N° 5 1 1

Dactyl

UNIVERSITE DE NANCY I

U.E.R. DE MATHEMATIQUES

Contribution à la formalisation de la sémantique d'un langage de programmation

Application à Algol 68

THESE

Présentée pour l'obtention du doctorat de spécialité de mathématiques appliquées

par :

Jean-Pierre FINANCE

Soutenue le 25 Juin 1974

JURY :

M. C. PAIR	Président
M. M. NIVAT	Examinateurs
M. J. C. DERNIAME	

UNIVERSITE DE NANCY I

U.E.R. DE MATHEMATIQUES

Contribution à la formalisation de la sémantique d'un langage de programmation

Application à Algol 68

THESE

Présentée pour l'obtention du doctorat de spécialité de mathématiques appliquées

par :

Jean-Pierre FINANCE

Soutenue le 25 Juin 1974

JURY :

M. C. PAIR	Président
M. M. NIVAT	Examinateurs
M. J. C. DERNIAME	

Que Monsieur le Professeur M. NIVAT soit remercié de l'honneur qu'il me fait en venant juger ce travail.

Monsieur J.C. DERNIAME a bien voulu participer à ce jury, qu'il trouve ici l'expression de mon amicale reconnaissance.

Monsieur le Professeur C. PAIR est à l'origine de cette étude, prodiguant avec une patiente bienveillance ses conseils et ses encouragements il n'a cessé de me faire profiter de sa haute compétence, aussi est-ce un grand plaisir pour moi de le remercier aujourd'hui de tout ce qu'il m'a apporté.

J'adresse mes remerciements à Monsieur REMY pour sa collaboration à une partie de cette étude et je ne saurais oublier Messieurs LESCANNE, MARCHAND, MOHR et QUERE pour les critiques, les conseils et les suggestions qu'ils m'ont adressés.

Mes remerciements iront aussi à Madame HAMADI qui a assuré l'ingrate réalisation matérielle de ce travail, à Madame RICARELLI qui y a participé ainsi qu'à Monsieur DEBERDT de l'IREM qui a assuré le tirage.

A Chantal

Sommaire

0

Rappels

0. RAPPELS

Dans ce chapitre on rappelle brièvement les définitions et propriétés utilisées ultérieurement et qui concernent les notions de système à point fixe [0.1] et de système formel [0.2] .

On fera les conventions suivantes :

- les références aux différents paragraphes de cette étude seront placées entre crochets (par exemple [3.1] désigne le premier paragraphe du chapitre 3)
- la bibliographie sera divisée en plusieurs rubriques ; on y trouvera des références relatives à :
  - Algol 68, elles commencent par la lettre A et sont placées entre crochets (ainsi [A4] se rapporte au quatrième ouvrage de cette rubrique. Une exception concernera le rapport Algol 68 pour lequel [A] sera préféré à [A2] ; de plus [A-1.1.5] , par exemple, en désignera le paragraphe 1.1.5).
  - la notion de calcul (en abrégé C)
  - la logique mathématique (L)
  - la théorie du point fixe (PF)
  - la sémantique des langages de programmation (S)
  - la théorie des structures d'information (SI)
  - la théorie des langages (T)

0.1 Fonctions continues et systèmes à point fixe

0.1.1 Théorème du point fixe :

0.1.1.1 Treillis :

Définition 1 : Un treillis est un graphe $(E, \leqslant)$ où $\leqslant$ est une relation d'ordre telle que toute partie de E à 2 éléments $\{x,y\}$ possède une borne supérieure (notée $\sup\{x,y\}$) et une borne inférieure (notée $\inf\{x,y\}$).

Définition 2 : Un treillis $(E, \leqslant)$ est dit complet si tout sous ensemble A de E admet une borne supérieure (notée sup A).

En particulier E admet un plus grand élément $\top_E$ (ou simplement $\top$ lorsqu'il n'y a pas d'ambiguïté).

Proposition 1 : Si $(E, \leqslant)$ est une treillis complet, tout sous ensemble A de E admet une borne inférieure (notée inf A) . En particulier E admet une borne inférieure $\bot_E$ (ou simplement $\bot$).

Démonstration :

L'ensemble des minorants de A est un sous ensemble de E (éventuellement vide), il admet une borne supérieure qui est la borne inférieure de E .

On remarque que $\sup \emptyset = \bot$ et $\inf \emptyset = \top$.

Soit $(E, \leqslant)$ un treillis. Pour tout ensemble F on définit le graphe $(\mathcal{F}(F,E), \leqslant^{F,E})$, où $\mathcal{F}(F,E)$ est l'ensemble des applications de F dans E, par :

$f \leqslant^{F,E} g$ si et seulement si $(\forall x \in F)\ (f(x) \leqslant g(x))$.

En particulier, si $[1,n]$ désigne l'ensemble des n premiers entiers naturels, on peut identifier E^n avec $\mathcal{F}([1,n],E)$ et donc définir le graphe $(E^n, \leqslant^{[1,n],E})$ plus simplement noté $(E^n, \leqslant^n)$.
(Il est donc caractérisé par : $(x_1,\dots,x_n) \leqslant^n (y_1,\dots,y_n)$ si et seulement si $(\forall 1 \leqslant i \leqslant n)\ (x_i \leqslant y_i)$).

Théorème 1 : Avec les notations précèdentes, $(E^n, \leqslant^n)$ et $(\mathcal{F}(F,E), \leqslant^{F,E})$ sont des treillis, quels que soient $n \in \mathbb{N}$ et F .

De plus si $(E, \leqslant)$ est complet il en est de même de ces treillis.

Dans la suite de [0.1] "fonction" doit être pris au sens de "fonction totale" (ou application).

0.1.1.2 Fonctions continues :

Définition 3 : Soient $(E, \leqslant)$ et $(F, \leqslant)$ deux treillis complets et f une application de E dans F . f est dite continue si pour toute suite croissante (x_n) d'éléments de E , $f(\sup\{x_n \mid n \in \mathbb{N}\}) = \sup\{f(x_n) \mid n \in \mathbb{N}\}$.

Proposition 2 : Toute fonction continue est monotone croissante.
Soient $x,y \in E$; posons $x_o = x$ et pour $i \geqslant 1$, $x_i = y$ alors si $x \leqslant y$:

$$f(x) = f(x_o) \leqslant \sup\{f(x_o), f(x_1)\} = f(\sup\{x_o, x_1\}) = f(y) .$$

Remarques :

1) $\sup\{x_n \mid n \in \mathbb{N}\}$ s'appelle limite de la suite (x_n) et sera noté $\lim_n x_n$; l'égalité définissant une fonction continue f devient :
$f(\lim_n x_n) = \lim_n f(x_n)$.

2) Scott dans [PF 7] et [PF 9] donne une définition plus générale de la continuité : un sous ensemble A de E est dit régulier s'il contient la borne supérieure de chacun de ses sous ensembles finis.

Une fonction $f : E \longrightarrow F$ est continue au sens de Scott si pour tout sous ensemble régulier A de E : $f(\sup A) = \sup\{f(x) \mid x \in A\}$

(continuité pour la topologie "borne supérieure sur E").

Il est clair qu'une telle fonction est continue au sens de la définition 3
La réciproque est vraie si E est dénombrable (démonstration a)), par contre elle peut être fausse si E a la puissance du continu comme le montre l'exemple b).

a) Supposons E dénombrable et prouvons que f est continue au sens de Scott dès qu'elle est continue au sens de la définition 3.

Soit A un sous ensemble régulier de E , $b = \sup A$ et f une fonction continue au sens de la définition 3. Deux cas sont à envisager :

a1) $b \in A$. Notons $a = \sup\{f(x) \mid x \in A\}$.

f est croissante (proposition 2) donc pour tout $x \in A$, $f(x) \leqslant f(b)$;

et par suite $a \leqslant f(b)$.

Mais $b \in A$: $f(b) \leqslant a$ d'où le résultat.

a2) $b \notin A$. Nécessairement card $A = \infty$ et E étant dénombrable il en est de même de A . Notons $a_1, a_2, \dots, a_n, \dots$ les éléments de A et posons

$$b_1 = a_1 \ ;$$

$$b_n = \sup\{b_{n-1}, a_n\} \quad \text{pour } n \geqslant 2 .$$

Par construction (b_n) est une suite croissante de A et $\lim_n b_n = b$; on en déduit que $f(b) = f(\lim_n b_n) = \lim_n f(b_n)$.

Mais $b_n \in A$ pour tout n , donc $\lim_n f(b_n) \leqslant \sup\{f(x) \mid x \in A\}$
et $(\forall x \in A)\ (\exists n \geqslant 1)\ (x \leqslant b_n)$ donc

$$\sup\{f(x) \mid x \in A\} \leqslant \lim f(b_n)$$

f conserve bien la borne supérieure de tout ensemble régulier.

b) Soit F un ensemble ayant la puissance du continu et $E = \mathcal{P}(F)$; et soit A l'ensemble des parties dénombrables de F .

$(E, \subset)$ est un treillis complet et A en est un sous ensemble régulier.

Définissons alors la fonction $\varphi : E \longrightarrow E$ qui associe à tout élément dénombrable de E l'ensemble vide et à tout élément non dénombrable de E l'ensemble F .

- φ est continue au sens de la définition 3 :
Soit B_n une suite croissante de E et donc $\lim_n B_n = \bigcup_n B_n$.

Si l'un des B_n n'est pas dénombrable, la réunion n'est pas dénombrable et donc $\varphi(\lim_n B_n) = F$ et $\lim_n \varphi(B_n) = F$.

Dans le pas contraire, la réunion est dénombrable et l'on a :

$$\varphi(\lim_n B_n) = \emptyset \quad \text{et} \quad \lim_n \varphi(B_n) = \emptyset \ .$$

- Cependant $\varphi(\sup A) = \varphi(F) = F$ et pour tout $B \in A$, $\varphi(B) = \emptyset$ donc $\sup \{ \varphi(B) \mid B \in A\} = \emptyset$, c'est-à-dire que φ n'est pas continue au sens de Scott.

0.1.1.3 Théorème du point fixe : [PF 7] , [PF 9] :

Théorème 2 :

Soit $f : E \longrightarrow E$ où $(E, \leqslant)$ est un treillis complet. Si f est croissante il existe une solution minimum x_o à l'équation $f(x) = x$ (i.e. x_o est solution de cette équation et toute autre solution y vérifie $x_o \leqslant y$) .
Si de plus f est continue x_o est donné par :

$$x_o = \lim_n f^n(\bot) \qquad (\bot \text{ borne inférieure de E}) \ .$$

Démonstration :

- Soit $A = \{y \in E \mid f(y) \leqslant y\}$ et notons $x_o = \inf A$ (qui existe proposition 1) $(\forall y \in A)\ (x_o \leqslant y \Longrightarrow f(x_o) \leqslant y)$ et donc $f(x_o) \leqslant \inf A = x_o$
de plus $f(f(x_o)) \leqslant f(x_o)$ donc $f(x_o) \in A$ et ainsi $x_o \leqslant f(x_o)$.

- Si y_o vérifie $f(y_o) = y_o$ alors $y_o \in A$ et donc $x_o \leqslant y_o$.

Enfin si f est continue : $\bot \leqslant x_o \Longrightarrow f(\bot) \leqslant f(x_o) = x_o$

et donc pour tout $n \in \mathbb{N}$: $f^n(\bot) \leqslant x_o$.

De plus $f(\sup\{f^n(\bot) \mid n \in \mathbb{N}\}) = \sup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\} = \sup\{f^n(\bot) \mid n \in \mathbb{N}\}$

et donc $x_o \leqslant \sup\{f^n(\bot) \mid n \in \mathbb{N}\}$

d'où le résultat.

0.1.1.4 Remarque :

En fait la notion de treillis complet est très riche, on peut introduire des structures plus pauvres dans lesquelles le théorème du point fixe sera encore vrai. Cela serait nécessaire si on voulait développer davantage [2] et notamment les problèmes "d'extension par définition d'accès nouveaux" [2.5.3] . La théorie précédente sera grandement suffisante pour notre étude, aussi contentons-nous d'indiquer brièvement un appauvrissement possible des notions (voir en particulier [PF 3] , [PF 4] , [PF 5] , [PF 10] , [SI 7]).

Définition 4 : Un ensemble ordonné $(E, \leqslant)$ est dit inductif si :

- il admet un plus petit élément $\bot$
- toute suite croissante d'éléments de E admet une borne supérieure.

Exemples : - Un treillis complet est un ensemble ordonné inductif

- Si $\mathbb{N}$ est l'ensemble des entiers naturels, $E = \mathbb{N} \cup \{\bot\}$ est inductif pour la relation d'ordre $a \leqslant b \Longleftrightarrow a = b$ ou $a = \bot$.
- Si $(E, \leqslant)$ est inductif, l'ensemble $\mathcal{F}(E,E)$ des fonctions de E dans E est inductif pour la relation d'ordre :

$$f \leqslant^{\mathcal{F}} g \Longleftrightarrow f(x) \leqslant g(x) \quad \text{pour tout } x \in E \ .$$

- Si $(E, \leqslant)$ est inductif, $(E^n, \leqslant^n)$ est inductif [0.1.1.1] .

Les notions de continuité et de croissance sont les mêmes que dans les treillis. Le théorème du point fixe est alors vrai pour les fonctions continues (et non plus pour toutes les fonctions croissantes) :

Toute fonction continue d'un ensemble inductif $(E, \leqslant)$ dans un ensemble inductif $(F, \leqslant)$ admet un plus petit point fixe x_o donné par

$$x_o = \sup \{f^n(\bot) \mid n \in \mathbb{N}\} \ .$$

0.1.2 Ensemble générateur de fonctions ([PF 1] , [PF 2]) :

Définition 5 : Soit $\mathcal{L}$ un ensemble, nous noterons $\mathcal{F}^k$ l'ensemble des fonctions définies dans $\mathcal{L}^k$ à valeurs dans $\mathcal{L}$. (En particulier on identifie l'élément a de $\mathcal{L}$ et la fonction à 0 variable f_a à valeur a dans $\mathcal{L}$: $\mathcal{F}^o = \mathcal{L}$). On appelle composition l'opérateur Γ qui, quels que soient les entiers k et ℓ , associe aux fonctions $g, g_1, \ldots, g_\ell$ (avec $g \in \mathcal{F}^\ell$ et $g_i \in \mathcal{F}^k$ pour $i = 1, \ldots, \ell$) la fonction $f = \Gamma(g, g_1, \ldots, g_\ell) \in \mathcal{F}^k$ définie par :

$$\forall (x_1, x_2, \ldots, x_k) \in \mathcal{L}^k \quad f(x_1, \ldots, x_k) = g(g_1(x_1, \ldots, x_k), \ldots, g_\ell(x_1, \ldots, x_k)) \ .$$

Définition 6 : Soit $\mathcal{C} \subset \bigcup_{k \in \mathbb{N}} \mathcal{F}^k$ un ensemble fini ou infini de fonctions appelées fonctions de base. On appelle famille de fonctions engendrée par $\mathcal{L}$ et $\mathcal{C}$, notée $\mathrm{Eng}(\mathcal{L}, \mathcal{C})$, le plus petit ensemble de fonctions contenu dans $\bigcup_{k \in \mathbb{N}} \mathcal{F}^k$ qui soit stable par composition et contienne les ensembles suivants :

(1) $\mathcal{P}$ ensemble des fonctions projections p_i^k $(k \in \mathbb{N} \ ; \ 1 \leqslant i \leqslant k)$ définies par :

$$(\forall (x_1, \ldots, x_k) \in \mathcal{L}^k)\ (p_i^k(x_1, \ldots, x_k) = x_i)$$

(2) $\mathcal{L}$ (ensemble des fonctions à 0-variable à valeurs dans $\mathcal{L}$)

(3) $\mathcal{E}$

Théorème 3 :

Si $(\mathcal{L}, \leqslant)$ est un treillis complet, pour que tout élément de $\mathrm{Eng}(\mathcal{L}, \mathcal{E})$ soit une fonction continue il faut et il suffit que les éléments de $\mathcal{E}$ soient des fonctions continues.

Démonstration :

Si les éléments de $\mathcal{E}$ sont des fonctions continues, l'ensemble des fonctions continues appartenant à $\bigcup_{k \in \mathbb{N}} \mathcal{F}^k$ contient $\mathcal{P}$, $\mathcal{E}$, $\mathcal{L}$ et il est stable par composition, il contient donc $\mathrm{Eng}(\mathcal{L}, \mathcal{E})$.

0.1.3 Système algébrique sur $\mathcal{L}$.

0.1.3.1 Définition.

Définition 7 :

Soit $(\mathcal{L}, \leqslant)$ un treillis complet. On appelle système algébrique (ou système à point fixe) à p inconnues sur $\mathcal{L}$ relativement aux fonctions de base $e_1, e_2, \ldots, e_n, \ldots$ de $\mathcal{E}$ $(e_i : \mathcal{L}^{k_i} \longrightarrow \mathcal{L})$ tout système du type :

$$\mathcal{S} \begin{cases} X_1 = f_1(X_1, \ldots, X_p) \\ \vdots \\ X_p = f_p(X_1, \ldots, X_p) \end{cases}$$

où $f_i \in \mathrm{Eng}(\mathcal{L}, \mathcal{E})$, les X_i sont les inconnues du système.

(Il est clair que pour un système donné, le nombre de fonctions de base qui interviennent est fini).

Théorème 4 :

Si les fonctions de base sont continues, le système $\mathcal{S}$ admet une solution minimum.

Il suffit d'appliquer les théorèmes 2 (point fixe) et 3 au treillis complet $\mathcal{L}^p$.

0.1.3.2 Cas particulier :

Soit E et F deux ensembles et $E^* = \bigcup_{n \geqslant 0} E^n$. On note $\mathcal{F}(E^*, \mathcal{P}(F))$ l'ensemble des applications φ de E^* dans $\mathcal{P}(F)$ telles qu'il existe un entier naturel unique n pour lequel $\varphi(x_1, \ldots, x_p) \neq \emptyset \Longrightarrow p = n$.

On dit que n est l'arité de φ.

Alors $\mathcal{L} = (\mathcal{F}(E^*, \mathcal{P}(F)), \subset)$ est un treillis complet (théorème 1) où $\subset$ est définie par : $\varphi \subset \psi \Leftrightarrow (\forall x \in E^*)\ (\varphi(x) \subset \psi(x))$.

Soit $\mathcal{E}$ un ensemble de fonctions de base ($\subset \bigcup_{k \in \mathbb{N}} \mathcal{F}^k$ (définitions 5 et 6)), un système algébrique sur $\mathcal{L}$ à p inconnues $\varphi_1, \ldots, \varphi_p$ relativement à $\mathcal{E}$ est de la forme :

$$\mathcal{S}_1 \begin{cases} \varphi_1 = f_1(\varphi_1, \ldots, \varphi_p) \\ \vdots \\ \varphi_p = f_p(\varphi_1, \ldots, \varphi_p) \end{cases}$$

(Les $f_1, \ldots, f_p$ sont parfois appelées fonctionnelles).

0.1.3.3 Sous systèmes algébriques :

Soit $\mathcal{S}$ un système algébrique. Un sous système $\mathcal{S}'$ de $\mathcal{S}$ est un sous ensemble des équations de $\mathcal{S}$:

$$\mathcal{S}' \begin{cases} X_{i_1} = f_{i_1}(X_1, \ldots, X_p) \\ \vdots \\ X_{i_k} = f_{i_k}(X_1, \ldots, X_p) \end{cases}$$

Si pour tout $j \notin \{i_1, \ldots, i_k\}$, pour tout $1 \leqslant \ell \leqslant k$ on a :

$$(\forall X_j, X_j')\ (f_{i_\ell}(X_1, \ldots, X_j, \ldots, X_p) = f_{i_\ell}(X_1, \ldots, X_j', \ldots, X_p))$$

(c'est-à-dire intuitivement que les fonctions f_{i_ℓ} ne peuvent "dépendre" d'autres variables que de $X_{i_1}, \ldots, X_{i_k}$), on dit que le sous système $\mathcal{S}'$ est complet.

Exemple : Soient f et g deux fonctions de $\mathcal{L}^2$ dans $\mathcal{L}$, h une fonction de $\mathcal{L}^3$ dans $\mathcal{L}$ (où $(\mathcal{L}, \leqslant)$ est un treillis complet) et $\mathcal{S}$ le système :

$$\mathcal{S} \begin{cases} X_1 = \Gamma(f, p_1^3, p_2^3)\ (X_1, X_2, X_3) \\ X_2 = \Gamma(g, p_2^3, p_1^3)\ (X_1, X_2, X_3) \\ X_3 = h(X_1, X_2, X_3) \end{cases}.$$

les deux premières fonctions $f_1 = \Gamma(f, p_1^3, p_2^3)$ et $f_2 = \Gamma(g, p_2^3, p_1^3)$ ne dépendent pas de X_3 : le sous système

$$\mathcal{S}' \begin{cases} X_1 = f_1(X_1,X_2,X_3) \\ X_2 = f_2(X_1,X_2,X_3) \end{cases}$$

est complet ; on vérifie qu'il n'en est pas de même du sous système formé des deux dernières équations de $\mathcal{S}$.

On peut alors transformer $\mathcal{S}'$ en un système algébrique à deux inconnues X_1 et X_2 , il suffit pour cela d'identifier f_1 à $\Gamma(f,p_1^2,p_2^2)$ et f_2 à $\Gamma(f,p_2^2,p_1^2)$. Plus généralement : à tout sous système complet à k équations $\mathcal{S}'$ est associé un système algébrique à k inconnues (nous ne définirons pas formellement cette association).

En particulier si les fonctions de base sont continues, le système algébrique ainsi obtenu admet une solution minimum.

On appelle réunion de deux sous systèmes $\mathcal{S}'$ et $\mathcal{S}''$ le sous système $\begin{cases} \mathcal{S}' \\ \mathcal{S}'' \end{cases}$ formé des équations de $\mathcal{S}'$ et $\mathcal{S}''$.

Il est clair que la réunion de deux sous systèmes complets est un sous système complet.

0.2 Généralités sur les systèmes formels :

0.2.1 Système formel :

Définition 1 : Un système formel $\mathcal{F}$ est un quadruplet (Alph, F,X,R) où

- Alph est un ensemble fini ou dénombrable : alphabet de $\mathcal{F}$
- F est un langage sur Alph : ensemble des formules
- X est un sous ensemble de F : ensemble des axiomes
- R est un ensemble fini de relations n-aires sur F appelées règles d'inférence (si $r \in R$ et $r(\varphi_1,\dots,\varphi_{n-1},\varphi_n)$ est vrai, on dit que des hypothèses $\varphi_1,\dots,\varphi_{n-1}$ de r on déduit la conclusion φ_n).

L'ensemble des théorèmes de $\mathcal{F}$ est le plus petit sous ensemble T de F contenant X et stable par application des règles de R (i.e. que si des hypothèses $\varphi_1,\dots,\varphi_{n-1}$ de r appartiennent à T , la conclusion φ_n appartient à T).

Exemple 1 :

Le calcul des propositions est un système formel défini ([L4], [L8]) par : Alph = $\{\neg, \supset, (\ ,\ )\} \cup A$

$\neg$ et $\supset$ sont deux connecteurs logiques (respectivement connecteur "non" et connecteur "implique") ; les éléments de A sont appelés formules atomiques. L'ensemble F des formules est la solution minimale du système à point fixe :

$$F = A \cup \neg F \cup (F \supset F) .$$

Les axiomes sont de l'une des trois formes suivantes :

Pour toutes formules φ, ψ, ρ :

(1) - $\varphi \supset (\psi \supset \varphi)$

(2) - $(\varphi \supset (\psi \supset \rho)) \supset ((\varphi \supset \psi) \supset (\varphi \supset \rho))$

(3) - $(\neg\varphi \supset \neg\psi) \supset (\psi \supset \varphi)$.

La seule règle d'inférence introduite est le Modus Ponens :

de φ et de $\varphi \supset \psi$ on déduit ψ .

Exemple 2 :

Brièvement un système formel du type calcul des prédicats du premier ordre avec égalité (ou théorie du premier ordre) est composé : [L9]

- d'un alphabet Alph contenant, outre l'ensemble de symboles $\{\neg, \supset, \exists, (\ ,\ )\}$ un ensemble V de variables, un ensemble L de symboles fonctionnels et un ensemble P de symboles de prédicats (qui contient en particulier le prédicat "$\equiv$" représentant l'égalité formelle ; par habitude on écrira $u \equiv v$ plutôt que $\equiv uv$)
- d'un ensemble F de formules construit à l'aide de $\neg, \supset, \exists$, (,) et d'un ensemble A de formules atomiques définies à partir de L , V et P par un mode de construction du "type" schémas fonctionnels [L6]
- d'un ensemble d'axiomes contenant en plus des axiomes du calcul des propositions (exemple 1), certains axiomes relatifs à $\equiv$ et à $\exists$; par exemple :

(4) - $u \equiv u$ pour toute formule u

(5) - $u_1 \equiv v_1(\supset \dots \supset (u_n \equiv v_n \supset (fu_1\dots fu_n \equiv fv_1\dots fv_n))\dots)$

pour toutes formules $u_1,\dots,u_n, v_1,\dots,v_n$ et tout symbole fonctionnel f (à n arguments)

(6) - $u_1 \equiv v_1(\supset \dots \supset (u_n \equiv v_n \supset (pu_1\dots u_n \supset pv_1\dots v_n))\dots)$

pour toutes formules $u_1,\dots,u_n,v_1,\dots,v_n$ et tout symbole de prédicat p

(7) - $\varphi_{u}[a] \supset \exists x\, \varphi$

où $\varphi_{u}[a]$ désigne la formule déduite de φ en remplaçant chaque occurence

libre de u dans φ par a (une occurence de u dans φ est dite liée si elle apparaît dans un facteur de φ de la forme $\exists u\ \psi$, sinon elle est dite libre).

En plus de ces axiomes dits logiques, $\mathcal{F}$ pourra comporter un certain nombre d'axiomes propres à la théorie (du premier ordre) considérée.

Enfin, outre le Modus Ponens, l'ensemble R des règles d'inférence contient une règle relative au quantificateur "$\exists$" , elle peut être :

de $\varphi \supset \psi$ on déduit $\exists u\ \varphi \supset \psi$ si u n'est pas libre dans ψ .

Remarque 1 : En plus des connecteurs $\neg$ et $\exists$ il est agréable d'introduire les connecteurs "$\wedge$" (et), "$\vee$" (ou) qui permettent d'abréger certaines formules.

Exemple 3 :

Dans toute la suite on s'intéressera à une version simplifiée d'un théorie du premier ordre :

- l'alphabet, outre $\{\neg, \supset, (\ ,\ )\}$, ne contiendra que le prédicat $\equiv$ et un ensemble L de symboles fonctionnels
- si $\hat{L}$ est l'ensemble des schémas fonctionnels sur L , l'ensemble A des formules atomiques est un sous ensemble de $\hat{L} \equiv \hat{L}$
- l'ensemble des axiomes logiques comprendra :
  - les axiomes du type (1), (2), (3) du calcul des propositions
  - les axiomes du type (4) et (5) concernant l'égalité formelle
  - les axiomes :

  (6) $u_1 \equiv v_1 \supset (u_1 \equiv v_2 \supset v_1 \equiv v_2)$ pour toute formules u_1, v_1, v_2

auxquels on ajoutera certains axiomes propres au problème considéré

- la seule règle d'inférence sera le Modus Ponens.

Un tel système formel peut encore être dit du type calcul des propositions avec égalité.

Définition 2 :

Une suite finie $D : \varphi_1, \varphi_2, \ldots, \varphi_n$ d'éléments de F est une démonstration si pour tout $1 \leqslant i \leqslant n$:

- ou φ_i est un axiome
- ou φ_i se déduit de certaines formules $\varphi_{j_1}, \ldots, \varphi_{j_n}$ le précèdant dans D par application de certaines règles de R. On montre alors :

Une formule est un théorème si et seulement si elle apparaît dans une démonstration. En particulier on dit que D est une démonstration de φ_n .

Définition 3 :

Si $\mathcal{F}$ est un système formel (Alph, F, X, R), un système

$\mathcal{F}'$ = (Alph, F, X', R) est une extension simple de $\mathcal{F}$ si

$X \subset X'$.

Notations :

- On note $\mathcal{F}[\Gamma]$ l'extension simple de $\mathcal{F}$ obtenue en ajoutant l'ensemble de formules Γ à X , en particulier $\mathcal{F}[\{\varphi\}]$ s'écrira plus simplement $\mathcal{F}[\varphi]$ (où φ est une formule).
- ψ est un théorème de $\mathcal{F}[\varphi]$ se notera $\varphi \vdash_{\mathcal{F}} \psi$ ou simplement $\varphi \vdash \psi$ s'il n'y a pas ambiguïté.
- En particulier si ψ est un théorème de $\mathcal{F}$ on écrira $\vdash \psi$ et toute règle d'inférence pourra s'écrire sous la forme :

$$\varphi_1, \ldots, \varphi_n \vdash \psi \quad .$$

Exemple : le Modus Ponens s'écrit :

pour toutes formules $\varphi, \psi \in F$; $\varphi, \varphi \supset \psi \vdash \psi$.

Remarque 2 :

Il faut distinguer les théorèmes du système formel et les théorèmes concernant ce système formel (appelés métathéorèmes) ; de même on doit distinguer une démonstration formelle d'une métadémonstration. Cependant pour alléger les notations nous éviterons le préfixe "méta" le plus souvent possible (essentiellement à partir de [2]).

Remarque 3 :

Lorsqu'on énonce une propriété d'un système formel (métathéorème par exemple) on utilise des symboles n'appartenant pas à Alph (métasymboles ou variables syntaxiques) par exemple φ et ψ ci-dessus.

On utilisera de telles variables pour l'énoncé d'axiomes : on écrira en fait des ensembles d'axiomes (schémas d'axiomes) plus souvent que des axiomes. Ainsi le schéma $\{\varphi \supset (\psi \supset \varphi) \mid \varphi, \psi \in F\}$ contient par exemple

$a \equiv b \supset (c \equiv d \supset a \equiv b))$.

Remarque 4 :

Souvent l'ensemble F est défini par un système à point fixe à partir d'un ensemble A de formules "atomiques" et des connecteurs logiques (c'est ce qui se passera dans la suite).

Un certain nombre de métadémonstrations pourront se faire :

- soit par récurrence sur la longueur $\|\varphi\|$ d'une formule φ :
  cette longueur est définie (dans le cas de l'exemple 3) par :

$$\|a\| = 0 \quad \text{si} \quad a \in A \quad \text{(formule atomique)}$$

$$\|\neg\varphi\| = 1 + \|\varphi\|$$

$$\|\varphi_1 \supset \varphi_2\| = 1 + \|\varphi_1\| + \|\varphi_2\|$$

(ainsi, par exemple, $\|u_1 \equiv v_1 \supset (u_1 \equiv v_2 \supset v_1 \equiv v_2)\| = 2$).

Il faut distinguer cette notion de celle de longueur d'un schéma fonctionnel u notée $|u|$ (par exemple si f est un symbole fonctionnel 3-aire et u_1, u_2, u_3 trois symboles 0-aires alors $|f\ u_1\ u_2\ u_3| = 4$)

- soit en utilisant la notion de réalisation, ce type de raisonnement sera développé en [2.4.3] .

Enonçons maintenant un important métathéorème reliant les notions de démonstration formelle et de métadémonstration.

0.2.2 Métathéorème de la déduction :

Soit $\mathcal{F}$ un système formel du type calcul des propositions avec égalité (exemple 3 de [0.2.1]).

Théorème 1 : Pour toutes formules φ, ψ de $\mathcal{F}$:

$$\vdash_{\mathcal{F}} \varphi \supset \psi \quad \text{si et seulement si} \quad \varphi \vdash_{\mathcal{F}} \psi$$

([L9] page 33).

Ce théorème est encore valable dans le cas d'une théorie du premier ordre lorsque φ est une formule fermée de $\mathcal{F}$ (i.e. ne contient aucune variable libre).

Corollaire :

Soient $\varphi_1, \varphi_2, \ldots, \varphi_n, \psi$ des formules de $\mathcal{F}$

$\vdash_{\mathcal{F}} \varphi_1 (\varphi_2 \supset (\ldots(\varphi_n \supset \psi)\ldots))$ si et seulement si ψ est un théorème de $\mathcal{F}[\{\varphi_1, \ldots, \varphi_n\}]$.

0.2.3 Tautologies :

Notation : nous noterons $\mathcal{B}$ = {VRAI,FAUX} l'ensemble des valeurs booléennes, $H_\neg$, $H_\wedge$, $H_\vee$, $H_\supset$ les fonctions booléennes habituelles (non, et, ou, implique). Rappelons qu'elles sont définies par :

- $H_\neg$ (VRAI) = FAUX ; H (Faux) = VRAI
- $H_\wedge$ (a,b) = VRAI si a = b = VRAI ; FAUX sinon
- $H_\vee$ (a,b) = FAUX si a = b = FAUX ; VRAI sinon
- $H_\supset$ (a,b) = FAUX si a = VRAI et b = FAUX ; VRAI sinon

Les systèmes formels $\mathcal{F}$ considérés ici sont toujours du type calcul des propositions avec égalité.

Définition 4 : Une fonction de vérité associée à un système formel $\mathcal{F}$ est une application $\mathfrak{V}$ de $\mathcal{F}$ dans $\mathcal{B}$ qui vérifie :

$$\mathfrak{V}(\neg\varphi) = H_\neg(\mathfrak{V}(\varphi)) \ ; \ \mathfrak{V}(\varphi \wedge \psi) = H_\wedge(\mathfrak{V}(\varphi), \mathfrak{V}(\psi)) \ ;$$

$$\mathfrak{V}(\varphi \vee \psi) = H_\vee(\mathfrak{V}(\varphi), \mathfrak{V}(\psi)) \ ; \ \mathfrak{V}(\varphi \supset \psi) = H_\supset(\mathfrak{V}(\varphi), \mathfrak{V}(\psi)) .$$

Définition 5 : Une formule φ de $\mathcal{F}$ est une tautologie si $\mathfrak{V}(\varphi)$ = VRAI pour toute fonction de vérité $\mathfrak{V}$ associée à $\mathcal{F}$.

Une formule ψ de $\mathcal{F}$ est une conséquence tautologique des formules $\varphi_1, \ldots, \varphi_n$ si, pour toute fonction de vérité $\mathfrak{V}$,

$$\mathfrak{V}(\varphi_1) = \ldots = \mathfrak{V}(\varphi_n) = \text{VRAI} \quad \text{entraîne} \quad \mathfrak{V}(\psi) = \text{VRAI} .$$

Exemples :

Si ρ est une formule de la forme $\varphi \supset (\psi \supset \varphi)$ c'est-à-dire est un axiome de type (1) de $\mathcal{F}$) ; $\mathfrak{V}(\rho) = H_\supset(\mathfrak{V}(\varphi), H_\supset(\mathfrak{V}(\psi), \mathfrak{V}(\varphi)))$.

Or si $\mathfrak{V}(\varphi)$ = VRAI alors $H_\supset(\mathfrak{V}(\psi), \mathfrak{V}(\varphi))$ = VRAI

ainsi on a toujours $\mathfrak{V}(\varphi \supset (\psi \supset \varphi))$ = VRAI : φ est une tautologie.

On peut vérifier qu'il en est ainsi de tout axiome du type (2) ou (3).

Théorème 2 (des tautologies) :

Si une formule ψ du système formel $\mathcal{F}$ est une conséquence tautologique de $\varphi_1, \ldots, \varphi_n$ alors ψ est un théorème de $\mathcal{F}[\{\varphi_1, \ldots, \varphi_n\}]$.
En particulier toute tautologie est un théorème de $\mathcal{F}$.

La réciproque de ce théorème est vraie dans le cas du calcul des propositions comme le montre l'exemple ci-dessus. (Le calcul des propositions étudie la démonstrabilité des formules d'un langage sans s'intéresser à la signification des formules atomiques).

1

Buts et méthodes d'une formalisation de la sémantique d'un langage de programmation

1 - Buts et méthodes d'une formalisation de la sémantique d'un langage de programmation

Dès l'apparition des langages de programmation évolués (en particulier d'Algol 60), un certain nombre d'études ont été menées, qui tentent de définir formellement la sémantique de ces langages. Dans ce chapitre après avoir réfléchi sur les buts d'une telle définition [1.1] on introduit les différents concepts qui se trouvent à sa base [1.2] puis on donne un bref apperçu des différentes formalisations déjà proposées [1.3] .

1.1 - Interêts d'une formalisation de la sémantique.

On trouve dans la littérature informatique une grande latitude d'interprétation du terme sémantique qui va de la conception des logiciens (liée à la notion de modèle) à une conception beaucoup plus vaste incluant les notions d'interprète, de calculs etc ...

Nous nous contenterons dans ce paragraphe d'une idée intuitive, le but des différentes méthodes de formalisation étant précisement d'en donner une définition complète. Intuitivement définir la sémantique d'un langage de programmation c'est associer un sens aux différents programmes de ce langage ; on dit encore lui associer une valeur semantique.

Le problème est donc double :

i) définir un ensemble de valeurs sémantiques (pour un langage donné)

ii) définir une application de l'ensemble des programmes (du langage) dans celui de ces valeurs.

L'étude de ces deux points passe par une reflexion sur les buts poursuivis :

a) En général, la définition d'un langage de programmation évolué comporte trois aspects :

a1) - définition formelle de la majeure partie de la syntaxe du langage à l'aide d'une grammaire (exprimée par exemple à l'aide de la notation de BACKUS : Algol 60, Simula, PL/1 ... ou en utilisant des formes de grammaires plus générales, comme la double grammaire d'Algol 68).

a2) - Définition moins formalisée de conditions syntaxiques supplémentaires (due essentiellement au fait qu'on se restreint en a1) à des grammaires du type "contexte libre" pour alléger les définitions)

a3) - Définition peu formelle de la sémantique.

Les définitions a2)et a3)sont formulées à l'aide d'une langue présentant quelques aspects d'une langue naturelle, en employant un style plus ou moins mathématique.

Si la formalisation de la syntaxe permet d'automatiser certaines parties de la compilation (notamment l'analyse syntaxique), le manque de précision dans la définition de la sémantique pose de nombreux problèmes à l'implémenteur. Il importe donc de donner un sens complet et non ambigu à tout programme du langage, ce qui devrait permettre de contribuer à la solution du difficile problème de la construction automatique d'un compilateur.

Remarquons que, même dans le cas d'une définition formelle, il peut être interessant de laisser indéfinis certains aspects du langage ce qui donne à l'implémentation certaines lattitudes d'optimisation et d'adaptation au matériel et au système d'exploitation utilisés.

b) Une définition précise de la sémantique d'un langage doit permettre au programmeur de le maitriser parfaitement et d'éviter ainsi la déplorable pratique actuelle qui consiste à aller vérifier ce qui fait un langage par des passages sur machine.
En particulier si jusqu'à présent les langages de programmation étaient essentiellement impératifs, on observe une certaine évolution vers des langages moins directifs, plus descriptifs, ou un plus grand choix est laissé au compilateur (calculs collatéraux par exemple). Cette tendance est une motivation supplémentaire à la définition précise de la sémantique qui doit permettre de connaitre la ou les séquences d'éxecution correspondant à une description donnée.

c) Une méthode générale de formalisation de la sémantique permettrait de définir plus facilement et plus correctement de nouveaux langages de programmation, en évitant en particulier certaines constructions contradictoires (de même que la notation de BACKUS est un outil simple et efficace pour la description d'une syntaxe). Elle doit permettre également une normalisation plus facile d'un langage ainsi qu'une comparaison aisée de différents langages.

d) Une formalisation doit contribuer à la résolution des problèmes de terminaison, de correction de programmes ou de compilateurs ; elle doit fournir des méthodes de preuves pour des langages évolués. On remarquera que certaines approches de la définition formelle de la sémantique sont précisement basées sur des méthodes de preuves [1.3] .

Il semble cependant que certains de ces objectifs soient difficilement conciliables sinon contradictoires : peut-on donner une définition formelle complète permettant des preuves (d) qui soit simple, claire et lisible par l'homme (a, b) ?

Le nombre d'approches différentes de la sémantique s'explique en partie par la diversité des buts poursuivis.

1.2 - Composantes de la définition de la sémantique

1.2.1 - Structure d'information

La première idée de valeur sémantique d'un programme est celle de fonction; un programme fait en effet passer de données à des résultats.

Pour définir une telle fonction, il faut définir ses ensembles de départ et d'arrivée. MAC CARTHY [S38] l'un des premiers à introduit la notion de vecteur d'état (suite finie de nombres) qui formalise l'idée intuitive d'état de mémoire.

Cependant, les informations manipulées par un programme peuvent être beaucoup plus complexes que de simples nombres (tableaux, structures, listes ...) ;

Une formalisation de la sémantique qui rend vraiment compte des langages évolués sans les ramener au niveau machine nécessite donc une théorie des structures d'informations ([SI1] , [SI2] , [SI3] et [SI7]) recouvrant aussi bien les structures logiques et abstraites que les structures physiques les représentant dans la mémoire d'un ordinateur (une telle étude a été menée par les auteurs de la méthode de Vienne [S36] qui se ramènent dans tous les cas à des informations arborescentes).

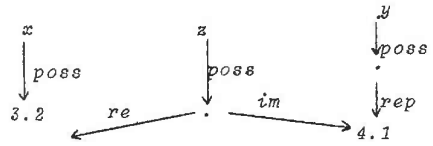
La première étape de notre travail [2] précise la notion de structure d'information : intuitivement, cette structure permet de rendre compte des caractéristiques d'une information, une information représentant un état de la mémoire d'un calculateur à un instant donné (le sens de "structure" dans "structure d'information" étant le même que dans "structure de groupe").

a) Brièvement, nous dirons qu'une information comprend un ensemble d'objets et un ensemble de relations ou accès entre ces objets ("objets" et "relations" étant pris au sens de [A] , les relations sont plus précisemment des fonctions).

Exemple 1 : Considérons le début de programme Algol 68 :

début réel x = 3.2 ; rep réel y = loc réel := 4.1 ;
compl z = (x, y) ;

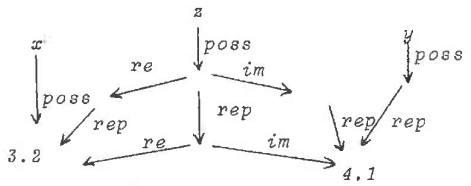
à la suite de l'élaboration de cette phrase l'information I_1 peut-être schématisée par :



("poss" pour"possède", "rep" pour"repère").

b) Le rôle d'une structure d'information est de préciser les différentes relations apparaissant dans une information de cette structure ainsi que leurs propriétés.

Exemple 2 : Si, dans l'exemple 1, on remplace *compl z = (x, y)* par *rep compl z = loc compl := (x, y)*, l'information I_2 obtenue alors est représentée par :



(un nombre complexe est une structure à 2 champs, donc un nom de complexe est un nom n de structure à 2 champs. A un tel nom on associe 2 noms repérant respectivement les champs du nombre complexe repéré par le nom n [A-2.2.3.5] .

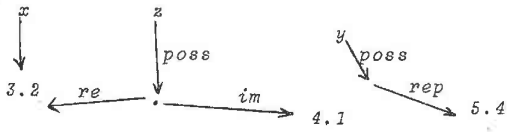
Les relations *rep*, *re*, *im* vérifient donc les propriétés :

rep re poss z = re rep poss z et *rep im poss z = im rep poss z*

qui s'expriment plus généralement en disant que *rep* commute avec *re* et avec *im* .

Cette commutativité est indépendante de l'information considérée, c'est-à-dire vraie pour toute information contenant un nom de nombre complexe. Ce sont de telles propriétés qui sont définies par la structure d'information.
Enfin, pour pouvoir non seulement ajouter des relations en cours d'élaboration, mais également en modifier certaines (dans l'exemple 1 après *compl z = (x, y)* on pourrait écrire *y := 5.4* nous définirons un ensemble de modifications élémentaires de la structure d'information qui à toute information associent une nouvelle information.

Exemple 3 : Schématiquement l'affectation y := 5.4 peut être représentée par la modification élémentaire affect (y, 5.4) et alors I_3 = affect (y, 5.4) (I_1) est schématisée par :



(en fait nous verrons que la définition d'une affectation est plus compliquée que cela en Algol 68 [6.3.2]).

1.2.2 - Calculs

La définition de la signification d'un programme par une fonction est assez éloignée de la réalité informatique puisqu'on ne peut rendre simplement compte de notions telles que : programmes qui bouclent indéfiniment , temps d'execution ... En fait, on utilise assez peu, dans cette première approche, la notion d'état de mémoire puisqu'on ne considère que l'état initial et l'état final.
Une deuxième approche de la notion de sémantique d'un programme consiste alors à considérer la suite d'états de mémoire (ou d'information) par laquelle passe le calculateur (abstrait) pendant l'exécution du programme ; une telle suite peut s'appeler calcul : c'est la démarche suivie par la méthode de Vienne pour laquelle

un état de mémoire contient en particulier le programme qui s'exécute
Parallèlement à une théorie des structures d'informations,(et s'appuyant sur elle) une théorie des calculs est donc nécessaire ([C1], [C2], [C3]) : c'est ce que nous développerons en [3.3].
Il serait plus commode, notamment pour rendre compte des calculs collateraux, de considérer un calcul, plutôt que comme une suite d'états de mémoire, comme une suite de modifications élémentaires, c'est-à-dire un mot fini ou infini sur l'ensemble $\mathcal{M}od$ des modifications élémentaires.
En reprenant l'exemple 1 ci-dessus et supposant qu'une déclaration d'identité *truc* a = b (où *truc* est un déclareur de mode) est représentée par la modification élémentaire id (a, b), nous représenterons l'élaboration de ce début de programme par le calcul :

id(*x*, *3.2*). id(*y*, *loc réel*). affect(y, *4.1*). affect(z, (x, y)

Nous mettons ainsi en évidence l'histoire de l'élaboration du programme. D'autre part, si maintenant nous interprétons le "." définissant la concaténation comme la composition des modifications élémentaire (à l'ordre près des opérandes) nous retrouvons la définition fonctionnelle de 1.2.1 (la fonction composée est appelée modification).
Cependant cette interprétation d'un calcul par une modification n'a de sens que s'il est fini.
Ainsi nous définirons la valeur sémantique d'un programme comme étant un calcul ou plus précisement un ensemble de calculs : en effet à un programme sont associées en général plusieurs élaborations possibles.
Cet ensemble sera déterminé par un système à point fixe sur un ensemble de modifications élémentaires, système défini de manière récursive par décomposition du programme en phases auxquelles sont associés des sous systèmes.

1.2.3 - Valeur d'une phrase

A toute phrase du programme on associe donc un ensemble de calculs qui en formalise l'élaboration. Mais d'autre part, une telle phrase peut fournir une valeur : c'est le cas, par exemple

d'une expression conditionnelle en Algol 60, c'est plus généralement le cas de la plupart des phrases en Algol 68.
La valeur d'une phrase P est donnée par un accès dans l'information : la définition de la sémantique doit donc associer à une telle phrase un accès de la structure d'information. Comme pour les calculs, la définition de cette association sera récursive, elle se fera par un système : le système des accès.

Exemple 4 : Dans *ent x ; y :=* F *début lire (x) ; x := x + 2 fin ;* en Algol 68) nous associerons à la proposition fermée F un accès, noté encore F, de la structure d'information défini par :

poss F = *poss x*

1.2.4 - Langage pivot

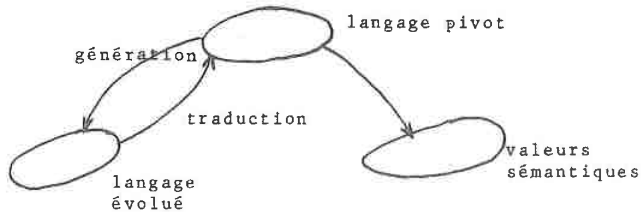
a) programme d'un langage habituel se présente sous la forme d'une chaine de caractères dont la signification n'est pas immédiate :
Par exemple pour définir le sens de la phrase Algol 60 :

DEBUT REEL X ; X := A + B FIN

Il faut en reconnaitre les différents composants pour leur associer un ensemble de calculs. En particulier l'élaboration de X := A + B commence par celle de A + B qui peut être collatérale, ce qui n'apparait pas sur la structure linéaire de la phrase.
Ainsi la définition récursive de l'ensemble de calculs ou de l'accès associé à une phrase plus que sur le texte de cette phrase porte sur sa structure syntaxique. Pour définir la sémantique d'un programme, il semble donc naturel de la représenter sous une forme plus commode qui peut être celle d'arbre (ou de ramification [T3], [T4]).

b) d'autre part, cette transformation de la représentation d'un programme permet de n'en conserver que les composants sémantiquement utiles ; ainsi un certain nombre de symboles syntaxiques n'apparaitront pas dans le programme transformé (par exemple, le symbole ALLERA en Algol 60 est redondant et la phrase "ALLERA ETI" peut fort bien être remplacée par "ETI"). Il est également possible au cours de cette transformation de supprimer certaines constructions du langage en les représentant par d'autres (plus élémentaires ou dont le sens est plus simple à définir). Ainsi, par exemple, les boucles POUR en Algol 60 peuvent être représentées à l'aide de tests et de sauts ; de même une procédure fonction à n paramètres peut être remplacée par une procédure sans résultat à n + 1 paramètres.

c) Ainsi la plupart des méthodes de formalisation de la sémantique d'un langage évolué commencent par définir un nouveau langage appelé langage pivot (ou parfois langage noyau car il comporte des constructions plus simples que celles du langage initial) et une transformation (traduction) permettant de représenter les programmes du langage initial (ou concret) en des programmes pivots. La sémantique d'un programme est alors définie sur le langage pivot associé, ce qui permet en particulier de ne traiter que les constructions essentielles du langage.
On peut schématiser cette démarche par :



Le langage pivot est à la base de la définition d'un langage évolué, il permet de définir :

- la structure linéaire des programmes
- la sémantique du langage

C'est la démarche proposée par MAC CARTHY ([S37], [S39]) et suivie par les auteurs de la méthode de Vienne ([S1], [S2], [S35]) qui opposent la syntaxe concrête (représentation linéaire) à la syntaxe abstraite (langage pivot dans lequel les programmes sont des arbres). C'est aussi la démarche suivie implicitement dans la

définition d'Algol 68 [A] où la sémantique s'appuie sur la structure syntaxique des différentes phrases du langage strict. On retrouve une démarche similaire à celle proposée par certains linguistes (N. CHOMSKY notamment [T1]) qui proposent, pour définir la sémantique d'une langue naturelle, de distinguer les notions de structure de surface et de structure profonde.
Dans la suite de notre étude nous utiliserons essentiellement cette notion de langage pivot.

d) <u>remarque</u> :

Si la structure d'arbre est la plus couramment utilisée pour représenter un programme du langage pivot, il faut remarquer que la ramification créée par une C-grammaire (ou grammaire à "contexte libre" [T2] peut être insuffisante : elle ne permet pas de rendre compte de notions telles que celle de <u>type</u> d'une phrase (<u>mode</u> en Algol 68).

Dans notre formalisation, le programme pivot associé à un programme Algol 68 sera grossièrement, la ramification engendrée par la <u>double grammaire</u> d'Algol 68 qui contient notamment des renseignements sur les modes.

Une autre manière d'obtenir une structure serait de lui associer des <u>attributs</u> (cette notion d'attribut a été proposée par KNUTH [S29]).

Rapidement on peut dire que les attributs permettent de sélectionner certaines ramifications parmi celles engendrées par une C-grammaires et de compléter les "étiquettes" des noeuds de la ramification engendrée par divers renseignements : c'est aussi le rôle des doubles grammaires.

Remarquons enfin que si la forme arborescente est la plus courante pour un programme pivot, certaines méthodes lui prèfèrent une forme linéaire (VAN WIJNGAARDEN [S55]).

1.2.5 - <u>Conclusion</u>

En résumé, pour nous, formaliser la sémantique d'un langage de programmation consiste à définir :

a) - une structure d'information

b) - un langage pivot (avec un processus de traduction dans ce langage).

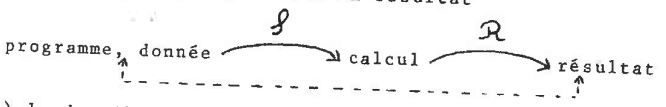
c) - l'association à chaque phrase du langage pivot :
   . d'un système à point fixe définissant un ensemble de calculs
   . d'un système définissant un accès de la structure d'information.

1.3 - Comparaison avec d'autres méthodes

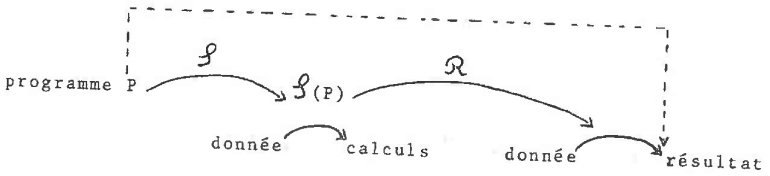
1.3.1 - Fonction sémantique

Une fonction sémantique associe à un programme sa valeur sémantique. De manière schématique, on peut grouper en deux familles les différentes approches de la sémantique d'un langage de programmation selon que l'on définit la valeur associée à un couple (programme, donnée) ou simplement à un programme.

a) le premier type correspond à un point de vue interprétatif : La fonction sémantique $\mathcal{S}$ associe à un couple (programme, donnée) un calcul qui peut posséder un résultat



b) le deuxième type correspond à un point de vue compilatoire (ou fonctionnel) :
La fonction sémantique $\mathcal{S}$ associe à un programme P une fonction $\mathcal{S}(P)$ définie dans l'ensemble des données, à valeurs dans l'ensemble des calculs. On peut adjoindre de manière naturelle à $\mathcal{S}(P)$ une fonction $\mathcal{R}(\mathcal{S}(P))$ associant un résultat (d'un calcul) à une donnée :



C'est en particulier la démarche que nous suivons, la fonction sémantique étant définie par le système à point fixe caractérisant l'ensemble de calculs associé à P ; le passage de $\mathcal{S}(P)$ à $\mathcal{R}(\mathcal{S}(P))$ consiste en l'interprétation de la concaténation par la composition [1.2.1] .

Les méthodes de définition de ces fonctions sémantiques peuvent être très diverses, elles sont caractéristiques de la formalisation choisie. En particulier ces fonctions doivent être calculables ; on retrouve donc dans la formalisation de la sémantique d'un langage, les différentes approches des logiciens de la notion de calculabilité (CHURCH [L1] , MARKOV [L7], TURING [L10]).

Nous allons brièvement présenter quelques unes de ces méthodes.

1.3.2 - Méthodes Interprétatives

a) Utilisation de la notion de machine abstraite

Une machine abstraite ou automate est caractérisée essentiellement par :

- un ensemble d'états
- une fonction de transition d'un état à un autre.

L'ancêtre de telles machines est celle de TURING où la mémoire est décomposée en deux parties : mémoire interne et mémoire externe (ruban sur lequel on trouve données et résultats). Un état, au sens où nous l'entendons ici, est parfois appelé description instantanée.

Une version très élaborée d'un tel automate est l'interpréte abstrait de Vienne ([S19] , [S36] , [S44]) où données, résultats et programme font partie des états. Ce travail considérable à permis de définir formellement PL/1 et Algol 60.

Un certain nombre de formalisation de la sémantique (méthode de Vienne, méthode proposée par ELGOT [S18]) représentent une fonction sémantique de type a) par une machine abstraite : elles définissent ainsi un interpréteur.

Remarquons enfin, que de telles machines sont parfaitement adaptées à rendre compte des notions impératives d'un programme (et en particulier des sauts).

b) Utilisation des algorithmes de MARKOV

Brièvement un algorithme de MARKOV [L7] est caractérisé par :

- un alphabet V
- un ensemble de règles de production R_i de la forme $\alpha_i \rightarrow \beta_i$ (α_i, $\beta_i \in V^*$) totalement ordonné.

Le résultat de l'application de la règle $\alpha_i \to \beta_i$ au mot μ est le mot μ' déduit de μ en y remplaçant la première occurence de α_i par β_i.

On peut alors définir la notion de calcul.

VAN WIJNGAARDEN [S56] , CARACCIOLO [S13] , De BAKKER [S3], WIRTH et WEBER [S58] ont utilisé et enrichi cette notion pour définir la sémantique de certains langages (notamment Algol 60).

Remarque :

De telles méthodes, si elles permettent d'exprimer complétement la sémantique d'un langage évolué, semblent permettre assez difficilement la résolution de problèmes tels que preuves de programmes, équivalences de programmes, preuves de compilateurs ...

