La « créativité calculatoire » et les heuristiques créatives en synthèse de prédicats multiples

Marta Fraňová, Yves Kodratoff

Équipe Inférence et Apprentissage, LRI, Bât. 490, 91405 Orsay, France
mf@lri.fr, yk@lri.fr

Résumé. Nous présentons une approche à ce que nous appelons la « créativité calculatoire », c'est-à-dire les procédés par lesquels une machine peut faire montre d'une certaine créativité. Dans cet article, nous montrons essentiellement que la synthèse de prédicats multiples en programmation logique inductive (ILP) et la synthèse de programmes à partir de spécifications formelles (SPSF), deux domaines de l'informatique qui s'attaquent à des problèmes où la notion de créativité est centrale, ont été amenés à ajouter à leur formalisme de base (l'ILP pour l'un, les tableaux de Beth pour l'autre) toute une série d'heuristiques. Cet article présente une collection d'heuristiques qui sont destinées à fournir au programme une forme de créativité calculatoire. Dans cette présentation, l'accent est plutôt mis sur les heuristiques de l'ILP mais lorsque cela était possible sans de trop longs développements, nous avons aussi présenté quelques heuristiques de la SPSF.
L'outil indispensable de la créativité calculatoire est ce que nous appelons un 'générateur d'atouts' dont une spécification (forcément informelle comme nous le verrons) est fournie comme première conclusion aux exemples décrits dans le corps de l'article.

1 Introduction et Motivations

Le but de cet article est de présenter un exemple non trivial d'une méthodologie de la créativité et, par là, de commencer à tracer les grandes lignes de ce qu'on pourrait appeler un peu pompeusement la « créativité calculatoire », un sujet que nous avons déjà abordé dans Franova et al. (1993). Ce domaine remonte aux travaux de Newell et Simon (1972) et a été décrit par Boden (1999). Cependant, cet article n'est pas destiné à aborder l'état de l'art de ce domaine mais à montrer comment les informaticiens spécialisés en programmation logique inductive (ILP) et en synthèse de programmes à partir de spécifications formelles (SPSF) ont affronté les problèmes posés par la synthèse de prédicats multiples. Les problèmes ainsi posés sont de nature récursive et exigent la programmation d'une sorte de créativité dont nous voulons donner quelques exemples.

Un des problèmes les plus difficiles que s'est posé la programmation logique inductive (ILP) est celui de la synthèse à partir d'exemples de prédicats multiples et mutuellement dépendants. Les articles de base dus à de Raedt et al. (1993a et b) et de Raedt et Lavrac (1996) ont analysé les difficultés rencontrées lors de la résolution de ce problème et ont donné lieu à un courant de recherche illustré par de nombreux travaux (en plus des articles et auteurs cités ci-après, on pourra voir aussi Martin et Vrain (1995), Zhang et Numao (1997), Fogel et Zaverucha 1998). Le problème que s'est posé la communauté de l'ILP est

exactement ce que nous appelons : un « problème récursif ». En effet, il ne faut pas confondre la programmation récursive qui consiste à écrire de façon récursive une solution déjà connue d'un problème (récursif ou non) et la solution des problèmes récursifs dont, justement, on ne connaît pas encore la solution, ce qui suppose une créativité différente de celle de la programmation récursive. Pour illustrer ce propos, donnons le principe de l'exemple (un peu simplifié) de la détermination du point en fin de phrase dans un texte dont la compréhension ne nécessite pas la connaissance de l'ILP mais ressort des mêmes mécanismes. Pour distinguer un point 'fin de phrase' d'un point 'partie ou fin d'abréviation', il est nécessaire de combiner deux sources d'information. L'une est le fait qu'un point final (sauf erreur de frappe dont nous ne parlons pas) est toujours accolé à une suite de lettres constituant un mot 'bien connu' et suivi d'un blanc. L'autre est qu'il est suivi d'un blanc (sauf fin de paragraphe, dont nous ne parlons pas) précédant immédiatement un mot débutant par une majuscule mais qui commence 'souvent' par une minuscule dans d'autres parties du texte. Il est possible d'écrire deux procédures, l'une de reconnaissance du point 'fin de phrase', l'autre des mots ne débutant pas toujours par une majuscule. L'une fait appel à l'autre pour prendre une décision. Dans la mesure où nous ne connaissons pas de spécification formelle du 'mot bien connu' ni du 'mot commençant normalement par une minuscule', le problème n'est pas d'écrire un programme récursif ou non sur ce sujet. Le problème est de rechercher une spécification des deux problèmes dépendants l'un de l'autre de façon récursive (et donc constituant un « problème récursif »). En d'autres mots, il s'agit de résoudre le problème récursif point-final/majuscule-inattendue. La thèse de Heitz (2008) propose une solution itérative à ce problème récursif.

Un exemple en ILP est celui de la génération automatique de prédicats mutuellement dépendants *odd* et *even* par le système ATRE de Malerba et al. (1998). Cette définition peut paraître surprenante mais si on « force la récursion » et qu'une préférence soit donnée à des relations utilisant une seule fois la fonction successeur, alors elle devient tout à fait naturelle :

even(X) :- zero(X) // odd(X) :- succ(Y,X), even(Y) // even(X) :- succ(Y,X), odd(Y).

