Utilisation de Grammaires en Programmation Logique Inductive

Fabien Torre
Laboratoire de Recherche en Informatique,
Bâtiment 490, Université Paris-Sud,
91405 ORSAY Cedex
e-mail : fabien@lri.fr

Mots clés : Programmation Logique Inductive, Grammaires Attribuées, DCG, Biais.

Dans le cadre de l'apprentissage, la programmation logique inductive (PLI) se propose d'utiliser un formalisme de logique du premier ordre. Même si l'on se réduit le plus souvent aux clauses de Horn, ce formalisme possède un très grand pouvoir expressif ; en outre, il autorise l'utilisation d'une théorie du domaine et fournit des résultats parfaitement lisibles. Traditionnellement, un problème en PLI peut être formalisé comme suit [MR94].

> Étant donnés un ensemble $E+$ d'exemples et un ensemble $E-$ de contre-exemples pour le concept à apprendre, une théorie du domaine B, il s'agit de trouver une hypothèse H telle que
>
> $$B \cup H \models e+, \forall e+ \in E+ \quad (H \text{ est complète}),$$
> $$B \cup H \models e-, \forall e- \in E- \quad (H \text{ est correcte}).$$

Un problème majeur en PLI est posé par la taille très importante, parfois infinie, de l'espace de recherche et l'on appelle *biais* tout moyen de rendre la recherche plus efficace [Mit80]. On en distingue deux sortes selon qu'ils suppriment a priori certaines clauses de l'espace (biais de langage) ou qu'ils orientent la recherche en désignant les clauses les plus prometteuses (biais de recherche).

Les biais peuvent être totalement arbitraires ; dans ce cas il s'agit d'a priori sur le problème à résoudre (par exemple, on peut poser que les définitions cherchées ne soient pas de taille supérieure à 2). Naturellement, ces

biais peuvent aussi être liés à la connaissance du domaine : ainsi, il est possible de spécifier que tous les paramètres du concept (la tête de la clause) doivent être utilisés et donc apparaître dans le corps (on dit alors que la clause est *range-restricted*).

Si l'utilisation de biais permet la recherche dans un espace immense, elle amène de nouveaux problèmes : il faut prévoir l'influence de chaque biais en termes d'efficacité (l'élagage produit par un biais est-il suffisant pour rendre la recherche possible ?), et de qualité de l'apprentissage (l'espace élagué contient-il toujours une définition du concept ?).

Nous allons ici fournir une méthode qui permet de répondre à ces questions pour les biais qui s'expriment comme des propriétés que doit présenter la définition du concept cherché : il s'agit des biais de langage, auxquels on ajoute les exigences liées aux exemples, exigences qui correspondent aux notions (ou à des notions affaiblies) de la correction et de la complétude.

Notre tâche se divisise donc en deux parties. Tout d'abord, nous devons être capables de représenter un ensemble de clauses. C'est la méthode des grammaires [Coh94] que nous avons choisie pour cela (l'aménagement effectué pour pouvoir utiliser des grammaires hors-contexte en PLI est que les symboles sont des littéraux). D'autre part, nous devons maintenant, exprimer les effets des biais sur cet espace en termes d'efficacité et de qualité. Avec notre choix d'utiliser des grammaires, il s'agit de trouver les mots du langage d'une grammaire donnée vérifiant les propriétés choisies.

La technique la plus simple consiste à développer exhaustivement le langage de cette grammaire, puis tester chacune des clauses obtenues contre les propriétés choisies. Cette méthode n'est évidemment pas viable, en particulier dans le cas où la grammaire dénote un langage infini. L'idée est alors de mettre en place un moyen de communication à l'intérieur même des grammaires dans le but d'amener à chaque terminal des informations sur les littéraux le précédant et ainsi, ne pas énumérer toutes les clauses du langage.

Dans ce but, nous nous proposons d'exploiter un formalisme qui est couramment utilisé pour spécifier la syntaxe et la sémantique des langages de programmation : les grammaires attribuées. Il s'agit d'une extension des grammaires hors-contexte où l'on associe aux symboles de la grammaire des attributs, un attribut pouvant représenter une quantité quelconque. Les valeurs de ces attributs sont définies par des règles sémantiques associées à chaque règle de la grammaire. On distingue deux types d'attributs selon qu'ils sont définis en fonction de leurs descendants (synthétisés) ou de leurs ancêtres (hérités).

Pour vérifier les propriétés, nous associons un attribut hérité C_h et un attribut synthétisé C_s à chaque symbole (terminal ou non-terminal) de la grammaire hors-contexte. Ces attributs vont nous permettre de transporter des contextes, c'est-à-dire une description de la partie de la clause déjà construite, une description qui s'exprime en termes de satisfaction des propriétés voulues pour les définitions de concept.

Par exemple, considérons la propriété vraie pour les clauses de taille inférieure ou égale à n. Pour la vérifier, on va inscrire dans le contexte, le couple composé de n et du nombre de littéraux déjà présents dans le corps de la clause.

