Définition de ALPHA : un langage pour la programmation systolique

Christophe Mauras

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1090

Programme 2
Structures Nouvelles d'Ordinateurs

DEFINITION DE ALPHA : UN LANGAGE POUR LA PROGRAMMATION SYSTOLIQUE

Christophe MAURAS

Septembre 1989

RR-1090

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Définition de ALPHA : un langage pour la programmation systolique *†

Christophe Mauras

Juin 1989
18 pages
Publication interne n^0 482

Résumé

Le langage Alpha a tout d'abord été conçu comme un langage spécialisé, pour faciliter la conception d'algorithmes et d'architectures systoliques. Un des objectifs était d'aider à la mise en oeuvre des techniques et modèles du projet Diastol.

Les réflexions sur la "propreté" du langage et sur sa "généralité" ont finalement conduit à la définition d'un langage de programmation "parallèle régulier". Ses principaux concepts (flux de données, synchrone, déclaratif) sont hérités du langage Lustre dont Alpha étend un sous-ensemble.

Ce rapport est la référence du langage Alpha. Il y est d'abord décrit informellement. On donne ensuite sa sémantique dénotationnelle. On présente finalement une approche transformationnelle de la programmation systolique, illustrée par un exemple.

Definition of ALPHA : a Language for Systolic Programmation

Abstract

The Alpha language has been designed to help synthesis of systolic arrays. One of the goals was to implement efficiently the equational model and technical methods of the Diastol project.

Studying desirable features of such a language, lead us to design a complete programming language, which is more general than our first idea. The major caracteristics are inherited from the Lustre language, which is a synchronous data-flow declarative language.

This report describes the language by means of an informal explanation, and of its denotational semantics. We then show our rewriting approach of systolic programmation. This is illustrated by an example.

*Ce projet est supporté par le PRC C^3 et par un contrat avec la société SOREP.

†Cet article a été présenté aux journées Afcet Groplan "Développement de Programmes pour Machines Parallèles", Chamonix, Mai 1989.

1 Introduction

Les recherches dans le domaine systolique ont tout d'abord porté sur la réalisation d'architectures VLSI spécialisées, et sur la définition de nouveaux algorithmes. La complexité croissante des problèmes à résoudre a ensuite suscité des efforts méthodologiques importants. Sans vouloir être exhaustif, on peut citer parmi les "méthodes temporelles" la resynchronisation [Leiserson et al.86], et parmi les "méthodes spatiales" la projection des dépendances [Gachet et al.86].

La nécessité de comparer, le désir d'unifier, méthodes et algorithmes oriente actuellement les recherches vers l'étude de modèles formels permettant synthèse preuve et vérification. Ces approches [Choo et al.88] [Perrin88] font un large usage des acquis de la programmation fonctionnelle et parallèle.

Parallèlement, l'intérêt des langages synchrones, et de Lustre en particulier, pour la programmation systolique a été montré dans [Halbwachs et al.86].

Notre étude s'inscrit dans cette double voie. On propose un langage baptisé Alpha qui repose sur le modèle des "Systèmes d'Equations Récurrentes Linéaires". On essaie de mettre en évidence les liens qui existent entre cette formulation spatiale d'un problème (calculs indexés par un point de $\mathbf{Z}^n$) et une formulation temporelle synchrone.

On présente le langage et sa sémantique, ainsi qu'un petit exemple de programmation systolique.

2 Le langage Alpha

2.1 Présentation générale

Les Systèmes d'Equations Récurrentes Linéaires (SERL pour la suite) constituent le modèle du langage Alpha. Ils sont définis ainsi :

Définition 2.1 *Un SERL est un système de m fonctions $U_1, U_2, ...U_m$ définies sur l'ensemble D des points entiers d'un polyèdre convexe de $\mathbf{R}^n$ tel que:*

$$\forall i \in \{1...m\}, \forall p \in D_i,$$

$$U_i(p) = f_i(U_{i_1}(I_{i_1}(p)), U_{i_2}(I_{i_2}(p)), ..., U_{i_s}(I_{i_s}(p))),$$

où :

- *D_i est un polyèdre convexe inclus dans D,*
- *I_{i_k}, $k \in [1..s]$, est une fonction affine entière, appelée fonction de dépendance*
- *f_i est soit une fonction s-aire stricte, soit une fonction conditionnelle dépendant des coordonnées des points de D, et telle que chaque branche de la condition définisse un sous-ensemble convexe de D. Si f_i est une fonction conditionnelle, elle a la forme suivante :*

$$\begin{array}{l} f_i(U_{i_1}(I_{i_1}(p)), \cdots, U_{i_s}(I_{i_s}(p))) = \\ \textbf{if } p \in D_i{}^1 \textbf{ then } f_i{}^1(U_{i_1}{}^1(I_{i_1}{}^1(p), \cdots) \\ \vdots \\ \textbf{if } p \in D_i{}^k \textbf{ then } f_i{}^k(U_{i_1}{}^k(I_{i_1}{}^k(p), \cdots) \end{array}$$

où :

  - *les fonctions f_i^j ont la même forme que les fonctions f_i,*
  - *$\{D_i^j\}_{j \in [1..k]}$ est une partition de D_i,*
  - *chaque argument $U_{i_r}^j$, pour $1 \leq j \leq k$ et $1 \leq r \leq s$, apparaît dans au moins une branche conditionnelle.*

La définition du langage a répondu au double objectif de représenter ces SERL, et d'être un langage de haut niveau, bien défini et proche des spécifications.

Les principales caractéristiques du langage sont les suivantes :

2.1.1 Langage flux de données

Chaque variable d'un programme Alpha est une fonction sur un domaine de $\mathbf{Z}^n$. On parlera de "variable spatiale" pour dire qu'une variable est une collection de valeurs indexées par un "domaine spatial". Il s'agit en quelque sorte d'une généralisation de la notion de "flux de données" telle qu'elle apparaît dans Lucid [Ashcroft et al.77] et Lustre [Caspi et al.87].

