La Métaprogrammation Logique. Un Outil pour Créer et Modifier de Grands Programmes.

Tristan Cazenave
LIP6, UPMC, 4, place Jussieu
75252 PARIS CEDEX 05, FRANCE
Tristan.Cazenave@lip6.fr

Résumé

La métaprogrammation logique est un outil puissant pour créer et modifier de grands programmes. Nous présentons brièvement la problématique de la métaprogrammation logique. Puis nous illustrons ses possibilités par des exemples tirés de notre système Introspect. Cette approche permet de mettre à jour et de transformer automatiquement de grands programmes.

Mots Clés : Métaprogrammation, Spécialisation de Programmes, Transformation de Programmes, Vérification de Cohérence.

Abstract

Logic metaprogramming is a powerful tool to create and modify large programs. We briefly present the goals of logic metaprogramming. And we illustrate its abilities by using examples of logic metaprogram used in our system Introspect. These tools enable to automatically update and transform large programs.

Key Words : Metaprogramming, Program Specialisation, Program Transformation, Integrity Checking.

Introduction

La métaprogrammation logique est un formalisme qui a de solides fondations mathématiques. Il a récemment été appliqué avec succès a des problèmes difficiles. Ces recherches s'incrivent dans le cadre des recherches sur la métaconnaissance [Pitrat 1990], elles sont liées aux recherches sur les bases de données, la programmation logique et la transformation automatique de programmes et de données.

La première section est une introduction à la métaprogrammation logique. La seconde section décrit le langage de métaprogrammation que nous avons utilisé dans notre système Introspect. La troisième partie montre comment ce langage a été utilisé pour créer de grands programmes. La quatrième partie explique comment les programmes peuvent être facilement mis a jour et modifiés en utilisant Introspect. Nous concluons en esquissant des voies prometteuses de recherches futures.

La Métaprogrammation Logique

Dans cette section nous définissons les termes et les enjeux de la métaprogrammation logique. La métaprogrammation (le fait d'écrire des programmes qui peuvent manipuler des représentations d'autres programmes) a des liens aussi bien avec l'informatique fondamentale qu'avec l'informatique appliquée. Des exemples de métaprogrammes sont les compilateurs, les interpréteurs, les debuggers, les analyseurs de programmes ou les évaluateurs partiels. Le choix de la programmation logique comme base de la métaprogrammation à des avantages à la fois pratiques et théoriques : entre autres, la possibilité d'aborder les fondements de la métaprogrammation dans un environnement théorique solide et la facilité de programmer en logique. Toutefois, les représentations classiquement utilisées en programmation logique doivent être étendues pour permettre la métaprogrammation.
Le langage de métaprogrammation logique le plus connu est sans doute le langage Gödel [Hill & Lloyd 1994]. Il permet de faire de la métaprogrammation grâce à des modules prédéfinis. Il a la possibilité de se représenter lui même et de se compiler lui-même.

Les quatre applications principales de la métaprogrammation [Barklund 1994] sont :

- Un formalisme qui permet d'écrire des outils de manipulation de programmes comme les compilateurs, les transformeurs de programmes, les debuggers et les interpréteurs abstraits.
- Des mécanismes pour contrôler l'exécution des programmes logiques.
- La représentation des connaissances, les raisonnements sur les connaissances, les raisonnements sur les raisonnements, etc.
- L'écriture d'interpréteurs pour un langage donné, ce qui permet de créer de nouveaux langages ou d'ajouter des fonctionnalités à un langage déjà existant.

Les outils qui nous intéresse plus particulièrement ici sont les programmes manipulateurs de programmes. Nous donnons un bref aperçu des différentes méthodes de manipulation de programmes actuellement utilisées :

La transformation de programmes est la réécriture automatique de programmes en de nouveaux programmes qui ont certaines propriétés voulues [Gallagher 1993]. La transformation de programme la plus connue et utilisée est le pliage/dépliage de programmes [Tamaki & Sato 1984].

L'évaluation partielle [Lloyd & Shepherdson 1991] connue aussi comme la déduction partielle permet de rendre un programme plus rapide en le dépliant et en le spécialisant sur certaines valeurs [Consel & Danvy 1993]. Elle est utilisée en métaprogrammation pour compiler les compilateurs et les interpréteurs [Jones & al 1993].