Ce simple exemple montre bien que ATRE a fait preuve d'une créativité incontestable par rapport aux définitions scolaires : even(X) :- zero(X) // odd(X) :- succ(0,X) // odd(X) :- succ(Y,X), succ(Z, Y), odd(Z) // even(X) :- succ(Y,X), succ(Z, Y), even(Z).

Afin de montrer que ces heuristiques ne sont pas réservées à l'ILP mais constituent une base de la créativité calculatoire, nous fournirons, sous forme d'exemples les plus simplifiés possibles, un équivalent à l'heuristique ILP dans le domaine de la SPSF. Après tout, une spécification formelle peut être assimilée à une suite infinie d'exemples et de contre-exemples permettant d'obtenir une preuve formelle de l'équivalence entre spécification et programme synthétisé. L'approche de l'ILP dite 'classique' (De Raedt, L. Lavrac 1996) est tout à fait semblable à ceci près qu'elle 'se contente' de fournir une suite finie d'exemples et de contre-exemples et une preuve informelle d'équivalence : le programme 'couvre' les exemples et ne couvre pas les contre-exemples. Le fait de rechercher à synthétiser des prédicats multiples augmente la complexité de la tâche des programmes d'ILP et ceci de façon beaucoup plus dramatique qu'on peut le prévoir, comme l'ont très bien montré De Raedt et al. (1993a et b). Du fait que la tâche comprend la synthèse de plusieurs prédicats à la fois, il est évident qu'une des difficultés centrales est que l'ordre dans lequel ces prédicats sont synthétisés est primordial et cependant très difficile à maîtriser.

Nous rassemblons ces heuristiques en trois grands groupes, les méthodes de généralisation/particularisation, l'accroissement de la connaissance du domaine

(« background knowledge ») et la découverte de nouvelles connaissances (les atouts) relatives au domaine. L'intuition nous dit que, 'évidemment', seule la découverte des atouts est 'réellement' créative dans un sens calculatoire. Un de nos buts est d'illustrer que la créativité calculatoire est une combinaison de ces trois groupes.

2 Les méthodes de généralisation/particularisation

Dans la mesure où l'ILP part d'exemples instanciés, il est évident que les programmes ou prédicats qu'elle va synthétiser sont des sortes de généralisations de ces instances. Nous n'insisterons pas sur ce sujet qui a été abondamment traité dans la littérature de l'ILP, excepté pour signaler que deux heuristiques très différentes peuvent être utilisées. Tout d'abord, la construction des clauses possibles utilise des techniques de généralisation qu'elle combine avec celles de spécialisation. Ensuite, quand des clauses sont synthétisées, il s'agit de choisir celles que l'on va conserver, en définissant la notion de couverture et une mesure d'intérêt.

La méthode la plus classique et qui a fait l'objet de nombreuses variations est celle de FOIL (Quinlan, 1990), la mesure du gain en entropie apporté par chaque clause synthétisée. On fournit au système T_0 exemples dont T^+_0 sont des exemples positifs et T^-_0 sont négatifs. L'entropie de départ de l'ensemble des exemples est

$E_0 = -[((T^+_0/T_0) \log_2 (T^+_0/T_0)) + ((T^-_0/T_0) \log_2 (T^-_0/T_0))]$. Quand on a synthétisé la clause C, on peut constater que cette clause est satisfaite par T^+_C exemples positifs et T^-_C exemples négatifs. Soit $T_C = T^+_C + T^-_C$, l'entropie associée à C est

$E_C = - T_C/T_0 [((T^+_C/T_C) \log_2 (T^+_C/T_C)) + ((T^-_C/T_C) \log_2 (T^-_C/T_C))]$. La différence de ces deux valeurs mesure le gain entropique associé à la clause C. Cette formule de base est une des plus classiques pour diriger le déplacement des programmes d'apprentissage automatique dans l'espace des hypothèses possibles. En fait, elle n'illustre notre propos que de façon très partielle : chaque système inductif, selon la nature des données qu'il traite peut utiliser une heuristique spéciale pour se déplacer dans l'espace des hypothèses. Par exemple, Kijsirikul et al. (1991) ont développé une mesure d'intérêt qui combine le gain en entropie et une distance syntaxique entre le but à atteindre et la transformation opérée par C. L'équivalent logique de cette heuristique est encore plus trivial : en SPSF, on démontre les lemmes et les théorèmes dans l'ordre dans lequel ils se présentent au cours de la démonstration. Les preuves par récurrence utilisent aussi une heuristique triviale : prouver d'abord le cas de base, puis le cas général.