Ces variables spatiales sont les objets de base du langage. On ne peut effectuer dessus que des manipulations globales. Par rapport aux approches plus classiques qui consistent à manipuler les instances itérativement, les avantages sont :

- de donner une description de plus haut niveau où la régularité est explicite,
- de permettre des manipulations formelles plus aisées.

Par exemple, la transposition d'une matrice s'écrirait en Pascal :

```
var A, B : array [1..10, 1..10] of integer;
begin
    for i := 1 to 10
        for j := 1 to 10
            B [i,j] := A [j,i]
end;
```

et en Alpha :

```
var A, B : {(i,j) | 1<=i,j<=10 } of integer;
    let
        B = A (lambda i,j. j,i)
    tel
```

A et B sont ici des fonctions :

$$\{(i,j) \mid 1 \leq i,j \leq 10\} \rightarrow Int$$

où Int dénote le domaine des entiers.

On parle de généralisation de flux : en effet, une variable définie sur le domaine $\{i \mid i \geq 0\}$ correspond à la notion courante de flux.

2.1.2 Langage applicatif - équationnel

La forme équationnelle était naturelle pour décrire des SERL. De plus une telle forme a des propriétés intéressantes, qui rendent la programmation et le raisonnement sur les programmes plus faciles.

Alpha respecte le "**principe de substitution**" que l'on peut énoncer ainsi :

Une équation $X = Exp$ spécifie une synonymie totale entre la variable X et l'expression Exp.

Dans l'exemple du paragraphe précédent, on peut remplacer toute utilisation de la variable B par l'expression qui la définit : $A(\ldots)$.

2.1.3 Langage de haut niveau

A travers ce qualificatif un peu flou et prétentieux, on veut simplement dire que Alpha adopte les principes classiques d'un langage fonctionnel, structuré et fortement typé.

Un programme Alpha définit une fonction entre des variables spatiales en entrée et en sortie. On peut hiérarchiser les définitions.

Le langage est fortement typé : outre le typage classique, un "calcul des domaines spatiaux" permet de vérifier qu'une variable est utilisée de façon cohérente avec sa déclaration. Pour chaque variable on doit en effet déclarer son domaine spatial.

L'importance accordée ici à la sémantique statique est à rapprocher de ce qui est fait en Lustre pou le calcul des horloges.

2.2 Définition du langage

2.2.1 Variables et équations

Comme les variables, toute expression Alpha est une fonction d'un domaine spatial (sous-ensemble de $\mathbf{Z}^n$) vers un ensemble de valeurs qui dépend du type de l'expression.

Pour préciser, prenons par exemple le domaine :

$$D = \{(i,j) \mid i > 0, 1 \leq j \leq 4\}$$

Une variable X déclarée sur D est une collection de valeurs :

$$\begin{pmatrix} X_{1,1} & X_{2,1} & \\ X_{1,2} & X_{2,2} & \\ X_{1,3} & X_{2,3} & \cdots \\ X_{1,4} & X_{2,4} & \end{pmatrix}$$

Une équation $X = Exp$ spécifie que :

$$\forall z \in D, X(z) = Exp(z)$$

Une expression Alpha est composée de variables, de constantes et d'opérateurs du langage.

Les constantes entières et booléennes sont définies sur le domaine singleton $\mathbf{Z}^0$ et peuvent être étendues à un domaine spatial quelconque. C'est-à-dire que la constante entière n est définie pour $z \in \mathbf{Z}^0$ comme $\lambda z.n$.

2.2.2 Opérateurs immobiles

On appelle opérateurs immobiles, les opérateurs qui prennent des arguments définis au moins sur le même domaine spatial et qui rendent un résultat sur ce domaine.

Ces opérateurs sont la généralisation spatiale des opérateurs classiques. Ils s'appliquent terme à terme sur leurs arguments.

Par exemple soit :

$$X = \begin{matrix} X_{1,1} & X_{2,1} & X_{3,1} \\ X_{1,2} & X_{2,2} & \\ X_{1,3} & & \end{matrix} \qquad Y = \begin{matrix} Y_{1,1} & Y_{2,1} & Y_{3,1} \\ Y_{1,2} & Y_{2,2} & \\ Y_{1,3} & & \end{matrix}$$

Alors l'expression $X + Y$ vaut :

$$\begin{matrix} X_{1,1}+Y_{1,1} & X_{2,1}+Y_{2,1} & X_{3,1}+Y_{3,1} \\ X_{1,2}+Y_{1,2} & X_{2,2}+Y_{2,2} & \\ X_{1,3}+Y_{1,3} & & \end{matrix}$$

On définit ainsi :

- les opérateurs sur les entiers : $+, -, *, div, mod$
- les opérateurs booléens : or, and, not
- les opérateurs de comparaison : $=, <, \leq, >, \geq, \neq$
- la conditionnelle : if cond $then$ expl $else$ exp2

Remarque : Le terme opérateurs immobiles est une analogie avec la terminologie Lustre, qui parle d'opérateurs instantanés, pour les opérateurs appliqués terme à terme sur les suites.

2.2.3 Opérateurs spatiaux

Ce sont les seuls opérateurs qui manipulent explicitement les domaines spatiaux. On présente successivement l'opérateur "case" qui permet de discriminer suivant les points du domaine, et l'opérateur de dépendance qui établit des relations entre les points du domaine.

L'opérateur case : L'expression :

$$\begin{array}{l} case \\ \quad Dom_1 : Exp_1 \\ \quad \ldots \\ \quad Dom_n : Exp_n \\ esac \end{array}$$

est définie, si les domaines Dom_i sont des parties de $\mathbf{Z}^n$, sur l'union convexe de tous ces domaines, notée Dom.

En un point z, si une et une seule expression Exp_i est définie alors le *case* vaut la valeur de cette expression. Il vaut "indéfini" en cas d'absence de définition ou "erreur" en cas de définition multiple.

