MÉMOIRES DE LA S. M. F.

MARC BERGMAN
HENRY KANOUI

Axiomatisation des manipulations symboliques en calcul des prédicats

Mémoires de la S. M. F., tome 49-50 (1977), p. 15-30
<http://www.numdam.org/item?id=MSMF_1977__49-50__15_0>

Calculateurs en Math. (1975 - LIMOGES)
Bull. Soc. math.France
Mémoire 49-50, 1977, p. 15-30

AXIOMATISATION DES MANIPULATIONS SYMBOLIQUES EN CALCUL DES PREDICATS

par Marc BERGMAN et Henry KANOUI *

INTRODUCTION

Les systèmes de manipulations algébriques ou symboliques sont implémentés, pour la plupart d'entre eux, en utilisant des langages de programmation, soit proches de la machine (FORTRAN, PLI), soit basés sur la logique mathématique des fonctions - au sens du λ - calculus (LISP).

On propose ici une approche radicalement différente en considérant la logique du 1er ordre comme langage de programmation de haut niveau plus proche de l'utilisateur que les précédents en ce sens qu'il procède de la pensée humaine, sans référence à des concepts spécifiques de la machine. Cette optique n'a de sens que si l'interprétation est réalisée par un démonstrateur automatique suffisamment puissant pour donner à l'utilisateur les possibilités suivantes :

- un langage pour les programmes et les données ;
- un système de manipulation des données basé sur l'unification ;
- un contrôle suffisant de la stratégie de la démonstration permettant une programmation déterministe ou non;
- pas de distinction entre paramètres d'entrée et de sortie pour une procédure ceci peut avoir un grand intérêt comme on le verra dans la suite.

Ces points constituent des outils de base pour les manipulations formelles et en particulier en intégration symbolique.

On présente ici un système écrit en PROLOG, langage fondé sur le calcul des prédicats et développé au Groupe d'Intelligence Artificielle à Marseille.

Il s'agit d'un système conversationnel de manipulations symboliques sur des expressions de plusieurs variables.

Ce travail fait suite à la réalisation d'un système d'intégration formelle [2], [4] implémenté avec une version antérieure et moins puissante de l'interpréteur de PROLOG.

I. LA LOGIQUE DU 1er ORDRE COMME LANGAGE DE PROGRAMMATION - PROLOG :

Une description précise des notions introduites peut être trouvée dans [10].

I.1- Syntaxe.

La syntaxe du langage PROLOG étant la même que celle du calcul des prédicats sous forme clausale, nous l'utiliserons en précisant les notations :

---

* Ce travail a été subventionné par le contrat DRME n° 73/828

- les variables sont précédées d'une " * " (qui les distingue des constantes)

Ex : * x, *y

- un identificateur représente une variable, une constante, une fonction ou un prédicat. C'est une suite de caractères alpha-numériques ou un seul caractère non alpha-numérique.

Ex : x1, LOG, +, 205

- un littéral est une formule atomique précédée du signe " + " ou du signe " - " en accord avec l'affirmation ou la négation de la formule ;

Ex : + P(*x, *y)

- une clause est une disjonction ($\vee$) de littéraux terminée par " . " ou " ! ". Les variables de la clause sont quantifiées universellement. Les quantificateurs et les disjonctions sont omis. Les clauses se terminant par un point sont à enregistrer. Celles se terminant par un point d'exclamation sont des instructions à exécuter immédiatement.

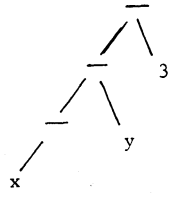
- un programme est une conjonction ($\wedge$) de clauses (où d'une façon analogue, les symboles sont omis).

- des fonctions, unaires ou binaires, peuvent être spécifiées comme opérateurs préfixés ou infixés. La déclaration d'opérateur précise le nom de la fonction, sa priorité, le nombre d'arguments et le sens d'évaluation. Les termes contenant de tels opérateurs sont convertis sous une forme standard préfixée et en sortie la conversion inverse est appliquée.

Ex : -AJOP(" - ", 1, "(0 ↑ x) ↑ x") !

déclare que " - " est de priorité 1, associatif à gauche, binaire, l'opérande gauche est optionnel.

- x - y - 3 est codé comme - (- [- (x), y), 3) ou



I.2.- Définitions de base.

