STRUCTURES DE CONTROLE ET PUISSANCES D'EXPRESSION

C. HENRY (1)

Résumé : La structure de contrôle d'un langage peut-être définie comme l'ensemble des éléments qui permettent d'enchaîner les opérations élémentaires du langage (affectations,entrées-sorties ...). A cette structure de contrôle peut être associée la puissance du langage, c'est-à-dire la capacité du langage à décrire des suites d'opérations de telle ou telle complexité.

En utilisant le formalisme des arbres à feuilles indicées comme modèle des programmes, on établit les résultats suivants :

- la classe des langages de branches des arbres solutions de programmes simples propres (i.e. avec des procédures à variables globales et sans itération) est exactement la classe des langages simples.
- la classe des arbres solutions de programmes hybrides (i.e. écrits avec des procédures à variables globales et des itérations) est la classe des arbres algébriques.
- la classe des arbres solutions des programmes hybrides propres (i.e. sans sortie multiple d'itérations) est égale à celle des arbres solutions des programmes simples propres.

(1) : LCR, THOMSON-CSF, Domaine de Corbeville, 91401 ORSAY.

INTRODUCTION

La structure de contrôle d'un langage peut être définie comme l'ensemble des éléments qui permettent d'enchaîner les opérations élémentaires du langage (affectations, entrées-sorties ...). A cette structure de contrôle sont associées une "puissance de calcul" et une "puissance d'expression" du langage.

En fait, la notion de puissance de calcul est de peu d'intérêt en ce qui concerne la programmation: si nous prenons par exemple pour domaine de calcul, le domaine des entiers, tout langage, pour peu qu'il possède une forme quelconque d'itération, permet de calculer toute fonction récursive.

Par contre, la puissance d'expression, c'est-à-dire la capacité du langage à décrire des suites d'opérations de telle ou telle complexité [AMP, AR, BJ, CH, FK, GL ...] est à notre avis une notion pertinente, car d'elle dépend la possibilité de programmer de façon naturelle le plus grand nombre possible d'algorithmes. Or, il se trouve que la puissance d'expression peut être définie et mesurée de façon très précise grâce à la notion d'arbre syntaxique [CN]. Le but de cet article est de caractériser et de comparer ainsi les puissances d'expression de différents langages de programmation.

Les structures de contrôle que nous étudions comprennent la concaténation, la sélection, des itérations et des procédures sans variables locales (i.e. à variables globales: procédures VG). Aux programmes ainsi écrits correspondent des systèmes d'équations algébriques dont la solution est un arbre infini définissant l'ensemble des exécutions possibles du programme, et que l'on appelle arbre syntaxique [CS1].

Les résultats énoncés dans cet article sont les suivants :

- la classe des langages de branches des arbres solutions de programmes simples propres (i.e. avec des procédures VG mais sans itération) est exactement la classe des langages simples.
- la classe des arbres solutions de programmes hybrides (i.e. écrits avec des procédures VG et des itérations) est la classe des arbres algébriques.
- la classe des arbres solutions des programmes hybrides propres (i.e. dont les sorties des procédures VG se font en séquence des appels récursifs) est égale à celle des arbres solutions des programmes simples propres (i.e. sans itération).

Ainsi le fait de programmer avec des procédures VG sans itération permet une puissance d'expression supérieure à la programmation itérative (car la classe des langages de branches des arbres solutions de programmes itératifs est la classe des rationnels [CS1]).

D'autre part, l'utilisation d'itération du type REPEAT-EXIT où les sorties d'ité-

rations peuvent être des "sorties multiples" (sortie de plusieurs itérations emboîtées) apporte un gain de puissance d'expression aux programmes écrits avec des procédures VG.

Par contre, l'apport d'itérations du type "WHILE DO", "REPEAT UNTIL", ou "FOR STEP UNTIL DO" et "LOOP EXIT-identifiant END LOOP" de ADA [*] n'engendre aucune puissance d'expression supplémentaire aux programmes écrits avec des procédures VG. Ainsi PASCAL [JW] et ADA ont même puissance d'expression (avec des procédures VG) et EXEL [NR] a une puissance d'expression supérieure à ces deux langages.