L'opérateur dépendance : On appelle "fonction de dépendance" une fonction entre domaines spatiaux. Soit :

- Exp une fonction : $Dom' \rightarrow V$ où V est un espace de valeurs,
- dep une fonction de dépendance : $Dom \rightarrow Dom'$

Alors l'opérateur dépendance compose Exp et dep. C'est une expression spatiale définie sur le domaine Dom' que l'on note $(Exp)dep$.

On a restreint les domaines spatiaux à des sous-ensembles convexes de $\mathbf{Z}^n$. On restreint les fonctions de dépendance aux fonctions affines.

On en a déjà donné un exemple pour la transposition d'une matrice (cf 2.1.1).

$$B = (A)(\lambda i, j.(j, i))$$

peut se lire :

$$B(i, j) = A(j, i)$$

L'intérêt de la première formulation, qui paraît peut être moins intuitive que la seconde, est de respecter le principe de substitution. i et j, qui sont des variables muettes, n'ont d'utilité et donc de réel sens que pour définir la fonction de dépendance $\lambda i, j.(j, i)$

Définir tous les objets spatiaux sans référence à des indices globaux permet d'obtenir une "transparence référentielle".

Exemple : Ces deux opérateurs spatiaux permettent de définir des fonctions récurrentes :

```
X    : { i | 1 <= i <= 10 } of integer
somme  : { i | 0 <= i <= 10 } of integer

somme = case
          {i=0} : 0 ;
          {i>0} : X +
                  somme (lambda i. i-1);
        esac;
```

La variable *somme* vaut la suite des sommes partielles des éléments de X.

2.2.4 Structuration

Un programme Alpha comporte :

- des déclarations de paramètres d'entrée et de sortie,
- des déclarations de variables locales,
- un système d'équations

Il définit une fonction entre les variables spatiales d'entrée et de sortie. Le système d'équations comporte une et une seule équation pour chaque variable locale et chaque variable de sortie. Ceci permet de vérifier l' assignation unique équation par équation.

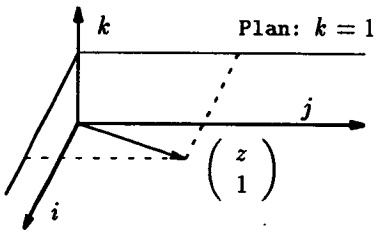

Figure 1 : Le plan affine dans l'espace vectoriel homogène

2.3 Les domaines spatiaux

On a défini le langage sans vraiment préciser la structure des domaines spatiaux considérés. On donnera d'ailleurs une sémantique du langage en considérant les domaines spatiaux et les fonctions de dépendances de façon abstraite.

Ceci a pour but de séparer les difficultés. Nous allons maintenant nous attacher à décrire une classe particulière de domaines spatiaux.

2.3.1 Les domaines polyédraux convexes

Définition 2.2 *On appelle domaine polyédral convexe, un sous ensemble de* $\mathbf{Z}^n$ *qui vérifie un ensemble fini d'inéquations linéaires (appelé système de contraintes)*

Ces domaines sont particulièrement intéressants parce que :

- ils comprennent les structures de données classiques utilisées en algèbre linéaire et en traitement du signal :
  - matrices $\{(i,j) \mid 1 \leq i,j \leq n\}$
  - flux $\{i \mid i \geq 0\}$
  - ...
- ce sont les domaines réguliers les plus généraux utilisés pour la systolisation d'algorithmes.

Un domaine polyédral convexe peut aussi être caractérisé par son système générateur: l'ensemble de ses sommets et rayons extrémaux.

Des algorithmes efficaces existent pour passer d'une représentation à l'autre [Fernandez et al.88] [Halbwachs].

On utilisera comme fonction de dépendance sur ces domaines les fonctions affines.

Propriétés :

- Les domaines polyédraux convexes sont stables par intersection.
- Les domaines polyédraux convexes sont stables par enveloppe convexe de l'union (plus petit domaine polyédral convexe contenant l'union, d'où la propriété).
- Les domaines polyédraux convexes sont stables par application d'une fonction affine unimodulaire [Schrijver86].

2.3.2 Codage des convexes et fonctions affines

On adopte le codage en coordonnées homogènes qui permet de considérer un espace affine de dimension n comme l'espace vectoriel de dimension $n+1$ (codage utilisé en vision et en robotique).

Sur cet exemple (voir figure 1), le plan affine est isomorphe à la coupe $k = 1$ de l'espace vectoriel.

Un domaine polyédral convexe $\mathbf{P}$ de $\mathbf{Z}^n$ est alors un cône $\mathbf{C}$ de $\mathbf{Z}^{n+1}$ (voir figure 2).

- Les rayons de $\mathbf{C}$ dont la dernière composante est nulle sont des rayons de $\mathbf{P}$.

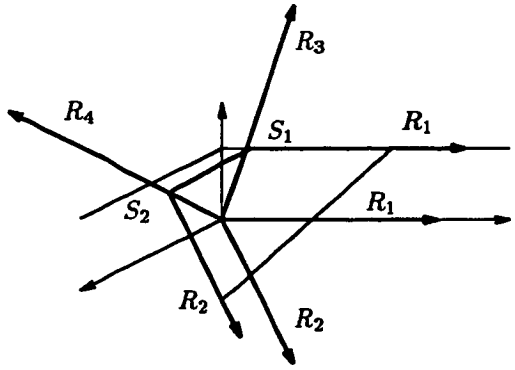

Figure 2 : Un polyèdre vu comme un cône dans l'espace homogène.

- Les rayons de **C** dont la dernière composante vaut 1 (à une normalisation près) correspondent à des sommets de **P**.

Sur cette exemple, les rayons $\mathbf{R_1}$ et $\mathbf{R_2}$ du cône sont aussi des rayons de **P**. Aux rayons $\mathbf{R_3}$ et $\mathbf{R_4}$ correspondent les sommets $\mathbf{S_1}$ et $\mathbf{S_2}$ du polyèdre **P** (hachuré).
L'intérêt de ce codage est :

- de simplifier l'algorithme de traduction des domaines [Fernandez et al.88] car il n'y a plus que des cônes.
- de voir les fonctions affines comme des fonctions linéaires, et donc de les composer par produit de matrices.