- Une substitution est un ensemble $\tau = \{x_1 \rightarrow t_1 ; \ldots x_n \rightarrow t_n\}$ où les x_i sont des variables, et les t_i des termes (logiques). Une substitution τ appliquée à une expression E remplace toute occurence de x_i dans E par t_i ;

Ex : E = + (x, +(y, z)) $\tau = \{x \rightarrow a, y \rightarrow + (b, c)\}$

donne $\tau(E)$ = + (a, + (+ (b, c), z))

a et + (b, c) sont des instances de x et y respectivement.

- Deux expressions E et F sont unifiables si il existe une substitution qui les rend formellement égales.

- Deux clauses débutant par des formules atomiques unifiables et de signes opposés peuvent engendrer une résolvante en appliquant la règle de résolution

comme règle d'inférence ; si P, Q, R sont des atomes, les clauses

+ P - Q (Q ⇒ P)

ont pour résolvante - Q + R (Q ⇒ R).

- P + R (P ⇒ R)

Nous dirons que P a été " cancelé ".

I.3.- Sémantique.

La Sémantique dénotationnelle de PROLOG est essentiellement celle du calcul des prédicats sous forme clausale.

Un programme en PROLOG peut cependant être interprété d'une façon analogue à une procédure ; une " horn clause " (i.e. qui contient au plus un litteral positif) par exemple

$$+ B - A_1 - A_2 \ldots A_n$$

est interprétée comme une déclaration de procédure dont le nom est B et dont le corps est une suite d'appels de procédures A_i qui vont constituer les étapes de la démonstration de B. Le passage d'une étape à la suivante étant réalisé par la sélection d'une procédure A_i par " matching " avec le nom A d'une procédure déclarée comme

$$+ A - B_1 - B_2 \ldots - B_n$$

La Sémantique opérationnelle (Stratégie) de PROLOG est la suivante : une fonction de sélection opère de la gauche vers la droite sur le corps de la procédure. Si pour un appel A_i , il existe plusieurs clauses C_1 , C_2 , ... C_p (i.e. ces clauses commencent toutes par le même littéral, + A par exemple) et enregistrées dans cet ordre, l'interpréteur choisit d'abord C_1 ; lorsqu'il aura exploré cette voie, il passera à C_2 et ainsi de suite. Il est cependant possible si une voie C_j permet d'évaluer le prédicat A_i d'arrêter l'exploration des voies suivantes, sinon il y a retour en arrière jusqu'à l'appel à A_i (backtracking) et essai sur la clause C_{j+1} . Naturellement, une voie j n'est explorée que si les arguments à l'appel de A_i sont unifiables avec les paramètres du littéral le plus à gauche de C_j . La forme de ceux-ci va donc être utilisée pour le choix de la clause sélectée.

Dans le cadre de manipulations symboliques, l'unification permet alors de résoudre naturellement le problème de la reconnaissance de forme (Pattern-matching)

La programmation est facilitée par l'utilisation de prédicats évaluables : il s'agit en fait de prédicats qui ne sont pas résolus avec les clauses d'entrée mais exécutés par l'interpréteur. Nous en citons quelques uns :

- opérations d'addition, soustraction, produit, division euclidienne sur des entiers positifs.

- le prédicat " / " (shash) qui interdit le backtracking sur tous les littéraux qui le précèdent dans la clause et permet ainsi une programmation déterministe.

- le prédicat UNIV (x, y) : x est un terme et y est une liste dont le 1er élément est le symbole fonctionnel qui domine dans x et dont les arguments constituent le reste de la liste. Celle-ci est construite avec la fonction " • " et terminée par " NIL " (resp. CONS et NIL de LISP). Ce prédicat est évalué si l'un au moins de ses arguments est constant. Il peut être utilisé dans trois voies différentes :

  * si x et y sont constants, il n'est évalué à vrai que si le terme x a une structure de la forme y ;
  * si x seul est constant, on accéde à sa structure y ;
  * si y seul est constant, on crée un terme x.

- le prédicat ANCETRE (x) : L'ancêtre A d'un littéral L est le littéral de la résolvante qui a été unifié avec le premier littéral de la clause contenant L ; tout ancêtre A est un ancêtre de L. Ce prédicat est évalué à vrai si l'un de ses ancêtres est unifiable avec le littéral x.

I.4.- Quelques remarques et deux exemples :

Le passage de la donnée d'un problème à l'exécution des calculs sur ordinateur nécessite le passage par une formulation algorithmique suivie de l'étape de programmation.

Il est clair cependant qu'un algorithme n'est pas nécessairement donné sous forme " mécanisable " et qu'il sera souvent indispensable d'en donner une telle forme équivalente.