C'est certainement l'une des raisons pour lesquelles, parallèlement au développement de telles méthodes, certains chercheurs on introduit des objets plus idéaux, plus mathématiques et mieux adaptés à l'étude des concepts fondamentaux de la programmation tels que les schémas de programmes introduits en particulier par IANOV [S27] . La définition de la sémantique d'un tel schéma peut être qualifiée d'interprétative.

1.3.3 - Méthodes compilatoires

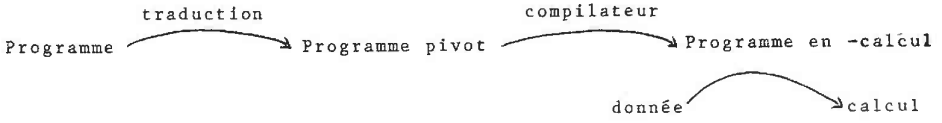
De nombreux auteurs ont utilisé des objets plus "fonctionnels" que les précédents :

- λ-calcul de CHURCH
- système d'équations.

a) λ-calcul

Le λ-calcul est plus adapté au traitement des expressions que des instructions, LANDIN l'a étendu pour rendre compte des notions impératives d'un programme. Il joue le rôle d'un langage machine universel et la définition d'une machine abstraite représentant une fonction sémantique de type b), qui associe à tout programme

(pivot) une λ-expression (généralisée) est analogue à la définition d'un compilateur.



Ainsi dans [S32] LANDIN définit formellement les deux premières étapes pour Algol 60, dans [S31] il associe un sens aux λ-expressions en introduisant une manière de les évaluer.

Dans cette direction on peut noter les travaux de STRACHEY S50 et de BÖHM (dans la définition du langage CUCH [S10] , [S11]).

Remarque :

D'autres méthodes moins dormelles ont tenté de définir la sémantique d'un langage en en donnant effectivement un compilateur (en particulier GARWIK [S21]). Elles ne semblent pas avoir grand succès par manque d'un langage machine universel (malgré certains efforts dans ce sens, voir en particulier le LMU de NOLIN [S43]) de plus ce genre de définition, très orienté vers la machine, risque d'être rebutant pour l'utilisateur.

Notons enfin que la méthode des attributs ([S29] , [S57] , [1.2.4d]) peut être envisagée comme étant fondée sur un procédé de compilation.

b) Systèmes d'équations et point de vue fonctionnel

MAC CARTHY, l'un des premiers, suggère, dans la définition d'un sous-ensemble d'Algol [S39] d'associer à un programme (pivot) une fonction récursive ; la notion de système à point fixe développée en particulier par SCOTT, De BAKKER ([PF7] , [S47]) permet de formaliser cette association.

Un certain nombre de travaux ont été effectués dans cette direction qui, en définissant la sémantique de langages de programmation

très simples étudient les concepts fondamentaux de la programmation (tels que la notion d'équivalence) et construisent des méthodes de preuves. Ainsi les études de MANNA [S40] , VUILLEMIN [S51] , [S52] , COURCELLE [S16] , CADIOU [S12] , De BAKKER, De ROEVER [S6], [S7] ou BLICKLE [S8] , [S9], pour n'en citer que quelques uns, définissent la valeur sémantique d'un programme comme étant une fonction, une relation ou un calcul ; l'association d'une telle fonction à un programme se fait par une fonction sémantique du type b) de [1.3.1] définie par un système à point fixe.

Le travail présenté ici utilise essentiellement cette notion de système à point fixe. Il est cependant éloigné des études mentionnées ci-dessus puique essentiellement descriptif : Il formalise la sémantique d'un langage de programmation complexe mais en contre partie il n'aborde pas les problèmes de preuve, d'équivalence etc ...

1.3.4 - Méthodes axiomatiques

a) Définition directe d'une sémantique par des axiomes

HOARE [S22] , [S23] a proposé une définition axiomatique de la sémantique d'un langage (au sens où les définitions de groupe, d'espace vectoriel ... sont axiomatiques) par opposition aux définitions précédentes (explicites).

L'idée est la suivante : si on associe des axiomes à un langage de programmation, axiomes qui caractérisent chaque objet et construction du langage (entiers, réels, procédures, expressions ...), on associe par là même un sens à tout programme de ce langage.

Un des intérêts de cette méthode est de laisser facilement certaines libertés à l'implémentation qui peut imposer à son tour certains axiomes spécifiques. On peut cependant se demander si une telle méthode est effectivement applicable par des langages complexes ; aucune réponse n'est jusqu'à présent apportée à cette question, les différents auteurs HOARE, MANNA et PNUELI [S42] , De BAKKER [S7] , De ROEVER [S6] se restreignant aux problèmes de preuves de programmes.

Enfin, certains inconvénients semblent difficilement résorbables tels que la difficulté de rendre compte des programmes qui bouclent.

b) Equivalence de programmes

C'est une idée déjà ancienne qui consiste à négliger la nature des valeurs sémantiques d'un langage et de se contenter de savoir quand deux programmes donnés sont équivalents.

IGARASHI [S28] en particulier définit un certain nombre de conditions d'équivalences. Ces équivalences seront exprimées sous forme d'axiomes d'un certain système formel. L'équivalence sémantique sera la plus petite relation satisfaisant aux conditions imposées ; les règles d'inférence du système formel traduiront en particulier la symétrie et la transitivité de cette relation.

L'inconvénient majeur de cette méthode est la nécessité d'imposer un grand nombre d'axiomes ce qui rend la définition peu maniable.

1.3.5 - Algol 68

Il existe d'autres approches de la sémantique d'un langage évolué, citons en particulier la définition d'Algol 68 donné par [A].

Cette définition, sans être formelle au sens où nous l'entendons montre un effort considérable, vers la précision et la rigueur. Comme le remarque De BAKKER dans [S5] cette méthode de description devrait être applicable à d'autres langages, à condition que l'on puisse séparer les concepts servant à la définition des concepts définis. Cette précision de la définition d'Algol 68, la richesse des notions rencontrées dans ce langage, son universalité nous ont conduits à le choisir pour donner un exemple de notre formalisation.

Dans la suite, comme nous l'avons déjà signalé, nous précisons les notions de structure d'information [2] et de calcul [3] ; puis en [4] sera définie la structure d'information d'Algol 68, en [5] on introduit le langage pivot Algol 68$_0$ et en [6] on définit l'ensemble de calculs associé à un programme du langage pivot.

2

Structures d'information

2. STRUCTURES D'INFORMATION

2.1 Introduction :

L'une des notions fondamentales sur laquelle repose notre formalisation de la sémantique d'un langage est celle de structure d'information ; brièvement il s'agit de rendre compte des "objets traités par un calculateur lors de l'élaboration d'un programme".

La formalisation présentée dans ce chapitre se révèlera applicable à d'autres notions informatique que la sémantique des langages ; citons par exemple des domaines tels que banque de données, compilation, systèmes d'exploitation.

On se contentera ici d'une théorie réduite, limitée essentiellement au problème de la sémantique, voir [SI7] pour une formalisation plus complète.

Une première étape de notre travail [2.2] consiste à étudier d'un point de vue fonctionnel (i.e. avec des outils ensemblistes) la notion d'objet composé ou donnée (ainsi une liste, l'état d'un calculateur à un moment donné de l'élaboration d'un programme sont des exemples de donnée).
On présente ensuite une formalisation logique de ces notions [2.3,2.4] (l'analogue formel d'une donnée se nomme alors information) ; cette formalisation permet de donner un sens précis aux termes :

- accéder à un élément d'une information
- modifier une information [2.6]
- représenter une information par une autre (ce dernier point ne sera pas abordé ici, il est traité dans [SI2], [SI6] et [SI7]).

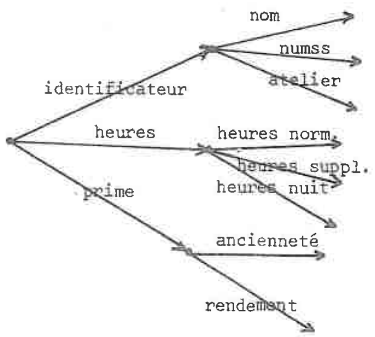
2.2 La notion de donnée :

2.2.1 Exemples :

Les objets traités en informatique sont, en général, formés d'objets élémentaires liés par certaines relations ; un tel objet composé peut s'appeler "donnée" (au même sens que dans banques de données).
Pour définir de manière précise la notion de donnée, nous essayons tout d'abord de comprendre sur quelques exemples ce qui la caractérise.

a) Un tableau de nombres réels est parfois considéré comme un groupement de nombres réels. En réalité, la chose importante est la manière dont on a accès à ces nombres réels : chacun d'eux est déterminé par des indices. Un tableau de nombres réels, à deux dimensions, par exemple, peut donc être considéré comme une fonction qui associe à un couple d'entiers, dans un certain domaine de définition, un nombre réel.

b) Dans le fichier salaire d'une entreprise, chaque article peut être schématisé de la manière suivante :

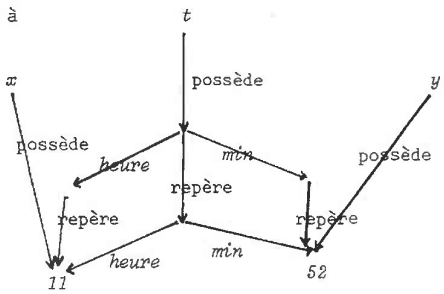


nom, numss, atelier ..., identification, ..., rendement sont des fonctions qui permettent l'accès à une chaîne de caractères, à un entier ou à un autre élément qui ne sert que de relai, de repère.

c) Considérons les déclarations suivantes du langage Algol 68 :

ent $x = 11$; *ent* $y = 52$; *struct*(*ent heure*, *ent min*) $t := (x,y)$.

L'identificateur x (resp. y) donne accès à (on dit possède) l'entier *11* (resp. *52*). L'identificateur t possède un nom n qui repère une valeur structurée v à deux champs sélectionnés respectivement par heure et min. (Après une nouvelle affectation ce nom pourra repérer une autre valeur). [A] précise de plus qu'à n sont associés, par l'intermédiaire des sélecteurs heure et min, deux autres noms repérant respectivement les champs de v. L'état de la mémoire de la "machine Algol 68" après ces déclarations est schématisé sur la figure. possède, repère, *heure*, *min* sont ici encore des fonctions permettant des accès.



Une donnée apparaît donc, sur ces exemples, comme étant constituée d'objets élémentaires (valeurs élémentaires et "relais", appartenant à des ensembles donnés) liés par des fonctions d'accès (fonctions à une ou plusieurs variables). L'opération essentielle sur ces fonctions est la composition qui à partir d'accès élémentaires construit des accès plus élaborés. C'est ce que traduit la formalisation suivante :

2.2.2 Formalisation fonctionnelle des objets composés :

2.2.2.1 Définition :

Une donnée est un m + 1 uple $(E_1,..,E_m,\bar{L})$ où E_i $(i = 1,...,m)$ est un ensemble d'objets élémentaires ; $\bar{L}$ est un ensemble de fonctions (partielles) définies dans des ensembles de la forme $E_{j1} \times ... \times E_{jq}$ $(q \geqslant 0$ et $1 \leqslant j_i \leqslant m)$, à valeurs dans l'un des E_i $(1 \leqslant i \leqslant m)$; une telle fonction s'appellera accès élémentaire à q variables, une fonction à 0 variable étant confondue avec sa valeur.

2.2.2.2 Exemples :

a) liste :

Une liste sur un ensemble E est une suite finie d'éléments de E . Pour préciser cette notion de suite, et en particulier pour exprimer les accès aux éléments d'une telle suite ; on introduit un ensemble F totalement ordonné (ensemble des "places" dans la suite étudiée) dont le premier élément $\bar{t}$ s'appelle tête de liste et le dernier $\bar{d}$ est la queue de liste. L'ordre total se traduit par la donnée d'une fonction de succession dans F , noté $\bar{s}$, qui est une bijection de $F - \{\bar{d}\}$ dans $F - \{\bar{t}\}$.

Alors tout élément de F est l'image de $\bar{t}$ par $\bar{s}^i$ ($= \underbrace{\bar{s} \circ \bar{s} \ \ldots \circ \bar{s}}_{i \text{ fois}}$) pour un certain i .

De plus une application $\bar{v}$ assigne à chaque élément de F une valeur dans E. Ainsi une liste est le triplet $(F,E,\bar{L})$ où $\bar{L} = \{\bar{s}, \bar{v}, \bar{t}\}$ et

- $\bar{s}$ est une bijection de F - $\{\bar{d}\}$ sur F - $\{\bar{t}\}$ vérifiant :

$$(\forall x \in F)\ (\exists i \geqslant 0)\quad (x = \bar{s}^i(\bar{t}))$$

- $\bar{v}$ est une fonction (totale) de F dans E

- $\bar{t}$ est un élément de F (fonction à 0-variables)

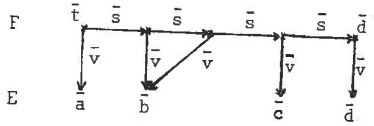


Schéma de la liste $\bar{a}\bar{b}\bar{b}\bar{c}\bar{d}$

b) mémoire :

La notion de mémoire adressable peut être formalisée de la manière suivante :

une mémoire adressable est un quadruplet (M,U,A,L) où :

- M est un ensemble de mots
- U est l'ensemble des valeurs susceptibles d'être contenues dans ces mots (par exemple ensemble des suites de 32 chiffres binaires)
- A est un sous ensemble de U : ensemble des adresses (par exemple éléments de U dont les 15 premiers chiffres binaires sont 0)
- $\bar{L} = \{\bar{C}, \overline{mem}\} \cup U \cup \bar{L}'$ ensemble des fonctions élémentaires où :

$\bar{C}$ est la fonction contenu définie sur M à valeurs dans U

$\overline{mem}$ est la fonction adressage définie sur A à valeurs dans M (injective).

On n'a pas directement accès aux mots mais aux valeurs de U , autrement dit parmi les fonctions d'accès élémentaires se trouvent les éléments de U comme fonctions à 0 variable.

$\bar{L}'$ est un ensemble d'opérations dans U, c'est-à-dire de fonctions à une ou plusieurs variables de U à valeurs dans U, par exemple une addition ou une fonction successeur : $u \longrightarrow u + 1$.

c) [2.2.1 c)] est un exemple d'une donnée Algol 68.

Plus généralement [A-2.2] permet de définir des données Algol 68 au sens précédent ; une telle donnée sera "l'état d'un calculateur hypothétique" à un

instant de l'élaboration d'un programme (en effet, toute relation introduite en [A-2.2] qui n'est pas fonctionnelle s'exprime comme une fonction à 2 arguments à valeurs dans vrai,faux).

Remarque : Une donnée Algol 68 contient les accès nécessaires à une définition agréable de la sémantique du langage ; par contre un calculateur ne contient qu'un nombre restreint d'accès (b) ci-dessus). On est donc conduit à se poser le problème de la représentation d'une donnée par une autre, cette autre étant bien souvent une mémoire (c'est le problème de l'implémentation) (voir [SI2] et [SI7]).

Ces trois exemples seront fréquemment utilisés dans la suite du chapitre. Cependant dans le cas d'Algol 68 on devra souvent se contenter de remarques ponctuelles à cause de la grande complexité des objets, le but du travail étant d'en effectuer une étude plus complète.

Notations : les objets élémentaires et les fonctions d'accès d'une donnée seront surlignés ce qui permet de les distinguer des symboles fonctionnels introduits par la suite [2.3] .

2.2.2.3 Remarque :

Il est possible de considérer des fonctions totales à la place des fonctions partielles en introduisant un élément $\perp_i$ dans E_i qui traduit la "non définition". Ainsi pour une liste à 4 éléments on petu adjoindre $\perp$ à F et poser :

$$\bar{s}(\perp) = \perp$$

$$\bar{s}(d) = \perp \ .$$

Plus généralement le prolongement d'une fonction partielle g en une fonction totale $\check{g}$ se fait de la manière suivante :

pour $x \in E$: $\check{g}(x) =$ si $\bar{g}(x)$ est définie alors $\bar{g}(x)$ sinon $\perp$

et $\check{g}(\perp) = \perp$.

2.2.3 Structure de donnée :

Une donnée est un objet "statique", elle n'a d'intérêt que dans la mesure où on peut lui appliquer des traitements.

. Pour les listes, par exemple, on veut pouvoir exprimer certaines transformations telles que :

- modification de la valeur d'un élément
- adjonction d'un élément en tête de liste
- suppression d'un élément etc...

Une telle modification transforme la liste initiale en une nouvelle liste ;

de plus la description de cette transformation est indépendante de la liste de départ. Ainsi, avec la définition précèdente de liste, modifier la valeur d'un élément revient à changer la fonction $\bar{v}$; ajouter ou supprimer un élément revient à changer l'ensemble F et les fonctions $\bar{v}$ et $\bar{s}$.

· Pour une donnée Algol 68 de la forme de [2.2.1 c)] on veut pouvoir définir ce qu'est l'affectation de (12,1) au nom possédé par t ; ce qu'est l'affectation de 3 au champ *min* du nom possédé par t.

De telles modifications consistent à changer les fonctions repère (resp. *min*).

Dans les transformations précédentes ce qui importe est le type de la donnée beaucoup plus que la donnée elle-même. Un type est, de manière intuitive, une classe de données ayant même entier m et des ensembles $\bar{L}$ "semblables".

Les deux notions fondamentales de type d'une donnée et de modification définie sur des données d'un certain type se regroupent dans la notion de structure de données ; structure a ici le même sens que dans structure de groupe : c'est le cadre dans lequel on peut exprimer certaines propriétés caractérisant certains objets. C'est pour les structures de données que se pose le problème de la représentation notamment en mémoire.
Cependant si la notion de structure de donnée semble naturelle, elle ne permet pas d'exprimer agréablement ce qu'est une modification : un "changement de fonction d'accès" n'est pas simple à formaliser. Aussi pour préciser ces notions de type, de modification, de structure (en exprimant en particulier ce que signifie la phrase "des ensembles $\bar{L}$ semblables"), on est conduit à une définition plus formelle, remplaçant les fonctions par des symboles fonctionnels, la notion de schéma fonctionnel formalisant la composition des fonctions.

2.3 Information et Structure d'information :

2.3.1 Introduction :

Dans l'approche précédente une donnée $(E_1, E_2, \ldots, E_m, \bar{L})$ est un objet "concret". Par opposition nous considérerons une information comme étant un objet "idéal" qui permet des affirmations ; en bref, une information sera un ensemble de théorèmes d'une certaine forme. De plus une information pourra être réalisée concrêtement (interprétée, au sens de la logique) par une donnée.

Exemples :

a) Dans une information de type liste, si s (resp. v, t) représente le symbole fonctionnel unaire (resp. unaire, 0-aire) pouvant être interprété comme la fonction $\bar{s}$ (resp. $\bar{v}$, $\bar{t}$) et si ω est le symbole fonctionnel 0-aire interprété comme "non défini", la liste schématisée en [2.2.2.2 b)] est caractérisé par les théorèmes suivants :

$$(*)\quad \begin{cases} vt \equiv a & (\equiv \text{ représente l'égalité formelle}) \\ vst \equiv b & \\ vs^{(2)}t \equiv b & \\ vs^{(3)}t \equiv c & \\ vs^{(4)}t \equiv d & \end{cases}$$

De plus, on exprime que s est une injection par l'ensemble des théorèmes suivants :

pour $i \neq j$; $i,j \in \mathbb{N}$ on a $s^{(i)}t \equiv s^{(j)}t \supset s^{(i)}t \equiv \omega$

Enfin on précise que $s^{(4)}t$ est le dernier élément par :

$$s^{(5)}t \equiv \omega \quad .$$

b) Dans une information de type mémoire, on exprime que le symbole fonctionnel *mem* doit être interprété comme une fonction injective ($\overline{\text{mem}}$) par :

$mem\ x \equiv mem\ y \supset x \equiv y$ pour tous symboles fonctionnels x et y .

c) Considérons les déclarations Algol 68 suivantes :

réel $x = 3.2$; *rep réel* y = *loc réel* := 4.1 ; *compl* $z = (x,y)$;

(*) notation : pour tout symbole fonctionnel f, $f^{(n)}$ représente la concaténation de n occurences de f $(\underbrace{f\ f \ldots f}_{\text{n fois}})$.

l'information représentant l'état de la mémoire à la suite de leur élaboration contient les théorèmes :

$$\begin{cases} & poss\ x \equiv poss\ 3.2 \\ rep & poss\ y \equiv poss\ 4.1 \\ re & poss\ z \equiv poss\ 3.2 \\ im & poss\ z \equiv poss\ 4.1 \end{cases}$$

où *poss* (resp. *rep*, *re*, *im*) est un symbole fonctionnel interprété comme la fonction possède (resp. repère, partie réelle, partie imaginaire).

Un type d'information est donc d'abord un cadre pour ces théorèmes. Ce sera un système formel $\mathcal{F}$ = (Alph, F, X, R) exprimant les propriétés générales communes à toutes les informations du type (Alph est l'alphabet de $\mathcal{F}$, F l'ensemble des formules, X l'ensemble des axiomes et R l'ensemble des règles d'inférence). Pour obtenir une structure d'information, nous ajouterons ultérieurement à un ensemble $\mathcal{M}od$ de modifications élémentaires.

2.3.2 Systèmes formels associés aux structures d'information :

Les formules du système formel sont construites en plusieurs étapes. On définit successivement :

- un ensemble L de symboles fonctionnels (ou accès)
- un ensemble S de schémas fonctionnels sur L et un sous ensemble A de S ≡ S (ensemble des formules atomiques)
- l'ensemble F des formules du système, solution minimale de l'équation

$$F = A \cup \neg F \cup (F \supset F) \quad .$$

2.3.2.1 Alphabet :

L'alphabet contient :

- un ensemble L de symboles fonctionnels
- le symbole ≡
- les symboles $\neg$, $\supset$, (,) .

Pour interpréter les symboles fonctionnels, on doit introduire m ensembles $E_1,\ldots,E_m$ et interpréter chaque symbole comme une fonction de $Ej_1 \times \ldots \times Ej_q$ dans E_k .

Plus généralement, et pour éviter l'introduction de trop nombreux symboles, on interprétera un symbole fonctionnel f comme une fonction de

$\bigcup_{(j_1,\ldots,j_q) \in I_f} Ej_1 \times \ldots \times Ej_q$ dans E_k où I_f est un sous-ensemble de $\{1,2,\ldots,m\}^q$.

On est donc amené à se donner un entier m et à associer à chaque élément f de L un entier $q \geqslant 0$ et un ensemble pl(f) de q+1 - uples $(j_1,\ldots,j_q,k)$ avec $1 \leqslant j_i \leqslant m$ pour $1 \leqslant i \leqslant q$. pl(f) sera appelé profil de f .

Définition : On appelle base symbolique tout triplet (L,m,pl) où

- L est un ensemble de symboles fonctionnels
- m est un entier : la puissance de la base
- pl est une application de L dans $\bigcup_{q \geqslant 0} \mathcal{P}(\{1,2,\ldots,m\}^q)$ qui associe à tout symbole fonctionnel son profil.

Enfin si les éléments de pl(f) sont des q+1 - uples, on dit que f est un symbole q-aire. Nous noterons L_q l'ensemble des symboles q-aires ($q \geqslant 0$) l'alphabet est donc caractérisé par la base symbolique (L,m,pl).

Exemple 1 : Pour les listes, m = 2 et L contient :

- s (qui sera interprété comme la fonction successeur $\bar{s}$) pl(s) = (1,1) (*)
- v (qui sera interprété comme la fonction valeur $\bar{v}$) pl(v) = (1,2) .
- t (interprété comme la tête t , fonction 0-aire) pl(t) = (1)
- ω (interprété comme l'élément non défini $\perp$) pl(ω) = (1)
- un ensemble V de symboles, interprétés comme les valeurs de la liste (éléments de E) ; le profil de chacun d'eux est (2).

Exemple 2 : Nous verrons [4] que l'alphabet du système formel Algol 68 contient entre autres :
- des symboles rep^i ($3 \leqslant i \leqslant 10$) qui s'interpréteront comme la fonction repère. Cette fonction unaire est à valeurs dans l'ensemble des valeurs multiples, des valeurs structurées, des valeurs simples etc... ce qui justifie la présence de plusieurs symboles *rep* .
- des symboles $poss^i$ ($3 \leqslant i \leqslant 10$) qui s'interpréteront comme la fonction possède
- des symboles ch^i_x ($3 \leqslant i \leqslant 10$) qui s'interpréteront comme la fonction sélecteur d'une valeur structurée associée au champ x .

Il faudra exprimer [A-2.2.3.5 b)] que "si un nom N repère une valeur structurée, tout champ x de cette valeur est repéré par un nom défini de façon unique

(*) lorsque pl(f) ne contient qu'un q+1 - uple, on convient d'omettre les accolades, ainsi (1,1) représente {(1,1)}.

à partir de N et de x".

Aussi sera-t-on conduit à "définir" ch_x sur les noms.

Un nom pourra donc être accédé par l'intermédiaire d'un symbole ch_x^i soit à partir d'un autre nom, soit à partir d'une valeur structurée.

Aussi, par exemple, le profil de ch_x^3 sera-t-il de la forme

$$pl(ch_x^3) = \{(3,3)\},\{(4,3)\}$$

(E_3 est l'ensemble des (exemplaires de) noms, E_4 l'ensemble des (exemplaires de) valeurs structurées).

Pour alléger les notations, nous négligerons les indices "i" dans les exemples concernant Algol 68

- un symbole *valeur* : tout objet interne Algol 68 est un exemplaire de valeur [A-2.2.1] , *valeur* est interprétable comme la fonction associant sa valeur à tout exemplaire

- un symbole *mode* caractérisant le mode d'un exemplaire de valeur

- un symbole *moins* formalisant la soustraction.

Définition 2 : On nomme interprétation d'une base symbolique (L,m,pl) tout m+1-uple $(E_1,\ldots,E_m,r)$ où les E_i sont des ensembles et r une application associant à tout $f \in L$ tel que $pl(f) = \{(j_1,\ldots,j_q,k) \mid (j_1,\ldots,j_q) \in I_f\}$ une application de $\bigcup_{(j_1,\ldots,j_q)\in I_f} Ej_1 \times \ldots \times Ej_q$ dans E_k .

2.3.2.2 Schémas fonctionnels du système :

Rappelons que pour tout ensemble L de symboles fonctionnels à chacun desquels est associé un entier q (son nombre d'arguments), on sait définir des schémas fonctionnels : leur ensemble est engendré par la grammaire : $\mathcal{G} = (L,\{S\},::=,S)$ de règles :

$S ::= f \in S^{(q)}$ pour tout f à q arguments.

On sait qu'un schéma fonctionnel s'interprète en termes de composition d'applications.

Ici nous considérons seulement certains schémas fonctionnels, ceux qui sont obtenus en tenant compte des profils dans la composition.

On ne tient pas, par exemple, à considérer des schémas tels que :

- $s\ v\ t$ dans la structure de la liste
- *poss mode x* en Algol 68.

Définition 3 : On appelle schémas fonctionnels compatibles avec (L,m,pl) les éléments du langage S sur L défini par : $S = \bigcup_{k=1}^{m} S_k$

où $(S_1,\ldots,S_m)$ est la solution unique du système Σ :

$$S_k = \bigcup_{q\geqslant 0} \{f\, Sj_1\, Sj_2 \ldots Sj_q \mid f \in L \text{ tel que } (j_1,j_2,\ldots,j_q,k) \in pl(f)\} .$$

Théorème 1 : Pour tout schéma fonctionnel $u \in S$, il existe $k \in \{1,\ldots,m\}$, $f \in L$ et $u_{j_1} \in Sj_1, \ldots, u_{j_q} \in Sj_q$ uniques tels que

$$u = f\, u_{j_1} \ldots u_{j_q} \quad \text{et} \quad (j_1,\ldots,j_q,k) \in pl(f) .$$

Résulte de la non ambiguïté de la grammaire $\mathcal{G}$.

Ce théorème évite la présence de parenthèses et de virgules dans l'écriture d'un schéma fonctionnel, cependant si, pour des raisons de clarté, il nous arrive d'en introduire nous le signalerons explicitement.

Théorème 2 : Soit $u\ a\ v$ un élément de S_k $(1 \leqslant k \leqslant m)$ tel que $a \in L_o$ (ensemble des symboles 0-aires) et $pl(a) = (j)$ $(1 \leqslant j \leqslant m)$.

Alors, pour tout $w \in S_j$, $u\ w\ v \in S_k$.

Théorème 3 : Soit $(E_1,\ldots,E_m,r)$ une interprétation de la base symbolique (L,m,pl). r détermine une application unique $\hat{r}$ de S dans $E = \bigcup_{1\leqslant i\leqslant m} E_i$ telle que pour tout $f \in L$ vérifiant $(j_1,\ldots,j_q,k) \in pl(f)$ et pour tout $(u_1,\ldots,u_q) \in Sj_1 \times \ldots \times Sj_q$ on ait :

$$\hat{r}(f\, u_1,\ldots,u_q) = r(f)\ (\hat{r}(u_1),\ldots,\hat{r}(u_q)) .$$

Les démonstrations de ces deux théorèmes sont immédiates et analogues aux démonstrations des théorèmes correspondants dans [L5] .

Exemple 1 : L'ensemble S des schémas fonctionnels de la structure de liste sur V est $S = S_1 \cup S_2$ où S_1 et S_2 forment la solution minimale du système à point fixe :

$$\begin{cases} S_1 = s\ S_1 \cup \{t,\omega\} \\ S_2 = v\ S_1 \cup V \end{cases}$$

Il s'agit de $S_1 = \{s^{(i)}t \mid i \geqslant 0\} \cup \{s^{(i)}\omega \mid i \geqslant 0\}$

$S_2 = \{vs^{(i)}t \mid i \geqslant 0\} \cup \{vs^{(i)}\omega \mid i \geqslant 0\} \cup V$.

Exemple 2 :

- *moins valeur poss x valeur poss y*

- *mode rep ch$_x$ poss y*

sont des exemples de schémas fonctionnels Algol 68.

2.3.2.3 Formules atomiques :

Avec les notations précédentes, l'ensemble A des formules atomiques est défini par le système :

$$A = \bigcup_{1 \leqslant k \leqslant m} S_k \equiv S_k$$ (on ne compare des schémas fonctionnels f et f' avec $pl(f) \ni (j_1,\ldots,{}^{j}q_1,k)$ et $pl(f') \ni (j'_1,\ldots,j'q_1,k')$ que si $k = k'$).

Exemple 1 :

Pour les listes on ne veut pas comparer une place $(s^{(i)}t)$ et une valeur $(vs^{(j)}t)$. Aussi a-t-on :

$$A = S_1 \equiv S_1 \cup S_2 \equiv S_2 \quad .$$

Exemple 2 :

On ne s'intéresse pas, en Algol 68, à des formules telles que *mode* u ≡ *poss 3.5* .

Aussi l'un des sous ensembles de S sera-t-il :

MODE ≡ MODE où MODE est l'ensemble des schémas fonctionnels interprétables comme un mode Algol 68.

2.3.2.4 Formules :

L'ensemble F est la solution minimale du système à point fixe

$$F = A \cup \neg F \cup (F \supset F) \ .$$

Il revient au même de dire :

1) toute formule atomique est une formule

2) Si φ est une formule, $\neg\varphi$ est également une formule

3) Si φ et ψ sont des formules, $(\varphi \supset \psi)$ est une formule

4) Toute formule est obtenue par application des 3 règles précédentes.

Afin d'abréger l'écriture de certaines formules nous introduirons les connecteurs logiques $\vee$, $\wedge$ définis par :

$(\varphi \vee \psi)$ remplace $(\neg\varphi \supset \psi)$

$(\varphi \wedge \psi)$ remplace $\neg(\neg\varphi \supset \psi)$

et nous adopterons les conventions de suppression de parenthèses définies par

les règles suivantes :

a) l'ordre de priorité des connecteurs logiques de la structure est le suivant : $\neg$, $\wedge$, $\vee$, $\supset$.

Exemple : - $((\varphi \wedge \psi) \vee (\varphi_1 \wedge \psi_1)$ sera remplacé par $\varphi \wedge \psi \vee \varphi_1 \wedge \psi_1$

- $((\neg\varphi \wedge \psi) \supset \rho)$ sera remplacé par $\neg\varphi \wedge \psi \supset \rho$

b) à priorité égale l'évaluation se fait de la droite vers la gauche

Exemple : - $\varphi \vee \psi \vee \rho$ remplace $\varphi \vee (\psi \vee \rho)$

$\varphi \supset \psi \supset \rho$ remplace $\varphi \supset (\psi \supset \rho)$.

2.3.2.5 Axiomes :

On distingue dans l'ensemble X des axiomes :

- des axiomes logiques, valables pour toute structure d'information nous noterons X_ℓ leur ensemble

- des axiomes propres à la structure considéré, soit X_p leur ensemble.

Les axiomes logiques sont :

- les axiomes du calcul des propositions, regroupés en trois schémas :

$$SL_1 = \{\varphi \supset (\psi \supset \varphi) \mid \varphi, \psi \in F\}$$

$$SL_2 = \{(\varphi \supset (\psi \supset \rho)) \supset ((\varphi \supset \psi) \supset (\varphi \supset \rho)) \mid \varphi, \psi, \rho \in F\}$$

$$SL_3 = \{(\neg\varphi \supset \neg\psi) \supset (\psi \supset \varphi) \mid \varphi, \psi \in F\}$$

- les axiomes caractérisant l'égalité formelle. Ils expriment simplement le fait que ≡ est une relation d'équivalence (plus précisément $\mathcal{R}$ définie sur S^2 par "a $\mathcal{R}$ b $\iff$ a ≡ b est un théorème" est une relation d'équivalence) et qu'elle est compatible avec la concaténation :

$$SL_4 = \{u \equiv u \mid u \in S\}$$

$$SL_5 = \{u_1 \equiv v_1 \supset \ldots \supset u_n \equiv v_n \supset f\,u_1\ldots u_n \equiv f\,v_1\ldots v_n \mid$$

$$\text{pour } 1 \leqslant i \leqslant n \ , \ u_i \equiv v_i \in A \text{ et } f\,u_1\ldots u_n \equiv f\,v_1\ldots v_n \in A\}$$

$$SL_6 = \{u_1 \equiv u_2 \supset u_1 \equiv u_3 \supset u_2 \equiv u_3 \mid u_i \equiv u_j \in A \text{ si } 1 \leqslant i < j \leqslant 3\} \ .$$

Donnons maintenant des exemples d'axiomes propres :

Exemple 1 : l'ensemble X_p des axiomes propres de la structure de la liste sur V sont regroupés en trois schémas :

Pour les exprimer on distingue dans V un élément ω', à interpréter comme la valeur non définie

$$SH_1 = \{s^{(i)}t \equiv s^{(j)}t \supset s^{(i)}t \equiv \omega \mid i \neq j\}$$

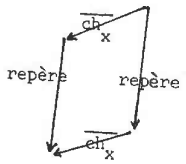
$$SH_2 = \{s\omega \equiv \omega \quad , \quad v\omega \equiv \omega'\}$$

$$SH_3 = \{\neg a \equiv b \mid a,b \in V \ , \ a \neq b\} \ .$$

SH_1 et SH_2 expriment que, ou bien tous les $s^{(i)}t$ sont "distincts" (liste infinie ou bien il existe un entier n tel que t, st ,.., $s^{(n-1)}t$ soient "distincts" et $s^{(n)}t$, $s^{(n+1)}$ ... soient "non définis" (liste de longueur n).

Exemple 2 :

- [A-2.2.3.5 b)] exprime la "commutativité" des fonctions repère et $\overline{ch_x}$ (sélecteur de champ x) ; i.e. que le diagramme suivant est commutatif :



formellement nous exprimerons cette propriété par le schéma d'axiomes :

$$\{rep\ ch_x\ u \equiv ch_x\ rep\ u \mid x \in L_o^{id} \ ; \ u \in EXNOM\}$$

(L_o^{id} est l'ensemble des symboles 0-aire interprétables comme des identificateurs ; EXNOM est l'ensemble des schémas fonctionnels interprétables comme des exemplaires de noms différents de *nil* [A-2.2.2 e)]

- "tous les exemplaires d'une valeur donnée autre que *nil* sont d'un même mode" [A-2.2.4.1 a)] . On traduit cette propriété par le schéma :

$$\{valeur\ u \equiv valeur\ v \supset mode\ u \equiv mode\ v \mid u,v \in EXVAL\}$$

(EXVAL est l'ensemble des schémas fonctionnels interprétables comme les exemplaires de valeurs différents de *nil*).

2.3.2.6 Règle d'inférence du système formel :

La seule règle d'inférence est la règle du détachement ou "Modus Ponens". Pour toute formule φ, ψ de F , on peut déduire ψ de φ et $\varphi \supset \psi$:

$$\varphi, \varphi \supset \psi \vdash \psi \ .$$

2.3.2.7 Remarque :

En résumé des définitions précédentes, ce qui différencie les systèmes formels de deux structures d'informations ce sont :

- les bases symboliques (L,m,pl)
- les ensembles X_p d'axiomes propres.

Ainsi le système formel d'une structure d'information est caractérisé par le quadruplet (L,m,pl,X_p) .

2.3.3 Information de la structure :

Définition 4 :

Une information de la structure est un sous ensemble de F contenant les théorèmes du système formel et stable par application de la règle du modus ponens. I est donc une information de la structure si et seulement si :

i) $X \subset I$

ii) Pour tous φ, ψ de F , $\varphi \in I$ et $\varphi \supset \psi \in I$ implique $\psi \in I$.

Exemple 1 : information de la structure de liste.

Une liste finie I, de longueur n, est spécifiée par la donnée d'une suite (a_i) $1 \leqslant i \leqslant n$ d'éléments de V. Elle peut être définie formellement par l'ensemble d'aximmes X_I suivant :

$$X_I = \{vs^{(i)}t \equiv a_i \mid 0 \leqslant i \leqslant n\} \cup \{s^{(n)}t \equiv \omega\} \ .$$

Une liste infinie est spécifiée par la donnée d'une suite infinie (a_i) $i \geqslant 0$ d'éléments de V. Elle peut être définie par

$$X_I = \{vs^{(i)}t \equiv a_i \mid i \geqslant 0\} \ .$$

Exemple 2 : Une information Algol 68 pourra être intuitivement considérée comme l'état d'un "calculateur formel" à un moment de l'élaboration d'un programme. Considérons le début de programme :

debut ent x = 3.2 ; rep reel y = loc reel := 4.1 ;

struct (rep reel x1, ent x2) z = (y,x) ;

L'information I obtenue après élaboration de ce morceau de programme est caractérisée, en plus des axiomes propres de la structure Algol 68, par les axiomes :

poss x ≡ *poss 3.2*

rep poss y ≡ *poss 4.1*

ch_{x1} *poss z* ≡ *poss y*

ch_{x2} *poss z* ≡ *poss x*

ainsi que par un certain nombre d'axiomes caractérisant la portée des différentes valeurs, les modes spécifiés par certains déclareurs "composés", les modes des différentes valeurs etc...

Alors *rep* ch_x *poss z* ≡ *poss 4.1* , *valeur poss x* ≡ *valeur poss 3.2* sont des exemples d'éléments de I .

Remarque :

Soient T l'ensemble des théorèmes du système formel et $\mathfrak{I}$ l'ensemble des informations de la structure. T appartient à $\mathfrak{I}$, ainsi que F. Toute intersection d'éléments de I est un élément de I . Donc, étant donné une famille $\{I_\lambda\}$ d'informations, il existe une plus petite information I contenant tous les I_λ . I est donc un treillis complet [0.1] pour la relation d'inclusion, d'élément minimal T et d'élément maximal F.

2.3.4 Retour sur la notion de profil :

a) Nous avons introduit la notion de profil pour éliminer certains schémas fonctionnels qui ne se prêtent pas à une interprétation agréable (exemple : *svt* pour une liste, ch_x *mode poss y* en Algol 68). C'est une première méthode pour éliminer ces schémas "parasites".

b) Une deuxième méthode consiste, par exemple, à introduire un (ou plusieurs) élément(s) ω interprétables comme le non défini ⊥ et d'imposer certains axiomes sur les schémas parasites.

Exemple : Si nous n'avions pas distingué les ensembles S_1 et S_2 dans la définition formelle d'une liste [2.3.2.2], une manière d'exprimer que les schémas fonctionnels du type *svt*, *svvt* ... sont indésirables consisterait à ajouter aux axiomes propres de la structure de liste le schéma

$$\{svs^{(k)}t \equiv \omega \mid k \geqslant 0\} \quad .$$

Une autre manière d'exprimer la même propriété, sans introduire ω, est d'imposer qu'un schéma parasite ne puisse apparaître dans une formule égalitaire ; en reprenant l'exemple précédent on écrirait

$$\{\neg svs^{(k)}t \equiv u \mid k \geqslant 0 \ , \ u \in S\} \quad .$$

c) Entre ces deux méthodes extrêmes, on peut souvent choisir une solution mixte en adjoignant la simplicité de la 1ère méthode à la souplesse de la 2ème.

Exemple : Dans la structure d'information Algol 68 on distingue plusieurs ensembles de schémas fonctionnels :

- EXNOM dont les éléments sont interprétables comme des exemplaires de noms
- EXMUL dont les éléments sont interprétables comme des exemplaires de valeur multiple
- EXSTRUC relatif aux exemplaires de valeurs structurées

etc...

L'équation définissant EXNOM contient dans son membre de droite :

rep^3 EXNOM pour exprimer qu'un exemplaire de nom peut repérer un autre exemplaire de nom.

Cependant *rep*^{3} EXNOM contient par exemple *rep*^{3} u où u est interprétable comme un exemplaire de nom repérant un entier, ce qui n'a aucun sens à cet endroit.
On pourrait introduire l'axiome :

$$\{mode\ u \equiv repère\ \mu \supset rep^3\ u \equiv \omega_3 \mid \mu \in \text{MODE} \ ; \ \mu \text{ ne commence pas par } repère\}$$

ω_3 étant l'élément indéfini associé à EXNOM.

d) Nous n'adopterons pas la solution précédente car, dans le cas d'un langage de programmation en particulier, une troisième méthode est applicable.
Pour alléger la formalisation on peut ne pas se donner la peine de "filtrer" tous les schémas fonctionnels et donc en admettre des indésirables au sens précédent, sans pour autant être gêné :

Nous avons déjà signalé que pour traduire l'élaboration d'un programme on peut considérer la suite d'états de la mémoire définie par ce programme ; c'est-à-dire une suite d'informations de la structure.
Si l'information initiale ne comporte que de "bons schémas" et si les modifications transformant une information en une autre n'introduisent pas de "parasites" on n'aura pas à considérer les informations indésirables. On utilise donc les contraintes syntaxiques imposées à un programme correct.

Il faut cependant remarquer que, si la première méthode est agréable à utiliser, elle est moins finie que la deuxième puisqu'elle ne permet d'engendrer que des langages algébriques (de schémas fonctionnels). Par contre la deuxième méthode alourdit considérablement la formalisation car elle exige un certain

nombre d'axiomes.

La troisième méthode quant à elle est étroitement liée à la forme des modifications élémentaires de la structure qui ne doivent pas introduire de schémas "parasites". Elle est impuissante à éliminer de tels schémas de l'ensemble T des théorèmes de la structure d'information.

Exemple : la propriété "le nom *nil* ne repère aucune valeur" [A-2.2.3.5 a)] ne pourra jamais résulter de cette méthode. Nous la traduirons à l'aide de la notion de profil mais on aurait pu tout aussi bien introduire le schéma d'axiomes $\{\neg \textit{repère nil} \equiv u \mid u \in S\}$.

Ainsi trois méthodes sont à notre disposition pour limiter les informations considérées, dans la suite nous utiliserons essentiellement la première et la troisième.

Dans la suite du chapitre on étudie quelques propriétés des structures d'informations dans le but de répondre aux questions concernant

- la manière de définir des accès dérivés à partir des accés initiaux [2.5]
- les modifications des informations d'une structure [2.6] .

2.4 Informations consistantes, complètes et Réalisations :

Ce paragraphe est une étude du treillis des informations d'une structure. On emploie la notation suivante : si I désigne une information et φ une formule, I $[\varphi]$ est l'information obtenue à partir de I en ajoutant φ aux axiomes, autrement dit la plus petite information contenant I et φ .

2.4.1 Informations consistantes :

Définition 1 :

Une information I est consistante si $I \neq F$.

Proposition 1 :

Une information I est consistante si et seulement si elle vérifie l'une des propriétés équivalentes suivantes :

i) Pour toute formule φ de F , $\varphi \in I \Rightarrow \neg\varphi \notin I$

ii) Pour toute formule atomique α de A , $\alpha \in I \Rightarrow \neg\alpha \notin I$.

Démonstration :

Si ii) est vérifié, i) se démontre par récurrence sur la longueur de φ. Il est clair que si i) est vérifié, I est consistante.

Montrons par l'absurde que si I est consistante alors ii) est vérifié : Si α et $\neg\alpha$ appartiennent à I il en est de même de $\alpha \wedge \neg\alpha$. Mais toute formule $\varphi \in I$ est conséquence tautologique de $\alpha \wedge \neg\alpha$ [0.2.3 définition 5] car $\vartheta(\alpha \wedge \neg\alpha)$ = FAUX pour toute fonction de vérité ϑ. Ainsi toute formule $\varphi \in I$ est un théorème de I [0.2.3 théorème 2] : I est inconsistante.

Proposition 2 :

Soit I une information. Une formule φ appartient à I si et seulement si I $[\neg\varphi]$ est inconsistante.

Démonstration :

Il est clair que si $\varphi \in I$, I $[\neg\varphi]$ est inconsistante.

Réciproquement soit α une formule atomique quelconque, alors $\alpha \wedge \neg\alpha \in I$ $[\neg\varphi]$ et avec le métathéorème de la déduction [0.2.2 théorème 1] : $\neg\varphi \supset \alpha \wedge \neg\alpha \in I$. On en déduit alors le résultat $\varphi \in I$ en remarquant que $\neg(\alpha \wedge \neg\alpha)$ est une tautologie et en utilisant les axiomes logiques, notamment SL_3 .

2.4.2 Informations complètes :

Définition 2 :

Une information I est complète si elle est maximale dans l'ensemble des

informations consistantes.

Proposition 3 :

Une information consistante I est complète si et seulement si elle vérifie l'une des conditions équivalentes suivantes :

i) Pour toute formule φ de F, $\varphi \notin I \Longrightarrow \neg\varphi \in I$

ii) Pour toute formule atomique α de A , $\alpha \notin I \Rightarrow \neg\alpha \in I$.

Démonstration :

- Si ii) est vérifié, i) se démontre par récurrence sur la longueur de
- Si I est complète ; soit α une formule atomique n'appartenant pas à I, si $\neg\alpha \notin I$, $I[\neg\alpha]$ est encore une information consistante (proposition 2) qui contient strictement I ce qui est impossible
- Si i) est vérifié, supposons I non complète alors il existe J consistante contenant strictement I. Si $\varphi \in J - I$, $\neg\varphi \in I$ donc $\neg\varphi \in J$ ce qui contredit la consistance de J.

Définition 3 :

Soit $\mathcal{S}$ une structure et I une information de $\mathcal{S}$. Une extension simple de I est une information I' de $\mathcal{S}$ contenant I.

Proposition 4 :

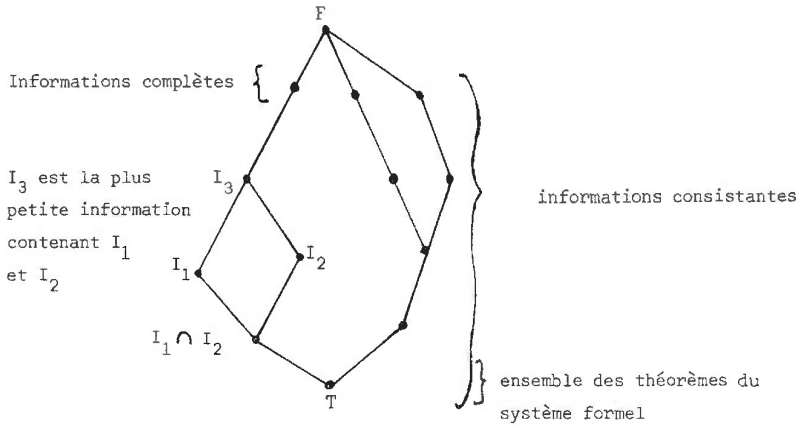
Toute information consistante I admet une extension simple complète.

Démonstration :

Elle est donnée intégralement dans [L9] page 47. Elle repose sur la notion de caractère fini : si E est un ensemble, $J \subset \mathcal{P}(E)$ est de caractère fini si : $A \in J \iff$ tout sous ensemble fini de A appartient à J .

Un lemme dû à Teichmüller-Tuckey précise que toute classe de caractère fini admet un élément maximal. Il suffit alors de prouver que la classe de tous les sous ensembles Γ de F tel que $I[\Gamma]$ est consistant est de caractère fini.

On peut résumer l'étude précédente par un schéma du treillis des informations d'une structure :



Treillis des informations

2.4.3 Réalisation d'une information :

Une information est un objet formel, parmi toutes les interprétations (données) d'une information I en existe-t-il pour lesquelles, si $u \equiv v$ est une formule atomique de I, $\bar{u} = \bar{v}$ ($\bar{u}$ et $\bar{v}$ étant les interprétations de u et v) ? Une telle interprétation s'appellera réalisation de I.

Cette notion de réalisation est un outil puissant pour l'étude d'une information, essentiellement en ce qui concerne la complétude et la consistance.

Définition 4 :

Soit I une information d'une structure et $R = (E_1, E_2, \ldots, E_m, r)$ une interprétation du triplet (L, pl, m) de la structure (4.2.1). Définissons une fonction de vérité $\tilde{r}$ sur l'ensemble des formules atomiques :

$$\tilde{r}(\alpha \equiv \beta) = \begin{cases} \text{VRAI si } \hat{r}(\alpha) = \hat{r}(\beta) \\ \text{FAUX sinon} \end{cases}$$

(voir [2.3.2.2 Théorème 3] pour la définition de $\hat{r}$)

$\tilde{r}$ se prolonge naturellement à toutes les formules à l'aide des fonctions $H_\neg$, $H_\wedge$, $H_\vee$, $H_\supset$ [0.2.3] .

On dit que R est une réalisation de I si, pour tout $\varphi \in I$, $\tilde{r}(\varphi) = \text{VRAI}$ Le m+1-uple $(E_1, \ldots, E_m, r(L))$ est une donnée au sens de [2.2.2] ; on peut dire que cette donnée réalise l'information I .
(On dit aussi que R est un modèle de I).

Proposition 5 :

R est une réalisation de I si et seulement si, pour tout axiome non logique φ de I , $\tilde{r}(\varphi)$ = VRAI.

Démonstration :

Tout axiome logique φ appartenant à l'un des schémas SL_1, SL_2 ou SL_3 est une tautologie, c'est-à-dire vérifie $\tilde{r}(\varphi)$ = VRAI [0.2.3] . D'autre part, il est immédiat par définition de la fonction de vérité $\tilde{r}$ que $\tilde{r}(\varphi)$ = VRAI si φ appartient à l'un des schémas SL_4, SL_5 ou SL_6 . La réciproque se justifie alors par récurrence sur la longueur de tout théorème φ de I .

Exemple :

Reprenons la liste d'ordre n considérée en [2.3.2] . On peut lui associer la réalisation suivante :

$E_1 = \{0, 1 - ., n-1, \perp\}$; $E_2 = V$;

$r(t) = 0$; $r(s) = \bar{s}$;

$r(\omega) = \perp$; $r(v) = \bar{v}$;

$r(a) = a$ pour $a \in V$;

où les fonctions $\bar{s}$ et $\bar{v}$ sont définies par :

$\bar{s}(i) = i+1$ pour $0 \leqslant i \leqslant n-1$

$\bar{s}(n-1) = \perp$

$\bar{s}(\perp) = \perp$

et $\bar{v}(i) = a_i$ pour $0 \leqslant i \leqslant n-1$

$\bar{v}(\perp) = \omega'$ (élément "non défini" dans V) .

Définition 5 :

Une formule φ de F est valide dans I (i.e. appartient à I) si et seulement si pour toute réalisation R de I , $\tilde{r}(\varphi)$ = VRAI .

Théorème 1 :

Une information I est consistante si et seulement si elle admet une réalisation.

Démonstration : ([L9] page 43).

Si I admet une réalisation et si $\varphi \in I$ alors $\tilde{r}(\varphi)$ = VRAI , donc $\tilde{r}(\neg\varphi)$ = FAUX i.e. $\neg\varphi \notin I$.

La réciproque se démontre en construisant la réalisation canonique de I : pour $1 \leqslant i \leqslant m$ E_i est l'ensemble quotient $S_{i/\#}$ où $\#$ est la relation

d'équivalence définie par $u \# v \iff u \equiv v \in I$

et pour tout symbole fonctionnel q-aire :

$$r(f)\,(\hat{r}(u_1),\ldots,\hat{r}(u_q)) = \widehat{f\,u_1\ldots u_q} \qquad \text{(classe de } fu_1\ldots u_q\text{)} .$$

Théorème 1' :

Une formule φ est un théorème de I si et seulement si elle est valide dans I . C'est une autre forme du théorème précédent.

- Il est clair que tout élément de I est valide
- réciproquement si φ est valide dans I supposons par l'absurde que $\varphi \notin I$ alors [2.4.1 proposition 2] I $[\neg\varphi]$ est consistante ; soit R une réalisation de I $[\neg\varphi]$, donc de I : $\tilde{r}(\varphi)$ = VRAI et $\tilde{r}(\neg\varphi)$ = VRAI ce qui est impossible.

Après avoir caractérisé les informations consistantes on utilise les réalisations pour caractériser les informations complètes.

Définition 6 :

Deux réalisations $R = (E_1,\ldots,E_m,r)$ et $R' = (E'_1,\ldots,E'_m,r')$ sont équivalents si $\tilde{r}(\varphi) = \tilde{r}'(\varphi')$ pour toute formule φ de F .

Par récurrence sur la longueur d'une formule il est immédiat que :

Proposition 6 : 2 réalisations R et R' sont équivalentes si et seulement si pour tout couple (u,v) d'éléments de S tels que $u \equiv v \in A$ on ait

$$\hat{r}(u) = \hat{r}(v) \text{ est équivalent à } \hat{r}'(u) = \hat{r}'(v) .$$

Théorème 2 :

Une information I est complète si et seulement si elle admet une réalisation et toutes ses réalisations sont équivalentes.

Démonstration :

Supposons I complète ; soit $\varphi \in F$

- si $\varphi \in I$ alors $\tilde{r}(\varphi)$ = VRAI pour toute réalisation
- si $\varphi \notin I$ alors $\neg\varphi \in I$ et donc $\tilde{r}(\varphi)$ = FAUX pour toute réalisation.

Inversement, si I admet au moins une réalisation, si toutes les réalisations sont équivalentes et si $\varphi \notin I$ alors (théorème 1') il existe une réalisation R_o de I telle que $\tilde{r}_o(\varphi)$ = FAUX . On en déduit que $\tilde{r}_o(\neg\varphi)$ = VRAI et donc que $\tilde{r}(\neg\varphi)$ = VRAI pour toute réalisation de I ce qui termine la démonstration (théorème 1').

2.5 Accès dérivés des accès élémentaires :

2.5.1 Extension d'une structure d'information :

Soient $\mathcal{S}$ et $\mathcal{S}'$ deux structures d'information, F et F' leurs ensembles de formules, T et T' leurs ensembles de théorèmes, $\mathcal{I}$ et $\mathcal{I}'$ leurs ensembles d'informations.

Définition 1 :

On dit que $\mathcal{S}'$ est une extension de $\mathcal{S}$ si $F \subset F'$ et $T \subset T'$.

Conséquence :

Toute information de $\mathcal{S}'$ contenue dans F est une information de $\mathcal{S}$ autrement dit : $\mathcal{I}' \cap \mathcal{P}(F) \subset \mathcal{I}$.

Définition 2 :

Soit I une information de $\mathcal{S}$; on appelle extension de I à $\mathcal{S}'$ la plus petite information I' de $\mathcal{S}'$ contenant I . On pose $I' = \mathcal{E}(I)$.

Définition 3 :

Soit I' une information $\mathcal{S}'$; on appelle restriction de I' à $\mathcal{S}$ l'information $I = I' \cap F$ de $\mathcal{S}$. On pose $I = \mathcal{R}(I')$.

Proposition :

Si $\mathcal{E}$ conserve la complétude

$\mathcal{R} \circ \mathcal{E}(I) = I$ pour toute information complète I de $\mathcal{S}$

$\mathcal{E} \circ \mathcal{R}(I') = I'$ pour toute information complète I' de $\mathcal{S}'$.

Remarque :

Si $\mathcal{E}$ conserve la complétude, elle conserve la consistance : en effet, toute information consistante I possède une extension simple complète I_1 . $\mathcal{E}(I_1)$ étant complète, $\mathcal{E}(I)$ est consistante car elle est contenue dans $\mathcal{E}(I_1)$.

