les étudiants de LINGI1101 pour avoir pris des notes pendant mon cours, ce qui faisait la base de ce syllabus.

Introduction

Ce document est le syllabus du cours LINGI1101 “Logique et Structures Discrètes” donné par Peter Van Roy.

1 Introduction à la programmation logique

1.1 Introduction à la programmation logique

Prolog est l'un des principaux langages de programmation logique. Il est à la base de nombreux fondements.

La programmation logique fait de la déduction sur les axiomes. On utilise la logique comme un langage de programmation : on va adapter l'algorithme de réfutation vu précédemment

Le programme (ressemble à une théorie) :
- Axiomes en logique des prédicats
- Une requête, un but (=goal)$\rightarrow$ le but du système est d'apporter une preuve
- Un prouveur de théorème $\rightarrow$ attention : il faut des conditions sur le prouveur car il faut être capable de prévoir le temps et l'espace utilisé par le programme.

Exécuter un programme = faire des déductions en essayant de prouver le but. Mais est-ce que cette idée peut donner un système de programmation pratique ?

Il y a une compromis entre expressivité et efficacité : Si c'est trop expressif, ça devient moins efficace, par contre si c'est trop peu expressif, on ne peut rien programmer, ça ne sert à rien non plus. Il faut donc être expressif tout en restant efficace. Le Prolog offre un bon équilibre entre expressivité et efficacité.

Mais pour arriver à cela, il y a quelques problèmes à surmonter :

a. un prouveur est limité :
   vérité = $p \models q$ (= q est vrai dans tous les modèles de p)
   preuve = $p \vdash q$
   $p \models q \Rightarrow p \vdash q$ (= Si c'est vrai dans tous les modèles, on peut trouver une preuve)
   Si $p \models q$ alors l'algorithme se terminera. Cependant, on ne peut pas trouver les preuves que pour des choses vraies dans tous les modèles. (Comme c'est impossible on ne prend qu'une partie des modèles, ce qui limite le programme).
b. Même si on peut trouver une preuve, le prouveur est peut-être inefficace (utilise trop de temps ou de mémoire) ou imprévisible. $\rightarrow$ On ne peut pas raisonner sur l'efficacité du prouveur.
c. La déduction faite par le prouveur doit être constructive

Si le prouveur affirme : $(\exists X)P(X)$ alors le prouveur doit donner une valeur de x (c'est quoi x).

Il faut construire un résultat.

Pour résoudre ces problèmes...

1. Restrictions sur la forme des axiomes.
   typiquement :

$$(\forall X_1)...(\forall X_n)A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow A \tag{1}$$

$$C_i \neg A_1 \vee \neg A_2 \vee ... \vee \neg A_n \vee A \tag{2}$$

   Il n'y a qu'un seul littéral sans négation. (Pour prouver A, il faut prouver A_1 ; A_2, ..., A_n).
   C_i est la clause. Le programme tout entier est une série de clauses :

$$C_1 \wedge C_2 \wedge ... \wedge C_k \tag{3}$$

2. Le programmeur va aider le prouveur. Par exemple : il faut commencer par prouver A_1 puis A_2... dans cet ordre là.
   → Le programmeur donnera des heuristiques. Attention : ces heuristiques ne changent pas la sémantique logique du programme. Elles ont seulement un effet sur l'efficacité !

$$C_1 = \neg A_1 \vee \neg A_2 \vee ... \vee A_n \vee A \tag{4}$$

$$C_2 = \neg B_1 \vee \neg B_2 \vee ... \vee B_k \vee A \tag{5}$$

Le langage Prolog utilise ces 2 ordres.

Bref historique :

1. 1965 : La règle de résolution a été inventée par A. Robinson
2. 1972 : Invention du langage Prolog / premier interprète (de Prolog) par A. Calmerauer, R. Kowalski et Ph. Roussel. Ils voulaient faire un langage de programmation logique, et connaissaient les différentes formes de logique ainsi que la résolution. Ils ont donc inventé un langage très simple qu'ils ont appelé Prolog (pour Programmation Logique). Il s'avère que ce langage a un compromis très intéressant par rapport à la tension entre efficacité et expressivité. Aujourd'hui, on peut faire une implémentation extrêmement efficace de Prolog. Il est extrêmement expressif, ce qui permet de faire des programmes complexes. C'est un langage à part entière.