Ceci reste dans le domaine purement mathématique car on ne met en jeu que les démarches mentales utilisées par le mathématicien de façon habituelle. En revanche l'aspect technique de la programmation peut être un frein sinon un obstacle à l'exécution sur calculateur.

Le calcul des prédicats considéré comme langage de programmation doit libérer le non-informaticien des contraintes spécifiques à l'utilisation d'un ordinateur. Il s'agit donc de lui permettre de programmer de façon " naturelle ", i. e. de produire rapidement un programme clair et concis dans le cadre conceptuel original de l'algorithme. Dans de nombreux cas le listing du programme suffit à la compréhension de l'algorithme qu'il interprête; PROLOG peut ainsi être considéré comme langage de description d'algorithmes.

Nous allons donner deux exemples simples pour illustrer ceci :

- Le programme de calcul du PGCD de deux nombres, s'écrit en PROLOG :

```
(1) + PGCD (*a, 0, *a) - / .
(2) + PGCD (*a, *b, *d) - RESTE (*a, *b, *r) - PGCD (*b, *r, *d).
```

On suppose ici que les paramètres d'entrée sont les deux premiers arguments du prédicat PGCD, le résultat étant obtenu en 3ème position.

RESTE est un prédicat évaluable qui donne (en * r) le reste de la division euclidienne de *a par * b.

Le calcul du PGCD de 18 et 15 s'effectue en cancelant la clause :

(3) - PGCD (18, 15, *x) - / - SORT (* x) !

SORT est un prédicat toujours évalué à vrai dont l'effet de bord est d'imprimer le terme qu'il a en argument.

L'exécution du programme s'effectue de la façon suivante :

les formules atomiques de (1) et (3) n'étant pas unifiables, la clause (1) n'est pas sélectée.

par contre la sélection de (2) est possible par la substitution : {* a → 18, *b → 15, * d → * x}

Une résolvante est engendrée :

(4) - RESTE (18, 15, * M) - PGCD (15, *M, *x) - / - SORT (* x)

RESTE est évalué et donne pour résultat 3 qui est substitué dans (4) avec suppression du littéral - RESTE et donne :

(4') - PGCD (15, 3, *x) - / - SORT (* x)

(1) n'est pas sélectée (pas d'unification possible sur les arguments)

(2) et (4') donne comme résolvante (substitution : {* a → 15, * b → 3})

(5) - RESTE (15, 3, * M) - PGCD (3, *r, *x) - / - SORT (* x)

L'évaluation de RESTE donne 0 pour résultat et engendre :

(5') - PGCD (3, 0, *x) - / - SORT (* x).

qui avec (1) donne, par la composition des substitutions, {* x, → *a} suivie de {* a → 3} :

- / - SORT (3)

Le prédicat / est alors supprimé, ce qui a pour effet d'interdire une tentative de résolution de (5') sur (2). Le résultat 3 est ensuite imprimé.

On remarquera qu'il suffit de remplacer RESTE par un prédicat calculant le reste de la division euclidienne de deux polynômes pour obtenir le programme de calcul du PGCD de deux polynômes. Il est également possible d'écrire un seul prédicat effectuant la division euclidienne indifféremment sur un couple d'entiers ou sur deux polynômes.

- Un autre exemple naïf : la dérivation.

On considère des expressions réelles d'une variable x.

D désigne l'opérateur unaire de dérivation dans l'algorithme :

$D\ p \rightarrow 0$ si p est constant

$D\ x \rightarrow 1$

$D\ (\alpha + \beta) \rightarrow D\ \alpha + D\ \beta$

$D\ (\alpha - \beta) \rightarrow D\ \alpha - D\ \beta$

$D\ (\alpha \cdot \beta) \rightarrow \beta \cdot D\ \alpha + \alpha \cdot D\ \beta$

$D\ (\alpha / \beta) \rightarrow (\beta \cdot D\ \alpha - \alpha \cdot D\ \beta) / \beta^2$

$D\ \mathrm{Exp}\ \alpha \rightarrow D\ \alpha \cdot \mathrm{Exp}\ \alpha$

$D\ \mathrm{Log}\ \alpha \rightarrow D\ \alpha / \alpha$

Le prédicat DERIVE interprête cet algorithme :

```
+ DERIVE ( *p, 0) - LIBRE ( *p, x)
+ DERIVE (x, 1)
+ DERIVE (*a + *b,  *a1 + *b1) - DERIVE (*a, *a1) - DERIVE (*b, *b1)
+ DERIVE (*a * *b, *b * *a1 + *a * *b1) - DERIVE (*a, *a1) - DERIVE (*b, *b1)
+ DERIVE (*a / *b, (*b * *a1 - *a * *b1) / *b ↑ 2) - DERIVE (*a, *a1)
                                                    - DERIVE (*b, *b1)
+ DERIVE (Exp *a, *a1 * Exp *a) - DERIVE (*a, *a1)
+ DERIVE (Log *a, *a1 / *a) - DERIVE (*a, *a1)
```

LIBRE est évalué à vrai si son premier argument ne contient pas d'occurence du second.