On codera donc un domaine polyédral convexe **P** de $\mathbf{Z}^n$, que l'on notera $\mathbf{P(C,R)}$, par :

- sa matrice de contraintes **C** :

$$\mathbf{C}_{(q,n+1)} = \left(\begin{array}{c|c} \mathbf{A} & b \\ 0 & 1 \end{array} \right)$$

P est l'ensemble des points entiers z vérifiant :

$$\mathbf{C} * \left(\begin{array}{c} z \\ 1 \end{array} \right) \geq 0$$

soit :

$$\mathbf{A} * z + b \geq 0$$

- la matrice **R** des rayons du cône associé :

$$\mathbf{R}_{(p,n+1)} = \left(\begin{array}{c|c} \mathbf{S_1} & 1 \\ \vdots & \\ \mathbf{S_m} & 1 \\ \mathbf{R_{m+1}} & 0 \\ \vdots & \\ \mathbf{R_p} & 0 \end{array} \right)$$

P est l'ensemble des points entiers z vérifiant :

$$\left(z^T 1 \right) = a^T * \mathbf{R}$$

où $a \in \mathbf{R}^p$

Une fonction affine f de $\mathbf{Z}^n$ dans $\mathbf{Z}^m$ sera représentée par une matrice :

$$\mathbf{M}_{(m+1,n+1)} = \begin{pmatrix} \mathbf{L} & | t \\ 0 & | 1 \end{pmatrix}$$

$$z \in \mathbf{Z}^n, \quad f(z) = \mathbf{M} * \begin{pmatrix} z \\ 1 \end{pmatrix} = \mathbf{L} * z + t$$

$\mathbf{L}$ est la partie linéaire, t une translation.

Proposition 2.1 *Soit* $\mathbf{P}(\mathbf{C},\mathbf{R})$ *un domaine polyédral convexe de* $\mathbf{Z}^n$, f *une fonction inversible sur* $\mathbf{Z}^n$ *de matrice* $\mathbf{M}$, *on a :*

$$f(\mathbf{P}(\mathbf{C},\mathbf{R})) = \mathbf{P}(\mathbf{C} * \mathbf{M}^{-1}, \mathbf{R} * \mathbf{M}^T)$$

La démonstration est directe en utilisant les caractérisations des matrices $\mathbf{C}$ et $\mathbf{R}$.

2.3.3 Représentation concrète

La représentation concrète des domaines utilise la notation indicielle classique. Un domaine est un ensemble d'indices et d'inéquations linéaires sur ces indices.

Exemple : $\{(i,j) \mid i > 0, 1 \leq j \leq 4\}$

Remarque : Il n'y a pas d'indices globaux, les noms "i" et "j" n'ont de sens que dans la portée des accolades.

Une fonction de dépendance d'un domaine $\{(i,j) \mid \ldots\}$ vers un domaine $\{(i,j,k) \mid \ldots\}$ sera notée :

$$\lambda i, j.(f(i,j), g(i,j), h(i,j))$$

On emprunte la notation du lambda-calcul pour exprimer la remarque précédente : les noms d'indices sont locaux à la lambda-expression.

2.3.4 Abstraction des domaines

On s'est doté de domaines possédant une structure forte. Ils reposent sur des propriétés d'algèbre linéaire et de programmation linéaire en nombres entiers. Pour la suite, on fera abstraction de cette structure.

On définira à la manière des types abstraits, les opérations que l'on veut effectuer sur les domaines. Le lecteur vérifiera que l'on a énoncé les résultats nécessaires pour les domaines polyédraux convexes.

On cherche seulement à munir l'ensemble des domaines spatiaux d'une structure d'ordre. Les objectifs sont:

- de comparer des domaines pour pouvoir dire qu'une variable est "moins définie" qu'une autre,
- d'opérer sur des domaines comparables pour, par exemple, calculer le domaine de définition d'une expression, contenant des sous-expressions définies sur des domaines différents.

Proposition 2.2 *L'ensemble* $D(n)$ *des domaines polyédraux convexes de* $\mathbf{Z}^n$ *muni de l'inclusion, est un treillis complet.*

Démonstration : Pour les notions et résultats élémentaires sur les treillis, on se reportera à [Birkhoff67]. L'argument principal est que "être un domaine polyédral convexe" est une propriété de fermeture. D'après le théorème de Moore [Moore10], les sous-ensembles fermés d'un treillis complet, forment un treillis complet, ce qui permet de conclure, car $\mathbf{Z}^n$ est un treillis complet.

L'opération "borne inférieure" notée $\wedge$ est l'intersection.

L'opération "borne supérieure" notée $\vee$ est l'enveloppe convexe de l'union.

On s'intéresse aux domaines de $\mathbf{Z}^n$ pour n variable. Cependant on s'interdit d'opérer sur des domaines de dimensions différentes. On aurait pu définir une projection canonique d'un espace donné dans un espace de dimension supérieure. Cela ne nous semble pas naturel et de plus est source d'erreurs. Par exemple $D_1 = \{i,j \mid 1 \leq i,j \leq 4\}$ et $D_2 = \{i,j,k \mid 1 \leq i,j \leq 4, k = 0\}$ ne sont pas comparables.

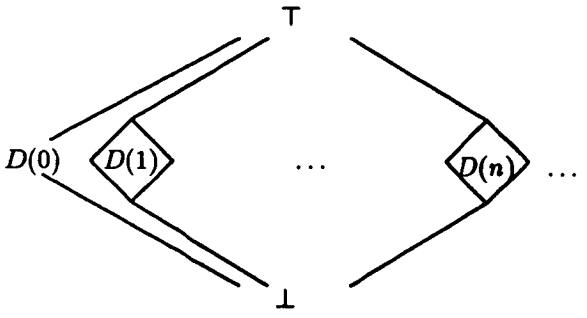

Figure 3 : Le treillis des domaines spatiaux

Le langage permet de manipuler l'image d'un domaine. Ainsi $\lambda i.j.(i, j, 0)$ permet de relier explicitement les deux domaines précédents.