Quand on se limite à la synthèse d'un seul prédicat, *p*, la cohérence de chaque nouvelle hypothèse (qui suggère l'adjonction d'une seule clause à la fois) est testée sur les exemples et les contre-exemples. Quand on passe à la synthèse de prédicats multiples, on peut continuer à utiliser cette procédure, pour chaque prédicat p_i de l'hypothèse multiple. On conçoit qu'il soit plus logique de tester à la fois la cohérence de tous les p_i constituant l'hypothèse, et non celle d'un seul d'entre eux. Cependant, cette forme d'entrecroisement de toutes les démonstrations augmente bien évidemment la complexité calculatoire. C'est pourquoi, afin de résoudre un problème réel d'analyse d'images, Esposito et al. (1998, 2000) ont été conduits à introduire ce qu'ils appellent une stratégie de recherche 'separate-and-parallel-conquer search'. Elle reprend l'idée classique du 'séparer pour conquérir' mais exécute plusieurs 'conquêtes' en parallèle. Plus précisément, le 'séparer pour conquérir' classique conquiert (= apprend) une seule clause à la fois, et on la rajoute dans la base de

connaissances puis on sépare (= élimine) les exemples couverts par cette clause. On reprend ensuite le processus sur les exemples restants. La stratégie de Esposito et al. part d'une sur-généralisation qui couvre un seul exemple. Les spécialisations permettant d'éviter de couvrir des exemples négatifs sont examinées en parallèle. Chaque spécialisation utilise une heuristique de gain pour choisir la spécialisation qu'elle va s'adjoindre afin de couvrir le moins de contre-exemples possibles. Toutes les spécialisations ne couvrant aucun exemple négatif sont candidates à devenir la clause synthétisée.

3 Augmenter la connaissance du domaine

Tout d'abord, il nous faut distinguer de façon précise ce qui est connaissance du domaine de ce qui est « création d'atouts ». La connaissance du domaine est le recueil de toutes les connaissances que l'on a au sujet du problème. Un exemple simple en est la connaissance de la possibilité des relations récursives qui est une connaissance très générale du domaine. Un exemple moins simple, que nous utiliserons dans la suite, est une propriété 'classique' de la récurrence (Péter 1967): supposons que l'on étudie une relation R(x, z) partiellement fausse où x est la variable d'entrée et z celle de sortie. Ceci suggère la construction récursive d'un nouveau prédicat P, comme nous le verrons plus loin. Si en appliquant l'hypothèse d'induction, on trouve une valeur de z pour tout x tel que G(x), alors P peut s'écrire récursivement sous la forme P(x) :- G(x), P(pred(x)). En ILP, ces connaissances du domaine sont généralement exprimées par une version quelconque du rasoir d'Occam. Inversement, la création des atouts constitue en la partie réellement créative de la synthèse de prédicats : il s'agit de découvrir des structures encore inconnues au sein des données.

Nous ne décrirons évidemment pas ici la masse des connaissances implicites que suppose le fait même de l'existence de l'ILP ou de se poser la question de la synthèse automatique de programmes à partir d'exemples. Ce thème a été développé de façon approfondie par les créateurs de FOCL (voir Pazzani et Kibler 1992) pour le sujet général de l'ILP et, pour les prédicats multiples par Giordana et al. (1993). De façon plus implicite, tous les systèmes d'apprentissage inductifs utilisent ce qu'on appelle des « biais d'apprentissage » afin de limiter la taille de l'espace des clauses que le système peut apprendre. Ces biais sont de nature multiple. Il est intéressant de décrire un cas où ces biais sont donnés sous forme syntaxique par la forme des clauses recherchées. On trouve un exemple de ce type d'approche dans un article de Baroglio et Botta (1995) où les auteurs introduisent des 'templates' qui sont associés aux prédicats à synthétiser. Ils donnent un exemple pour les prédicats binaires p(x, y), auxquels ils associent, entre autres, le 'template' suivant : p'(x, y) :- p(x, z), diff(z, y). Ceci leur permet de prendre en compte les cas où le prédicat 'différent' est nécessaire à l'expression qu'ils cherchent à synthétiser. De même, Brazdil et Jorge (1994) introduisent la notion de 'sketch' qui a une forme intuitive évidente mais qui impose des liens entre variables qui constituent déjà une partie importante de la solution.

4 Découvrir de nouvelles connaissances relatives au domaine (les 'atouts')

4.1 Inclure dans la base de connaissances la connaissance déjà synthétisée

Ceci est pratiqué en général dans les programmes d'ILP où les clauses nouvellement apprises sont rajoutées à la base de connaissance des clauses connues. Il est classique de supprimer alors les exemples couverts par la clause qu'on vient d'ajouter. On assure ainsi la convergence du système qui s'arrête quand tous les exemples positifs sont couverts. Cependant, il est aussi possible aussi de rajouter aux exemples d'apprentissage tous les faits (« ground facts ») qu'on peut déduire des clauses déjà synthétisées, c'est le principe de ce que Jorge et Brazdil (1996) nomment « iterative bootstrap induction ». Bien entendu, ceci complique beaucoup le déroulement du programme puisque des faits doivent être ajoutés pour chaque hypothèse nouvelle, et rétractés chaque fois qu'une clause est enlevée de la base de connaissances.