II.- FORMALISATION ET INTERPRETATION D'UN SYSTEME DE MANIPULATIONS SYMBOLIQUES

Ce chapitre donne quelques définitions et des remarques générales qui introduisent la présentation du système SYCOPHANTE décrit dans les chapitres suivants.

II.1.- Sémantique des expressions, manipulations :

On définit un système de manipulations symboliques par la donnée de :

- L'ensemble E des expressions reconnues par le système ;
- une " fonction sémantique " [7] idempotente, ou fonction de normalisation $\mu : E \rightarrow C \subset E$;
- un ensemble $\mathcal{M}$ de relations n-aires sur E.

C est l'ensemble des points fixes de μ, ou ensemble des expressions constantes (car $\forall\ x \in C,\ \mu(x) = x$).

$\mathcal{M}$ est l'ensemble des manipulations que le système est capable de réaliser sur des expressions de E.

Dans $\mathcal{M}$ on considère le sous-ensemble $\mathcal{C}$ des relations de communications (ou commandes) avec le milieu externe : une commande est une relation n-aire incluse dans $E^p \times C^q$ avec $p + q = n$, où

- p est le nombre d'arguments d'entrée,
- q est le nombre d'arguments de sortie.

Les relations éléments de $\mathcal{M} - \mathcal{C}$ sont des sous-ensembles de C^n, i. e. n'admettent comme arguments que les expressions constantes au sens de μ.

II.2.- Interpretation et implémentation :

A toute manipulation $m \in \mathcal{M}$ $(m \subset E^n)$ interprétée en logique du 1er ordre, correspond un prédicat évalué à vrai pour tout n-uplet d'expressions de E en relation par m.

II.2.1.- Variables de programmation et variables algébriques.

Une variable de programmation (au sens de I.1), est un terme logique qui, à l'unification, est substitué par une de ses instances (qui peut elle-même contenir d'autres variables).

Une variable algébrique peut avoir trois types logiques au sens mathématique : constante, variable ou fonction. Ce type est déterminé par la sémantique du prédicat; il est présupposé ou invoqué à l'appel du prédicat et devient effectif quand tous les termes de la clause sélectée sont " ground " (i. e. : il n'apparait que des constantes au sens de PROLOG) et que le prédicat est évalué à vrai : les variables algébriques sont alors des constantes au sens de PROLOG.

II.2.2.- Opérateurs de structure, opérateurs constructeurs et opérateurs applicatifs [7].

Les opérateurs de structure sont ceux qui ont été déclarés comme indiqué en I.1. A une expression $e \in E$ correspond une forme standard unique (ou forme normale) dont la représentation interne est un arbre :

- si $e \in C$, alors les opérateurs de structure sont interprétés comme constructeurs; aucun calcul n'est effectué par le système $(\mu(e) = e)$.
- si $e \in E - C$ alors $\mu(e)$ $(\in C)$ a une représentation avec des opérateurs constructeurs : ceci implique que les opérateurs de structures ont été interprétés comme des applications sur leurs opérandes et que des calculs ont été effectués.

Par exemple, si l'expression $(a + b) \uparrow 2$ (1) est transformée en $a \uparrow 2 + 2 \cdot a \cdot b + b \uparrow 2$ (2) par le système, alors :

dans (1) + et $\uparrow$ sont applicatifs

dans (2) + et $\uparrow$ sont constructeurs.

Ainsi, aux opérateurs constructeurs : + , − , · , / , $\uparrow$ ... correspondent les opérateurs d'évaluation qui seront interprétés par les prédicats SOMME, DIFFERENCE, MULTIPLIER, DIVISER, DEVELOPPER ... qui effectueront les calculs.

Il n'est pas nécessaire d'associer à tout opérateur constructeur un opérateur applicatif : on disposera d'une fonction de représentation qui transforme le caractère constructeur en applicatif. Ce sera notamment le cas lors des substi-

tutions de variables par des constantes pour des symboles fonctionnels (Log, Exp ...) déclarés comme opérateurs.