On notera **U** l'univers des points des domaines:

$$\mathbf{U} = \mathbf{Z}^0 \cup \mathbf{Z}^1 \cup \ldots \cup \mathbf{Z}^n \ldots$$

L'ensemble **DOM** des domaines spatiaux considérés est alors un ensemble de parties de **U** ordonnées selon la figure 3.

Proposition 2.3 *L'ensemble* **DOM** *muni de l'inclusion est un treillis complet. La borne supérieure notée* $\top$ *est le "domaine erroné". La borne inférieure notée* $\bot$ *est le "domaine vide".*

On appelle **DEP** le domaines des fonctions $dom \rightarrow dom'$ où $dom, dom' \in \mathbf{DOM}$
On se donne les opérations suivantes :

Inf	**DOM x DOM**	→	**DOM**
Sup	**DOM x DOM**	→	**DOM**
Expansion	**DOM**	→	$\mathbf{U}^*$
Application	**DEP x DOM**	→	**DOM**
Image	**DEP x U**	→	$\mathbf{U} \cup \{\top\}$
Point		→	**DOM**

L'opérateur expansion donne la liste des points du domaine polyédral convexe.
L'opérateur application calcule l'image d'un domaine par une fonction affine unimodulaire.
L'opérateur image calcule l'image d'un point par une fonction affine unimodulaire.
L'opérateur point donne le domaine élémentaire $\mathbf{Z}^0$.

3 Sémantique de Alpha

On donne d'abord une sémantique dénotationnelle du langage. C'est assez direct, vu la forme du langage. Les valeurs spatiales sont définies comme des plus petits points fixes, calculés par induction sur les domaines spatiaux.

3.1 Sémantique dénotationnelle

On définit la sémantique pour un noyau suffisant du langage. On suppose que l'analyseur fournit un programme Alpha sous cette forme abstraite.

3.1.1 Domaines syntaxiques

Les domaines syntaxiques de base suivants vont permettre de construire les expressions du langage.

- ID : domaine des identificateurs $id \in ID$
- INT : domaine des entiers relatifs $i \in ID$
- BOOL : domaine des booléens $b \in BOOL$
- UOP : domaine des opérateurs unaires $uop \in UOP$
- BOP : domaine des opérateurs binaires $bop \in BOP$
- DOM : domaine des domaines spatiaux $dom \in DOM$
- DEP : domaine des fonctions de dépendance $dep \in DEP$

On peut maintenant définir le domaine EXP des expressions par la syntaxe abstraite suivante ($e \in EXP$) :

$$e ::= id \mid i \mid b \mid \textbf{if}\, e_1\, \textbf{then}\, e_2\, \textbf{else}\, e_3 \mid \textbf{uop}\, e \mid$$

$$e_1\, \textbf{bop}\, e_2 \mid dom : e \mid \textbf{sup}\, e_1 \ldots e_n \mid e(dom : dep)$$

L'opérateur *case* est réalisé par le *sup* de n expressions exp_i restreintes à un domaine dom_i.

$$\textbf{case}\, dom_1 : e_1 \ldots dom_n : e_n\, \textbf{esac}$$

$$=_{def}$$

$$\textbf{sup}\, dom_1 : e_1 \ldots dom_n : e_n$$

Le domaine EQ des equations est défini par la syntaxe abstraite suivante ($eq \in EQ$) :

$$eq ::= id = e$$

Le domaine EQS des systèmes d'équations est défini par la syntaxe abstraite suivante :

$$eqs ::= \textbf{let}\, eq_1 \ldots eq_n\, \textbf{tel}$$

le domaine PRG des programmes est défini par la syntaxe abstraite suivante :

$$prog ::= \textbf{input}\, decls\, \textbf{output}\, decls\, \textbf{local}\, decls\, eqs$$

$$decls ::= decl_1 \ldots decl_n$$

$$decl ::= id : dom$$

3.1.2 Domaines sémantiques

Les domaines sémantiques de base sont les domaines usuels, étendus par la valeur indéfinie $\perp$.

- $Z = \mathbf{Z} \cup \{\perp\}$ (entiers relatifs)
- $B = \{tt, ff\} \cup \{\perp\}$ (booléens)
- $V = Z \cup B$
- UOP (opérateurs unaires)
- BOP (opérateurs binaires)
- $OP = UOP \cup BOP$

On a défini DOM le treillis des domaines spatiaux (voir "abstraction des domaines").
On peut maintenant construire les domaines sémantiques des expressions.

- $VAL = dom \rightarrow V, dom \in DOM$
- $SP = DOM \times VAL$ domaine des valeurs spatiales; la redondance entre le domaine et la fonction définie sur ce domaine simplifie l'expression de la sémantique. On se donne des "fonctions composantes" (dom , val) pour accéder aux champs correspondants.
- $EV = ID \rightarrow SP$ environnements.
- $EXP = EV \rightarrow SP$, une expression est une valeur spatiale, fonction de la valeur dans l'environnement des identificateurs contenus dans l'expression.

3.1.3 Sémantique des expressions

On définit la fonction sémantique des expressions $\mathcal{E}$ qui associe à chaque construction syntaxique, sa sémantique dans EXP.

Identificateurs

$$\mathcal{E}(id) = \lambda\epsilon.\epsilon(id)$$

Constantes

$$\mathcal{E}(k) = \lambda\epsilon.(Point, \lambda z.k)$$

$Point$ est la fonction qui rend le domaine singleton, sur lequel sont définies toutes les constantes.