1 - STRUCTURE DE CONTRÔLE

Nous nous plaçons dans le cadre d'une structure de contrôle de type général, contenant des procédures à variables globales. On y retrouve, enchainant les instructions de base (affectations, entrées-sorties ...), les opérations suivantes :

- concaténation (que nous noterons ";")

- la sélection: "IF P THEN A ELSE B FI". On admet la possibilité d'avoir une sélection-aiguillage du type: "IF P THEN1 A1 THEN2 A2 ... THENn An ELSE B FI" qui peut être interprétée en : si P=1 alors effectuer A1, sinon si P=2 alors effectuer A2, ... si P=n alors effectuer An, sinon effectuer B.

- l'itération : il existe plusieurs types d'itérations :

"WHILE DO", "REPEAT UNTIL", "FOR STEP UNTIL DO", et la forme en "REPEAT-EXIT".

Cette dernière est la plus générale; en effet :

"WHILE P DO A OD" se traduit en "REPEAT IF P THEN A ELSE EXIT FI ENDR(EPEAT)",

"REPEAT A UNTIL P" se traduit en "REPEAT A ; IF P THEN EXIT ELSE FI ENDR",

et "FOR I=N STEP Q UNTIL P DO A OD" se traduit en

"I=N ; REPEAT IF I=P THEN EXIT ELSE A ; I=I+Q FI ENDR".

Par contre "REPEAT IF P THEN EXIT ELSE IF Q THEN EXIT ELSE A FI FI ENDR" ne peut se traduire en aucune des trois autres formes d'itérations (sauf avec adjonction de booléens ou de marqueurs). C'est donc cette forme d'itération que nous utiliserons. Nous admettrons également que nous pouvons disposer d'EXITn qui signifient : sortir de n itérations emboîtées.

On utilisera également des procédures à variables globales (VG), que nous noterons: $\alpha\text{PROC} = C \neq \emptyset$ où C est le corps de la procédure, défini à partir de cette structure de contrôle.

On définira un programme comme étant un ensemble de procédures VG, l'appel du programme se faisant par l'appel d'une procédure particulière.

2 - NOTIONS D'ARBRE DE PROGRAMME, DE SOLUTION, D'EQUIVALENCE

A toute procédure VG, on associe une équation arborescente de la forme :

$$\alpha\text{PROC} = \text{TREE}(C)$$

où $TREE(C)$ est défini récursivement, à partir du corps C de la procédure $\alpha PROC$ par:

- si $C = EXITn$ alors $TREE(C) = n$.
- si $C = I$, où I est une instruction élémentaire, alors $TREE(C) = I$.
- si $C = IF P THEN1 C1 THEN2 C2 \dots THENn Cn ELSE Cn+1 FI$

alors $TREE(C) =$

- si $C = C1 ; C2$ alors $TREE(C) =$

```

graph TD
    Semicolon[;] --- TREE_C1[TREE(C1)]
    Semicolon --- TREE_C2[TREE(C2)]
  
```
- si $C = \alpha PROC$ alors $TREE(C) = \alpha PROC$
- si $C = REPEAT C1 ENDR$ alors $TREE(C) = *$

```

graph TD
    Asterisk[*] --- TREE_C1[TREE(C1)]
  
```

(On posera $F = \{f / f \text{ est un symbole d'instruction élémentaire ou de prédicat}\}$)

D'autre part pour des raisons d'homogénéité d'écriture et pour des raisons théoriques, on termine toutes les branches par un symbole d'appel de procédure VG ou par un symbole correspondant à un $EXIT$ (fût-ce $EXIT0$ qui correspond à "sortir de 0 itération emboîtée", c'est-à-dire ne rien faire).

Notons également que du fait de l'ambiguïté de la notation linéaire, " $I2 ; I3$ " peut se transcrire en $I2$ si l'on considère qu'il s'agit de la composition de $I2$

et $I3$, ou se transcrire en

si l'on considère qu'il s'agit de la concaténation de $I2$ et $I3$. Il est clair que ces deux formes sont équivalentes et nous passerons de l'une à l'autre sans problème.