4.2 La détection des chemins relationnels (« relational pathfinding »)

Richards et Mooney (1992) ont introduit cette notion afin de chercher à résoudre le problème rencontré quand il est nécessaire d'ajouter à la fois plusieurs littéraux à une clause pour pouvoir observer un gain en entropie. Ce problème est particulièrement grave quand plusieurs prédicats sont synthétisés ensemble. En un sens, leur solution consiste à rechercher, au sein du graphe des relations exprimées par les exemples, un sous-graphe caractéristique d'une relation donnée, celle du prédicat à synthétiser. Ils présentent une heuristique qui permet d'utiliser au mieux les connaissances mais qui ne constitue évidemment pas une solution générale au problème de trouver tous les sous-graphes d'un graphe donné. Supposons que, au cours d'une démonstration, on ait besoin de prouver un prédicat binaire P (a, b) où P est un prédicat à synthétiser et a et b sont deux instances présentes dans la base des exemples, reliées entre elles par aucune relation connue, mais ayant un ensemble de relations, supposées ici binaires afin de ne pas compliquer l'exemple, connues {R} avec d'autres instances. Rappelons que la base d'exemples étant constituée d'exemples instanciés, nous parlons ici des valeurs de ces instances comme, par exemple, dans une base de relations familiales, les noms propres des individus constituant la famille. L'heuristique développée par Richards et Mooney (1992) consiste à considérer les chemins définis par {R}. Si ces chemins partiels créent un chemin entre a et b. Alors, les liens entre a et b sont constitués par une suite ordonnée des opérateurs de {R} qui est de la forme $R_i(a, a_i)$, …, $R_j(b_j, b)$. On suppose que ce chemin est un exemple de P c'est-à-dire que P (a, b) :- $R_i(a, a_i)$, …, $R_j(b_j, b)$. En remplaçant les instances par des variables maintenant le lien entre les instances, on obtient facilement un prédicat P(x, y) :- $R_i(x, x_i)$, …, $R_j(y_j, y)$ qui, une fois ajoutée à la base de connaissance, permettra de prouver P (a, b), sauf si elle nécessite encore un prédicat unaire (forcément 'oublié' au cours de l'heuristique), ce qui nécessitera une autre heuristique que nous évoquons au paragraphe suivant. En tous cas, nous avons au moins obtenu une première suite de prédicats nécessaires ensemble pour pouvoir observer un gain en entropie. Si ces chemins partiels ne créent pas un chemin entre a et b, le procédé est itéré sur les sommets libres qui ont encore des relations {R'} avec des instances non utilisées

précédemment. Si plusieurs chemins sont trouvés, on peut encore se servir de leur mesure d'entropie pour les séparer. Quand cette heuristique réussit, elle met donc en évidence une structure présente au sein des données, et qui est importante pour la suite de la preuve. C'est un exemple de ce que nous appelons un atout dont nous donnerons une définition plus générale en conclusion.

4.3 L'analyse des échecs

Un exemple de récupération des échecs en synthèse de prédicats multiples est fourni par Kijsirikul et al. (1992) et Zelle et al. (1994). Supposons que nous soyons en situation d'échec parce qu'il manque à la clause C un prédicat P présent dans l'ensemble des exemples. Il s'agit donc de déterminer quel prédicat il faut ajouter à la clause C qui couvre encore des exemples négatifs. C est utilisée pour prouver des exemples positifs et négatifs. On obtient ainsi une liste d'instances {L} appartenant à la preuve que C couvre des exemples positifs et une liste d'instances {L'} appartenant à la preuve que C couvre des exemples négatifs. En analysant les différences entre les instances de {L} et ceux de {L'}, il est souvent possible de mettre en évidence le prédicat qui les différencie et qui est donc P.

Nous allons maintenant développer en détail deux méthodes de « récupération des échecs » en SPSF.

Rappelons quelques principes fondamentaux de l'application des preuves par récurrence à la synthèse de programmes. La formule la plus simple qu'on puisse avoir à démontrer est de la forme $\forall x \exists z\ Q(x, z)$. Cette formule F1 détermine la fonction de Skolem SF associée à z, qui est la valeur calculée par cette fonction de Skolem quand elle s'applique à la variable x, c'est-à-dire que $z = SF(x)$ pour tout x. En d'autres termes, du fait des quantificateurs qui leurs sont associés, on attribue à x la sémantique d'une variable d'entrée et à z celle d'une variable de sortie. Le problème de la SPSF est de trouver 'la' SF qui vérifie F1, c'est-à-dire qu'on doit trouver une preuve constructive de $\forall x \exists z\ Q(x, z)$.

Faire une preuve par récurrence consiste à analyser ce qu'on appelle « le cas de base » et « le cas général » puis à appliquer au cas général ce qu'on appelle « l'hypothèse d'induction ».

Nous allons nous restreindre à ne considérer que l'arithmétique des entiers naturels. Dans ce contexte, le cas de base est $x = 0$ car la formule la plus simple dont nous partons ne comporte pas de conditions sur x. Le cas général est $x = s(a)$ où 's' est la fonction successeur. L'hypothèse d'induction (« si la formule est vraie pour n, alors on peut prouver qu'elle est vraie pour n+1 ») s'écrit $\exists e\ Q(a, e) \Rightarrow \exists z\ Q(s(a), z)$ ou bien , en introduisant les fonctions de Skolem, $Q(a, SF(a)) \Rightarrow Q(s(a), SF(s(a)))$.

Supposons alors que l'on puisse prouver le cas de base et que l'on trouve que, pour $x = 0$, la formule F1 est vérifiée si $z = a_0$, c'est-à-dire que $SF(0) = a_0$.