Chaque opérateur de structure est interprété comme constructeur ou applicatif suivant la commande.

La représentation d'une expression par les constructeurs est unique : nous l'appellerons forme normale (on réservera le terme " canonique " lorsqu'une structure algébrique particulière est définie).

II.2.3.- Normalisation, simplification, réductions, substitutions.
Politique dans les manipulations :

L'ensemble E des expressions étant défini par la donnée des éléments atomiques et d'une syntaxe, on définira un ordre total sur les expressions par extension de l'ordre lexicographique.

A toute expression de E correspond par la fonction μ sa forme normale unique dans C. La normalisation effectue les simplifications " immédiates " du type
$a + 0 \to a$ $a \cdot 1 \to a$ $a \cdot 0 \to 0$ $a^b \cdot a^c \to a^{b+c}$ $a^0 \to 1$
$\alpha\, a + \beta\, a \to (\alpha + \beta) \cdot a$ ... (où α et $\beta \in \mathbb{N}$); mais une expression, par exemple : $(a + b) \uparrow 2$ ne sera pas développée a priori mais sur demande de l'utilisateur.

Les autres simplifications restent de nature ambiguë sans référence à un contexte (calcul trigonométrique par exemple) ou à une structure algébrique. Il s'agit alors de réductions.

Le système devra donc fournir à l'utilisateur un dispositif de reconnaissance de forme et de substitution suffisamment puissant pour lui permettre soit d'utiliser des relations connues du système, soit d'introduire lui-même les égalités ou identités nécessaires.

La politique générale est que toute manipulation qui n'est pas une commande aura tous ses arguments sous forme normale. En particulier les résultats seront toujours sous cette forme. Cela permettra d'écrire des algorithmes plus performants

III.- DESCRIPTION GENERALE DU SYSTEME SYCOPHANTE

Nous présentons brièvement les principales possibilités du système. Une description complète peut être trouvée dans [3].

Les expressions manipulées par le système sont construites à partir des nombres rationnels, e, π par application des opérations rationnelles (+ , - , $\uparrow$, $\cdot$, /) et des symboles fonctionnels usuels (log, sin, cos, tg, cotg, arcsin, arctg, ...). Les expressions peuvent comporter un nombre quelconque de variables, représentées par des identificateurs.

Les variables utilisées lors d'une session doivent être déclarées (à n'importe quel moment).

L'ordre dans lequel les variables sont déclarées induit un ordre sur la classe des expressions que le système peut traiter. Cet ordre sert à définir la forme normale des expressions (cf. chap. II).

Le système est conversationnel et l'utilisateur peut employer des métavariables pour nommer les expressions qu'il définit. Dans un calcul ultérieur, on peut alors utiliser la métavariable qui référence une expression évitant ainsi de réécrire et de renormaliser celle-ci.

D'une façon générale, la syntaxe d'une commande est de la forme :

< métavariable > = < expression généralisée > .

Les opérations intervenant dans < expression généralisée > sont indiquées par les opérateurs ou symboles fonctionnels associés. Les opérandes peuvent être des métavariables ou faire intervenir des commandes elles-mêmes.

Des commandes permettant les opérations suivantes sont actuellement disponibles

- Développement d'une expression : la commande A = DEV(B) affecte à A le résultat du développepent de l'expression référencée par B.
- Factorisation formelle :
  . A = FACT(B) factorise l'expression B en mettant en évidence le plus grand facteur commun des termes qui constituent B.
  . A = FACT(B, C) recherche si l'expression C peut être formellement divisée par l'expression B. Si oui, le quotient est affecté à A.
- Mise au dénominateur commun : A = DECOM(B).
- Dérivation A = DERIV(B, X ↑ P • Y ↑ Q • Z ↑ R) signifie $A = \frac{\partial^{p+q+r} B}{\partial^p X \, \partial^q Y \, \partial^r Z}$
- Substitution : A = SUBST(B, C, D) substitue toute occurence de l'expression C par D dans B.
- Passage des coordonnées cartésiennes en coordonnées polaires, sphériques ou cylindriques.
- Identités trigonométriques :
  . Transformation de produit en somme
  . " de somme en produit
  . " de SIN/COS en TG
  . " de COS/SIN en 1/TG
  . " de COTG en 1/TG
  . " du type SIN(-U) en -SIN U, $COS(\frac{\pi}{2} - U)$ en SIN U, ...
  . Linéarisation.
  . Développement de SIN(U + V), COS(U + V), ...
  . Passage à l'arc moitié
- Manipulations sur des équations.