L'analyse de programmes a pour but de comprendre les propriétés d'un programme afin de le rendre plus rapide ou plus compact. Une méthode populaire d'analyse de programmes est l'interprétation abstraite [Cousot & Cousot 1992].

La métaprogrammation logique a été appliquée avec succès aux bases de données [Leuschel 1997], pour synthétiser des programmes efficaces de vérification des contraintes d'intégrité dans des bases de données déductives, à partir de programmes standards de vérification de la cohérence. Les métaprogrammes utilisés par Michael Leuschel permettent des gains en vitesse compris entre 98 et 918 par rapport aux programmes standard de vérification de cohérence.

Le Langage Introspect

Notre système, Introspect, est basé sur la logique des prédicats. Contrairement aux programmes Prolog, les programmes Introspect sont réellement déclaratifs : le résultat de leur exécution ne dépend pas de la stratégie de résolution. Prolog est fondé sur la stratégie SLDNF, ce qui amène à avoir un ordre entre les clauses qui nuit à la déclarativité d'un programme. Une stratégie de résolution fixe est nuisible lorsqu'on veut efficacement spécialiser un programme logique inefficace mais concis en un programme logique efficace mais plus long, elle empêche le réordonnancement des atomes à l'intérieur des clauses et conduit parfois à des programmes spécialisés faux. Pour des raisons similaires nous nous défendons d'utiliser la négation par l'absence. De plus nous autorisons plusieurs atomes en conclusion d'une règle.

Introspect utilise des métaprédicats qui lui permettent de manipuler ses propres programmes. Nous allons en décrire quelques uns dans la suite de cette section. Les variables et les constantes sont typées dans Introspect. Par convention, les variables commencent par des majuscules, alors que les constantes commencent par des minuscules. Chaque exemple de métaprédicat est suivi de sa définition.

Règle (R) : Instancie dans la variable R de type Règle toutes les règles du programme qui a été sélectionné.

Conclusion (R, P) : Instancie dans la variable P tous les prédicats en conclusion de la règle R.

Conclusion (R, Couleur (V1)) : Instancie dans la variable V1 toutes les variables et les constantes des conclusions de la règle R qui sont en argument des prédicats Couleur.

InitialiseCoupe () : Met a 0 la variable interne qui test si une déduction a été faite dans la règle. L'initialisation est faite pour chaque instanciation différentes des variables au dessus de ce métaprédicat.

CoupeSiDeduction () : N'effectue les instructions suivantes que si aucune déduction n'a encore été faite après que InitialiseCoupe () ait été appelé.

ListeCoupsModifiant (Liste, Liste1, Liste2) : Ajoute dans la liste Liste les coups qui permettent de changer la valeur d'un jeu gi (un jeu gi est une menace d'atteindre un but, les coups de Liste sont les coups forcés pour parer à une menace). Ces coups sont déterminés à partir de la liste des conditions Liste2, les conditions à ajouter sont dans la liste Liste1. Ce métaprédicat est un peu spécial puisqu'il fait appel à une base contenant les règles du jeu pour déterminer Liste et Liste1 à partir de Liste2.

Condition (R, CouleurOpposee (V1, V2)) : Instancie dans les variables V1 et V2 toutes les variables et les constantes des conditions de la règle R qui sont en argument des prédicats CouleurOpposee.

Introspect utilise ainsi de nombreux métaprédicats dont la signification peut être intuitivement comprise à partir de leurs noms, nous ne les définirons pas tous ici.

Créer de Grands Programmes

Les métaprogrammes peuvent être utilisés pour créer automatiquement des programmes. Nous donnons dans cette section un exemple de règle qui permet de créer d'autres règles dans Introspect.

```
OteCondition ( R1, CouleurJoueur ( V1 ) ),
AjouteCondition ( R1, CouleurJoueur ( V4 ) ),
AjouteConclusion ( R1, ListeCoupsBloc ( V, V4, vivre, V2, Liste, ip ) ),
AjouteRegle ( R1 ) :-
    Regle ( R ),
    Conclusion ( R, CoupBloc ( V, V1, Prendre, V2, V3, gi ) ),
    Condition ( R, Couleur ( V1 ) ),
    Condition ( R, CouleurOpposee ( V1, V4 ) ),
    ListeCondition ( R, Liste2 ),
    NouvelleRegle ( R1 ),
    ListeCoupsModifiant ( Liste, Liste1, Liste2 ),
    TailleListe ( Liste, N ),
    Superieur ( 5, N ),
    AjouteListeCondition ( R1, Liste1 ),
    AjouteListeCondition( R1, Liste2 ).
```