1.2 Introduction à Prolog

En Prolog, on a des clauses (règles) :

$$A1 \leftarrow B1, \ldots, Bn \tag{6}$$

(On peut prouver A en prouvant $B1$ jusqu'à Bn. Remarque :$\leftarrow$ou :-)

$$\neg(B_1 \wedge ... \wedge B_n) \vee A_1 \tag{7}$$

$$(\neg B_1 \vee \neg B_2 \vee ... \vee \neg B_n \vee A_1) \tag{8}$$

Programme = ensemble de clauses.

Exemple d'un petit programme en Prolog : (extrait du livre « The Art of Prolog » par L. Sterling et E. Shapine)

Règle : $grandpere(x, z) \leftarrow pere(x, y), pere(y, z)$. ($x$, y,... sont des variables. En Prolog, elles sont souvent en majuscule)

Faits : $pere(terach, abraham)$ (terach, abraham,... sont des constantes)
$pere(abraham, isaac)$
$pere(haram, lot)$
...

$\rightarrow$ Syntaxe clausale : $pere(terach, abraham) \wedge pere(abraham, isaac) \wedge pere(haram, lot) \wedge (\neg pere(x, y) \vee \neg pere(y, z) \vee grandpere(x, z))$

Il existe une corrélation évidente entre Prolog et les bases de données. Elles ont été inventées quasi au même moment et aujourd'hui Prolog est utilisé comme une sémantique pour les bases de données déductibles. Ici, on peut avoir une relation avec deux colonnes qui auraient l'argument père. le grand-père serait une combinaison de ces deux relations.

Prolog peut être vu comme une sorte de base de données relationnelle mais beaucoup plus puissante : on peut faire des programmes qui sont plus que des simples requêtes, avec des calculs beaucoup plus complexes.

1.3 Algorithme d'exécution de Prolog

Dans la version de l'algorithme de preuve par résolution, l'ensemble S grandit (ce qui n'est pas très efficace).

L'idée :
- On commence par mettre le but (G) que l'on veut prouver dans r (sans négation).
- Ensuite, jusqu'à ce que r soit vide, on prend le premier littéral dans $r(A_1)$.

- Puis, on parcourt un à un les axiomes de P (P est le programme, la base de faits) pour trouver une clause Ax_i unifiable avec A_1 au moyen de $\sigma(u.p.g)$
  - Si on trouve une telle clause, on ajoute à r les littéraux de Ax_i après unification avec A_1 (et on recommence au début).
  - Si on ne trouve pas de clause unifiable, on revient sur le dernier choix (Par exemple, pour un A_1 qui aurait plusieurs clauses unifiables, on a dû en choisir une. Et bien, on retourne en arrière pour en choisir une autre, sans oublier de modifier r. (En effet, il faut éliminer les résultats de toutes les unifications qui ont été réalisées entre le moment où le point de choix a été mémorisé et le moment du retour en arrière.))
  - Si on épuise tous les choix sans que r soit vide alors nous sommes en cas d'échec.
- Lorsque le programme s'arrête, si r est vide, on a un résultat.

Programme : $P = Ax_1, ..., Ax_n$

Un « but » (un goal, une « requête ») G ($\simeq$ théorie)

$r :=< G > \dots$ résolvante (une séquence de littéraux $\rightarrow$
$r =< A_1, A_2, ..., A_m >$ Il n'y a qu'un seul r.)
while *r est non vide* **do**
- Choisir un littéral A_1 dans r (on prend le premier littéral)
- Choisir une clause $Ax_1 = (A \leftarrow B_1, ..., B_k)$ dans P. (D'abord on prend la première clause, puis la suivantes jusqu'à ce qu'on trouve une clause unifiable avec A_1. Si aucune clause n'est unifiable on revient sur le dernier choix (backtrack))
- Nouvelle résolvante $:=< B_1, ..., B_k, A_2, ..., A_m > \sigma$
- $G^{'} = G\sigma$

end
if *r est vide* **then**
le résultat est le dernier $G^{'}$
end
if *on épuise les choix sans que r soit vide* **then**
le résultat est NON. (On n'a pas prouvé G). (Attention : G est peut-être vrai, mais les heuristiques ne suffisent pas pour le prouver.)
end

On peut également avoir une boucle infinie (l'algorithme est non déterministe).

Conclusion

Pour conclure, avec LaTeX on obtient un rendu impeccable mais il faut s'investir pour le prendre en main.

Références

[Nis] Nimal Nissanke. *Introductory Logic and Sets for Computer Scientists.*

[LPP] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets : Reasoning About a Highly Connected World.*