Ceci permet de définir et de référencer une relation d'égalité entre deux expressions. La forme générale de ces commandes est :
(< référence >) : < expression généralisée > = < expression généralisée >
où < référence > est un nombre ou une suite de nombres séparés par des points. En outre, une commande peut avoir comme argument la référence d'une équation : la commande agit alors sur les deux membres de l'équation référencée.

- Appel au système d'intégration formelle :

A = INT(B, X) signifie $A = \int B\, dX$.

IV.- LE SYSTEME D'INTEGRATION FORMELLE.

Il se compose de trois grandes étapes qui examinent successivement le problème considéré.

Dans la première étape, si l'intégrand est une somme, la première méthode de recherche s'il peut s'exprimer comme la dérivée d'un produit de facteurs, avant d'appliquer la linéarisation (deuxième méthode).

La troisième méthode de cette étape résoud les problèmes qui se mettent sous la forme $\int c\, u'\, h(u)\, dx$ où

- c est une constante (par rapport à x);
- u est une expression quelconque;
- h est une fonction dont la primitive g est l'une des fonctions de base (identité, log, sin, cos, tg, cotg, arcsin, arctg), ou bien définie par : $g(x) = a^x$ ou $g(x) = x^a$, a ne contenant pas x.

La deuxième étape est composée de 7 méthodes, chacune spécifique d'une forme particulière de l'intégrand (cf. [3]). Le coeur de cette étape est l'intégration des fractions rationnelles qui utilise une procédure de factorisation des polynômes à une variable basée sur l'algorithme modulaire de Berlekamp.

Dans la troisième étape, est implémentée la méthode d'intégration par parties dont une propriété intéressante est la résolution automatique des problèmes tels que $\int x\, e^x \cos x\, dx$ dans lesquels certains des sous buts engendrés en répétant la méthode sont identiques à des buts de niveau plus haut. De plus, lorsque deux sous buts identiques apparaissent dans des branches distinctes de " l'arbre de recherche ", il n'est pas nécessaire de résoudre le problème correspondant deux fois.

Cette méthode est discutée en détail dans [5].

V.- UN EXEMPLE EN INTEGRATION FORMELLE

Nous allons examiner un type de problèmes particulier résolu par la première méthode de la première étape du système d'intégration formelle qui reconnait l'intégrand comme la dérivée d'un produit.

Nous donnons l'algorithme utilisé dans le cas d'un produit de deux facteurs. Naturellement, le programme est valable pour un nombre quelconque de facteurs.

Supposons que l'intégrand ait été unifié avec l'expression a + b qui doit être reconnue sous la forme :

$$f \cdot g' + f' \cdot g$$

x étant la variable d'intégration.

L'algorithme s'énonce ainsi :

(I) Déterminer une nouvelle partition du terme b telle que b soit égal au produit de l'expression t par l'expression h.
S'il n'existe pas de telle partition, il y a échec.

(II) [identifie b = t • h avec f • g'].
Calculer par " intégration immédiate " la primitive de h par rapport à une expression u, i. e. calculer g = $\int$ h du.
Si ce n'est pas possible, aller en (I).

(III) [A ce stade, g', dérivée de g par rapport à x, est égale à h • u'. Puisque g' = h • u' doit être contenu dans b = t • h, u' doit être contenu dans t].
Calculer u', dérivée de u par rapport à x, et essayer de diviser t par u'. Si cela est possible, soit f le quotient, sinon aller en (I).

(IV) [A ce stade, b est de la forme f • g'; a doit être identifié avec f' g].
Soit f' la dérivée de f par rapport à x. Développer le produit f' • g et comparer le résultat avec a. Si les deux expressions sont identiques, retourner f • g comme résultat de l'intégration; sinon aller en (I).

Cet algorithme s'exprime par une seule clause PROLOG qui est :

```
(1)     + I II  (a + b, x, k)
                - PRO (h, t, b)                 pas I
                - DIM (g, x, u, h)              pas II
                - DERIV (u, x, u1)      }
                - FACTEUR (t, u1, f)    }       pas III
                - DERIV (f, x, f1)      }
                - DEVELOPPER (f1 • g, a) }      pas IV
                - PRODUIT (f, g, k)     }
```