Exemple : à la procédure

$\alpha PROC = REPEAT IF P THEN I1 ; EXIT1 ELSE I2 ; I3 ; \alpha PROC FI ; I4 ENDR \neq$
correspond l'équation :

$\alpha PROC =$ qui peut également s'écrire: $\alpha PROC =$

Deux programmes (systèmes de procédures VG) sont syntaxiquement équivalents si leurs arbres solutions associés sont égaux. Intuitivement, cela signifie que l'histoire des calculs effectués dans les deux systèmes est la même.

3 - CLASSES DE PROGRAMMES

3 - 1. Les Rationnels (Rat (F))

Un programme est rationnel ssi il ne contient pas d'occurrence d'appel récursif de procédures VG, ce qu'on peut encore exprimer par "un programme est rationnel ssi il est itératif" .

Exemple:

```
α1=REPEAT IF P1 THEN1 I1 ; I2 THEN2 I3 ; EXIT1 ELSE
  REPEAT IF P2 THEN1 I4 ; EXIT2 THEN2 I5 ; EXIT1 ELSE I6 ; I7 FI ENDR ; I8 FI ENDR≠
```

3 - 2. Les Réguliers (Reg(F))

Un programme est régulier ssi - il est écrit sans itération

- les occurrences d'appel de procédures VG sont en position terminale (intuitivement, sont telles que leur exécution termine l'exécution des termes dans lesquels elles sont appelées).

Exemple:

```
α1 = IF P1 THEN I1 ; α1 ELSE IF P2 THEN I2 ; α2 ELSE I3 FI FI ≠
α2 = IF P3 THEN1 I4 ; α1 THEN2 I5 ; α2 ELSE I6 FI ≠
```

3 - 3. Les Simples (Sim(F))

Un programme est simple ssi il est écrit sans itération.

Exemple:

```
α1 = IF P1 THEN1 I1 ; α1; α1 THEN2 I2 ; α1 ; I3 ELSE I4 FI ≠
```

3 - 4. Remarque:

La présence d'un EXIT, autre que EXIT0, dans un programme régulier ou simple peut être interprétée comme présence d'un "stop" autre que la sortie en séquence.

3 - 5. Les Hybrides (Hyb(F))

Un programme hybride ne présente aucune restriction d'écriture (hybride au sens itératif et récursif).

Exemple:

```
α1 = REPEAT IF P1 THEN I1 ; α1 ELSE I2 ; EXIT2 FI ENDR ≠
```

3 - 6. Les programmes Propres

Intuitivement, un programme est propre ssi toutes les procédures qui le constituent n'ont que des sorties en séquence. Plus formellement, un programme est propre ssi les arbres solutions associés à chacune des procédures du programme n'ont que "0"

pour feuilles (et en ont au moins une).

On peut ainsi définir les programmes rationnels propres ($\text{PRat}(F)$), réguliers propres ($\text{PReg}(F)$), simples propres ($\text{PSim}(F)$) et hybrides propres ($\text{PHyb}(F)$).

3 - 7. Propriété :

Le fait d'être propre, pour un système, est décidable.

L'algorithme qui permet de connaître l'ensemble des valeurs de sortie de chacune des équations du système repose sur un système de marquage des occurrences pouvant être atteintes à partir du début de chacune des équations. Son coût est linéaire en $n \times k$ où n est le nombre de procédures du système et k la valeur du plus grand EXIT du système. Pour vérifier si un système d'équations est propre, il suffit de vérifier que l'ensemble des valeurs de sortie de chacune des équations est bien réduit à $\{0\}$.

Pour fixer le sens de cette classification, on va y rattacher quelques langages de programmation. On étudie sur ces langages la restriction de leur structure de contrôle qui opère uniquement sur des variables globales (pas de procédures ni de blocs à variables locales), et sans GOTO (il est bien connu que ceux-ci n'apportent aucune puissance d'expression supplémentaire et qu'on peut donc s'en passer).

Ainsi PASCAL [JW] possède les trois formes d'itérations FOR STEP UNTIL DO, WHILE DO et REPEAT UNTIL qui peuvent se traduire en REPEAT EXIT1 (cf 1). L'emploi de procédures sans variables locales classe donc ce langage dans les hybrides propres.