Comme première illustration de la notion de récupération d'un échec en SPSF, supposons encore que, dans le cas général, on soit capable de trouver une fonction G telle que $z := G(e) = G(SF(a))$ à condition que P(z) soit vrai. Ceci est une condition sur les sorties qui constitue un cas d'échec de la preuve par récurrence. En effet, on ne peut pas imposer des conditions sur la variable de sortie qu'on ne sait pas encore calculer (ou alors, ses conditions auraient dû être incluses dans F1). L'heuristique de réaction à l'échec que nous proposons ici est de construire un programme que nous qualifions de « hypothétique », F, dont nous disons qu'il est 'inspiré' par ce que nous savons déjà de G, mais qui ne comporte pas la condition P(z). On espère que P(F(x)) est vrai pour tout x. Bien entendu, si cette espérance est déçue par les

faits, la bataille n'est pas terminée, mais nous allons nous contenter d'illustrer d'abord le cas simple où ∀x P(F(x)).

Cette illustration est issue de preuves relatives à la fonction d'Ackermann publiées dans Anonyme (2008). Le lecteur devra donc admettre que certains pas de calcul sont exposés en détail ailleurs. Ces calculs nous amènent à chercher à prouver que ∀x ∀y ∃z, Ack (x, y) = y + z. Dans le cas de base, on cherche à résoudre l'équation Ack (0, y) = y + z dont on sait que la solution est z = 1. On a donc SF (0, y) = 1. Dans le cas général, et après un calcul assez long, on s'aperçoit qu'il nous faut prouver le lemme ∃z Ack (a+1, y) = y + z. Nous sommes donc conduits à tenter une preuve par induction sur l'autre variable d'entrée, y. Dans le cas de base, y = 0 et nous obtenons que SF (a + 1, 0) = 1 + SF (a, 1). Dans le cas général, nous posons y = b + 1 et nous obtenons que SF (a+1, b+1) + 1 = SF (a + 1, b) + SF (a, b + SF (a + 1, b)). Si nous pouvions prouver que SF (a + 1, b) + SF (a, b + SF (a + 1, b)) > 0, nous aurions ainsi obtenu une formule récursive pour le calcul de SF. Nous sommes incapables de le faire et donc nous sommes dans un cas d'échec de la preuve par récurrence. Comme nous l'avons dit, nous allons construire une fonction hypothétique F qui éliminera la condition sur la variable de sortie (z > 0). Nous utilisons tout ce que nous savons de SF pour construire la fonction F définie par :

F(0, y) = 1 // F(a + 1, 0) = 1 + F (a, 1) // F (a+1, b+1) = F (a + 1, b) + F (a, b + F (a + 1, b)) -1

On prouve alors que ∃z F (x, y) = 1 + z c'est-à-dire que ∀x ∀y, F (x, y) > 0 et donc que SF qui présente les mêmes relations récursives a aussi cette propriété.

Dans ce cas, la « récupération des erreurs » consiste en une preuve que l'erreur n'existe pas en réalité. Dans des cas plus complexes, elle peut amener à la génération de prédicats comme nous allons le voir maintenant.

Comme seconde illustration de la méthode de récupération des échecs en SPSF, considérons le problème de la construction de prédicat à partir d'une formule partiellement fausse.

Considérons la formule suivante : (F1) ∀a ∀b b>0, ∃z b = a + z

L'analyse de cette formule, au cours d'une tentative de preuve par récurrence, nous montre qu'elle est partiellement fausse et donc la preuve échoue. Comme Zelle et al. (1994), nous supposons alors qu'il existe un prédicat hypothétique P(a,b) qui permettra de caractériser les cas ou F1 est vraie. Pour construire P, nous utilisons les cas d'échec de la preuve de F1.

Dans le cas de base, la condition b>0 implique que b = s(0). En considérant la formule ∀a ∃z s(0) = a + z, on constate qu'elle a deux solutions partielles, c'est-à-dire z = s(0) si a = 0, ainsi que z = 0 si a = s(0). Ces conditions d'échec nous fournissent deux propriétés du prédicat hypothétique P :

P(a,s(0)) :- a = 0 // P(a,s(0)) :- a = s(0)

Dans le cas général, on a b=s(c), c>0, et on doit prouver que ∃e c = a + e => ∃z s(c) = a + z. Nous utilisons systématiquement une heuristique qui nous conseille d'étudier les solutions dites triviales du problème ∃z s(c) = a + z, sans utiliser encore l'hypothèse d'induction. Nous obtenons deux solutions triviales, z = s(c) si a = 0, ainsi que z = 0 si a = s(c). Ceci nous fournit deux nouvelles propriétés de P :

P(a,s(c)) :- a = 0 // P(a,s(c)) :- a = s(c)

Nous utilisons alors explicitement l'hypothèse d'induction et nous trouvons z = s(e) sans aucune condition sur a. La logique des propriétés récursives (Péter 1967) nous permet alors d'en tirer la propriété récursive de P : P(a,s(c)) :- P(a,c).

Les cinq propriétés que nous venons de construire définissent complètement le prédicat P tel que $\forall a\ \forall b\ b>0$, $(P(a,b) \Rightarrow \exists z\ b = a + z)$. Nous n'obtenons donc pas la preuve de F1 mais les conditions auxquelles F1 est vérifiée.