Remarques -

1) La procédure I 11 qui implémente l'algorithme a trois arguments : lorsque la procédure est activée, le premier argument est unifié avec l'intégrand, le deuxième avec la variable d'intégration, le troisième argument sera substitué par le résultat de l'intégration (si celui-ci peut être calculé).
   Il faut remarquer que le premier paramètre formel (i. e. l'intégrand) est dominé par " + ", de sorte que la procédure ne peut être activée que si l'intégrand effectif est lui-même une somme : l'appel de procédure est fait par unification, ce qui réalise automatiquement un test conditionnel sur la structure des données.
2) L'algorithme fait appel à la procédure PRO (h, t, b) qui calcule des partitions possibles d'un terme donné.
   A partir du terme d'entrée (donné en troisième argument, b) la procédure délivre deux expressions, substituées par h et t dont le produit est égal à b.
   Naturellement, toutes les partitions ne seront pas déterminées en même temps ! Une nouvelle partition sera délivrée à chaque activation de PRO.
3) DIM prend ici en donnée h et x et donne en résultat g et u tels que h est une fonction élémentaire de l'expression u dont la primitive est connue et $g = \int h\,du$.
   La même procédure DIM est utilisée dans le programme de dérivation, où les entrées sont g et x et les sorties h et u.
   Ceci est un cas où une même procédure peut effectuer un calcul ou son opposé en échangeant seulement les paramètres d'entrée et de sortie.
4) FACTEUR (a, b, c) teste si l'expression donnée en a peut être formellement divisée par l'expression donnée en b, c étant le quotient.
5) La procédure DEVELOPPER (v, w) prend d'ordinaire comme entrée v et affecte à w le résultat du développement de v.
   Dans le cas présent, les paramètres effectifs sont tous deux substitués par des termes sans variables lors de l'appel. Il y aura donc ici une comparaison entre le résultat du développement d'un produit et le résultat présumé.
   Il est à noter qu'il n'est pas nécessaire de développer d'abord le produit, puis de comparer le résultat obtenu avec le deuxième argument.
   Le développement et la comparaison se font simultanément et le calcul s'arrête dès qu'il y a conflit entre la structure de a et celle du résultat intermédiaire.

6) Un échec en I, II, III conduit au pas I de l'algorithme.
En PROLOG, cela signifie que le(s) prédicat(s) correspondant(s) ne peut (peuvent) être supprimé(s); un " backtracking " automatique est alors effectué et le programme tente d'évaluer les littéraux précédents d'une façon différente.
Mais la seule procédure non déterministe est PRO (qui correspond au pas I) de sorte que chaque échec active PRO qui détermine une nouvelle partition avec laquelle le processus se poursuit.

Nous illustrons cette procédure par le calcul de :

$$\int (e^{X^2} + 2\,X^2\,e^{X^2})\,dX$$

qui résoud : -I 11(e^{X^2} + 2 X^2 e^{X^2}, X, k avec la clause (1) par la substitution

$$\{a \to e^{X^2};\ b \to 2\,X^2\,e^{X^2};\ x \to X\}$$

Voici le détail des calculs (le résultat est obtenu dans k).

Nom de la procédure activée	substitution	1ère éval. de PRO	2ème éval. de PRO	3ème éval. de PRO
PRO	h →	X	X^2	e^{X^2}
	t →	$2Xe^{X^2}$	$2e^{X^2}$	$2X^2$
DIM	u →	X	X	X^2
	g →	$X^2/2$	$X^3/3$	e^{X^2}
DERIV	u1 →	1	1	2X
FACTEUR	f →	$2Xe^{X^2}$	$2e^{X^2}$	X
DERIV	f1 →	$2e^{X^2}+4X^2e^{X^2}$	$4Xe^{X^2}$	1
DEVELOPPER	a →	unification impossible		e^{X^2}
PRODUIT	k →			Xe^{X^2}

CONCLUSION

Certains concepts tels que la reconnaissance des formes, la manipulation de structures d'arbres, le non déterminisme, les facilités d'extension des programmes sont des outils de base qui sont intrinsèques au langage de la logique du premier ordre, alors qu'ils doivent être explicitement programmés lorsqu'on utilise des langages de programmation classiques.

Unique parmi les langages de programmation, le calcul des prédicats sous forme clausale a été défini pour la formalisation de la pensée humaine [6] . Ceci permet une programmation très naturelle, où l'on ne se soucie plus de tâches purement

techniques qui ne relèvent que de la machine.

L'utilisateur peut ainsi exprimer son problème en termes proches de sa conception originale, produisant ainsi des programmes plus clairs et plus courts. La formalisation en est plus élégante et rend souvent inutiles les descriptions par algorithmes et organigrammes.

PROLOG est une implémentation pratique de la logique du premier ordre considérée comme langage de programmation. C'est ce qui a permis la réalisation rapide de SYCOPHANTE qui soutient la comparaison avec les systèmes déjà existants.