Le LOOP EXIT ENDOLOOP d'ADA [*] recouvre strictement en puissance d'expression, les deux formes particulières FOR IN LOOP ENDOLOOP et WHILE LOOP ENDOLOOP. La présence d'"EXIT identifier" peut être "assimilée" à des EXITn (à ceci près que leur gestion est différente). Cependant ce langage ne permet pas de renvoi en fin d'itérations extérieures à la procédure (c'est-à-dire englobant l'appel de la procédure) dans laquelle se trouvent les EXIT. Il s'agit donc d'un langage hybride propre, mais cette fois avec des EXITn.

EXEL [NR], avec ses "actions", (procédures à variables globales) et sa forme d'itération REPEAT-EXITi, appartient à la classe des hybrides (non propres). En particulier on peut définir des actions (procédures VG) dont on peut sortir au moyen d'EXITn, en faisant également sortir d'itérations emboîtées contenant l'appel. (cf exemple 3.5: EXIT2). EXEL est d'ailleurs à notre connaissance le seul langage possédant une telle puissance d'expression.

3 - 5.1. Quelques rappels

La profondeur d'itération d'une occurrence E' d'une sous-formule dans une formule

E , notée $\delta(E', E)$, est le nombre d'itérations emboîtées de E contenant E' .

A toute occurrence s d'EXITi dans E , on associe la valeur $\tau(s, E) = i - \delta(s, E)$

Intuitivement, les signes terminaux d'une expression E sont les EXITi qui font sortir (en sequence ou non) de E .

$k \in \mathbb{N}$ est une valeur de sortie d'un terme T (respectivement d'une équation α_i) si k apparaît comme feuille de l'arbre solution associé au terme T (respectivement à α_i).

4 - COMPARAISON DES CLASSES DE PROGRAMMES

On peut définir deux sortes de comparaisons entre classes de programmes :

- d'une part, l'inclusion directe, à savoir: tout programme de la première appartient à la seconde.
- d'autre part, l'inclusion en puissance d'expression : (notée $\subset_{pe}$): à tout programme de la première on peut associer un programme de la seconde qui lui soit équivalent.

Pour ce qui est du premier type d'inclusion, il est clair que l'on a le schéma suivant :

Notre étude va porter sur le 2ème type de comparaison.

4 - 1. Propriété

A tout organigramme, on peut associer un système de procédures VG régulier propre qui définisse le même calcul.

On peut transcrire et ce, facilement, tout organigramme en un système de procédures VG sans utiliser de "GOTO", ce qui justifie la restriction faite au début de cet article. On dispose pour cela d'un algorithme qui, à partir d'un organigramme donne le programme régulier propre lui correspondant et réciproquement [AR].

Pour le passage organigramme $\rightarrow$ programme, on choisit un ensemble de points dans l'organigramme qui "coupent" tous les cycles (on peut pour cela choisir l'ensemble des points où il arrive plus d'une flèche : on définit un ensemble bien sûr non minimal mais qui vérifie ce que l'on désire). On attribue alors un nom de procédure à chacun de ces points et on éclate l'organigramme en un ensemble d'arbres (car il n'y a pas de cycle) dont les feuilles sont des occurrences d'appel de procédures. On en déduit immédiatement le système de procédures VG.

Exemple :

Dans le sens programme régulier $\rightarrow$ organigramme, on "regroupe" les appels de procédures pour reformer un graphe qui est alors l'organigramme.

4 - 2. Théorème

- La puissance d'expression des réguliers est la même que celle des rationnels:

$$\text{Reg}(F) \stackrel{\text{pe}}{=} \text{Rat}(F)$$

- La puissance d'expression des réguliers propres est la même que celle des rationnels propres : $\text{PReg}(F) \stackrel{\text{pe}}{=} \text{PRat}(F)$

On peut trouver la démonstration de ce théorème dans [CS2].