Intuitivement, un symbole S_i va opérer dans le contexte $S_i.C_h$ (contexte qui est hérité de ses prédécesseurs) ; si tout se passe bien, S_i informe ses successeurs de son travail à travers le contexte $S_i.C_s$. Plus formellement, pour chaque règle $S_0 \rightarrow S_1 \ldots S_n$ de la grammaire hors-contexte, on retrouve la règle sémantique suivante :

$$
\begin{array}{ll}
S_0.C_s = S_n.C_s & \\
S_1.C_h = S_0.C_h & \\
S_i.C_h = S_{i-1}.C_s & \text{pour } i > 1 \\
S_i.C_s = valide(S_i, S_i.C_h) & \text{pour } S_i \text{ terminal}
\end{array}
$$

Dans le cas d'une dérivation vide $S_0 \rightarrow$, on a simplement $S_0.C_s = S_0.C_h$.

La fonction *valide* teste chaque non-terminal vis-à-vis des propriétés présentes dans le contexte. Plus précisément, cette fonction va, pour chaque information I_j de $S_i.C_h$, considérer la propriété P_j, c'est-à-dire tester et mettre à jour I_j en fonction du nouveau littéral S_i. Dans le cas de la limitation de la taille des clauses à n, *valide* prend en entrée un contexte et y prélève le couple *(n,taille actuelle de l'hypothèse)* ; si cette dernière ne vaut pas déjà n, on lui ajoute 1 (pour prendre en compte l'ajout de S_i) et cette nouvelle valeur constitue alors le contexte synthétisé.

Si *valide* échoue c'est que l'ajout du littéral est incompatible avec le début de la définition par rapport à l'une des propriétés considérées et, par suite, cette combinaison est abandonnée. Grâce à ces grammaires attribuées, nous allons pouvoir stopper au plus tôt la construction d'une clause non satisfaisante.

D'un point de vue pratique, de telles grammaires (avec attributs et une instruction *valide*) peuvent être réalisées sous forme de DCG (toute implémentation de PROLOG possède un module capable de développer le langage dénoté par une DCG). Si l'on suppose donnée une grammaire hors-contexte, le passage à une DCG simulant une grammaire dotée d'une in-

struction *valide*, se fait en ajoutant deux variables à chaque non-terminal, la première représentant le contexte hérité, la seconde le contexte synthétisé. Les terminaux eux, ne sont pas modifiés mais sont suivis d'une instruction *valide*. Cette instruction récupère le dernier contexte et construit, si l'ajout est effectivement validé, le nouveau contexte (qui est donc propagé dans la suite de la règle).

Cette méthode de vérification des propriétés nous permet de réduire le coût de vérification d'une propriété. D'une part, pour une clause donnée, le test ne porte pas sur l'ensemble de ses littéraux mais sur le dernier ajouté et sur l'information représentant le début de la clause (le coût n'est donc pas dépendant de la taille de la clause). D'autre part, le résultat de ce test élémentaire circule à l'aide des attributs, et sera donc connu par toutes les hypothèses construites par ajout sur la clause testée.

Autre aspect positif, on ne considère que les clauses du langage vérifiant nos propriétés et les clauses ne les vérifiant pas construites par adjonction d'un unique littéral à une clause vérifiant les propriétés. En particulier, cela signifie que si l'on utilise une propriété qui n'est vraie que pour un ensemble fini de clauses, nous sommes capables d'isoler les clauses satisfaisantes d'un espace infini en n'observant qu'un nombre fini de clauses.

Ces considérations ne sont vraies que dans le cas où l'on est autorisés à abandonner définitivement une hypothèse ne satisfaisant pas l'une des propriétés. Autrement dit, nous n'allons tester que les propriétés telles que l'application de la dérivation sur une clause non satisfaisante ne peut amener que des clauses non satisfaisantes.

Le pas suivant consiste donc à caractériser à l'aide de nos grammaires les opérateurs habituellement utilisés. La grammaire elle-même va représenter l'espace potentiellement engendrable par l'opérateur considéré. Ensuite, il faut traduire le comportement de l'opérateur dans cet espace et cela à travers la stratégie de dérivation.

Ainsi la grammaire suivante, indique que l'on peut introduire n'importe quel littéral construit à partir d'un symbole de prédicat et des variables quelconques.

$$\begin{aligned} \text{concept}(\vec{V}) &\rightarrow \text{littéraux}(\vec{V}) \\ \text{littéraux}(L) &\rightarrow [] \\ \text{littéraux}(L) &\rightarrow \{\text{variables}(\vec{V}, L, NL)\}, [P(\vec{V})], \text{littéraux}(NL) \end{aligned}$$

Pour simuler l'ajout de FOIL [Qui90], il reste à indiquer que l'ajout ne peut se produire qu'en fin de clause : cela est réalisé si l'on dérive notre langage

en utilisant une stratégie gauche-droite, profondeur d'abord (la stratégie de PROLOG).

A terme, nous serons donc capable de construire automatiquement des systèmes PLI, simplement à partir de la donnée du problème, simulant tel ou tel autre système.

References

[Coh94] Cohen (W. W.). – Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, vol. 68, 1994, pp. 303–366.

[Mit80] Mitchell (T. M.). – The need for biases in learning generalizations. *In: Readings in Machine Learning*, pp. 184–191. – Morgan Kaufmann, 1980. Published in 1991.

[MR94] Muggleton (S.) et Raedt (L. De). – Inductive logic programming: Theory and methods. *Journal of Logic Programming*, vol. 19, 1994, pp. 629–679.

[Qui90] Quinlan (J. R.). – Learning logical definitions from relations. *Machine Learning*, vol. 5, n° 3, 1990, pp. 239–266.