Figure 1

Cette règle est utilisée dans notre application d'Introspect aux jeux. Elle permet de créer les règles donnant les coups forcés pour parer à une menace à partir d'une règle concluant sur une menace. Les règles concluant sur les menaces sont elles-mêmes crées par Introspect. A partir des règles d'un jeu, Introspect est capable, grâce à un métaprogramme, de créer automatiquement un programme qui résout des problèmes tactiques pour ce jeu [Cazenave 1996b]. Dans son application au jeu de Go, Introspect a ainsi créé un programme d'un million de lignes à partir d'un métaprogramme de quelques centaines de lignes décrivant les règles du jeu de Go. Ce programme a terminé 6ème lors du tournoi organisé durant IJCAI'97 qui comptait 40 participants. Les cinq premiers programmes (et un certain nombre de ceux qui suivent) sont commerciaux et ont demandé de nombreuses personnes*années de travail.

Modifier de Grands Programmes

Les programmes engendrés automatiquement comprennent des centaines de milliers de lignes. Lorsqu'on modifie la représentation des connaissance, il est très utile et parfois indispensable d'avoir des outils qui permettent de modifier automatiquement tous les programmes d'une application pour les mettre en conformité avec la nouvelle représentation. Cette section montre comment nous modifions nos programmes objets (par opposition aux métaprogrammes) à l'aide de métaprogrammes. Nous décrivons deux modifications utilisées : la suppression de conditions inutiles dans les règles engendrées automatiquement et le changement de représentation.

Suppression de conditions inutiles

Introspect crée parfois des règles qui contiennent des conditions inutiles. Par exemple, la Figure 2 donne une règle qui trouve un chemin de longueur quatre entre deux intersections d'une grille sans passer deux fois par la même intersection. Après chaque nouvelle instanciation de variable dans les conditions, la règle vérifie que l'intersection instanciée est différente des intersections précédemment instanciées.

```
PreferPath ( X, Y ) :-
    CurrentState ( X ),       1
    Connected ( X, Y ),       4
    Different ( X, Y ),       4
    Connected ( Y, Z ),      16
    Different ( Z, X ),      12
    Different ( Z, Y ),      12
    Connected ( Z, W ),      48
    Different ( W, X ),      48
    Different ( W, Y ),      36
    Different ( W, Z ),      36
    Connected ( W, D ),     144
    Desired ( D ).
```

Figure 2

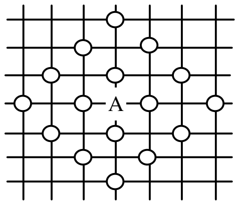

Figure 3

```
PreferPath ( X, Y ) :-
      CurrentState ( X ),          1
      Connected ( X, Y ),          4
      Connected ( Y, Z ),         16
      Different ( Z, X ),         12
      Connected ( Z, W ),         48
      Different ( W, Y ),         36
      Connected ( W, D ),        144
      Desired ( D ).
```

Figure 4

Toutefois, dans certains cas, il est inutile de verifier que des intersections sont différentes les unes des autres à cause de la topologie de la grille. Par example, deux intersections voisines sont toujours différentes. Nous utilisons des métarègles pour oter la condition ‘Different (V V1)’ si la condition ‘Connected (V V1)’ est présente dans la règle. Cette métarègle est donnée dans la Figure 5. Une autre métarègle donnée dans la figure 6 permet d’enlever cette même condition lorsqu‘il y a un chemin de longueur trois entre deux intersections. C’est une conséquence de la figure 3, qui montre toutes les intersections qui sont à une longueur trois de l’intersection A : elles sont toutes différentes de l’intersection A. La règle initiale de la figure 2 effectue 361 opérations élémentaires, après le passage des métarègles, on obtient la règle de la figure 4 qui n’effectue plus que 261 opérations élémentaires et qui donne exactement le même résultat.

```
OteCondition ( R, Different ( V, V1 ) ) :-
      Regle ( R ),
      Condition ( R, Connected ( V, V1 ) ),
      Condition ( R, Different ( V, V1 ) ).
```