Il existe un algorithme qui permet, à partir d'un programme régulier, de construire le programme rationnel équivalent et réciproquement (Cf. les transformations de programmes [CS2]), basé sur l'équivalence :

$$(\alpha_i = t_i) \equiv (\alpha_i = \text{REPEAT } t_i[\alpha \setminus 0 : (\text{pour toute occurrence } \alpha \text{ de } \alpha_i \text{ dans } t_i) ; \\ s \setminus s+1 : (\forall s \text{ signe terminal de } t_i) \text{ENDR}])$$

4 - 3. Conséquences

A tout organigramme peut être associé un programme itératif (et ce, sans adjonction de booléens ou autres marqueurs, dans la mesure où l'on dispose de l'itération de type général REPEAT-EXITn avec $n > 1$).

L'un des intérêts de ce résultat est que l'on peut transformer un programme itératif (dont la compilation exige une gestion de pile) à un programme récursif dont les appels récursifs sont terminaux. La gestion se fait donc par simple réécriture sans gestion de pile.

4 - 4. Propriété

$$\text{Rat}(F) \underset{\text{pe}}{=} \text{Reg}(F) \underset{\text{pe}}{\neq} \text{Sim}(F)$$

$$\text{PRat}(F) \underset{\text{pe}}{=} \text{PReg}(F) \underset{\text{pe}}{\neq} \text{PSim}(F)$$

Soit le système simple suivant : $\Sigma : \{ \alpha 1 = \text{IF } P1 \text{ THEN } I3 \text{ ELSE } I1 ; \alpha 1 ; I2 \text{ FI } \neq \}$

Il est clair que l'arbre solution est :

$$\text{où } I2^n = I2 \\ \quad \quad \quad | \\ \quad \quad \quad I2^{n-1}$$

$$I2^1 = I2$$

Cet arbre a donc une infinité de sous-arbres distincts ($I2^n(0), n \in \mathbb{N}$). Or d'après [CS2], le nombre de sous-arbres distincts d'un programme régulier est fini. D'où la propriété 4 - 4.

Cela amène la conclusion (évidente) que l'utilisation de procédures VG permet une puissance d'expression supérieure à celle des programmes itératifs ou issus d'organigramme.

4 - 5. Théorème

La classe des arbres solutions de programmes hybrides est exactement la classe des arbres algébriques.

La démonstration complète de ce théorème se trouve dans [HE]. Nous en donnons ici les grandes lignes.

Partant d'un résultat de B. Courcelle [CR] selon lequel la classe des arbres solutions d'un schéma récursif polyadique est la classe des arbres algébriques (i.e. arbres dont le langage de branches est algébrique déterministe complet), on donne un algorithme qui à tout schéma récursif polyadique associe un système d'équations ayant même arbre solution. L'idée générale est d'associer, pour toute équation du schéma de la forme $F(x_1, \dots, x_n) = \tau(x_1, \dots, x_n)$ (où τ est un terme sur les variables $x_1, \dots, x_n$), une valeur de sortie à tout x_i (par exemple $EXIT_{i-1}$).

D'autre part, on fournit un automate à pile déterministe qui reconnaît le langage de branche des arbres solutions des programmes hybrides. L'idée est d'associer un état q_0 à une sortie en séquence et des états q_i aux exits i . Dépilant un symbole de pile correspondant à l'itération, on passe de l'état q_i à l'état q_{i-1} et on dépille ainsi jusqu'à retrouver l'état q_0 .

4 - 6. Conséquence

L'une des conséquences importante de ce théorème est que la décidabilité de l'équivalence de tels programmes se rattache au problème ouvert de la décidabilité de l'équivalence des langages algébriques déterministes complets.

Du même coup, il paraît difficile d'établir un ensemble de transformations syntaxiques de programmes qui soit complet pour l'équivalence syntaxique (i.e. préservant l'arbre solution). Il devient donc intéressant de déterminer quels programmes peuvent se ramener à des programmes dont l'équivalence est décidable.

4 - 7. Théorème

- La classe des langages de branches des arbres solutions des programmes simples est celle des langages reconnus par pile vide ou symbole terminal par un automate déterministe simple.

- La classe des langages de branches des arbres solutions des programmes simples propres est exactement la classe des langages simples.

En reprenant la construction de l'automate reconnaissant le langage de branches d'un hybride définie dans [HE], on obtient le résultat voulu en l'appliquant aux programmes simples et aux programmes simples propres.