Opérateurs unaires

$$\mathcal{E}(uop\, e)(\epsilon) = (dom(\mathcal{E}(e)(\epsilon)), uop(val(\mathcal{E}(e)(\epsilon))))$$

Opérateurs binaires

$$\mathcal{E}(e_1\, bop\, e_2)(\epsilon) =$$
$$(inf(dom(\mathcal{E}(e_1)(\epsilon)), dom(\mathcal{E}(e_2)(\epsilon))),$$
$$(val(\mathcal{E}(e_1)(\epsilon)))bop(val(\mathcal{E}(e_2)(\epsilon))))$$

Conditionnelle

$$\mathcal{E}(if\, e_1\, then\, e_2\, else\, e_3)(\epsilon) =$$
$$(inf(dom(\mathcal{E}(e_1)(\epsilon)), \ldots, dom(\mathcal{E}(e_3)(\epsilon))),$$
$$\lambda z.si(val(\mathcal{E}(e_1)(\epsilon)))(z)$$
$$alors(val(\mathcal{E}(e_2)(\epsilon)))(z)$$
$$sinon(val(\mathcal{E}(e_3)(\epsilon)))(z))$$

Restriction

$$\mathcal{E}(d : e)(\epsilon) = (Inf(d, dom(\mathcal{E}(e)(\epsilon))), e)$$

Sup

$$\mathcal{E}(sup\, e_1 \ldots)(\epsilon) =$$
$$(Sup(dom(\mathcal{E}(e_1)(\epsilon)) \ldots),$$
$$Supval(val(\mathcal{E}(e_1)(\epsilon)) \ldots))$$

La fonction $Supval$ prend point par point le sup des valeurs définies ce qui donne "erreur" en cas de définition multiple.

Dépendance

$$\mathcal{E}(e(d : dep))(\epsilon) =$$
$$(Inf(d, dep^{-1}(dom(\mathcal{E}(e)(\epsilon)))),$$
$$(val(\mathcal{E}(e)(\epsilon))) \circ dep)$$

3.1.4 Sémantique des programmes

On définit les fonctions $\mathcal{Q}, \mathcal{S}, \mathcal{P}$ qui donnent respectivement les sémantiques d'une équation, d'un système d'équations, et d'un programme.

Equations

$$\mathcal{Q}(id = e)(\epsilon) =$$
$$(sup(dom(\mathcal{E}(e)(\epsilon)), (dom(\epsilon(id)))),$$
$$sup(val(\mathcal{E}(e)(\epsilon))), (val(\epsilon(id))))$$

On prend le sup de la définition de *id* contenu dans l'environnement initial, et de la définition fournie par l'expression *e*.

Systèmes La sémantique d'un système d'équations est le plus petit point fixe de l'enrichissement de l'environnement par les équations.

$$\mathcal{S}(\textbf{let}\ eq_1 \ldots eq_n\ \textbf{tel})(\epsilon) =$$
$$pppf(\lambda S.\mathcal{Q}(eq_n) \circ \ldots \circ \mathcal{Q}(eq_1) \circ S(\epsilon))$$

Programmes Un programme Alpha est un transformateur de valeurs spatiales. Le domaine sémantique d'un programme est donc le suivant :

$$PROG = SP^* \times SP^*$$

Un programme construit l'environnement initial avec les données en entrée et l'enrichit par le système d'équations.

$$\mathcal{P}(\textbf{input}\ d_1 \ldots d_m\ \textbf{output}\ d_{m+1} \ldots d_n\ \textbf{local}\ d_{n+1} \ldots d_p\ eqs)$$
$$=$$
$$\lambda v_1 \ldots v_m.(\lambda\epsilon.val(\epsilon(id_{m+1})) \ldots val(\epsilon(id_p))$$
$$(\mathcal{S}(eqs)([(dom_1, v_1) \ldots (dom_m, v_m)(dom_{m+1}, \perp) \ldots (dom_p, \perp)/id_1 \ldots id_p])))$$

où d_i dénote $id_i : dom_i$

3.2 Transformation de programmes

La définition d'une sémantique mathématique pour le langage Alpha est le support formel de ce qui suit. Les principes de base enrichis des propriétés des opérateurs constituent une "théorie de programmation". L'objectif est de voir le processus de conception d'un programme Alpha comme une suite de réécritures et de raffinements.

3.2.1 Principe de sustitution

Cette règle est fondamentale pour la manipulation de programmes mais aussi pour la vérification. Elle se formule ainsi :

On peut remplacer toute occurence d'une variable par sa définition et réciproquement

Le raffinement d'un programme consiste souvent à introduire des variables locales permettant de préciser la méthode de calcul du résultat. Par exemple, on peut remplacer :

$$S = X1 + X2 + X3 + X4$$

par

$$S = S1 + S2;$$
$$S1 = X1 + X2;$$
$$S2 = X3 + X4;$$

La vérification est utile quand la conception n'est pas totalement constructive, pour prouver qu'une mise en oeuvre respecte bien les spécifications.

Des spécifications décrivent directement les sorties en fonction des entrées et au besoin de variables locales récurrentes.

Une mise en oeuvre précise contient beaucoup plus de variables locales. Pour la comparer aux spécifications, on applique le principe de substitution en expansant le plus possible les variables locales.

Ces manipulations peuvent faire apparaître des expressions complexes à simplifier; on fera alors un usage important des propriétés algébriques des opérateurs.

3.2.2 Propriétés des opérateurs

On s'intéresse aux propriétés des opérateurs spatiaux du langage (sup, restriction, dépendance).

Soit *op* un opérateur immobile n-aire Alpha, on a les propriétés de distributivité suivantes :

Sup:

$$op(sup(t_1, \ldots t_m), e_2, \ldots e_n) =$$
$$sup(op(t_1, e_2, \ldots e_n) \ldots op(t_m, e_2, \ldots e_n))$$

Dépendance:

$$(op(e_1 \ldots e_n))(dom : dep) =$$
$$op(e_1(dom : dep), \ldots e_n(dom : dep))$$

La distributivité de l'opérateur de dépendance par rapport à un opérateur immobile est une généralisation de la propriété de resynchronisation [Leiserson et al.86].