Démonstration :

a) Soit I une information complète. $\mathcal{E}(I)$ est consistante ainsi que $\mathcal{E}(I) \cap F$. $\mathcal{E}(I) \cap F$ contient I . Puisque I est maximale, on a l'égalité.

b) Soit I' une information complète de $\mathcal{S}'$. $I' \cap F$ est également complète dans $\mathcal{S}$ (d'après [5.2], proposition 3), ainsi que $\mathcal{E}(I' \cap F)$ dans $\mathcal{S}'$. Puisque I' contient $\mathcal{E}(I' \cap F)$ on a l'égalité.

Définition 4 :

Si $\mathcal{E}$ onserve la complétude, on dit que $\mathcal{S}'$ se déduit de $\mathcal{S}$.

2.5.2 Structure déduite d'une autre par définition de symboles (ou accès) nouveaux :

Soit $\mathcal{S}$ une structure d'information et $f_1,\ldots,f_n$ des symboles fonctionnels n'appartenant pas à L. On considère une extension $\mathcal{S}'$ de $\mathcal{S}$ telle que $L' = L \cup \{f_1,\ldots,f_n\}$. L'ensemble des axiomes propres de $\mathcal{S}'$ est formé des axiomes propres de $\mathcal{S}$ et d'un ensemble $X[f_1,\ldots,f_n]$ d'axiomes "définissant" $f_1,\ldots,f_n$.

Exemple 1 :

[A-2.2] précise que les seules opérations arithmétiques nécessaires pour définir Algol 68 sont la soustraction, la multiplication et la division ; ce qu'on formalisera à l'aide de 3 symboles fonctionnels 2-aires : *moins*, *mul* et *div*.

Pour simplifier la définition de la sémantique on pourrait souhaiter introduire un symbole fonctionnel 2-aire *som* permettant d'accéder à la somme de deux entiers et défini à l'aide de *moins*.

Pour cela on considérerait l'extension $\mathcal{S}'$ de la structure Algol 68 définie par :

$L' = L \cup \{som\}$; $pl(som) = (15,15,15)$ (S_{15}, noté plus agréablement ENT, est l'ensemble des schémas fonctionnels interprétables comme des entiers) ; ainsi que l'ensemble d'axiomes :

$$X[som] = \{som\ u\ v \equiv moins\ u\ moins\ valeur\ poss\ 0\ v \mid u,v \in ENT\} .$$

Exemple 2 :

Etant donné une structure $\mathcal{S}$, on peut définir un symbole 4-aire *cond* par l'ensemble d'axiomes :

$$X[cond] = \{u \equiv v \supset cond\ (u,v,\omega,t) \overset{(*)}{\equiv} \omega \mid u,\ v,\ \omega,\ t \in S_i\}$$

$$\cup\{\neg u \equiv v \supset cond\ (u,v,\omega,t) \equiv t \mid u,\ v,\ \omega,\ t \in S_i\}$$

(S_i a été défini en [2.3.2.2] : c'est l'ensemble des schémas fonctionnels "à résultat" dans le $i^{ème}$ ensemble").

Soit I une information, I' l'information obtenue en ajoutant le symbole *cond* . Si $R = (E_1,\ldots,E_m,r)$ est une réalisation de I, on peut obtenir une réalisation $R' = (E_1,\ldots,E_m,r')$ de I' en posant :

$r'(f) = r(f)$ pour $f \in L$

$r'(cond) = \overline{Cond}$

(*) parenthèses et virgules sont utilisées dans la définition de *cond* pour faciliter la compréhension.

où $\overline{Cond}$ (a,b,c,d) = si a = b alors c sinon d pour a,b,c,d dans E_1 on peut montrer que pour toute structure $\mathcal{S}$, $\mathcal{S}'$ se déduit de $\mathcal{S}$.

Un tel symbole sera utile dans la formalisation de la sémantique de tout langage de programmation qui contient des propositions conditionnelles (si ... alors ... sinon).

Ces deux exemples ne posent aucun problème théorique, les nouveaux symboles étant définis simplement à l'aide des anciens.
Le paragraphe suivant va étudier une adjonction de symboles nouveaux beaucoup plus délicate : c'est le cas où un symbole doit être défini de manière récursive.

2.5.3 Accès nouveaux définis par un système d'axiomes à point fixe :

Etudions d'abord ce type de définition sur des exemples.

2.5.3.1 Exemples :

Exemple 1 :

Soit $\mathcal{S}$ la structure déduite de la structure de liste sur V en ajoutant le symbole *cond*. Soit $\mathcal{S}' = \mathcal{S}[ass]$ la structure obtenue en ajoutant le schéma d'axiomes

$$X_1 = \{ass\ s^{(i)}t,\ a) \equiv cond\ (vs^{(i)}t),\ a,\ s^{(i)}t,\ cond\ (s^{(i+1)}t,\ \omega,\ \omega,\ ass\ (s^{(i+1)}t,\ a))) \mid i \geqslant 0\ ,\ a \in V\}\ .$$

(Des parenthèses et des virgules ont été insérées pour rendre ce schéma lisible). Le symbole *ass* formalise l'accès associatif dans une liste à partir d'un élément quelconque.

. Un premier problème se pose quant à la correction de l'écriture ci-dessus : Le symbole *cond* a été défini pour des schémas fonctionnels de $\mathcal{S}$ et non pas de $\mathcal{S}'$ (tels que *ass* $(s^{(i+1)}t, a)$).

. Si nous supposons ce problème résolu, est-on capable de caractériser l'extension ainsi définie ; en particulier si I est une information de $\mathcal{S}$ consistante (par exemple une liste finie de longueur n) $\mathcal{S}(I)$ est-elle consistante ?

Pour étudier cette question considérons la réalisation R de I définie en [2.4.3]. Pour obtenir une réalisation de $\mathcal{S}(I)$ prolongeant R on doit choisir $\bar{r}(ass)$ de sorte que pour tout axiome φ de X_1, $\tilde{\bar{r}}(\varphi)$ = VRAI .
Il faut et il suffit pour cela que $\bar{r}(ass)$ vérifie l'équation d'inconnue $\bar{g}$:

$$\begin{cases} \bar{g}(i,a) = \text{si } \bar{v}(i) = a \text{ alors } i \text{ sinon si } \bar{s}(i) = \perp \text{ alors } \perp \text{ sinon } \bar{g}(\bar{s}(i),a) \\ \text{si } i = 0, 1, \ldots, n-1 \text{ et } a \in V \\ \bar{g}(\perp, a) = \perp\ . \end{cases}$$

On peut ordonner les ensembles E_1 et E_2 par la relation [0.1.1.4 exemples] :

a < b si et seulement si a = b ou a = $\perp$.

On peut de même ordonner l'ensemble des applications $E_1 \times E_2$ dans E_2 par la relation

$\bar{f} < \bar{g}$ si et seulement si, pour tout a $E_1 \times E_2$, $\bar{f}(a) < \bar{g}(a)$.

$\perp$ étant le symbole "non défini" $\bar{f} < \bar{g}$ si et seulement si, pour tout a tel que $\bar{f}(a)$ soit défini, on a $\bar{f}(a) = \bar{g}(a)$.

La fonction $r(ass)$ doit être solution de l'équation décrite ci-dessus :

$$\bar{g} = \mathcal{C}(\bar{g})$$

Pour l'ordre défini précédemment l'application $\mathcal{C}$ est continue. Cette équation admet donc une solution minimale [0.1.1.4] .

En fait toute solution peut être choisie comme image de *ass* par r .
Par exemple considérons maintenant la liste infinie d'axiomes :

$$vs^{(i)}t = b \text{ pour } i \geqslant 0\ .$$

Il est facile de voir que $\bar{g}$ est solution de l'équation si et seulement si

$$\bar{g}(i,b) = i \quad \text{pour} \quad i \geqslant 0$$

et $\bar{g}(i,a) = g(i',a)$ pour $a \neq b$, $i \geqslant 0$, $i' \geqslant 0$.

En effet pour $a \neq b$ l'équation se réduit à : $\bar{g}(i,a) = \bar{g}(i+1,a)$.
La multiplicité des solutions du système traduit le fait que $\mathcal{S}(I)$ n'est pas complète, ou de manière plus concrète le fait que l'accès associatif n'est pas défini pour les listes infinies. Dans cet exemple un programme cherchant le plus petit entier i tel que $\bar{v}(i) = a$ ne se termine pas si a est différent de b .

Exemple 2 :

Considérons en Algol 60 la déclaration de procédure :

entier procédure f(x) ;

*f := si x = 0 alors 1 sinon x * f(x-1)* .

Il est naturel d'associer à cette déclaration de procédure un symbole fonctionnel unaire *fact* qui donne accès au résultat de la procédure, c'est-à-dire que dans une réalisation *fact* puisse être interprété comme la fonction $\overline{fact}$

(définie par $\overline{fact}(n) = n!$ pour tout entier $n \geqslant 0$).

En notant $\mathcal{S}$ la structure d'information Algol 68, soit $\mathcal{S}' = \mathcal{S}[fact]$ la structure obtenue en ajoutant le schéma d'axiomes :

$$X[fact] = \{fact\ x \equiv cond\ (x,\ 0,\ 1,\ mul(x,\ fact(moins\ x\ 1)) \mid x \in ENT\}$$

(avec des notations immédiates).

Cette définition pose alors les mêmes problèmes que dans l'exemple 1. La suite du paragraphe présente une formalisation rapide de ce type de définition d'accès nouveaux ; pour une étude plus poussée voir [SI7] .

2.5.3.2 Caractérisation d'une extension définie par des accès nouveaux :

a) Pour donner une définition générale de ce type d'extension, nous devons d'abord généraliser la notion de schéma fonctionnel en y permettant des arguments. Pour cela, il suffit d'ajouter à l'ensemble L des accès, pour $j = 1, 2, \ldots, m$, un ensemble dénombrable Z_j de paramètres formels de profils (j) (on dit encore de variables de profils (j)).

On définit ainsi de nouveaux ensembles des schémas fonctionnels [2.3.2.2] S'_j $(1 \leqslant j \leqslant m)$ avec $S'_j = S_j \cup Z_j$.

b) Substitution :

Considérons $h \in S'_k$ (où $k \in \{1,\ldots,m\}$) et une suite de paramètres formels $z_1 \in Z_{j_1}, \ldots, z_q \in Z_{j_q}$ parmi lesquels tous ceux qui entrent dans h : $(j_1,\ldots,j_q,k)$ est un profil associé à h .

Pour tous $u_i \in S_{j_i}$ $(1 \leqslant i \leqslant q)$ nous noterons $h[u_1,\ldots,u_q]$ le résultat de la substitution dans h de u_1 à z_1 ... u_q à z_q . Alors [2.3.2.2 théorème 2] :

$h[u_1,\ldots,u_q] \in S_k$.

c) Accès nouveau :

Un accès nouveau est caractérisé par la donnée
- d'un symbole nouveau f
- d'un profil à un élément $(j_1,\ldots,j_q,k)$
- d'un élément h de S'_k ; S'_k étant défini comme ci-dessus mais relativement à l'ensemble de symboles fonctionnels $L \cup \{f\}$
- d'un ensemble d'axiomes $X[f]$ de la forme :

$$X[f] = \{f\ u_1 \ldots u_q \equiv h[u_1,\ldots,u_q] \mid u_1 \in S_{j_1}, \ldots, u_q \in S_{j_q}\} .$$

Exemples :

1) En reprenant l'exemple 1 [2.5.3.1] : l'accès nouveau est caractérisé par :
- le symbole *ass*
- le profil (1,2,1) (en reprenant les notations de [2.3.2.2 exemple 1])
- le schéma $h = cond\ (v\ z_1,\ z_2,\ z_1,\ cond\ (s\ z_1,\ \omega,\ \omega,\ ass\ (s\ z_1,\ z_2)))$
- $X[f] = \{ass\ u_1\ u_2 \equiv h[u_1,\ u_2] \mid u_1 \in S_1\ ,\ u_2 \in S_2\}$.

2) Le schéma h sur $L \cup \{fact\}$ formalisant l'exemple 2 [2.5.3.1] est donc :

$$h = cond(z_1,0,1,\ mul(z_1,\ fact\ (moins\ z_1\ 1))) .$$

Remarque 1 :

On peut donner une définition analogue dans le cas où f doit posséder un profil non réduit à un élément ; si $\ell = card(pl(f))$ il faudra alors introduire ℓ schémas fonctionnels $h_i \in S'_k$ (sur $L \cup \{f\}$).

Remarque 2 :

On aura, la plupart du temps, à introduire plusieurs nouveaux accès, on peut le faire par étapes successives, mais on peut être obligé d'en introduire plusieurs en même temps (pour formaliser, par exemple, deux fonctions récursives, chacune étant définie à partir de l'autre ; un autre exemple est fourni par les modes récursifs en Algol 68).

On est donc conduit à généraliser la définition précédente :
Un n-uple $(n \geqslant 1)$ d'accès nouveaux est caractérisé par la donnée :
- d'un n-uple $(f_1,\ldots,f_n)$ de symboles nouveaux
- d'un n-uple de profils à un élément $((j^i_1,\ldots,j^i_{q_i},k^i))_{1\leqslant i\leqslant n}$
- d'un n-uple d'éléments de S'_{i_k} sur l'ensemble $L \cup \{f_1,\ldots,f_n\}$
- d'un ensemble d'axiomes $X[f_1,\ldots,f_n] = \bigcup_{i=1}^{n} X[f_i]$ où $X[f_i]$ est défini comme ci-dessus.

Remarque 3 :

Dans toute la suite de ce travail on supposera définis les Z_j et donc on considérera implicitement les S'_j à la place de S_j mais avec la restriction suivante : une interprétation de la structure n'est définie que sur les S_j .

d) Propriétés de l'extension définie par un système d'axiomes à point fixe :

Soit I une information de la structure $\mathcal{S}$. Une réalisation $R = (E_1,\ldots,E_m,r)$ de I permet d'écrire un système à point fixe (Ⓗ) associé aux schémas d'axiomes définissant les accès nouveaux $f_1,\ldots,f_n$; un exemple de tel système est donné en [2.5.3.1 exemple 1] .

Une définition correcte de (Ⓗ) nécessite une généralisation de la notion de réalisation au cas d'un système formel admettant des ensembles de variables Z_i $(1 \leqslant i \leqslant m)$, dans une telle réalisation un schéma h dont un profil associé est $(j_1,\ldots,j_q,k)$ est interprété comme une fonction de $E_{j_1} \times \ldots \times E_{j_q}$ dans E_k (dans [2.5.3.1 exemple 1], la fonction $\bar{g}$ est l'interprétation de h).

Ce système permet de prolonger R à $\mathcal{E}(I)$.
On obtient alors le résultat suivant :

Théorème : - Si I est consistante, $\mathcal{E}(I)$ est consistante
- Si (Ⓗ) admet plusieurs solutions, $\mathcal{E}(I)$ n'est pas complète.

Idée de la démonstration :

i) On démontre que Ⓗ est un système à point fixe associé à une fonction continue, il admet donc une solution minimale $(F'_1,\ldots,F'_n)$

On obtient une réalisation $R' = (E'_1,\ldots,E'_m,r')$ de $\mathcal{E}(I)$ à partir de R en posant :

$$E'_i = E_i \quad (1 \leqslant i \leqslant m)$$

$$r'(f) = r(f) \quad \text{pour} \quad f \in L$$

$$r'(f_i) = F'_i \quad \text{pour} \quad 1 \leqslant i \leqslant n \ .$$

ii) Si $(F'_1,\ldots,F'_n)$ et $(F''_1,\ldots,F''_n)$ sont deux solutions distinctes de (Ⓗ), les réalisations R' et R'' sont non équivalentes et donc [2.4.3 ; théorème 2] $\mathcal{E}(I)$ n'est pas complète.

2.6 Modifications élémentaires d'une structure

Nous avons dit qu'une structure d'information était la donnée d'un système formel $\mathcal{F}$ et d'un ensemble de modifications dites élémentaires.

2.6.1 Modifications élémentaires

Soit une structure $\mathcal{S}$ et $\mathcal{J}$ son ensemble d'informations.

Définition 1 :

Une modification est une application de $\mathcal{J}$ dans $\mathcal{J}$.

Se donner un ensemble de modifications revient à décider quelles modifications élémentaires sont acceptables dans le cadre d'une structure d'information.

Exemple 1 :

On veut adjoindre, dans une liste sur V , un élément derrière un élément donné u intuitivement, cela signifie que su va devenir le successeur du successeur de u, ssu .
Pour traduire cela, on introduit la structure $\mathcal{S}'$ obtenue à partir de $\mathcal{S}$ en remplaçant s par s', puis la structure $\mathcal{S}''$ "contenant" $\mathcal{S}$ et $\mathcal{S}'$. Plus précisément :

$$L'' = \{t, v\} \cup \{s, s'\} \cup \{\omega\} \cup V \ .$$

$\mathcal{S}''$ traduit la cohabitation de $\mathcal{S}$ et de $\mathcal{S}'$. L'adjonction d'un élément $s^{(i)}t$ se traduit alors par les axiomes :

$$\begin{cases} s's'u \equiv su \\ \{\neg u \equiv v \supset s'v \equiv sv \mid v \in S_1\} \ . \end{cases}$$

Examinons comment ces axiomes définissent une modification comme une application de $\mathcal{J}$ dans $\mathcal{J}$.
Soit I une information de $\mathcal{S}$, on considère successivement
- l'information I'', extension de I dans la structure $\mathcal{S}''$, notée $\mathcal{E}''(I)$
- l'information I', restriction de I'' dans la structure $\mathcal{S}'$, notée $\mathcal{R}'(I'')$
- l'information I_1, obtenue à partir de I' en remplaçant s' par s .

I_1 est une information de $\mathcal{S}$, résultat de la modification définie par les axiomes écrits précédemment. On notera cette modification adj(u) .

Revenons maintenant au cadre général.

Définition 2 :

Soit une structure $\mathcal{S}$ définie par l'ensemble d'accès L, l'entier m, la fonction profil pl, l'ensemble d'axiomes propres X. Soit une bijection σ de L sur un ensemble L', dont la restriction à $L \cap L'$ est l'identité. On définit une

fonction profil pl' dans L' par :

$$pl'(\sigma(f)) = pl(f) \quad \text{pour } f \in L \ .$$

σ se prolonge naturellement en une application de l'ensemble des formules F sur un ensemble F' : F' est l'ensemble des formules de la structure $\mathcal{S}'$ définie par L', m, pl' et X' = σ(X) .

Soit $\mathcal{S}''$ la structure définie par L $\cup$ L', l'entier m, la fonction profil égale à pl sur L et à pl' sur L', ainsi qu'un ensemble d'axiomes contenant X et X' :

$$X'' = X \cup X' \cup Y \ .$$

La modification élémentaire π définie par η et l'ensemble d'axiomes Y est :

$$\pi = \sigma^{-1} \circ \mathcal{R}' \circ \mathcal{E}''$$

où $\mathcal{R}''$ et $\mathcal{E}'$ ont la même signification que dans l'exemple précédent.

$$I \xrightarrow{\mathcal{E}''} I'' \xrightarrow{\mathcal{R}'} I' \xrightarrow{\sigma-1} \pi(I) \ .$$

Exemple 2 :

La modification π = aff(u a) qui affecte la valeur a à l'élément u d'une liste est associée à la bijection σ changeant v en v' (et laissant invariants les autres symboles de L) et à l'ensemble X_π d'axiomes :

$$X_\pi = \{v'\ u \equiv a\ ,\ (\neg u \equiv v) \supset v'\ v \equiv v\ v \mid v \in S_1\} \ .$$

Exemple 3 :

Dans un sous langage d'Algol 68 ne contenant pas de valeurs multiples (tableaux) l'affectation $x := y$ (où x et y sont des identificateurs) serait traduite par la modification affect(x,y) définie par :

a) $\sigma(rep) = rep'$; $\sigma(valeur) = valeur'$; $\sigma(f) = f$ pour tout $f \in L$

qui est différent de *rep* et *valeur*

b) $X_{affect(x,y)} = \{\neg\, valeur\ poss\ x \equiv nil \wedge ppq\ portée\ possx\ portée\ possy \equiv vrai \supset valeur'\ rep'\ possx \equiv valeur\ possy\} \cup$

$\{valeur\ v \equiv valeur\ possy \supset \neg v \equiv rep'\ possx \mid v \in EXVAL - \{possy\}\} \cup$

$\{rep'\ v \equiv rep\ v \mid v \in EXNOM - \{possx\}\} \cup$

$\{valeur'\ v \equiv valeur\ v \mid v \in EXVAL - \{rep'\ possx\}\}$.

Explicitons ces axiomes :

(on traduit [A-8.3.1.2 a), b), c) pas 1 et 2]).

i) le premier axiome exprime
- que le nom possédé par x ne peut être *nil*
- que la portée de ce nom doit être plus petite que la portée de l'objet interne possédé par y ; (le symbole fonctionnel 2-aire *ppq* "induit une relation d'ordre" sur l'ensemble des schémas fonctionnels interprétables comme des portées)
- et que si les conditions précédentes sont vérifiées la valeur de l'objet interne repéré par *poss x* est la valeur de l'objet possédé par y

ii) le deuxième ensemble d'axiomes précise que *rep' poss x* est une nouvel exemplaire de valeur

iii) les deux ensembles suivants expriment que *rep'* et *valeur'* ne sont modifiés que par les deux premiers ensembles d'axiomes.

Nous verrons [6] que l'introduction de valeurs multiples complique notablement cette définition.

Convention :

Une modification élémentaire π est définie par la donnée
- de la bijection σ
- de l'ensemble d'axiomes X_π .

Dans la suite nous conviendrons de ne définir σ que sur les symboles fonctionnels effectivement modifiés (éléments de L - L') ; elle se prolongera implicitement en l'identité sur les autres symboles.

On note $\mathcal{M}od$ l'ensemble des modifications élémentaires de la structure est donc le couple $(\mathcal{F}, \mathcal{M}od)$.
Une modification de $\mathcal{S}$ est soit une modification élémentaire, soit la composée de deux modifications de $\mathcal{S}$; nous noterons $\hat{\mathcal{M}}od$ leur ensemble.

2.6.2 Schémas de modifications :

Dans l'exemple 2 précédent, π dépend de u et a : nous avons en fait défini un schéma de modifications à deux paramètres aff.
Une structure d'informations est définie par un système formel $\mathcal{F}$ et un nombre fini de schémas de modifications élémentaires. Les modifications élémentaires sont des applications et leur composition permet de définir celle des schémas de modifications. Il est utile pour la suite de généraliser la notion de schéma

3

Langage pivot

et ensemble de calculs

3. LANGAGE PIVOT ET ENSEMBLE DE CALCULS

Ce concept de structure d'information étant précisé, rappelons [1.2] que notre formalisation de la sémantique d'un langage de programmation nécessite :

i) l'étude des notions de langage pivot et de calcul

ii) la définition d'un mécanisme permettant d'associer à toute phrase du langage pivot un ensemble de calculs et éventuellement un accès de la structure d'information.

Dans ce chapitre, à la suite de brèves considérations sur les langages pivots [3.1], nous définissons un langage simple de programmation (appelé LS) [3.2] qui permet, après que l'on ait étudié la notion de calcul [3.3], d'illustrer ii) [3.4]. En [3.5] on définit complètement la sémantique de LS.

3.1 Langage pivot :

Rappelons que cette notion de langage pivot permet essentiellement de représenter tout programme sous une forme qui met en évidence la nature (syntaxique) des différentes phrases qui le composent. De plus le passage d'un programme (concret) au programme pivot associé (traduction) permet d'introduire explicitement certains renseignements (tels que portée d'un nom en Algol 68, domaine de validité d'une déclaration d'identité ...) et de simplifier certaines constructions (boucles par exemple).

En bref le langage pivot possède des propriétés telles que la définition de la sémantique d'un de ses programmes soit plus simple que celle du programme concret associé.

Dans notre formalisation, la forme d'un programme pivot est une ramification ce qui est naturel dès qu'on remarque que la signification d'une phrase ne dépend que de ses composantes (dans la méthode de Vienne la forme retenue est celle d'arbre dont les branches sont valuées).

En fait nous ne présenterons pas, dans ce travail, la définition formelle d'un langage pivot : s'il est certain qu'une telle étude serait indispensable pour une définition complète, il ne différerait pas sensiblement de celui de la méthode de Vienne.

On peut d'ailleurs remarquer que, avec cette méthode, la démarche logique pour définir un nouveau langage consiste à décrire en premier lieu le langage pivot puis, de manière indépendante, définir la syntaxe du langage concret et la sémantique du langage pivot.

3.2 LS : un langage simple de type Algol :

Pour illustrer notre méthode de formalisation et en particulier l'association d'un ensemble de calculs à une phrase d'un langage de programmation nous utiliserons le plus souvent

- des exemples provenant d'Algol 68
- des exemples issus d'un langage de type algol beaucoup plus simple appelé LS et dont la syntaxe est définie ci-dessous (de manière classique, sous forme de Backus) :

3.2.1 Syntaxe de LS :

a) Syntaxe de LS :

<programme> ::= <bloc>
<bloc> ::= *début* {<partie déclaration>}$^{0|1}$ <proposition> {;<proposition>}* *fin*(*)
<partie déclaration> ::= <déclaration>; {<déclaration>;}*
<déclaration> ::= <déclaration simple> | <déclaration collatérale>
<déclaration collatérale> ::= <déclaration simple> ,< déclaration simple> {,<déclaration simple>}*
<déclaration simple> ::= <déclaration de variable> | <déclaration de constante> | <déclaration de procédure>
<déclaration de variable> ::= <type> <identificateur>
<type> ::= *bool* | *ent* | *reel*
<déclaration de constante> ::= <identificateur> = <proposition>
<déclaration de procédure> ::= *proc* <identificateur> {[<paramètres formels>]}$^{0|1}$: <bloc>
<paramètres formels> ::= <déclaration de variable> {;<déclaration de variable>}*
<proposition> ::= <proposition conditionnelle> | <affectation> | <appel de procédure> | <terme>
<proposition conditionnelle> ::= *si* <terme> *alors* <bloc> *sinon* <bloc>
<affectation> ::= <identificateur> := <terme>
<terme> ::= <constante> | <identificateur> | <formule> | <bloc>
<formule> ::= (<terme> <op. bin> <terme>)
(<op.un> <terme>)
<op.bin> ::= + | - | × | / | ∧ | ∨ | ⩽ | = | <
<op.un> ::= - | ¬

(*) . {<notion>}$^{0|1}$ représente <notion> ou rien
. {<notion>}* représente l'écriture, un nombre quelconque de fois éventuellement nul de <notion>

<appel de procédure> ::= <identificateur> {[<paramètres effectifs>]}$^{0|1}$
<paramètres effectifs> ::= <identificateur> {;<identificateur>}

nous n'expliciterons pas <constante> et <identificateur> et nous supposerons seulement que parmi les constantes se trouvent *vrai* et *faux*.

3.2.2 Structure d'information de LS :

Nous nous contenterons d'en donner une idée intuitive mais en même temps, dans le but d'illustrer suffisamment de notions, nous la présenterons sous une forme plus complexe que celle qui serait nécessitée par le strict problème de la formalisation de LS.

En reprenant la terminologie de [A], on distingue les objets externes (qui sont les phrases du langage) des objets internes (qui sont les valeurs manipulées par ces phrases). Les objets externes sont les symboles 0-aires de la structure d'information $\mathcal{S}$; les objets internes sont des schémas fonctionnels.

Parmi les symboles fonctionnels n-aires ($n \geqslant 1$) on rencontre :

- *poss* : qui sera interpété comme (brièvement "qui sera") la fonction associant à un objet externe l'objet interne correspondant. En particulier un identificateur peut posséder une constante (à la suite d'une déclaration de constante) ou une variable (nom au sens d'Algol 68) qui repère une valeur. On introduit le symbole :
- *rep* : qui formalise la fonction repère
- *type* : qui associe à chaque objet interne son type : *reel*, *ent*, *bool* pour une constante ; *repère reel*, *repère ent*, *repère bool* pour une variable. Enfin un dernier type sera utilisé pour les phrases qui ne possèdent pas de valeur [3.4] (ex : appel de procédure car nous ne considérons que des procédures sans résultat) : c'est le type *neutre*.

Enfin parmi les autres symboles fonctionnels, un certain nombre formalisent les opérations arithmétiques et logiques (*moins*, *plus*, *mul*, *div*, *ou*, *et*, *non*). Les principales modifications élémentaires de $\mathcal{S}$ sont :

- affect : schéma de modifications à deux paramètres x et E formalisant l'affectation x := E . Il modifie le seul symbole *rep* et est défini par les axiomes :

$$X_{affect(x,E)} = \{type\ poss\ x \equiv type\ poss\ E \supset rep'\ poss\ x \equiv rep\ poss\ E\} \cup \{type\ poss\ x \equiv repère\ type\ poss\ E \supset rep'\ poss\ x \equiv poss\ E\} \cup \{rep'\ u \equiv rep\ u \mid u \neq poss\ x\}$$

le premier axiome concerne les affectations dont le deuxième membre possède un nom,

le deuxième concerne celles dont le deuxième membre possède une constante.

- var : schéma de modifications à deux paramètres $\underline{t}$ et x qui formalise la déclaration de variable $\underline{t}$ x.

Seuls les symboles *type* et *poss* sont transformés :

$$X_{var(\underline{t},x)} = \{type'\ poss'\ x \equiv repère\ \underline{t}\} \cup \{type'\ u \equiv type\ u \mid u \neq poss'\ x\} \cup \{poss'\ y \equiv poss\ y \mid y \neq x\}$$

- const : schéma de modifications à deux paramètres x et E , qui formalise la déclaration de constante x = E .

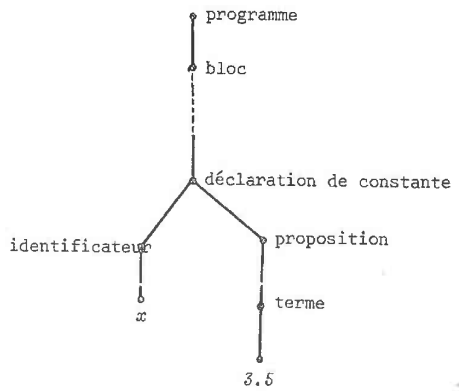
Le symbole *poss* est modifié :

$$X_{const(x,E)} = \{type\ poss\ E \equiv repère\ \underline{t} \supset poss'\ x \equiv rep\ poss\ E \mid \underline{t} \in \{\underline{bool},\underline{ent},\underline{reel}\}\}$$
$$\{type\ poss\ E \equiv \underline{t} \supset poss'\ x \equiv poss\ E \mid \underline{t} \in \{\underline{bool},\underline{ent},\underline{reel}\}\}$$
$$\{poss'\ y \equiv poss\ y \mid y \neq x\}\ .$$

3.2.3 Le langage pivot LS_o :

Soit P le programme de LS. Le programme P_o du langage pivot LS_o associé à P est la ramification engendrée par la grammaire de LS et dont le mot des feuilles est P . Toutefois nous supposerons exclues de cette ramification certaines branches qui n'ont aucune utilité sémantique.

Par exemple la phrase pivot associée à la déclaration de constante $x = 3.5$ du programme P sera :



De la ramification syntaxique initiable on a supprimé les branches relatives aux symboles *début*, *fin*, = .

On peut également utiliser la traduction langage évolué → langage pivot pour résoudre des problèmes tels que la protection d'identificateurs au sens d'Algol 68 [A-6.0.2 d)]. C'est-à-dire que si un même identificateur fait l'objet de deux déclarations (de variable ou de constante) dans deux blocs dont l'un est inclus dans l'autre il faut changer l'un de ces identificateurs.

Dans la suite et pour alléger les notations, nous ne représenterons que le mot des feuilles d'une telle ramification. Etudions maintenant la notion de calcul.

3.3 Calculs :

3.3.1 Calculs finis et infinis :

Une élaboration d'un programme P d'un langage algorithmique $\mathcal{L}_g$ est une suite d'élaborations de phrases élémentaires de P définie : - par les données de P
- par la structure de P.

Une telle suite s'appellera un calcul. On est conduit à :

Définition 1 :

Un calcul du langage $\mathcal{L}_g$ est une suite finie ou non d'éléments de l'ensemble $\mathcal{M}od$ des modifications élémentaires de la structure d'information de $\mathcal{L}_g$.

- Un calcul fini est un élément du monoïde libre $\mathcal{M}od^*$
- Nous noterons $\mathcal{M}od^\omega$ l'ensemble des calculs infinis de $\mathcal{L}_g$ et

$$\mathcal{M}od^\infty = \mathcal{M}od^* \cup \mathcal{M}od^\omega\ .$$

On étend l'opération de concaténation à $\mathcal{M}od^\infty$ par :

- si $m_1 = (a_1,a_2,\ldots,a_n) \in \mathcal{M}od^*$
et $m_2 = (b_1,b_2,\ldots,b_m) \in \mathcal{M}od^\omega$
alors $m_1 \cdot m_2 = (a_1a_2,\ldots,a_n,b_1,b_2,\ldots,b_m)$
- si $m_1 \in \mathcal{M}od^\omega$ et $m_2 \in \mathcal{M}od^\infty$ alors $m_1 \cdot m_2 = m_1$.

Ainsi $(\mathcal{M}od^*,.,\Lambda)$ et $(\mathcal{M}od^\infty,.,\Lambda)$ sont deux monoïdes (où Λ est le mot vide).

En particulier $(a_1.a_2.\ldots.a_n)$ et $a_1 \cdot a_2 \ldots \cdot a_n$ sont deux représentations du même mot.

Exemples :

1) Dans LS, en début de programme,

début réel x ; *réel* y ; y := 3.4

sera associé le calcul :

var(*réel*,x) . var(*réel*,y) . affect(y,3.4)

avec var schéma de modification associé à une déclaration de variable ; affect formalisant l'affectation.

2) Considérons la déclaration de procédure de LS :

proc f : *début* $x := 1$; f *fin* .

Le calcul formalisant un appel de f est le calcul infini :

affect(x,1) . affect (x,1) . affect (x,1) ... .

a) Calculs finis :

Tout calcul fini peut être interprété comme une modification de $\mathcal{S}$ lorsqu'on interprète la concaténation comme la composition (dans l'ordre inverse) des modifications élémentaires.

Ainsi $C = \pi_1 \cdot \pi_2 \cdot \cdots \cdot \pi_n$ sera interprété comme la modification

$\pi = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_2 \circ \pi_1 \in \widehat{\mathcal{M}od}$.

On retrouve la notion de fonction de changement d'état sans oublier pour autant l'histoire de l'élaboration du programme.

Exemple : Soit adj (resp. aff) le schéma de modifications élémentaires défini en [2.6.1 exemple 1] (resp. [2.6.1 exemple 2]) traduisant l'adjonction d'un élément à une liste (resp. l'affectation d'une valeur à un élément d'une liste).

Au calcul $c = adj(s^k t) \cdot aff(s^{k+1}t,a) \cdot adj(s^{k+1}t) \cdot aff(s^{k+2}t,b)$ est associée la modification π transformant, par exemple, la liste finie :

$$I \begin{cases} \forall\, s^i t \equiv a_i & 0 \leqslant i \leqslant n-1 \\ s^n t \equiv \omega & \\ \text{où } n > k \text{ en la liste} & \end{cases}$$

$$\pi(I) \begin{cases} \forall\, s^i t \equiv a_i & 0 \leqslant i \leqslant k \\ \forall\, s^{k+1} t \equiv a & \\ \forall\, s^{k+2} t \equiv b & \\ \forall\, s^i t \equiv a_{i-2} & k+3 \leqslant i \leqslant n+1 \\ \forall\, s^{n+2} t \equiv \omega & \end{cases}$$

b) Calculs infinis :

Un tel calcul ne peut être interprété comme une fonction :
Deux attitudes (au moins) sont possibles :

- décider de ne pas interpréter un tel calcul
- associer à ce calcul $a_1\, a_2 \cdots a_n \cdots$ la suite de fonctions $a_1, a_1, \ldots, a_n, \ldots$

Remarque 1 : d'autres approches (méthode de Vienne par exemple) préfèrent définir un calcul comme une suite d'états de mémoire. Il faut remarquer qu'un tel état contient le programme considéré et un compteur ordinal indiquant le numéro de l'instruction en cours d'exécution. Un tel point de vue serait possible ici en associant au couple formé par ce calcul $a_1 \ldots a_n$ et une information "initiale" I_o de la structure $\mathcal{S}$ la suite infinie d'informations définie par $I_n = a_n(I_{n-1})$ pour $n \geqslant 1$.

L'association d'une suite de modifications à un calcul infini peut être plus délicate que ne semble le montrer ce qui précède :
Soit g la procédure de LS définie par :

proc g : *debut* g ; $x := 1$ *fin* .

Nous verrons qu'à un appel de g on associe l'équation à point fixe [3.3.3] :

$\tilde{g} = \tilde{g}$. affect(x,1) .

Or avec les définitions précèdentes cette équation n'admet pas de solution dans $\mathcal{M}od^\infty$.
Pour rendre compte de tels appels il suffit d'introduire un symbole δ qui s'interprétera comme la modification identique et d'associer à l'appel de g l'équation :

$\tilde{g} = \delta \cdot \tilde{g}$. affect(x,1) qui admet comme solution le calcul infini :

$\delta \cdot \delta \cdot \delta \ldots$ auquel on associe la suite de modifications :
i, i, i, ... où i est l'identité.

L'introduction de δ correspond bien à la réalité car on réappelle sans cesse la procédure g et chaque appel prend un certain temps. Pour être tout-à-fait correct il serait donc nécessaire d'introduire un ensemble $\mathcal{M}od$ dont les éléments seraient des "symboles" et une application de $\mathcal{M}od$ dans $\mathcal{M}od$. Un calcul serait alors un élément de $\mathcal{M}od^\infty$, le symbole δ ayant pour image la modification identique. Par abus de notation nous confondrons $\mathcal{M}od$ et $\mathcal{M}od$.

3.3.2 Ensembles et schémas de calculs :

3.3.2.1 Ensembles de calculs :

Si la concaténation de $\mathcal{M}od^\infty$ permet d'exprimer l'élaboration des phrases sérielles (au sens de [A]), il faut pouvoir rendre compte des propositions conditionnelles. En effet à tout programme qui contient une telle proposition C est associé

plusieurs élaborations possibles selon que, en fonction de données, c'est la "proposition *alors*" de C ou la "proposition *sinon*" qui est élaborée. La signification d'un tel programme n'est pas un calcul mais un ensemble de calculs. Cette notion est également nécessaire pour rendre compte de l'indéterminisme, et en particulier de celui qui provient des propositions collatérales [3.3.3.2 c)].

Exemple : à la phrase *si b alors x := 3 sinon y := 4* de LS

est associé l'ensemble de calculs :

$$j(\textit{rep poss b, vrai}).\text{affect}(x,3) \cup \bar{j}(\textit{rep poss b, vrai}).\text{affect}(y,4)$$

où j(u,v) (resp. $\bar{j}$(u,v)) est la modification identique sur l'ensemble des informations I telles que $u \equiv v \in I$ (resp. $\neg u \equiv v \in I$) et elle transforme I en l'information inconsistante F si $\neg u \equiv v \in I$ (resp. $u \equiv v \in I$).

Remarque :

Cette notion d'inconsistance sera fréquemment utilisée par la suite pour "éliminer" certains calculs associés à un programme :

- soit parce qu'à la suite d'un test un tel calcul correspond à la branche de l'organigramme du programme que l'on ne suit pas. C'est le rôle de j et $\bar{j}$ d'exprimer ce choix. Ainsi pour un programme déterministe et une information initiale donnée, un calcul au plus conduit à une information inconsistante.
- soit parce qu'il faut exprimer le non défini au sens de [A]. Par exemple tout calcul Algol 68 qui contient le calcul associé à la déclaration d'identité : [*1:2*] *ent t = (4,2,1)* conduit à une information inconsistante [6.2.3].

Si on généralise la notion de langage au cas des mots infinis, un ensemble de calculs sur $\mathcal{M}od$ est un langage. En particulier suivant la manière dont il est défini on obtiendra un langage régulier (associé à un organigramme ou schéma de programme), un langage algébrique (associé à un programme contenant des appels de procédures sans résultat ni identificateurs locaux) etc... L'étude de ces ensembles de calculs est ébauchée par PAIR [C3].

3.3.2.2 Schémas de calculs :

A une procédure à paramètres doit correspondre une infinité de calculs. Une façon d'en rendre compte consiste à introduire la notion de schéma d'ensembles de calculs (c'est-à-dire d'ensembles de calculs dépendant de paramètres). Elle permettra de rendre compte également des propositions élémentaires définies de manière récursive (telle que l'affectation en Algol 68) et qui ne peuvent donc être formalisées par un simple schéma de modifications élémentaires.

Pour alléger le texte nous écrirons schéma de calculs plutôt que schéma d'ensembles de calculs.

Un tel schéma C sera une application de S^n (S est l'ensemble des schémas fonctionnels de la structure d'information) dans $\mathcal{P}(\mathcal{M}od^\infty)$ où $n \in \mathbb{N}$. Pour simplifier les définitions ultérieures on le prolonge à $S^* = \bigcup_{n \in \mathbb{N}} S^n$ de manière que $C(u_1,\ldots,u_p) = \emptyset$ si $p \neq n$. Plus précisément :

Définition 2 :

Un schéma de calculs C de la structure d'information est une application de S^* dans $\mathcal{P}(\mathcal{M}od^\infty)$ telle qu'il existe un entier naturel n unique vérifiant :

$p \neq n \Longrightarrow C(u_1,\ldots,u_p) = \emptyset$

n est le nombre d'arguments de C .

Tout schéma de calculs C peut être interprété comme un schéma de modifications Π

avec $\Pi(u_1,\ldots,u_n) = \bigcup_{i \in I} m_{n_i} \circ \ldots \circ m_{1_i}$

si $C(u_1,\ldots,u_n) = \bigcup_{i \in I} m_{1_i} \cdot \ldots \cdot m_{n_i}$

C et Π sont associés.

Exemples :

1) Dans la structure de liste, $C(s^k t,a,b) = \text{adj}(s^k t).\text{aff}(s^{k+1} t,a).\text{adj}(s^{k+1} t).\text{aff}(s^{k+1} t,b)$

[3.2.2 exemple] est un schéma de calculs 3-aire. Il peut être interprété comme le schéma de modifications :

$$\Pi(s^k t,a,b) = \text{aff}(s^{k+1} t,b) \circ \text{adj}(s^{k+1} t) \circ \text{aff}(s^{k+1} t,a) \circ \text{adj}(s^k t) .$$

2) Dans la structure de LS, à la procèdure f définie par :

proc f [*ent a, ent b, ent d*] : *début si* $a \leq b$ *alors début* $d := a$ *fin sinon début* $d := b$ *fin fin*

est associé le schéma de calcul $\tilde{f}$ défini par :

$\tilde{f}(a,b,d) = j(\textit{inf rep poss a rep poss b, vrai}).\text{affect}(d,a) \cup \bar{j}(\textit{ind rep poss a rep poss b, vrai}).\text{affect}(d,b)$

$\tilde{f}$ est interprétable comme le schéma de modifications Π :

$\Pi(a,b,d) = \text{affect}(d,a) \circ j(\textit{inf rep poss a rep poss b, vrai}) \cup \text{affect}(d,b) \circ \bar{j}(\textit{inf rep poss a rep poss b, vrai})$

Notation : $\mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty))$ représente l'ensemble des schémas de calculs de $\mathcal{L}$.

3.3.3 Système de calculs et opérations sur les schémas de calculs :

3.3.3.1 Système associé à un programme :

La définition de la syntaxe d'un langage de type algol est récursive ; elle peut être donnée, par exemple, par une C-grammaire (LS), ou par une double grammaire (Algol 68). La définition du schéma de calculs (en fait ensemble de calculs) associé à un programme P d'un tel langage sera aussi récursive ; elle résultera de l'association à P d'un système à point fixe.

Exemple :

Soit P le programme LS :

début *ent x, ent y ;* *lire(x) ;* *si x > 100 alors début y:=x-10 fin sinon début y:=91 fin fin*

(E_1) (E_2) (E_3)

A un tel programme on associe le système :

$$\mathcal{M}(P)\begin{cases}\tilde{P} = \tilde{E}_1 \,.\, \tilde{E}_2 \,.\, \tilde{E}_3 \\ \mathcal{M}(E_i) \quad \text{pour } i=1,2,3\end{cases}$$

Il est formé d'une équation liant l'inconnue $\tilde{P}$ associée à P aux inconnues associées aux phrases composantes "directes" de P; ainsi que des sous sytèmes associés à chacune de ces phrases.

Par exemple si C_1 est le bloc suivant *alors* dans E_3
et C_2 est le bloc suivant *sinon* on a :

$$\mathcal{M}(E_3)\begin{cases}\tilde{E}_3 = j(\textit{inf poss100 rep possx,vrai}).\tilde{C}_1 \cup \bar{j}(\textit{inf poss100 rep possx,vrai}).\tilde{C}_2 \\ \mathcal{M}(C_i) \quad i=1,2\end{cases}$$

où j et $\bar{j}$ sont définies en [3.3.2.1].

Ainsi définir le schéma de calculs associé à un programme quelconque d'un langage $\mathcal{L}_g$, c'est associer à chaque type de construction des phrases du langage un système à point fixe appelé système des calculs.

Notation : Si P est une phrase d'un langage $\mathcal{L}_g$, $\mathcal{M}(P)$ désigne le système de calculs associés à P ; $\tilde{P}$ est l'inconnue correspondant à P dans ce système.
Précisons la forme de ces systèmes :
En reprenant les notations de [0.1.2] et [0.1.3] soit $\mathcal{L} = \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty))$; nous considérons dans la suite le treillis $(\mathcal{L}, C)$ (où C est l'inclusion ensembliste

étendue à $\mathcal{F}[0.1.1.1]$). Il est clair que les ensembles de calculs sont des éléments de ce treillis (schémas 0-aires).

Les fonctions de base des systèmes de calculs, qui sont des systèmes algébriques sur $\mathcal{L}$, sont les opérations sur les schémas de calculs.

Définissons-les maintenant :

3.3.2.2 Opérations sur les schémas de calculs :

a) Concaténation :

la concaténation "." de $\mathcal{M}od^\infty$ s'étend en une application :

$$. : \begin{cases}\mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \times \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \longrightarrow \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \\ (C_1, C_2) \longmapsto C_1 \,.\, C_2\end{cases}$$

définie par :

$$C_1.C_2(u_1,\ldots,u_n) = \{\alpha.\beta \mid \alpha \in C_1(u_1,\ldots,u_n) \ ; \ \beta \in C_2(u_1,\ldots,u_n)\} \ .$$

(On peut toujours se ramener au cas où C_1 et C_2 ont même arité, quitte à les composer avec des projections).

$C_1.C_2$ peut être interprété comme $\Pi_2 \circ \Pi_1$ si Π_1 et Π_2 sont les schémas de modifications associés respectivement à C_1 et C_2.

Rappelons que cette notion permet de rendre compte de la notion de proposition sérielle.

Exemple :

Si P = *début* $D_1;D_2;\ldots;D_n;P_1;P_2;\ldots;P_m$ *fin* est un bloc quelconque de LS (les D_i étant des déclarations, les P_i des propositions) le système associé est donc :

$$\mathcal{M}(P)\begin{cases}\tilde{P} = \tilde{D}_1 \,.\, \tilde{D}_2 \,\ldots\, \tilde{D}_n \,.\, \tilde{P}_1 \,.\, \tilde{P}_2 \,\ldots\, \tilde{P}_m \\ \mathcal{M}(D_i) \quad i=1,n \\ \mathcal{M}(P_i) \quad i=1,m\end{cases}$$

b) Réunion :

$$\cup : \begin{cases}\mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \times \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \longrightarrow \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \\ (C_1, C_2) \longmapsto C_1 \cup C_2\end{cases}$$

avec $C_1 \cup C_2(u_1,\ldots,u_n) = C_1(u_1,\ldots u_n) \cup C_2(u_1,\ldots,u_n)$.

Cette réunion permet en particulier de rendre compte des propositions conditionnelles :

Exemple :

Soit P = *si* T *alors* P_1 *sinon* P_2 une proposition conditionnelle quelconque de LS.

On doit distinguer parmi les termes ceux qui possèdent des variables de ceux qui possèdent des constantes. Dans le premier cas ce n'est pas la variable qui est intéressante mais la valeur qu'elle repère (il faut un dereperage au sens de [A]),

$$\mathcal{M}(P)\begin{cases} \mathcal{P} = \mathcal{P} \,.\, \bigcup_{t \in \{reel,bool,ent\}} (j(type\ poss\ T, repère\ t)\ .\ (j(rep\ poss\ T, vrai)\ .\ \widetilde{C_1} \cup \bar{j}(rep\ poss\ T, vrai)\ .\ \widetilde{C_2}) \\ \qquad \cup\ j(type\ poss\ T, t)\ .\ (j(poss\ T, vrai)\ .\ \widetilde{C_1} \cup \bar{j}(poss\ T, vrai)\ .\ \widetilde{C_2}) \\ \mathcal{M}(T) \\ \mathcal{M}(C_i) \quad i=1,2 \end{cases}$$

c) Mélange :

Si la multiplication permet de rendre compte de la notion de proposition sérielle, le mélange permettra de formaliser celle de la proposition collatérale :

$$m : \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \times \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty)) \longrightarrow \mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty))$$

défini par :

1) Si α et β appartiennent à $\mathcal{M}od^\infty$ alors :

$$m(\alpha,\beta) = \{a_o.b_1 \ldots b_n.a_n \mid a_o.a_1 \ldots a_n = \alpha \ ;\ b_1 \ldots b_n = \beta\} \in \mathcal{P}(\mathcal{M}od^\infty)\ .$$

2) Si A et B appartiennent à $\mathcal{P}(\mathcal{M}od^\infty)$ alors :

$$m(A,B) = \{m(\alpha,\beta) \mid \alpha \in A\ ,\ \beta \in B\} \text{ si A et B} \neq \emptyset\ ;\ m(A,\Phi) = \Phi\ .$$

3) Si C_1 et $C_2 \in \mathcal{F}(S, \mathcal{P}(\mathcal{M}od^\infty))$:

$$\mathfrak{m}(C_1,C_2)(u_1,\ldots,u_n) = \mathfrak{m}(C_1(u_1,\ldots,u_n),\ C_2(u_1,\ldots,u_n)).$$

On peut vérifier que $\mathfrak{m}$ ainsi définie est associative et définir le mélange d'un nombre quelconque (fini) de schémas de calculs.

En particulier la notation : $\underset{1\leqslant i\leqslant n}{\mathfrak{m}}\ (C_i)$ sera préférée à $\mathfrak{m}(C_1,C_2,\ldots,C_n)$ et $\underset{1\leqslant i\leqslant n}{\mathfrak{m}}\ (C_i^1,\ldots,C_i^k)$ aura le même sens que $\underset{\substack{1\leqslant i\leqslant n\\ 1\leqslant j\leqslant k}}{\mathfrak{m}}\ (C_i^j)$ ou que $\underset{1\leqslant i\leqslant n}{\mathfrak{m}}\ (\underset{1\leqslant j\leqslant k}{\mathfrak{m}}\ (C_i^j))$.

(On traduit que deux propositions s'élaborent collatéralement en exprimant qu'il n'y a aucun ordre à priori entre l'élaboration de l'un des constituants de cette proposition et un constituant de l'autre).

Exemple :

Soit P = $D_1, D_2, \ldots, D_n$ une déclaration collatérale de LS,

le système de calculs associé est donc :

$$\mathcal{M}(P)\begin{cases} \mathcal{P} = \underset{1\leqslant i\leqslant n}{\mathfrak{m}}\ (D_i) \\ \mathcal{M}(D_i) \quad i=1,\ldots,n \end{cases}$$

d) Substitution :

L'introduction de schémas de calculs permet de rendre compte des procédures dépendant de paramètres. Il faut introduire un outil traitant le cas des identificateurs locaux à une procédure. Pour une procédure récursive par exemple, un identificateur local représente un objet différent pour chaque réincarnation de cette procédure. Intuitivement, à chaque appel de la procédure on transforme les identificateurs locaux en identificateurs qui n'ont jamais été utilisés avant cet appel et qui ne seront plus utilisés après (on peut concevoir lors du premier appel de transformer tout identificateur local x en x', lors du deuxième appel en x'' etc... en faisant l'hypothèse qu'à l'extérieur de la procédure, x', x'' ... n'ont aucune occurence). Nous formaliserons cette idée en associant à tout appel A_p de procédure une application τ_{A_p} de $\mathcal{F}(S^*, \mathcal{P}(\mathcal{M}od^\infty))$ dans lui-même. Plus précisément un symbole fonctionnel 2-aire *sub* de la structure d'information permet d'exprimer la "transformation" de tout identificateur local x à la procédure P en un nouvel identificateur *sub* A_p x lors de l'appel A_p de P. τ_{A_p} est alors défini par :

$$\tau_{A_p}(\Pi(u_1,\ldots,u_n)) = \Pi(sub\ A_p\ u_1,\ldots,sub\ A_p\ u_n)$$

pour tout schéma de modifications élémentaires Π à n arguments. τ_{A_p} se prolonge alors immédiatement aux schémas de calculs si on impose que :

(1) $\tau_{A_p}(C_1.C_2) = \tau_{A_p}(C_1)\ .\ \tau_{A_p}(C_2)$

(2) $\tau_{A_p}(C_1 \cup C_2) = \tau_{A_p}(C_1) \cup \tau_{A_p}(C_2)$

pour tous schémas de calculs C_1, C_2

(en particulier on déduit de (1), (2) et c) que

$$\tau_{A_p}(m(C_1,C_2)) = m(\tau_{A_p}(C_1), \tau_{A_p}(C_2)).$$

Remarque :

Un problème subsiste quant aux identificateurs non locaux : il est bien évident qu'une substitution doit les laisser invariants.

Deux solutions peuvent être envisagées :

- ne pas admettre d'identificateurs globaux dans une procédure d'un langage $\mathcal{L}_g$, ou du moins dans une procédure du langage pivot associé
- imposer certaines propriétés au symbole *sub* : *sub* A_p $x \equiv x$ si x est un identificateur global à la procédure dont l'appel est A_p.

e) L'opérateur λ :

L'association d'un système à une phrase ne contenant pas de paramètres formels (n'apparaissant pas dans un corps de procédure par exemple) est relativement aisée, mais dès qu'apparaissent de tels paramètres un certain nombre de problèmes se posent.

Par exemple dans un système algébrique il faut que les fonctions figurant à gauche et à droite du signe = de toute équation "dépendent des mêmes variables" (qui sont ici des paramètres formels, c'est-à-dire des identificateurs). Or lorsqu'on défini le système associé à une procédure, il se peut que des "variables" du membre gauche de l'équation n'apparaissent plus directement dans le deuxième membre mais par l'intermédiaire de schémas fonctionnels.

On résout le problème en introduisant l'opérateur d'abstraction λ qui permet de définir correctement des schémas de calculs de la même manière que l'on définit les ensembles de calculs [L1].

Exemple :

Considérons la procédure f de LS définie par :

proc f [*réel* z_1, *réel* z_2, *réel* z_3] : *début si* $z_1 \leqslant z_2$ *alors début* $z_3 := z_2 - z_1$ *fin sinon début* $z_3 := z_1 - z_2$ *fin fin* .

On peut définir le schéma de calculs $\tilde{f}$ associé à f par :

$$\begin{cases} \tilde{f} = \lambda z_1\ \lambda z_2\ \lambda z_3\ (\mathrm{j}(\mathit{inf\ rep\ poss}\ z_1\ \mathit{rep\ poss}\ z_2, \mathit{vrai})\ .\ \widetilde{P_1} \cup \\ \qquad\qquad \mathrm{j}(\mathit{inf\ rep\ poss}\ z_1\ \mathit{rep\ poss}\ z_2, \mathit{vrai})\ .\ \widetilde{P_2}) \\ \mathcal{M}_6(\lambda z_1\ \lambda z_2\ \lambda z_3\ \widetilde{P_i}) \qquad i=1,2 \end{cases}$$

Remarque :

Pour que la définition de λ soit correcte, il est nécessaire que z_1, z_2, z_3 soient des variables du système formel de la structure d'information. Rappelons [2.5.3.2] que de telles variables sont associées à chaque ensemble de symboles

fonctionnels S_j $(1 \leqslant j \leqslant m)$.

λ vérifie les propriétés suivantes :

Soient $z_1, \ldots, z_n \in \mathcal{Z} = \bigcup_{1 \leqslant i \leqslant m} Z_i$, $\pi_1, \pi_2 \in \mathcal{M}od$ et $\mathcal{C}_{A_p}$ une substitution.