4 - 8. Conséquence

$$\text{PSim}(F) \stackrel{\text{pe}}{\neq} \text{Sim}(F) \stackrel{\text{pe}}{\neq} \text{Hyb}(F)$$

4 - 9. Théorème

$$\text{PSim}(F) \stackrel{\text{pe}}{=} \text{PHyb}(F)$$

Si T est un terme à valeur sur un système d'équations propres, l'ensemble des valeurs de sortie du terme T est réduit à l'ensemble des valeurs terminales de ses signes terminaux (à quelques substitutions de α_i par $\alpha_i \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{smallmatrix} 0 \\ 0 \end{smallmatrix}$ près).

Par conséquent " $T+n = T[s \setminus s+n : \forall s \text{ signe terminal de } T]$ " qui est vrai sur les rationnels l'est aussi sur les hybrides propres (aux substitutions citées au-dessus près).

Introduisons un nouveau symbole de fonction " i " (increase) tel que $\{i\} \cap F = \emptyset$.

On lui adjoint l'opérateur $\underline{i}$ défini par :

$$\underline{i}(A) = A[s \setminus s+1 : \forall s \text{ signe terminal de } A]$$

Il est clair alors que i^n est équivalent à $T+n$, (si l'on fixe cette interprétation de i)

De même on montre, cf [HE], que :

$$\left\{ \begin{array}{c} \alpha = * \\ | \\ A \end{array} \right\} \equiv \left\{ \begin{array}{c} \alpha = d \\ | \\ ; \\ \swarrow \quad \searrow \\ A \quad i \\ | \\ \alpha \end{array} \right\} \quad (1)$$

A partir de ces remarques et résultats, on peut définir un algorithme qui à partir d'un programme hybride propre construit le programme simple propre équivalent. La description totale de l'algorithme étant trop longue, on va en donner les grandes lignes et prendre un exemple d'application.

Cet algorithme opère en deux étapes :

1 - a) remplacer les occurrences d'appel de procédures α par $\alpha \begin{smallmatrix} \nearrow \\ \searrow \end{smallmatrix} \begin{smallmatrix} ; \\ 0 \end{smallmatrix}$

b) créer des procédures dont le corps correspond à une itération extérieure (non emboîtée) d'une procédure déjà existante (et ce pour toutes les itérations extérieures).

c) remplacer ces itérations extérieures par l'appel des procédures correspondantes.

d) décomposer les procédures ainsi créées en deux nouvelles procédures, d'après l'équivalence (1).

$$\text{Ainsi : } \left\{ \begin{array}{c} \alpha = * \\ | \\ A \end{array} \right\} \equiv \left\{ \begin{array}{c} \alpha = d \ ; \ \alpha' = ; \\ | \qquad \qquad \qquad / \quad \backslash \\ \alpha' \qquad \qquad \qquad A \quad i \\ \qquad \qquad \qquad \qquad \qquad | \\ \qquad \qquad \qquad \qquad \qquad \alpha \end{array} \right\}$$

e) absorber i dans A , c'est-à-dire remplacer A par $A[s \setminus i^{s+1}] : \forall s$ signe
 α α
 terminal de A tel que $\tau(s, A) = 0$

On itère le procédé (b,c,d,e) jusqu'à élimination totale des symboles d'itération.

2 - évaluer α , en substituant le terme obtenu pour α' dans l'équation $\alpha = d$ puis
 α'
 faire "descendre" le symbole "d" dans l'arbre A , en remarquant [HE] que :

$$\left(\begin{array}{c} d \equiv j-i \\ | \\ pe \\ j \end{array} \right), \left(\begin{array}{c} d \equiv A \\ | \\ pe \\ i \\ A \end{array} \right) \text{ et } \left(\begin{array}{c} d \\ | \\ pe \\ ; \\ \alpha_k \quad i \\ \qquad | \\ \qquad i \\ \qquad | \\ \qquad \alpha_1 \end{array} \right) \equiv \left(\begin{array}{c} ; \\ | \\ pe \\ \alpha_k \quad d \\ \qquad | \\ \qquad i \\ \qquad | \\ \qquad \alpha_1 \end{array} \right) \equiv \left(\begin{array}{c} ; \\ | \\ pe \\ \alpha_k \quad \alpha_1 \end{array} \right) \text{ ssi } \alpha_k \text{ est propre}$$

Exemple

Soit le système propre $\Sigma = \{\alpha_1, \alpha_2, \alpha_3\}$ avec $\alpha_1 = \dots$ $\alpha_2 = \dots$ $\alpha_3 =$

dans la première étape, on obtient :

dans la seconde étape, on obtient:

Le système $\{\alpha_3, \alpha_5\}$ ainsi défini est équivalent à α_3 défini au début, et est simple propre. (En procédant de la même manière pour α_1 et α_2 , on obtient le système simple propre équivalent à $\{\alpha_1, \alpha_2, \alpha_3\}$).

4 - 10. Conséquences

Le fait d'inclure des itérations dans des procédures VG n'apporte aucune puissance d'expression supplémentaire si ces procédures sont propres (i.e. si toutes les sorties de ces procédures se font en séquence de l'appel). En particulier, les formes d'itération "WHILE DO", "REPEAT UNTIL", et "FOR STEP UNTIL DO" et "LOOP EXIT-identifiant ENDFOR" n'augmentent pas la puissance d'expression des programmes écrits avec des procédures VG. Les programmes écrits en PASCAL et ADA avec des procédures VG peuvent donc s'écrire dans ces langages sans itération. Ces deux langages ont donc même puissance d'expression. Seul, à notre connaissance, le langage EXEL possède une puissance d'expression supérieure, puisqu'il permet l'écriture de programmes non propres.

Un deuxième intérêt est que l'on peut décider de l'équivalence de deux programmes hybrides propres (et en particulier sur PASCAL, ADA et tous les langages ayant même puissance d'expression), étant donné que les arbres solutions associés sont les arbres simples, i.e. le langage de branches de ces arbres est algébrique déterministe simple.

On obtient le schéma final d'inclusion (en puissance d'expression) suivant :

REFERENCES

- [AMP] E. ASCHCROFT - Z. MANNA et A. PNUELI - "Decidable properties of monadic functional schemes" in Theory of Machines and Computation (pp 3-17). ed. KOHAVI et PAZ Academic Press. Londres et New-York (1971)
- [AR] J. ARSAC - "La construction de programmes structurés" ed. DUNOD, Paris (1977)
- [BJ] C. BOHM - C. JACOPINI - "Flow diagrams, turing machines and languages with only two formation rules" Com. ACM 9 (1966) pp 366-371
- [CA] P. CASTERAN - "Structures de contrôle: Définitions opérationnelles et algébriques" Thèse 3ème cycle Paris 7 (1979)
- [CH] A. CHANDRA - "On the properties and applications of program schemes" Ph. D. Stanford (1977)
- [CR] B. COURCELLE - "Recursive schemes algebraic trees and deterministic languages" SWAT (1974)
- [CN] G. COUSINEAU - M. NIVAT - "On rational expressions representing infinite rational tree application to the structure of flow-charts" MFCS Olomouc(1979)
- [CS1] G. COUSINEAU - "An algebraic definition for control structures" à paraître dans Theoretical Computer Science
- [CS2] G. COUSINEAU - "Les arbres à feuilles indicées: un cadre algébrique pour l'étude des structures de contrôle." Thèse d'état Paris 7 (1977)
- [FK] R. FLOYD - D. KNUTH - "Notes on avoiding GOTO statements." Inf. Processing Letter 1 (1971)
- [GL] S. GARLAND - D. LUCKHAM - "Program schemes, recursions schemes and formal languages." JCSS 7, 2 pp 119-160 (1973)
- [CH] C. HENRY - "Résolution d'équations algébriques sur le magma libre. Application aux transformations de programmes." Thèse 3ème cycle Paris 7 (1978)
- [JW] K. JENSEN - N. WIRTH - "Pascal user's manual and report." Seconde édition SPRINGER-VERLAG (1975)
- [NR] L. NOLIN - G. RUGGIU - "A formalisation of EXEL." Sigact-Sigplan Symposium on the principles of programming languages Boston (1973)
- [*] " Preliminary ADA Reference manual". ACM - Sigplan Notices - Vol 14, nb 6, part A (June 1979)