4.4 L'abduction

Une méthode proposée par Kakas et al. (1998) pour résoudre le problème de la synthèse de prédicats multiples consiste à modifier la technique de preuve en utilisant la programmation logique abductive (ALP - voir Kakas et al. 1998, et Adé et Denecker 1995) au lieu de l'ILP. Ce n'est pas une heuristique à proprement parler mais un changement de cadre logique. Il n'est donc pas étonnant que n'ayons pas à proposer d'équivalent évident en SPSF. La définition de l'ALP fournie par Kakas et al. (1998) propose d'associer au programme logique habituel un ensemble de prédicats incomplètement définis, appelés les « abductibles », et un ensemble de constantes d'intégrité assurant que les inférences sur les abductibles restent cohérentes avec P. De ce fait, il se donne la possibilité d'ajouter à la « négation par l'échec », utilisée en programmation logique, une « négation par défaut » de la façon suivante. À chaque symbole prédicatif, *pred*, utilisé dans P, il associe une négation, *non_pred*, et il ajoute à l'ensemble des contraintes d'intégrité la contrainte :- *non_pred*, *pred*. La dérivation logique ne consiste plus seulement à conclure sur Vrai ou Faux, mais à obtenir, comme en abduction classique, un ensemble de prédicats qui constituent l'explication abductive de la preuve. Cette approche illustre, en tous cas, l'inventivité dont a fait preuve la communauté de l'ILP pour tenter de résoudre le problème de la synthèse de prédicats multiples à partir d'exemples.

Quand nous avons collecté les échecs de la preuve de la formule (F1) $\forall a\ \forall b\ b>0\ \exists z\ b = a + z$, nous avons utilisé une approche semblable mais beaucoup moins bien formalisée que celle de Kakas et al. (1998). Ce travail de formalisation reste donc à faire en SPSF.

5 Conclusion

5.1 Le générateur d'atouts

Un générateur d'atouts décrit un comportement humain ou automatisé. Des actions s'enchaînent et sont conduites par des stratégies, elles-mêmes définies informellement. Il est donc impossible de donner une définition formelle complète d'un générateur d'atouts. Voici d'abord une définition formelle d'un atout que nous aurons donc à compléter de façon informelle.

Définition formelle (incomplète) :

Etant donnés une théorie formelle indécidable, éventuellement incomplète, et un théorème premier, un **atout** (sous-entendu: pour réussir à démontrer le théorème) est obtenu par l'analyse de l'échec de la preuve du théorème premier. Cette analyse peut conduire à trouver trois différentes causes de l'échec, chacune conduisant à une stratégie de reprise différente. Premièrement et le plus souvent, une partie « bien-connue » de la théorie du doamine a été oubliée et elle doit être récupérée dans les manuels du domaine de spécialité concerné. Deuxièmement, un lemme très particulier et non classique aurait été nécessaire et l'atout est une conséquence de la théorie qui doit être exprimée de sorte qu'elle élimine

l'échec auquel on vient de se heurter. Troisièmement, la théorie est incomplète et l'atout complète la théorie juste assez pour rendre prouvable le théorème premier.

Voici maintenant les *commentaires informels* qui décrivent, à partir de cette définition, ce qu'est un générateur d'atouts.

Tout d'abord, imaginons un humain placé dans la situation de prouver un théorème dans une théorie indécidable. Il va d'abord tenter de démontrer directement le théorème premier à partir de la théorie. Supposons que ce soit un excellent mathématicien et qu'il échoue à trouver la preuve. Il sait donc qu'au moins la preuve sera soit difficile, soit impossible. Il va donc commencer un long processus par lequel il va combiner, en principe, deux stratégies. La première de ces stratégies consiste à rechercher ce qui lui manque pour effectuer la preuve. Il va s'aider de son intuition de sa connaissance des preuves constructives, de sa connaissance de la théorie formelle, et des propriétés affirmées par le théorème à prouver. Cette partie du comportement humain, nous l'appelons « stratégies de choix des atouts » car elle permet de trouver des théorèmes (peut-être indécidables !) qui serviraient en effet à prouver notre théorème. Cependant, notre expérience nous a montré que nous ne savons pas simuler – pour le moment – cette partie du comportement humain.

La deuxième de ces stratégies consiste à engendrer tous les théorèmes décidables à partir de la théorie, et d'examiner – en fait de leur attribuer une probabilité de succès – chacun des ces théorèmes afin de savoir s'il peut ou non intervenir quelque part dans une preuve constructive du théorème à prouver.

Cet article illustre plusieurs techniques de dérivation d'atouts utilisées en ILP et en SPSF.

Définition informelle (mais précise) d'un générateur d'atout :

Soit une stratégie de preuve définie informellement par le fait qu'elle peut être soit un signifié dans l'esprit d'un mathématicien, ou une stratégie de preuve automatique.

Etant donnés une théorie formelle indécidable, un théorème premier, nous appelons générateur d'atouts (sous-entendu: pour réussir à démontrer le théorème premier) une stratégie de preuve présentant les deux propriétés suivantes : 1. elle engendre des atouts (qui sont donc, par définition, des maillons dans une preuve constructive du théorème premier). 2. elle n'engendre que des atouts qui seront plus faciles à démontrer que le théorème premier. 3. si elle engendre une suite potentiellement infinie d'atouts, alors il existe toujours une généralisation possible à la suite finie obtenue, et cette généralisation est censée représenter la suite infinie qui prolonge la suite finie obtenue en pratique. Dans le cas d'une spécification informelle, la condition 3 est formulée plutôt comme l'exclusion des suites infinies : le générateur d'atouts ne doit pas engendrer de suite infinie d'atouts.

Nous constatons donc qu'un générateur d'atout n'a pas la prétention d'assurer qu'on atteindra une preuve formelle du théorème premier. Il est simplement une stratégie qui a donné des résultats intéressants en construction automatique de programmes. Il nous semble constituer une stratégie qui, une fois adaptée aux besoins de nos 'utilisateurs', peut les aider à mieux gérer les problèmes de récurrence qu'ils rencontrent dans leur activité de programmation. Une application réelle de toutes ces techniques nous paraît beaucoup plus probable dans un cadre de programmation assistée par ordinateur (pour les cas difficiles et peu intuitifs).

5.2 Les atouts et la connaissance du domaine

Nous avons définis les atouts comme des connaissances inventées en cours de preuve. Il est clair que si une connaissance du domaine est absente, les problèmes vont se poser de

façon totalement différente et, en particulier, il sera impossible de découvrir les atouts nécessaires. Un exemple nous est fournit par l'ILP qui selon qu'elle favorise ou non la création de définitions récursives et de définitions récursives croisées obtient des résultats extrêmement différents. En SPSF, du fait de la complexité des preuves, le rôle de la connaissance du domaine est moins évident, mais le mettre en évidence de façon claire est un travail que nous désirons entreprendre.

5.3 Un exemple d'une propriété découverte par notre méthodologie

Le but de cet article n'est pas de montrer combien notre méthode est puissante mais plutôt combien elle est simple. C'est pourquoi les exemples de SPSF décrits ici peuvent paraître peu convaincants. Afin d'illustrer le fait qu'elle peut conduire à des résultats créatifs, voici un exemple d'un problème encore non résolu, autant que nous le sachions.

La fonction d'Ackermann est habituellement définie par récurrence sur son premier argument. Appelons 'ACK' cette définition qui est:

ACK(0,n) = n+1 // ACK(m+1,0) = ACK(m,1) // ACK(m+1,n+1) = ACK(m,ACK(m+1,n))

Cette définition étant tellement utilisée, nous avons eu la curiosité de savoir s'il était possible de trouver une définition de la fonction d'Ackermann, AK(x,y), récurrente par rapport à son second argument. Pour découvrir cette définition, nous avons eu l'idée (en fait, inspirée par notre méthodologie) de tenter de prouver les deux lemmes suivants:

$$\forall x\ \exists y,\ ACK(x, 0) = z \ //\ \forall x\ \forall y\ \exists z,\ ACK(x, y+1) = ACK(x, y) + z$$

Bien évidemment, nous ne tentons pas de prouver directement $\forall x\ \forall y\ [ACK(x,y) = AK(x,y)]$ puisque, de toute façon, nous ignorons la définition de 'AK' qui sera inventée au cours de la preuve des ces deux lemmes. La preuve récursive du premier lemme exige une fonction auxiliaire AUX(x) et celle du second exige une autre fonction auxiliaire AUX3(x).

Finalement la fonction AK est définie par:

AK(x,0) = AUX(x) // AK(x,y+1) = AK(x,y) + AUX3(x,y)

Cette solution exige deux autres fonctions auxiliaires AUX1 et AUX2.

AUX(0) = 1 // AUX(a+1) = AUX(a) + AUX1(a)

AUX1(0) = 1 // AUX1(b+1) = AUX2(b,AUX(b+1))

AUX2(0,y) = 1 // AUX2(a+1,0) = 1+ AUX2(a,1) // AUX2(a+1,b+1) = AUX2(a+1,b) + AUX2(a,b+AUX2(a+1,b)) – 1

AUX3(0,y) = 1 // AUX3(a+1,y) = AUX2(a,AK(a+1,y))

Maintenant que cette définition de AK a été trouvée, les systèmes de preuve de théorèmes peuvent essayer de prouver que $\forall x\ \forall y\ [ACK(x,y) = AK(x,y)]$. Notre méthodologie l'a aussi prouvé en 'inventant' AK. Vous constatez que la définition de AUX2 est fortement semblable à celle de ACK si bien que notre nouvelle définition est malgré tout beaucoup influencée par la définition classique. Si cela présentait un intérêt particulier, il faudrait rechercher d'autres lemmes, moins évidents, pour trouver d'autres définitions. IL n'empêche qu'une définition complètement nouvelle de la fonction d'Ackermann a été trouvée. La preuve complète est très longue et exige l'utilisation complète de notre méthodologie.

Références

Adé H., Denecker M. (1995) *AILP: Abductive inductive logic programming*; in Proceedings of the fourteenth International Joint Conference on Artificial Intelligence, pp. 1201-1207.

Baroglio C., Botta M. (1995). *Multiple Predicate Learning with RTL*; in Topic in Artificial Intelligence, LNAI 992, pp. 44-55.

Beth, E. (1959). *The Foundations of Mathematics*; Amsterdam, 1959.

Boden M. (1999). *Computational models of creativity.*, Handbook of Creativity, Sternberg R. J. (Ed.) Cambridge University Press, pp 351–373.

Brazdil P., Jorge A. (1994). *Learning by Refining Algorithm Sketches*; in ECAI 94, 11th European Conference of Artificial Intelligence, pp. 427-433.

De Raedt, L. Lavrac N., Dzeroski S. (1993a). *Multiple Predicate Learning* ; in Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, pp. 1037-1042.

De Raedt, L. Lavrac N., Dzeroski S. (1993b). *Multiple Predicate Learning* ; in Proceedings of the third International Workshop on Inductive Logic Programming, pp. 221-240.

De Raedt, L. Lavrac N. (1996). *Multiple Predicate Learning in Two Inductive Logic Programming Settings* ; J. of the IGPL, vol. 4 No. 2, pp. 227-254.

Esposito F., Malerba D., Lisi F.A. (2000). *Multiple Predicate Learning for Document Image Understanding*; in Proceedings of the Fourteenth American Association for Artificial Intelligence Conference, pp. 372 – 376.

Fogel L., Zaverucha G. (1998). *Normal Programs and Multiple Predicate Learning*; in Proceedings of the 8th International Workshop on Inductive Logic Programming, Lecture Notes In Computer Science; Vol. 1446, pp. 175 – 184.

M. Franova, Y. Kodratoff, M. Gross (1993). *Constructive Matching Methodology: Formally Creative or Intelligent Inductive Theorem Proving?*; in: J. Komorowski, Z.W. Ras (ed.): Methodologies for Intelligent Systems; proc. of ISMIS'93, L.N.A.I. 689, Springer-Verlag, pp. 476-485.

Franova M. (2008). *A Construction of a Definition Recursive with respect to the Second Variable for the Ackermann function*; à paraître comme rapport de recherche interne du LRI, Orsay, France.

Giordana A., Saitta L., Baroglio C. (1993). *Learning simple recursive theories*, in Methodologies for Intelligent Systems, Komorowski J. Ras, Z. W. (Eds.) LNAI 689, 425-44, Springer Verlag, Berlin.

Heitz T. (2008). *Une méthode récursive pour le prétraitement des textes.* Thèse, Université Paris-Sud.

Jorge A., Brazdil P. (1996). *Architecture for iterative learning of recursive definitions*; in. Advances in Inductive Logic Programming, pp. 206-218.

Kakas A., Lamma E., Riguzzi F. (1998). *Learning multiple predicates*; in AIMSA'98 : Artificial intelligence : methodology, systems, and applications, Lecture Notes in Computer Science vol. 1480, pp. 303-316.

Kijsirikul B., Numao M., Shimura M. (1991). *Efficient learning of logic programs with non-determinate non-discrimonating literals*; in ML-91 – Machine Learning Conference, ed. Kaufmann, pp. 417-421.

Kijsirikul B., Numao M., Shimura M. (1992). *Discrimination-based constructive induction of logic programs* in Proceedings of the tenth National Joint Conference on Machine Learning, pp. 44-49, San Jose, CA.

Malerba D., Esposito F., Lisi F.A. (1998) *Learning Recursive Theories with ATRE*; in Proceedings of the Thirteenth European Conference on Artificial Intelligence, pp. 435-439.

Martin L., Vrain C. (1995). *MULT_ICN: An empirical multiple predicate learner*, Proc. 5th International Workshop on ILP, pp. 129-144.

Newell A., Simon H. A. (1972). *Human Problem Solving*, Prentice Hall.

Pazzani, M., Kibler, D. (1992). *The Utility of Knowledge in Inductive Learning*; Machine Learning 9, 57-94.

Péter R. (1967). *Recursive Functions*; Academic Press, New York.

Richards B.L., Mooney R.J. (1992). *Learning Relations by Pathfinding*; in Proceedings of the Tenth National Conference on Artificial Intelligence, Cambridge, MIT Press.

Zelle J.M., Mooney R.J., Konvisser J.B. (1994). *Combining Top-down and Bottom-up Techniques in Inductive Logic Programming*; in Machine Learning: Proceedings of the Eleventh International Conference, pp. 343-351.

Zhang X., Numao M. (1997). *MPL-Core: An Efficient Multiple Predicate Learner Based on Fast Failure Mechanism*; Journal of Japanese Society for Artificial Intelligence, Vol. 12, No. 4, pp. 582-590.

Summary

We present the idea of «computational creativity», that is, the whole set of the methods by which a computer may simulate creativity. This paper is restricted to multiple predicate learning in Inductive Logic Programming (ILP) and to Programs Synthesis from their Formal Specification (PSFS – called SPSF in the paper). These two subfields of Computer Science deal with problems where creativity is of primary importance. They had to add to their basic formalism, Beth tableaus for PSFS, sets of heuristics enabling the program to solve the problem of Multiple Predicate synthesis. This paper shows heuristics the goal of which is to provide the program with some kind of inventiveness. In our presentation, we insist more on ILP than on PSFS. When this is possible in a relatively short way, we describe also a few heuristics of PSFS.