- $\lambda z_1 \ldots \lambda z_n\ \pi_1 \,.\, \pi_2 = \lambda z_1 \ldots \lambda z_n\ \pi_1 \,.\, \lambda z_1 \ldots \lambda z_n\ \pi_2$
- $\lambda z_1 \ldots \lambda z_n\ \pi_1 \cup \pi_2 = \lambda z_1 \ldots \lambda z_n\ \pi_1 \cup \lambda z_1 \ldots \lambda z_n\ \pi_2$
- $\mathcal{C}_{A_p}(\lambda z_1 \ldots \lambda z_n\ \pi) = \lambda z_1 \ldots \lambda z_n\ \mathcal{C}_{A_p}(\pi)$.

f) L'opérateur de troncation $\mathcal{S}$:

Depuis quelques années un certain nombre d'auteurs dénoncent la présence des instructions de saut dans les langages du programmation usuels, en lui reprochant de "casser" la structure logique des algorithmes. En particulier DIJKSTRA dans [S 17] propose de supprimer les sauts et de ne conserver que les procédures et les instructions d'itérations (boucles). L'actuel essor de la méthode dite de "programmation structurée" ne peut que fortifier ce point de vue.

Dans notre formalisation de la sémantique d'un langage, pas plus que les sauts nous n'accepterons d'instructions d'itération. Mais il a déjà été montré (VAN WIJNGAARDEN [S 56], LANDIN [S 31], HOARE [S 24]) que toute instruction de saut peut être remplacée par un appel de procédure. Cette procédure doit cependant présenter la caractéristique suivante : sa dernière instruction exécutable doit être une instruction d'arrêt du programme complet, et non pas un retour à l'instruction suivant l'appel.

Pour cela nous introduirons une instruction élémentaire "*stop*" dans tout langage pivot dont nous définirons la sémantique, la modification élémentaire $\widetilde{stop}$ associée ne sera pas définie : une fonction de base $\mathcal{S}$ du système de calculs $\mathcal{M}_6(P)$ associé à un programme P quelconque a pour rôle de tronquer à partir de $\widetilde{stop}$ tout (schéma de) calcul la contenant.

$\mathcal{S}$ est définie par :

$\mathcal{S}(\pi) = \pi$ pour toute modification élémentaire π différente de *stop*

$\mathcal{S}(C_1 \,.\, stop \,.\, C_2) = \mathcal{S}(C_1)$ pour tous calculs C_1, C_2

$\mathcal{S}(C_1 \cup C_2) = \mathcal{S}(C_1) \cup \mathcal{S}(C_2)$.

g) Conclusion :

On vérifie que les fonctions de base définissant le système de calculs $\mathcal{M}_6(P)$ associé à un programme P sont continues dans le treillis $(\mathcal{F}(S^*, \mathcal{F}(\mathcal{M}od^\infty)), \mathcal{C})$.

(P) admet donc un point fixe minimal [0.1.1.3] ; le schéma de calculs (en fait ensemble de calculs) défini par P est la composante correspondante dans cette solution minimale.

3.4 Axiomes associés à une proposition : système des accès :

Nous avons déjà remarqué que la frontière séparant les notions d'instructions et d'expressions n'est pas très nette ; c'est en particulier le cas en Algol 68. Si nous rendons compte des actions décrites par une proposition en lui associant un système de calculs, il faut rendre compte de la notion de valeur de cette proposition et pour cela nous lui associerons un accès dans la structure d'information.

Cette association se fera par la définition d'axiomes, ou plus souvent d'un système d'axiomes caractérisant cet accès ; comme le système de calculs il sera, en général, défini de manière récursive : c'est le système d'accès.

Notation : Si P est une phrase d'un langage $\mathcal{L}_g$, on note $\mathfrak{V}(P)$ le système d'accès associé à P et il n'y a aucune ambiguïté à noter encore P l'accès associé à cette phrase dans la structure d'information.

Exemple :
Soit P la proposition conditionnelle de LS :

si $d \leqslant 0$ *alors* $\overbrace{\textit{début } x := d \textit{ ; vrai fin}}^{P_1}$ *sinon* $\overbrace{\textit{début } x := -d \textit{ ; faux fin}}^{P_2}$

$$\mathfrak{V}(P)\begin{cases} \textit{poss } P \equiv \textit{cond}(\textit{inf rep poss } d \textit{ poss } 0, \textit{vrai}, \textit{poss } P_1, \textit{poss } P_2) \\ \mathfrak{V}(P_1) \\ \mathfrak{V}(P_2) \end{cases}$$

(les parenthèses dans *cond* [2.5.2 exemple 2] ne servent qu'à améliorer la compréhension)
avec, par exemple, $\mathfrak{V}(P_1)$ {*poss* $P_1 \equiv$ *vrai* .

Remarque :
La notion d'accès associé à une phrase peut se concevoir de deux manières :

i) On suppose à priori que toutes les phrases du langage $\mathcal{L}_g$ sont des symboles 0-aires de la structure d'information de $\mathcal{L}_g$. L'ensemble des axiomes des systèmes $\mathfrak{V}(P)$ fait alors partie de l'ensemble X_p des axiomes propres de la structure (voir en particulier [5.1] remarque 1).

ii) Si par contre on exclut ces axiomes de X_p on est conduit à considérer le système d'axiomes $\mathfrak{V}(P)$, associé à un programme P, comme définissant une extension de la structure initiale [2.5] par adjonction d'accès nouveaux.

La deuxième interprétation permet peut être de concevoir notre démarche de manière intuitive :

- les axiomes propres X_p de la structure définissent un cadre pour les informations qui représenteront les états de mémoire
- les axiomes de $\mathfrak{V}(P)$ correspondent à la phrase statique du traitement du programme P. Toutes les "relations" qui y sont définies sont connues à la compilation
- le système $\mathcal{M}_0(P)$ décrit l'exécution du programme : phrase dynamique.

Le choix entre i) et ii) est affaire de goût, il est certainement guidé par un souci de simplicité de description.
Nous utiliserons ii) dans la définition de la sémantique de LS en [3.5] et i) dans la formalisation d'Algol 68.

Notation : $\mathfrak{S}(P)$ représentera le double système de calculs ($\mathcal{M}_0(P)$) et d'accès ($\mathfrak{V}(P)$) associé à la phrase P .

3.5 Sémantique de LS :

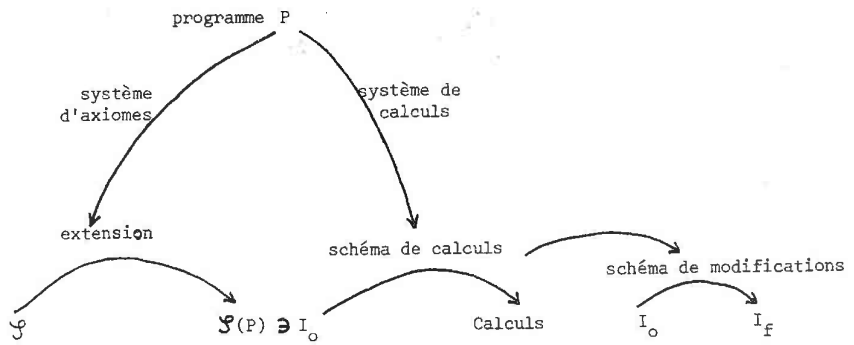
3.5.1 Introduction :

Rappelons brièvement la méthode proposée pour définir la sémantique d'un langage de programmation $\mathcal{L}_g$:

a) définition de la structure d'information associée à $\mathcal{L}_g$
b) définition du langage pivot $\mathcal{L}_{g_o}$ et du procédé de traduction de $\mathcal{L}_g$ dans $\mathcal{L}_{g_o}$
c) association d'un double système de calculs et d'accès à tout type de phrase défini par la syntaxe.

Dans LS le système d'accès associé à une phrase P définira l'accès P à l'aide d'axiomes nouveaux [3.4 remarque ii)] .

On peut schématiser cette démarche de la façon suivante :



où $\mathcal{S}$ est la structure d'information, I_o l'information initiale (contenant les données du programme si on formalise les ordres de lecture) et I_f est une information de $\mathcal{S}(P)$: information finale.
(Dans le cas d'un programme indéterministe I_f devrait être remplacée par un ensemble d'informations).

Les étapes a) et b) ont été effectuées pour LS de manière informelle en [3.2.2] et [3.2.3] . Décrivons maintenant c).

3.5.2 Doubles systèmes associés aux phrases de LS :

On traite tour à tour les différents types de phrases définis par la syntaxe de LS [3.2.1] :

a) Si P est le bloc *début* $D_1;D_2;\ldots;D_n;P_1;P_2;\ldots;P_m$ *fin*
$\mathcal{M}(P)$ est défini en [3.3.2.2 a) exemple]
$\mathcal{V}(P)$ est donné par :

$$\mathcal{V}(P) \begin{cases} \textit{poss } P \equiv \textit{poss } P_m \\ \mathcal{V}(P_m) \end{cases}$$

celle de P_m (ce peut être une variable si P_m est un identificateur défini par une déclaration de variable).

b) Déclarations :

b1) Si P est la déclaration collatérale $D_1,D_2,\ldots,D_n$:
$\mathcal{M}(P)$ est définie en [3.3.2.2 c) exemple] et comme à une déclaration n'est associée aucune valeur $\mathcal{V}(P) = \emptyset$ (il ne s'agit pas non plus d'une instruction auquel cas on aurait *type poss* P ≡ *neutre*)

b2) Si P est la déclaration de variable $\underline{t}\ x$ où $\underline{t}$ est un type et x un identificateur, le double système la formalisation est donné par :

$$\mathcal{S}(P) \begin{cases} \mathcal{M}(P) \begin{cases} \hat{P} = \text{var}(\underline{t},x) \end{cases} \\ \mathcal{V}(P) = \emptyset \end{cases}$$

où var est le schéma de modifications élémentaires associé aux déclarations de variables [3.2.2].

b3) Si P est la déclaration de constante x = E où x est un identificateur et E une proposition, le double système associé est donné par :

$$\mathcal{S}(P) \begin{cases} \mathcal{M}(P) \begin{cases} \hat{P} = \text{const}(x,E) \end{cases} \\ \mathcal{V}(P) = \emptyset \end{cases}$$

const est défini en [3.2.2].

b4) Si P est la déclaration de procédure *proc* $f\ [\underline{t}_1 z_1,\ldots,\underline{t}_n z_n]$: P_1
où P_1 est un bloc dans lequel apparaissent les n paramètres formels $z_1,\ldots,z_n$.
Intuitivement une déclaration de procédure doit "créer" les objets utilisés dans un appel :

- schéma de calculs formalisant les actions résultant de l'activation de la procédure
- symbole fonctionnel permettant d'accéder au résultat de la procédure (exemple : f en [2.5.3.1 exemple 2]).

LS ne contient que des procédures sans résultat, le schéma de calculs $\hat{f}$ associé à f sera défini de manière naturelle par :

$$\mathcal{M}(P) \begin{cases} \hat{P} = \Lambda \\ \hat{f} = \lambda z_1 \,.\, \lambda z_2 \ldots \lambda z_n \,.\, \widetilde{P_1} \\ \mathcal{M}(\widetilde{P_1}) \end{cases}$$

On exprime en particulier que l'élaboration de la déclaration P ne comporte aucune action (Λ est l'élément neutre de $\mathcal{M}od^{\infty}$).
Comme f est une procédure sans résultat :

$$\mathcal{V}(P) = \bar{\emptyset} \ .$$

Remarque : Si f était une procédure à résultat on définirait le symbole nouveau n-aire f dans $\mathcal{V}(P)$.

c) Propositions :

c1) Si P est la proposition conditionnelle *si* T *alors* P_1 *sinon* P_2 , $\mathcal{M}(P)$ est défini en [3.3.2.2 b)] et $\mathcal{V}(P)$ est donné par :

$$\mathcal{V}(P)\begin{cases} \{\textit{type poss } T \equiv \textit{repère } \underline{t} \supset \textit{poss } P \equiv \textit{cond}(\textit{rep poss } T, \textit{vrai}, \textit{poss } P_1, \textit{poss } P_2) \mid \\ \quad \underline{t} \in \{\underline{\textit{réel}}, \underline{\textit{ent}}, \underline{\textit{bool}}\} \\ \{\textit{type poss } T \equiv \underline{t} \supset \textit{poss } P \equiv \textit{cond}(\textit{poss } T, \textit{vrai}, \textit{poss } P_1, \textit{poss } P_2) \mid \\ \quad \underline{t} \in \{\underline{\textit{réel}}, \underline{\textit{ent}}, \underline{\textit{bool}}\} \\ \mathcal{V}(P_1) \\ \mathcal{V}(P_2) \\ \mathcal{V}(T) \end{cases}$$

Selon que la valeur possédée par T (ou repérée par la variable possédée par P) est *vrai* ou *faux*, la valeur de P est celle de P_1 ou celle de P_2 .

c2) Si P est l'affectation x := T le double système associé est donné par :

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(P) \begin{cases} \tilde{P} = \tilde{T} \,.\, \text{affect}(x, \textit{poss } T) \\ \mathcal{M}_0(T) \end{cases} \\ \mathcal{V}(P) \begin{cases} \textit{poss } P \equiv \textit{poss } x \\ \mathcal{V}(T) \end{cases} \end{cases}$$

c3) Si P est l'appel de procédure $f(x_1, x_2, \ldots, x_n)$ le double système associé est :

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(P) \begin{cases} \tilde{P} = \mathcal{C}_p(\tilde{f}(x_1, \ldots, x_n) \end{cases} \\ \mathcal{V}(P) \begin{cases} \textit{type poss } P \equiv \textit{neutre} \end{cases} \end{cases}$$

Rappelons que $\mathcal{C}_p$ est une substitution des variables locales de la procédure [3.3.2.2 d)].

d) Termes :

d1) Constantes :

Le type et la valeur associée sont définis dans la structure d'information. Aussi pour toute constante P

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(P) \begin{cases} \tilde{P} = \Lambda \end{cases} \\ \mathcal{V}(P) = \emptyset \end{cases}$$

d2) Si P est un identificateur, il est défini par une déclaration de variables ou de constante qui précise : type et valeur. Aussi :

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(P) \begin{cases} \tilde{P} = \Lambda \end{cases} \\ \mathcal{V}(P) = \emptyset \end{cases}$$

d3) Si P est une formule les parenthèsages imposés ne permettent que l'une des formes :

α) T_1 op T_2

β) op T_1

α) les seules actions associées à cette formule sont les élaborations de T_1 et T_2 :

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(D) \begin{cases} \tilde{P} = m(\widetilde{T_1}, \widetilde{T_2}) \\ \mathcal{M}(T_i) \quad i=1,2 \end{cases} \\ \mathcal{V}(P) \begin{cases} \{\textit{type poss } T_1 \equiv \textit{repère}^{(\varepsilon)} \underline{t} \wedge \textit{type poss } T_2 \equiv \textit{repère}^{(\varepsilon')} \underline{t} \\ \quad \textit{poss } P \equiv \overline{\text{op}}\ \textit{rep}^{(\varepsilon)} \textit{poss } T_1\ \textit{rep}^{(\varepsilon')} \textit{poss } T_2 \mid \underline{t} \in \{\underline{\textit{reel}}, \underline{\textit{ent}}, \underline{\textit{bool}}\}; \\ \quad \varepsilon = 0,1,\ \varepsilon' = 0,1\} \\ \mathcal{V}(T_i) \quad i=1,2 \end{cases} \end{cases}$$

On exprime d'une part l'élaboration collatérale de T_1 et T_2 , d'autre part que la valeur de la formule se déduit de celles de T_1 et T_2 : on note $\overline{\text{op}}$ le symbole fonctionnel 2-aire associé à op (par exemple *moins* est associé à "-", *plus* à "+" etc...) ; enfin $\textit{repère}^{(\varepsilon)}$ représente *repère* si $\varepsilon = 1$ et le mot vide sinon.

Pour être tout-à-fait correct il faudrait distinguer, suivant l'opérateur, différents types de formules et effectuer un contrôle sur le type des opérandes.

β)

$$\mathcal{S}(P)\begin{cases} \mathcal{M}_0(P) \begin{cases} \tilde{P} = \widetilde{T_1} \\ \mathcal{M}(T_1) \end{cases} \\ \mathcal{V}(P) \begin{cases} \{\textit{type poss } T_1 \equiv \textit{repère}^{(\varepsilon)} \underline{t} \supset \textit{poss } P \equiv \overline{\text{op}}\ \textit{rep}^{(\varepsilon)} \textit{poss } T_1 \mid \\ \quad t \in \{\underline{\textit{réel}}, \underline{\textit{ent}}, \underline{\textit{bool}}\}\} \\ \mathcal{V}(T_1) \end{cases} \end{cases}$$

Les remarques sont analogues à celles de α).

enfin le cas où le terme est un bloc est traité en a) ce qui termine la brève définition de la sémantique de LS. Nous allons maintenant formaliser celle d'Algol 68.

4

La structure d'information d'Algol 68

4. LA STRUCTURE D'INFORMATION D'ALGOL 68

Nous allons appliquer la méthode précèdente au langage Algol 68 (plus précisément ([5]) à un sous langage d'Algol 68).

Brièvement, rappelons qu'il s'agit de formaliser (avec la terminologie de [A]) :

L'association à tout programme Algol 68 d'un ensemble d'actions formelles (calculs) qui sera l'élaboration du programme ; actions qui portent sur un ensemble d'objets entre lesquels, à tout moment, certaines relations peuvent être vraies (information) [A-2.2] .

En reportant au chapitre [6] la définition des calculs associés à un programme nous allons décrire, dans la suite de ce chapitre, d'une part le système formel $\mathcal{F}$ associé au langage Algol 68 ([4.1]), d'autre part l'ensemble $\mathcal{M}od$ des modifications élémentaires associé à ce système ([4.2]).

Au chapitre [5] on définit sommairement un langage noyau d'Algol 68 qui permet de simplifier la forme de certaines constructions.

Dans ce chapitre et dans les deux suivants, $\mathcal{S} = (\mathcal{F}, \mathcal{M}od)$ désignera la structure d'information Algol 68.

4.1 Le système formel d'Algol 68

4.1.1 Introduction :

Les objets et relations sur lesquels portent les traitements d'un calculateur hypothètique sont définis en [A-2] , aussi la définition du système formel $\mathcal{F}$ va-t-elle reposer sur ce chapitre du rapport Algol 68.

En [4.1.2] et [4.1.3] sont décrits les schémas fonctionnels de $\mathcal{F}$ qui formalisent ces notions d'objets et relations de [A-2] ; ils seront, pour la plupart, déduits directement des paragraphes [A-2.2.1] à [A-2.2.4] .

En [4.1.4] on définit l'ensemble A des formules atomiques et en [4.1.5] on introduit les axiomes spécifiques de $\mathcal{F}$ qui formalisent les propriétés des relations entre objets.

De l'étude menée en [2] , il ressort [2.3.2.7] que le système formel associé à une structure d'information dépend de quatre paramètres :

i) l'ensemble L des symboles fonctionnels

ii) la puissance m de ces symboles (c'est le nombre d'ensembles qu'il faut introduire pour interpréter les éléments de L)

iii) la fonction profil pl

iv) l'ensemble X_p des axiomes propres de la structure

(la donnée des trois premiers paramètres permet de-déterminer successivement :

- l'ensemble S des schémas fonctionnels de $\mathfrak{F}$ [2.3.2.2] et en particulier l'ensemble S_k des schémas fonctionnels de type k $(1 \leqslant k \leqslant m)$
- l'ensemble A des formules atomiques [2.3.2.3]
- l'ensemble F des formules

enfin l'ensemble X_p réuni à l'ensemble X_ℓ des axiomes logiques définit les axiomes de $\mathfrak{F}$).

Notations : Il est agréable pour la suite d'introduire des connecteurs logiques n-aires $(n \geqslant 2)$: $\bigwedge_{1 \leqslant k \leqslant n}$ et $\bigvee_{1 \leqslant k \leqslant n}$ définis par récurrence :

- si $n = 2$ alors $\bigwedge_{1 \leqslant k \leqslant 2} \varphi_k$ remplace $\varphi_1 \wedge \varphi_2$ et $\bigvee_{1 \leqslant k \leqslant 2} \varphi_k$ remplace $\varphi_1 \vee \varphi_2$
- si $n > 2$ alors $\bigwedge_{1 \leqslant k \leqslant n} \varphi_k$ remplace $\left(\bigwedge_{1 \leqslant k \leqslant n-1} \varphi_k\right) \wedge \varphi_n$ et $\bigvee_{1 \leqslant k \leqslant n} \varphi_k$ remplace $\left(\bigvee_{1 \leqslant k \leqslant n-1} \varphi_k\right) \vee \varphi_n$.

4.1.2 Définition de l'ensemble L des symboles fonctionnels

$L = \bigcup_{q \geqslant o} L_q$ où L_q est l'ensemble des symboles fonctionnels q-aires ; les ensembles L_q sont disjoints deux à deux $(q \geqslant o)$.

4.1.2.1 Définition de L_o :

Tout objet interne [A-2.2.1] est l'image d'un objet externe par la composée d'un certain nombre de fonctions d'accès ; c'est-à-dire en termes de schémas fonctionnels : tout schéma fonctionnel se termine par un symbole 0-aire dont l'interprétation sera un objet externe. L'ensemble L_o des symboles 0-aires sera donc essentiellement constitué d'éléments dont les interprétations seront des objets externes (ensemble de définition de la fonction possède [A-2.2.2 d)] : ce sont les objets directement accessibles par le programmeur.

On trouvera de plus dans L_o des ensembles non explicitement décrits dans le rapport Algol 68 et qui sont introduits pour les besoins de notre formalisation. Les différents sous ensembles de L_o sont donc :

a) L_o^{id} (resp. L_o^{op}) ensemble (dénombrable) des symboles 0-aires pouvant être interprétés comme des identificateurs (resp. des indicateurs d'opérations) utilisables dans un programme Algol 68

Dans la suite de [4] nous abrègerons en général "est le symbole pouvant être interprété comme" en "est".

b) $L_o^{re} = \bigcup_{k \geqslant 1} L_o^{rek}$ (resp. $L_o^{ent} = \bigcup_{k \geqslant 1} L_o^{ent\,k}$) ; L_o^{rek} (resp. $L_o^{ent\,k}$) est l'ensemble des notations de réels (resp. d'entiers) de longueur k [A-2.3.3.1. b)] .

c) L_o^{car} (resp. L_o^{bool}) est l'ensemble des notations de caractères (resp. de booléens) ; en particulier $L_o^{bool} = \{\underline{vrai}, \underline{faux}\}$

d) $\{\underline{nil}\}$ [A-2.2.3.5 a)]

e) L_o^{ind} est l'ensemble des indicateurs de mode [A-4.2 b)]

f) L_o^{prim} est l'ensemble des déclarateurs primitifs de mode algol 68 :

$$L_o^{prim} = \{\underline{ent}, \underline{reel}, \underline{bool}, \underline{car}, \underline{proc}\}$$ [A-7.1.2 c)]

(on inclut *proc* dans cet ensemble comme déclarateur du mode procedure sans paramètre ni résultat).

Nous définirons un symbole unaire "*indique*" qui sera interprété comme une fonction associant à un déclareur de mode le mode qu'il spécifie [A-7.1].

Enfin nous introduirons un certain nombre de symboles fonctionnels (*repère*, *structure* $x_1 \ldots x_n$, *long* etc...) qui permettront d'obtenir par combinaison entre eux et les schémas fonctionnels "*indique* x" (x appartenant à $L_o^{prim} \cup L_o^{ind}$) des schémas fonctionnels interprétables comme des modes quelconques d'Algol 68 et tous les modes pourront s'obtenir de cette manière.

Remarque 1 :

L'introduction de "l'ensemble des modes" dans la structure d'information est justifiée par le fait que l'égalité des modes doit parfois être testée de manière dynamique (à l'exécution) par la relation de conformité ; ce qui pose certains problèmes dans le cas des modes récursifs [6.3.4] .

Remarque 2 :

Pour tout x appartenant à l'un des ensembles définis de a) à d), le schéma fonctionnel *poss* x aura un sens [4.1.3.2] (où "*poss*" peut être interprété comme la fonction "possède" [A-2.2.2 d)]). De plus pour rendre compte de l'élaboration d'un programme on est amené à "définir" le symbole fonctionnel *poss* [4.1.2.2] sur d'autres objets que ceux des ensembles précèdents, par exemple sur des éléments représentant les objets externes du type "proposition fermée", "proposition conditionnelle" etc...

De manière analogue un mode peut être spécifié par un déclarateur primitif, par un indicateur mais aussi par un déclarateur non primitif (production terminale de "déclarateur VIFFEL de MODE" où "MODE" est différent de "PRIMITIF", par exemple

struct (reel x, ent y), long long ent etc...).

Aussi lors de l'élaboration d'un programme, sera-t-on amené à définir le symbole "*indique*" sur d'autres objets que ceux de $L_o^{prim} \cup L_o^{ind}$.

Nous avons remarqué [3.4] que plusieurs possibilités sont envisageables :

- Associer à chaque programme P une extension $\mathcal{S}(P)$ de la structure obtenue par adjonction d'un certain nombre d'accès nouveaux : chaque accès est associé biunivoquement à une phrase du programme P , les axiomes qui le caractérisent définissent *poss* E (si E est une proposition) ou *indique* E (si E est un déclarateur).

- Supposer que la structure $\mathcal{S}$ contient toutes les phrases du langage, ainsi que tous les axiomes caractérisant ces phrases (qui définissent essentiellement *poss* et *indique*).

C'est cette dernière solution qui est retenue dans notre formalisation :

g) L_o^{ph} (resp. L_o^{nprim}) est l'ensemble des propositions Algol 68 qui ne sont pas réduites aux éléments des ensembles définis de a) à d) (resp. est l'ensemble des déclarateurs de modes non primitifs).

Remarquons qu'en fait les éléments de L_o^{ph} (resp. L_o^{prim}) sont des phrases du langage noyau défini en [5] .

Remarque 3 : On peut remarquer le parallèlisme entre *poss* (resp. $L_o^{id} \cup L_o^{op} \cup L_o^{re} \cup L_o^{ent} \cup L_o^{car} \cup L_o^{bool}$, L_o^{ph}) et *indique* (resp. $L_o^{prim} \cup L_o^{ind}$, L_o^{nprim}) .

h) $\{\theta\}$ est la portée de tout un programme.

Nous définirons deux symboles fonctionnels *desc* et *succ* (descendant et successeur) qui permettront de définir un ensemble de schémas fonctionnels interprétables comme un ensemble de "portées".

Un autre symbole *ppq* (plus petit que) sera introduit et pourra être interprété comme une relation d'ordre partiel sur l'ensemble des "portées".

i) Dans un simple but d'allègement des notations, nous introduirons les ensembles de symboles 0-aires suivants :

- $\{vrai, faux, nil\}$ qui sont les valeurs des exemplaires "possédés" par *vrai*, *faux* et *nil*.

- $L_o^{mo} = \{entier, booleen, reel, caractère, proc_o\}$ est l'ensemble des modes primitifs et du mode procédure sans paramètre à résultat neutre ; ce sont les objets "indiqués" par L_o^{prim} .

C'est-à-dire que *vrai* (resp. *faux, nil*) devra être considéré comme une abréviation de *valeur poss vrai* (resp. *valeur poss faux, valeur poss nil*) et que *entier* (resp. *booleen, reel, caractère, proc*$_o^1$) représente *indique ent* (resp. *indique bool, indique reel, indique car, indique proc*).

4.1.2.2 Définition de L_1 :

L_1 est l'ensemble de symboles fonctionnels unaires. Un symbole fonctionnel n-aire ($n \geqslant 1$) contiendra éventuellement plusieurs indices ; l'indice se trouvant en haut et à droite sera le $n+1^{ième}$ composant du profil de ce symbole, c'est-à-dire le numéro du type qui est "l'ensemble d'arrivée de ce symbole" (exemple : $poss^3$ est le symbole fonctionnel à "valeurs" dans S_3 [2.3.2.2] ; les autres indices sont explicités dans chaque cas particulier.

L_1 est formé de :

- $poss^i$ ($i = 1$ ou $3 \leqslant i \leqslant 11$) symboles interprétables comme la "fonction possède" [A-2.2.2 d)] ; il y a plusieurs tels symboles pour rendre compte des différents domaines d'arrivée possible (qui doivent être disjoints par définition de la notion de profil)

- rep^i ($3 \leqslant i \leqslant 10$) symboles interprétables comme la fonction repère [A-2.2.2 h)]

- $valeur^i$ ($12 \leqslant i \leqslant 19$) qui associent à un exemplaire de valeur ("objet interne") cette valeur

- ch_x^i ($3 \leqslant i \leqslant 10$ et $x \in L_o^{id}$) sélecteurs de champ de tout exemplaire de valeur structurée dont x est un champ [A-2.2.3.2]

- a_i^j (resp. b_i^j, s_i^k, t_i^k) (avec $i \geqslant 1$; $j = 1$ ou 15 ; $k = 15$ ou 24) est la fonction qui associe à chaque exemplaire de valeur multiple de dimension supérieure ou égale à i la valeur de la $i^{ème}$ borne inférieure (resp. de la $i^{ème}$ borne supérieure, du $i^{ème}$ état inférieur, du $i^{ème}$ état supérieur) [A-2.2.3.3 b)] .

De plus nous verrons [5.2] que tout déclareur de valeur multiple D se présente sous la forme $[A_1,\Gamma_1 : B_1,\Delta_1 , \ldots , A_n,\Gamma_n : B_n,\Delta_n]\ D_1$ dans le langage noyau où A_i (resp. B_i) est un tertiaire ou vide, Γ_i (resp. Δ_i) prend la valeur 0 (resp. 2,1) si la $i^{ème}$ borne inférieure (resp. supérieure) est flexible (resp. adlib, fixe).

Nous introduirons alors des axiomes caractérisant les schémas fonctionnels a_i D (resp. b_i D , s_i D , t_i D) à l'aide de A_i (resp. B_i,Γ_i,Δ_i) [6.2.2.1 f)] pour

tout déclareur D.

- *dim* est la fonction associant sa dimension à tout exemplaire de valeur multiple. Comme les a_i (resp. b_i, s_i, t_i), *dim* pourra être prolongé à un déclareur de valeur multiple [6.2.2.1 b)]

- *sous flex* fonction qui associe *vrai* à chaque composant d'un exemplaire d'une valeur multiple [A-2.2.2 k)] si la valeur est flexible. (Nous dirons qu'une valeur multiple est flexible si l'un de ses états [A-2.2.3.3 b)] est égal à 0

- $equi_1$ est la fonction associant à tout entier un réel de même numéro de longueur [A-2.2.3.1 d)]

- $equi_2^j$ avec j = 15 (resp. 16) est la fonction associant à tout entier (resp. réel) de longueur n l'entier (resp. le réel) équivalent de longueur n+1 [A-2.2.3.1 d)]

- $equi_3$ est la fonction associant à tout caractère l'entier équivalent [A-2.2.3.1 d)]

- $\overline{equi_1}$ est la fonction réciproque de $equi_1$

- $\overline{equi_2^j}$ avec j = 15 (resp. 16) est la fonction réciproque de $equi_2^j$

- *indique* associe à un déclareur de mode le mode qu'il spécifie [4.1.2.1 f)]

- *mode* associe à toute valeur interne son mode [A-2.2.2 h,i]

- *long* est la fonction associant au mode μ le mode "long μ"

- $procn_1$ est la fonction associant au mode μ le mode "procédure avec paramètre à résultat neutre"

- $procr_1$ est la fonction associant au mode μ le mode "procédure à résultat μ"

- *rang* fonction qui associe au mode μ le mode "rang de μ"

- *repère* fonction qui associe au mode μ le mode "repère de μ"

- *structure* $_x$ est la fonction associant au mode μ le mode "structure avec μ à titre x"

- $union_1$ est la fonction identité sur les modes ; elle permet de généraliser $union_n$ [4.1.2.4] et est utilisée en [6.2.2.1 f)]

- *portée* est la fonction qui associe à une valeur interne sa "portée"

- *succ* (resp. *desc*) permet de définir l'ensemble des portées

- *derep* (resp. *elarg*) est la fonction qui permet d'exprimer la modification(*) dereperer (resp. elargir) du langage Algol 68 [A-8.2] (voir remarque [4.2.1])

4.1.2.3 Définition de L_2 :

L_2 est constitué des symboles fonctionnels suivants :

- *div* (resp. *mult*) est la fonction associant à 2 réels leur quotient (resp. leur produit) [A-2.2.3.1 c)]

- *inf* est la fonction à valeurs booléennes associée à la relation d'ordre "être plus petit que" [A-2.2.3.1 c)]

- $moins^i$ (i = 15,16) sont les fonctions qui associent à deux entiers ou réels leur différence [A-2.2.2.3.1 c)]

- *ppq* permet de définir une relation d'ordre partiel sur l'ensemble des portées [4.1.5.6]

- $procr_2$ associe aux modes μ_1 et μ_2 le mode "procédure avec paramètre μ_1 à résultat μ_2"

- $procn_2$ associe aux modes μ_1 et μ_2 le mode "procédure avec paramètre μ_1 et paramètre μ_2 à résultat neutre"

- *structure* $_{x_1\ x_2}$ est la fonction associant aux modes μ_1 et μ_2 le mode "structure avec μ_1 à titre x_1 et avec μ_2 à titre x_2"

- $elem_1^i$ ($3 \leqslant i \leqslant 10$, $i \neq 5$) est la fonction associant à un exemplaire de valeur multiple v de dimension 1 et à un entier i l'élément d'indice i de v (noté $elem_1$vi)

- $union_2$ associe aux modes μ_1 et μ_2 le mode "union de μ_1 et μ_2"

- sub^i qui permet de formaliser les appels de procédures [6.3.11] .

4.1.2.4 Définition de L_n ($n \geqslant 3$) :

Pour $n \geqslant 3$ quelconque, L_n est formé de :

- $elem_{n-1}^i$ ($3 \leqslant i \leqslant 10$ et $i \neq 5$) fonction n-aire associant à un exemplaire de valeur multiple v de dimension n-1 et à un (n-1)-uple d'entiers $(i_1,\ldots,i_{n-1})$ l'élément d'indice $(i_1,\ldots,i_{n-1})$ de v (noté $elem_{n-1}$ v $i_1 \ldots i_{n-1}$)

(*) Il importe de distinguer la notion de modification d'une structure d'information de celle de modification du langage Algol 68 [A-8.2] ; chaque fois qu'il s'agira d'une modification du deuxième type on l'indiquera explicitement.

- $procr_n$ associe aux modes $\mu_1,\ldots,\mu_n$ le mode "procédure avec paramètre μ_1 et paramètre μ_2 ... et paramètre μ_{n-1} à résultat μ_n"
- $procn_n$ associe à $\mu_1,\ldots,\mu_n$ le mode "procédure avec paramètre μ_1 et ... et paramètre μ_n à résultat neutre"
- $union_n$ associe aux modes $\mu_1,\ldots,\mu_n$ le mode "union de μ_1 et ... et de μ_n"
- $structure_{x_1\ldots x_n}$ $(x_1,\ldots,x_n \in L_o^{id})$ associe aux modes $\mu_1,\ldots,\mu_n$ le mode "structure avec μ_1 à titre x_1 et ... et avec μ_n à titre x_n"

Si de plus n est de la forme $4p+1$ $(p \in \mathbb{N}^*)$, L_n contient également :

- $tranche_p^i$ $(i = 3$ ou $5)$ qui associe à un exemplaire de valeur multiple v et à un $(4p+1)$-uple d'entiers $(\alpha_i,\gamma_i,\beta_i,\delta_i)_{1 \leqslant i \leqslant p}$ un exemplaire d'une sous valeur dont les indexeurs [A-8.6.1.1] sont les $\alpha_i\gamma_i\beta_i\delta_i$; α_i (resp. β_i, δ_i) est la $i^{ème}$ borne inférieure (resp. borne supérieure, nouvelle borne inférieure) ; γ_i prend la valeur 1 pour exprimer la présence du symbole "à" (symbole ":"), 0 pour en exprimer l'absence (autrement dit $\gamma_i = 1$ si le $i^{ème}$ indexeur est un massicot ; $\gamma_i = 0$ si c'est un indice).

Nous verrons ([6.3.8]) que les symboles $elem_n$ sont réservés à la sélection d'un élément de tableau, les symboles $tranche_p$ concernant les sous tableaux. L'introduction de deux types de symboles différents rend plus agréable l'écriture de certaines formules.

Enfin lorsque $n = 4$, L_4 contient également $cond^i$ $(12 \leqslant i \leqslant 19)$ qui permet de définir la valeur d'une proposition conditionnelle ; il est défini par les schémas d'axiomes $SH_{5.1}$ et $SH_{5.2}$.

On trouvera en Annexe 1 le tableau des différents éléments de L.

4.1.3 Schémas fonctionnels d'Algol 68 :

4.1.3.1 Introduction :

Rappelons [2.3.2.2] que l'ensemble des schémas fonctionnels est la réunion de n ensembles $S_1,S_2,\ldots,S_m$ formant la solution minimale du système à point fixe Σ :

$$1 \leqslant k \leqslant m \qquad S_k = \bigcup_{q \geqslant 0} \{f\, S_{j_1}\ldots S_{j_q} \mid f \in L_q \text{ tel que } (j_1,\ldots,j_q,k) \in pl(f).$$

Inversement, L et Σ étant donnés il est immédiat d'en déduire m et pl. Il est donc équivalent d'écrire le système Σ définissant les S_k ou de définir m et pl. Il est plus naturel ici de décrire Σ.

Afin de fournir une aide mnémotechnique à la compréhension de Σ, nous représenterons l'ensemble des schémas de type k non pas par S_k mais par une notation

plus intuitive (par exemple EXENT sera préféré à S_6 pour désigner l'ensemble des schémas interprétables comme des exemplaires d'entiers).

Cependant pour conserver la notion de profil, nous numéroterons chacune des équations du système Σ : si l'équation définissant l'inconnue Y a pour numéro k, Y désigne l'ensemble des schémas de type k (noté S_k auparavant).

En [4.1.2] nous sommes restés volontairement imprécis quant aux domaines de définition et d'arrivée des fonctions interprétant les symboles fonctionnels. Rappelons que la notion de profil a été introduite pour distinguer ces ensembles de départ et d'arrivée : on n'admet de "comparer" 2 éléments que s'ils appartiennent au même ensemble (ce qui se traduit par la définition formelle des formules atomiques : $A = \bigcup_k S_k \equiv S_k$). Mais dans le cas d'Algol 68, la fonction possède, par exemple, est à valeurs dans l'ensemble des exemplaires de réels, d'entiers, de noms, de valeurs multiples etc...
Dans notre formalisation il lui correspondra un certain nombre de symboles fonctionnels $poss^i$, l'indice i précisant quel est le "domaine d'arrivée" correspondant.

Remarque : On aurait pu en faire autant pour les ensembles de départ, ce qui aurait évité qu'un profil ne soit un ensemble, mais cela aurait compliqué encore davantage la formalisation.

4.1.3.2 Définition du système Σ :

Avec les remarques précèdentes, les types des schémas fonctionnels vont correspondre aux ensembles d'objets que l'on peut rencontrer en Algol 68.

Ces objets peuvent être :

i) soit des objets externes : il leur correspondra 2 inconnues dans Σ
   - IDENT associée aux identificateurs, notations, opérateurs...
   - INDIC associée aux modes

ii) soit des objets internes : les seuls objets internes définis dans le rapport Algol 68 sont les exemplaires de valeurs. Dans Σ on trouvera :
   - EXNOM associée aux exemplaires de noms différents de "*poss nil*"
   - $EXNOM_1$ exemplaires de noms
   - EXSTRU inconnue associée aux exemplaires de structure
   - EXMUL exemplaires de valeurs multiples
   - Avec des notations évidentes, EXENT, EXREEL, EXBOOL, EXCAR sont les inconnues de Σ qui correspondent aux exemplaires de valeurs simples
   - Enfin EXPROC est relative aux exemplaires de routines.

Remarque : Nous ne traitons pas les entrées-sorties, donc nous n'introduisons pas les formats dans la structure.

iii) En plus des objets décrits explicitement dans [A] , nous introduirons dans la structure d'information les valeurs des objets internes. La notation utilisée pour désigner (l'inconnue relative à) un ensemble de valeurs se déduira de la notation utilisée pour désigner (l'inconnue relative à) l'ensemble des exemplaires de valeurs correspondants en supprimant le "EX" (par exemple si EXMUL se rapporte aux exemplaires de valeurs multiples, MUL se rapporte aux valeurs multiples).

iv) Deux inconnues seront introduites pour rendre compte de l'ensemble des modes et de l'ensemble des portées : MODE et PORTEE.

Rappelons les notations utilisées :

- Pour tout symbole fonctionnel f , $f^{(n)}$ représente $\underbrace{f\ f\ \dots\ f}_{n \text{ fois}}$
- Pour tout ensemble S de schémas fonctionnels, $S^{(n)}$ représente l'ensemble des mots de longueur n sur S .

Le système Σ est un système à point fixe sur $\mathfrak{P}(L)$ relativement aux fonctions de base réunion et concaténation, la seconde ayant priorité sur la première.

Nous pourrons utiliser des parenthèses soit pour modifier cette priorité, soit pour apporter plus de clarté à l'écriture.

Nous allons tout d'abord définir Σ puis en [4.1.3.3] nous commenterons ce système.

Système Σ :

① IDENT = $L_o^{id} \cup L_o^{bool} \cup L_o^{car} \cup L_o^{op} \cup L_o^{ent} \cup L_o^{re} \cup L_o^{ph} \cup \bigcup_{i \in \mathbb{N}} a_i^1$ INDIC $\cup \{\underline{vide}\}$
$\bigcup_{i \in \mathbb{N}} b_i^1$ INDIC $\cup$ *derep* IDENT $\cup$ sub^1 IDENT IDENT $\cup$ $poss^1$ IDENT $\cup$ *elarg* IDENT

② INDIC = $L_o^{prim} \cup L_o^{ind} \cup L_o^{nprim} \cup sub^2$ IDENT INDIC

③ EXNOM = $poss^3$ IDENT $\cup$ rep^3 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^3 (\text{EXNOM} \cup \text{EXSTRU})\right) \cup sub^3$ IDENT EXNOM
$\left(\bigcup_{n \in \mathbb{N}} elem_n^3 \text{ (EXMUL} \cup \text{EXNOM)ENT}^{(n)}\right) \cup \left(\bigcup_{n \in \mathbb{N}} tranche_n^3 \text{ EXNOM(ENT ZUN ENT}^{(2)}\text{)}^{(n)}\right)$

④ EXSTRU = $poss^4$ IDENT $\cup$ rep^4 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^4 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}} elem_n^4 \text{ EXMUL ENT}^{(n)}\right) \cup sub^4$ IDENT EXSTRU

⑤ EXMUL = $poss^5$ IDENT $\cup$ rep^5 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^5 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}} tranche_n^5 \text{ EXMUL(ENT ZUN ENT}^{(2)}\text{)}^{(n)}\right) \cup sub^5$ IDENT EXMUL

⑥ EXENT = $poss^6$ IDENT $\cup$ rep^6 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^6 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}^*} elem_n^6 \text{ EXMUL ENT}^{(n)}\right) \cup sub^6$ IDENT EXENT

⑦ EXREEL = $poss^7$ IDENT $\cup$ rep^7 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^7 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}^*} elem_n^7 \text{ EXMUL ENT}^{(n)}\right) \cup sub^7$ IDENT EXREEL

⑧ EXBOOL = $poss^8$ IDENT rep^8 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^8 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}^*} elem_n^8 \text{ EXMUL ENT}^{(n)}\right) \cup sub^8$ IDENT EXBOOL

⑨ EXCAR = $poss^9$ IDENT rep^9 EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^9 \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}^*} elem_n^9 \text{ EXMUL ENT}^{(n)}\right) \cup sub^9$ IDENT EXCAR

⑩ EXPROC = IDENT $\cup$ $poss^{10}$ IDENT $\cup$ rep^{10} EXNOM $\cup \left(\bigcup_{x \in L_o^{id}} ch_x^{10} \text{ EXSTRU}\right) \cup$
$\left(\bigcup_{n \in \mathbb{N}^*} elem_n^{10} \text{ EXMUL ENT}^{(n)}\right) \cup sub^{10}$ IDENT EXPROC

⑪ EXNOM 1 = EXNOM $\cup$ $poss^{11}$ *nil*

⑫ NOM 1 = $valeur^{12}$ EXNOM 1 $\cup$ $cond^{12}$ VALEUR$^{(2)}$ STRU VALEUR $\cup$ $cond^{12}$ VALEUR$^{(3)}$ NOM 1

⑬ STRU = $valeur^{13}$ EXSTRU $\cup$ $cond^{13}$ VALEUR$^{(2)}$ NOM1 VALEUR $\cup$ $cond^{13}$ VALEUR$^{(3)}$ STRU

⑭ MUL = $valeur^{14}$ EXMUL $\cup$ $cond^{14}$ VALEUR$^{(2)}$ MUL VALEUR $\cup$ $cond^{14}$ VALEUR$^{(3)}$ MUL

⑮ ENT = $valeur^{15}$ EXENT $\cup$ $moins^{15}$ ENT ENT $\cup$ $equi_2^{15}$ ENT $\cup$ $equi_3$ CAR $\cup$ $\overline{equi_2^{15}}$ ENT $\cup$
$\overline{equi_1}$ REEL $\cup$ $cond^{15}$ VALEUR$^{(2)}$ ENT VALEUR $\cup$ $cond^{15}$ VALEUR$^{(3)}$ ENT $\cup$
$\left(\bigcup_{i \in \mathbb{N}^*} a_i^{15} \text{ EXMUL}\right) \cup \left(\bigcup_{i \in \mathbb{N}^*} b_i^{15} \text{ EXMUL}\right) \cup dim$ EXMUL $\cup$ *dim* INDIC

⑯ REEL = $valeur^{16}$ EXREEL $\cup$ $moins^{16}$ REEL REEL $\cup$ *mult* REEL REEL $\cup$ *div* REEL REEL $\cup$
$equi_1$ ENT $\cup$ $equi_2^{16}$ REEL $\cup$ $\overline{equi_2^{16}}$ REEL $\cup$ $cond^{16}$ VALEUR$^{(2)}$ REEL VALEUR $\cup$
$cond^{16}$ VALEUR$^{(3)}$ REEL

(17) BOOL = $valeur^{17}$ EXBOOL ∪ ppq PORTEE PORTEE ∪ inf ENT ENT ∪ inf REEL REEL ∪ $souflex$ (EXNOM ∪ EXMUL ∪ EXSTRU ∪ EXENT ∪ EXREEL ∪ EXBOOL ∪ EXCAR ∪ EXPRO) ∪ $cond^{17}$ VALEUR$^{(2)}$ BOOL VALEUR ∪ $cond^{17}$ VALEUR$^{(3)}$ BOOL

(18) CAR = $valeur^{18}$ EXCAR ∪ $cond^{18}$ VALEUR$^{(2)}$ CARVALEUR ∪ $cond^{18}$ VALEUR$^{(3)}$ CAR

(19) PROC = $valeur^{19}$ EXPROC ∪ $cond^{19}$ VALEUR$^{(2)}$ PROCVALEUR ∪ $cond^{19}$ VALEUR$^{(3)}$ PROC

(20) PORTEE = {θ} ∪ PORTEE 1 ∪ $portée$ EXVAL

(21) PORTEE 1 = $succ$ PORTEE 1 ∪ $desc$ PORTEE

(22) MODE = $long$ MODE ∪ $repère$ MODE ∪ $\left(\bigcup_{x_1,\ldots,x_n \in L_o^{id}} structure\ x_1\ldots x_n\ \text{MODE}^{(n)}\right)$ ∪ $rang$ MODE ∪ $\left(\bigcup_{n\in\mathbb{N}^*} procn_n\ \text{MODE}^{(n)}\right)$ ∪ $\left(\bigcup_{n\in\mathbb{N}^*} procr_n\ \text{MODE}^{(n)}\right)$ ∪ $\left(\bigcup_{n\in\mathbb{N}^*} union_n\ \text{MODE}^{(n)}\right)$ ∪ $indique$ INDIC ∪ $mode$ EXVAL

(23) ZUN = {0,1}

(24) ZUD = {0,1,2} ∪ $\left(\bigcup_{i\in\mathbb{N}^*} s_i^{24}(\text{EXMUL} \cup \text{INDIC})\right)$ ∪ $\left(\bigcup_{i\in\mathbb{N}^*} t_i^{24}(\text{EXMUL} \cup \text{INDIC})\right)$

VALEUR = ENT ∪ REEL ∪ BOOL ∪ CAR ∪ NOM1 ∪ PROC ∪ MUL ∪ STRU

EXVAL = EXENT ∪ EXREEL ∪ EXBOOL ∪ EXCAR ∪ EXNOM1 ∪ EXPROC ∪ EXMUL ∪ EXSTRU

STMUL = $\left(\bigcup_{x_1\ldots x_n \in L_o^{id}} structure\ x_1\ldots x_n\ \text{MODE}^{(n)}\right)$ ∪ $rang$ MODE

RSTMUL = $repère$ STMUL

AUTRE = $long$ MODE ∪ $\left(\bigcup_{n\in\mathbb{N}^*} procn_n\ \text{MODE}^{(n)}\right)$ ∪ $\left(\bigcup_{n\in\mathbb{N}^*} procr_n\ \text{MODE}^{(n)}\right)$ ∪ $indique$ L_o^{prim}

UNION = $\bigcup_{n\in\mathbb{N}^*} union_n\ \text{MODE}^{(n)}$

RUAUTRE = $repère$ AUTRE ∪ $repère$ $repère$ MODE ∪ $repère$ UNION

NOTENT = $valeur^{15}$ $poss^6$ L_o^{ent}

4.1.3.3 Commentaires :

a) Les deux premières équations définissent les ensembles de schémas fonctionnels interprétables comme les objets externes Algol 68. Les a_i^1 INDIC et b_i^1 INDIC

permettront de rendre compte des bornes inférieures et supérieures de déclarateurs de valeurs multiples, éventuellement vides (d'où le symbole 0-aire *vide*).

derep est le symbole fonctionnel associant à une phrase (objet externe) possèdant un nom, une autre phrase possèdant la valeur repérée par ce nom [SH. 1.10] ; c'est un symbole de profil (1,1).

Les deux derniers ensembles de schémas fonctionnels seront utiles pour rendre compte des procédures [6.3.11] ; les profils de sub^1 et $poss^1$ sont donnés par : pl(sub^1) = (1,1,1), pl($poss^1$) = (1,1).

b) L'équation (3) définit l'ensemble des schémas fonctionnels qui seront interprétés comme des exemplaires de noms qui ne sont pas des exemplaires de *nil* :

b1) $poss^3$ est le symbole fonctionnel associant à un objet externe l'exemplaire de nom (s'il existe, voir [2.3.4]) possédé par cet identificateur. Son profil est donc pl($poss^3$) = (1,3).

b2) Un exemplaire de nom peut être repéré par un autre exemplaire de nom, c'est ce qui exprime rep^3 EXNOM ; donc pl(rep^3) = (3,3).

b3) Un exemplaire de nom peut être un composant (resp. un élément) d'un exemplaire de valeur structurée (resp. multiple), d'où les schémas ch_x^3 EXSTRU et $elem_n^3$ EXMUL ENT$^{(n)}$.

b4) Il résulte de [A-2.2.3.5 b) et c)] que si N (resp. N_1) est un exemplaire de nom de valeur structurée (resp. multiple) les composants (resp. les éléments) de N (resp. N_1) obtenus de manière unique à partir des sélecteurs de champs (resp. sélecteurs d'éléments) sont des exemplaires de noms, d'où les schémas ch_x^3 EXNOM (resp. v_n^3 $elem_n^3$ EXNOM ENT$^{(n)}$). On déduit de b3) et b4) : pl(ch_x^3) = {(4,3), (3,3)} et pl($elem_x^3$) = {(5,$\overbrace{6,\ldots,6}^{n}$,3), (3,$\overbrace{6,\ldots,6}^{n}$,3)}.

b5) Les symboles $poss^3$, rep^3 , ch_x^3 , $elem_n^3$ et $tranche_n^3$ sont "définis" sur des exemplaires de valeurs plutôt que sur des valeurs : ils permettent de construire les "atomes" de la structure et il peut être indispensable de distinguer deux exemplaires différents.

Cette manière de faire permet d'exprimer simplement les traitements élémentaires (affectation, déclaration d'identité...) qui sont essentiellement relatifs à des exemplaires de valeurs. Il en sera de même dans la suite.

Remarquons cependant que les indices d'un élément ou d'une tranche sont des valeurs entières et non pas des exemplaires : un exemplaire d'élément se distingue d'un autre exemplaire d'élément ayant même valeur par l'exemplaire de valeur multiple qui le définit et non pas par ses indices.

Les valeurs étant plus agréables à manipuler que les exemplaires, on définira les symboles fonctionnels au "niveau des valeurs" chaque fois que cela sera possible.

c) Les équations ④ et ⑤ expriment, de manière analogue, que tout exemplaire de structure (resp. de valeur multiple) peut être possédé par un identificateur, repéré par un exemplaire de nom, sélectionné dans un exemplaire de structure ou de valeur multiple (resp. possédé par un identificateur, repéré par un exemplaire de nom, sélectionné dans une structure ou obtenu comme tranche d'un exemplaire d'une autre valeur multiple).

On en déduit les profils des différents symboles apparaissant dans ces équations ; dorénavant nous ne les exprimerons plus explicitement.

d) Les équations ⑥ , ⑦ , ⑧ , ⑨ définissent les schémas fonctionnels interprétables comme exemplaires de valeurs simples ; ils sont obtenus à l'aide des symboles $poss^i$, rep^i, ch^i_x et $elem^i_n$.

e) L'équation ⑩ caractérise les exemplaires de routines ; nous verrons [6.3.10] qu'une notation de routine possède un exemplaire de routine qui est une phrase Algol 68 (proposition éventuellement réduite à un identificateur, une constante...).

f) ⑪ caractérise tous les exemplaires de noms. Remarquons que la distinction entre ③ et ① permet d'exprimer [A-2.2.3.5 a)] que *nil* ne repère aucune valeur.

g) ⑫ , ⑬ , ⑭ , ⑱ , ⑲ caractérisent les schémas interprétables comme des valeurs d'Algol 68 qui ne sont ni des entiers, ni des réels. On les obtient soit à partir de leurs exemplaires respectifs par l'intermédiaire du symbole $valeur^i$, soit par l'intermédiaire du symbole *cond* et des schémas appartenant à VALEUR.

Par exemple un schéma interprétable comme une structure pourra s'écrire sous la forme *cond* $v_1 v_2$ u u lorsque le test portant sur v_1 et v_2 sera positif et que u sera interprétable comme une structure.

Remarquons que le seul intérêt des équations placées après ㉔ est de permettre des simplifications d'écritures, elles ne font pas partie du système Σ . Les ensembles EXVAL, STMUL ... RUAUTRE, NOTENT sont utilisés en [6] .

h) ⑮ caractérise les valeurs réelles ; on trouve en plus des schémas analogues à ceux de ⑫ ⑬ ⑭ et ⑱ ⑲ des schémas formalisant :

- les relations définies en [A-2.2.3.1 c), d) et f)]
- les fonctions définies en [A-2.2.3.3 b)] et [A-10.4].

Remarquons que *moins*, $equi_1$ etc... sont définies "sur des valeurs" et non pas sur des exemplaires ; cette définition se révèle suffisante dans la suite car le

résultat d'une opération est un *nouvel* exemplaire de valeur , c'est-à-dire que seule la valeur a de l'importance.

Par contre les symboles a_i , b_i , s_i , t_i (relatifs au descripteur d'une valeur multiple) sont définis sur des exemplaires : dans la définition de l'affectation nous verrons [6.3.2] qu'une partie du descripteur de l'exemplaire de valeur que l'on affecte, peut être modifiée et il ne faut pas que cette modification se répercute sur les autres exemplaires de la même valeur.

Par soucis d'homogénéité, *dim* est également défini sur des exemplaires.

i) ⑯ définit les schémas fonctionnels interprétables comme des valeurs réelles. On y trouve les schémas caractérisant les relations de [A-2.2.3.1 c), d)].

j) Les équations ⑳ et ㉑ caractérisent les portées :

les schémas fonctionnels interprétables comme portées forment un C-langage sur l'alphabet {θ , *succ*, *desc*, *portée* EXVAL}.

En fait le seul rôle de portée EXVAL est de "relier" l'ensemble des portées à l'ensemble des exemplaires de valeurs.

On peut interpréter PORTEE de la manière suivante :

θ est interprété par l'entier 1

succ est interprété par la fonction qui à une suite d'entiers $a_1\ a_2\ \dots\ a_n$ associe la suite d'entiers $a_1\ a_2\ \dots\ (a_n+1)$

desc est interprété par la fonction qui à une suite d'entiers $a_1\ a_2\ \dots\ a_n$ associe la suite d'entiers $a_1\ a_2\ \dots\ a_n 1$.

L'ensemble PORTEE est alors interprété comme l'ensemble des suites d'entiers :

{1, 2, ... ; 11, 12, ... ; 111, 112, ... } .

Nous munirons cet ensemble d'une relation d'ordre partielle à l'aide du symbole *ppq*.

k) L'équation ㉒ exprime que tout mode s'obtient à partir des modes de base (éléments de indique L_o^{prim}) et d'indicateurs de modes (*indique* INDIC) par composition des symboles *long*, *repère*, *rang* etc...

l) ZUN est utilisé dans la définition des tranches : il permet d'exprimer la présence ou l'absence du symbole ":" ; ZUD est utilisé dans les déclarateurs de valeurs multiples.

m) Il peut être utile pour la suite [6] de remarquer que
- RSTMUL ∪ RUAUTRE = *repère* MODE
- STMUL ∪ AUTRE ∪ UNION = MODE - *repère* MODE .

n) Rappelons [2.3.4] que la notion de profil permet de limiter l'ensemble des schémas fonctionnels considérés. En particulier il est agréable (définition de A [4.1.4]) de ne comparer des éléments que s'ils appartiennent au même ensemble.

Remarquons cependant que pour éviter d'alourdir la formalisation nous admettons certains schémas fonctionnels qui ne sont pas souhaitables (par exemple $rep^3\ ch_x^3\ u$ où u n'est pas un nom de structure, équation ③) et nous utiliserons le contrôle syntaxique imposé à tout programme correct Algol 68 pour ne pas obtenir de tels schémas dans les informations manipulées.

4.1.4 Formules atomiques :

Seules les 24 équations numérotées font partie de Σ [4.1.3.2] , aussi

$$A = \bigcup_{1 \leqslant k \leqslant 24} S_k \equiv S_k \qquad (S_1 = \text{IDENT}, S_2 = \text{INDIC etc...})$$

Dans la suite et pour alléger les notations, nous confondrons les symboles fonctionnels "relatifs à la même fonction" : on supprime les indices distinguant $poss^3$, $poss^4$ ... ; rep^3, rep^4 ... etc.

4.1.5 Définition de l'ensemble X des axiomes du système formel :

Outre les axiomes logiques $\{SL_i \mid i = 1,2,\ldots,6\}$ communs à toutes les structures d'information [2.3.2.5] , X contient les axiomes propres de la structure Algol 68 (X_p est leur ensemble). Comme nous l'avons remarqué en [4.1.1] , ces axiomes spécifiques de $\mathcal{S}$ formalisent les propriétés des "relations" existant entre les "objets" Algol 68 [A-2.2].

4.1.5.1 Schémas d'axiomes relatifs aux noms :

$SH_{1.1} = \{rep\ ch_x\ u \equiv ch_x\ rep\ u \mid x \in L_o^{id}\ ;\ u \in \text{EXNOM}\}$

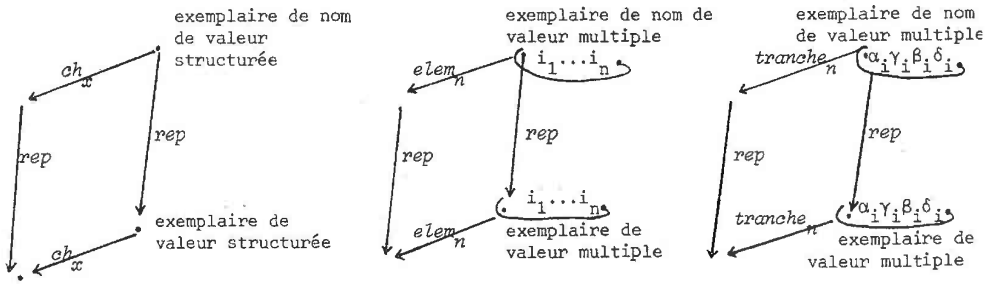
$SH_{1.2} = \{rep\ elem_n\ u\ i_1\ldots i_n \equiv elem_n\ rep\ u\ i_1\ldots i_n \mid n \geqslant 1\ ;\ i_1\ldots i_n \in \text{ENT}\ ;$
$u \in \text{EXNOM}\}$

$SH_{1.3} = \{rep\ tranche_n\ u\ \alpha_1\gamma_1\beta_1\delta_1\ \ldots\ \alpha_n\gamma_n\beta_n\delta_n \equiv$
$tranche_n\ rep\ u\ \alpha_1\gamma_1\beta_1\delta_1\ \ldots\ \alpha_n\gamma_n\beta_n\delta_n \mid n \geqslant 1\ ;\ \alpha_i,\beta_i,\gamma_i \in \text{ENT}\ ,$
$\gamma_i \in \text{ZUN}$ pour $i = 1,\ldots,n$; $u \in \text{EXNOM}\}$.

Ces schémas d'axiomes expriment en partie [A-2.2.3.5 b) et c)] . En particulier ces 3 axiomes permettront de traduire [A-8.5.2.2 pas 2] et [A-8.6.1.2 pas 8] :

si la valeur d'une sélection ou d'une tranche est un nom, on déduit de $SH_{1.1}$, $SH_{1.2}$ ou $SH_{1.3}$ quel est l'exemplaire repéré par ce nom.

En termes d'interprétation de $\mathcal{F}$, ces trois axiomes signifient que les "diagrammes" suivants sont commutatifs :



$SH_{1.4} = \{mode\ u \equiv repère\ structure\ \underset{x_1\ldots x_n}{\mu_1\ldots\mu_n} \wedge portée\ u \equiv p \supset$
$\bigwedge_{1 \leqslant j \leqslant n} (mode\ chx_j\ u \equiv repère\ \mu_j \wedge portée\ chx_j\ u \equiv p \mid u \quad \text{EXNOM}\ ;$
$\mu_1,\ldots,\mu_n \in \text{MODE}\ ;\ x_1,\ldots,x_n \in L_o^{id}\ ;\ p \in \text{PORTEE}\}$

$SH_{1.5} = \{mode\ u \equiv repère\ union_k\ \nu_1\ldots\nu_k \wedge \nu_i \equiv structure\ \underset{x_1\ldots x_n}{\mu_1\ldots\mu_n} \wedge$
$portée\ u \equiv p \supset \bigwedge_{1 \leqslant j \leqslant n} (mode\ chx_j\ u \equiv repère\ \mu_1 \wedge portée\ chx_j\ u \equiv p)$
$u \in \text{EXNOM}\ ;\ \nu_1,\ldots,\nu_k\ ,\ \mu_1,\ldots,\mu_n \subset \text{MODE}\ ,\ x_1,\ldots,x_n \in L_o^{id}\ ;\ k,n \geqslant 1$
et $1 \leqslant i \leqslant k$; $p \in \text{PORTEE}\}$.

Si le rôle de $SH_{1.4}$ est évident, il faut s'attarder un instant sur $SH_{1.5}$: si un nom repère une valeur structurée ; le mode de ce nom est "repère de" suivi du mode de la valeur structurée, ou "repère de" suivi d'un mode uni à partir du mode de

la valeur structurée [A-2.2.4.1 g)] . Ainsi un nom repérant une valeur structurée peut avoir un mode commençant par "*repère structure* $x_1 \ldots x_n$" ou par "*repère union*$_n$".

Remarque 1 : Pour que l'ensemble T des théorèmes de la structure ne soit pas inconsistant, il faut exiger que si le mode d'une valeur quelconque est *union* de $\nu_1 \ldots$ de ν_k et si ν_i = *structure* $x_{1_i} \ldots x_{n_i}$ $\mu_{1_i} \ldots \mu_{n_i}$ et ν_j = *structure* $y_{1_j} \ldots y_{n_j}$ $\mu_{1_j} \ldots \mu_{n_j}$ alors $y_{k_j} \neq x_{\ell_i}$ pour tous $1 \leqslant k$, $\ell \leqslant n$. C'est ce que nous supposerons dans le langage noyau.

Deux schémas d'axiomes caractérisent de même les noms possédant une valeur multiple [A-2.2.3.5 c)]

$SH_{1.6}$ = {*mode* u ≡ *repère rang*$^{(n)}$ μ ∧ *portée* u ≡ p ⊃
mode elem$_n$ u $i_1 \ldots i_n$ ≡ *repère* μ ∧ *portée elem*$_n$ u $i_1 \ldots i_n$ ≡ p | n ⩾ 1 ;
u ∈ EXNOM ; μ ∈ MODE ; $i_1 \ldots i_n$ ∈ ENT ; p ∈ PORTEE}

$SH_{1.7}$ = {*mode* u ≡ *repère union*$_k$ $\nu_1 \ldots \nu_k$ ∧ ν_i ≡ *rang*$^{(n)}$ μ ∧ *portée* u ≡ p ⊃
mode elem$_n$ u $i_1 \ldots i_n$ ≡ *repère* μ ∧ *portée elem*$_n$ u $i_1 \ldots i_n$ ≡ p | n ⩾ 1 ;
k ⩾ 1 ; 1 ⩽ i ⩽ k ; u ∈EXNOM ; $\nu_1, \ldots, \nu_k$, μ ∈ MODE ;
$i_1 \ldots i_n$ ∈ ENT ; p ∈ PORTEE} .

Le cas particulier des tranches est traité en [4.1.5.2] .

Les quatre schémas précèdents traitent le cas des "sous noms" des noms de valeurs multiples ou structurées. Pour rendre compte complètement de [A-2.2.4.1 g)] il faut exprimer la relation existant entre le nom et la valeur qu'il repère :

$SH_{1.8}$ = {*mode* u ≡ *repère* μ ⊃ *mode rep* u ≡ μ | u ∈EXNOM, μ ∈ MODE-UNION}

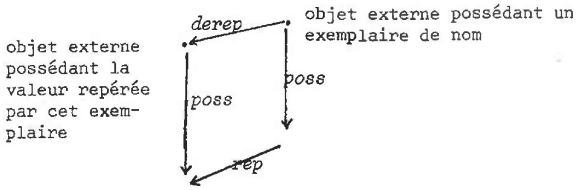
$SH_{1.9}$ = {*mode* u ≡ *repère union*$_n$ $\mu_1 \ldots \mu_n$ ⊃ $\bigvee_{1 \leqslant i \leqslant n}$ *mode rep* u ≡ μ_i |
u ∈ EXNOM ; n ⩾ 2 ; $\mu_1, \ldots, \mu_n$ ∈ MODE} .

Les schémas $SH_{1.4}$ à $SH_{1.9}$ sont utiles dans la définition de l'élaboration d'un générateur [5.3.7] .

On doit également définir le symbole "derep" qui permet de rendre compte du dereperage [5.2] :

$SH_{1.10}$ = {*poss derep* u ≡ *rep poss* u | u ∈ IDENT}

en termes d'interprétation, le diagramme suivant est commutatif :



Enfin il faut relier les notions d'exemplaires de noms et celles de noms :

$SH_{1.11}$ = {*valeur* u ≡ *valeur* v ⊃ *rep* u ≡ *rep* v | u,v ∈ EXNOM}

schéma d'axiomes qui exprime ([A-2.2.2 b)]) que tout nom repère un seul exemplaire de valeur. La "fonction repère" n'étant définie dans notre formalisation que sur des exemplaires de noms, nous exprimons que deux exemplaires du même nom repèr nt le même exemplaire de valeur.

$SH_{1.12}$ = {*rep* u ≡ *rep* v ⊃ *valeur* u ≡ *valeur* v | u,v ∈ EXNOM}

schéma qui traduit [A-2.2.3.5 d)] .

Remarque 2 :

Une conséquence de $SH_{1.11}$ est de permettre la "définition" de la "fonction repère" sur les noms, à valeurs dans les exemplaires de valeurs.

On aurait pu ajouter à ③ , ④ ... ⑪ : *rep*^{i} NOM où NOM se déduirait de EXNOM comme NOM1 se déduit de EXNOM1. Ces nouvelles définitions auraient permis d'exprimer la notion d'affectation conformément à [A] , c'est-à-dire de traduire l'affectation d'un exemplaire de valeur à un nom. Nous ne le ferons pas pour ne pas alourdir cette formalisation.

Remarquons également que si *rep* était "défini" sur les noms, $SH_{1.12}$ exprimerait "l'injectivité" de cette "fonction".

Nous utiliserons cette propriété dans la définition de l'affectation sous la forme : un exemplaire de valeur peut être repéré par au plus un exemplaire de nom (c'est-à-dire, plus brièvement avec $SH_{1.11}$, par un nom).

Rappelons que *nil* est une abréviation de *valeur poss nil* et que *rep poss nil* $\notin$ S par définition de Σ .

4.1.5.2 Schémas d'axiomes relatifs aux valeurs multiples

$$SH_{2.1} = \{ \bigwedge_{1 \leqslant i \leqslant n} (\alpha_i \equiv a_i\ v \wedge \alpha_i \equiv a_i\ u \wedge \beta_i \equiv b_i\ v \wedge \beta_i \equiv b_i\ u) \wedge$$

$$dim\ v \equiv n \wedge dim\ u \equiv n \wedge \bigwedge_{\alpha_i \leqslant j_i \leqslant \beta_i \text{ pour } 1 \leqslant i \leqslant n} elem_n\ u\ j_1 \ldots j_n \equiv elem_n\ u\ j_1 \ldots j_n \supset$$

$$v \equiv u \mid n \in NOTENT,\ n \geqslant 1;\ v,u \in EXMUL;\ j_i, \alpha_i, \beta_i \in NOTENT\ \}\ .$$

Schéma d'axiomes qui exprime [A-2.2.3.3 d)] : 2 exemplaires d'une valeur multiple composés des mêmes éléments sont égaux.

Remarque 1 :

En toute rigueur il est incorrect d'écrire $\alpha_i \leqslant j_i \leqslant \beta_i$ pour α_i , j_i , $\beta_i \in$ NOTENT. Mais il existe une application de NOTENT (= *valeur poss* L_o^{ent}) dans l'ensemble $\mathbb{Z}$ des entiers relatifs. Nous confondrons donc couramment un élément de NOTENT et son image dans $\mathbb{Z}$ (c'est pour cette raison que dans $SH_{2.1}$ on se restreint à α_i , j_i , $\beta_i \in$ NOTENT au lieu de α_i , j_i , $\beta_i \in$ ENT).

Remarque 2 :

L'introduction de α_i , β_i et n permet d'éviter l'emploi de quantificateurs dans le système formel ; cette manipulation est rendue possible car le nombre de formules apparaissant en partie gauche de " $\supset$ " est fini.

Définissons maintenant les axiomes caractérisant les tranches qui, dans notre formalisation ne sont pas des variables indicées mais des sous tableaux ([6.3.8])

$$SH_{2.2} = \{u \equiv tranche_n\ v\ \alpha_1\gamma_1\beta_1\delta_1 \ldots \alpha_n\gamma_n\beta_n\delta_n \wedge (\bigwedge_{1 \leqslant i \leqslant q} \gamma_{\ell_i} \equiv 1) \wedge$$

$$(\bigwedge_{q+1 \leqslant i \leqslant n} \gamma_{\ell_i} \equiv 0) \wedge mode\ v \equiv repère^{(\varepsilon)}\ rang^{(n)}\ \mu \supset$$

$$dim\ u \equiv q \wedge mode\ u \equiv repère^{(\varepsilon)}\ rang^{(q)}\ \mu \wedge portée\ u \equiv portée\ v \wedge$$

$$(\bigwedge_{1 \leqslant i \leqslant q} (a_i\ u \equiv \gamma_{\ell_i} \wedge b_i\ u \equiv moins\ \gamma_{\ell_i}\ moins\ \alpha_{\ell_i}\ \beta_{\ell_i} \wedge$$

$$s_i\ u \equiv 1 \wedge t_i\ u \equiv 1)) \mid q, n \in NOTENT;\ n \geqslant q \geqslant 1;\ \alpha_i, \beta_i, \gamma_i \in ENT;$$

$$\gamma_i \in ZUN;\ u,v \in EXMUL \cup EXNOM;\ \mu \in MODE;\ \varepsilon \in \{0,1\};$$

$$\ell \text{ permutation de } \{1,\ldots,n\} \text{ telle que } \ell_1 \leqslant \ell_2 \leqslant \ldots \leqslant \ell_q \text{ et}$$

$$\ell_{q+1} \leqslant \ell_{q+2} \leqslant \ldots \leqslant \ell_n\ \}\ .$$

Ce schéma d'axiomes caractérise la dimension, la portée, le mode et le descripteur d'une tranche, c'est-à-dire formalise [A-8.6.1.2] pas 3 à 6 et une partie au pas 7 .

En particulier la connaissance des γ_i permet de déterminer la dimension de u . De plus :

- $repère^{(\varepsilon)}$ remplace *repère* si $\varepsilon = 1$; $repère^{(0)}$ $rang^{(q)}$ μ remplace $rang^{(q)}$ μ.
- ℓ_i (préféré à $\ell(i)$ pour alléger les notations) permet de distinguer les massicots des indices.
- *moins* δ_{ℓ_i} *moins* α_{ℓ_i} β_{ℓ_i} "fournit" la $i^{ème}$ nouvelle borne supérieure $(\delta_{\ell_i} + \beta_{\ell_i} - \alpha_{\ell_i})$.

$$SH_{2.3} = \{u \equiv tranche_n\ v\ \alpha_1\gamma_1\beta_1\delta_1 \ldots \alpha_n\gamma_n\beta_n\delta_n \wedge (\bigwedge_{1 \leqslant i \leqslant q} \gamma_{\ell_i} \equiv 1) \wedge$$

$$(\bigwedge_{q+1 \leqslant i \leqslant n} \gamma_{\ell_i} \equiv 0) \wedge \bigwedge_{1 \leqslant i \leqslant q} (r_{\ell_i} \equiv moins\ j_i\ moins\ \delta_{\ell_i}\ \alpha_{\ell_i})$$

$$(\bigwedge_{q+1 \leqslant i \leqslant n} (r_{\ell_i} \equiv \alpha_{\ell_i})) \supset elem_q\ u\ j_1 \ldots j_q \equiv elem_n\ v\ r_1 \ldots r_n \mid$$

$$n,q \in NOTENT,\ n \geqslant q \geqslant 1;\ \alpha_i, \beta_i, \delta_i, r_i, j_i \in ENT;\ \gamma_i \in ZUN;$$

$$u,v \in EXMUL \cup EXNOM;\ \ell \text{ permutation de } \{1,\ldots,n\} \text{ telle que}$$

$$\ell_i \leqslant \ell_2 \leqslant \ldots \leqslant \ell_q \text{ et } \ell_{q+1} \leqslant \ell_{q+2} \leqslant \ldots \leqslant \ell_n\ \}\ .$$

Ce schéma formalise la 2ème partie de [A-8.6.1.2] pas 7 : les éléments d'une tranche u de v sont obtenus à partir des éléments de v , l'élément de u sélectionné par l'index $j_1,\ldots,j_q$ est l'élément de v sélectionné par l'index $r_1,\ldots,r_n$.

En particulier on peut déduire de $SH_{2.2}$ et $SH_{2.3}$ une propriété analogue à celle exprimée en $SH_{2.1}$ et relative aux tranches.

Il est nécessaire dans la suite de pouvoir caractériser les composants [A-2.2.2 k)] d'une valeur multiple flexible. Le symbole *unaire sousflex* est introduit dans ce but, il sera notamment utilisé dans la formalisation de la déclaration d'identité [6.2.3] ou d'une affectation [6.3.2] .

(Dans la sémantique d'une telle phrase apparaît la condition :

"si v ne repère pas un composant d'une valeur multiple ayant un ou plusieurs champs égaux à 0".)

Rappelons qu'un composant d'une valeur mutliple est un élément ou une sous valeur de cette valeur multiple ou de l'un de ses composants.

Cette définition est récursive, la définition de sousflex le sera :

$SH_{2.4}$ = {u ≡ *elem*$_n$ v $i_1 \ldots i_n$ ∨ u ≡ *tranche*$_n$ v $\alpha_1\gamma_1\beta_1\delta_1 \ldots \alpha_n\gamma_n\beta_n\delta_n$ ∨

u ≡ *ch*$_x$ v ⊃ $\bigvee_{1 \leqslant j \leqslant n}$ (*s*$_j$ v ≡ 0 ∨ *t*$_j$ v ≡ 0) ∨ *sousflex* v ≡ *vrai* ⊃

sousflex u ≡ *vrai* | u ∈ EXVAL ; *elem*$_n$ v $i_1 \ldots i_n$ ∈ EXVAL ;

tranche$_n$ v $\alpha_1 \ldots \delta_n$ ∈ EXVAL ; *ch*$_x$ *v* ∈ EXVAL} .

Cet axiome définit les exemplaires de valeurs qui sont composants d'un exemplaire de valeur **flexible** : ils peuvent en être composants directs (c'est ce qu'exprime *s*$_j$ v ≡ 0 ∨ *t*$_j$ v ≡ 0) ; ils peuvent être composants directs d'une valeur elle-même composante d'une valeur flexible (*sousflex* v ≡ *vrai*).

4.1.5.3 Schémas d'axiomes relatifs aux exemplaires de valeurs

En plus des objets externes, deux types d'objets coexistent dans toute interprétation de $\mathcal{S}$: les valeurs et les exemplaires de valeurs.

En fait les éléments de "base" , ceux qui permettent de définir précisément toutes les constructions, sont les exemplaires.

Comme nous l'avons déjà remarqué, la différence entre valeurs et exemplaires n'est pas très nette dans [A] , pour des raisons d'abus de langage. En particulier certaines propriétés telles que :

"si v_1 et v_2 sont deux exemplaires de la même valeur structurée et si *ch*$_x$ est un sélecteur de champ "défini" sur v_1 et v_2 alors *ch*$_x$ v_1 et *ch*$_x$ v_2 sont deux exemplaires de la même valeur" ne sont pas explicitement énoncés ; il faut les sous entendre (de [A-2.2.3.2] pour la propriété citée en exemple). Des propriétés analogues se retrouvent pour les éléments ou les tranches de valeurs multiples ; nous introduirons les 3 schémas :

$SH_{3.1}$ = {*valeur* u ≡ *valeur* v ⊃ *valeur ch*$_x$ u ≡ *valeur ch*$_x$ v | x ∈ L_o^{id} ;

u,v ∈ EXSTRU }

$SH_{3.2}$ = {*valeur* u ≡ *valeur* v ∧ *dim* u ≡ n ⊃ *valeur elem*$_n$ u $i_1 \ldots i_n$ ≡

valeur elem$_n$ v $i_1 \ldots i_n$ | n ∈ NOTENT ; $i_1 \ldots i_n$ ∈ ENT ; u,v ∈ EXMUL }

$SH_{3.3}$ = {*valeur* u ≡ *valeur* v ∧ *dim* u ≡ n ⊃

valeur tranche$_n$ u $\alpha_1\gamma_1\beta_1\delta_1 \ldots \alpha_n\gamma_n\beta_n\delta_n$ ≡

valeur tranche$_n$ v $\alpha_1\gamma_1\beta_1\delta_1 \ldots \alpha_n\gamma_n\beta_n\delta_n$ | n ∈ NOTENT ; $\alpha_i, \beta_i, \delta_i$, ∈ ENT,

γ_i ∈ ZUN ; u,v ∈ EXMUL }.

De plus tous les exemplaires d'une même valeur (différente de *nil*) sont d'un même mode [A-2.2.4.1 a)] :

$SH_{3.4}$ = {*valeur* u ≡ *valeur* v ⊃ *mode* u ≡ *mode* v | u,v ∈ EXVAL - {*poss nil*}}

Ce schéma d'axiome permettant d'étendre la définition de "mode" aux valeurs ; c'est ce que fait [A] . Pour alléger la formalisation nous ne le ferons pas (cf. Remarque 2 [4.1.5.1]).

4.1.5.4 Schémas d'axiomes relatifs aux éléments de L_o :

On imposera que deux symboles 0-aires différents ne soient pas égaux formellement : s'il n'en était pas ainsi, 2 symboles au moins joueraient le même rôle et donc l'un au moins serait inutile. On introduit le schéma :

$SH_{4.1}$ = { ¬x ≡ y | x,y ∈ L_o , x ≠ y } .

Alors, grâce à l'axiome SL_4 (= {u ≡ u | u ∈ S}) il y aura donc confusion entre l'égalité ensembliste et l'égalité formelle pour les éléments de L_o .

De plus on exigera que si une valeur booléenne est formellement égale à "*vrai*" elle ne puisse être égale à "*faux*" et réciproquement. Ceci provient du schéma :

$SH_{4.2}$ = { ¬*vrai* ≡ *faux* }

(*vrai* et *faux* étant respectivement les abréviations de "*valeur poss vrai*" et "*valeur poss faux*" [4.1.2.1 i)] .

Il résulte immédiatement de $SH_{4.2}$ et SL_6 que si u ≡ *vrai* est un théorème alors ¬u ≡ *faux* est un théorème.

$SH_{4.3}$ = {*poss elarg* x ≡ *equi*$_1$ *poss x* | x ∈ L_o}

elarg permet d'exprimer l'élargissement ([A-8.2.5]) d'entier à réel.

4.1.5.5 Schémas d'axiomes caractérisant le symbole *cond* :

cond permet de formaliser les propositions conditionnelles [6.4.5] ; il est défini par :

$SH_{5.1}$ = {u ≡ v ⊃ *cond* u v w_1 w_2 ≡ w_1 | u,v,w_1,w_2 ∈ VALEUR}

$SH_{5.2}$ = { u ≡ v ⊃ *cond* u v w_1 w_2 ≡ w_2 | u,v,w_1,w_2 ∈ VALEUR} .

Remarque :

Malgré la présence de valeurs booléennes dans la structure, *cond* est un symbole à quatre paramètres, le test portant sur les deux premiers.

En effet nous serons amenés à comparer des valeurs non numériques (par exemple des noms lors d'une relation d'identité [6.3.3]), si alors *cond* ne dépendait que de trois paramètres, le premier étant une valeur booléenne, il serait nécessaire d'introduire de nouveaux symboles à "valeurs booléennes" pour rendre compte de ce genre de test (par exemple pour les relations d'identité, il faudrait un symbole 2-aire "*égal*" défini par des axiomes du genre : $u \equiv v \supset egal\ \alpha\beta \equiv$ vrai). La formalisation serait encore plus lourde.

4.1.5.6 Schéma d'axiomes relatifs aux portées :

Rappelons que l'ensemble des schémas fonctionnels interprétables comme des portées algol 68 [A-2.2.4.2] (brièvement nous l'appellerons ensemble des portées) est défini par le sous système de Σ :

(20) PORTEE = {θ} ∪ PORTEE1 ∪ *portée* EXVAL

(21) PORTEE1 = *succ* PORTEE1 ∪ *desc* PORTEE .

Comme nous l'avons déjà remarqué [4.1.3.3 j)] le seul rôle des schémas "*portée* EXVAL" est de "relier" l'ensemble des portées aux autres valeurs ; ils permettent d'utiliser les portées mais non pas de les construire. Dans la suite du paragraphe nous allons imposer certaines "conditions" (sous forme d'axiomes aux portées. Ces conditions ne concernerons que les portées appartenant à l'ensemble $\overline{\text{PORTEE}}$ défini par :

$\overline{\text{PORTEE}}$ = {θ} ∪ $\overline{\text{PORTEE1}}$

$\overline{\text{PORTEE1}}$ = *succ* $\overline{\text{PORTEE1}}$ ∪ *desc* $\overline{\text{PORTEE}}$.

Elles s'étendront aux éléments de "*portée* EXVAL" utilisés dans un programme grâce aux axiomes logiques SL_5 et SL_6 relatifs à l'égalité formelle et aux définitions de tels éléments (définitions qui sont des axiomes de la forme *portée* u ≡ p avec p ∈ PORTEE [6]) .

Pour comprendre la suite de la formalisation, il est utile de retenir qu'une interprétation souhaitée de PORTEE est [4.1.3.3 j)] l'ensemble
E = {1,11,12,...,111,112,...} des suites d'entiers strictement positifs.

Pour que le système formel n'admette que des interprétations très proches de E nous imposerons les axiomes suivants sur "*succ*" et "*desc*" :

$SH_{6.1}$ = {¬*succ* a ≡ *succ* b | a,b ∈ $\overline{\text{PORTEE}}$, a ≠ b }

$SH_{6.2}$ = {¬*desc* a ≡ *desc* b | a,b ∈ $\overline{\text{PORTEE}}$, a ≠ b }

$SH_{6.3}$ = {¬*succ* a ≡ *desc* b | a,b ∈ $\overline{\text{PORTEE}}$ }

$SH_{6.4}$ = {¬θ ≡ *succ* a , ¬θ ≡ *desc* a | a ∈ $\overline{\text{PORTEE}}$ }.

On exprime ainsi

- "l'injectivité" des symboles *succ* et *desc*
- que toute portée est soit la "portée initiale" θ , soit le descendant, soit le successeur d'une unique autre portée (et ceci de manière exclusive).

Nous caractériserons alors un symbole 2-aire *ppq* qui sera interprété, dans l'interprétation E des suites d'entiers, comme une fonction à valeurs booléennes définissant une relation d'ordre partiel $\lessdot$ sur E :

$\alpha \lessdot \beta$ si et seulement si β est facteur gauche de α .

ppq est défini par les axiomes :

$SH_{6.5}$ = {*ppq* ab ≡ *vrai* ⊃ *ppq succ*a b ≡ *vrai* | a,b ∈ $\overline{\text{PORTEE}}$, a ≠ b}

$SH_{6.6}$ = {*ppq succ* ab ≡ *vrai* ⊃ *ppq* ab ≡ *vrai* | a,b ∈ $\overline{\text{PORTEE}}$, b ≠ *succ* a}

$SH_{6.7}$ = {*ppq* ab ≡ *vrai* ⊃ *ppq desc*a b ≡ *vrai* | a,b ∈ $\overline{\text{PORTEE}}$}

$SH_{6.8}$ = {*ppq desc* b ≡ *vrai* ⊃ *ppq* ab ≡ *vrai* | a,b ∈ $\overline{\text{PORTEE}}$, *desc* a ≠ b}

$SH_{6.9}$ = {*ppq* aa ≡ *vrai* | a ∈ $\overline{\text{PORTEE}}$}

$SH_{6.10}$ = {*ppq succ*a a ≡ *faux* | a ∈ $\overline{\text{PORTEE}}$ - {θ}}

$SH_{6.11}$ = {*ppq* θb ≡ *faux* | b ∈ $\overline{\text{PORTEE}}$ - {θ}} .

Soit $I_p = \mathcal{R}_p(I)$ la restriction d'une information I de $\mathcal{S}$ à la structure $\mathcal{S}_p$ [2.5.1] où $\mathcal{S}_p$ est définie par l'ensemble L_p = {θ , *succ*, *desc*, *vrai*, *faux*} de symboles fonctionnels, l'ensemble $\overline{\text{PORTEE}} \equiv \overline{\text{PORTEE}} \cup \overline{\text{BOOL}} \equiv \overline{\text{BOOL}}$ de ses formules atomiques ($\overline{\text{BOOL}}$ = *ppq* $\overline{\text{PORTEE}}$ $\overline{\text{PORTEE}}$ ∪ {*vrai*, *faux*}) et les schémas d'axiomes propres de $SH_{6.1}$ à $SH_{6.11}$ ainsi que $SH_{4.2}$.

En fait I_p est "indépendante" de I puisqu'il n'y a aucun "lien" entre $\overline{\text{PORTEE}}$ et les autres ensembles de schémas fonctionnels.

Théorème 1 : I_p est consistante.

Démonstration :

Montrons que (E, {$\overline{\text{vrai}}$, $\overline{\text{faux}}$}, r) est une réalisation de I_p où

E est l'ensemble des suites finies d'entiers strictement positifs

r définit une interprétation de L_p :

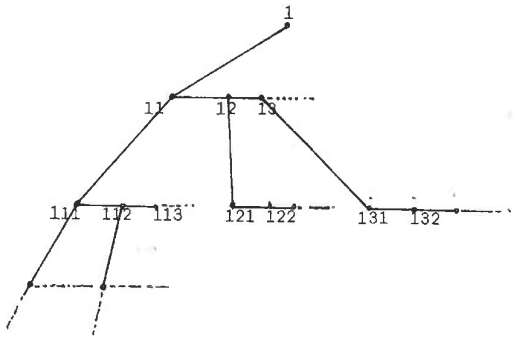
r(θ) = 1

$r(succ) = \overline{\text{succ}} : \begin{cases} a_1a_2\ldots a_n \longrightarrow a_1a_2\ldots(a_n+1) \\ E \longrightarrow E \end{cases}$

$r(dsec) = \overline{desc} : \begin{cases} a_1a_2\cdots a_n \longrightarrow a_1a_2\cdots a_n 1 \\ E \longrightarrow E \end{cases}$

$r(ppq) = \overline{ppq} : \begin{cases} (u\ ,\ v) \longrightarrow \begin{cases} \overline{vrai} \text{ si } v \text{ est facteur gauche de } u \\ \overline{faux} \text{ sinon} \end{cases} \\ E^2 \longrightarrow \{\overline{vrai}, \overline{faux}\} \end{cases}$

On peut schématiser E muni de la relation d'ordre "facteur gauche" par l'arborescence infinie :



Soit $\tilde{r}$ la fonction de vérité associée à r :

$$\tilde{r}(u \equiv v) = \begin{cases} \text{VRAI} & \text{si } \hat{r}(u) = \hat{r}(v) \\ \text{FAUX} & \text{sinon} \end{cases}$$

Il est alors clair que pour tout axiome φ défini par l'un des schémas $SH_{6.1}$ à $SH_{6.4}$, $\tilde{r}(\varphi) = \text{VRAI}$.

Par exemple si φ est l'axiome $\neg$ succ a $\equiv$ desc b

alors $\hat{r}(succ\ a) = \overline{succ}\ (\hat{r}(a))$ se termine par un entier différent de 1

$\hat{r}(desc\ b) = \overline{desc}\ (\hat{r}(b))$ se termine par l'entier 1

d'où $\hat{r}(succ\ a) \neq \hat{r}(desc\ b)$ et donc $\tilde{r}(succ\ a \equiv desc\ b) = \text{FAUX}$,

ainsi $\tilde{r}(\varphi) = \text{VRAI}$.

On démontre de même que pour tout axiome φ défini en $SH_{6.5}$ ... $SH_{6.11}$, $\tilde{r}(\varphi) = vrai$.

Prouvons-le, par exemple, pour φ : $ppq\ succ a\ b \equiv vrai \supset ppq\ ab \equiv vrai$ $(b \neq succ\ a)$:

$\tilde{r}(\varphi) = H_{\supset}\ (\tilde{r}(ppq\ succ\ ab \equiv vrai), \tilde{r}(ppq\ ab \equiv vrai))$

mais $\tilde{r}(ppq\ succ\ ab \equiv vrai) = \text{VRAI}$ si et seulement si $\hat{r}(ppq\ succ\ ab) = \overline{vrai}$

c'est-à-dire si et seulement si $\hat{r}(b)$ est facteur gauche de $\overline{succ}\ \hat{r}(a)$ (1).

D'autre part il résulte immédiatement de la définition de r, de $\overline{succ}$, de $\overline{desc}$ et de $\overline{ppq}$ que $\hat{r}$ est injective.

Alors comme $b \neq succ\ a$, $\hat{r}(b) \neq \hat{r}(succ\ a)$

c'est-à-dire encore : $\hat{r}(b) \neq \overline{succ}\ \hat{r}(a)$ (2)

Si donc $\hat{r}(ppq\ succ a\ b \equiv vrai) = \text{VRAI}$, il résulte de (1) et (2) que

i) $\hat{r}(b) = a_1a_2\cdots a_p$

ii) $\overline{succ}\ \hat{r}(a) = a_1a_2\cdots a_p\cdots a_n$ avec $n > p$ et $a_n \geqslant 2$

ainsi $\hat{r}(b)$ est encore facteur gauche de $\hat{r}(a) = a_1\cdots a_p\cdots (a_n-1)$, c'est-à-dire que $\tilde{r}(ppq\ ab \equiv vrai) = \text{VRAI}$.

On a donc prouvé que $\tilde{r}(\varphi) = \text{VRAI}$.

Enfin il est immédiat que $\tilde{r}(\neg\, vrai \equiv faux) = \text{VRAI}$.

Théorème 2 : I_p est complète.

Démonstration :

D'après [2.4.3 théorème 2] il suffit de prouver que deux réalisations $(E_1, \{vrai_1, faux_1\}, r_1)$ $(E_2, \{vrai_2, faux_2\}, r_2)$ quelconques sont équivalentes ; c'est-à-dire avec [2.4.3 proposition 6] que pour toute formule atomique $u \equiv v$ de $\mathcal{S}_p$:

$$\tilde{r}_1(u \equiv v) = \tilde{r}_2(u \equiv v)$$

ce qui équivaut à :

$$\hat{r}_1(u) = \hat{r}_1(v) \iff \hat{r}_2(u) = \hat{r}_2(v)\ .$$

L'ensemble A_p des formules atomiques de $\mathcal{S}_p$ est la réunion de $\overline{BOOL} \equiv \overline{BOOL}$ et $\overline{PORTEE} \equiv \overline{PORTEE}$

a) $a \equiv b \in \overline{PORTEE} \equiv \overline{PORTEE}$

Lemme : $\forall a,b \in \text{PORTEE}\ (a \neq b \Longrightarrow \neg\, a \equiv b \in I_p)$

Résulte immédiatement des schémas d'axiomes $SH_{6.1}$ à $SH_{6.4}$.

Alors $\forall a,b \in \text{PORTEE}$: si $a = b$ alors $\tilde{r}_1(a \equiv b) = \tilde{r}_2(a \equiv b)$ par définition d'une réalisation et de SL_4

si $a \neq b$ alors $\tilde{r}_1(\neg\, a \equiv b) = \text{VRAI} = \tilde{r}_2(\neg\, a \equiv b)$

donc $\tilde{r}_1(a \equiv b) = \text{FAUX} = \tilde{r}_2(a \equiv b)$

ce qui prouve que les deux réalisations sont équivalentes sur $\overline{PORTEE} \equiv \overline{PORTEE}$.

b) $u \equiv v \in \overline{BOOL} \equiv \overline{BOOL}$

i) il est clair que $\tilde{r}_1(u \equiv v) = \tilde{r}_2(u \equiv v)$ si $u,v \in \{vrai, faux\}$ (avec $SH_{4.2}$)

ii) montrons que pour tout $a,b \in \overline{PORTEE}$:

$$\tilde{r}_1(ppq\ ab \equiv vrai) = \tilde{r}_2(ppq\ ab \equiv vrai)$$

récurrence sur la longueur $|a|$ de a .

$|a| = 1$ $\quad a = \theta$ et donc $(SH_{6.11})$ $ppq\ \theta b \equiv faux \in I_p$

alors $\tilde{r}_1(ppq\ \theta b \equiv faux) = VRAI = \tilde{r}_2(ppq\ \theta b \equiv faux)$

donc $\tilde{r}_1(ppq\ \theta b \equiv vrai) = \tilde{r}_1(ppq\ \theta b \equiv vrai)$ d'après $SH_{4.2}$

$|a| > 1$ deux cas sont alors à envisager :

$ii_1)$ $a = succ\ a_1$

- ou $a_1 = b$ et donc $ppq\ ab \equiv ppq\ succb\ b$

avec $SH_{6.10}$ $ppq\ succb\ b \equiv faux \in I_p$

et donc $\tilde{r}_1(ppq\ ab \equiv vrai) = FAUX = \tilde{r}_2(ppq\ ab \equiv vrai)$

- ou $a = b$ et $\tilde{r}_1(ppq\ ab \equiv vrai) = VRAI = \tilde{r}_1(ppq\ ab \equiv vrai)$ d'après $SH_{6.9}$

- ou $a \neq b$ et $a_1 \neq b$, alors par hypothèse de récurrence :

(1) $\tilde{r}_1(ppq\ a_1\ b \equiv vrai) = \tilde{r}_2(ppq\ a_1\ b \equiv vrai)$.

Mais d'après $SH_{6.5}$ et $SH_{6.6}$:

(2) $\tilde{r}_1(ppq\ a_1 b \equiv vrai \supset ppq\ ab \equiv vrai) = VRAI = \tilde{r}_2(ppq\ a_1 b \equiv vrai \supset ppq\ ab \equiv vrai)$

(3) et $\tilde{r}_1(ppq\ ab \equiv vrai \supset ppq\ a_1 b \equiv vrai) = VRAI = \tilde{r}_2(ppq\ ab \equiv vrai \supset ppq\ a_1 b \equiv vrai)$

de (1), (2) et (3) par définition de $\tilde{r}_1$ à partir de $H_{\supset}$ (2.4.3) on déduit

$$\tilde{r}_1(ppq\ ab \equiv vrai) = \tilde{r}_2(ppq\ ab \equiv vrai)$$

$ii_2)$ $a = desc\ a_1$

- ou $a = b$ et alors $\tilde{r}_1(ppq\ ab \equiv vrai) = VRAI = \tilde{r}_2(ppq\ ab \equiv vrai)$ d'après $SH_{6.9}$

- ou $a \neq b$ et on démontre comme ci-dessus :

$$\tilde{r}_1(ppq\ ab \equiv vrai) = \tilde{r}_2(ppq\ ab \equiv vrai)$$ en utilisant

cette fois les schémas d'axiomes $SH_{6.7}$ et $SH_{6.8}$.

iii) Il résulte de ii) et $SH_{4.2}$ que

$$\forall a,b \in PORTEE\ ,\ \tilde{r}_1(ppq\ ab \equiv faux) = \tilde{r}_2(ppq\ ab \equiv faux)$$

iv) de ii) , iii) , $SH_{4.2}$ et des axiomes logiques on déduit :

$$\forall a,b,c,d \in PORTEE\ ,\ \tilde{r}_1(ppq\ ab \equiv ppq\ cd) = \tilde{r}_2(ppq\ ab \equiv ppq\ cd)$$

ce qui termine la démonstration du théorème.

Théorème 3 : $\forall a,b,c \in \overline{PORTEE}$

$$ppq\ ab \equiv vrai \in I_p \text{ et } ppq\ bc \equiv vrai \in I_p \Longrightarrow ppq\ ac \equiv vrai \in I_p$$

$$ppq\ ab \equiv vrai \in I_p \text{ et } ppq\ ba \equiv vrai \in I_p \Longrightarrow a = b .$$

Démonstration :

En reprenant la réalisation introduite au théorème 1 :

$ppq\ uv \equiv vrai \in I_p \iff \overline{ppq}\ \hat{r}(u)\ \hat{r}(v) = \overline{vrai} \iff$

$\hat{r}(v)$ facteur gauche de $\hat{r}(u)$.

La première affirmation est alors évidente ; l'hypothèse de la seconde entraîne $\hat{r}(a) = \hat{r}(b)$ et donc $a = b$ car il est clair que $\hat{r}$ est injective. Ainsi ppq induit nécessairement une relation d'ordre dans toute réalisation : aucune modification élémentaire ne transforme l'un des symboles fonctionnels *succ*, *desc* ou *ppq*.

Dans le langage pivot Algol 68_o nous supposerons associé [5.2] à tout programme un sous ensemble de $\overline{PORTEE}$ qui permettra de rendre compte de la notion de portée d'une valeur, l'inclusion des régions étant associée à la relation d'ordre engendrée par *ppq* .

A tout exemplaire de valeur u créé par un programme, un axiome : *portée* $u \equiv p$ $(p \in PORTEE)$ associera une portée

En particulier la portée d'une valeur simple (entier, réel, booléen, caractère) et la portée de *nil* sont tout le programme [A-2.2.4.2 b)] et [A-2.2.3.5] (ce sont les seules portées fixes, c'est-à-dire indépendantes d'un programme particulier).

Nous traduirons cette propriété par :

$$SH_{6.12} = \{portée\ poss\ \underline{nil} \equiv \theta\ ,\ portée\ poss\ x \equiv \theta \mid x \in L_o^{ent} \cup L_o^{re} \cup L_o^{bool} \cup L_o^{car}\} .$$

D'autre part, la portée d'un exemplaire de valeur structurée (resp. multiple) est la plus petite des portées de ses champs (resp. éléments) [A-2.2.4.2 b)] , ce qu'on traduit par :

$SH_{6.13}$ = {*mode* u ≡ *structure* $\underset{x_1 \ldots x_n}{\mu_1 \ldots \mu_n}$ ∧ $\bigwedge_{1 \leqslant i \leqslant n}$ (*ppq portée* ch_{x_j} u *portée* ch_{x_i} u ≡ *vrai*) ⊃ *portée* u ≡ *portée* ch_{x_j} u | $n \geqslant j \geqslant 1$; $x_i \in L_o^{id}$; $\mu_i \in$ MODE ; $u \in$ EXSTRU}

$SH_{6.14}$ = {*dim* u ≡ *n* ∧ ($\bigwedge_{1 \leqslant j \leqslant n}$ a_j u ≡ α_j ∧ b_j u ≡ β_j) ∧ $\bigwedge_{\alpha_j \leqslant i_j, i_j^o \leqslant \beta_j}$ *ppq portée* $elem_n$ u $i_1^o \ldots i_n^o$ *portée* $elem_n$ u $i_1 \ldots i_j$ ≡ *vrai*) pour $1 \leqslant j \leqslant n$

⊃ *portée* u ≡ *portée* $elem_n$ u $i_1^o \ldots i_n^o$ | $n \geqslant 1$, $n \in$ NOTENT ; $\alpha_j, \beta_j \in$ NOTENT ; $i_1^o, \ldots, i_n^o, i_1, \ldots, i_n \in$ NOTENT ; u ∈ EXMUL} .

Enfin la portée d'une routine (resp. d'un nom) est définie par le système associé à sa notation [6.3.10] (resp. à son générateur [6.3.7]).

4.1.5.7 Schémas d'axiomes relatifs aux modes :

- Rappelons tout d'abord que les symboles *entier*, *réel* ... sont des abréviations des schémas fonctionnels *indique ent*, *indique reel* ... [4.1.2.1 i)] .
- Le schéma fonctionnel *indique* D sera défini en [6.2.2] pour tout déclarateur ou indicateur de mode D par un système d'axiomes $\mathfrak{V}(D)$ (il sera caractérisé à l'aide des déclarateurs ou indicateurs apparaissant dans D). Cette démarche, qui place sur le même plan indicateurs de modes et déclarateurs, permet de définir simplement les modes récursifs.
- Pour simplifier notre description nous avons défini $union_1$ par :

$SH_{7.1}$ = {$union_1$ μ ≡ μ | μ ∈ MODE} .

- Les axiomes qui suivent expriment [A-2.2.4.1 b) et c)] :

$SH_{7.2}$ = {*mode poss b* ≡ *booleen* ∧ *mode poss c* ≡ *caractère* | $b \in L_o^{bool}$, $c \in L_o^{car}$

$SH_{7.3}$ = {*mode poss e* ≡ $long^{(n-1)}$*entier* ∧ *mode poss r* ≡ $long^{(n-1)}$*reel* | $n \geqslant 1$; $e \in L_o^{entn}$; $r \in L_o^{reeln}$} .

Les modes des exemplaires de valeurs possédés par une "notation" de structure, de valeur multiple ou de routine seront précisés dans la définition de ces notations : [6.4.6] et [6.3.10] (nous généralisons le terme Algol 68 "notation" au cas de certaines propositions collatérales qui permettent de définir des valeurs multiples et des valeurs structurées). Précisons simplement les axiomes :

$SH_{7.4}$ = {*mode* u ≡ *structure* $\underset{x_1 \ldots x_n}{\mu_1 \ldots \mu_n}$ ⊃ $\bigwedge_{1 \leqslant i \leqslant n}$ *mode* ch_{x_i} u ≡ μ_i | $n \geqslant 1$; $\mu_1, \ldots, \mu_n \in$ MODE ; $x_1, \ldots, x_n \in L_o^{id}$; u ∈ EXSTRU }

$SH_{7.5}$ = *mode* u ≡ $rang^{(n)}$ μ ∧ ($\bigwedge_{1 \leqslant j \leqslant n}$ *inf* a_j u i_j ≡ *vrai* ∧ *inf* i_j b_j u ≡ *vrai*) ⊃ *mode* $elem_n$ u $i_1 \ldots i_n$ ≡ μ | $n \geqslant 1$; $i_1 \ldots i_n \in$ NOTENT ; μ ∈ MODE - rang MODE ; u ∈ EXMUL } .

D'autre part les schémas d'axiomes $SH_{1.4}$ à $SH_{1.9}$ traitent le cas des noms ; $SH_{2.3}$ est relatif aux tranches. Rappelons enfin que $SH_{3.4}$ exprime en partie [A-2.2.4.1 a)].

- Il faut également introduire un certain nombre d'axiomes relatifs à l'égalité de deux modes. Ces axiomes seront essentiellement utilisés lors de la comparaison de 2 modes pour les relations de conformité [6.3.4]

$SH_{7.6}$ = {¬ *indique* a ≡ *indique* b | a,b ∈ L_o^{prim} , a ≠ b}

$SH_{7.7}$ = { φ μ_1 ≡ φ μ_2 ⊃ μ_1 ≡ μ_2 | $\mu_1, \mu_2 \in$ MODE ; φ ∈ {*long*, *repère*, *rang*}}

$SH_{7.8}$ = { *structure* $\underset{x_1 \ldots x_n}{\mu_1 \ldots \mu_n}$ ≡ *structure* $\underset{x_1 \ldots x_n}{\nu_1 \ldots \nu_n}$ ⊃ $\bigwedge_{1 \leqslant i \leqslant n}$ μ_i ≡ ν_i | $\mu_i \in$ MODE ; $\nu_i \in$ MODE ; $x_i \in L_o^{id}$; $n \geqslant 1$}

$SH_{7.9}$ = {$procn_n$ $\mu_1 \ldots \mu_n$ ≡ $procn_n$ $\nu_1 \ldots \nu_n$ ⊃ $\bigwedge_{1 \leqslant i \leqslant n}$ μ_i ≡ ν_i | μ_i , $\nu_i \in$ MODE ; $n \geqslant 0$}

$SH_{7.10}$ = {$procr_n$ $\mu_1 \ldots \mu_n$ ≡ $procr_n$ $\nu_1 \ldots \nu_n$ ⊃ $\bigwedge_{1 \leqslant i \leqslant n}$ μ_i ≡ ν_i | μ_i , $\nu_i \in$ MODE , $n \geqslant 1$}

$SH_{7.11}$ = {$union_n$ $\mu_1 \ldots \mu_n$ ≡ $union_n$ $\nu_1 \ldots \nu_n$ ⊃ $\bigwedge_{1 \leqslant i \leqslant n}$ μ_i ≡ ν_i | μ_i , $\nu_i \in$ MODE ; $n \geqslant 1$}

$SH_{7.12}$ = {¬ $\varphi_1 \ldots \varphi_n$ ≡ $\psi_1 \ldots \psi_k$ | $\varphi_1 \ldots \varphi_n \in$ MODE ; $\psi_1 \ldots \psi_k \in$ MODE ; $n,k \geqslant 1$; $\varphi_1 \neq \psi_1$; $\varphi_1, \psi_1 \in$ {long, repère, rang, $\underset{x_1 \ldots x_n}{\text{structure} \ldots}$} ∪ MODE}

Commentaires :

- Le premier schéma d'axiomes précise que tous les modes de "base" sont différents ; les cinq suivants traduisent l'injectivité des différents symboles fonctionnels permettant de définir les modes.

- En particulier $SH_{7.11}$ nécessite une explication :

De la définition des modes Algol 68 [A-1.2] il ressort que, par exemple, union reel entier et union entier reel sont deux objets distincts en tant que production terminale de MODE ; d'autre part il est précisé en [A-4.4.3 d)] (conditions sur les modes) que ces deux modes seront confondus.

Dans notre formalisation nous supposerons *union*$_n$ "injective" ($SH_{7.11}$) et pour tenir compte de [A-4.4.3 d)] nous supprimerons le problème en exigeant, au niveau du langage pivot [5.2] que si union $(D_1,D_2,\ldots,D_n)$ est un déclareur apparaissant dans un programme P , aucun déclareur union $(D_{\ell(1)},\ldots D_{\ell(n)})$ ne peut apparaître dans P si ℓ est une permutation de $\{1,2,\ldots,n\}$ différente de l'identité.

- Le dernier schéma précise que deux modes ne commençant pas par le même symbole sont différents.

4.1.5.8 Schémas d'axiomes caractérisant le symbole *sub* :

sub est un symbole fonctionnel 2-aire qui permet de traiter les variables locales et les paramètres formels d'une routine. Il est complètement défini en [6.3.11] et ce n'est que par souci d'unité que nous indiquons ici les schémas d'axiomes qui le caractérisent :

$$SH_{8.1} = \{poss\ G \equiv F' \supset \neg sub\ G(E_1,\ldots,E_n)\ x \equiv y \mid F',G \in L_o^{ph}\ ;\ x \in L_o^{F'}\ ;\ y \in IDENT \text{ et } y \neq sub\ G(E_1,\ldots,E_n)\ x\ ;\ E_i \in IDENT\}$$

($L_o^{F'}$ est l'ensemble des identificateurs locaux à la routine F')

$$SH_{8.2} = \{poss\ G \equiv F' \supset sub\ G(E_1,\ldots,E_n)\ x \equiv x \mid x \in IDENT - L_o^{F'} \cup \{\sim_i \mid 1 \leqslant i \leqslant n\}\ ;\ F',G \in L_o^{ph}\}$$

$$SH_{8.3} = \{sub\ A_p x \equiv sub\ B_p y \supset x \equiv y \wedge A_p \equiv B_p \mid x,y \in L_o^{id}\ ;\ A_p,B_p \in L_o^{ph}\}$$

$$SH_{8.4} = \{sub\ A_p\ f\ u_1\ldots u_n \equiv f\ sub\ A_p\ u_1 \ldots\ sub\ A_p\ u_n \mid u_1,\ldots,u_n \in S,\ A_p \in L_o^{ph}\}$$

$$SH_{8.5} = \{sub\ A_p\ \underline{rep}\ D \equiv \underline{rep}\ sub\ A_p\ D \mid D \in INDIC,\ A_p \in L_o^{ph}\}$$

$$SH_{8.6} = \{sub\ A_p\ \underline{struct}\ (D_1x_1,\ldots,D_nx_n) \equiv \underline{struct}(sub\ A_p\ D_1x_1,\ldots,sub\ A_p\ D_nx_n) \mid D_1,\ldots,D_n \in INDIC\ ;\ A_p \in L_o^{ph}\ ;\ x_1,\ldots,x_n \in L_o^{id}\}$$

$$SH_{8.7} = \{sub\ A_p\ [A_1\Gamma_1 : B_1\Delta_1,\ldots,A_n\Gamma_n : B_n\Delta_n]\ D \equiv [sub\ A_p\ A_1\Gamma_1 : sub\ A_p\ B_1\Delta_1,\ldots, sub\ A_pA_n\Gamma_n : sub\ A_pB_n\Delta_n]\ sub\ A_pD \mid [A_1\Gamma_1:B_1\Delta_1,\ldots,A_n\Gamma_n:B_n\Delta_n]\ D \in INDIC,\ A_p \in L_o^{ph}\}$$

$$SH_{8.8} = \{sub\ A_p\ D \equiv D \mid D \in L_o^{prim} \cup L_o^{\overline{nprim}}\ ;\ A_p \in L_o^{ph}\}$$

où $L_o^{\overline{nprim}}$ est l'ensemble des déclareurs non primitifs qui ne commencent pas par *struct*, *rep* ou [.

$$SH_{8.9} = \{sub\ G(E_1,\ldots,E_n) \sim_i \equiv E_i \mid 1 \leqslant i \leqslant n\ ;\ G(E_1,\ldots,E_n) \in L_o^{ph}\}$$

qui permet de rendre compte de la substitution des paramètres effectifs aux paramètres formels lors d'un appel.

4.2 Les modifications élémentaires de la structure $\mathcal{S}$:

4.2.1 Introduction :

Rappelons qu'une modification élémentaire π est définie [2.6] par la donnée d'une bijection σ_π de L sur un ensemble L', et d'un ensemble X_π d'axiomes ; nous noterons $\mathcal{M}od$ leur ensemble.

Par souci de clarté nous définirons chaque modification là où elle sera utile au chapitre [6] . Une exception toutefois concerne les schémas de modifications j et $\bar{j}$, dont l'emploi n'est pas local, et qui seront définis en [4.2.2] . La liste de tous les schémas de modifications élémentaires est donnée en annexe 2.

4.2.2 Les schémas de modifications élémentaires j et $\bar{j}$:

j (resp. $\bar{j}$) permet de "réduire" le domaine de définition d'une modification qui est l'interprétation d'un calcul. Il est utilisé en particulier dans la définition du schéma de calculs associé à une proposition conditionnelle [6.4.5] . j (resp. $\bar{j}$) est un schéma à 2 paramètres.

4.2.2.1 Définition de j :

j est construit de telle sorte que j(u,v) soit l'identité sur l'ensemble des informations I contenant le théorème $u \equiv v$ $(u,v \in S)$ et ne soit "pas défini" sur l'ensemble des informations contenant le théorème $\neg u \equiv v$:

a) $\sigma_{j(u,v)}$ est l'identité sur L

b) $X_{j(u,v)} = \{u \equiv v\}$.

Si I est une information complète deux cas peuvent se produire :

- $u \equiv v \in I$ et alors $j(u,v)(I) = I$
- $\neg u \equiv v \in I$ et alors $j(u,v)(I) = F$ (information inconsistante).

Si I est consistante sans être complète, $u \equiv v$ et $\neg u \equiv v$ peuvent ne pas appartenir à I , alors j(u,v)(I) est l'extension de I obtenue par adjonction de l'axiome $u \equiv v$.

Remarque :

Tout calcul commençant par j(u,v) s'interprétera comme une modification "définie" seulement sur les informations I telles que $\neg u \equiv v \notin I$.

4.2.2.2 Définition de $\bar{j}$:

Avec des remarques analogues à celles de [4.4.2.1] :

a) $\sigma_{\bar{j}(u,v)}$ est l'identité sur L

b) $X_{\bar{j}(u,v)} = \{\neg u \equiv v\}$.

5

Le langage pivot

Algol 68

5. LE LANGAGE PIVOT ALGOL 68_o

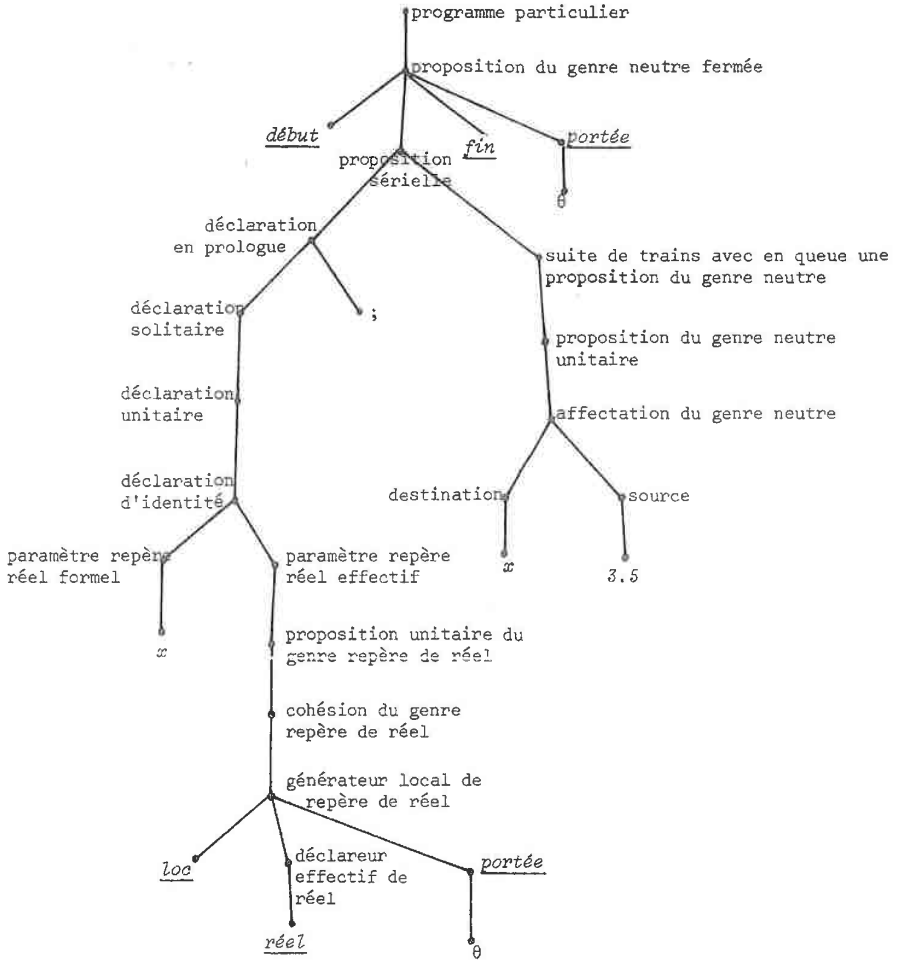
5.1 Introduction :

Nous avons signalé en [3.1] que ce travail ne comporte pas d'étude théorique de la notion de langage pivot. Ainsi dans la suite nous contenterons-nous de définir informellement le langage pivot Algol 68_o, sans insister sur la structure syntaxique du programme pivot.

Par exemple lorsqu'on parlera du programme P :

début réel x = loc réel ; x:=3.5 fin

plus que la suite de caractères on sous-entendra l'arbre syntaxique suivant (dans lequel on a supprimé certaines branches intermédiaires) :



Remarquons qu'à la ramification engendrée par la double grammaire on ajoute certaines branches qui représentent les portées de certains composants d'une phrase (régions et générateurs) [5.2 c)]).

Remarque 1 :

Si nous définissons complètement la forme d'un langage pivot (ainsi que le processus de traduction), plus que des ramifications (ou des arbres syntaxiques) il serait intéressant de considérer des schémas fonctionnels et d'unifier ainsi complètement la description. La démarche pourrait alors être la suivante :

a) Adjonction à l'ensemble des symboles fonctionnels de la structure d'information de certains symboles qui permettraient de définir un programme pivot comme étant un schéma fonctionnel. Par exemple la phrase x:=3.5 pourrait être représentée par le schéma fonctionnel affect x 3.5.

b) Adjonction au système Σ définissant les ensembles de schémas fonctionnels de la structure [2.3] d'un certain nombre d'équations caractérisant les ensembles de schémas qui seraient les phrases du langage pivot. Ces équations permettraient une définition de la syntaxe de ce langage analogue à celle d'une grammaire. (Il est clair que dans cette hypothèse on ne considèrerait plus l'ensemble L_o^{ph} [4.1.2]).

c) Association à chaque schéma fonctionnel du type précédent d'un double système de calculs et d'accès.

d) Définition de la syntaxe du langage évolué (syntaxe concrète).

Une telle voie de recherche serait à développer, d'autant plus qu'elle permettrait de définir en même temps et à l'aide des mêmes outils syntaxe et sémantique.

Remarque 2 :

La méthode que nous utilisons, qui consiste à définir la sémantique d'un programme comme étant celle du programme pivot associé est contenue dans [A] . Par exemple dans la définition d'une affectation on parle [A-8.3.1.2 d)] de l'élaboration de la destination et de la source, ce qui sous-entend qu'on connait l'arbre syntaxique associé à la double grammaire engendrant Algol 68.

La définition d'Algol 68, utile pour la description de la sémantique, sera indiquée au fur et à mesure de l'étude en [6] de ses différentes phrases. Nous nous bornerons ici [5.2] à indiquer sommairement les traits essentiels de ce langage pivot pour convaincre le lecteur qu'à deux exceptions près [5.2 j)] on peut traduire Algol 68 en Algol 68_o .

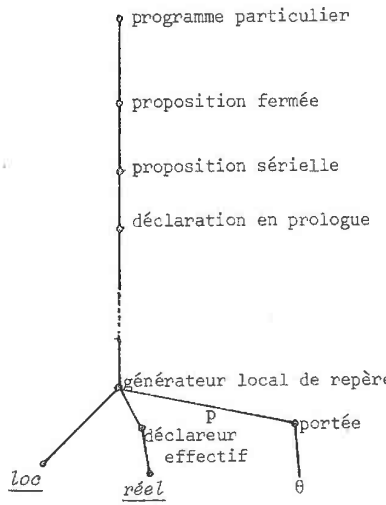
5.2 Définition rapide d'Algol 68_o :

a) Algol 68_o est défini à partir du langage strict Algol 68 : aucune extension (contraction, instruction d'itération ...) n'est admise ; c'est-à-dire que les possibilités décrites en [A-9] n'existent pas dans le langage pivot.

b) Un programme du langage pivot ne contient aucune instruction de saut. Rappelons [3.3.3.2 f)] qu'il est possible de remplacer un saut par un appel de procédure dont la dernière instruction est un ordre d'arrêt noté "*stop*".

"*stop*" joue le rôle d'une instruction de saut à "exit" [A-2.1 e)], il lui sera associé une modification élémentaire notée $\widetilde{stop}$ qui ne sera pas définie : l'opérateur ζ a pour rôle de tronquer chaque calcul contenant *stop* à partir de cette modification. Par exemple si $\widetilde{E}_1 \,.\, \widetilde{stop} \,.\, \widetilde{E}_2$ est un calcul, $\zeta(\widetilde{E}_1 \,.\, \widetilde{stop} \,.\, \widetilde{E}_2) = \zeta(\widetilde{E}_1)$. ζ est une fonction de base du système à point fixe associé à tout programme Algol 68_o.

c) Certaines "phrases" Algol 68_o, qui sont des ramifications, (générateurs, notations de routines, propositions sérielles) contiennent une branche dont la feuille est un schéma fonctionnel appartenant à PORTEE (c'est-à-dire interprétable comme une portée). L'association d'une portée à une phrase est possible dans le langage pivot car il s'agit d'une notion statique. Ainsi la portée du générateur *loc réel* dans l'exemple de [5.1] est θ , ce générateur étant la ramification :



Dans la suite nous conviendrons de ne représenter que les feuilles de telles ramifications, étant toujours conscient de la signification de cette notation.

Plus précisément nous noterons un générateur (resp. une proposition sérielle, une notation de routine) sous la forme *loc D* | *p* (resp. $E_1;E_2;\ldots;E_n$ | *p* , $(D_1z_1$ & D_2z_2 ... & $D_nz_n)$ D : E | *p*) où *p* est un schéma fonctionnel appartenant à PORTEE.

L'ensemble des portées associées aux phrases d'un programme permet de rendre compte de la notion de région [A-4.1.1], la relation définie par une interprétation du symbole *ppq* permet de comparer certaines portées.

d) Tout déclareur virtuel, formel ou effectif de valeur multiple est de la forme : $[A_1.\Gamma_1 : B_1.\Delta_1 , \ldots, A_n.\Gamma_n : B_n.\Delta_n]$ D où D n'est pas un déclareur de valeur multiple ; Γ_i (resp. Δ_i) est égal à 0 si la borne correspondante est flexible, 2 si elle est suivie de adlib et 1 sinon.
A_i (resp. B_i) est soit un tertiaire, soit le symbole 0-aire *vide* que nous introduisons à cet effet dans le vocabulaire (plus précisément, ce que nous venons de définir comme étant un déclareur est formé en fait des feuilles de la ramification qui est ce déclareur).

e) Toute tranche est de la forme :

E $[A_1.\Gamma_1.B_1$ apd $\Delta_1 , \ldots, A_n.\Gamma_n.B_n$ apd $\Delta_n]$ où Γ_i est égal à 1 si le $i^{ème}$ indexeur est un massicot [A-8.6.1.1 b)], et égal à 0 si c'est un indice [A-8.6.1.1 i)]. Par exemple dans *t* [*3:5,1*] le premier indexeur est le massicot *3:5* , le deuxième est l'indice *1*.

f) Les modifications du langage Algol 68 sont effectuées au niveau du langage pivot :

- l'élargissement [A-8.2.5] s'exprime simplement à l'aide du symbole fonctionnel *elarg* [4.1.2.3] : Algol 68_o place ce symbole devant chaque phrase devant être élargie. Rappelons que ni les complexes, ni les bits, ni les catènes n'appartiennent au langage pivot, ainsi le seul élargissement autorisé est celui d'entier à réel. *elarg* est caractérisé par le schéma d'axiomes $SH_{4.3}$.

- Le dereperage [A-8.2.1] est traduit à l'aide du symbole fonctionnel *derep* [4.1.2.3] (défini par le schéma d'axiomes $SH_{1.10}$) que place le langage pivot devant chaque phrase à dereperer.

- Le procédurage [A-8.2.3] s'exprime simplement à l'aide de la notation :
( ) D : E qui représente une procédure sans paramètre, à résultat de genre D (éventuellement neutre).

- Le déprocédurage [A-8.2.2] s'exprime simplement à l'aide de la notation :
G( ) qui représente l'appel d'une procédure sans paramètre.

- Le rangement [A-8.2.6] peut s'exprimer à l'aide de la notation :
(E) qui représente une proposition collatérale ne contenant qu'un seul unité E [A-6.2.1]. Remarquons qu'ici la forme ramifiée des phrases du langage pivot est indispensable : la simple chaîne de caractère ne permet pas d'exprimer, à elle seule, qu'il s'agit d'une proposition collatérale.

- Les autres modifications (unir, hisser, neutraliser) ne donnent lieu à aucune action.

Ainsi à toute phrase Algol 68_o nous supposerons associé son mode (contextuel au sens de [A 3]). En conséquence le langage pivot ne contiendra pas de forceur.

g) Toutes les déclarations d'identificateurs et d'indicateurs (de modes et d'opérations) portent sur des éléments différents : bien qu'elle existe, la structure de bloc ne sera pas utilisée, ceci afin d'éviter d'introduire la notion de domaine de validité d'un identificateur (resp. indicateur).

h) Les déclareurs d'unions vérifient la propriété suivante :
si *union*$(D_1,\ldots,D_n)$ est une phrase du langage pivot, alors pour toute bijection σ de $\{1,2,\ldots,n\}$ non réduite à l'identité le déclareur *union*$(D_{\sigma(1)},\ldots,D_{\sigma(n)})$ n'appartient pas à Algol 68_o .

Cette condition permet une résolution plus simple du problème de l'égalité des modes [A-6.3.4].

i) Les symboles fantômes [A-8.2.7.1] apparaissant dans un programme Algol 68_o sont nécessairement de genre neutre. Pour définir une notation de routine nous introduisons des symboles particuliers $\sim_i$ [6.3.10] .

j) Enfin signalons que la définition de la sémantique d'une proposition collatérale contrôlée, des formats et des entrées/sorties n'a pas été abordée dans ce travail.

6

Sémantique d'Algol 68

6. Formalisation de la sémantique d'Algol 68_o

6.1 Introduction :

6.1.1 Généralités :

Tour programme Algol 68_o est formé [A-2.1] de phrases [A-6] de différents types. Pour définir le double système $\mathfrak{S}(P)$ de schémas de calculs et de schémas d'axiomes associé à un programme P , il suffit de décrire le double système associé à chaque phrase de P .

Parmi les différentes phrases Algol 68_o on distingue :

a) les phrases élémentaires ; ce sont :

- soit des déclarations unitaires [A-7.0.1 a)]
- soit des propositions unitaires [A-8.1.1] qui ne sont pas des propositions parenthésées (i.e. fermée, conditionnelle ou collatérale, par exemple affectation, relation d'identité, générateur etc...)

b) les phrases composées qui sont :

- soit des déclarations collatérales [A-6.2.1 a)]
- soit des propositions "non élémentaires" (fermées, sérielles, conditionnelles ou collatérales).

Dans la suite du chapitre on associe à chaque type de phrase un double système (en [6.2] on traite le cas des déclarations unitaires, en [6.3] celui des propositions unitaires non fermées et en [6.4] celui des phrases composées) et il est clair que cette définition est récursive (par exemple dans le cas des phrases composées). En particulier dans la décomposition d'un programme en phrases on passe d'une structure composée à une structure de plus en plus simple pour arriver à des constituants formés de phrases élémentaires auxquelles sont associés des schémas de calculs définis immédiatement à partir de modifications élémentaires.

Les notations utilisées sont celles des chapitres précédents.

6.1.2 Schéma de calculs associé à une phrase :

Si P est une phrase ou un déclarateur Algol 68_o , P représente le symbole fonctionnel et $\tilde{P}$ le schéma de calcul qui lui sont associés ([3.3.1] et [3.4.1]).

Pour généraliser cette deuxième notation à certains schémas fonctionnels u, on souhaite définir $\tilde{u}$ de telle sorte que, si u est formellement égal au symbole 0-aire P alors $\tilde{u} = \tilde{P}$.

Pour cela on introduit un schéma de calculs à un paramètre ψ :

(1) $$\overset{\sim}{\psi} = \bigcup_{E \in L_o^{ph} \cup L_o^{nprim}} \lambda u \,.\, j\,(u,E)\ \tilde{E} \quad .$$

Ceci permet, par exemple, de définir l'ensemble de calculs associés à un indicateur de mode *d* comme étant l'ensemble de calculs du déclareur D si *d* est caractérisé par la délcaration de mode : *mode* *d* = D .
(Il faut bien remarquer que ce schéma de calculs ne présente d'intérêt que pour des objets externes qui ne sont pas caractérisés directement par un système de calculs).

L'équation (1) appartient au système de calculs $\mathcal{M}(P)$ pour tout programme P. Dans la suite par abus de notation il nous arrivera d'écrire $\tilde{u}$ pour $\overset{\sim}{\psi}(u)$.

6.2 Systèmes associés aux déclarations unitaires :

6.2.1 Introduction :

Une déclaration [A-7.0.1] est - soit une déclaration de mode
- soit une déclaration de priorité
- soit une délcaration d'identité
- soit une déclaration d'opération.

Remarquons tout d'abord que la notion de priorité est de nature syntaxique, elle est traitée au niveau du langage pivot Algol 68_o et n'apparaît pas ici.
Nous allons nous intéresser successivement aux trois autres types de déclarations.

6.2.2 Déclaration de mode :

Rappelons birèvement que l'ensemble des symboles fonctionnels L contient :

- L_o^{prim} = {*reel*, *ent*, *car*, *bool*, *proc*} ensemble des déclarateurs primitifs
- *long*, *repère*, *structure* $_{x_1 \ldots x_n}$, *rang* ... symboles fonctionnels qui permettent de définir des schémas fonctionnels interprétables comme des modes Algol 68
- *indique* symbole fonctionnel qui associe à un déclarateur de mode, le mode qu'il spécifie
- L_o^{nprim} ensemble des déclarateurs non primitifs (par exemple *long long reel*, *struct* (*reel* *x*, *ent* *y*) ...)
- L_o^{ind} ensemble des indicateurs de modes.

En [6.2.2.1] on associe un double système à tout déclarateur non primitif D , en particulier le système d'accès $\mathcal{V}(D)$ est un ensemble d'axiomes de la structure $\mathcal{S}$; en [6.2.2.2] on associe un double système à tout indicateur de mode.

6.2.2.1 Systèmes associés à un déclarateur de mode non primitif :

Soit D un déclarateur non primitif et

$$\mathcal{S}(D) \begin{cases} \mathcal{M}(D) & \text{le double système associé} \\ \mathcal{V}(D) \end{cases}$$

- $\mathcal{V}(D)$ caractérise le schéma *indique* D , il sera défini à l'aide de [A-7.1.1] .
- Les déclarateurs seront utilisés essentiellement dans les déclarations d'identité [6.2.3] et les générateurs [6.3.7] . Pour rendre compte de l'élaboration de ces phrases dans notre formalisation, il faudra exprimer l'élaboration collatérale des

bornices d'un déclarateur [A-7.4.2 pas 2] et [A-7.1.2 pas 5] . Pour cette raison nous définirons le schéma de calculs D associé à un déclareur $\tilde{D}$ comme étant le schéma représentant l'élaboration collatérale des bornices de D (s'ils existent). Cette définition sera bien entendu récursive et $\mathcal{M}(D)$ contiendra l'équation définissant $\tilde{D}$ (si D est un déclareur de valeur multiple) et les équations de $\mathcal{M}(D')$ pour tout sous déclareur D' de D .

- De même que nous devons rendre compte de l'élaboration des bornices éventuels, nous devons pouvoir accéder à la valeur de ces bornices : si D est un déclareur de valeur multiple de dimension n , a_i D (resp. b_i D , s_i D , t_i D) ($1 \leqslant i \leqslant n$) est un schéma fonctionnel qui représentera la $i^{ème}$ borne inférieure (resp. la $i^{ème}$ borne supérieure, $i^{ème}$ "état" inférieur, $i^{ème}$ "état" supérieur) de D (éventuellement *vide*). Ainsi *poss* a_i D (resp. *poss* b_i D) permet d'accéder aux valeurs des bornes de D ; s_i D (resp. t_i D) définit le type (fixe, flex, adlib) de la $i^{ème}$ borne inférieure (resp. supérieure).

a_i D (resp. b_i D, s_i D, t_i D) seront caractérisés par des axiomes appartenant à $\mathcal{V}(D)$.

- Enfin puisque la définition de $\mathcal{S}(D)$ est récursive, il faut préciser ce double système pour D $\in L_o^{prim}$, Un tel déclareur ne contient ni bornices, ni sous déclareurs alors $\mathcal{M}(D) = \emptyset$. De plus les éléments de L_o^{prim} sont des déclareurs primitifs, donc $\mathcal{V}(D) = \emptyset$.

a) D est de la forme *long* D' :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\ \mathcal{M}(D')^{(*)}\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{long indique } D'\end{cases}$$

résulte immédiatement de [A-7.1.1 d)] et des considérations précèdentes

b) D est de la forme *struct* $(D_1 x_1, \ldots, D_n x_n)$:

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D_i) \quad i = 1,\ldots,n\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{structure indique } D_1 \ldots \textit{indique } D_n \\ & \qquad\qquad x_1 \ldots x_n\end{cases}$$

(résulte de [A-7.1.1] de e) à k)]).

(*) En fait D' ne peut être qu'un déclareur d'entier ou de réel et donc $\mathcal{M}(D') = \emptyset$.

c) D est de la forme *rep* D' :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D')\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{repère indique } D'\end{cases}$$

([A-7.1.1] l) à n))

d) D est un déclarateur de routines :

d1) D est de la forme *proc* $(D_1,\ldots,D_n)$:

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D_i) \quad i=1,\ldots,n\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{proc}_n \textit{ indique } D_1 \ldots \textit{indique } D_n\end{cases}$$

d2) D est de la forme *proc* $(D_1,\ldots,D_n)$ D_{n+1} :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D_i) \quad i=1,\ldots,n+1\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{procr}_{n+1} \textit{indique } D_1 \ldots \textit{indique } D_n \textit{ indique } D_{n+1}\end{cases}$$

d3) D est de la forme *proc* D_1 :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D_1)\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{procr}_1 \textit{ indique } D_1\end{cases}$$

(résultent de [A-7.1.1] ii) à bb)) .

Rappelons ici que le déclarateur *proc* est un élément de L_o^{prim} .

e) D est de la forme *union* $(D_1,\ldots,D_n)$:

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D) & \{\mathcal{M}(D_i) \quad i=1,\ldots,n\\ \mathcal{V}(D) & \{\ \textit{indique } D \equiv \textit{union}_n \textit{ indique } D_1 \ldots \textit{indique } D_n\end{cases}$$

[A-7.1.1] cc) à jj) :

De plus la définition de $union_n$ a été généralisée au cas n=1 :

$$union_1\ D \equiv D \qquad (SH_{7.1})$$

f) D est de un déclarateur de valeur multiple :

D est de la forme $[A_1.\Gamma_1 : B_1\Delta_1, \ldots, A_n.\Gamma_n : B_n\Delta_n]$ D' où A_i (resp. B_i) est soit un tertiaire, soit le symbole Algol 68_o *vide* [4.5.2] . En particulier si

D est un déclarateur virtuel alors pour tout $1 \leqslant i \leqslant n$ A_i (resp. B_i) est *vide*. Γ_i (resp. Δ_i) prend la valeur 0 (resp. 2,1) si la $i^{ème}$ borne inférieure (resp. supérieure) est flex (resp. adlib, fixe).

Rappelons que nous notons encore *vide* le symbole 0-aire associé au symbole Algol 68_o *vide*.

Le double système associé à D est défini par :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D)\begin{cases}\tilde{D} = \underset{1\leqslant i\leqslant n}{\mathfrak{m}} (\tilde{A}_i , \tilde{B}_i)\\ \mathcal{M}(A_i) \quad i=1,\dots,n\\ \mathcal{M}(B_i) \quad i=1,\dots,n\\ \mathcal{M}(D')\end{cases}\\ \mathcal{V}(D)\begin{cases}\textit{indique } D \equiv \textit{rang}^{(n)}\ D'\\ a_i\ D \equiv A_i\\ b_i\ D \equiv B_i\\ s_i\ D \equiv \Gamma_i \qquad i=1,\dots,n\\ t_i\ D \equiv \Delta_i\\ \textit{dim } D \equiv n\end{cases}\end{cases}$$

Commentaires :

- Le schéma de calculs $\tilde{D}$ sera utilisé en [6.2.3.3] et [6.3.7.3 c)]
- Les systèmes $\mathcal{M}(A_i)$ et $\mathcal{M}(B_i)$ définissent les schémas $\tilde{A}_i$ et $\tilde{B}_i$; leur définition est donnée en [6.3] lorsque A_i ou B_i est un tertiaire et ci-dessous si A_i ou B_i est *vide*.
- Le premier axiome de $\mathcal{V}(D)$ définit le mode associé au déclareur D
- Les axiomes suivants permettent de définir *poss* a_i D (resp. *poss* b_i D) lorsque A_i (resp. B_i) est un tertiaire ; dans l'hypothèse contraire on obtient le théorème a_i D ≡ *vide* (resp. b_i D ≡ *vide*) ce qui permet de tester l'existence ou la non existence de bornices [6.2.3.6].

Précisons le double système associé à *vide*

$$\mathcal{S}(\textit{vide})\begin{cases}\mathcal{M}(\textit{vide}) \quad \{\widetilde{\textit{vide}} = \Lambda\\ \mathcal{V}(\textit{vide}) = \emptyset \quad \text{c'est-à-dire est le système ne contenant aucune équation}\end{cases}$$

Λ est l'élément neutre de $\mathcal{M}od^{\infty}$ (pour la concaténation). On peut l'interpréter comme la modification identique.

g) Remarque :

Le schéma de calculs $\tilde{D}$ associé à un déclareur de valeur multiple ne contient pas les schémas de calculs des "sous déclareurs" de D .

Ceci posera un problème dans la formalisation d'une déclaration d'identité [6.2.3].

Exemple 1 :

L'élaboration de

$$\overbrace{[a*b : 4]\ \textit{struct}\ (\underbrace{[1 : 3*(a+b)]\ \textit{ent}\ x_1,\ \textit{reel}\ x_2}_{D'})}^{D}\ t = z$$

nécessite "l'élaboration collatérale de tous les bornices contenus dans le déclareur D mais non contenus dans un bornice lui-même contenu dans D" [A-7.4.2 pas 2] . Dans cet exemple il faudra donc élaborer collatéralement : $a*b$, 4 , 1 , $3*(a+b)$. Cette élaboration n'est pas exprimée simplement par le schéma $\tilde{D}$ puisqu'il ne contient pas $\tilde{D}'$; il sera nécessaire d'introduire en [6.2.3] un schéma de calculs BOR, défini récursivement et qui traduira cette élaboration collatérale.
On peut s'étonner de cette démarche alors qu'il est aussi simple de définir directement $\tilde{D}$ de manière récursive en y incluant les sous déclareurs. Il suffirait pour cela de très peu transformer les systèmes $\mathcal{M}(D)$ de a) à f) ci-dessus ; par exemple dans le cas d'un déclareur de valeur multiple on poserait :

$$\mathcal{M}(D)\begin{cases}\tilde{D} = \mathfrak{m}(\underset{1\leqslant i\leqslant n}{\mathfrak{m}} (\tilde{A}_i, \tilde{B}_i), \tilde{D}')\\ \mathcal{M}(A_i) \quad i=1,\dots,n\\ \mathcal{M}(B_i) \quad i=1,\dots,n\\ \mathcal{M}(D')\end{cases}$$

La raison du choix fait dans la formalisation précèdente tient à la définition de l'élaboration d'un déclareur effectif [A-7.1.2 d)] qui est utilisée dans la définition de l'élaboration d'un générateur [6.3.7] .

Exemple 2 :

Considérons le générateur G : $\textit{loc}\ \overbrace{[a*b : 4]\ \textit{struct}\ (\underbrace{[1 : 3*(a+b)]\ \textit{ent}x_1\ ,\ \textit{reel}x_2}_{D'})}^{D}$

l'élaboration de G commence par l'élaboration collatérale des bornices constituants de D [A-7.1.2 d) pas 5] et non pas de tous les bornices contenus dans D ; c'est-à-dire que G est de la forme :

$$m(\widetilde{a * b}\ ,\ \mathcal{A})\ .\ \mathcal{C}\ .\ \mathcal{D}$$

(où $\mathcal{C}$ est un schéma de calcul formalisant d'autres actions que l'élaboration des bornices dans l'élaboration de G)

et non pas de la forme

$$m(\widetilde{a * b}\ ,\ \mathcal{A}\ ,\ \mathcal{D})$$

ce dont on ne pourrait rendre compte si on définissait $\mathcal{D}$ de manière récursive comme cela est envisagé dans l'exemple 1.

C'est cette différence entre la définition de l'élaboration d'une déclaration d'identité et d'un générateur qui nous conduit à la formalisation de a) à f) ci-dessus ; nous introduirons un schéma de calculs BOR (resp. GEN) pour traiter les déclarations d'identité (resp. les générateurs).

Il est clair que le fait de négliger cette différence conduirait à une formalisation plus simple est plus agréable.

6.2.2.2 Système associé à une déclaration de mode :

Une déclaration de mode M est de la forme *mode* d = D où D est un déclareur de mode [A-7.2.1] et d l'indicateur de mode que M définit. Nous rendrons compte de l'ensemble de calculs associés à l'élaboration éventuelle de bornices contenus dans D par la définition du système $\mathcal{M}(M)$. Remarquons cependant que ce schéma de calculs n'a pas de sens en lui même, il n'apparaîtra dans un schéma de calculs associé à un programme P que si l'indicateur d est "utilisé" dans P (lors d'une déclaration d'identité par exemple).

Le rôle de cette déclaration est d'associer d et D , ainsi :

$$\mathcal{S}(M)\begin{cases}\mathcal{M}(M)\begin{cases}\tilde{M} = \Lambda\\ \mathcal{M}(D)\end{cases}\\ \mathcal{V}(M)\ \{\ d \equiv D\end{cases}$$

Remarques :

1) $\mathcal{V}(M)$ engendre le théorème *indique* $d \equiv$ *indique* D qui permet de définir le mode spécifié par d . En particulier un mode récursif est caractérisé par un ensemble fini de tels théorèmes dont l'interprétation est un système à point fixe.

Le problème de l'égalité des modes est traité en [6.3.4] .

2) de $\mathcal{V}(M)$ on déduit $\tilde{d} = \tilde{D}$ avec [6.1.2] .

3) On ne s'intéresse pas ici aux problèmes de domaine de validité des indicateurs de modes. On suppose dans le langage pivot que :

- tout indicateur de mode est défini par au plus une déclaration de mode
- les vérifications syntaxiques relatives aux domaines de validité ont été faites.

4) $\mathcal{V}(M)$ n'a de sens que si d n'est pas simplement une suite alphanumérique mais tout un arbre syntaxique [5.1] dont l'unique feuille est la suite de caractères représentant l'indicateur.

Enfin pour être complet il reste à définir le double système associé à un indicateur de mode d, comme pour les déclareurs primitifs :

$$\mathcal{S}(d)\begin{cases}\mathcal{M}(d\ ) = \emptyset\\ \mathcal{V}(d\ ) = \emptyset\end{cases}$$

car d et $\tilde{d}$ sont caractérisés par une déclaration de mode.

6.2.3 Déclaration d'identité :

Une déclaration d'identité Id est de la forme :

Dx = E [A-7.4.1]

où

- Dx est le paramètre formel de la déclaration, il est formé du déclareur formel de mode D et de l'identificateur x

- E est le paramètre effectif ; c'est une proposition unitaire (ce ne peut être un transformat car nous ne traitons pas les formats dans cette formalisation).

L'élaboration d'une telle déclaration est définie en [A-7.4.2] ; elle est composée :

a) De l'élaboration de E :

E est une proposition unitaire, on définit en [6.3] le double système $\mathcal{S}(E)$ qui lui est associé et qui traduit son élaboration

b) Du développement de D :

Il consiste essentiellement [A-7.1.2] à remplacer tout indicateur de mode "contenu mais non couvert" dans D par le déclareur effectif de la déclaration de mode correspondant.

Ceci est traduit simplement dans notre formalisation par les axiomes définissant les indicateurs et déclareurs de modes [6.2.2] et les axiomes logiques.

c) De l'élaboration des bornices directs de D :

(Nous appellerons bornice direct de D tout bornice contenu dans D mais non contenu dans un bornice lui-même contenu dans D [A-7.4.2 pas 2]).

Cette déclaration est traduite par le schéma de calculs BOR [6.2.2.1 g)] .

d) De la "définition"du schéma fonctionnel *poss x* [A-7.4.2 pas 7] .

e) D'un certain nombre de "tests de cohérence" [A-7.4.2 pas 3,5,6] :

Une partie d'entre eux [pas 5 et 6] consiste en la comparaison entre un état (resp. une borne) d'une sous valeur de *poss* E et l'état (resp. la borne) correspondant (resp. correspondante) dans le déclareur D .

La démarche formalisant la sémantique de la déclaration d'identité Id : D*x* = E sera alors la suivante :

- la définition de *poss x* ainsi que le test [A-7.4.2 pas 3] sur les composants d'une valeur multiple flexible (qui n'est pas récursif) seront exprimés par un schéma de modifications élémentaires ident [6.2.2.3]

- l'élaboration des bornices D est traduite par le schéma de calculs BOR

- les autres tests de cohérence [A-7.4.2 pas 5 et 6] nécessitent une définition récrusive, ceci pour rendre compte de la notion de correspondant (e) ci-dessus). Ils seront traduits par le schéma de calculs EXIST [6.2.3.5] .

- tout d'abord définissons les systèmes associés à Id :

6.2.3.2 Systèmes associés à une déclaration d'identité :

Soit Id : D*x* = E une déclaration d'identité. Le double système qui lui est associé est défini par :

$$\mathcal{S}(\mathrm{Id}) \begin{cases} \mathcal{M}(\mathrm{Id}) \begin{cases} \widetilde{\mathrm{Id}} = m(\widetilde{E}, \mathrm{Bor(D)}) \ .\ \mathrm{EXIST(D,D,}\ \textit{poss}\ \mathrm{E)}\ .\ \mathrm{ident}(x, \textit{poss}\ \mathrm{E}) \\ \mathcal{M}(\mathrm{E}) \\ \mathcal{M}(\mathrm{EXIST}) \\ \mathcal{M}(\mathrm{BOR}) \\ \mathcal{M}(\mathrm{D}) \end{cases} \\ \mathcal{V}(\mathrm{Id}) = \emptyset \end{cases}$$

BOR(D) est l'ensemble de calculs formalisant l'élaboration des bornices directs de D.

Remarquons que c'est la nécessité d'exprimer que cette élaboration est collatérale avec celle de E qui conduit à introduire le schéma BOR ; si ce n'était pas

le cas, l'élaboration pourrait très bien être traduite dans le schéma de calculs EXIST (voir [6.2.2.1 g)]).

Définissons maintenant les différents schémas de modifications élémentaires et de calculs qui apparaissent dans $\mathcal{S}$(Id).

6.2.3.3 Définition du schéma ident :

ident est un schéma de modifications à deux paramètres x et v ; il ne modifie que les seuls symboles fonctionnels *poss* et *valeur*

$\sigma_{\mathrm{ident}(x,v)}$ (*poss*) = *poss'* ; $\sigma_{\mathrm{ident}(x,v)}$ (*valeur*) = *valeur'* (et $\sigma_{\mathrm{ident}(x,v)}$ est l'identité sur les autres symboles fonctionnels).

ident est défini par l'ensemble d'axiomes :

$$\begin{aligned} \chi_{\mathrm{ident}(x,v)} = {} & \{\textit{valeur' poss' } x \equiv \textit{valeur } v\} \cup \{\textit{valeur } u \equiv \textit{valeur } v \supset \\ & \neg u \equiv \textit{poss' } x \mid u \in \mathrm{EXVAL} - \{\textit{poss' } x\}\} \cup \\ & \{\textit{poss' } y \equiv \textit{poss } y \mid y \in \mathrm{IDENT} - \{x\}\} \cup \\ & \{\textit{valeur' } u \equiv \textit{valeur } u \mid u \in \mathrm{EXVAL} - \{\textit{poss' } x\}\} \cup \\ & \{\textit{sousflex rep } v \equiv \textit{faux}\} \end{aligned}$$

ce qui traduit [A-7.4.2 pas 7] et [A-7.4.2 pas 3] :

i) on fait "posséder" à x un nouvel exemplaire de v ; ce qui est plus exigeant que [A-7.4.2 pas 7] mais rendu nécessaire dans notre formalisation car nous n'exprimons pas qu'à chaque occurence d'une notation ou d'un identificateur est associé à un nouvel exemplaire de valeur [6.3.9] . Une remarque analogue sera faite pour les affectations.

ii) le dernier axiome exprime que v ne peut repérer un composant d'une valeur multiple flexible sous peine d'obtenir l'information inconsistante.

6.2.3.4 Système de calculs associé au schéma BOR :

BOR est un schéma de calculs à un paramètre D qui est un déclareur de mode. Il exprime l'élaboration collatérale des bornices directs de D [A-7.4.2 pas 2] ; nous le définirons de manière récursive, la récursivité portant sur D.

On veut que BOR, suivant la valeur du paramètre D , possède les propriétés suivantes :

a) Si D est un déclareur de valeur simple ou si D commence par *rep rep* , par *proc*, par *union*, par *rep proc* ou par *rep union* alors il ne contient aucun bornice

(c'est clair dans le cas d'une valeur simple et dans les autres cas les déclareurs apparaissant dans D sont virtuels [A-7.1.1]).

Alors BOR(D) doit être l'élément neutre $\mathcal{N}$ de $\mathcal{M}od^{\infty}$ (c'est-à-dire qu'il sera interprétable comme la modification identité).

b) Si D est de la forme *rep* D' où D' commence par *struct* ou [l'élaboration des bornices de D est celle des bornices de D' :

$$BOR(D) = BOR(D') .$$

c) Si D est de la forme *struct* $(D_1, \ldots, D_n)$ les bornices de D sont ceux de D_1, D_2, ..., D_n et donc, en exprimant la collatéralité de l'élaboration :

$$BOR(D) = \underset{1\leq i\leq n}{\mathfrak{m}}^{n} (BOR(D_i)) .$$

d) Si D est de la forme $[A_1\Gamma_1 : B_1\Delta_1, \ldots, A_n\Gamma_n : B_n\Delta_n]$ D' , l'élaboration des bornices directs de D est celle des A_i , B_i ainsi que celle des bornices directs de D' . Ainsi, puisque $\overset{\sim}{\mathcal{D}}$ est le schéma de calcul qui formalise l'élaboration collatérale des bornices de D [6.2.2.1] :

$$BOR(D) = \mathfrak{m}(\overset{\sim}{\mathcal{D}}, BOR(D')) .$$

e) Il faut définir également BOR(*d*) pour tout indicateur de mode *d*, et ceci à l'aide de BOR(D) si d est caractérisé par la déclaration de mode : *mode* *d* = D .

On est donc conduit à définir le schéma de calculs BOR par le système à point fixe :

$$\mathcal{M}(\mathrm{BOR})\begin{cases} \mathrm{BOR} = \lambda D\left[\bigcup_{\mu\in \mathrm{AUTRE}\cup\mathrm{UNION}\cup\mathrm{RUAUTRE}} j(\textit{indique } D,\mu)\right] \cup \\ \quad \lambda D\left[\bigcup_{\substack{D'\in \mathrm{INDIC}\\ \mu\in \mathrm{STMUL}}} j(\textit{indique } D, \textit{indique } \underline{\textit{rep}}\ D').j(\textit{indique } D',\mu).\mathrm{BOR}(D')\right] \cup \\ \quad \lambda D\left[\bigcup_{\substack{D_1,\ldots,D_n\in \mathrm{INDIC}\\ x_1,\ldots,x_n\in L_o^{id}}} j(\textit{indique } D, \textit{indique } \underline{\textit{struct}}(D_1x_1,\ldots,D_nx_n).\underset{1\leq i\leq n}{\mathfrak{m}}(\mathrm{BOR}(D_i))\right] \cup \\ \quad \lambda D\left[\bigcup_{\substack{n\geq 0\\ D'\in \mathrm{INDIC}}} j(\textit{indique } D, \textit{rang}^{(n)}\textit{indique } D').\mathfrak{m}(\tilde{D}, \mathrm{BOR}(D'))\right] \end{cases}$$

Remarques :

1) chaque ligne du deuxième membre de l'équation $\mathcal{M}$(BOR) correspond à l'un des cas envisagé ci-dessus (a), b), c) ou d)).

2) Si D est l'une des formes considérée en a), l'un des calculs défini par BOR(D) est bien l'élément neutre de $\mathcal{M}od^{\infty}$.

6.2.3.5 Système de calculs associé au schéma EXIST :

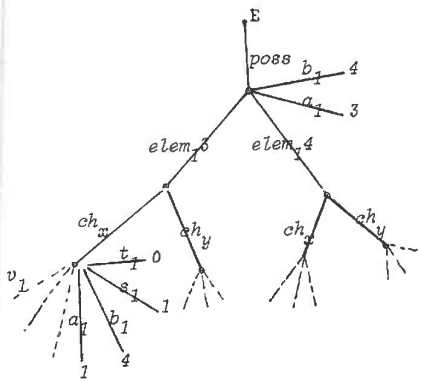
EXIST est un schéma de calculs à trois paramètres D, D_1, v_1 qui permet de traduire [A-7.4.2 pas 4,5,6] .

- D'après le pas 4, les tests de cohérence n'ont lieu que si *poss* E (dans la déclaration d'identité D*x* = E) est une valeur de mode:structure, valeur multiple, repère de structure ou repère de valeur multiple ; de plus une distinction est faite selon que ce mode commence par repère ou non. Le rôle du premier paramètre D est de traduire ce pas 4 : D est un déclareur formel et sa connaissance équivaut à celle du mode de *poss* E .

- Les tests exprimés en [A-7.4.2 pas 5 et 6] font intervenir une correspondance entre le déclareur D et le mode de *poss* E . Pour définir correctement cette correspondance, il faut "explorer parallèlement" D et le mode de *poss* E en comparant les sous déclareurs D_1 de D aux modes des sous valeurs (correspondantes) v_1 de v . C'est la raison pour laquelle EXIST est défini de manière récursive, la récursivité portant sur les déclareurs constituants de D , c'est le rôle du deuxième paramètre D_1 :

dans $\mathcal{M}$(Id) on introduit le schéma EXIST (D,D, *poss* E) qui est défini de manière récursive à partir des EXIST (D,D_1,v_1) où D_1 est un sous déclareur de D, v_1 la sous valeur de *poss* E correspondant à D_1 .

Exemple : soit D le déclareur [*3.1:4.1*] *struct*([*1.1:4.0*] *ent* *x*, *réel* *y*) et *poss* E l'exemplaire de valeur schématisé par :



Il s'agit alors

1) de comparer . *3* et a_1 *poss* E

. *4* et b_1 *poss* E

2) puis

. *1* et a_1 ch_x $elem_1$ *poss* E *3*

Ainsi EXIST(D,D, *poss* E) sera composé des calculs formalisant 1) et des calculs EXIST(D,D_1,v_1) formalisant 2) (avec D_1 = [*1.1:4.0*] *ent* et $v_1 = ch_x\ elem_1\ poss\ E\ 3$).

Dans une première étape définissons informellement l'ensemble de calculs EXIST(D,D_1,v_1) en fonction de D, D_1, v_1 :

A) D n'est ni un déclareur de structure, ni un déclareur de valeur multiple, ni un déclareur de nom de structure ou de valeur multiple :
Alors [A-7.4.2 pas 4] :

EXIST(D,D_1,v_1) = Λ quel que soit D_1 .

B) D est un déclareur de valeur structurée, multiple ou de nom de valeur structurée ou multiple :

EXIST(D,D_1,v_1) est alors défini récursivement par rapport à D_1 .

a) D_1 est un déclareur de valeur simple, où il commence par *proc*, *union*, *rep rep*, *rep proc* ou *rep union* :

D_1 ne contient aucun bornice et donc aucun test n'est à faire :

EXIST (D,D_1,v_1) = Λ

b) D_1 est un déclareur de valeur structurée ou de nom de valeur structurée :

D_1 = *rep*$^{(\varepsilon)}$ *struct* ($D'_1\ x_1$,..., $D'_n\ x_n$) (*rep* est absent si ε = 0 , présent si ε = 1)

EXIST (D,D_1,v_1) = $\underset{1\leqslant i\leqslant n}{\mathfrak{m}}$ (EXIST (D,D'_i,$chx_i\ v_1$))

(les bornes et états correspondants à ceux de $chx_i\ v_1$ sont spécifiés dans D'_i)

c) D_1 est un déclareur de valeur multiple ou de nom de valeur multiple :

D_1 = *rep*$^{(\varepsilon)}$ [A_1,Γ_1 : B_1,Δ_1 ,..., A_n,Γ_n : B_n,Δ_n] D'_1 .

Les tests de cohérence sont porter d'une part sur les A_i , B_i , Γ_i , Δ_i (que l'on compare aux bornes et états correspondants dans V_1) et d'autre part sur D'_1 , ainsi :

EXIST(D,D_1,v_1) = test(D,D_1,v_1). $\bigcup_{I_1,..,I_n\in\text{NOTENT}\ J_1,..,J_n\in\text{NOTENT}}$ j(*valeur poss* A_1,I_1)...j(*valeur poss* A_n,I_n). j(*valeur poss* B_1,J_1)...j(*valeur poss* B_n,J_n . $\underset{I_j\leqslant i_j\leqslant J_j}{\mathfrak{m}}$ (EXIST(D,D'_1,$elem_n\ v_1\ i_1..i_n$))

pour $1\leqslant j\leqslant n$

- test est un schéma de modifications élémentaires défini en [6.2.3.6] qui exprime [A-7.4.2 pas 5 et 6]

- *poss* A_i (resp. *poss* B_i) ($1 \leqslant i \leqslant n$) est déterminé par *poss* a_i D_1 (resp. *poss* b_i D_1) grâce aux axiomes de $\mathfrak{V}(D_1)$ [6.2.2.1]

- D'_1 peut contenir des sous déclareurs sur lesquels des vérifications doivent être faites. C'est le rôle de la dernière partie du schéma EXIST(D,D_1,v_1) .

En regroupant les considérations précèdentes, on obtient le système de calculs $\mathcal{M}$(EXIST) :

$\mathcal{M}$(EXIST) =

EXIST = $\lambda D\lambda D_1\lambda v_1 \left[\bigcup_{\mu\in\text{AUTRE}\cup\text{RUAUTRE}\cup\text{UNION}} j(\textit{indique}\ D,\mu)\right] \cup$
$\lambda D\lambda D_1\lambda v_1 \left[\bigcup_{\mu\in\text{STMUL}\cup\text{RSTMUL}} j(\textit{indique}\ D,\mu)\right]$.

$\left[\bigcup_{\mu_1\in\text{AUTRE}\cup\text{RUAUTRE}\cup\text{UNION}} j(\textit{indique}\ D_1,\mu_1)\right] \cup$
$\bigcup_{D'_1,...,D'_n\in\text{INDIC}\ x_1,...,x_n\in L_o^{id}\ \varepsilon=0,1} j(\textit{indique}\ D_1, \textit{repère}^{(\varepsilon)}_{x_1...x_n}\ \textit{structure indique}D'_1...\textit{indique}D'_n)$.

$\underset{1\leqslant i\leqslant n}{\mathfrak{m}}$ (EXIST(D,D'_i,$chx_i\ v_1$)) $\cup$
$\bigcup_{n\in\mathbb{N}\ D'_1\in\text{INDIC}\ \text{et}\ \varepsilon=0,1} j(\textit{indique}\ D_1, \textit{repère}^{(\varepsilon)}_{rang^{(n)}}\ \textit{indique}\ D'_1)$.

test(D,D_1,v_1) . $\left[\bigcup_{I_1,..,I_n\in\text{NOTENT}\ J_1,..,J_n\in\text{NOTENT}} (j(\textit{valeur poss}\ a_1\ B_1,J_1)\ ....\right.$

$j(\textit{valeur poss}\ a_n\ D_1, I_n)$. $j(\textit{valeur poss}\ b_1\ D_1, J_1)$ ....
$j(\textit{valeur poss}\ b_n\ D_1, J_u)$.

$\left.\left.\underset{I_j\leqslant i_j\leqslant I_j}{\mathfrak{m}} (\text{EXIST}(D,D'_1,elem_n\ v_1\ i_1...i_n))\right]\right]$

pour $1\leqslant j\leqslant n$

6.2.3.6 Définition du schéma de modifications élémentaires test :

Le schéma test dépend de trois paramètres : D, D_1 et v_1 en reprenant de [6.2.3.5] . Il n'apparaît dans un calcul que si D est un déclareur de structure, de valeur multiple, de nom de structure ou de valeur multiple et si D_1 est de la forme $\underline{rep}^{(\varepsilon)}\ [A_1\Gamma_1 : B_1\Delta_1\ ,\ldots,\ A_n\Gamma_n : B_n\Delta_n]\ D'_1$; nous ne le définirons que dans ce cas.

test (D,D_1,v_1) ne modifie aucun symbole fonctionnel, il ajoute simplement les axiomes suivants :

$$\chi_{test(D,D_1,v_1)} = \{dim\ v_1 \equiv n \supset (\bigwedge_{1\leqslant i\leqslant n} (s_i\ D_1 \equiv 0 \vee a_i\ D_1 \equiv vide \vee valeur\ poss\ a_i\ D_1 \equiv a_i\ v_1) \wedge (t_i\ D_1 \equiv 0 \vee b_i\ D_1 \equiv vide \vee valeur\ poss\ b_i\ D_1 \equiv b_i\ v_1)) \mid n \in \text{NOTENT}\} \cup$$

$$\{dim\ v_1 \equiv n \wedge D \equiv \underline{rep}\ D' \supset (\bigwedge_{1\leqslant i\leqslant n} (s_i\ D_1 \equiv 2 \vee s_i\ D_1 \equiv s_i\ v_1) \wedge (t_i\ D_1 \equiv 2 \vee t_i\ D_1 \equiv t_i\ v_1)) \mid n \in \text{NOTENT}\ ,\ D' \in \text{INDIC}\} \ .$$

Le premier ensemble d'axiomes exprime [A-7.4.2 pas 6] : il "compare" les bornes correspondantes de D_1 et v_1 .

Le deuxième ensemble d'axiomes exprime [A-7.4.2 pas 5] : il "compare" les états correspondants de D_1 et v_1 .

6.2.4 Déclaration d'opération :

Une déclaration d'opération Op est de la forme $\underline{op}(D_1,D_2)\ D_3\ x = E$ ou $\underline{op}(D_1)\ D_2\ x = E$.

D_1, D_2 et D_3 sont des déclareurs virtuels ; x est un opérateur (élément de L_o^{op}) et E une proposition unitaire du "genre opera" [A-7.5.1] .

La valeur possédée par E est une routine [6.3.10] .

L'élaboration de E est exprimée dans $\mathcal{M}(E)$, [A-7.5.2] qui définit l'élaboration de Op se formalise par :

$$\mathcal{S}(Op)\begin{cases}\mathcal{M}(Op) \begin{cases}\overset{\vee}{Op} = \overset{\vee}{E}\ .\ opera(x,poss\ E)\\ \mathcal{M}(E)\end{cases}\\ \mathcal{V}(Op) = \emptyset\end{cases}$$

opera étant le schéma de modifications élémentaires à deux paramètres qui fait posséder *poss* E à x.

opera modifie le seul symbole fonctionnel *poss*

$$\sigma_{opera(x,v)}(poss) = poss' \ .$$

Il est défini par :

$$\chi_{opera(x,v)} = \{poss'\ x \equiv v\} \cup \{poss'\ u \equiv poss\ u \mid u \in \text{IDENT} - \{x\}\} \ .$$

Remarque :

On aurait pu également éliminer opérateurs et formules [6.3.5] du langage pivot en les remplaçant respectivement par des procédures et des appels de procédures.

6.3 Systèmes associés aux propositions unitaires (non parenthésées) :

6.3.1 Introduction :

Une proposition unitaire non parenthésée est [A-8.1.1] :

i) soit une confrontation : - affectation
- relation d'identité
- relation de conformité
- forceur

ii) soit une formule

iii) soit une cohésion : - sélection
- générateur

iv) soit une base : - identificateur
- notation
- tranche
- appel de procédure

A chacune de ces phrases élémentaires sera associé un double système, avec toutefois la restriction suivante :

Nous ne considérerons pas les forceurs qui sont de nature purement syntaxique et traités dans le langage pivot [5.2] .

6.3.2 Affectation :

6.3.2.1 Introduction :

L'élaboration d'une affectation est définie de façon récursive [A-8.3.1.2] ; pour comprendre la formalisation qui va suivre il est utile de faire les remarques suivantes :

i) Les "opérandes" d'une affectation sont respectivement un exemplaire v de valeur (source) et un nom u (destination). En fait dans notre formalisation le symbole *rep* est défini sur des exemplaires et donc u sera un exemplaire de nom. Remarquons que nous rendons bien compte de [A-8.3.1.2] car tous les exemplaires d'un même nom repèrent le même exemplaire (voir $SH_{1.11}$ et la remarque qui suit).

ii) La définition d'une affectation comporte essentiellement deux étapes :

- une première qui consiste en des "tests" et des "ajustements" d'états portant sur des descripteurs de valeurs multiples repérées par u ou des sous noms de u . C'est cette étape qui est définie de manière récursive en [A-8.3.1.2 c)] .

- la deuxième qui ne commence [A-8.3.1.2 c) pas 2] que lorsque tous les "tests" et "ajustements" précédents ont été faits et qui consiste en la modification effective de la fonction repère [A-8.3.2.1 a) et b)] , on dit alors que v (ou les exemplaires de v) supplante (nt) l'exemplaire de valeur (ou les sous exemplaires correspondants) antérieurement repéré (s) par u (ou les sous noms de u).

iii) Enfin certains tests de cohérence [A-8.3.1.2 c) pas 1] sont nécessaires avant toute affectation : ils constituent une étape antérieure à celles de ii).

Remarque : Les vérifications syntaxiques de compatibilité des modes de N et V sont supposées effectuées dans le langage pivot.

La formalisation qui suit s'appuie essentiellement sur ces remarques :

- les "tests" et "ajustements" (ii) étape 1) seront exprimés par l'intermédiaire d'un schéma de calculs AFFECT défini récursivement par un système à point fixe [6.3.2.4]

- la modification du symbole *rep* (ii) étape 2) sera traduite par un schéma de modifications élémentaires suppl (pour supplante) [6.3.2.6] .

- iii) sera exprimé par un schéma de modifications élémentaires coh [6.3.2.3] .

6.3.2.2 Elaboration d'une affectation :

Soit A_f l'affectation Z := E , son élaboration est définie par le double système :

$$(A_f)\begin{cases} \mathcal{M}(A_f)\begin{cases} \tilde{A}_f = m(\tilde{Z},\tilde{E}) \ .\ \text{coh}(poss\ Z,\ poss\ E)\ .\ \text{AFFECT}(poss\ Z,\ poss\ E) \\ \mathcal{M}(Z) \\ \mathcal{M}(E) \\ \mathcal{M}(\text{AFFECT}) \end{cases} \\ \mathcal{V}(A_f)\{\ poss\ A_f \equiv poss\ Z \end{cases}$$

On traduit ainsi [A-8.3.1.2 d)] : l'élaboration de la destination de la source est collatérale par définition de l'opération mélange ; la valeur de la source est affectée à celle de la destination (schémas coh et AFFECT) et la valeur de l'affectation est celle de sa destination.

Définissons tout d'abord le schéma de modifications coh :

6.3.2.3 Définition du schéma de modifications coh :

coh est un schéma à deux paramètres u et v traduisant [A-8.3.1.2 c) pas 1] ; traduction qui se fait comme à l'habitude à l'aide de la notion d'inconsistance [3.3.2.1]

coh(u,v) ne modifie aucun accès, il ajoute simplement les axiomes :

$$X_{coh(u,v)} = \{\neg\, valeur\ u \equiv nil\} \cup \{ppq\ portée\ u\ portée\ v \equiv vrai\} \cup \{sousflex\ rep\ \ v \equiv faux\}$$

6.3.2.4 Définition du schéma de calculs AFFECT :

AFFECT est un schéma de calculs à deux paramètres u et v où u est un exemplaire de nom et v un exemplaire de valeur ; sa définition est récursive, la récursivité portant sur le mode de u :

AFFECT permet de traduire les tests entre les descripteurs de v (ou de ses sous valeurs) et de la valeur antérieurement repérée par u (ou de ses sous noms).

Dans un premier temps décrivons AFFECT de manière informelle :

a) Si le mode de u ne commence pas par "repère de structure" ou "repère de rang de" il s'agit d'exprimer [A-8.3.1.2 c) pas 2] .

Alors : AFFECT(u,v) = suppl(u,v) .

b) Si le mode de u commence par "repère de structure" alors il faut traduire [A-8.3.1.2 c) pas 5] .

Ainsi, si le mode de u est "*repère structure* $\mu_1 \ldots \mu_n$" :
$x_1 \ldots x_n$

$$AFFECT(u,v) = \mathop{\mathfrak{m}}_{1 \leqslant i \leqslant n} (AFFECT(ch_{x_i}\ u,\ ch_{x_i}\ v)) .$$

c) Si le mode de u commence par "repère de rang" il s'agit d'exprimer [A-8.3.1.2 c) pas 3), 4), 5)] :

$$AFFECT(u,v) = traitmul(u,v).\ \bigcup_{\substack{I_1,\ldots,I_n \in NOTENT \\ J_1,\ldots,J_n \in NOTENT}} [j(a_1 v,J_1)\ldots j(a_n v,I_n).j(b_1 v,J_1)\ldots j(b_n v,J_n).\ \mathop{\mathfrak{m}}_{\substack{I_k \leqslant j_k \leqslant J_k \\ \text{pour } 1 \leqslant k \leqslant n}} (AFFECT(elem_n u\ i_1 \ldots i_n, elem_n v\ i_1 \ldots i_n))]$$

traitmul est un schéma de modifications élémentaires qui permet d'exprimer "tests" et "ajustements" du descripteur de v [6.3.2.5].

d) En regroupant les considérations précèdentes, on obtient la définition suivant du schéma AFFECT :

$$\mathcal{M}(AFFECT) \begin{cases} AFFECT = \lambda u\ \lambda v \bigcup_{\mu \in RUAUTRE} j(mode\ u,\mu)\ .\ suppl(u,v) \cup \\ \lambda u\ \lambda v \bigcup_{\substack{x_1,\ldots,x_n \in L_o^{id} \\ \mu_1,\ldots,\mu_n \in MODE \\ \text{et } n \geqslant 1}} [j(mode\ u, repère\ structure\ \mu_1 \ldots \mu_n)_{x_1 \ldots x_n}\ . \mathop{\mathfrak{m}}_{1 \leqslant i \leqslant n} (AFFECT(ch_{x_i}\ u,\ ch_{x_i}\ v))] \\ \lambda u\ \lambda v \bigcup_{\substack{n \geqslant 1 \\ \mu \in MODE\text{-}rang\ MODE}} [j(mode\ u,\ repère\ rang^{(n)}\mu)\ . \\ traitmul(u,v)\ .\ \bigcup_{\substack{I_1,\ldots,I_n \in NOTENT \\ J_1,\ldots,J_n \in NOTENT}} (j(a_1 v,I_1)\ldots j(a_n v,I_n).j(b_1 v,J_1)\ldots j(b_n v,I_n)\ . \mathop{\mathfrak{m}}_{\substack{I_k \leqslant j_k \leqslant J_k \\ \text{pour } 1 \leqslant k \leqslant n}} (AFFECT(elem_n u\ i_1 \ldots i_n, elem_n v\ i_1 \ldots i_n)]. \end{cases}$$

6.3.2.5 Définition du schéma de modifications élémentaires traitmul :

traitmul est un schéma à deux paramètres u et v (u est un exemplaire de nom, v un exemplaire de valeur), il exprime [A-8.3.1.2 c)] pas 3 et 4 .

a) Si dans une information I le $i^{ème}$ état inférieur (resp. supérieur) de *rep* u est 1 et si les bornes correspondantes de rep u et v ne sont pas égales alors traitmul(u,v)(I) est l'information inconsistante.

Ainsi $X_{traitmul(u,v)}$ contient les axiomes :

$$\{dim\ v \equiv n \supset (\bigwedge_{1 \leqslant i \leqslant n} \neg(s_i\ rep\ u \equiv 1 \wedge \neg ai\ rep\ u \equiv a_i\ v)) \wedge (\bigwedge_{1 \leqslant i \leqslant n} \neg(t_i\ rep\ u \equiv 1 \wedge \neg b_i\ rep\ u \equiv b_i\ v)) \mid n \geqslant 1\ ,\ n \in NOTENT\}$$

b) Si l'une au moins des bornes de *rep* u est flexible, il s'agit :

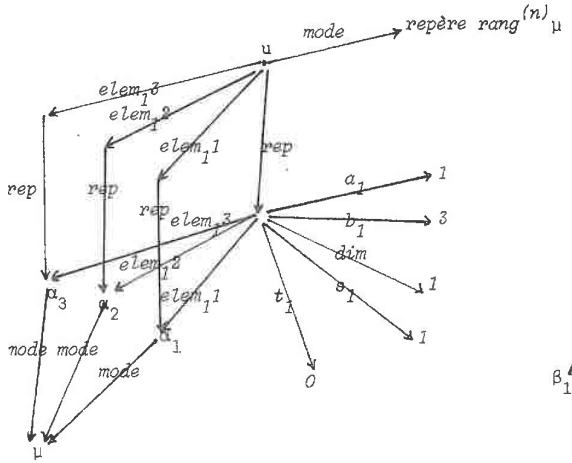
b1) de "transformer" le descripteur de v en mettant ses états aux états correspondants de l'exemplaire de valeur précédemment repéré par u ; dans notre formalisation cela revient à modifier certains s_i ou t_i , c'est le rôle de l'ensemble d'axiomes :

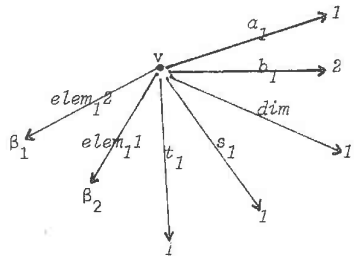
$$\{dim\ v \equiv n \wedge \bigvee_{1 \leqslant i \leqslant n} (s_i\ rep\ u \equiv 0 \vee t_i\ rep\ u \equiv 0)) \supset \bigwedge_{1 \leqslant i \leqslant n} (s'_i\ v \equiv s_i\ rep\ u \wedge t'_i\ v \equiv t_i\ rep\ u) \mid n \in NOTENT\ ,\ n \geqslant 1\} \ .$$

b2) De "créer" un nouvel exemplaire w d'une certaine valeur multiple et de le faire repérer par u . Dans notre formalisation nous ne "créerons" pas un tel exemplaire, nous contentant de "créer", éventuellement, de nouveaux sous noms de u permettant de repérer les éléments correspondants de v . Cette création est ici automatique, elle résulte des calculs AFFECT ($elem_n$ u $i_1 \ldots i_n$, $elem_n$ v $i_1 \ldots i_n$) (les nouveaux sous noms éventuels étant de la forme $elem_n$ u $i_1 \ldots i_n$) et des schémas d'axiomes $SH_{1.2}$, $SH_{1.6}$, $SH_{1.7}$ qui caractérisent les sous noms d'un nom de valeur multiple, leur portée et leur mode.

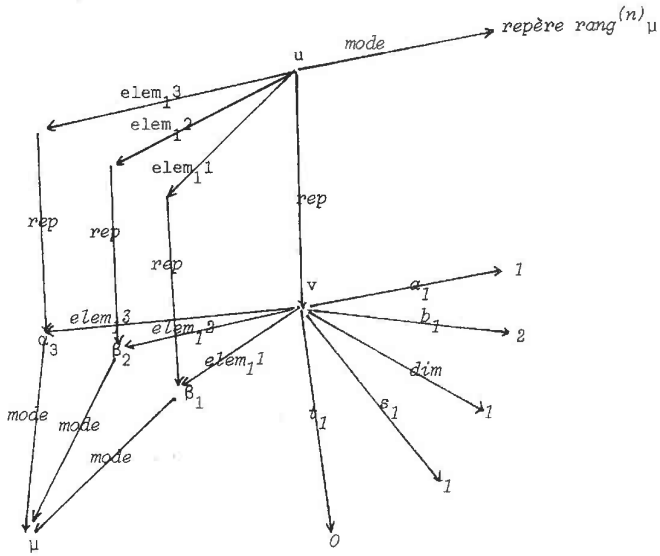
Un problème semble subsister dans le cas où il s'agit de "supprimer" des sous noms de u . En fait nous ne les supprimerons pas mais les valeurs qu'ils repèrent ne seront plus "accessibles" dans une information consistante car les "conditions" inf a_j rep v i_j ≡ vrai ∧ inf i_j b_j rep v ≡ vrai ne sont pas vérifiées [6.3.8].

Exemple : De manière schématique représentons une partie des théorèmes caractérisant u et v de la manière suivante :





Après affectation, les théorèmes correspondant aux théorèmes précédents peuvent être schématisés par :



en particulier t_1 v ≡ *1* est transformé en t_1 v ≡ *0* (b1)).

c) En résumé traitmul(u,v) modifie les seuls symboles fonctionnels s_i et t_i ($i \geqslant 1$) et est définie par :

$$\chi_{traitmul(u,v)} = \{dim\ v \equiv n \supset (\bigwedge_{1 \leqslant i \leqslant n} \neg(s_i\ rep\ u \equiv 1 \wedge a_i\ rep\ u \equiv a_i\ v)) \wedge$$

$$(\bigwedge_{1 \leqslant i \leqslant n} \neg(t_i\ rep\ u \equiv 1 \wedge \neg b_i\ rep\ u \equiv b_i\ v)) \mid n \geqslant 0\ ,$$

$$n \in \text{NOTENT}\} \cup$$

$$\{dim\ v \equiv n \wedge (\bigvee_{1 \leqslant i \leqslant n} (s_i\ rep\ u \equiv 0 \vee t_i\ rep\ u \equiv 0)) \supset$$

$$\bigwedge_{1 \leqslant i \leqslant n} (s'_i\ v \equiv s_i\ rep\ u \wedge t_i\ v \equiv t_i\ rep\ u) \mid n \geqslant 1\ ,$$

$$n \in \text{NOTENT}\} \cup$$

$$\{s'_i\ w \equiv s_i\ w \wedge t'_i\ w \equiv t_i\ w \mid w \in \text{EXMUL} - \{v\}\}\ .$$

6.3.2.6 Définition du schéma de modifications élémentaires suppl :

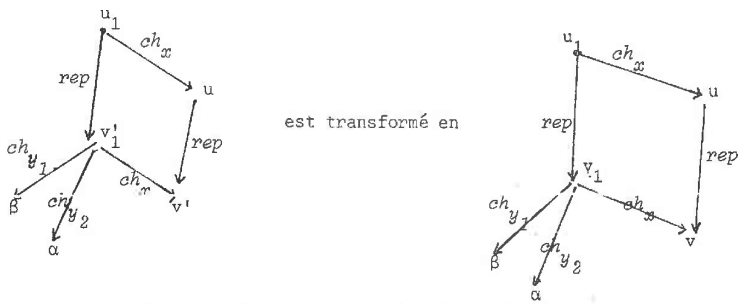
Ce schéma permet d'exprimer [A-8.3.1.2 a) et b)] dans notre formalisation. Le but est de "faire repérer" un exemplaire v de valeur par un exemplaire de nom u , c'est-à-dire ici de modifier le symbole fonctionnel *rep* . Il faut de plus modifier ce symbole en tout nom w qui repère un exemplaire d'une valeur structurée ou multiple dont *rep* u est un composant.

A priori il semble donc nécessaire se "créer" de nouveaux exemplaires de nouvelles valeurs dont l'un des champs ou élément sera v . Si on adoptait ce point de vue, il faudrait certainement définir suppl de manière récursive ; ce devrait donc être un schéma de calculs. Remarquons alors la lourdeur de cette démarche :

- le schéma de calculs AFFECT est défini par une récursivité "descendante" : on définit l'affectation d'une structure (par exemple) à un nom de structure, par les affectations des champs de cette structure.

- le "schéma de calculs suppl" (dans l'optique précèdente) devrait être défini par une récursivité "ascendante". Donc, dans le cas d'une structure par exemple, on serait amené à "parcourir" l'arborescence associée de 2 manières inverses l'une de l'autre.

La remarque suivante permet une définition beaucoup plus simple :
Si u_1 est un exemplaire de nom qui repère un exemplaire d'une valeur structurée ou multiple donc *rep* u est un composant v' , une manière de "créer" le nouvel exemplaire de la nouvelle valeur que l'on fera repérer par u_1 peut consister à "détruire" l'exemplaire v'_1 repéré par u_1 en le transformant en un exemplaire v_1 dont tous les composants sont les mêmes que ceux de v'_1 excepté v' qui devient v . Schématisons cette "création" de la manière suivante :



est transformé en

Pour effectuer cette transformation il suffit de modifier *rep* en u ; les schémas d'axiomes $SH_{1.1}$ et $SH_{1.2}$ entraîne la modification de ch_x ou $elem_n$ en *rep* u_1 et donc la création de nouveaux exemplaires de nouvelles valeurs en remplacement

de tout exemplaire dont v' est un composant. Enfin il est clair que tout autre exemplaire n'est pas modifié.

Remarque 1 : Cette transformation n'est possible que si les exemplaires "détruits" ne sont pas "utilisés" ailleurs. Intuitivement cela signifie qu'il ne doit pas exister d'exemplaire de nom u_2 tel que u_2 et u soient des exemplaires de deux noms différents et qui repèrent v' . Autrement dit il ne doit pas exister d'exemplaires de noms u_2 et u repèrant le même exemplaire de valeurs et qui soient des exemplaires de 2 noms différents, ce qui résulte de $SH_{1.12}$.
(L'un des intérêts de la notion d'exemplaire réside donc dans cette définition simple de [A-8.3.1.2 a)] .

En résumé on peut définir suppl(u,v) par :

suppl(u,v) modifie *rep*, ch_x $(x \in L_o^{id})$, $elem_n$ $(n \geqslant 1)$, *valeur* et laisse invariants les autres symboles fonctionnels de L :

$\sigma_{suppl(u,v)}(rep) = rep'$; $\sigma_{suppl(u,v)}(ch_x) = ch_x'$;

$\sigma_{suppl(u,v)}(elem_n) = elem'_n$; $\sigma_{suppl(u,v)}(valeur) = valeur'$.

Il est caractérisé par :

$$\chi_{suppl(u,v)} = \{valeur'\ rep'\ u \equiv valeur\ v\} \cup$$
$$\{valeur\ w \equiv valeur\ v \supset \neg rep'\ u \equiv w \mid w \in EXVAL - \{rep'\ u\}\} \cup$$
$$\{rep'\ w \equiv rep\ w \mid w \in EXVAL - \{u\}\} \cup$$
$$\{u \equiv elem_n\ u_1\ i_1 \ldots i_n \supset elem'_k\ wi_1 \ldots i_n \equiv elem_k\ wi_1 \ldots i_n \mid$$
$$elem_n\ u_1\ i_1 \ldots i_n \in EXVAL\ ;\ elem_k\ wi_1 \ldots i_n \in EXVAL \text{ et } k \neq n \text{ ou}$$
$$k = n \text{ et } w \text{ ne contenant aucune occurence de } u_1\}$$
$$\{u \equiv ch_x\ u_1 \supset ch'_y\ w \equiv ch_y\ w \mid ch_x\ u_1 \in EXVAL\ ;\ ch_y\ w \in EXVAL \text{ et}$$
$$y \neq x \text{ ou } y = x \text{ et } w \text{ ne contenant aucune occurence de } u_1\} .$$

Remarque 2 : [A-8.3.1.2 b)] est bien traduit ici car si w a pour sous nom u, *rep* w n'est pas modifié et donc *mode rep* w non plus.

Remarque 3 : la lourdeur des deux premiers ensembles d'axiomes est dûe au fait que nous n'exprimons pas qu'à chaque occurence d'un identificateur (d'une notation,...) est associé un nouvel exemplaire de valeur [6.2.3.3 i)].

Remarque 4 : Il se peut que le mode de u ne soit pas un mode simple mais commence, par exemple, par "*repère repère*", "$union_n$" etc ... [6.3.2.4 a)] .

Les schémas d'axiomes $SH_{1.1}$ et $SH_{1.2}$ entraînent alors "l'affectation" des champs (resp. éléments) de v ou de sous valeurs de v aux champs (resp. éléments) correspondants de u ou de sous noms de u . C'est la raison pour laquelle dans $X_{suppl(u,v)}$ on précise : $elem_k$, ch_y n'est pas modifié en tout schéma fonctionnel w qui ne contient aucune occurence de u_1.

6.3.3 Relations d'identité :

Une relation d'identité est de l'une des deux formes suivantes :

N_1 :=: N_2 ou N_1 :≠: N_2 ; N_1 et N_2 étant deux tertiaires dont le mode commence par "repère" ; sa sémantique est définie en [A-8.3.3 c)] .

L'ensemble de calculs qui lui est associé est le mélange des ensembles de calculs associés à N_1 et N_2 [A-8.3.3.2 a)] ; sa valeur dépend du connecteur d'identité et des valeurs possédées par N_1 et N_2. Ainsi :

6.3.3.1 Systèmes associés à R : N_1 :=: N_2 :

$$\mathcal{S}(R)\begin{cases}\mathcal{M}(R)\begin{cases}\tilde{R} = m(\tilde{N}_1,\tilde{N}_2)\\ \mathcal{M}(N_i) \quad i = 1,2\end{cases}\\ \mathcal{V}(R)\ \{\ \textit{valeur poss}\ R \equiv \textit{cond valeur poss}\ N_1\ \textit{valeur poss}\ N_2\ \textit{vrai faux}\end{cases}$$

6.3.3.2 Systèmes associés à R : N_1 :≠: N_2 :

$$\mathcal{S}(R)\begin{cases}\mathcal{M}(R)\begin{cases}\tilde{R} = m(\tilde{N}_1,\tilde{N}_2)\\ \mathcal{M}(N_i) \quad i = 1,2\end{cases}\\ \mathcal{V}(R)\ \{\ \textit{valeur poss}\ R \equiv \textit{cond valeur poss}\ N_1\ \textit{valeur poss}\ N_2\ \textit{faux vrai}\end{cases}$$

6.3.4 Problème de l'égalité des modes et relation de conformité :

6.3.4.1 Introduction :

Il est nécessaire, pour formaliser les relations de conformité, de pouvoir "comparer" deux modes : c'est-à-dire de savoir si, étant donnés deux modes μ_1 et μ_2 et une information I , l'un des deux théorèmes $\mu_1 \equiv \mu_2$ ou $\neg\mu_1 \equiv \mu_2$ appartient à I ?

Si les schémas d'axiomes logiques et les schémas $SH_{7.6}$ à $SH_{7.12}$ [4.1.5.7] donnent une réponse affirmative dans le cas de modes "simples" (c'est-à-dire obtenus par composition des symboles *long*, *rang* ... à partir des modes primitifs), il n'en

est pas de même dans le cas des modes "récursifs". Par exemple aucun des schémas d'axiomes cités ne permet de prouver l'égalité (ou l'inégalité) des modes définis par les deux déclarations :

mode *a* = *struct* (*rep* *a* *x*)

mode *b* = *struct* (*rep* *b* *x*)

En fait les schémas $SH_{7.6}$ ... $SH_{7.12}$ ne permettent de caractériser que des modes différents. Or le problème de l'égalité des modes est décidable [A5] . Pour permettre les tests de conformité on est amené à modifier légèrement la définition d'une information Algol 68 en y adjoignant des axiomes caractérisant des modes égaux.
(Une formalisation de [A8] serait plus simple à ce sujet car la définition de l'égalité de deux modes est beaucoup plus précise qu'en [A]).

6.3.4.2 Information complétée par rapport aux modes :

a) Rappelons que MODE est défini par l'équation :

$$\text{MODE} = \textit{long}\ \text{MODE} \cup \textit{repère}\ \text{MODE} \cup \Big(\bigcup_{\substack{n \geqslant 1\\ x_1,\ldots,x_n \in L_o^{id}}} \textit{structure}_{x_1\ldots x_n}\ \text{MODE}^{(n)}\Big) \cup \textit{rang}\ \text{MODE} \cup$$

$$\Big(\bigcup_{n \in \mathbb{N}} \textit{procn}_n\ \text{MODE}^{(n)}\Big) \cup \Big(\bigcup_{n \in \mathbb{N}^*} \textit{procr}_n\ \text{MODE}^{(n)}\Big) \cup \Big(\bigcup_{n \in \mathbb{N}^*} \textit{union}_n\ \text{MODE}^{(n)}\Big) \cup$$

$$\textit{indique}\ \text{INDIC} \cup \textit{mode}\ \text{EXVAL} .$$

Mais les schémas du type *mode* EXVAL ne permettant pas de "construire" de nouveaux modes, ils servent seulement à établir un "lien" entre les exemplaires de valeurs et MODE.

Ainsi, pour le seul problème d'égalité de modes, peut-on se restreindre au sous ensemble $\overline{\text{MODE}}$ de MODE défini par :

$$\overline{\text{MODE}} = \textit{long}\ \overline{\text{MODE}} \cup \textit{repère}\ \overline{\text{MODE}} \cup \Big(\bigcup_{\substack{n \geqslant 1\\ x_1,\ldots,x_n \in L_o^{id}}} \textit{structure}_{x_1\ldots x_n}\ \overline{\text{MODE}}^{(n)}\Big) \cup \textit{rang}\ \overline{\text{MODE}} \cup$$

$$\Big(\bigcup_{n \in \mathbb{N}} \textit{procn}_n\ \overline{\text{MODE}}^{(n)}\Big) \cup \Big(\bigcup_{n \in \mathbb{N}^*} \textit{procr}_n\ \overline{\text{MODE}}^{(n)}\Big) \cup \Big(\bigcup_{n \in \mathbb{N}^*} \textit{union}_n\ \overline{\text{MODE}}^{(n)}\Big) \cup$$

$$\textit{indique}\ \text{INDIC} .$$

b) Soit $\mathcal{S}_m$ la structure d'information définie par :

- l'ensemble de symboles fonctionnels :

L_m = INDIC $\cup$ {*long*, *rang*, *repère*} $\cup$ {*procr*$_n$, *union*$_n$, | $n \geq 1$} $\cup$

{*procn*$_n$ | $n \geq 0$} $\cup$ {*structure* $_{x_1 \ldots x_n}$ | $1 \leq i \leq n$; $x_1, \ldots, x_n \in L_o^{id}$}

- l'ensemble de fonctions atomiques :

$A_m = \overline{\text{MODE}} \equiv \overline{\text{MODE}}$

- l'ensemble d'axiomes X_m obtenu par l'union des schémas d'axiomes $SH_{7.6}$ ... $SH_{7.12}$ avec l'ensemble des axiomes caractérisant les éléments de INDIC (ils sont de la forme *indique* D $\equiv$ u et sont définis dans les systèmes d'axiomes apparaissant en [6.2.2.1] et [6.2.2.2])

- Nous noterons F_m l'ensemble des formules de $\mathfrak{S}_m$.

Pour toute information I de $\mathfrak{S}$, nous noterons I_m la restriction de I à $\mathfrak{S}_m$:

$I_m = I \cap F_m$; ou encore $I_m = \mathfrak{R}_m(I)$.

Pour que l'on puisse "comparer" deux éléments de $\overline{\text{MODE}}$ dans I_m il suffirait que I_m soit complète. Comme ceci n'a aucune raison d'être (avec l'exemple de [6.3.4.1] ni *indique* a $\equiv$ *indique* b , ni $\neg$*indique* a $\equiv$ *indique* b n'appartient à I_m) nous allons la compléter

c) Complétée de I_m :

Les axiomes de X_m sont choisis pour que si deux modes μ_1, μ_2 sont différents (au sens intuitif) alors $\mu_1 \equiv \mu_2 \in I_m$.

L'idée permettant de construire une information complète $\overline{I_m}$ contenant I_m est donc : Si deux modes μ_1 et μ_2 ne sont pas (formellement) différents (i.e. $\neg\mu_1 \equiv \mu_2 \in I_m$) alors ils sont égaux (i.e. $\mu_1 \equiv \mu_2$ doit appartenir à $\overline{I_m}$)

Soit donc $\overline{I_m}$ l'extension de I_m définie en ajoutant à X_m l'ensemble d'axiomes :

$$Y = \{\mu_1 \equiv \mu_2 \mid \neg\mu_1 \equiv \mu_2 \quad I_m\} .$$

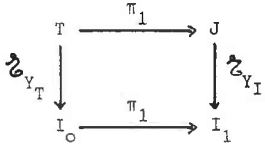
Prouver la complétude de $\overline{I_m}$ se réduit à prouver sa consistance (ce qui est fait en d)).

Remarques :

1) On suppose à priori que I_m , comme I , est consistante.

2) Aucune modification élémentaire ne transforme les axiomes de X_m , c'est-à-dire que I_m est indépendante de l'information I . En particulier on peut définir l'information initiale comme étant la plus petite information I_o contenant

l'ensemble des théorèmes T de $\mathfrak{S}$ (y compris les axiomes définis par les systèmes $\mathfrak{V}$(E) pour toute phrase Algol 68$_o$ E) et Y . Alors toute information I_1 déduite de I_o par application d'une modification de la structure et soit inconsistante, soit complète relativement aux modes, c'est-à-dire que pour tout $\mu_1, \mu_2 \in$ MODE, l'une des deux formules $\mu_1 \equiv \mu_2$ ou $\neg\mu_1 \equiv \mu_2$ appartient à I_1 .
Plus précisément si $\mathfrak{Z}_{Y_I}(J)$ est l'extension de J obtenue par adjonction des axiomes $Y_I = \{\mu_1 \equiv \mu_2 \mid \neg\mu_1 \equiv \mu_2 \notin \mathfrak{R}_m(I)\}$ on démontre simplement que le diagramme suivant est commutatif



où π_1 est une modification élémentaire de la structure $\mathfrak{S}$.

d) Théorème : $\overline{I_m}$ est consistante.

Démonstration :

Il suffit de prouver [2.4.3] qu'elle admet une réalisation. La démonstration repose essentiellement sur [A5] ; en reprenant ses notations, soit $\mathfrak{P}$ l'ensemble des protonotions Algol 68 [A-1.1.2 b)] tel qu'il est précisé par l'article cité.

d1) Définissons une application r de L_m dans $\bigcup_{n \geq 0} \mathfrak{F}(\mathfrak{P}^n, \mathfrak{P})$ par :

i) r(*indique reel*) = réel
r(*indique ent*) = entier
r(*indique car*) = caractère
r(*indique bool*) = booléen
r(*indique proc*) = procédure

ii) r(*long*) est une application de $\mathfrak{P}$ dans $\mathfrak{P}$ définie par :

r(*long*(u) = long u

on définit de même :

r(*rang*), r(*repère*), r(*structure*)$_{x_1 \ldots x_n}$, r(*procn*$_n$), r(*procr*$_n$), r(*union*$_n$)

comme étant des applications à valeurs dans $\mathfrak{P}$

par exemple r(*structure*)$_{x_n \ldots x_n}$: $(u_1, u_2, \ldots, u_n) \longmapsto$ structure avec u_1 à titre x_1 et ... et avec u_n à titre x_n .

iii) pour définir complètement r , il faut caractériser r(*indique* d) pour tout déclareur non primitif ou indicateur d.

On procède de la manière suivante :

soit r_1 une application de L_m dans $\bigcup_{n\geq 0} \mathcal{F}(\mathcal{P}^n,\mathcal{P})$ égale à r sur les éléments de $L_o^{prim} \cup \{long, rang, repère, structure_{x_1 \ldots x_n} \ldots\}$

On pose r_1(*indique* d) = d pour tout $d \in L_n^{nprim} \cup L_o^{ind}$

r_1 se prolonge alors naturellement en une application r_1 de $\overline{MODE}$ dans $\mathcal{P}$; et donc tout axiome *indique* d ≡ u caractérisant un élément $d \in L_o^{nprim} \cup L_o^{ind}$ définit une équation $d = \hat{r}_1(u)$.

On associe de manière naturelle à tout programme Algol 68_o P un système à point fixe formé des équations $d = r_1(u)$ pour tout élément $d \in L_o^{nprim} \cup L_o^{ind}$ qui apparaît dans P . Si le programme est syntaxiquement correct ce système $\mathcal{S}(P)$ (est complet et il) admet une solution unique [A-5.2.2] . Comme tout élément d de $L_o^{nprim} \cup L_o^{ind}$ apparaît dans un et un seul programme Algol 68_o (rappelons [5.1] qu'un tel élément n'est pas réduit à une suite de caractères mais est une arborescence) on définit r(*indique* d) comme étant la valeur de d dans la solution du système où il apparaît.

Ainsi $(\mathcal{P},r)$ est une interprétation de $\mathcal{S}_m$.

iv) C'est de plus une réalisation de I_m en effet :

- Il est clair d'après les propriétés de $\mathcal{P}$ [A5-2.1 axiome 1] que pour tout axiome $\mathfrak{X}$ provenant d'un des schémas $SH_{7.6} \cdots SH_{7.12}$, $\tilde{r}(\mathfrak{X})$ = VRAI ($\tilde{r}$ est la fonction de vérité associée à r).

Exemple : Si $\mathfrak{X}$ provient de $SH_{7.7}$ on a =*repère* μ_1 ≡ *repère* $\mu_2 \supset \mu_1 \equiv \mu_2$. Alors de $\hat{r}$(*repère* μ_1) = $\hat{r}$(*repère* μ_2) on déduit $\mu_1 = \mu_2$ (en tant que protonotions)

et donc $\hat{r}(\mu_1) = \hat{r}(\mu_2)$.

- Si $\mathfrak{X}$ est un axiome de la forme *indique* d ≡ u , d est un déclareur de mode qui apparaît dans un certain programme P ; l'équation associée $d = \hat{r}_1(u)$ admet comme solution r(*indique* d), c'est-à-dire : $\hat{r}$(*indique* d) = $\hat{r}$(u)

ainsi : $\tilde{r}$(*indique* d ≡ u) = VRAI .

d2) $(\mathcal{P},r)$ est encore une réalisation de $\overline{I_m}$:

I_m étant une extension de I_m définie par l'ensemble d'axiomes Y , il suffit de vérifier que si u ≡ v ∈ Y , $\tilde{r}$(u ≡ v) = VRAI .

Soient u,v ∈ $\overline{MODE}$ tels que ¬u ≡ v ∉ I_m. A la réunion sur P des systèmes $\mathcal{S}$(P) (iii)) , on ajoute les deux équations $x_1 = \hat{r}_1(u)$

$x_2 = \hat{r}_1(v)$.

De ce système (qui contient une infinité d'équations) on peut extraire un sous système complet et fini contenant les deux équations précédentes

(Exemple : si u = *indique* a avec *mode* a = *struct*(*rep* b x)
mode b = *struct*(*rep* a y)

et v = *indique* c avec *mode* c = *struct*(*rep* c x)
mode d = *struct*(*rep* d y)

le système obtenu sera : $x_1 = \hat{r}_1(u)$
$x_2 = \hat{r}_1(v)$
$a = \hat{r}_1(structure_x\ repère\ b)$
$b = \hat{r}_1(structure_y\ repère\ a)$
$c = \hat{r}_1(structure_x\ repère\ d)$
$d = \hat{r}_1(structure_y\ repère\ c)$) .

Alors en appliquant [A5] à ce sous système, les définitions de la relation binaire Γ et de l'ensemble E [A5-4.2] confrontées aux axiomes $SH_{7.6} \cdots SH_{7.12}$ permettent de montrer immédiatement par récurrence sur n que si (1,2) ∈ Γ^n(E) alors ¬u ≡ v ∈ I_m .

Ainsi ¬u ≡ v ∉ I_m ⟹ (1,2) ∉ Γ^*(E)

et donc [A5 proposition 4] ¬u ≡ v ∉ I_m ⟹ $\hat{r}$(u) = $\hat{r}$(v)

ce qui termine la démonstration.

Remarque : Une autre réalisation est définie en [SI7] .
On peut alors définir les relations de conformité, en tenant compte de la remarque 2 ci-dessus.

6.3.4.3 Relation de conformité :

Une relation de conformité R [A-8.3.2.1] est de la forme N :: E ou N ::= E ; N et E sont des tertiaires et le mode de N commence par "repère".
L'élaboration de R est décrite en [A-8.3.2.2] pas 1... pas 4. Il s'agit d'élaborer E et de comparer le mode de N au mode de E ou d'une valeur repérée par E ou par un de ses sous noms. Cette définition est récursive, nous introduirons deux schémas de calculs : $CONF_1$ qui traite le cas des relations N :: E , $CONF_2$ celui des relations N ::= E .

a) Double système associé à R :

$$\mathcal{S}_{(R)}\begin{cases}\mathcal{M}_{(R)}\begin{cases}\tilde{R} = \tilde{E}\ .\ CONF_i(R,N,poss\ E)\\ \mathcal{M}_{(CONF_i)}\end{cases}\\ \mathcal{V}_{(R)} = \emptyset\end{cases}$$

dans lequel i = 1 si R est la relation N :: E , i = 2 si elle est N ::= E .
$\mathcal{V}_{(R)}$ est vide car *poss* E sera défini dynamiquement par $CONF_i$.

b) Le schéma de calculs $CONF_1$:

$CONF_1$ est un schéma de calculs à trois paramètres : une relation de conformité R dont le connecteur est "::" , le tertiaire N qui se trouve en partie gauche de R et un exemplaire de valeur v . Il permet d'exprimer [A-8.3.2.2] pas 2 et 3

$$\mathcal{M}_{(CONF_1)}\begin{cases}CONF_1 = \lambda R\ .\ \lambda N\ .\ \lambda v\ .\\ \quad \Big[\Big(\bigcup_{\substack{\mu_1,\ldots,\mu_n \in MODE\\ \text{pour } n\geqslant 1}} j(mode\ poss\ N,\ repère\ union_n\ \mu_1\ldots\mu_n)\ .\\ \quad \Big[\bigcup_{1\leqslant i\leqslant n} j(mode\ v,\ \mu_i)\ .\ ver(R)\Big]\Big) \cup\\ \quad \Big(\bigcup_{\nu \in MODE} j(mode\ v,\nu)\ .\ \Big[\bigcup_{\substack{\mu_1,\ldots,\mu_n \in MODE-\{\nu\}\\ \text{pour } n\geqslant 1}} j(mode\ poss\ N, repère\ union_n\ \mu_1\ldots\mu_n)\ .\\ \quad \Big[\Big(\bigcup_{\mu \in repère\ MODE} j(\nu,\mu)\ .\ CONF_1(R,N;rep\ v)\Big) \cup\\ \quad \Big(\bigcup_{\mu \in MODE-repère\ MODE} j(\nu,\mu)\ .\ fau(R)\Big)\Big]\Big]\Big)\Big]\end{cases}$$

Commentaires :

- le premier ensemble de calculs traite le cas où le mode de N est repère d'un mode uni à partir du mode de v ; rappelons que ceci comprend le cas où ce mode est *repère mode* v (n = 1) . Il se termine par la modification élémentaire ver(R) (définie en d)) qui associe *vrai* à R .

- le deuxième ensemble traite l'autre possibilité, deux cas sont à envisager :
  - le mode de v commence par "repère" et alors on confronte N et *rep* v
  - sinon on associe à R la valeur *faux*, c'est le rôle du schéma de modifications fau.

c) Le schéma de calculs $CONF_2$:

C'est un schéma de calculs à trois paramètres, de même types différents que ceux de $CONF_2$. Il permet d'exprimer [A-8.3.2.2] pas 2,3 et 4 . Sa définition est obtenue à partir de celle de $CONF_1$ en faisant suivre ver(R) de :

$$\tilde{N}\ .\ coh(poss\ N,v)\ .\ AFFECT(poss\ N,V)$$

qui permet d'exprimer l'affectation [6.3.2] de v à *poss* N ; affectation qui comprend certains tests de cohérence et qui doit être précédée de l'élaboration de N.

Remarque : l'introduction d'un symbole fonctionnel *conn* (connecteur) associant son connecteur à toute relation de conformité permettrait de regrouper $CONF_1$ et $CONF_2$.

d) Les schémas de modifications élémentaires ver et fau :

Ce sont deux schémas à un paramètre (relation de conformité R) , ils modifient le seul symbole fonctionnel *poss* :

$$\sigma_{ver(R)}(poss) = poss' \quad \text{et} \quad \sigma_{fau(R)}(poss) = poss' \ .$$

Ils sont définis par :

$$\chi_{ver(R)} = \{poss'\ R \equiv poss\ \underline{vrai}\} \cup \{poss'\ u \equiv poss\ u \mid u \in IDENT - \{R\}\}$$

$$\chi_{fau(R)} = \{poss'\ R \equiv poss\ \underline{faux}\} \cup \{poss'\ u \equiv poss\ u \mid u \in IDENT - \{R\}\} \ .$$

6.3.5 Formules

Dans le langage pivot toute formule E est de l'une des deux formes :

. $\square E_1$

. $E_1\ \square\ E_2$

où $\square$ est un opérateur ; E_1 et E_2 sont des tertiaires (soit des formules, soit des secondaires) [A-8.4] .

L'élaboration d'une formule consiste essentiellement en un appel de la routine possédée par son opérateur.

6.3.5.1 Systèmes associés à une formule monadique :

Soit donc E : $\square E_1$. Les appels de routines sont caractérisés en [6.3.1.1] , en adoptant les notations de ce paragraphe :

$$\mathcal{S}(E)\begin{cases}\mathcal{M}(E)\begin{cases}\tilde{E} = \mathcal{C}_E(\delta.\tilde{\psi}(poss\ \square))\\ \mathcal{M}(E_i)\quad i = 1,2\end{cases}\\ \mathcal{V}(E)\ \{\ poss\ E \equiv sub\ E\ poss\ poss\ \square\end{cases}$$

poss □ est la routine associée à □, nous verrons que c'est une phrase Algol 68_o (symbole 0-aire). $\tilde{\psi}(poss\ \square)$ [6.1.2] est l'ensemble de calcul associé à *poss* □. $\mathcal{C}_E$ et *sub* permettent de traiter le cas des variables locales au corps de procédure ainsi que la transmission du paramètre effectif E_1 .

6.3.5.2 Systèmes associés à une formule dyadique :

Si E : E_1 □ E_2 alors, comme ci-dessus :

$$\mathcal{S}(E)\begin{cases}\mathcal{M}(E)\begin{cases}\tilde{E} = \mathcal{C}_E(\delta.\tilde{\psi}(poss\ \square))\\ \mathcal{M}(E_i)\quad i = 1,2\end{cases}\\ \mathcal{V}(E)\ \{\ poss\ E \equiv sub\ E\ \ poss\ poss\ \square\ .\end{cases}$$

6.3.6 Sélections :

Soit P la sélection *x* de E , *x* est un sélecteur de champ et E un secondaire [A-8.5.2] .

L'élaboration de P est décrite en [A-8.5.2.1] , nous la formaliserons par le double système suivant :

$$\mathcal{S}(P)\begin{cases}\mathcal{M}(P)\begin{cases}\tilde{P} = \tilde{E}\\ \mathcal{M}(E)\end{cases}\\ \mathcal{V}(P)\begin{cases}\neg\ valeur\ poss\ E \equiv nil\\ \{valeur\ poss\ P \equiv valeur\ ch_x\ poss\ E) \wedge (valeur\ v \equiv valeur\ poss\ P \supset\\ \quad \neg v \equiv poss\ P) \mid v \in EXVAL - \{poss\ P\}\}\ .\end{cases}\end{cases}$$

On exprime que la valeur de E doit être différente de nil et que la sélection possède un nouvel exemplaire de valeur.

Remarquons que la caractérisation, dans notre formalisation, de ch_x u lorsque u ∈ EXNOM (schéma d'axiomes $SH_{1.1}$) évite de traiter distinctement le cas où *poss* E est un exemplaire de nom.

6.3.7 Générateurs :

6.3.7.1 Introduction :

Soit G le générateur *loc* D|q (resp. *tas* D) de déclareur effectif D [5.2 c)]. L'élaboration de G est définie en [A-8.5.1.2] et se ramène essentiellement à celle de son déclareur effectif D [A-7.1.2 d)] . De plus c'est le seul endroit où l'élaboration d'un tel générateur est utilisée ; c'est la raison pour laquelle nous n'associons pas, dans notre formalisation, de valeur à un déclareur effectif D nous contentant de le faire pour un générateur (rappelons que $\mathcal{V}(D)$ caractérise *indique* D ainsi que les éventuels bornices de D [6.2.2]).

Etudions en détail la définition de l'élaboration d'un déclareur effectif :

- cette définition peut être récursive
- la récursivité porte sur la forme du déclareur, c'est-à-dire sur D et les sous déclareurs de D (après développement [A-7.1.2 c)])
- la définition n'est récursive que si D est un déclareur de mode structure ou de mode valeur multiple

Ceci est dû au fait que seuls ces déclareurs conservent "l'effectivité" des sous déclareurs alors que dans les autres procédés de construction de déclareurs de mode (routine, repère, union) les sous déclareurs sont virtuels [A-7.1.1 w), ℓ), jj)].

- Un traitement particulier est cependant réservé aux déclareurs de mode union pour permettre de choisir un des sous déclareurs.
- Enfin il faut "créer" : un nom u de mode "repère de D", un nouvel exemplaire de valeur v que l'on fait repérer par ce nom, ainsi qu'un certain nombre (éventuellement nul) de sous noms de u repérant les sous valeurs correspondantes de v .

Cette définition récursive permet, et c'est là son seul rôle, de définir les descripteurs de toutes les valeurs multiples repérées par les noms que l'on crée (dans le cas où les déclareurs correspondants sont effectifs).

La formalisation qui suit repose essentiellement sur cette remarque : l'élaboration d'un générateur sera traduite par un schéma de calculs GEN qui permet d'exprimer la récursivité de la définition ; le traitement relatif aux déclareurs de valeurs multiples sera exprimé par un schéma de modifications élémentaires "créa".

Remarque :

Nous n'allons cependant pas traduire entièrement [A-7.1.2 d)] : sauf dans le cas d'un nom de valeur multiple caractérisé par un déclareur effectif, nous ne définirons pas de "nouvel exemplaire de certaines valeurs" à faire repérer par les

noms créés par le générateur [A-7.1.2 d)] pas 2), 3), 4) et 7). En effet le rôle d'un générateur est de "créer" des noms et de donner certains "renseignements" sur ces noms : portée [A-8.5.1.2 b)], mode, descripteur de la valeur repérée dans le cas d'une valeur multiple.

Ces renseignements seront donnés ici par des théorèmes caractérisant les symboles fonctionnels *portée, mode, rep, poss,* a_i, b_i, s_i, t_i, *dim, sousflex* . La plupart de ces théorèmes (sauf ceux concernant les descripteurs) se déduisant de certains schémas d'axiomes propres et d'axiomes caractérisant l'exemplaire de nom possédé par le générateur G.
Cette demande est bien équivalente, quant à l'utilisation ultérieure des noms, à celle qui consiste à faire repérer par les noms que l'on crée des exemplaires v de certaines valeurs, sans autres renseignements sur v que ceux concernant l'éventuel descripteur.

6.3.7.2 Systèmes associés à un générateur G :

G apparaît dans un certain programme P . Dans le langage pivot à G est associée sa portée q (déterminée par la plus petite région du programme P qui contient G).

Soit p défini par $p = \begin{cases} q & \text{si } G = \underline{loc}\ D|q \\ \theta & \text{sinon} \end{cases}$

le double système associé à G est défini par :

$$\mathcal{S}(G) \begin{cases} \mathcal{M}(G) \begin{cases} \tilde{G} = GEN(poss\ G, D) \\ \mathcal{M}(GEN) \\ \mathcal{M}(D) \end{cases} \\ \mathcal{V}(G) \begin{cases} mode\ poss\ G \equiv repère\ indique\ D \\ portée\ poss\ G \equiv p \\ \{\neg\, valeur\ poss\ G \equiv valeur\ poss\ G_1 \mid G_1 \in L_o^{gen} - \{G\}\} \end{cases} \end{cases}$$

Remarque 1 :

Les deux premiers axiomes de $\mathcal{V}(G)$ caractérisent le mode et la portée du nom créé par le générateur, ainsi que tout sous nom de *poss* G grâce aux schémas d'axiomes : $SH_{1.4}$ ... $SH_{1.7}$

On exprime d'autre part que le nom créé est différent de tout autre nom, L_o^{gen} est l'ensemble des générateurs Algol 68_o .

Remarque 2 :

On ne donne aucun axiome concernant *rep poss* G alors que pour traduire correctement [A-7.1.2 d)] il semble nécessaire de préciser son mode. Or deux cas sont envisageables :

- Le mode de *poss* G commence par *repère union* et alors peu de renseignements peuvent être donnés : c'est ce qui est exprimé en [A-7.1.2 d) pas 3] où on "choisit" un certain mode à partir duquel *indique* D est uni. A ce cas correspondent les schémas d'axiomes : $SH_{1.5}$, $SH_{1.7}$ et $SH_{1.9}$

- le mode de *poss* G ne commence pas par *repère union*. Alors les schémas d'axiomes : $SH_{1.4}$, $SH_{1.6}$ et $SH_{1.8}$
définissant le mode des exemplaires de valeurs possédés par G et les sous noms de *poss* G .
Il faut définir maintenant le schéma GEN.

6.3.7.3 Définition du schéma de calculs GEN :

GEN est un schéma à deux paramètres u et D où u est un exemplaire de valeur et D un déclareur effectif. GEN est défini de manière récursive, la récursivité portant sur D . Il permet d'exprimer l'élaboration du descripteur de la valeur repérée par u si u est un exemplaire de nom de valeur multiple.
Tout d'abord définissons informellement GEN.

a) Si D n'est pas un déclareur de mode structure ou valeur multiple :
Alors [A-7.1.2 d) pas 2 et 3] il faut créer un nom et lui faire repérer une certaine valeur. Nous avons vu (remarques de [6.3.7.1] et [6.3.7.2] que certains axiomes permettent d'associer à G tous les renseignements souhaités, sans toutefois lui faire repérer de valeur.

Ainsi cette élaboration ne correspond à aucune action : GEN(u,D) = Λ .

b) D est un déclareur de mode structure :
Soit D = *struct*($D_1 x_1, \ldots, D_n x_n$) . Il faut exprimer [A-7.1.2 d) pas 4] que l'élaboration de D est l'élaboration collatérale des D_i :

$$GEN(u,D) = \underset{1 \leq i \leq n}{\text{co}} \ (GEN(ch_{x_i}\ u, D_i))$$

les ch_{x_i} u sont les sous noms directs de u .

c) D est un déclareur de mode de valeur multiple :
Soit D = [$A_1\Gamma_1$: $B_1\Delta_1$,..., $A_n\Gamma_n$: $B_n\Delta_n$] D' un déclareur effectif de valeur multiple. Il s'agit d'exprimer [A-7.1.2 pas 5,6,7] , c'est-à-dire :

- l'élaboration des A_i, B_i
- la création d'un exemplaire de valeur multiple dont le descripteur est spécifié par les A_i, B_i, Γ_i, Δ_i .

L'élaboration des A_i,B_i est exprimée par le schéma de calculs D [6.2.2.1 f)];

de plus si D_1 est un sous déclareur d'un déclareur D définissant un générateur G (*loc* (resp. *tas*) D) , $\mathcal{M}(D)$ contient $\mathcal{M}(D_1)$ qui définit $\tilde{D}_1$. Ainsi grâce à [6.2.2.1] , $\mathcal{M}(G)$ contient les équations définissant les calculs associés à l'élaboration de tous les bornices de D ou des déclareurs effectifs contenus dans D . La création du descripteur de l'exemplaire de valeur multiple repéré par u est formalisée par le schéma de modifications élémentaires "créa"

$$GEN(u,D) = \tilde{D} \,.\, créa(u,D) \,.\, \bigcup_{\substack{I_1,\ldots,I_n \in NOTENT \\ J_1,\ldots,J_n \in NOTENT}} [j(\textit{valeur poss } A_1,I_1) \ldots j(\textit{valeur poss } A_n,I_n) \,.$$

$$j(\textit{valeur poss } B_1,J_1) \ldots j(\textit{valeur poss } B_n,J_n) \,.$$

$$\underset{I_k \leqslant i_k \leqslant J_k}{\text{pour } 1 \leqslant k \leqslant n} (GEN(\textit{elem}_n \text{ u } i_1 \ldots i_n, D'))]$$

en regroupant les trois cas précèdents on définit formellement GEN par le système :

$\mathcal{M}(GEN)$:

$$GEN = \lambda N \,.\, \lambda D \,.\, \bigcup_{\mu \in RSTMUL \cup AUTRE \cup RUAUTRE \cup UNION} j(\textit{indique } D,\mu) \quad \cup$$

$$\lambda N \,.\, \lambda D \,.\, \bigcup_{\substack{D_1,\ldots,D_n \in INDIC \\ x_1,\ldots,x_n \in L_o^{id} \\ \text{pour } n \geqslant 1}} j(\textit{indique } D, \textit{indique } \underline{\textit{struct}}(D_1 x_1,\ldots, D_n x_n) \,.$$

$$\underset{1 \leqslant i \leqslant n}{\text{pour}} (GEN(ch_{x_i} \text{ u},D_i)) \cup$$

$$\lambda N \,.\, \lambda D \,.\, \bigcup_{\substack{n \geqslant 1 \\ D' \in INDIC}} j(\textit{indique } D, rang^{(n)} \textit{indique } D') \,.\, \tilde{D} \,.\, créa(D,u) \,.$$

$$[j(\textit{valeur poss } a_1 D,I_1) \ldots j(\textit{valeur poss } a_n D,I_n) \,.$$

$$\bigcup_{\substack{I_1,\ldots,I_n \\ J_1,\ldots,J_n \in NOTENT}}$$

$$j(\textit{valeur poss } b_1 D,J_1) \ldots j(\textit{valeur poss } b_n D,J_n) \,.$$

$$\underset{I_k \leqslant i_k \leqslant J_k}{\text{pour } 1 \leqslant k \leqslant n} (GEN(\textit{elem}_n \text{ u } i_1 \ldots i_n, D'))] \,.$$

(Rappelons que a_i D (resp. b_i D) est la $i^{ème}$ borne inférieure (resp. supérieure) du $i^{ème}$ bornice contenu dans D).

Il reste maintenant à définir "créa".

6.3.7.4 Définition du schéma de modifications élémentaires créa :

créa est un schéma de modifications à deux paramètres u et D , qui n'est défini que si D est un déclareur effectif de valeur multiple. Il permet d'exprimer [A-7.1.2 pas 6 et 7] .

créa (u,D) modifie les symboles *valeur*, *rep*, a_i, b_i, s_i, t_i et *dim* et laisse les autres symboles invariants. Il est défini par :

$$\chi_{créa(u,D)} = \{\textit{valeur' rep'} \text{ u} \equiv \textit{valeur} \text{ v} \supset \neg \textit{rep'} \text{ u} \equiv \text{v} \mid \text{v} \in EXVAL\}$$

$$\{\textit{dim' rep'} \text{ u} \equiv \textit{dim} \text{ D}\}$$

$$\{\textit{dim} \text{ D} \equiv n \supset \bigcup_{1 \leqslant i \leqslant n} (a_i' \textit{ rep'} \text{ u} \equiv \textit{valeur poss } a_i \text{ D} \wedge b_i' \textit{ rep'} \text{u} \equiv$$

$$\textit{valeur poss } b_i \text{ D} \wedge s_i' \textit{ rep'} \text{ u} \equiv s_i \text{ D} \wedge$$

$$t_i' \textit{ rep'} \text{ u} \equiv t_i \text{ D} \mid n \in NOTENT \,,\, n \geqslant 1\}$$

$$\{\textit{valeur'} \text{ v} \equiv \textit{valeur} \text{ v} \mid \text{v} \in EXVAL - \{\textit{rep'} \text{ u}\}\}$$

$$\{\textit{rep'} \text{ v} \equiv \textit{rep} \text{ v} \mid \text{v} \in EXNOM - \{\text{u}\}\}$$

$$\{\textit{dim'} \text{ v} \equiv \textit{dim} \text{ v} \wedge a_i' \text{ v} \equiv a_i \text{ v} \wedge b_i' \text{ v} \equiv b_i \text{ v} \wedge s_i' \text{ v} \equiv s_i \text{ v} \wedge$$

$$t_i' \text{ v} \equiv t_i \text{ v} \mid \text{v} \in EXVAL - \{\textit{rep'} \text{ u}\}\} \;;\; i \geqslant 1\} \,.$$

Remarque 1 : La formalisation précèdente doit exprimer que deux occurences d'un même générateur Algol 68 créent deux noms distincts. Ceci est rendu possible car un générateur Algol 68_o n'est pas une suite de caractères mais un arbre syntaxique ; ce qui permet de distinguer ici les occurences d'un même générateur (Algol 68).

Remarque 2 : Les schémas de calculs et de modifications précédents sont définis aussi bien lorsque D est un indicateur qu'un déclareur de mode , ceci grâce aux définitions du schéma de calculs $\tilde{\psi}$ [6.1.2] et d'une déclaration de mode [6.2.2] .

6.3.8 Tranches :

Rappelons l'hypothèse selon laquelle dans le langage noyau tout tranche F est de la forme $E\,[A_1\Gamma_1 B_1 \text{ apd } C_1 \,,\ldots,\, A_n\Gamma_n B_n \text{ apd } C_n]$

où
- Γ_i prend la valeur 1 si le $i^{ème}$ indexeur est un massicot, la valeur 0 si c'est un indice (et alors nécessairement $A_i = B_i = C_i$)
- E , A_i , B_i , C_i (1⩽i⩽n) sont des propositions unitaires (E est un primaire, A_i, B_i, C_i sont des tertiaires)

En particulier F est un élément de tableau si et seulement si $A_i = B_i = C_i$ pour 1⩽i⩽n .

L'élaboration de F consiste [A-8.6.1.2] en l'élaboration collatérale de E, A_i, B_i, C_i $(1 \leqslant i \leqslant n)$ et en la définition de *poss* F . Ce que l'on exprime par le double système :

$$\mathcal{M}(F)\begin{cases} \tilde{F} = m(\tilde{E},\ \underset{1\leqslant i\leqslant n}{m}\ (\tilde{A}_i, \tilde{B}_i, \tilde{C}_i)) \\ \mathcal{M}(E) \\ \mathcal{M}(A_i) \\ \mathcal{M}(B_i) \qquad i = 1,\ldots,n \\ \mathcal{M}(C_i) \end{cases}$$

$$\mathcal{V}(F)\begin{cases} \neg\, \textit{valeur poss } E \equiv \textit{nil} \\ \{\textit{mode poss } E \equiv \mu \supset \textit{inf } a_j \textit{ poss } E \ \ \textit{valeur poss } A_j \equiv \textit{vrai} \wedge \\ \qquad \textit{inf valeur poss } B_j \ \ b_j \textit{ poss } E \equiv \textit{vrai} \mid \mu \in \textit{rang MODE}, \ 1 \leqslant i \leqslant n\} \\ \{\textit{mode rep poss } E \equiv \mu \ \ \textit{inf } a_j \textit{ rep poss } E \ \textit{valeur poss } A_j \equiv \textit{vrai} \wedge \\ \qquad \textit{inf valeur poss } B_j \ \ b_j \textit{ rep poss } E \equiv \textit{vrai} \mid \mu \in \textit{rang MODE}, \ 1 \leqslant i \leqslant n\} \\ \{(\bigwedge_{1\leqslant i\leqslant n} \Gamma_i \equiv 0) \supset \textit{valeur poss } F \equiv \textit{valeur } \textit{elem}_n \textit{ poss } E \ \textit{valeur poss } A_1 \ldots \\ \qquad \textit{valeur poss } A_n\} \\ \{(\bigvee_{1\leqslant i\leqslant n} \Gamma_i \equiv 1) \supset \textit{valeur poss } F \equiv \textit{valeur } \textit{tranche}_n \textit{ poss } E \ \textit{valeur poss } A_1 \Gamma_1 \\ \qquad \textit{valeur poss } B_1 \ \textit{valeur poss } C_1 \ \ldots \ \textit{valeur poss } A_n \Gamma_n \\ \qquad \textit{valeur poss } B_n \ \textit{valeur poss } C_n\} \\ \{\textit{valeur } v \equiv \textit{valeur poss } F \supset \neg\, v \equiv \textit{poss } F \mid v \in \text{EXVAL} - \{\textit{poss } F\}\} \ . \end{cases}$$

Commentaires :

1) Le premier axiome exprime que la valeur du primaire E ne peut être *nil* [A-8.6.1.2 pas 1]

2) Les deux premiers ensembles d'axiomes expriment, en distinguant le cas où *poss* E est un nom du cas où il ne l'est pas :

- que la borne inférieure (resp. supérieure) stricte du $i^{ème}$ massicot doit être supérieure (resp. inférieure) à la $i^{ème}$ borne inférieure (resp. supérieure) du primaire (ou de la valeur repérée par ce primaire) [A-8.6.1.2 pas 5] .

- Si le $i^{ème}$ indexeur est un indice on exprime ainsi qu'il doit être compris entre la $i^{ème}$ borne inférieure et la $i^{ème}$ borne supérieure du primaire (ou de la valeur qu'il repère) [A-8.6.1.2 pas 4] .

3) Le quatrième axiome traite le cas d'un élément de tableau (lorsque tous les indexeurs sont des indices) ; le cas où *poss* E est un exemplaire de nom se déduit de $SH_{1.2}$.

4) Le cinquième axiome est relatif aux tranches (ou aux noms de tranches à l'aide de $SH_{1.3}$). Rappelons que dans notre formalisation nous introduisons des symboles fonctionnels différents $elem_n$ et $tranche_n$ pour distinguer les éléments de tableaux des autres tranches.

5) Le dernier ensemble d'axiomes exprime que *poss* F est un nouvel exemplaire de valeur.

Enfin les schémas d'axiomes : $SH_{2.2}$, $SH_{2.3}$ permettent d'exprimer complètement [A-8.6.1.2] .

6.3.9 Notations de valeurs simples et Identificateurs :

a) Les notations d'entiers (resp. de réels, de booléens, de caractères) sont les éléments de L_o^{ent} (resp. L_o^{reel}, L_o^{bool}, L_o^{car}). Les identificateurs sont les éléments de L_o^{id}.

A toute notation N de valeur simple est associé un exemplaire de valeur : *poss* N ne sera pas défini par un axiome (mais dans une interprétation quelconque on lui associe une valeur). De plus le mode de l'exemplaire de valeur possédé par une telle notation est précisé par les schémas d'axiomes $SH_{7.2}$ et $SH_{7.3}$, sa portée par $SH_{6.12}$.

L'exemplaire de valeur associé à un identificateur, est défini par une déclaration d'identité qui précise également son mode et sa portée.

Ainsi, aussi bien pour un identificateur que pour une notation de valeur simple, le double système associé n'apporte aucun "renseignement". Il faut cependant le définir puisque la définition du double système associé à un programme est récursive.

Si N est une notation de valeur simple ou un identificateur :

$$\mathcal{G}(N)\begin{cases} \mathcal{M}(N) \ \{\tilde{N} = \Lambda \\ \mathcal{V}(N) \ = \ \emptyset \ . \end{cases}$$

Remarque : On ne traduit pas ainsi tout à fait Algol 68 pour qui deux occurences différentes de la même notation (resp. du même identificateur) possèdent des exemplaires de valeurs différents. Une manière d'exprimer cette condition

consisterait à numéroter les différentes occurences. Ici cette propriété sera exprimée au niveau des déclarations d'identité et des affectations : lorsqu'une occurence de notation ou d'identificateur N se trouve en partie droite d'une telle phrase, on associe au membre de gauche un nouvel exemplaire de la valeur possédée par N [6.2.3.3] et [6.2.3.6] .

b) *nil* et *fant* jouent des rôles assez proches de ceux des notations de valeurs simples :

- La différence entre *nil* et une notation de valeur simple provient de ce que son mode dépend du contexte [A-8.2.7.2 c)] . En réalité ce mode n'est d'aucun intérêt en Algol 68 où on interdit de le tester. Pour éviter d'alourdir la formalisation, nous ne numéroterons pas chaque occurence de *nil* dans un programme et donc nous ne lui associerons pas de mode.

La portée de *poss nil* est définie par le schéma d'axiomes $SH_{6.12}$.
Ainsi comme l'élaboration de nil n'entraîne aucune action [A-8.2.7.2 c)] :

$$\mathcal{S}(nil)\begin{cases}\mathcal{M}(nil)\ \{\ \widetilde{nil} = A\\ \mathcal{V}(nil) = \emptyset\end{cases}$$

- En Algol 68_o les seuls symboles *fant* considérés seront de genre neutre, sauf dans le cas des routines où nous introduisons des symboles spécifiques $\sim_i$ ($i \geqslant 1$) pour traiter les paramètres formels [6.3.10].

Ainsi [A-8.2.7.2 a)] :

$$\mathcal{S}(fant)\begin{cases}\mathcal{M}(fant)\ \{\ \widetilde{fant} = A\\ \mathcal{V}(fant) = \emptyset\end{cases}$$

6.3.10 Notations de routines :

6.3.10.1 Introduction :

Soit F la notation de routine $(D_1 z_1\ \&\ D_2 z_2\ \&\ \ldots\ \&\ D_n z_n)\ D : E|p$
où E est une proposition unitaire contenant les paramètres formels $z_1, \ldots, z_n$; est soit le symbole continuer (";") soit le symbole virgule (",") [A-5.4.1] . De plus dans le langage pivot Algol 68_o, $(\ )\ D : E|p$ est une notation de procédure sans paramètre. Enfin si D est absent la procédure est sans résultat.

Une telle notation F "possède une routine qui est une même suite de symboles qu'une certaine proposition fermée" F' [A-5.4.2].

En Algol 68_o F' est la proposition :

$$(D_1 z_1 = \sim_1\ \&\ D_2 z_2 = \sim_2\ \ldots\ \&\ D_n z_n = \sim_n\ ;\ E'|p$$

- les $\sim_i$ sont des identificateurs qui n'apparaissent que dans les routines ; ils jouent le rôle du symbole *fant* [A-5.4.2 pas 2] et sont numérotés pour permettre de les associer simplement aux paramètres effectifs correspondants lors d'un appel.

- En Algol 68_o les forceurs sont exprimés à l'aide de certains "opérateurs" (symboles fonctionnels *derep, elarg*) qui permettent "d'effectuer" les modifications du langage Algol 68 ; ainsi E' est déduit de E de telle manière que son mode (s'il existe) soit celui précisé par le déclareur D .

L'élaboration (resp. la valeur) d'un appel de cette routine est l'élaboration (resp. la valeur) de la proposition fermée déduite de F' en remplaçant les sybmoles $\sim_i$ par les paramètres effectifs correspondants [A-8.6.2.2] .

6.3.10.2 Objet possédé par une notation de routine :

Une notation de routine doit permettre d'accéder, lors d'un appel, à un schéma de calculs et à un exemplaire de valeur.
Dans un certain nombre de langages de programmation une solution consisterait à associer à une telle notation (si la procédure est à $n \geqslant 0$ paramètres) :

- un schéma de calculs n-aire
- un symbole fonctionnel à n-arguments (si c'est une procédure à résultat).

Il serait alors nécessaire de définir un lien entre la notation et les schéma et symbole associés ; par exemple à l'aide de numéros [3.5] . Remarquons également que dans cette éventualité certains symboles fonctionnels à n arguments ($n > 0$) devraient appartenir à l'ensemble S des schémas fonctionnels de la structure, ce qui reviendrait à transformer notre système formel en une sorte de λ-calcul ([L1] , [L2]) , le nombre de types serait infini et une interprétation serait de la forme des modèles proposés par SCOTT ([S54], [S55], [PF8], [L7]) .

En Algol 68 la définition sera plus simple, le problème de transmission des paramètres sera résolu conformément à l'esprit de [A] , à l'aide de "substitutions" (exprimées par le symbole fonctionnel 2-aire *sub*).

6.3.10.3 Double système associé à une notation de routine :

(avec les notations de [6.3.10.1]) :

a) Système de calculs :

$$\mathcal{M}(F) \begin{cases} \overset{\vee}{F} = \Lambda \\ \mathcal{M}(F') \end{cases}$$

On exprime que l'élaboration de F ne correspond à aucune action et on place dans le système le schéma de calculs $\overset{\vee}{F'}$ qui sera utilisé dans les appels de la routine.

b) Système d'accès :

Si F est de la forme $(D_1 z_1 \ \& \ D_2 z_2 \ \ldots \ \& \ D_n z_n) : E|p$

$$\mathcal{V}(F) \begin{cases} \textit{poss} \ \ F \equiv F' \\ \textit{mode} \ \ F \equiv \textit{procn}_n \ \textit{indique} \ D_1 \ \ldots \ \textit{indique} \ D_n \\ \textit{portée} \ F' \equiv p \\ \quad \mathcal{V}(F') \end{cases}$$

où p est la portée de cette notation F ; c'est un élément de PORTEE qui est associé à F dans le langage pivot [5.2 c)].

On exprime ainsi que F possède la routine F' (symbole 0-aire). C'est le seul cas où le symbole *poss* associe un objet externe à un autre. Si maintenant F définit une procédure fonction :

$$(D_1 z_1 \ \& \ D_2 z_2 \ \ldots \ \& \ D_n z_n) \ D : E|p$$

seul le deuxième axiome de $\mathcal{V}(F)$ est différent :

mode F' ≡ $\textit{procr}_{n+1}$ *indique* D_1 ... *indique* D_n *indique* D

6.3.11 Appel de procédure :

6.3.11.1 Introduction :

Soit A_p l'appel $G(E_1,\ldots,E_n)$; G est un primaire possédant une routine à n paramètres ($n \geqslant 0$) ; $E_1,\ldots,E_n$ sont les paramètres effectifs.

L'élaboration de A_p est définie en [A-8.6.2.1]. Elle consiste essentiellement en :

i) l'élaboration du primaire G qui "fournit" une notation de routine

ii) la protection de la routine F' obtenue à l'étape précèdente.

Il faut remarquer que nous ne définirons pas de manière générale la protection d'une proposition sérielle [A-6.0.2 d)] : au niveau du langage pivot nous supposons que tous les identificateurs (resp. indicateurs, opérateurs) définis dans un programme sont différents. Cependant cette hypothèse ne suffit pas à

traiter le cas des procédures, notamment celui des procédures récursives où, lors de chaque appel, tout identificateur local au corps de la procédure doit être différent de tout autre identificateur déjà déclaré. Nous traitons ce problème en [6.3.11.2].

iii) Le "remplacement" (dans l'ordre textuel) des symboles $\sim_i$ dans les n déclarations d'identité qui se trouvent en tête de la routine par les paramètres effectifs correspondants. C'est ce qui est étudié en [6.3.11.3] .

iv) L'élaboration de la proposition fermée obtenue à la suite des étapes précèdentes. Nous définirons en [6.3.11.4] le double système associé à l'appel A_p .

6.3.11.2 Traitement des identificateurs locaux à la routine :

A la suite de l'élaboration $\overset{\vee}{G}$ du primaire G, l'information courante (i.e. celle qui est l'image de l'information de départ par la modification interprétation du calcul se terminant par $\overset{\vee}{G}$) contient un théorème de la forme *poss* G ≡ F' (avec les notations de [6.3.10]) où F' est une routine.
F' est une proposition fermée contenant, outre les $\sim_i$ (qui sont traités en [6.3.11.3]), un certain nombre d'identificateurs locaux parmi lesquels les paramètres formels. Pour exprimer qu'à chaque appel A_p correspond de nouveaux identificateurs locaux, on transforme ces identificateurs à l'aide du symbole fonctionnel *sub*.

a) Le schéma fonctionnel associé à *poss* A_p se déduira du schéma *poss* F' en "substituant" à tout identificateur local x, l'élément *sub* A_p x de IDENT.
On exige alors :

- que le schéma fonctionnel *sub* A_p x soit différent (formellement) de tout autre élément de IDENT si x est local
- que *sub* A_p x soit égal (formellement) à x si x est une constante ou un identificateur global.

Ce que l'on exprime par les axiomes suivants, dans lesquels $L_o^{F'}$ représente l'ensemble des identificateurs locaux à F' :

$$SH_{8.1} = \{\textit{poss}\ G \equiv F' \supset \neg \textit{sub}\ G(E_1,\ldots,E_n)\ x \equiv y \mid F', G \in L_o^{ph} \ ; \ x \in L_o^{F'} \ ; \ y \in \text{IDENT}, \ y \neq \text{sub}\ G(E_1,\ldots,E_n)\ x \ ; \ E_i \in \text{IDENT}\}$$

$$SH_{8.2} = \{\textit{poss}\ G \equiv F' \supset \textit{sub}\ G(E_1,\ldots,E_n)\ x \equiv x \mid x \in \text{IDENT} - L_o^{F'} \cup \{\sim_i \mid 1 \leqslant i \leqslant n\} \ ; \ F', G \in L_o^{ph}\}$$

(le cas de *sub* $G(E_1,\ldots,E_n)$ $\sim_i$ est traité en [6.3.11.3]).

On exige également l'injectivité de *sub* :

$$SH_{8.3} = \{sub\ A_p\ x \equiv sub\ B_p\ y \supset x \equiv y \wedge A_p \equiv B_p \mid x,y \in L_o^{id}\ ;\ A_p,B_p \in L_o^{ph}\}\ .$$

Il faut également définir *sub* A_p u pour tout schéma fonctionnel u ; en particulier on souhaite que l'axiome *poss* $A_p \equiv$ *sub* A_p *poss* F' associe à *poss* A_p le schéma déduit de *poss* F' en substituant *sub* A_p x à tout identificateur ; c'est le rôle des axiomes :

$$SH_{8.4} = \{sub\ A_p\ f\ u_1 \ldots u_n \equiv f\ sub\ A_p\ u_1 \ldots\ sub\ A_p\ u_n \mid f\ u_1 \ldots u_n \in S\ ,\ A_p \in L_o^{ph}\}\ .$$

Ce schéma n'a d'interêt que pour des schémas fonctionnels contenant des identificateurs ; plus précisément aucun schéma tel que *structure reel entier*, $x_1 x_2$ par exemple, ne pourra être précédé de *sub* A_p comme nous le verrons en [6.3.11.4] .

Enfin un identificateur local à une routine peut apparaître dans un déclareur de valeur multiple, mais un tel déclareur à été considéré comme un symbole 0-aire (ce n'est pas un schéma fonctionnel comportant l'identificateur) aussi sommes-nous conduit à traiter séparément les déclareurs :
Par déclareur Algol 68_o nous entendrons déclareur Algol 68 (en fait son arbre syntaxique) où tout objet déduit de ces déclareurs en faisant précéder les bornes éventuelles de symboles de la forme *sub* A_p (où A_p est un appel).
On introduit alors les schémas d'axiomes :

$$SH_{8.5} = \{sub\ A_p\ \underline{rep}\ D \equiv \underline{rep}\ sub\ A_p\ D \mid D \in INDIC\ ,\ A_p \in L_o^{ph}\}$$

$$SH_{8.6} = \{sub\ A_p\ \underline{struct}\ (D_1x_1,\ldots,D_nx_n) \equiv \underline{struct}(sub\ A_p\ D_1x_1,\ldots,sub\ A_p\ D_nx_n) \mid D_1,D_2,\ldots,D_n \in INDIC\ ;\ A_p \in L_o^{ph}\ ;\ x_1,\ldots,x_n \in L_o^{id}\}$$

$$SH_{8.7} = \{sub\ A_p\ [A_1\Gamma_1 : B_1\Delta_1,\ldots,A_n\Gamma_n : B_n\Delta_n]\ D \equiv [sub\ A_p\ A_1\Gamma_1 : sub\ A_p\ B_1\Delta_1,\ldots,sub\ A_p\ B_n\Gamma_n : sub\ A_p\ B_n\Delta_n]\ sub\ A_p\ D \mid [A_1\Gamma_1 : B_1\Delta_1,\ldots,A_n\Gamma_n : B_n\Delta_n]\ D \in INDIC\ ,\ A_p \in L_o^{ph}\}$$

$$SH_{8.8} = \{sub\ A_p\ D \equiv D \mid D \in L_o^{prim} \cup \overline{L_o^{nprim}}\ ,\ A_p \in L_o^{ph}\}$$

où $\overline{L_o^{nprim}}$ est l'ensemble des déclareurs non primitifs qui ne commencent pas par

struct, *rep* ou [. Ce dernier schéma est dû au fait que tous ces déclareurs ne contiennent pas de borne [A-7.11].

Remarque : Ces différents schémas, joints à $SH_{8.4}$ permettent de définir *poss* a_i D et *poss* b_i D ($1 \leqslant i \leqslant n$) pour tout déclareur de valeur multiple D de dimension n (schémas fonctionnels utilisés lors des déclareurs d'identité [6.2.3.5]).

b) Le schéma de calculs associé à A_p contiendra un schéma de calculs déduit de $\tilde{F'}$ en substituant *sub* A_p x à tout identificateur x (qui apparaît dans certains paramètres des schémas de modifications élémentaires composant $\tilde{F'}$). Pour cette raison on définit une fonction de base $\mathcal{C}_{A_p}$ du système à point fixe $\mathcal{M}(P)$ (si P est le programme contenant F') par :

$\mathcal{C}_{A_p}(\Pi(u_1,\ldots,u_n)) = \Pi(sub\ A_p\ u_1,\ldots,sub\ A_p\ u_n)$ pour tout schéma de modifications élémentaires Π à un paramètre $u_1,\ldots,u_n$.

(Rappelons [3.3.3] que si c_1 et c_2 sont deux schémas de calculs,

$$\mathcal{C}_{A_p}(c_1.c_2) = \mathcal{C}_{A_p}(c_1)\ .\ \mathcal{C}_{A_p}(c_2) \quad \text{et} \quad \mathcal{C}_{A_p}(c_1 \cup c_2) = \mathcal{C}_{A_p}(c_1) \cup \mathcal{C}_{A_p}(c_2))\ .$$

De plus dans le cas d'un déclareur de valeur multiple il peut être nécessaire d'introduire les schémas de calculs $\widetilde{sub\ A_p\ A_i}$ ou $\widetilde{sub\ A_p\ B_i}$ (lors de l'élaboration d'une déclaration d'identité ou d'un générateur. Il est alors nécessaire d'adjoindre au système de calculs d'un programme P quelconque les équations :

$$\widetilde{sub\ A_p\ E} = \mathcal{C}_{A_p}(\tilde{E}) \quad \text{pour toute phrase E et tout appel } A_p \text{ de P}\ .$$

6.3.11.3 Traitement des paramètres effectifs :

Il s'agit de remplacer les symboles $\sim_i$ de F' par les paramètres effectifs correspondants E_i . On utilise encore le symbole *sub* :

$$SH_{8.9} = \{sub\ G(E_1,\ldots,E_n)\sim_i \equiv E_i \mid 1 \leqslant i \leqslant n\ ;\ G(E_1,\ldots,E_n) \in L_o^{ph}\}\ .$$

Ce schéma d'axiomes et la définition de $\mathcal{C}_{A_p}$ permettent de "remplacer" $\sim_i$ par E_i dans les paramètres des schémas de modifications.
De plus $\tilde{\sim}_i$ apparaît dans le schéma de calculs associé à F', il doit être remplacé par E_i dans les schéma A_p : c'est ce que permet $\tilde{\Psi}$ [6.1.2] . Plus précisément :

$$\mathcal{S}(\sim_i)\begin{cases}\mathcal{M}(\sim_i) \ \{\ \tilde{\sim}_i = \tilde{\Psi}(\sim_i)\\ \mathcal{V}(\sim_i) = \emptyset\end{cases}$$

Alors comme $\tilde{\tilde{F}}'$ contient (au début du schéma de calcul associé à la déclaration $D_i z_i = \sim_i$) $\tau_{A_p}(\tilde{\tilde{F}}')$ contient $\tau_{A_p}(\tilde{\sim}_i) = \tilde{\psi}(sub\ A_p \sim_i) = \tilde{\tilde{E}}_i$ (à l'endroit correspondant).

6.3.11.4 Systèmes associés à un appel :

On regroupe les résultats précédents dans la définition de $\mathcal{S}(A_p)$ avec $A_p : G(E_1,\ldots,E_n)$:

$$\mathcal{S}(A_p)\begin{cases}\mathcal{M}(A_p)\begin{cases}\tilde{\tilde{A}}_p = \tilde{\tilde{G}} \cdot \tau_{A_p}(\delta.\tilde{\psi}(poss\ G))\\ \mathcal{M}(G)\\ \mathcal{M}(E_i) \quad i = 1,\ldots,n\end{cases}\\ \mathcal{V}(A)\ \{\ poss\ A \equiv sub\ A_p\ poss\ poss\ G\end{cases}$$

(Remarquons que $\tilde{\psi}(poss\ G) = \tilde{\psi}(F') = \tilde{\tilde{F}}'$ si F' est la routine possédée par G). δ est introduit pour rendre compte de certains calculs infinis [3.3.1 b)].

6.4 Systèmes associés aux phrases composées :

6.4.1 Introduction :

En plus des déclarations et propositions unitaires, Algol 68_o définit des procédés généraux de construction, ce sont les phrases composées. Une telle phrase est [A-6]

- soit une déclaration collatérale
- soit une proposition sérielle
- soit une proposition fermée
- soit une proposition conditionnelle
- soit une proposition collatérale.

En particulier, une proposition collatérale peut être :

i) d'un certain mode [A-6.2.1 c) à h)], elle permet alors de définir une valeur multiple ou une valeur structurée

ii) neutre, nous distinguerons alors deux cas [A-6.2.1 b)] :

- proposition parallèle (ce sont celles qui sont précédées du symbole parallèle)
- proposition non contrôlée (elle ne contient aucun sémaphore).

Dans la suite du paragraphe on associe un double système à chaque type de phrase composée.

6.4.2 Déclarations collatérales :

Une déclaration collatérale D est de la forme $D_1, D_2, \ldots, D_n$ où chaque D_i est une déclaration unitaire [A-6.2.1 a)] ; [6.2].

La sémantique de D est définie en [A-6.2.2 a) et b)] ; elle s'exprime simplement dans notre formalisation par :

$$\mathcal{S}(D)\begin{cases}\mathcal{M}(D)\begin{cases}\tilde{\tilde{D}} = m(\tilde{\tilde{D}}_1, \tilde{\tilde{D}}_2, \ldots, \tilde{\tilde{D}}_n)\\ \mathcal{M}(\tilde{\tilde{D}}_i) \quad i = 1,n\end{cases}\\ \mathcal{V}(D) = \emptyset\end{cases}$$

$\mathcal{V}(D) = \emptyset$: aucun axiome particulier n'est associé à la déclaration collatérale.

6.4.3 Propositions sérielles :

Une proposition sérielle peut être réduite à une proposition unitaire, ce cas est donc traité en [6.3], sinon une proposition unitaire Algol 68_o E est de la forme

$$E_1\ ;\ E_2\ ;\ldots;\ E_n\,|p.$$

où chaque E_i est soit une déclaration (simple ou collatérale), soit une proposition unitaire. Rappelons en effet que nous excluons toute instruction de saut et toute étiquette du langage pivot.

D'autre part aucun identificateur, opérateur ou indicateur de mode ne possède (resp. n'indique) deux exemplaires de valeurs (resp. deux exemplaires de routines, deux modes) différents dans un même programme Algol 68_o ; il est alors inutile d'exprimer ici [A-6.1.2 a)] concernant la protection de la proposition sérielle.

Enfin E est une région, il lui est donc associée une certaine portée p (par exemple, c'est la portée de tout nom défini par un générateur apparaissant dans E mais non contenu dans un sous noyau de E). Il faut distinguer p de la portée de l'exemplaire de valeur (s'il existe) possédé par E.

On formalise [6.1.2] (et en particulier e)) par le double système :

$$\mathcal{S}(E)\begin{cases}\mathcal{M}(E)\begin{cases}\tilde{\tilde{E}} = \tilde{\tilde{E}}_1 \,.\, \tilde{\tilde{E}}_2 \,.\, \ldots \,.\, \tilde{\tilde{E}}_n\\ \mathcal{M}(E_i) \quad i = 1,\ldots,n\end{cases}\\ \mathcal{V}(E)\ \{\ ppq\ p\ portée\ E_n \equiv vrai \supset poss\ E \equiv poss\ E_n\end{cases}$$

6.4.4 Propositions fermées :

a) Soit P la proposition fermée [A-6.3] d E f où d (resp. f) est soit un symbole ouvrir, soit un symbole début (resp. fermer ou fin) et E est une proposition sérielle.

La sémantique de P [A-6.3.2] est traduite par :

$$\mathcal{S}(P)\begin{cases}\mathcal{M}(P)\begin{cases}\tilde{\tilde{P}} = \tilde{\tilde{E}}\\ \mathcal{M}(E)\end{cases}\\ \mathcal{V}(P)\begin{cases}poss\ P \equiv poss\ E\\ \mathcal{V}(E)\end{cases}\end{cases}$$

Si *poss* E n'est pas "défini" par un axiome [6.4.3] alors *poss* P ne sera pas "défini".

b) Dans le cas où P est un programme, $\mathcal{M}(P)$ contient l'équation définissant le schéma de calculs $\tilde{\psi}$ [6.1.2] . De plus le schéma de calculs $\tilde{\tilde{P}}$ doit éventuellement être tronqué pour rendre compte des appels de procédures qui remplacent les sauts (c'est-à-dire ceux qui se terminent par *stop*) [3.3.2.2 f)]. Ainsi :

$$\mathcal{M}(P)\begin{cases}\tilde{\tilde{P}} = \zeta(\tilde{\tilde{E}})\\ \tilde{\psi} = \bigcup_{E \in L_o^{ph} \cup L_o^{nprim}} \lambda u\ .\ j(u,E)\ .\ \tilde{\tilde{E}}\\ \mathcal{M}(E)\end{cases}$$

6.4.5 Propositions conditionnelles :

Soit P la proposition conditionnelle [A-6.4] : *si* B *alors* E_1 *sinon* E_2 *fsi* où *si* (resp. *alors*, *sinon*, *fsi*) est un symbole si (resp. alors, sinon, fsi) ; B, E_1, E_2 sont des propositions sérielles.
L'élaboration de P [A-6.4.2] est traduite par :

$$\mathcal{S}(P)\begin{cases}\mathcal{M}(C)\begin{cases}\tilde{\tilde{P}} = \tilde{\tilde{B}}\ .\ (j\ valeur\ possB, vrai).\tilde{\tilde{E}}_1 \cup \bar{j}(valeur\ possB, vrai).\tilde{\tilde{E}}_2)\\ \mathcal{M}(B)\\ \mathcal{M}(E_1)\\ \mathcal{M}(E_2)\end{cases}\\ \mathcal{V}(C)\ \{\ valeur\ possP \equiv cond\ valeur\ possB\ vrai\ valeur\ possE_1\ valeur\ possE_2\end{cases}$$

6.4.6 Propositions collatérales de mode μ :

Soit P la proposition collatérale $(E_1, E_2, \ldots, E_n)$ de mode μ ($n \geqslant 1$) . μ est donc un mode de valeur structurée ou multiple [A-6.2.1] déterminé par le contexte, c'est-à-dire introduit par le langage pivot dans notre formalisation.

Il est clair que $\mathcal{V}(P)$ dépend de μ. Notons alors X(P,μ) l'ensemble des axiomes caractérisant l'accès nouveau P :

$$\mathcal{S}(P)\begin{cases}\mathcal{M}(P)\begin{cases}\tilde{\tilde{P}} = \mathop{\mathrm{m}}_{i=1}^{n}(\tilde{\tilde{E}}_i)\\ \mathcal{M}(E_i) \quad i=1,n\end{cases}\\ \mathcal{V}(P)\ \{\ X(P,\mu)\end{cases}$$

Il faut alors définir X(P,μ) suivant le mode μ :

6.4.6.1 μ est un mode de valeur structurée :

Alors μ est de la forme : *structure* $\underset{x_1 \ldots x_n}{\mu_1 \ldots \mu_n}$ et donc :

$$X(P,\mu) = \{sousflex\ rep\ poss\ E_i \equiv faux \mid 1 \leqslant i \leqslant n\} \cup$$
$$\{valeur\ ch_{x_i}\ poss\ P \equiv valeur\ poss\ E_i \mid 1 \leqslant i \leqslant n\} \cup \{\text{valeur } v \equiv \text{valeur poss } P \supset$$
$$\neg v \equiv poss\ P \mid v \in EXVAL,\ v \neq poss\ P\} \cup \{mode\ poss\ P \equiv \mu\}\ .$$

Remarques :

1) le premier ensemble d'axiomes traduit [A-6.2.2 c) pas 3)]

2) les deuxième, troisième et quatrième ensembles d'axiomes traduisent [A-6.2.2 c) pas 4 et 7] .

Enfin la portée de P est définie par $SH_{6.13}$.

6.4.6.2 μ est un mode de valeur multiple de dimension 1 :

μ est de la forme : rang μ' . Alors :

$X(P,\mu) = \{$*sousflex rep poss* $E_i \equiv$ *faux* $\mid 1 \leqslant i \leqslant n\} \cup$

$\{$*valeur elem*$_1$ *poss* P i $\equiv$ *valeur poss* $E_i \mid 1 \leqslant i \leqslant n\} \cup$

$\{$*valeur* v $\equiv$ *valeur poss* P $\supset \neg$v $\equiv$ *poss* P $\mid$ v $\in$ EXVAL, v $\neq$ *poss* P$\} \cup$

$\{a_1$ *poss* P$\equiv 1$, b_1 *poss* P$\equiv n$, s_1 *poss* P$\equiv 1$, t_1 *poss* P$\equiv 1$, *dim poss* P$\equiv 1\} \cup$

$\{$*mode poss* P $\equiv \mu\}$.

On exprime ainsi [A-6.2.2 c) pas 3, pas 5, pas 7] .

6.4.6.3 μ est un mode de valeur multiple de dimension supérieure à 1 :

Soit μ le mode $rang^{(k+1)}$ μ' où $k \geqslant 1$; nous noterons k' le nombre k+1 .

$X(P,\mu) = \{$*sousflex rep poss* $E_j \equiv$ *faux* $\mid 1 \leqslant j \leqslant n\} \cup$

$\{$*dim poss* $E_j \equiv k$, a_i *poss* $E_j \equiv a_i$ *poss* E_ℓ, b_i *poss* $E_j \equiv b_i$ *poss* $E_\ell \mid$

$1 \leqslant i \leqslant k$, $1 \leqslant j \leqslant \ell \leqslant n\} \cup \{ \bigwedge_{1 \leqslant i \leqslant k} (a_i$ *poss* $E_1 \equiv u_i \wedge b_i$ *poss* $E_1 \equiv v_i) \supset$

valeur elem$_{k'}$ *poss* P $i_o i_1 \ldots i_k \equiv$ *valeur elem*$_k$ *poss* $E_{i_o\ i_1 \ldots i_k} \mid$

$u_j \leqslant i_j \leqslant v_j$ pour $1 \leqslant j \leqslant k$; $1 \leqslant i_o \leqslant n\} \cup$

$\{$*valeur* v $\equiv$ *valeur poss* P $\supset \neg$v $\equiv$ *poss* P $\mid$ v $\in$ EXVAL, v $\neq$ *poss* P$\} \cup$

$\{$*dim poss* P $\equiv$ k' , a_1 *poss* P $\equiv$ 1 , b_1 *poss* P $\equiv n\} \cup$

$\{a_i$ *poss* P $\equiv a_{i-1}$ *poss* E_1 , b_i *poss* P $\equiv b_{i-1}$ *poss* $E_1 \mid 2 \leqslant i \leqslant k'\} \cup$

$\{s_i$ *poss* P $\equiv$ 1 , t_i *poss* P $\equiv$ 1 $\mid 1 \leqslant i \leqslant k'\} \cup$

$\{$*mode poss* P $\equiv \mu\}$.

Le deuxième ensemble d'axiomes précise que les bornes inférieures (resp. supérieures) des différentes valeurs multiples *poss* E_i doivent être égales [A-6.2.2 c) pas 6] .

Les troisième et quatrième ensembles d'axiomes définissent les éléments du nouvel exemplaire *poss* P ; les trois suivants caractérisent le descripteur.

6.4.7 Propositions collatérales de genre neutre, non contrôlées :

C'est une production terminale de [A-6.2.1 b)] "unité du genre neutre en position forte en liste propre en paquet".

Soit P : $(E_1, \ldots, E_n)$ une telle proposition, il est clair que $\mathcal{V}(P) = \emptyset$. D'autre part l'élaboration de P consiste en l'élaboration collatérale de ses unités constituantes ; aussi :

$$\mathcal{S}(P) \begin{cases} \mathcal{M}(P) \begin{cases} \tilde{P} = \underset{i=1}{\overset{n}{\mathfrak{m}}} (\tilde{E}_i) \\ \mathcal{M}(E_i) \quad i=1,\ldots,n \end{cases} \\ \mathcal{V}(P) = \emptyset \end{cases}$$

6.4.8 Propositions collatérales contrôlées :

Nous ne les traitons pas dans cette formalisation.

7

Conclusion

7. CONCLUSION

L'application de notre méthode de formalisation à la définition d'Algol 68 nous a conduit à étudier avec précision le mécanisme de description de ce langage. Ainsi, par exemple, le rôle de la notion d'exemplaire de valeur a été mis en évidence lors de l'étude de l'affectation [6.3.2.6 remarque 1].
Il faut cependant remarquer que le fait de considérer un langage défini par une autre méthode (celle du rapport [A] en l'occurence) alourdit la formalisation : La description de l'élaboration des bornices d'un déclareur de valeur multiple, par exemple, pourrait certainement être simplifiée à condition de ne pas suivre strictement [A] (voir la remarque [6.2.2.1 g)]).

Il conviendrait de compléter ce travail au niveau de la description d'Algol 68 :

- traitement des sémaphores et des propositions contrôlées
- traitement des formats
- traitement des mécanismes d'entrée/sortie. Pour cela il serait nécessaire d'introduire dans la structure d'information des accès permettant de formaliser les notions de canaux, fichiers, volumes [A-10.5.1]. Une information initiale contiendrait alors les données nécessaires à l'élaboration du programme.

Cette étude, par le double fait qu'elle introduit un nouveau formalisme et qu'elle s'applique à des langages évolués peut se situer à la frontière entre une certaine informatique théorique (étude mathématique de langages simples définis par des équations) et une conception plus pratique (problèmes de compilation par exemple). Il semble alors naturel de la prolonger dans ces deux directions :

1) d'un point de vue théorique :

   - Il serait nécessaire de "justifier" chaque modification élémentaire en démontrant qu'elle conserve la consistance de certaines classes d'informations.
   - Il faudrait étudier de manière précise l'information initiale et plus précisément l'ensemble des théorèmes de la structure d'information : peut-on démontrer sa consistance ? La réponse à cette question permettrait de "justifier" ou de "refuter" le rapport [A].

- Il conviendrait de perfectionner nos outils, en particulier la notion de calculs qui pourrait, par exemple, être axiomatisée. La description d'Algol 68 a montré la puissance de ce formalisme ; pour l'étudier plus profondément il serait utile de définir un langage plus simple contenant essentiellement les notions d'affectation, de déclaration d'identité et de procédure. L'intérêt de notre méthode provient de l'utilisation de deux niveaux formalisant deux réalités informatiques complémentaires :

  - niveau des accès
  - niveau des calculs

  ce qui permet de rendre compte des divers types de propositions apparaissant dans les langages de programmation usuels. On peut ainsi espérer utiliser ces outils pour construire des méthodes de preuves de correction, de terminaison, ou pour définir des notions d'équivalences concernant des langages évolués non simples.

Un point de départ d'une telle étude pourrait consister à comparer notre point de vue avec celui de Bakker et de Roever qui définissent des opérations sur des ensembles de relations semblables à certaines de nos opérations sur les ensembles de calculs ([S6]), et qui obtiennent ainsi un certain nombre de résultats sur l'équivalence et les preuves de programmes de langages simples.

- Nous avons déjà signalé [5.1] que la définition sous forme de schémas fonctionnels des programmes du langage pivot permettrait d'unifier la description d'un langage.
  Elle pourrait également faciliter l'étude théorique évoquée ci-dessus.

2) d'un point de vue plus pratique on peut espérer que ce travail fournisse des outils utiles à la conception d'une implémentation.
En particulier la notion de représentation d'une structure par un autre [SI7] permet d'unifier le langage de description de la sémantique et de l'implémentation. Ce serait certainement une manière d'aborder le problème de la correction d'un compilateur.

8

Annexe

Annexe 1.

Différents symboles fonctionnels n-aires (n ⩾ 1) structure Algol 68

Symbole	Arité	Rôle
a_i $(i \geqslant 1)$	1	définit la $i^{ème}$ borne inférieure d'un exemplaire de valeur multiple ou d'un déclareur de valeur multiple
b_i $(i \geqslant 1)$	1	définit la $i^{ème}$ borne supérieure d'un exemplaire de valeur multiple ou d'un déclareur de valeur multiple
ch_x $(x \in L_o^{id})$	1	définit le sélecteur associé au champ x d'un exemplaire de valeur structurée
derep	1	permet de traiter la modification Algol 68 dereperer
desc	1	utilisé dans la définition des schémas interprétables comme des portées
dim	1	associe sa dimension à tout exemplaire de valeur multiple. A tout déclareur de valeur multiple il associe la dimension de la valeur multiple spécifiée
div	2	représente la division
elarg	1	permet de traiter la modification Algol 68 élargir
$elem_n$ $(n \geqslant 1)$	n+1	permet de sélectionner les éléments d'une valeur multiple de dimension n
$equi_1$	1	associe à un entier un réel de même numéro de longueur
$equi_2$	1	associe à un entier (resp. réel) de longueur n l'entier (resp. le réel) équivalent de longueur n+1
$equi_3$	1	associe à tout caractère l'entier équivalent

Symbole	Arité	Rôle
$\overline{equi_1}$	1	symbole "réciproque" de $équi_1$
$\overline{equi_2}$	1	symbole "réciproque" de $equi_2$
indique	1	associe à un déclareur de mode le mode spécifié
inf	2	permet de représenter la relation "être plus petit que" sur les nombres
long	1	associe au mode μ le mode "long μ"
mode	1	associe à tout exemplaire de valeur son mode
moins	2	représente la soustraction
mult	2	représente la multiplication
portée	1	définit la portée d'une valeur
pose	1	formalise la fonction possède qui associe à tout objet externe un exemplaire de valeur
ppq	2	(pour plus petit que) permet de définir une relation d'ordre sur l'ensemble des portées
$procn_n$ $(n \geqslant 0)$	n	permet de définir les modes procédure à n paramètres et résultat neutre
$procr_n$ $(n \geqslant 1)$	n	permet de définir les modes procédure à $n-1$ paramètres et résultat d'un certain mode
rang	1	associe au mode μ le mode "rang de μ"
rep	1	formalise la fonction repère (qui associe à tout exemplaire de nom l'exemplaire de valeur repéré)

Symbole	Arité	Rôle
repère	1	associe au mode μ le mode "repère de μ"
s_i $(i \geqslant 1)$	1	définit le $i^{ème}$ état inférieur d'un exemplaire de valeur multiple ou caractérise "l'état" (fixe, flexible, adlib) de la $i^{ème}$ borne inférieure d'un déclareur de valeur multiple
sousflex	1	caractérise les composantes d'une valeur multiple flexible
$structure\ {}^{x_1 \ldots x_n}\ (x_i \in L_o^{id})$	n	permet de définir les modes structure à n champs de sélecteurs $x_1, \ldots, x_n$
sub	2	permet de traiter les appels de procédures
succ	1	utilisé dans la définition des schémas fonctionnels interprétables comme des portées
t_i $(i \geqslant 1)$	1	définit le $i^{ème}$ état supérieur d'un exemplaire de valeur multiple ou caractérise "l'état" (fixe, flexible, adlib) de la $i^{ème}$ borne inférieure d'un déclareur de valeur multiple
$tranche_n$ $(n \geqslant 1)$	$4n+1$	définit les tranches d'une valeur multiple de dimension n
$union_n$ $(n \geqslant 1)$	n	permet de définir les modes union de plus, par convention, $union_1\ \mu \equiv \mu$ pour tout mode μ
valeur	1	associe sa valeur à un exemplaire de valeur

Annexe 2 :

Différents schémas de modifications élémentaires de la structure Algol 68 :

- coh [6.3.2.3] : schéma à deux paramètres qui permet d'exprimer [A-8.3.1.2 c) pas 1] certaines conditions sur les valeurs des propositions apparaissant en partie droite et gauche d'une affectation.

- créa [6.3.7.4] : schéma à deux paramètres qui permet d'exprimer [A-7.1.2 pas 6 et 7] la création d'un nouvel exemplaire de valeur multiple par un générateur.

- fau [6.3.4.3 d)] : schéma à un paramètre qui permet d'associer la valeur faux à une relation de conformité.

- ident [6.2.3.3] : schéma à deux paramètres qui permet de définir [A-7.4.2 pas 7].

- j (resp. $\bar{j}$) [4.2.2] : schéma à deux paramètres qui permet de réduire le domaine de définition de certains modifications (interprétations de calculs).

- opéra [6.2.4] : schéma à deux paramètres qui traduit les déclarations d'opérations [A-7.5.1].

- suppl [6.3.2.6] : schéma à deux paramètres qui permet d'exprimer l'action de supplanter [A-8.3.1.2 a) et b)] dans notre formalisation.

- test [6.2.3.6] : schéma à trois paramètres qui exprime les conditions portant sur les bornes d'un déclareur de valeur multiple lors d'une déclaration d'identité [A-7.4.2 pas 5 et 6].

- traitmul [6.3.2.5] : schéma à deux paramètres qui exprime le traitement relatif à une valeur multiple lors d'une affectation [A-8.3.1.2 c)] pas 3 et 4.

- ver [6.3.4.3 d)] : schéma à un paramètre qui permet d'associer la valeur vrai à une relation de conformité.

Réfèrences

ALGOL 68

[A1] P. BRANQUART, J. LEWI, M. SINTZOFF and P.L. WODON : The Composition of semantics in Algol 68. CACM, 1971.

[A2] GROUPE ALGOL DE L'AFCET : Définition du langage algorithmique Algol 68 (traduction). Hermann, 1972.

[A3] GROUPE ALGOL DE L'AFCET : Mannuel Algol 68. Diffusion restreinte.

[A4] GROUPE ALGOL DE L'AFCET : Mannuel Algol 68 révisé. Hermann A paraitre.

[A5] C. PAIR : Concerning the syntax of Algol 68. Algol bulletin 31, 1970.

[A6] M. SINTZOFF : Introduction à la description d'Algol 68. Rev. Franç. d'informatique et de Rech. Op. 1969, 3.16.

[A7] A. VAN WIJNGAARDEN, B.J MAILLOUX, J.E.L. PECK, C.H.A KOSTERE : Report on the algorithmic language Algol 68. Second Printing by the Mathematisch Centrum, Amsterdam, M.R. 101, 1969.

A8 Rapport sur le langage algorithmique Algol 68 révisé. A paraitre.

CALCULS

[C1] H. BEKIC : Toward a mathematical theory of processes. Technical report 25.125, I.B.M Laboratory Vienna, 1971.

[C2] A. BLIKLE : An Algebraic Approach to semantics of Programs. Advanced course on semantics of Programming Languages, University of Saarbrücken, 1974.

[C3] C. PAIR : Calculs, ensembles de calculs, équivalence de programmes . Convegno Informatica teorica Rome, 1973.

LOGIQUE

[L1] A. CHURCH : The Calculi of Lambda conversion. Ann. of Math-Studies, Princeton, 1941, 6.

[L2] H.B. CURRY, R. FEYS : Combinatory logic. North Holland Amsterdam, 1958.

[L3] J.R. HINDLEY, B. LERCHER and J.P. SELDIN : Introduction to combinatory logic. London mathematical society. Cambridge University Press, 1972, Lecture note séries 7.

[L4] S.C. KLEENE : Introduction to meta mathematics, North Holland, Amsterdam, 1952.

[L5] S.C. KLEENE : Logique mathématique. Armand Colin Coll. U 1971.

[L6] G. KREISEL, J.L. KRIVINE : Eléments de logique mathématique. Théorie des modèles Dunod, 1966.

[L7] A.A. MARKOV : Theory of algorithms. Moscou : USSR Academy of Sciences, 1954. (traduit en anglais par Israeli Program for Scientific Translations, Jerusalem 1962, volume 42).

[L8] D. PONASSE : Logique mathématique. O.C.D.L. Paris

[L9] J.R. SHOENFIELD : Mathematical Logic. Addison-Wesley, 1967.

[L10] A.M. TURING : On computable Numbers, with an application to the Entscheidunegsproblem. Proc. London Math. Soc, ser 2 vol 42, 1936-37, 115-154.

POINT FIXE

[PF1] A. BLIKLE : Equational languages. Information and control 21, 1972, 134-147.

[PF2] A. BLIKLE : Equation in nets. Computer oriented Lattices CC PAS Reports 99. Computation Centre Polish Academy of Sciences, Warsaw 1973.

[PF3] B. COURCELLE, G. KAHN, J. VUILLEMIN : Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. Laboria 1973, rapport de recherche numéro 37.

[PF4] Z. MANNA, J. VUILLEMIN : Fixpoint Approach to the theory of computation symposium automata, Languages and Programming. IRIA 1972. (Ed. M. NIVAT) North Holland. American Elsevier, 273-291.

[PF5] Z. MANNA, S. NESS, J. VUILLEMIN : Inductive Methods for Proving Properties of Programs. CACM, 1973, 16, 491-502.

[PF6] D. PARK : Fixpoint induction and Proofs of Program Properties. Machine Intelligence 5, Edinbourgh University Press, 1969.

[PF7] D. SCOTT : Data Types as Lattices. Cours de l'Ecole d'été d'Amsterdam, 1972.

[PF8] D. SCOTT : The Lattices of Flow Diagrams. Symposium on semantics of algorithmic Languages. Springer Verlay 1971, 188, 311-366.

[PF9] D. SCOTT : Continuous Lattices. Oxford Mono PRG-7, Oxford University, 1972.

[PF10] J. VUILLEMIN : Proof Techniques for recursives Programs. Standford Computer Science Department. Ph.D.Thesis, 1973.

SEMANTIQUE DES LANGAGES

[S1] K. ALBER and P. OLIVIA : Translation of PL/1 into Abstract syntax. Technical Report TR 25.086, IBM. Laboratory Vienna, 1968.

[S2] K. ALBER, P. OLIVIA and G. URSCHLER : Concrete syntax of PL/1. Technical Report TR 25.084, IBM. Laboratory Vienna, 1968.

[S3] J.W. DE BAKKER : Formal Definition of Programming Languages. Mathematical Center Tracts. Mathematich Centrum Amsterdam, 1967, volume 16.

[S4] J.W. DE BAKKER : Axiomatics of simple assignement statements. Report MR 94. Mathematich Centrum Amsterdam, 1968.

[S5] J.W. DE BAKKER : Semantics of Programming Languages. Advances information systems science (Tou, J.T. ed). Plenum Press, 1969, 2, 173-227.

[S6] J.W. DE BAKKER, W.P. DE ROEVER : A calculus for recursive program schemes. Proceedings of IRIA Collogium North Holland, 1972.

[S7] J.W. DE BAKKER : Fixed point in programming theory. Lecture notes for the advanced course on the foundations of computer science. Amsterdam, 1974.

[S8] A. BLICKLE : An Algebraic Approach to mathematical theory of programs. CCPAS reports 119. WARSZAWA Poland, 1973.

[S9] A. BLICKLE : An Algebraic Approach to semantics of programs Advanced course on semantics of Programming Languages, Saarbrücken, 1974.

[S10] C. BÖHM : CUCH as a Formal and description language. Formal Languages description Languages, Proc. IFIP 1964 (Steel ed.). North Holland, Amsterdam 1966, 179-197.

[S11] C. BÖHM, W. GROSS : Introduction to the CUCH. "Automata theory" (E.R. Caianiello ed.). Academic Press New-York, 1966, 35-65.

[S12] J.M. CADIOU : Recursive definition of partial fonctions and their computation. Ph. D. Th. Computer science Dept. Standford U. , 1972.

[S13] A. CARACCIOLO DI FORINO : Generalized Markov algorithms and automata . Automata theory (Caianiello ed.). Academic Press New-York, 1966, 115-130.

[S14] A. CARACCIOLO DI FORINO : String Processing Languages and generalized Markov Algorithms. Symbol Manipulation languages and techniques, Proc IFIP 1966 (D.G. Bobrow ed.) North Holland Amsterdam, 1968, 191-206.

[S15] D.C. COOPER : Programm schemes Equivalences and second order Logic. Machine Intelligence (Meltzer and Michies ed.). Edinburg Univ. Press, 1969, 4, 3-15.

[S16] B. COURCELLE, J. VUILLEMIN : Semantics and Axiomatics of a simple Recursive Language. IRIA Laboria, Rapport de recherche, 1974.

[S17] E. DIJKSTRA : GØTO statement considered Harmful. CACM vol 11 numéro 3, 147-148.

[S18] C.C. ELGOT : Machines species and theer computation Languages. Formal Language description Languages, Proc. IFIP 1964 (Steel ed.). North Holland, Amsterdam, 1966, 160-178.

[S19] M. FLECK and E. NEUHOLD : Formal definition of the PL/1 Compile time Facilities. Technical Report T R 25.080, IBMLaboratory, Vienna, 1968.

[S20] R.W. FLOYD : Assigning meanings to Programs. Mathematical Aspects of computer science. Proc of symp. in applied. math. (J.T. Schwarz ed.). American mathematical Society, Providence, Rhode Island, 1967, 19, 19-32.

[S21] J.V. GARWICK : The definition of Programming Languages by their compilers. Formal Language Description Languages, Proc. IFIP 1964 (Steel ed.). North Holland, Amsterdam, 1966, 139-147.

[S22] C.A.R. HOARE : The axiomatic Method. The National Computing Centre Manchester, England, 1968.

[S23] C.A.R. HOARE : An axiomatic Basis for computer Programming. CACM, 1969, 12, 576-583.

[S24] C.A.R. HOARE and M. CLINT : Program proving : Jumps and Functions. IFIP, WG. 22, bulletin numéro 6, 1970.

[S25] C.A.R. HOARE : Procedures and Parameters : an axiomatic Approach. Symposium on semantics of Programming Languages. Springer-Verlag , 1971, 188, 102-116.

[S26] C.A.R. HOARE : Proof of a Program Find. CACM, 1971.

[S27] Y.I. IANOV : The Logical schemes of algorithms. Problems in cybernetics, Pergamon Press New-York, 1960, 1, 82-140.

[S28] S. IGARASHI : Semantics of Algol Like statements. Symposium on semantics of programming Languages. Springer Verlag, 1971, 188, 117-177.

[S29] D.E. KNUTH : Semantics of Context free Languages. Math. Systems theory, 1968, 2, 127-145.

[S30] P.J. LANDIN : The Mechanical Evaluation of Expressions Comp. J. , 1964, 6, 308-320.

[S31] P.J. LANDIN : A correspondence between Algol 60 and Church's Lambda Notation. CACM, 1965, 8, 89-101 and

158-165.

[S32] P.J. LANDIN : A formal Description of Algol 60. Formal Language description Languages, Proc IFIP 1964. (Steel ed.). North Holland Amsterdam, 1966, 266-294.

[S33] P.J. LANDIN : The Next 700 Programming Languages. CACM, 1966, 9, 157-164.

[S34] P. LAUER : Formal Definition of Algol 60. IBM Laboratory Vienna. Technical report TR 25.88, 1968.

[S35] P. LUCAS, K. WALK : On the Formal description of PL/1 Ann. Rev. in Autom. Progr. Pergamon Press New-York, 1968, 6, part 3.

[S36] P. LUCAS, P. LAUER and H. STIGLEITNER : Method and Notation for the Formal definition of Programming Languages. IBM laboratory Vienna. Technical Report TR 25.087, 1968.

[S37] J. Mc CARTHY : A Basis for a Mathematical theory of computation. Computer Programming and formal systems (P. Braffort and D. Hirshberg ed.). North Holland Amsterdam, 1963.

[S38] J. Mc CARTHY : Towards a Mathematical science of computation. Information Processing (Popplewell ed.). North Holland, Amsterdam, 1963, 62, 21-28.

[S39] J. Mc CARTHY : A formal description of a subset of Algol Formal Language Description Languages (Steel ed.). North Holland, Amsterdam, 1966, 1-12.

[S40] Z. MANNA : The correctness of Programs. J.C.S.S., 1969, 3.

[S41] Z. MANNA and A. PNUELI : Formalization of Properties of Functionnal Programs. J A C M, 1972, 15, numéro 7.

[S42] Z. MANNA and A. PNUELI : Correction totale des programmes méthode axiomatique. Acta informatica Springer, 1974.

[S43] Z. MANNA, S. NESS and J. VUILLEMIN : Inductives methodes for Proving Properties of Programs. CACM, 1973, 16, 491-502.

[S44] E.J. NEUHOLD : The formal description of Programming Languages. IBM Syst. J., 1971, 2, 86-113.

[S45] L. NOLIN : Formalisation des notions de machine et de programme. GAUTHIER-VILLARS, Paris, 1969.

[S46] C. PAIR : La formalisation des langages de programmation. Mathématiques et sciences humaines numéro 34. GAUTHIER-VILLARS, Paris, 1971.

[S47] D. SCOTT : Outline of a Mathematical theory of computation Oxford Mono PRG-2, Oxford University, 1970.

[S48] D. SCOTT : Mathematical concepts in Programming Languages semantics. Spring Joint Computer Conference, 1972.

[S49] D. SCOTT and C. STRACHEY : Toward a mathematical semantics for computer Languages. Oxford Mono PRG-6, Oxford University, 1972.

[S50] C. STRATCHEY : Towards a Formal semantics. Formal Languages Description Languages, Proc IFIP, 1964 (Steel ed.) North Holland, Amsterdam, 1966, 198-220.

[S51] J. VUILLEMIN : Proof technics for recursives Programs. Ph. D. Thesis Standford University, 1973.

[S52] J. VUILLEMIN : Syntaxe, semantique et axiomatique d'un langage de programmation simple. Thèse d'état (à paraitre).

[S53] P. WEGNER : The Vienna definition Language. TR 70-21-2. Center for computer and information. Sciences, Brown University, 1970. Computing Surveys.

[S54] P. WEGNER : Programming Language semantics. Courant computer Science symposium 2, 1970. Formal semantics of Programming Languages (Rustin ed.). Prentice Hall. Inc Englewood Cliffs New Jersey.

[S55] A. VAN WIJNGAARDEN : Generalized Algol. Symbolic Language in Data Processing, Proc Icc sym. Rome 1962. Gordon and Breach New-York, 409-419.

[S56] A. VAN WIJNGAARDEN : Recursive Definition of syntax and semantics. Formal Language Description Languages, Proc. IFIP 1964 (Steel ed.). North Holland Amsterdam, 1966, 13-24.

[S57] W. WILNER : Formal semantic definition using synthesized and inherited attributes. Courant computer science symposium 2, 1970. Formal semantics of Programming Languages (Rustin ed.). Prentice Hall Inc. Englewood Cliffs New Jersey.

[S58] N. WIRTH and H. WEBER : EULER, a generalization of Algol and its formal definition. CACM, 1966, 9, 13-23 and 89-99.

STRUCTURES D'INFORMATION

[SI1] D'IMPERIO : Data structures and their Representation in storage. Ann. Rev. in Aut. Programming 5, part 1. Pergamon Press New-York, 1968.

[SI2] J.P. FINANCE, J.L. REMY : Structure d'information et sémantique d'un langage de programmation. Ecole d'été d'information de l'AFCET, Grenade, 1973.

[SI3] C.A.R. HOARE : Notes on the theory and Practice of Data structuring. IFIP WG.22, Bulletin numéro 6, 1970.

[SI4] G.H. MEALY : Another look at data. Proc. Fall Joint Comp, 1967, Conf 31, 525-534.

[SI5] C. PAIR : Les structures d'informations et leurs représentations en mémoire. Cours de l'Ecole d'été d'informatique de l'AFCET, Alès, 1971.

[SI6] C. PAIR, J.P. FINANCE : Formalisation des notions de donnée, d'information et de structure d'information. Université de Nancy 2, 1973.

[SI7] J.L. REMY : Structure d'information, formalisation des notions d'accès et de modification d'une donnée. Thèse de troisième cycle, Université de Nancy 1, 1974.

THEORIE DES LANGAGES

[T1] N. CHOMSKY : Aspects of the theory of syntax. The M.I.T Press, Cambridge, Mass, 1965

[T2] S. GINSBURG : The mathematical theory of context free languages. Mac Graw - Hill, New-York, 1966.

[T3] C. PAIR et A. QUERE : Définition et études des bilangages réguliers. Information and control 13, 1968, 565-593.

[T4] A. QUERE : Etude des ramifications et des bilangages. Thèse de spécialité, Faculté des sciences de Nancy, 1969.

NOM DE L'ETUDIANT : FINANCE J.P.

NATURE DE LA THESE : Doctorat de Spécialité en Mathématiques Appliquées

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 14 Juin 1974

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

UNIVERSITÉ DE NANCY I
Président

J.R. HELLOT