En effet avec les déclarations :

$$X, Y, Z : \{t \mid t \geq 0\} of\ integer$$

les deux équations suivantes sont équivalentes :

$$Z = (X + Y)(\lambda t.t - 1)$$
$$Z = X(\lambda t.t - 1) + Y(\lambda t.t - 1)$$

Ceci correspond, pour l'architecture sous-jacente, à distribuer le délai sur les deux fils d'entrée de l'additionneur.

Restriction:

$$dom : op(e_1, \ldots e_n) =$$
$$op(dom : e_1, \ldots dom : e_n)$$

D'autres propriétés plus complexes existent, sur les combinaisons entre opérateurs spatiaux.

3.2.3 Déplacement d'une équation

Les méthodes de projection des dépendances affectent pour chaque variable définie sur un espace discret, leur temps et allocation par des fonctions linéaires. Ceci revient à effectuer un changement de repère (aussi appelé réindiçage), sur le domaine de chaque variable.

Dans notre modèle, cette opération est simplifiée par l'indépendance des domaines des variables et le style fonctionnel des définitions.

Le résultat suivant, qui exprime la "transparence référentielle", permet d'effectuer le changement de repère.

Proposition 3.1 *Le changement de repère, défini par une matrice P inversible, appliqué à une variable z d'un programme Alpha, préserve la sémantique de ce programme. Il est défini ainsi :*

Si z est définie sur le domaine dom

- *l'équation $z = exp$ est remplacée par : $z = exp(P(dom) : P^{-1})$*
- *toute occurence de $d : z$ en partie droite d'une équation est remplacée par $z(d : P)$*
- *le domaine de déclaration de z est $dom' = P(dom)$ c'est-à-dire si $dom = (C, R)$ alors $dom' = (C.P^{-1}, R.P^T)$*

Démonstration P est la matrice de passage de la nouvelle base à l'ancienne. C'est donc la matrice de l'identité de l'ancienne vers la nouvelle base. On remplace donc z par $z' \circ P$. z' est alors définie sur $P(dom)$.

$$z = exp$$

qui peut s'écrire :

$$z = exp(dom : id)$$

se réécrit en :

$$z' \circ P = exp(dom : id)$$

qui est équivalent à :

$$z' = exp(P(dom) : P^{-1})$$

d'après la sémantique de l'opérateur dépendance. On renomme ensuite z' en z.

L'expression analytique du domaine de z est donné par la proposition 2.1.

4 Un exemple de programmation systolique

On prend l'exemple d'un corrélateur qui doit reconnaître un mot déterminé de 8 bits r parmi un flot continu de signaux binaires e_t. Pour cela on calcule la distance de Hamming h_t entre les signaux e et r.

La spécification initiale en Alpha est la suivante :

```
system correlateur_specif
        (e_in, r_in {t|t>=0} of bool)
returns (h_out: {t|t>=0} of integer;
         e_out: {t|t>=0} of bool);
var
    r : {i|1<=i<=8} of bool ;
let
    h_out =  (diff (r(lambda t.1),
                    e_in(lambda t. t+1))
              +     ...
              + diff (r(lambda t.8),
                      e_in(lambda t. t+8)) )
               (lambda t. t-16);
    e_out = e_in (lambda t. t-16);
    r     = {i|1<=i<=8} : r_in ;
tel
```

Où la fonction diff donne 1 si $e \neq r$ et 0 sinon.

On va d'abord étendre la définition de h-out pour sérialiser les additions. On obtient alors une définition récurrente de h sur un domaine de dimension 2.

```
system correlateur_serl
        (e_in, r_in {t|t>=0} of bool)
returns (h_out: {t|t>=0} of integer;
         e_out: {t|t>=0} of bool);
var
    r : {i|1<=i<=8} of bool ;
    h : {t,i|t>=0, 0<=i<=8} of integer;
let
    h_out =  h (lambda t . t,0)
                (lambda t. t-16);
```

```
    h   = case
            {t,i| i=8} : 0 ;
            {t,i| i<=7}: h (lambda t,i. t,i+1)
               + diff ( r(lambda t,i. i+1),
                        e_in(lambda t,i. t+1));
          esac;
    e_out = e_in (lambda t. t-16);
    r     = {i|1<=i<=8} : r_in ;
tel
```

On fait de même pour les variables e et r jusqu'à obtenir le système entier sur un même domaine. Notons que les opérateurs ont des propriétés qui permettent de faire des simplifications. On peut par exemple composer les fonctions de dépendance.

Quand les calculs sont ainsi étalés dans l'espace, on peut appliquer la projection des dépendances, qui affecte à chaque calcul une coordonnée de temps et une d'espace.

On applique pour cela un changement de base approprié aux équations définissant h, e et r.

Après simplification, on obtient le programme Alpha suivant :

```
system correlateur
        (e_in, r_in {t|t>=0} of bool)
returns (h_out: {t|t>=0} of integer;
         e_out: {t|t>=0} of bool);
var
    h : {t,i|t>=0, 0<=i<=8} of integer;
    e,r : {t,i|t>=0, 0<=i<=8} of bool;
let
  h_out =  h (lambda t . t,0);
  h     = case
            {t,p| p=8} : 0 ;
            {t,p| p<=7}: h (lambda t,p. t-1,p+1)
                         + diff ( r , e );
          esac;
  e_out = e (lambda t. t,0);
  e     = case
            {t,p|p=8} : e_in ;
            {t,p|p<=7}: e (lambda t,p. t-2, p+1);
          esac;
  r     = case
            {t,p|t>=8} : r (lambda t,p. t-1,p);
            {t,p|t<=7,p<=7}:r(lambda t,p. t-1,p+1);
            {t,p|t<=7,p=8} : r_in(lambda t,p. t);
          esac;
tel
```

Si on interprète l'indice t comme le temps, on peut vérifier que ce programme est causal. En effet toutes les références sont passées et il n'y a pas de cycle instantané. On peut donc traduire ce programme en Lustre: il suffit d'éliminer la dimension du temps et de remplacer les références à $t-1$ par l'opérateur *pre*.