La conception du " système noyau " qui existe actuellement permet d'envisager des prolongements, en introduisant des concepts fins de l'analyse mathématique par exemple, et à plus long terme donner à l'utilisateur la possibilité d'adapter le système à son problème.

REFERENCES

(1) G. BATTANI et H. MELONI.- "Interpréteur du langage de programmation PROLOG".- Rapport de DEA, G.I.A., U.E.R. de LUMINY, Université d'AIX-MARSEILLE.- (1973).

(2) M. BERGMAN.- "Résolution par la démonstration automatique de quelques problèmes en intégration symbolique sur calculateur".- Thèse de 3ème cycle.- (1973).

(3) M. BERGMAN et H. KANOUI.- "Sycophante, système de calcul formel et d'intégration symbolique sur ordinateur".- Convention DRME n° 73/828, Rapport Final, Octobre 1975.

(4) H. KANOUI.- "Application de la démonstration automatique aux manipulations algébriques et à l'intégration formelle sur ordinateur".- Thèse de 3ème cycle.- (1973).

(5) H. KANOUI.- "Some aspects of symbolic integration via Predicate-Logic programming".- (soumis pour publication).

(6) R. KOWALSKI.- "Logic for problem solving".- MEMO n° 75, Dept of Artificial Intelligence, University of EDINBURG.- (1974)

(7) R. LOOS.- "Toward a formal implementation of Computer Algebra".- Proceedings of EUROSAM 74, SIGSAM BULLETIN V8, N3, (1974)

(8) J. MOSES.- "Symbolic Integration".- Rapport MAC TR-47, PROJECT MAC, MIT,(1967)

(9) J. MOSES.- "Symbolic Integration, the stormy Decade".- Proceedings of the second Symposium on symbolic and algebraic Manipulation, ACM (1971).

(10) Ph. ROUSSEL.- "PROLOG : manuel d'utilisation", G.I.A., U.E.R. de LUMINY, Université d'AIX-MARSEILLE (1975).

Marc BERGMAN et Henry KANOUI
Groupe d'Intelligence Artificielle
U.E.R. de Luminy
13288 Marseille Cédex 2

ANNEXE : QUELQUES EXEMPLES

```
 EXECUTIØN BEGINS...

-HELLØ!



VARIABLES=X.Y.Z,


A=Y+E!X ,
***    A=E!X+Y


B=DEV(A!2) ,
***    B=E!(2.X)+2.E!X.Y+Y!2


C=FACT(A,B) ,
***    C=E!X+Y


Q4=6.A.X!3+16.B!2.X!2.Y+3/2.X!5.Y!2 ,
***    Q4=6.A.X!3+16.B!2.X!2.Y+3.X!5.Y!2/2


Q5=DERIV(Q4,X!2.Y) ,
***    Q5=32.B!2+60.X!3.Y


"RECØNNAISSANCE ET INTEGRATIØN DE U DV + V DU ",


A=E!X!2+2.X!2.E!X!2 ,
***    A=E!X!2+2.E!X!2.X!2


IA=INT(A,X) ,
***    IA=E!X!2.X


B=(CØS X)!5-5.X.(CØS X)!4.SIN X ,
***    B=-5.X.SIN X.(CØS X)!4+(CØS X)!5


IB=INT(B,X) ,
***    IB=X.(CØS X)!5


"INTEGRATIØN PAR PARTIES" ,


D=LØG(X.Y) ,
***    D=LØG(X.Y)


IDX=INT(D,X) ,
***    IDX=-X+X.LØG(X.Y)
```

```
E=LØG(SIN X+1)+X!2-4 ,
***    E=-4+X!2+LØG(1+SIN X)

F=SUBST(E,X,SIN Y) ,
***    F=-4+LØG(SIN SIN Y)+(SIN Y)!2

G=SUBST(F,Y,PI) ,
***    G=-4

G=SUBST(F,SIN Y,X)-E ,
***    G=0

F=2+3.CØS X+3.CØS Y+CØTG Z ,
***    F=2+3.CØS X+3.CØS Y+CØTG Z

G=TRIGSP(F,C) ,
***    G=2+6.(SIN(1/2.(X-Y)).CØS(1/2.(X+Y)))+CØTG Z

A=TRIGØ(2+LØG R+6.(CØS X)!2+CØTG Z,VII:1.2),
***    A=2+3.(1-CØS(2.X))+LØG R+CØTG Z

STØP,
```