Figure 5

```
OteCondition ( R, Different ( V, V3 ) ) :-
      Regle ( R ),
      Condition ( R, Connected ( V, V1 ) ),
      Condition ( R, Connected ( V1, V2 ) ),
      Condition ( R, Connected ( V2, V3 ) ),
      Condition ( R, Different ( V, V3 ) ).
```

Figure 6

Modification de la représentation

Il est important de pouvoir facilement modifier la représentation des connaissances. Par exemple, si un gros programme contient des milliers de règles qui concluent sur des propriétés d'une intersection et que nous changeons notre représentation pour prendre en compte des propriétés de groupes d'intersections. Il est intéressant de transformer automatiquement les règles qui concluent sur les propriétés d'une intersection en règles qui concluent sur des propriété d'un groupe d'intersections. Introspect permet de le faire de manière élégante.

```
OteConclusion ( R, OeilIntersection ( V1, V2 ) ),
AjouteConclusion ( R, OeilGroupe ( V1, V3 ) ),
AjouteCondition ( R, GroupeConcerné ( V3 ) ),
AjouteCondition ( R, CouleurJoueur ( V1 ) ),
AjouteCondition ( R, CouleurGroupe ( V3, V1 ) ) :-
    Regle ( R ),
    InitialiseCoupe (),
    Conclusion ( R, OeilIntersection ( V1, V2 ) ),
    Variable ( V3 ),
    CoupeSiDeduction (),
    TypeVariable ( V3, groupe ).
```

Figure 7

La figure 7 donne un exemple de transformation de règles concluant sur la propriété Oeil concernant une intersection en la propriété Oeil concernant un groupe. On peut noter au passage l'utilisation des métaprédicats InitialiseCoupe () et CoupeSiDéduction () qui permettent de ne pas faire autant de fois le travail qu'il y a de variables de type groupe, mais de ne faire le travail qu'une seule fois.

Un autre mécanisme indispensable de modification des programmes engendrés est le réordonnancement. Un bon ordre d'exécution amène a des gains de temps substantiels [Ishida 1988]. Ce mécanisme d'Introspect a déjà été décrit antérieurement [Cazenave 1996a].

Application au Maintien de la Cohérence

Le but de l'application de la métaprogrammation logique au maintien de la cohérence est d'avoir une approche déclarative pour les programmes de mise a jour de la base et pour la spécification des contraintes d'intégrité. Une approche déclarative permet de modifier facilement les programmes sans avoir a se soucier de la façon dont ils seront exécutés, c'est donc une approche intéressante pour la réingéniérie. La contrepartie est que les programmes déclaratifs sont en général très inefficaces.

La spécialisation automatique de programmes permet de concilier déclarativité et efficacité. Elle permet par exemple de spécialiser des règles de mise a jour en utilisant les contraintes d'intégrités. Si nous avons par exemple la contrainte d'intégrité suivante :

```
ajoute ( age ( PERSONNE, X ) ) :- X >= 0, X<120.
```

Et les règles de mise à jour suivantes :

```
ajoute ( pere ( X, Y ) ) :- element ( baseajouts, pere ( X, Y ) ).
ajoute ( age ( PERSONNE, X ) ) :- element ( baseajouts, age ( PERSONNE, X ) ).
```

Nous utilisons la métarègle générale suivante pour spécialiser les mises à jour sur les contraintes d'intégrité :

```
ajoute ( reglesmiseajourcontraintes, R2 ) -:
    element ( reglesmiseajour, R ),
    element ( reglescontraintes, R1 ),
    conclusion ( R, P ),
    conclusion ( R1, P1 ),
    substitution ( P, P1, ListeSubstitutions ),
    listeconditions ( R1, ListeContraintes ),
    effectuesubstitutions ( R1, ListeSubstitutions ),
    duplique ( R1, R2 ),
    ajoutelisteconditions ( R2, ListeContraintes ).
```

L'application de cette règles sur les règles définies précédemment donne les règles spécialisées de mise a jour suivantes :

```
ajoute ( pere ( X, Y ) ) :- element ( baseajouts, pere ( X, Y ) ).
ajoute ( age ( PERSONNE, X ) ) :- element ( baseajouts, age ( PERSONNE, X ) ),
                                  X>=0, X<120.
```

On voit que dans le cas de la première règle de mise à jour, le gain est important puisqu'il n'y a pas à déclencher les règles de contrôle de la cohérence. Dans le cas de la deuxième règle, il est aussi plus intéressants de vérifier que les contraintes sont vérifiées directement.

Cette technique permet de faciliter la modification des contraintes et des règles de mise à jour sans pour autant sacrifier l'efficacité des mises à jour.

Conclusion

La métaprogrammation logique est un outil efficace et général. Nous pensons qu'il a des applications dans beaucoup de domaines liés à la manipulation automatique de programmes.

Nous avons écrit un système appelé Introspect basé sur ces principes. Il crée et modifie des programmes qui sont ensuite compilés par un métaprogramme en C++ (la compilation en C++ leur permet de s'exécuter 60 fois plus vite). Ces programmes comprennent actuellement jusqu'à un million de lignes de C++ : les outils de métaprogrammation sont indispensables pour gérer ces programmes.

Les recherches sur la métaprogrammation en logique ont des liens forts aussi bien avec le milieu industriel comme le montrent [Rabaute & al. 1997] et [Cazenave 1997], qu'avec le milieu académique [Barklund 1994] [Cazenave 1996b] [Leuschel 1997]. Nous pensons que les concepts de la métaprogrammation logique peuvent être appliqués dans des environnements informatiques conventionnels. La manipulation et la représentation de programmes par d'autres programmes est un sujet de recherche qui peut permettre de développer des outils corrects et efficaces pour la ré-ingéniérie des systèmes d'information.

Bibliographie

[Barklund 1994] - J. Barklund. *Metaprogramming in Logic*. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.

[Cazenave 1996a] T. Cazenave, *Automatic Ordering of Predicates by Metarules*. Proceedings of the 5th International Workshop on Metareasonning and Metaprogramming in Logic, Bonn, 1996.

[Cazenave 1996b] - T. Cazenave. *Système d'Apprentissage par Auto-Observation. Application au Jeu de Go*. Thèse de l'Université Paris 6, Décembre 1996.

[Cazenave 1997] - T. Cazenave. *Automatically Improving Agents Behaviors in an Urban Simulation*. Second International Conference on Industrial Engineering Application and Practice, USA, 1997.

[Consel & Danvy 1993] - C. Consel, O. Danvy. *Tutorial Notes on Partial Evaluation.* 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, Charleston, South Carolina, January 1993.

[Cousot & Cousot 1992] - P. Cousot, R. Cousot. *Abstract Interpretation and Application to Logic Programs.* Journal of Logic Programming, 13 :103-179, 1992.

[Gallagher 1993] - J. Gallagher. *Specialization of Logic Programs.* Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.

[Hill & Lloyd 1994] - P. M. Hill, J. W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Mass., 1994.

[Ishida 1988] - T. Ishida. *Optimizing Rules in Production System Programs*, AAAI 1988, pp 699-704, 1988.

[Jones et al. 1993] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[Leuschel 1997] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. Thèse de l'Université Catholique de Louvain, 1997.

[Lloyd & Shepherdson 1991] - J. W. Lloyd and J. C. Shepherdson. *Partial Evaluation in Logic Programming*. J. Logic Programming, 11 :217-242., 1991.

La Métaprogrammation Logique. Un Outil pour Créer et Modifier de Grands Programmes.

[Pitrat 1990] - J. Pitrat, *Métaconnaissance - Futur de l'Intelligence Artificielle*, Hermès, Paris, 1990.

[Rabaute & al. 1997] - J. F. Rabaute, Ph. Devienne, A. Parrain, S. Varennes, P. Taillibert. *Unfolding and Aggregating Transformations of Real Prolog Programs*. ILP97 Workshop on Specialization of Declarative Programs, NY, 1997.

[Tamaki & Sato 1984] - H. Tamaki, T. Sato. *Unfold/Fold Transformations of Logic Programs*. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.