5 Conclusion

Par ce dernier exemple, nous avons voulu illustrer la façon dont on considère le processus de programmation systolique : une suite de réécritures utilisant les propriétés des opérateurs. Pour mettre en oeuvre ces techniques, une maquette de système d'édition et de manipulation syntaxique de programmes Alpha, est en cours de développement [Gachet et al.89]. Il utilise l'environnement Centaur [Borras et al.87]. De nombreux travaux restent à faire, particulièrement la définition d'une sémantique opérationnelle efficace.

Pour conclure on peut tenter une comparaison avec les approches contemporaines déjà citées. Le système Crystal de M.C.Chen repose sur le λ-calcul typé. Chaque "expression spatiale" y est typée par son domaine de

définition. Crystal n'impose aucune restriction sur la nature des domaines et des fonctions de dépendances. Le problème ouvert est de savoir si les problèmes où ce gain en généralité est nécessaire justifient le prix payé.

L'approche utilisée dans Systol est comparable à la nôtre bien que moins générale. La notion de "relation de communication" [Perrin88] fournit un modèle satisfaisant pour l'exécution parallèle asynchrone des systèmes d'équations.

La proximité avec Lustre n'est plus à montrer. La traduction de Alpha vers Lustre permettrait une exécution synchrone d'un programme Alpha.

En résumé, la proximité de ces modèles et leurs spécificités les rendent complémentaires, et laissent ouvertes de nombreuses perspectives dans le domaine de l'exécution parallèle de programmes équationnels.

Bibliographie

[Ashcroft et al.77] E.A. Ashcroft et W.W. Wadge. LUCID : a non procedural language with iteration. *CACM*, 20(7), July 1977

[Birkhoff67] G. Birkhoff. *Lattice Theory.* Volume 25, American Mathematical Society, 1967.

[Borras et al.87] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, et V. Pascual. *CENTAUR : the System.* Rapport technique 777, INRIA, December 1987.

[Caspi et al.87] P. Caspi, D. Pilaud, N. Halbwachs, et J.A. Plaice. LUSTRE : a declarative language for programming synchronous systems. *in Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 178–188, ACM, Munich (West Germany), January 1987.

[Choo et al.88] Y. Choo et M.C. Chen. *A Theory of Parallel Program Optimization.* Technical Report YALEU/DCS/TR-608, Yale University, July 1988.

[Fernandez et al.88] F. Fernández et P. Quinton. *Extension of Chernikova's Algorithm for Solving General Mixed Linear Programming Problems.* Rapport technique, IRISA - Rennes (France), 1988.

[Gachet et al.86] P. Gachet, B. Joinnault, et P. Quinton. Synthesizing systolic arrays using DIASTOL. *in* W. Moore, A. McCabe, et R. Urquhart, editeurs, *International Workshop on Systolic Arrays*, pages 25–36, Adam Hilger, University of Oxford, UK, July 2-4 1986.

[Gachet et al.89] P. Gachet, C. Mauras, P. Quinton, et Y. Saouter. ALPHA du CENTAUR: a prototype environment for the design of parallel regular algorithms. *in International Conference on Supercomputing*, A.C.M. SIGARCH, june 1989.

[Halbwachs] N. Halbwachs. Determination automatique de relations lineaires verifiees par les variables d'un programme, chapitre 1, definition et resultats fondamentaux sur les polyedres convexes, pp. 5-120. Thèse de 3ieme Cycle.

[Halbwachs et al.86] N. Halbwachs et D. Pilaud. Use of real-time declarative language for systolic array design and simulation. *in* W. Moore, A. McCabe, et R. Urquhart, editeurs, *International Workshop on Systolic Arrays*, pages 81–90, Adam Hilger, University of Oxford, UK, July 2-4 1986.

[Leiserson et al.86] C. E. Leiserson et J. B. Saxe. *Retiming Synchronous Circuitry.* Rapport technique 13, Digital, Systems Research Center, August 1986.

[Moore10] E. H. Moore. *Introduction to a form of general analysis.* Volume 2, AMS Colloquium Publication, New Haven, 1910.

[Perrin88] R.G. Perrin. Parallel solving of equation systems. *in* M. Cosnard, P. Quinton, M. Raynal, et Y. Robert, editeurs, *Parallel and Distributed Algorithms*, North-Holland, 1988.

[Schrijver86] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience series in Discrete Mathematics*, John Wiley and Sons, 1986.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 476 **SIMULATION REPARTIE DE SYSTEMES A EVENEMENTS DISCRETS: PARTIE 1 : MODELISATION ET SCHEMAS D'EXECUTION**
Philippe INGELS, Michel RAYNAL
26 Pages, Juin 1989.

477 **PROGRAMMING WITH MALI - UNIFICATION OR ORDERED TYPES**
Olivier RIDOUX
18 Pages, Juin 1989.

PI 478 **CLASSIFICATION OF CONCEPTS DESCRIBED BY TAXONOMIC PREORDONNANCE VARIABLES WITH MULTIPLE CHOICE**
Israël-César LERMAN, Philippe PETER
16 Pages, Juin 1989.

PI 479 **A SIMPLE GRAPH CONSTRUCTION OF SEMILINEAR REACHABILITY SETS OF VECTOR ADDITION SYSTEMS**
Gilles LESVENTES
16 Pages, Juin 1989.

PI 480 **THE MODELLING SYSTEM PYRAMIDE AS AN INTERACTIVE HELP FOR THE GUIDANCE OF THE INSPECTION VEHICLE CENTAURE**
Philippe EVEN, Lionel MARCE
22 Pages, Juin 1989.

PI 481 **VERS UNE INTERPRETATION QUALITATIVE DE COMPORTEMENTS CINEMATIQUES DANS LA SCENE A PARTIR DU MOUVEMENT APPARENT**
Edouard FRANCOIS, Patrick BOUTHEMY
40 Pages, Juin 1989.

PI 482 **DEFINITION DE ALPHA : UN LANGAGE POUR LA PROGRAMMATION SYSTOLIQUE**
Christophe MAURAS
18 Pages, Juin 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique