RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

FRANÇOISE GARCIA

Étude et implémentation d'un système de déduction pour logique algorithmique

RAIRO. Informatique théorique et applications, tome 22, n° 1 (1988), p. 57-92

<http://www.numdam.org/item?id=ITA_1988__22_1_57_0>

© AFCET, 1988, tous droits réservés.

L'accès aux archives de la revue « RAIRO. Informatique théorique et applications » implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/conditions). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM

Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 22, n° 1, 1988, p. 57 à 92)

ÉTUDE ET IMPLÉMENTATION D'UN SYSTÈME DE DÉDUCTION POUR LOGIQUE ALGORITHMIQUE (*)

par Françoise GARCIA (1)

Communiqué par P. COUSOT

Résumé. – *Cet article présente la maquette d'un outil LAPD, automatisant la déduction dans une logique pour schémas de programme. LAPD permet de définir des théories algorithmiques qui axiomatisent les structures de données et leurs opérations, puis de raisonner sur les programmes manipulant ces structures.*

En premier lieu, on définit cette logique, dérivée de la Logique Algorithmique de Salwicki, dans laquelle on introduit une formalisation des entiers. Ceci rend le système de déduction incomplet. La preuve de non-complétude est cependant présentée car elle précise les limites du système.

La deuxième partie détaille l'architecture de la maquette, réalisée à partir du système LCF (4.2) *et de son méta-langage ML.*

Abstract. – *In this paper, we present LAPD, a prototype for automatic reasoning in logic for abstract programs. With LAPD, you can define algorithmic theories modelizing data structures and operations, then reason on programs using these structures.*

We first define the logic supported by LAPD, derived from Salwicki's Algorithmic Logic, in which we introduce integers. This makes the deduction system uncomplete. Nevertheless, we prove it, in order to point out the limits of the system.

In the second part, we detail the prototype. Implementation is based on LCF system (4.2) *and meta language, ML.*

Les travaux présentés prennent place dans le domaine de la preuve de programmes par les méthodes de la logique mathématique. Il s'agit de l'implémentation d'un système automatique de déduction pour une logique des programmes, en l'occurence la logique algorithmique définie par Salwicki à l'Université de Varsovie. La réalisation de cet outil permet d'évaluer les possibilités de développement pratique d'un système de ce type. Il s'agit donc

(*) Reçu janvier 1986.
(1) C.I.M.S.A.-S.I.N.T.R.A., 10-12, avenue de l'Europe, 78140 Velizy.

0296-1598 88/01 57 35/$5.50/© Gauthier-Villars

d'une étude de faisabilité : partant de la logique algorithmique, que peut-on offrir au programmeur du point de vue puissance d'expression et de démonstration, souplesse d'utilisation ?

L'outil réalisé, LAPD, offre à l'utilisateur : (1) les moyens de définir ses théories algorithmiques; (2) les éléments de base des preuves : axiomes et règles algorithmiques; (3) enfin un langage puissant, ML, pour construire des preuves en combinant ces éléments sous forme de tactiques. En effet, LAPD est construit à partir de LCF, système de démonstration pour la logique de Scott (logique des fonctions calculables), dont certains principes ont été repris, et en particulier l'utilisation du méta-langage ML de manipulation des formules, règles et tactiques (dans LAPD comme dans LCF une preuve est un programme ML).

La classe des programmes reconnus par cet outil, c'est-à-dire la classe des programmes dont on voudra étudier les propriétés à l'aide d'une modélisation par axiomes algorithmiques, est celle des programmes définis par systèmes de procédures mutuellement récursives sans paramètre [28].

PLAN

En première partie on présente la classe de programmes et la logique algorithmique pour lesquelles notre système a été réalisé. Puis nous verrons comment des problèmes spécifiques de logique algorithmique nous conduisent à adapter la logique choisie à des contraintes d'implémentation. Ces problèmes viennent du fait que l'on veut manipuler des théories algorithmiques axiomatisant des structures de données, théories qui pourront être infinies, et de la présence d'une règle infinitaire d'induction dans les systèmes de déduction complet : comment représenter et manipuler cet infini ?

La deuxième partie expose le fonctionnement de l'outil LAPD, précédé d'une description générale de LCF.

I. PREMIÈRE PARTIE

I.1. Logique Algorithmique (LA)

La Logique Algorithmique (LA) a été développée à Varsovie par l'école polonaise de logique (Salwicki, etc.). Comme la logique dynamique de Harel dont elle est proche [12], LA fait partie de la même famille de logiques des programmes que la logique de Hoare (LH).

Un des avantages de LA est d'être une logique très intuitive dans laquelle les formules sont construites par simple juxtaposition de programmes et de formules. C'est le programme lui-même qui est manipulé, sous sa forme algorithmique, et non pas sa sémantique comme dans les logiques à la Scott. Un outil implémentant une logique algorithmique aurait donc l'avantage *a priori* de faire manipuler un formalisme proche de celui de la programmation courante.

Ainsi on a dans LA deux connecteurs de « formules algorithmiques » :

$$\Box K\alpha \qquad \text{et} \qquad \Diamond K\alpha$$

où K est un programme (dans son expression concrète, un texte en langage de programmation) et α une formule.

$\Box K\alpha$ est interprétée par : si K s'arrête alors α est vraie après K.

$\Diamond K\alpha$ est interprétée par : K s'arrête *et* α est vraie après K.

LA est puissante : elle contient la logique de Hoare, et des propriétés inexprimables dans cette logique s'énoncent très naturellement par des formules de LA.

Ainsi, une formule $P\{S\}Q$ de LH (si la précondition P est vérifiée et si S s'arrête, alors Q est vraie après S), devient en LA :

$$P \Rightarrow \Box SQ.$$

La propriété de terminaison d'un programme que les formules de LH ne permettent pas de traduire s'exprime directement grâce à l'opérateur $\Diamond$:

$\Diamond K$ Vrai est une formule vraie si et seulement si K s'arrête.

Enfin on peut exprimer l'équivalence de 2 programmes S et S' vis-à-vis du résultat calculé :

si x_0 est une variable qui n'apparaît pas dans les programmes S et S' :

$$\Diamond S(x=x_0) \Leftrightarrow \Diamond S'(x=x_0).$$

Cette formule signifie que la valeur x_0 de x calculée par S est la même que celle calculée par S'; si un des 2 programmes ne s'arrête pas, il en est de même pour l'autre, et on les considère comme équivalents.

Chaque classe de programme manipulée par les constructeurs $\Box$ et $\Diamond$ determine une LA particulière : à une syntaxe de programmes donnée, correspondent les formules d'une LA qui permettra d'axiomatiser les propriétés de ce type de programme.

I.2. Exemple : LA pour programmes WHILE

Considérons une LA pour programmes While.

Le système de déduction comprend un ensemble d'axiomes et de règles dans lesquels on retrouve les axiomes et règles de LH.

Axiomes :

$\Diamond (x := e)\,\alpha \Leftrightarrow \alpha[x/e]$

% axiome pour les programmes élémentaires d'affectation %

$\Diamond$ (begin p; q end) $\alpha \Leftrightarrow \Diamond p(\Diamond q\,\alpha)$

% axiome pour la concaténation de programme %

$\Diamond$ (if T then p else q)

$$\alpha \quad \Leftrightarrow \quad (T \wedge \Diamond p\,\alpha) \vee (\sim T \wedge \Diamond q\,\alpha)$$

% axiome pour le test %.

Règles :

$$\frac{\alpha \quad \alpha \Rightarrow \beta}{\beta} \qquad \frac{\alpha \Rightarrow \beta}{\Diamond p\,\alpha \Rightarrow \Diamond p\,\beta}$$

$$\frac{(\Diamond (\text{if } T \text{ then } p)^i (\sim T \wedge \alpha) \Rightarrow \beta\}_{i<0}}{\Diamond (\text{while } T \text{ do } p)\,\alpha \Rightarrow \beta}$$

La deuxième règle est dite « règle de nécessitation » : elle exprime que si α implique β, alors cette propriété est conservée après l'exécution du programme p.

Enfin la troisième est une règle infinitaire d'induction : il suffit qu'une itération de p termine sur T faux pour que la boucle de p sur T s'arrête. Sachant que le connecteur $\Box$ s'exprime en fonction de $\Diamond$ par :

$$\Box\, a\,\text{alpha} =\tilde{} \Diamond a\,(\tilde{}\text{alpha}),$$

la règle d'induction reste valide en remplaçant $\Diamond$ par $\Box$.

La classe de programmes que nous voulons étudier pour notre système doit cependant être plus large que celle des programmes While.

I.3. Choix d'une classe de programmes

Les programmes que l'on désire étudier avec LA (pour en axiomatiser les propriétés), sont les programmes définis par systèmes de procédures mutuellement récursives sans paramètre. Cette classe de programme est très générale; elle a par exemple été étudiée par De Bakker dans [1].

Plus précidément, notre intérêt porte surtout sur les structures des programmes : l'objectif premier de LA est d'axiomatiser les structures des programmes, comme on le voit dans le système de déduction présenté au paragraphe précédent, où les principaux axiomes et règles portent sur les constructeurs de séquencement (;), test, boucle.

Nous considérerons donc les programmes comme des squelettes ou « schémas » de programme dans lesquels la sémantique des instructions élémentaires n'a pas d'importance.

I.3.*a. EXEL*

Étant donnés le choix de cette classe de programme et notre interêt pour la structure des programmes essentiellement, il était naturel de considérer pour notre logique le langage EXEL, qui possède toutes les caractéristiques que nous venons de mettre en évidence.

EXEL, développé au Laboratoire Central de Recherche de Thomson-CSF à Orsay, est en effet un langage de structures de contrôle. Il contient une forme très générale de boucle (avec « exits » de niveau n) ainsi que la notion d'action; une action est une procédure sans paramètre. Un programme EXEL est défini à partir d'un *système d'actions mutuellement récursives.*

Considérons l'exemple simple d'un système à 2 actions :

$$S = \begin{cases} X_1 = A.B.X_2 \\ X_2 = \langle T \to C \ \diamond X_1 \rangle. \end{cases}$$

A partir d'un système d'actions, un programme est défini par le numéro de l'action point d'entrée. Ici on a donc 2 programmes potentiels $\langle S, 1 \rangle$ commençant par A, B puis appelant X_2; et $\langle S, 2 \rangle$ commençant par le test.

EXEL est un « surlangage » en ce sens que, dans l'exemple ci-dessus, A, B, C et T ne « font pas partie » d'EXEL ce ne sont que de simples « variables de programme » et « variable propositionnelle » dont la signification ne nous importe pas dans l'étude de ce programme. Seules les structures de contrôle

(séquencement. test $\langle \rightarrow \diamond \rangle$, appels d'actions X_i, programme vide VIDE) appartiennent à EXEL.

On définit donc bien avec EXEL les schémas de programme que nous voulons manipuler.

I.3.*b. Sémantique*

La sémantique que l'on choisit pour ces programmes est une sémantique définie par systèmes d'équations récursives dont les variables et les solutions sont des *arbres de programmes.* L'arbre associé à un programme est intuitivement l'ensemble de tous les déroulements ou exécutions possibles de ce programme. Dans le cas des programmes EXEL, cet arbre peut-être infini.

Associer un arbre à un programme c'est définir une sémantique algébrique; celle-ci a été fondée par G. Cousineau dans sa théorie des arbres à feuilles indicées [4].

D'autre part, la constitution du magma des arbres en CPO [4] permet de définir un arbre par un système d'équations. L'association d'un arbre à un schéma de programme EXEL se fait alors naturellement par traduction des opérations sur les schémas (séquencement, test, ...) en opérations sur les arbres (substitution, concaténation, ...) : à un système d'actions tel que ceux définis dans EXEL correspond un système d'équations sur les arbres.

L'interêt de définir la sémantique des programmes par une sémantique de systèmes d'équations est que ces systèmes ont une puissance d'expression dépassant celle du langage choisi, à savoir EXEL : tout langage de programmation pour lequel on peut exprimer un programme sous la forme d'équations algébriques peut avoir la même sémantique.

Dans ce type de sémantique, la sémantique d'un programme récursif est définie en terme de point fixe, soit dans un formalisme fonctionnel à la Scott, soit dans un formalisme de limites de suites d'arbres dans notre cas.

Dans les deux cas, la sémantique d'un programme récursif est toujours la borne supérieure d'une suite d'approximants, arbres ou fonctions.

I.3.*c*

La classe de programmes que nous voudrons traiter par notre système est donc définie : notre logique sera une Logique Algorithmique pour les schémas de programme EXEL; celle-ci a été complétement définie et étudiée par P. Enjalbert (systèmes de déduction consistants et complets) [10, 9].

L'étude des programmes et de leur structure par une logique est par ailleurs plus facile si on s'abstrait des problèmes de manipulation de variables, déjà

traités par les logiques classiques du premier ordre. Les logiques algorithmiques propositionnelles et monadiques (programme à une seule variable ou vecteur de variables, prédicats portant sur cette variable) sont adaptées à ce type d'étude : ce sont typiquement des logiques pour schémas de programme.

Ainsi nous concentrerons notre attention sur les problèmes de structure de contrôle et notamment la récursivité, en écartant des aspects beaucoup plus connus de traitement des variables.

D'autre part on choisit une logique déterministe : il existe en effet par ailleurs des logiques algorithmiques ou dynamiques traitant le non-déterminisme. L'introduction d'un non-déterminisme du type « a ou b » ne poserait pas de problème de fond pour le type d'implantation que l'on veut faire.

Notre Logique Algorithmique est donc une Logique Algorithmique Proportionnelle, déterministe et monadique pour Schémas de programme : on la note LAPD_s.

Abordons alors les problèmes liés à l'implémentation d'un système de preuve pour cette logique algorithmique, en commençant par celui des théories algorithmiques.

I.4. Théories algorithmiques

En logique des programmes il convient de distinguer 2 approches.

– dans les logiques de type logique dynamique, on s'intéresse aux propriétés vraies dans toute structure. Les théorèmes de complétude portent sur les tautologies des modèles contenant l'arithmétique.

– en Logique Algorithmique : on s'intéresse plutôt à des structures particulières, dont on recherche une caractérisation algorithmique. On les décrit sous forme d'un ensemble d'axiomes algorithmiques, exprimant plus naturellement certaines propriétés de calcul que des axiomes du premier ordre (ceux utilisés pour la description par types abstraits par exemple).

Si on considère qu'une théorie algorithmique – un ensemble de formules algorithmiques – décrit une structure, établir une propriété α de cette structure T, c'est prouver que α est vraie dans la théorie T. Autrement dit, c'est montrer que α est conséquence sémantique de T. On note :

$$T = \alpha.$$

Notre système de déduction automatique pour LA devra donc prouver les expressions « $T = \alpha$ », ce qui signifie qu'il nous faudra implémenter des objets permettant d'exprimer ces théories et des opérations pour les manipuler

au cours des preuves. Avant d'examiner les problèmes posés par l'expression des théories (*voir* I. 5), on considère un exemple de l'application la plus immédiate de la notion de théorie algorithmique : l'axiomatisation des structures de données manipulées par les programmes. En effet c'est essentiellement par leur comportement vis-à-vis des opérateurs du langage que l'on caractérise les données définies pour une application, donc par des propriétés de nature algorithmique.

Exemple : théorie des piles

Salwicki a abondamment développé cet exemple d'axiomatisation d'une structure de donnée par une caractérisation algorithmique. On donne ici une version avec variables, dans la logique algorithmique pour programmes While présentée ci-dessus [25].

On considère un ensemble E d'éléments et un ensemble S de piles V_e et V_s sont deux ensembles de variables, désignant respectivement des éléments de E et de S.

On considère trois opérateurs représentant des programmes élémentaires :

$$\text{pop}: \quad S \to S$$
$$\text{push}: \quad E \times S \to S$$
$$\text{top}: \quad S \to E$$

et les prédicats : empty (sur les éléments de S), $=_e$ et $=_s$ (égalité sur E et S).

La théorie ATS (Algorithmic Theory of Stacks) est composée des axiomes suivants :

1. $\Diamond$ (*while* ($\sim$ empty (s)) *do* $s := \text{pop}(s)$) VRAI
2. ($\sim$ empty (s)) $\Rightarrow$ ($s =_s$ push (top (s), pop (s)))
3. $e =_e$ top (push (e, s))
4. $s =_s$ pop (push (e, s))
5. $\sim$ empty (push (e, s))
6. $s =_s s' \Leftrightarrow \Diamond p$ (bool $\wedge$ empty (s_1) $\wedge$ empty (s_2))

où p est le programme :

```
begin s1 := s; s2 := s'; boll := VRAI;
  while (bool ∧ (∼ empty(s1)) ∧ (∼ empty(s2)) do
    bool := bool ∧ (top(s1) =e top(s2));
    s1 := pop(s1);
    s2 := pop(s2)
  end
end
```

L'axiome 6 est une définition algorithmique de l'égalité dans S, qui permet de « calculer » par un programme l'égalité de deux piles.

L'axiome 1 qui établit que toute boucle de dépilement s'arrête permet de sélectionner les modèles standards des piles finies.

Salwicki établit que les modèles de cette théorie propres pour l'identité sont tous isomorphes à un modèle standard dans lequel les piles sont les suites finies de E.

Il montre que l'on peut interpréter les entiers naturels à l'aide des opérateurs de la théorie ATS (l'entier n est représenté par l'empilement de n fois le même élément e_0) et les axiomes de ATS permettent de prouver tous les théorèmes de l'arithmétique algorithmique.

Ces résultats « métamathématiques » permettant de valider l'axiomatisation des piles proposée.

1.5. Non-compacité

La notion de théorie algorithmique nous conduit à aborder le problème de la non-compacité de LA. Rappelons qu'une logique compacte vérifie : pour toutes théorie T et formule α telles que $T \models \alpha$, il existe une théorie finie $T_f \subset T$ telle que $T_f \models \alpha$. Considérons un exemple :

Soient la théorie :

$$T = \{Q \Rightarrow \Box A^n(\sim Q)\}_{n<0}$$

le système S à une seule action :

$$S = \{X_1 = A.\ \langle Q \rightarrow \text{VIDE} \diamond X_1 \rangle\}$$

la formule α :

$$\alpha \equiv Q \quad \Rightarrow \quad \Box \langle S, 1 \rangle (\sim Q).$$

La formule alpha exprime, dans T, que $\langle S, 1 \rangle$ boucle; en effet $\langle S, 1 \rangle$ est une itération de A jusqu'à ce que Q devienne vrai, et l'axiome de T traduit la propriété qu'après toute itération de A, Q est faux. Or Q est précisément la condition de sortie de boucle du programme.

α est donc vraie dans T: $T \models \alpha$.

Or T est infinie et si on réduit T en une théorie finitaire T_f, qui sera du

type :

$$T_f = \{Q \Rightarrow \Box A^n(\sim Q)\}_{n < n0}$$

quelque soit $n0$, on n'a plus $T_f \vdash \alpha$. En effet dans toute théorie T_f, on ne sait rien du comportement de Q au-delà de l'itération $n0$ de A.

Ceci signifie que la propriété α, conséquence sémantique de T, n'est réductible à aucun sous-ensemble fini de T. Dans la preuve d'une telle propriété on devra donc faire appel à un moment donné à un ensemble infini de formules de T.

LA est donc non compacte. La conséquence de cette non-compacité sur les systèmes de déduction pour une telle logique est la suivante : ces systèmes devront comporter une règle à nombre infini de prémisses pour prendre en compte de tels ensembles infinis de formules, nécessaires à la preuve de certaines propriétés. Les systèmes de déduction *complets* pour LA comporteront donc une *règle infinitaire.*

On a déjà vu (§ I.2) une règle infinitaire pour les programmes While. Elle a la forme suivante pour les systèmes de procédures mutuellement récursives sans paramètre :

$$\frac{\{\alpha \Rightarrow \Box \mu_i^n S\, \beta\}_{n \geq 0}}{\alpha \Rightarrow \Box \langle S, i \rangle \beta}.$$

Dans cette règle apparaît une expression de programme : $\mu_i^n S$. Elle désigne le « n-ième approximant » du programme $\langle S, i \rangle$.

La notion d'approximant, dans une sémantique à base d'arbres ou autre, correspond à un déroulement du programme récursif pour lequel la profondeur de récursivité est limitée à n. La sémantique d'un programme récursif est alors définie comme la limite de la suite de ces approximations.

La règle d'induction permet d'établir les propriétés d'un programme récursif à partir de celles de tous ses approximants : si tous les approximants d'un programme satisfont une spécification $\{\alpha, \beta\}$, alors le programme tout entier satisfait cette même spécification.

1.6

Nous pouvons maintenant résumer les problèmes à résoudre pour implémenter cette LA pour schémas de programme :

— Notre objectif est d'axiomatiser l'opération de conséquence sémantique, c'est-à-dire prouver des expressions $T \vdash$ alpha.

– Nous devons manipuler des ensembles infinis de formules, soit dans la règle infinitaire d'induction, soit dans certaines théories : sous forme de suites de formules « compteur », comme celle du contre-exemple de non-compacité $(Q \Rightarrow \Box A^n(\sim Q))$, ou plus classiquement sous forme de schémas d'axiomes instanciés par n'importe quel n-uplet de formules (*voir* théorie algorithmique de l'arithmétique en Annexe).

Pour le deuxième point nous devrons formaliser la présence d'entiers dans les formules, en introduisant un nouveau type de variables, et la notion de schéma d'axiome engendrant l'ensemble de ses instanciations.

Ceci nous conduit donc à définir une LA pour schémas de programme dérivée de $\mathrm{LAPD_s}$: $\mathrm{LAPD_s E}$, Logique Algorithmique Propositionnelle, Déterministe et monadique pour Schémas de programme avec Entiers.

I.7. $\mathrm{LAPD_s E}$

Pour les 2 problèmes exposés ci-dessus, on présente la solution proposée dans $\mathrm{LAPD_s E}$.

Premier point : axiomatisation de $T \vDash \alpha$. Puisque c'est ce type de propriété que l'on veut établir, on intègre les théories au système de déduction; on manipulera donc un couple :

$$T \vdash\!\!\rightarrow \alpha$$

et ce sont ces expressions que l'on prouvera avec notre système; leur sémantique est celle de $T \vDash$ alpha.

Pour le point suivant, théories et règles infinitaires, on introduit une composante « entier » dans les programmes, à l'aide des opérateurs :

$$\mathrm{Iter}(a, k) \quad \text{et} \quad \mathrm{Approx}(S, i, k)$$

Iter permet de coder les itérations de programme : à un programme a et un entier k on associe un programme consistant à exécuter a k fois. Approx (S, i, k) représente l'approximant de rang k du programme représenté par le schéma $\langle S, i \rangle$.

En quantifiant les variables d'entier dans les formules contenant de telles expressions de programme, on obtiendra bien l'expression d'ensembles infinis de formules, formés à partir de suites de programmes.

En ce qui concerne les schémas d'axiome, on les définit comme des applications n-aires dans l'ensemble des formules de $\mathrm{LAPD_s E}$. Pour utiliser un schéma d'axiome d'arité n, on manipulera des expressions $s(f1, \ldots, fn)$,

application du schéma s au n-uplet $(f1, \ldots, fn)$. On construira ces applications à partir de schémas de base définis à l'aide des connecteurs et des constructeurs de programme de LAPD$_S$E.

Avec cette syntaxe, la théorie infinie T du contre-exemple de non-compacité s'exprime dans LAPD$_S$E par l'unique formule quantifiée :

$$T = \{\forall k \,.\, Q \Rightarrow \Box \,\mathrm{Iter}(A,\ k+1)(\sim Q)\}$$

(k prenant ses valeurs à partir de 0) et la règle infinitaire pour les schémas de formule par :

$$\mathrm{IND\_RE}: \frac{Z \mapsto \forall k\, \alpha \quad \Rightarrow \quad \Box\, \mathrm{Approx}(S,\ i,\ k)\, \beta}{Z \mapsto \alpha \quad \Rightarrow \quad \Box (S,\ i)\, \beta}$$

où apparaissent les expressions $Z \mapsto \alpha$ du système de déduction.

Bien sûr ce codage ne permet pas d'exprimer par des théories finies avec variables d'entiers n'importe quel ensemble infini de formules algorithmiques « classiques ». Cependant il résout nombre de cas courants. Ainsi, dans le contre-exemple utilisé pour la non-compacité, on trouve effectivement avec ce nouveau formalisme le moyen d'exhiber une théorie finitaire dans laquelle alpha est vraie; et on pourra faire une preuve de alpha au moyen de la nouvelle règle IND_RE.

1.8. Syntaxe et Sémantique de LAPD$_S$E

Les programmes de LAPD$_S$E sont construits à partir des programmes élémentaires VIDE et oméga (programmes vide et indéfini), un ensemble V_p de variables de programme que l'on trouve dans les schémas : A, B, C... et un ensemble V_0 de prédicats ou variables propositionnelles : P, Q, R, ... avec les opérateurs suivants :

— les constructeurs de test et séquence :

$$\langle T \to a \diamond b \rangle \quad \text{et} \quad a.b$$

— les opérateurs Iter(a, e) et Approx(S, i, e), *où* e est une « expression » d'entier, formée à partir d'un ensemble de variables d'entier Ent, de 0, 1 et $+$, a est un programme sans schéma $\langle S, i \rangle$

— enfin les schémas de programme EXEL $\langle S, i \rangle$ définis par systèmes d'actions.

(Remarque : dans les 2 premières clauses, on désigne des programmes non récursifs directement par le corps de leur action unique.)

L'ensemble F_E des formules est construit à partir des constantes Vrai, Faux et des variables propositionnelles avec :

- les connecteurs propositionnels $\wedge$, $\vee$, $\sim$, ...
- le connecteur algorithmique $\square\, a\, \alpha$
- la quantification $\forall k_1 \ldots \forall k_n \alpha$.

Cette quantification est limitée : elle est regroupée à gauche (quantification universelle qui ne peut être à l'intérieur d'aucun autre opérateur que lui-même). Sous cette restriction, elle exprime bien ce que nous voulons coder : des ensembles infinis de formules indexés par les entiers.

La sémantique des formules de Logique Algorithmique est généralement définie par une structure composée d'un domaine D et de fonctions sur ce domaine interprétant chaque variable de programme et prédicat, soit :

$$\mathcal{I} = (D, I_1, I_2)$$

où

$$I_1 : \quad V_0 \to (D \to \{V, F\})$$
$$I_2 : \quad V_p \to (D \to D).$$

La sémantique d'un schéma de programme est définie à partir de l'arbre associé : la valeur renvoyé par le programme a à partir de $d \in D$ est le résultat du parcours de cet arbre (éventuellement infini) en effectuant les opérations et tests définis par I_1 et I_2.

La sémantique des formules algorithmiques est :

dans $\mathcal{I}$, $\square\, a\, \alpha$ est vraie en $d \in D$ si et seulement si : ou bien la valeur d' calculée par a à partir de d existe (a s'arrête en d) et α est vraie en d'; ou bien le programme a ne s'arrête pas en d.

Dans notre cas la structure sémantique est une structure à 2 types dont l'un est l'ensemble des entiers naturels N. Les programmes élémentaires et les prédicats sont interprétés dans D. Les variables d'entier de LAPD$_s$E sont valués par une fonction φ à valeurs dans N, telle que $\varphi(0) = 0$ et $\varphi(1) = 1$ étendue aux expressions d'entier par : $\varphi(h+k) = \varphi(h) + \varphi(k)$. Un modèle de LAPD E est donc un doublet $(\mathcal{I}, \varphi)$ où $\mathcal{I}$ est la structure (D, I_1, I_2) définie ci-dessus.

La sémantique des expressions « $\forall k\, \alpha(k)$ » est définie comme la conjonction de toutes les instanciations $\alpha(n)$, quelle que soit la valeur n donnée par phi à k : $\forall k\, \alpha(k)$ est vraie si et seulement si chacun des $\alpha(n)$, où n est dans le domaine des entiers, est vraie.

Les théories de $\mathrm{LAPD_sE}$ sont des ensembles finis de schémas d'axiome. Un schéma d'axiome est une application n-aire dans F_E définie à partir des schémas constants que sont les formules de F_E, de l'identité, et des constructeurs de formules de $\mathrm{LAPD_sE}$.

Exemples :

$$s: \quad \gamma \to \forall k \, \Box \, \mathrm{Iter}(A, k+1)\gamma \quad \Leftrightarrow \quad \Box \, \mathrm{Iter}(A, k)\gamma$$

$$s: \quad \gamma \to \Box \, A.B\gamma \quad \Leftrightarrow \quad \Box \, B.A\gamma.$$

1.9. Système de déduction pour $\mathrm{LAPD_sE}$

AXIOMES :

– CP : Ensemble des axiomes du calcul propositionnel classique.

– Axiomes algorithmiques : (a, b sont des schémas de programme, A une variable de programme, S un système d'actions, α, β des formules, e une expression d'entier; si S a n actions, $i \in [1, n]$)

ALG1 $\quad \varnothing \mapsto \Box \, a(\alpha \wedge \beta) \quad \Leftrightarrow \quad (\Box \, a\alpha \wedge \Box \, a\beta)$

ALG2 $\quad \varnothing \mapsto \sim (\Box \, A\alpha) \quad \Leftrightarrow \quad \Box \, A(\sim \alpha)$

ALG3 $\quad \varnothing \mapsto \Box \, \mathrm{VIDE}\,\alpha \quad \Leftrightarrow \quad \alpha$

ALG4 $\quad \varnothing \mapsto \Box \, \mathrm{omega}\,\alpha$

ALG5 $\quad \varnothing \mapsto \Box \langle T \to a_1 \Diamond a_2 \rangle \alpha$

$$\Leftrightarrow \quad (T \Rightarrow \Box \, a_1 \alpha) \wedge (\sim T \Rightarrow \Box \, a_2 \alpha)$$

ALG6 $\quad \varnothing \mapsto \Box \, a_1 . a_2 \alpha \quad \Leftrightarrow \quad \Box \, a_1 \Box \, a_2 \alpha$

ALG7 $\quad \varnothing \mapsto \Box \, A . a\alpha \quad \Leftrightarrow \quad \Box \, A \Box \, a\alpha$

ALG8 $\quad \varnothing \mapsto \Box \langle S, i \rangle \alpha \quad \Rightarrow \quad \Box \, \mathrm{Approx}(S, i, e)\alpha.$

Dans cette liste, $\varnothing$ désigne la théorie vide : on a ici les axiomes classiques de la logique algorithmique pure. Les deux premiers définissent le comportement des programmes vis-à-vis des connecteurs propositionnels; le troisième axiome exprime la « neutralité » du programme VIDE; le deuxième, la non-terminaison du programme indéfini (s'il terminait, il rendrait le faux vrai); enfin le dernier exprime le fait que tout programme récursif est plus défini que chacun de ses approximants (avec une sémantique en terme d'arbre, on a un ordre sur les arbres où oméga est le plus petit élément, et qui fait de cet ensemble un CPO).

Axiomes structurels :

TH1 $Z\,U\,\{\text{alpha}\} \mapsto \text{alpha}$

TH2 $\{f\} \mapsto f(\gamma_1, \gamma_2, \ldots, \gamma_n)$

(f schéma d'axiome d'arité n, $(\gamma_1, \ldots, \gamma_n)$ n-uplet de F_E^n).

On doit munir ce système d'axiomes pour traiter les expressions d'entier (associativité, commutativité de $+$, propriétés de 0 et 1), et définir les opérateurs Iter et Approx.

ENT1 $\varnothing \mapsto \alpha(t+t') \Leftrightarrow \alpha(t'+t)$

ENT2 $\varnothing \mapsto \alpha(t+0) \Leftrightarrow \alpha(t)$

ENT3 $\varnothing \mapsto \alpha((t+t')+t'') \Leftrightarrow \alpha(t+(t'+t''))$

ENT4 $\varnothing \mapsto \Box\,\text{Iter}(a, 0)\,\alpha \Leftrightarrow \Box\,\text{VIDE}\,\alpha$

ENT5 $\varnothing \mapsto \Box\,\text{Iter}(a, 1)\,\alpha \Leftrightarrow \Box\,a\,\alpha$

ENT6 $\varnothing \mapsto \Box\,\text{Iter}(a, t+t')\,\alpha \Leftrightarrow \Box\,\text{Iter}(a, t)\,\Box\,\text{Iter}(a, t')\,\alpha$

ENT7 $\varnothing \mapsto \Box\,\text{Approx}(S, i, 0)\,\alpha$.

Si a_i est le corps de la i-ème équation du système S de rang n et $\alpha_i(x_1 \ldots x_n)$ est le programme résultant de la substitution des n variables de l'action i par le n-uplet de programmes $(x_1 \ldots x_n)$:

*ENT*8 $\varnothing \mapsto \Box\,\text{Approx}(S, i, 1)\,\alpha \Leftrightarrow \Box\,a_i(\text{omega}^n)\,\alpha$

ENT9 $\varnothing \mapsto \Box\,\text{Approx}(S, i, t+1)\,\alpha$

$$\Leftrightarrow \Box\,a_i(\text{Approx}(S, j, t)_{j=1\ldots n})\,\alpha.$$

Règles :

MP $$\frac{Z \mapsto \alpha \quad Z \mapsto (\alpha \Rightarrow \beta)}{Z \mapsto \beta}$$

NEC $$\frac{Z \mapsto (\alpha \Rightarrow \beta)}{Z \mapsto (\Box\,a\,\alpha \Rightarrow \Box\,a\,\beta)}$$

GEN $$\frac{Z \mapsto \alpha(k)}{Z \mapsto V\,k\,\alpha(k)} \quad k \text{ non libre dans } Z$$

INST $$\frac{Z \mapsto \forall\, k\,\alpha(k)}{Z \mapsto \alpha(t)}$$

ADD $$\frac{Z \mapsto \alpha}{Z \cup \{\beta\} \mapsto \alpha}$$

SCHEM $$\frac{Z \mapsto \forall k\, \beta \Rightarrow \Box a \Box \operatorname{Approx}(S, i, k)\beta}{Z \mapsto \Box a \Box \langle S, i \rangle \beta}$$

alpha, beta, a sans k libre.

IND $$\frac{Z \mapsto \alpha(0)\ Z \cup \alpha(k) \mapsto \alpha(k+1)}{Z \mapsto \forall k\, \alpha(k)}$$

k non libre dans Z.

A côté des règles algorithmiques, on retrouve des règles classiques d'instantiation et de généralisation des variables libres (GEN, INST), une règle structurelle permettant d'introduire des axiomes dans le champ théorie (ADD). De plus, on remarquera :

— les manipulations auxquelles donnent lieu les théories;

— une expression généralisée de la règle d'induction sur les approximants, qui est devenue une règle finitaire (baptisée SCHEM);

— l'application d'une règle du type recurrence sur les entiers qui s'avère nécessaire pour faire la preuve des formules quantifiées. C'est en fait la véritable règle d'induction de notre système (un « schéma d'induction ») et pour cette raison c'est elle que l'on a nommée IND;

— l'apparition (du fait des prémisses de la règle IND) de théories paramétrées par les entiers, ce qui impose des vérifications sur la présence de variables libres dans les théories lors de l'application de certaines règles (k non libre dans Z).

La règle d'induction a la signification suivante :

si α (0) est conséquence sémantique de Z, si α $(k+1)$ est conséquence de Z et de l'axiome supplémentaire $\alpha(k)$, alors $\forall k$ α (k) est une conséquence sémantique de Z.

La sémantique donnée aux formules « $\forall k$ α (k) » assure la validité de cette règle.

THÉORÈME : *Le système de déduction ci-dessus est consistant.*

I. 10. Problèmes de complétude

L'introduction dans la syntaxe de $\mathrm{LAPD_sE}$ de variables qui sont interprétées dans les entiers standard rend évidemment le système de déduction ci-dessus non complet.

Cependant, du fait de la puissance d'expression du language, la preuve de cette non-complétude est non triviale : en effet la méthode classique consistant à passer par la non-compacité devient difficile à utiliser, car la plupart des contre-exemples de théories infinitaires exhibées (telles que $T \models \alpha$ et $\sim(T_f \models \alpha)$) tombent de par notre codage : il devient possible de les exprimer par un ensemble fini d'axiomes (*cf.* I. 7). Nous devons alors chercher un autre moyen de mettre en évidence la non-complétude.

Nous utilisons des travaux réalisés par G. Mirkowska.

G. Mirkowska définit dans une LA du même type que $\mathrm{LAPD_s E}$ (propositionnelle et déterministe) une axiomatisation de l'arithmétique, la théorie Axar.

La LA utilisée est « traduisible » dans $\mathrm{LAPD_s E}$; nous pouvons alors définir une théorie de l'arithmétique qui est la traduction de Axar dans $\mathrm{LAPD_s E}$. On note TN cette théorie propositionnelle de l'arithmétique, qui nous permet au passage de montrer la puissance d'expression de $\mathrm{LAPD_s E}$.

G. Mirkowska établit un résultat de *programmabilité des fonctions partielles récursives* (pr) dans sa logique, résultat que nous pouvons également transporter dans $\mathrm{LAPD_s E}$ et qui y prend la forme :

pour toute fonction pr f et tout entier x, il existe un terme x de $\mathrm{LAPD_s E}$ qui code x, et une formule α_x de $\mathrm{LAPD_s E}$ contenant ce terme, telle que :

$$f(x) \text{ non défini} \quad \Leftrightarrow \quad \models \mathrm{TN} \mapsto \alpha_x.$$

La formule α_x est définie par :

si n est la dimension du domaine de f,

si $x = (x_1, \ldots, x_n)$,

soient $P_1, \ldots, P_n$ des variables de programmes interprétées comme la décrémentation de chacune des composantes du domaine,

soient $z_1, \ldots, z_n$, des prédicats interprétés par :

$$zi(x) = V \quad \Leftrightarrow \quad x_i = 0$$

(Remarque : les *zi et Pi* sont axiomatisés dans *TN*.)

soit M le programme qui calcule $f(x)$ (G. Mirkowska construit M pour chaque fonction primitive récursive), alors :

$$\alpha_x = \quad P1^x \ldots \mathrm{Pn}^x(z1 \wedge \ldots \wedge zn) \quad \Rightarrow \quad \sim \Diamond M \,\mathrm{Vrai}.$$

C'est à partir de ce résultat que nous prouvons la non-complétude :
on sait qu'il existe au moins une fonction pr f dont le domaine de non-définition est non récursivement énumérable (re); ce qui implique, d'après la programmabilité de toute fonction partielle récursive, que l'ensemble des formules α_x est non re.

Donc

$$\{\alpha / \vDash TN \mapsto \alpha\}$$

est non re. Il en est de même de :

$$\{Z \mapsto \alpha / \vDash \mapsto \alpha\}.$$

Notre système de déduction, finitaire, ne peut donc pas générer un tel ensemble de formules. Il existe donc des expressions $Z \mapsto \alpha$ pour lesquelles on ne peut pas prouver $\vDash Z \mapsto \alpha$. Notre système de déduction est incomplet.

(*Voir* en Annexe la présentation de Axar et *TN*.)

I. 11. Origine de la non-complétude : dérivation de LAPD$_s$E′

C'est le caractère finitaire de notre système de déduction qui le rend incomplet, et non pas l'éventuel oubli d'axiomes ou de règles sur les constructeurs de formules ou de programmes. Pour le prouver, on étudie une variante de LAPD$_s$E. Dans cette nouvelle logique LAPD$_s$E′, les programmes sont construits de la même façon avec des variables d'entier, mais on réintroduit des théories infinitaires, en remplaçant dans les théories de LAPD$_s$E le codage « $\forall k\, \alpha\, (k)$ » par l'ensemble des instanciations de $\alpha(k)$ sur N; et on remplace le schéma d'indication IND par une véritable ω-règle au sens de l'arithmétique, du type :

$$\frac{\{P(n)\}_{n \in N}}{\forall x\, P(x)}.$$

On montre que le système de déduction ainsi généré pour cette nouvelle logique est complet : la preuve se fait selon une méthode classique en logique algorithmique, consistant à construire l'algèbre de Lindenbaum d'une théorie.

La dérivation de la logique LAPD$_s$E′ à partir de LAPD$_s$E consiste à faire « réapparaître l'infini » dans le système de déduction. Puisque cette opération suffit à rétablir la complétude, nous pouvons conclure que la non-complétude provient effectivement de la réduction finitaire opérée par le codage de LAPD$_s$E.

Le type de raisonnement que nous avons adopté pour localiser la non-complétude de $LAPD_sE$ est à comparer avec une démarche plus habituelle adoptée en Logique de Hoare et Logique Dynamique, et consistant à établir la complétude par rapport à un système d'axiomes enrichi de toutes les formules valides dans les entiers standard (Cook-complétude).

En ce sens, la démonstration proposée ici nous paraît une alternative aux techniques des complétude relative.

Salwicki a par ailleurs montré [25] que le recours à une règle infinitaire permet d'obtenir des résultats plus fins de complexité logique que le plongement dans la théorie de l'arithmétique.

I. 12. Conclusion de la première partie

La logique que nous venons de définir va nous permettre de réaliser un outil de déduction pour les expressions $Z \mapsto \alpha$ dans lequel nous pourrons faire des preuves d'expressions $Z \models \alpha$.

Bien sûr le système ainsi implémenté ne nous permettra pas de démontrer toute formule $Z \models \alpha$ vraie puisqu'il est incomplet. Cependant, d'après les problèmes rencontrés lors de la preuve de non-complétude, on peut penser qu'il sera difficile pour un utilisateur courant de tomber sur les formules « improuvables ».

DEUXIÈME PARTIE

La deuxième partie expose la réalisation du système LAPD d'aide à la déduction et à la preuve dans la logique $LAPD_sE$.

Les spécifications de cet outil sont constituées des fonctionnalités suivantes :

- reconnaître le langage de $LAPD_sE$
- offrir les axiomes et les règles de $LAPD_sE$
- proposer des stratégies de preuve.

L'utilisateur pourra alors définir une théorie T et prouver des théorèmes $T \mapsto \alpha$.

LAPD a été réalisé à partir du système LCF, sur VAX780/UNIX ([2]) du Greco de Programmation du C.N.R.S. à l'Université de Bordeaux, version LCF 4.2; une présentation rapide de LCF permettra de faire comprendre les

([2]) UNIX : marque déposée de Bell Laboratories.

principes de fonctionnement de ce système, et en particulier l'interface utilisateur et la manipulation de théories, principes qui ont été repris pour concevoir LAPD.

II. 1. LCF

LCF (logic for Computable Functions) est un système qui permet de faire de la déduction et de la preuve dans la logique de Scott pour les termes du λ-calcul typé. Les travaux autour de LCF ont débuté à Stanford et se sont poursuivis à l'université d'Edimburgh puis Cambridge, et en France à l'I.N.R.I.A.

Du point de vue de l'utilisateur, LCF est structuré en 2 parties :

Un langage « objet » dans lequel sont écrits les termes, formules et théorèmes de la théorie définie par l'utilisateur.

Un langage de commande ou Meta-Langage ML qui permet de manipuler les objets de la logique et en particulier les théorèmes pour faire des preuves.

Conçu à l'origine à partir des besoins d'un langage de commande pour élaborer les preuves de LCF, il a subi une évolution propre développant ses caractéristiques premières. Actuellement, ML est un langage de haut niveau, fonctionnel et typé, dans lequel des mécanismes tels que polymorphisme, inférence de type et récupération d'erreurs accroissent à la fois la puissance d'expression et la souplesse d'utilisation. D'abord implémenté en Lisp, ML dispose maintenant de compilateurs propres.

Le fonctionnement de LCF tel qu'il est décrit ci-dessus correspond à ce que nous voulons réaliser pour la Logique Algorithmique. En particulier, il faut noter que LCF n'est pas un démonstrateur automatique.

Plus concrètement, comment se déroule une cession de travail dans LCF ?

Une théorie est définie par son langage : des types, des constantes, des prédicats, selon la syntaxe du λ-calcul typé; et par ses axiomes écrits dans ce langage.

Dans ML, les objets de base sont les fonctions typées. Des types élémentaires sont prédéfinis : int, bool, string (ou token), ainsi que des types spécifiques du langage objet manipulé : *term*, *form*, *thm*. Quand on définit un terme, une formule ou quand on génère un théorème dans une théorie, on crée des objets de ML qui ont respectivement le type *term*, *form*, *thm*.

Des fonctions possédant ces types dans leur type argument permettront de les manipuler. Ces fonctions sont par exemple des règles de déduction.

En effet une règle est une opération qui à partir d'un théorème (ou plusieurs) fournit un nouveau théorème. Ainsi Modus Ponens construit à partir de 2 théorèmes : $\vdash \alpha$ et $\vdash \alpha \Rightarrow \beta$ le nouveau théorème $\vdash \beta$.

Dans ML, MF sera donc une fonction de type : thm # thm → thm (où # est le constructeur du type produit cartésien, → est le constructeur du type fonction, fondamental dans ML. Un 3e constructeur est *list*, à un argument).

Toutes les règles de déduction de la logique de Scott sont implémentée comme des fonctions de ML prédéfinies.

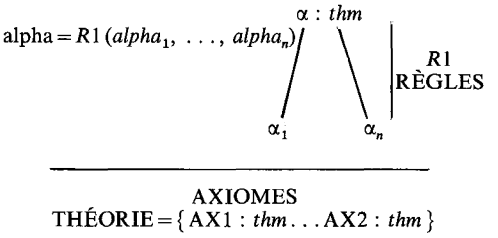
Ce sont ces opérations de base qui permettent de construire les preuves. Ces preuves sont élaborées grâce aux puissantes propriétés de fonctionnalité et de typage de ML.

II. 1. 1. *Preuves*

Que fait-on quand on prouve un théorème dans une théorie T?

On fixe d'abord un objectif soit à prouver le théorème $\vdash \alpha$. Dans LCF, il s'agit de générer un objet de ML qui aura le type *thm*.

Si on considère le processus de déduction d'un théorème α comme un arbre inversé :



les régles apparaissent comme les opérations qui permettent de remonter niveau par niveau, à partir des axiomes jusqu'au théorème à prouver.

La preuve de alpha consiste à mettre en évidence cet arbre de déduction : les feuilles qui sont des axiomes de la théorie, et la succession des règles permettant de progresser des feuilles vers la racine.

La recherche au hasard des axiomes et des règles pour construire l'abre de bas en haut risquant d'être longue et (dans un système automatisé) couteuse, la démarche naturelle de preuve adoptée dans LCF est de procéder en sens inverse : de la racine vers les feuilles.

Les objets de ML mettant en œuvre cette démarche sont les « tactiques ».

II.1.2. *Tactiques*

Une tactique consiste à décomposer la formule à prouver selon divers critères et générer des sous-objectifs plus élémentaires, jusqu'à obtenir des sous-objectifs dans lesquels on reconnaît les axiomes (les feuilles de l'arbre). Une tactique est donc une opération de descente dans l'arbre de déduction.

Les critères utilisés pour décomposer une formule consistent généralement à reconnaître dans la formule la structure de la conclusion d'une des règles de déduction; les sous-buts générés sont alors les prémisses correspondants. Les tactiques de base de LCF sont tout simplement des « inversions » de règles.

Dans LCF ce seront des fonctions de ML du type

$$\text{form} \rightarrow \text{form list}.$$

C'est bien un type d'inversion de règle (thm list → thm).

Une fois trouvés les axiomes et les règles constituant les feuilles et les « branches » de l'arbre de déduction, l'établissement du théorème proprement dit se fait par propagation de l'attribut *thm* depuis les axiomes jusqu'à la formule à prouver, en le remontant dans l'arbre.

Une tactique doit donc contenir le processus de remontée de cet attribut, qui accompagne la décomposition d'un but; ce processus prend le nom « proof » dans LCF. Pour les tactiques élémentaires, il correspond bien sûr à l'application d'une règle.

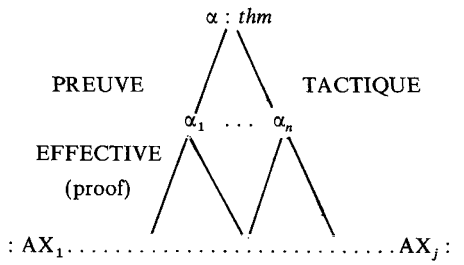
Les 2 types de base du processus de preuve dans LCF sont donc :

tactic : form → form list # proof

proof : thm list → thm

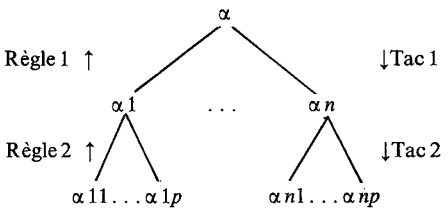
où l'on voit en quoi la fonctionnalité de ML répond aux besoins d'un démonstrateur de théorèmes.

Si on revient à l'arbre de déduction on a le schéma :



A partir des tactiques de base qui inversent les règles, ML permet de construire des tactiques raffinées. Ici encore c'est le caractère fonctionnel de ML qui offre de multiples possibilités.

On peut par exemple composer les tactiques en utilisant l'opérateur THEN :
soient tac1 et tac2 2 tactiques; alors tac1 THEN tac2 a l'effet schématisé par l'arbre suivant :



Par application de tac2 à chacun des sous-buts générés par tac1, on obtient l'ensemble de sous-buts $\{ \alpha 11, \ldots, \alpha 1p, \ldots, \alpha n1, \ldots, \alpha np \}$; la preuve associé est la composition des 2 preuves R1 et R2 : R1 ∘ R2 (cela correspond effectivement à une composition de fonctions dans ML).

Le type de l'opérateur THEN est tactic # tactic → tactic.
On peut raffiner la composition de tactique en ramplaçant tac2 par une liste de tactiques, chacune d'elles s'appliquant à l'un des sous-buts générés par tac1 :

THENL : tactic # (tactic list) → tactic

(list : constructeur du type liste polymorphique dans ML).

L'opérateur OR permet de « tester » des tactiques : Tac1 OR T renvoie le résultat de Tac1 s'il n'y a pas d'échec lors de son application, sinon le résultat de Tac2. ML utilise ici son mécanisme de génération et récupération d'erreurs.

Enfin l'opérateur REPEAT tac permet d'appliquer la tactique itérativement jusqu'au dernier pas sans échec.

Tous ces mécanismes de composition et enchaînement des tactiques nous intéressent : ils sont très généraux, simples à utiliser et composables à merci; ils remplissent l'objectif que nous nous fixons : fournir des moyens faciles et puissants d'assistance à la preuve. Pourquoi dans ce cas ne pas réutiliser directement LCF avec des mécanismes performants déjà implémentés ?

II.2. Spécificités de la logique algorithmique

En logique de Scott, la règle permettant de raisonner sur la sémantique des programmes récursifs est une règle d'induction du point fixe :

$$\frac{P(UU)\ P(X) \Rightarrow P(F(X))}{P(\mathrm{FIX}\ F)}.$$

Dans cette règle les arguments du prédicat P sont des termes (prédicat du premier ordre).

Notre règle d'induction sur les schémas de programme :

SCHEM : $$\frac{\forall k\, \alpha \Rightarrow \Box\, \mathrm{Approx}(S, i, k)\ \beta}{\alpha \Rightarrow \Box \langle S, i \rangle\ \beta}$$

est la « transposée » dans notre logique de la règle d'induction du point fixe.

Le connecteur de formule $\Box$ est ici du deuxième ordre. Si on veut conserver le caractère intuitif de LA, avec expression directe des programmes, il faut donc représenter une règle comportant un prédicat du second ordre : LCF ne le permet pas, car il implémente une logique du premier ordre. Pour utiliser directement le système de déduction de LCF, il faudrait implémenter $\mathrm{LAPD_s E}$ en réalisant un codage du constructeur $\Box$ qui dénaturerait la logique algorithmique prise comme point de départ de cette étude.

Il nous faut donc reconstruire un système qui sera du type de LCF, mais avec des formules et des règles de déduction spécifiquement algorithmiques. Cependant, pour conserver les avantages du langage ML, et les principes de LCF, il sera conçu autour de LCF.

II.3. Système LAPD

Nous avons donc réalisé un système d'aide à la déduction et à la preuve dans $\mathrm{LAPD_s E}$, qui s'appelle LAPD, et dont le langage de commande est, comme dans LCF, ML.

Cet outil reconnaît le langage de $\mathrm{LAPD_s E}$: toutes les structures des programmes, formules, théorèmes, etc. sont définies comme des types de ML. On aura donc dans $\mathrm{LAPD_s E}$ de nouveaux types prédéfinis :

programme, formule, theorème.

Les règles de déduction de $\mathrm{LAPD_s E}$ sont définies comme des fonctions sur ces types.

On y associe des tactiques élémentaires « algorithmiques » qui seront les briques de construction des preuves à l'aide des mécanismes déjà vus dans LCF.

Comme dans LCF, l'utilisateur commencera par définir sa théorie algorithmique, puis programmera ses preuves. Considérons en premier lieu comment sont définies les théories dans LAPD.

II.3.1. *Théories, théorèmes*

Une théorie algorithmique est un ensemble fini de schémas d'axiomes avec méta-variables de formules; dans LAPD, les schémas d'axiome sont implémentés par le type *sf* (schémas de formules) et construits par composition à partir de schémas de base (*cf.* I.8). Les axiomes eux-mêmes (objets de type *thm*) sont obtenus par application d'un schéma à un ensemble de formules variables.

Les sf peuvent contenir des variables d'entier et être quantifiés (concrètement il s'agit de fonctions de ML dont les arguments sont des formules).
Le type des théories est donc :

$$\textit{théorie} = \text{sf list}.$$

La structure des expressions du système de déduction nous conduit à manipuler ces objets théories; en particulier si on considère le schéma d'induction sur les entiers :

$$\frac{Z \mapsto \alpha(0)\ \ Z \cup \alpha(k) \mapsto \alpha(k+1)}{Z \mapsto \forall k\, \alpha(k)}$$

on voit que le théorie est un champ variable au travers des règles de déduction. Il faudra donc l'inclure dans les arguments des règles de déduction, c'est-à-dire dans les théorèmes. De plus il faudra réaliser un contrôle sévère sur les entiers libres de ces théories.

Les théorèmes $Z \mapsto \alpha$ de LAPD sont donc des objets du type :

$$\textit{théorème} = \text{théorie } \# \text{ formule}.$$

En fait pour l'implémentation du système de déduction on est amené à préciser cette structure et on adopte la solution choisie dans LCF induite des principes de la déduction naturelle : on scinde la formule en hypothèses et conclusion; ceci permet d'isoler l'ensemble des hypothèses qui sont transmises sans modification le long de l'arbre de déduction. Le type théorème est donc

finalement :

$$\textit{théorème} = \text{théorie } \# \text{ fsq liqt } \# \text{ formule}$$

$$\text{th} = Z \mapsto A\,1 \wedge \ldots \wedge A\,n \Rightarrow f$$

où le type fsq désigne les formules sans quantificateur, condition sur les hypothèses induite par le type de quantification dans LAPD_sE.

II.3.2. *Axiomes, règles, tactiques*

Les *règles* sont des fonctions du type : théorème list → théorème.

La plupart des axiomes propositionnels sont transformés en règles de déduction, comme dans LCF. La partie calcul propositionnel du système de déduction de LAPD est donc assez proche de celle de LCF.

Les tactiques permettent de réaliser des opérations du type :

$$(Z, \text{hyp}, \alpha) \mapsto [\underset{P}{Z\,1}, \text{hypl}, \alpha\,1 ; \ldots ; Z\,n, \text{hyp}\,n, \alpha\,n]$$

où P est la preuve du théorème (Z, hyp, α) à partir des théorèmes $\{Z\,i, \text{hyp}\,i, \alpha\,i\}$.

La tactique IND_TAC associée à la règle IND fournit le résultat :

$$[Z, \varnothing, \alpha(0) ; (Z, \varnothing, \forall k\, \alpha(k) \mapsto \underset{\text{IND}}{Z \cup \alpha(k)}, \varnothing, \alpha(k+1)].$$

Pour illustrer les mécanismes de preuve dans LAPD E nous allons traiter le contre-exemple présenté en Première partie (I.5) lorsqu'on a abordé la question de la non-compacité.

II.4. Exemple

Rappelons qu'il s'agit de prouver la formule :

$$\alpha \equiv Q \quad \Rightarrow \quad \Box \langle S, 1 \rangle (\sim Q)$$

dans la théorie : $T = \{\forall k\ Q \Rightarrow \Box \text{Iter}(A, k+1)\ (\sim Q)$ avec

$$S = \{X_1 = A\,.\,\langle Q \mapsto \text{VIDE} \Box X_1 \}.$$

La cession de LAPD commence par la définition de T :

CRÉER-THÉORIE 'COMPACT';;

VPROP 'P';;

VPG 'A';;

c'est-à-dire en premier lieu par la définition du langage (variables propositionnelles et de programme).

Elle se poursuit par la définition des axiomes, ici un seul :

$$\#\ \text{ADD\_AXIOME 'AX 1' « } \forall k.\, Q \Rightarrow \Box\, \text{Iter}(A, k+1)(\tilde{Q}) \text{ »;;}$$

La théorie doit être figée; en particulier l'ensemble de ses axiomes ne pourra plus être augmenté :

$$\#\ \text{SAUVER\_THÉORIE 'COMPACT';;}$$

Enfin on définit le système S à une seule action :

$$\#\ \text{SYSTÈME ('}S\text{', [1, « } A.\ \langle Q \mapsto \text{VIDE} \diamond 1 \text{ »}\}.$$

On peut établir le but à prouver : c'est une formule à champs théorie et hypothèses nuls :

$$\#\ \text{let but} = (\varnothing, \varnothing, \text{« } Q \Rightarrow \Box \langle S, 1 \rangle (\sim Q) \text{ »})$$

Le champ théorie est nul car lors du travail dans une théorie T, tous les axiomes de T font implicitement partie du champ théorie de tout théorème.

Puisqu'apparaît un schéma de programme dans la formule on commence par appliquer une tactique d'induction sur les schémas de programme; elle enchaîne les 2 tactiques élémentaires associées aux règles :

$$\frac{Z \mapsto \forall k\, \alpha \Rightarrow \Box\, \text{Approx}(S, i, k)\, \beta}{Z \mapsto \alpha \Rightarrow \Box\, (S, i)\, \beta}$$

et

$$\frac{Z \mapsto \alpha(0) \qquad Z \cup \alpha(k) \mapsto \alpha(k+1)}{Z \mapsto \forall k\, \alpha(k)}$$

#let step 1 = IND_SCHEMA_TAC « S » but;;

step 1 = [$\varnothing$, $\varnothing$, « $Q \Rightarrow \Box$ Approx(S, i, 0) ($\tilde{Q}$) »;

« $Q \Rightarrow \Box$ Approx (S, i, k) ($\tilde{Q}$) », $\varnothing$, « $Q \Rightarrow \Box$ Approx (S, i, $k+1$) ($\tilde{Q}$) »],
prINDS.

Remarque : C'est l'étude du pattern reconnu dans le but (ici la présence d'un schéma $\langle S, i \rangle$) qui fait élire une tactique. On imagine donc comment

en automatisant la reconnaissance de patterns, on peut automatiser l'application de tactiques. La notion de tactique au sens LCF ouvre donc la voie au raisonnement type « Système expert ».

Notons step 0 et step k les 2 nouveaux sous-buts et remarquons que step k a maintenant un champ théorie non nul : c'est l'hypothèse d'induction.

Résolution de step $0 = [\varnothing, \varnothing, \text{« } Q \Rightarrow \Box \text{ Approx}(S, i, 0)\ (\tilde{Q}) \text{»}]$. Pour résoudre step 0 (étape 0 de l'induction), on utilise une tactique de réécriture :

```
# let step01 = CONVTAC STAND_CONV step0;
step01 = [∅, ∅, VRAI], p0.
```

CONVTAC est une tactique qui permet de simplifier un but en appliquant des règles de réécriture. L'ensemble des règles de réécriture utilisées est spécifié par le premier argument (ici STAND_CONV).

CONVTAC permet de réduire step 0 à « VRAI », les réécritures générant le théorème :

$$(Q \Rightarrow \Box \text{Approx}(S, 1, 0)(\sim Q)) \Leftrightarrow \text{VRAI}$$

en utilisant des équivalences qui sont des axiomes propositionnels ou algorithmiques.

La preuve de step 0 se trouve donc achevée puisque le nouveau but est VRAI.

La notion de réécriture existe dans LCF et porte le nom de conversion. Elle permet la simplification de termes et formules. Dans LAPD, les mêmes principes ont été repris : les conversions permettent de simplifier des termes (expressions d'entier et programmes) et des formules, et sont principalement utilisées pour transformer des buts à prouver.

Détaillons le principe des conversions car nous allons l'utiliser pour résoudre step 1.

II.5. Conversions

Les règles de réécriture qui sont utilisées par CONVTAC sont des théorèmes d'équivalence de formules, orientés de gauche à droite. Ces théorèmes expriment :

- des simplifications standard du calcul propositionnel
- de la réécriture de programmes, d'entiers (cas de STAND_CONV)

mais ce sont aussi des axiomes de la théorie en cours ou des théorèmes algorithmiques prouvés par l'utilisateur.

Classiquement, à partir d'un théorème « $\alpha \Leftrightarrow \beta$ » orienté $\mapsto$, la réécriture fonctionne par pattern matching : dans une formule

$$F(\alpha')$$

on reconnaît la structure de alpha, et on utilise le théorème instancié :

$$\alpha' \Leftrightarrow \beta'$$

pour générer le théorème

$$\vdash F(\alpha') \Leftrightarrow F(\beta')$$

ce qui donne la réécriture :

$$F(\alpha') \rightarrow F(\beta')$$

Une tactique de conversion utilisant le théorème « $\alpha \Leftrightarrow \beta$ » transformerait donc un but $F(\alpha')$ en $F(\beta')$. La preuve associée utilise « $\vdash F(\alpha') \Leftrightarrow F(\beta')$ » pour prouver $F(\alpha')$.

Les conversions prédéfinies dans LAPD sont établies à partir du système de déduction, les seules que peut rajouter l'utilisateur sont définies à partir de théorèmes algorithmiques prouvés. La validité des opérations de conversion est donc garantie. Celles-ci permettent de « court-circuiter » l'application des règles qui peut être très laborieuse : il s'agit de manipulation de termes plutôt que de démarche de déduction.

Résolution de step k

$$\text{step } k \equiv [Q \Rightarrow \Box \text{ Approx } (S, 1, k)\ (\sim Q) \text{ « , } \varnothing, \text{ » } Q \Rightarrow \Box \text{Approx } (S, 1, k+1)\ (\sim Q)].$$

Notons $\text{step } k = [\alpha(k), \varnothing, \alpha(k+1)]$.

On suppose qu'on a démontré le théorème :

$$\text{TH} \equiv \varnothing \mapsto \forall k \Box \text{Approx } (S, i, k+1)\ (\tilde{Q}) \Leftrightarrow \Box \text{ Approx } (S, i, k)\ (\tilde{Q}) \wedge \Box \text{Iter } (A, k+1)\ (\tilde{Q})$$

soit avec la même notation :

$$\text{TH} = \varnothing \mapsto \forall k\ \alpha(k+1) \Leftrightarrow \alpha(k) \wedge \text{Iter}(A, k+1)\ (\tilde{Q}).$$

Nous allons utiliser cette équivalence comme règle de réécriture pour traiter step k. Remarquons que ce théorème n'est pas généré automatiquement par

une tactique de LAPD : la mise en évidence des lemmes utiles à la démonstration reste à la charge de l'utilisateur.

On passe donc TH en argument de CONVTAC :

$$\begin{aligned}
&\#\ \text{let step}\,k\ 1 = \text{CONVTAC TH step}\,k\,;;\\
&\text{step}\,k\ 1 = [\alpha(k),\ \varnothing,\ \alpha(k);\\
&\qquad\qquad \alpha(k),\ \varnothing,\ Q \Rightarrow \Box\,\text{Iter}(A,\ k+1)(\sim Q)],\\
&\qquad\qquad pk
\end{aligned}$$

TH permet de convertir alpha $(k+1)$ et fait apparaître 2 buts qui se résolvent par application d'une règle et d'un axiome structurels sur les théories :

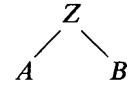
TH 1 $$\{f\} \mapsto f$$

ADD $$\frac{Z \mapsto \text{alpha}}{Z \cup \{\text{beta}\} \mapsto \text{alpha}}.$$

Il ne reste plus qu'à prouver *tep k* par *pk* à partir de ces axiomes, puis *but* à partir de step 0 et step k par prINDS qui appliquera les règles IND et SCHEM.

II. 6. Gestion des théories

Les théories algorithmiques sont organisées de façon hiérarchique : une théorie A est fille d'une théorie Z si elle est définie dans le contexte de Z.



A et B héritent des axiomes de Z; ceci correspond à l'idée de construction de structures de données à partir de structures déjà définies (en particulier pour implémenter une structure de donnée à l'aide d'une autre). Z peut avoir plusieurs théories filles ou descendantes. Inversement on peut affecter plusieurs théories mères à une nouvelle théorie, et cette dernière hérite alors de tous les axiomes.

Le problème de gestion des théories : sauvegarde, restitution, archivage des théorèmes prouvés, nous a amenés à utiliser la gestion faite par LCF. Pour cela il nous faut opérer un codage des théories algorithmiques en théories de LCF, et c'est LCF qui se charge de leur gestion. Ce codage consiste grossièrement en une transcription des formules algorithmiques en termes du langage

objet de LCF. Ceci n'est ni génant ni restrictif car on ne fait aucun travail de preuve dans ces théories codées.

Le codage des formules algorithmiques en termes de LCF nous permet de plus de récupérer les mécanismes de conversion de LCF et toutes les fonctions associées (compositions par exemple) : les conversions des formules algorithmiques se font via un codage en PPLAMBDA, les conversions de LCF, puis un « décodage » du résultat de la conversion LCF en une formule algorithmique.

II.7. Conclusion

Le système présenté ci-dessus n'est qu'une maquette, relativement lourd à l'emploi et proposant des fonctionnalités de base pour la construction des preuves ; cependant, il peut être facilement étoffé en ce qui concerne l'ensemble des tactiques et conversions prédéfinies, avec en particulier des stratégies caractéritiques des démonstrations de la logique algorithmique (comme l'est déjà la tactique IND_SCHEMA_TAC). Ses performances peuvent être améliotées en reprenant la partie définition du langage de LAPD par des types de ML, pour utiliser les nouvelles notions de ML, notamment les facilités de définition de grammaire.

LAPD doit être considéré comme une première étape propositionnelle vers un système avec variables, permettant ainsi de traiter des programmes écrits dans des langages plus courants.

ANNEXE

THÉORIE ALGORITHMIQUE PROPOSITIONNELLE DE L'ARITHMÉTIQUE

1. Logique algorithmique propositionnelle. Théorie Axar. Résultats de programmabilité

G. Mirkowska définit une théorie de l'arithmétique dans une logique algorithmique propositionnelle déterministe DPAL où les formules ont la syntaxe suivante :

- classe des programmes = programmes While
- connecteurs propositionnels classiques
- les formules $M\alpha$ où M est un programme et α une formule.

La structure sémantique équivaut au triplet (D, I_1, I_2) donné pour LAPD$_s$E. Une formule $M\alpha$ a la sémantique de $\diamond M\alpha$ ou $\sim\Box M \sim\alpha$.

Si $s\in D$, on note : $\mathscr{M}, s\vDash\alpha$ pour « la formule α est vraie en s dans le modèle $\mathscr{M}$ ».

Mirkowska considère un langage noté $L^{\mathscr{X}}$ (où $\mathscr{X}$ est un cardinal $\leqq\aleph_0$) c'est-à-dire un ensemble de variables propositionnelles $V\mathrm{o}=\{Z_i\}_{i\in x}$ et de programme $V\mathrm{p}=\{N_i\}_{i\in\mathscr{X}}\cup\{P_i\}_{i\in\mathscr{X}}$, tous deux indicés par $\mathscr{X}$.

La théorie de l'arithmétique Axar est composée des schémas d'axiomes :

1. $N_i(\sim Z_i)$
2. $Z_i\Rightarrow(P_i\alpha\Leftrightarrow\alpha)$
3. $\alpha\Rightarrow N_iP_i\alpha$
4. $\sim Z_i\Rightarrow(\alpha\Rightarrow P_iN_i\alpha)$
5. (while $(\sim Z_i)$ do P_i) Vrai
6. $P_i\beta_i\Leftrightarrow\beta_i$
7. $N_i\beta_i\Leftrightarrow\beta_i$
8. $N_iN_j\alpha\Leftrightarrow N_jN_i\alpha$
9. $P_iP_j\alpha\Leftrightarrow P_jP_i\alpha$
10. $N_kP_l\alpha\Leftrightarrow P_lN_k\alpha$ si $k\neq l$

où i, j, k, 1 sont des indices de $\mathscr{X}$, α est une formule quelconque, β_i *une formule où n'apparaît pas* Z_i.

$T_{\mathscr{X}}$ notera une théorie (sur $L^{\mathscr{X}}$) dont les axiomes contiennent toutes les instanciations de Axar. Donnons quelques modèles de $T_{\mathscr{X}}$:

— pour $\mathscr{X}=1$: $\mathscr{M}=\langle N, w, 1\rangle$ où $i(N_1)$ et $i(P_1)$ sont les fonctions successeur et prédécesseur et

$$w(S)(Z_1)=V\Leftrightarrow s=0$$

— Modèle standard : $\mathscr{M}=\langle S, w, I\rangle$ où

$$S=\{g\in N^{\mathscr{X}} : \exists m/i>m=>g(i)=0\}$$

est l'ensemble des $\mathscr{X}$-uplets dont un nombre fini seulement de variables sont non nulles

$$I(N_i)(f)=g \text{ où } g(j)=f(j) \text{ si } j\neq i \text{ et } g(i)=f(i)+1;$$
$$I(P_i)(f)=g' \text{ où } g'(j)=f(j) \text{ si } j\neq i,\ g'(i)=\mathrm{pred}(f(i)).$$

Programmabilité des fonctions partielles récursives.

DÉFINITION : Une fonction $f: N^n \to N$ est dite *programmable* dans la théorie $T_{\mathscr{X}}$ si et seulement si il existe un schéma de programme M tel que dans tout modèle $\mathscr{M}$ de Axar et pour tout n-uplet $(i_1, \ldots, i_n)$, pour tout état s, on ait :

(1) Si $f(i_1, \ldots, i_n)$ est défini et si la condition

$$(\text{cond})\ \mathscr{M}, s \models P_1^{i_1} \ldots P_n^{i_n}(Z_1 \wedge \ldots \wedge Z_n) \text{ est vérifiée.}$$

Alors :

$$\mathscr{M}, s \models M(P_{n+1}^{f(i_1, \ldots, i_n)} Z_{n+1}).$$

(2) Si $f(i_1, \ldots, i_n)$ n'est pas définie, et si (cond) est vérifiée.

Alors :

$$\mathscr{M}, s \models \sim M \text{ Vrai.}$$

De cette définition, nous pouvons déduire pour f programmable par M :

$$f(i_1 \ldots i_n) \text{ définie} \Leftrightarrow \text{Axar} \models P_1^{i_1} \ldots P_n^{i_n}(Z_1 \wedge \ldots \wedge Z_n) \Rightarrow M(P_{n+1}^{f(i_1, \ldots, i_n)} Z_{n+1})$$

et

$$f(i_1 \ldots i_n) \text{ non définie} \Leftrightarrow \text{Axar} \models P_1^{i_1} \ldots P_n^{i_n}(Z_1 \wedge \ldots \wedge Z_n) \Rightarrow \sim M \text{ Vrai.}$$

THÉORÈME : *Toute fonction partielle récursive est programmable.*

La démonstration de Mirkowska est *constructive*, et traite cas par cas les fonctions de base.

II. Traduction de Axar en théorie de LAPD$_s$E

Supposons que les variables propositionnelles et de programme sont celles de $L^{\mathscr{X}}$. Soient F et F_E l'ensemble des formules de DPAL et LAPD$_s$E respectivement, $\mathscr{A}$ l'ensemble des schémas de programmes sur $L^{\mathscr{X}}$ Si $(\mathscr{M}, \varphi)$ est un modèle de LAPD$_s$E, on note $\text{Val}_{\mathscr{A}}(\mathscr{M}, \varphi, M)$ et $\text{Val}_E(\mathscr{M}, \varphi, \alpha)$ les fonctions sémantiques de M et α dans $(\mathscr{M}, \varphi)$ ($\text{Val}_{\mathscr{A}}(\mathscr{M}, \varphi, M)(s)$ est la valeur calculée par $\mathscr{M}$ à partir de $s \in D$, $\text{Val}_E(\mathscr{M}, \varphi, \alpha)(s)$ est la valeur de α en s).

On définit une fonction $\tau: F \to F_E$ où

$\tau(P) = P$ pour $P \in V0$

$\tau(\text{Vrai}) = \text{VRAI}$

$\tau(\text{Faux}) = \text{FAUX}$

$\tau(\sim \alpha) = \sim \tau(\alpha)$

$\tau(\alpha_1) \wedge (\alpha_2) = \tau(\alpha_1) \wedge \tau(\alpha_2)$

$\tau(\alpha_1, \vee \alpha_2) = \tau(\alpha_1) \mathrm{V} \tau(\alpha_2)$

$\tau(\alpha_1 \Rightarrow \alpha_2) = \tau(\alpha_1) \rightarrow \tau(\alpha_2)$

$\tau(M\alpha) = \Diamond M' \tau(\alpha) = \sim \Box M' \sim \tau(\alpha)$

où M' est le schéma régulier dont l'arbre associé est celui du schéma de programme structuré construit directement à partir de M. Puisque les constructeurs de programme de DPAL sont sémantiquement équivalents à ceux des programmes structurés, M' et M ont la même valeur sémantique (ils représentent la même fonction sur $D \equiv S$). M', schéma régulier donc schéma simple est bien un élément de $\mathscr{A}$.

On a alors, pour toute interprétation m de DPAL et (m, φ) de $\mathrm{LAPD_s E}$: $M_m(s) = \mathrm{Val}_{\mathscr{A}}(m, \varphi, M')(s)$, pour tout s et tout M et par conséquent $\alpha_m, (s) = \mathrm{Val}_E(m, \varphi, \tau(\alpha))(s)$ *quelque soit* φ ceci d'après la définition de τ.

Pour traduire Axar en théorie de $\mathrm{LAPD_s E}$ on opère auparavant la transformation suivante : on remplace l'axiome 6 par les 2 axiomes :

$$\begin{cases} P_i Z_i \Leftrightarrow Z_j \\ N_i Z_j \Leftrightarrow Z_j \end{cases} \quad \text{pour} \quad i \neq j$$

On montre que la théorie Axar' ainsi constituée et Axar ont les mêmes modèles.

On suppose $\mathscr{X}$ *fini.*

On pose donc $V0 = \{Zi\}_{i \in \mathscr{X}}$, $Vp = \{Ni\}_{i \in \mathscr{X}} \cup \{Pi\}_{i \in \mathscr{X}}$.

TN est constituée des schémas de formule suivants :

TN1	$f \rightarrow \Diamond N_i(\sim Z_i)$
TN2	$f \rightarrow (Z_i \rightarrow \Diamond P_i f \leftrightarrow f))$
TN3	$f \rightarrow f \rightarrow \Diamond N_i P_i f$
TN4	$f \rightarrow \sim Z_i \rightarrow (f \rightarrow \Diamond P_i N_i f)$
TN5	$f \rightarrow \quad \langle S_i, 1 \rangle V$
où $S_i =$	$\{X_1 = \langle \sim Z_i \rightarrow P_i . X_1 \Diamond \mathrm{VIDE} \rangle\}$
$\mathrm{TN6_1}$	$f \rightarrow \Diamond P_i Z_j \leftrightarrow Z_j\}$
$\mathrm{TN6_2}$	$f \rightarrow \Diamond N_i Z_j \leftrightarrow Z_j\}^{i \neq j}$
$\mathrm{TN7_1}$	$f \rightarrow \Diamond N_i N_j f \leftrightarrow \Diamond N_j N_i f$
$\mathrm{TN7_2}$	$f \rightarrow \Diamond P_j P_i f \rightarrow \Diamond P_j P_i f$
TN8	$f \rightarrow \Diamond N_k P_l f \leftrightarrow \Diamond P_l N_k f (k \neq l)$

TN est finie et contient les transformées par τ de tous les axiomes ou instances d'axiome de Axar' (on instancie Axar' par des formules dont l'image par τ est dans fsq).

PROPOSITION TN1 : *Soit* $(\mathcal{M}, \varphi)$ *une interprétation de* $LAPD_s E$;

$$\mathcal{M}, \varphi \models \mathrm{TN} \Rightarrow \mathcal{M} \models \mathrm{Axar}'.$$

Démonstration : Soit α une instance d'axiome ou un axiome de Axar'

$$\mathcal{M}, \varphi \models \mathrm{TN} \Rightarrow \mathcal{M}, \varphi \models \tau(\alpha) \quad (cf. \text{ remarque précédente})$$

et

$$\mathrm{Val}_E(\mathcal{M}, \varphi, \tau(\alpha)) = \alpha_m$$

donc

$$\mathcal{M}, \varphi \models \mathrm{TN} \Rightarrow \mathcal{M} \models \alpha$$

d'où $\mathcal{M}, \varphi \models \mathrm{TN} \Rightarrow \mathcal{M} \models \mathrm{Axar}'$.

PROPOSITION TN-2 : *Pour toute fonction partielle récursive* $f: N^m \to N$, *il existe une valeur de* $\mathcal{X}$ *telle que, pour tout m-uptlet* $(n_1 \ldots n_m)$, *il existe une formule* γ *de* $LAPD_s E$ *telle que :* $f(n_1, \ldots, n_m)$ *non définie* $\Leftrightarrow \models \mathrm{TN} \to \gamma$.

La procédure qui a f et $(n_1, \ldots, n_m)$ associé γ est *constructive.*

REMERCIEMENTS

Je tiens à remercier MM. Patrice Enjalbert, Professeur à l'Université de Caen, pour sa participation à ses travaux, Guy Cousineau pour ses conseils, Emmanuel Girard ainsi que la société C.I.M.S.A.-S.I.N.T.R.A. pour leur soutien technique.

BIBLIOGRAPHIE

1. J. W. DE BAKKER, *Recursive Programs as Predicates Transformers*, in *Formal Descriptions of Programming Concepts*, E. J. NEUHOLD éd., North Holland, 1978.
2. J. W. DE BAKKER, *Semantics and Termination of Non Deterministic Recursive Programs*, in *Automata, Languages and Programming*, Edimburgh, 1976.
3. B. S. CHLEBUS, *Completness Proofs for Some Logics of Programs*, Z. Math Logik und Grundlagen Math., vol. 28, 1982, p. 49-62.
4. G. COUSINEAU, *Les arbres à feuilles indicées : un cadre algébrique de définition des structures de contrôle*, Thèse d'état, Paris, 1977.
5. G. COUSINEAU, *An Algebraic Definition of Control Structures*, Theoretical computer science, vol. 12, 1980.

6. G. COUSINEAU, *A Programmation en EXEL*, Revue technique THOMSON-CSF, vol. 10, n° 2, 1978 et Vol 11, n° 1, 1979.
7. G. COUSINEAU, *The Algebraic Structure of Flowcharts, 8th MFCS Symposium* 1979, Lecture Notes in Computer Science, n° 74, Springer Verlag.
8. E. ENGELER, *Algorithmic Properties of Structures*, Math. System Theory, 1, 1967.
9. P. ENJALBERT, *Contribution à la logique algorithmique. Systèmes de déduction pour arbres et schémas de programmes*, Thèse d'état, Paris, 1981.
10. P. ENJABERT, *Systèmes de déduction pour les arbres et les schémas de programmes*, RAIRO Informatique théorique, vol. 14, n° 3 et vol. 14, n° 4, 1980.
11. P. ENJALBERT, *Preuves de programmes*, Revue technique THOMSON-CSF, vol. 12, n° 3, 1980.
12. D. HAREL, *First Order Dynamic Logic*, Lecture Notes in Computer Science, n° 68, 1979, Springer Verlag.
13. M. GORDON, R. MILNER et C. WADSWORTH, *Edimburgh LCF*, Lecture Notes in Computer Science, n° 78, Springer Verlag.
14. R. MILNER, *Logic for Computable Functions Description of a Machine Implementation*, Stanford Artificial Intelligence Project, Memo AIM-169, Computer Science Department Report CS 288, mai 1972.
15. R. MILNER, *LCF : A Way of Doing Proofs with a Machine.* Department of Computer Science, Univ. of Edimburgh.
16. R. MILNER, *A Methodology for Performing Rigorous Proofs About Programs*, Proc 1st I.B.M. Symposium on Mathematical Foundations of Computer Science, 1976.
17. R. MILNER, *A Theory of Type Polymorphism in Programming*, Journal of computer and system science, n° 17, 1978.
18. R. MILNER, *How ML evolved*, in *Polymorphism*, vol. 1, n° 1, janvier 1983, Bell Cabs., L. CARDELLI et D. MCQUEEN éd.
19. G. MIRKOWSKA, *Propositionnal Algorithmic Theory of Arithmetic*, Communication manuscrite.
20. G. MIRKOWSKA, *Propositionnal Algorithmic Logic*, Lecture Notes in Computer Science, n° 74, 1979, Springer Verlag.
21. G. MIRKOWSKA, *Algorithmic Logic and its Application in the Theory of Programs*, Fundamentale Informaticae, vol. I, n° 1 et vol. 1, n° 2, 1977.
22. L. NOLIN et G. RUGGIU, *A Formalization of EXEL*, Assoc. Comput. Mathematics SIGACT-SIGPLAN, Symposium on the Principles of Programming Languages, Boston, 1973.
23. H. RASIOWA, *Algorithmic Logic*, I.C.S. P.A.S. Reports n° 281, Institute of Computer Science, Academie des Sciences, Varsovie, 1977.
24. A. SALWICKI, *Formalized Algorithmic Languages*, Bull. Acad. Pol. Sci. Ser. Math. Astr. Phys., vol. 18, 1970, p. 227-232.
25. A. SALWICKI, *On Algorithmic Theory of Stacks*, Fundamentae Informaticae, vol. III, n° 1, 1980.
26. A. SALWICKI, *On Algorithmic Logic and its Applications*, Internal Report, Pol. Ac. of Sci., 1978.
27. A. SALWICKI, *An Algorithmic Approach to Set Theory*, Proc. F.C.T. 1977, Lecture Notes in Computer Science, n° 56, 1977, Springer Verlag.
28. F. GARCIA, *Étude et implémentation en ML/LCF d'un système de déduction pour logique algorithmique*, Thèse Docteur-Ingénieur, juin 1985, Université Paris-VII.