# Structuration of the ALPHA language

Florent de Dinechin, Patrice Quinton, Tanguy Risset
IRISA
Rennes, France

## Abstract

*This paper presents extensions to* ALPHA, *a language based upon the formalism of affine recurrence equations (*AREs*). These extensions address the need for parametric and structured systems of such AREs. Similar to, but more general as the* map *operator of classical functional languages, the* ALPHA *structuration techniques provide a dense and powerful description of complex systems referencing each other. Such structured systems of AREs may be interpreted as (or translated into) sequential function calls, hierarchical hardware description, or any SIMD flavour of structured programming. With the help of examples, we give an overview of these techniques, and their substitution semantics based on the homomorphic extension of convex polyedra and affine functions.*

## Introduction

The ALPHA Language [3, 6] is a strongly typed functional language which embodies the formalism of *affine recurrence equations* (AREs) first introduced by Karp, Miller and Winograd [2]. An environment has been developped around this language to provide a set of automatic and semi-automatic tools to analyze or transform systems of AREs expressed as ALPHA programs. These tools include the static analysis of the data domains and dependences, the pipelining (or localization) of computations, their space/time reindexing, and others. Using these tools, the high-level description of a regular problem may be transformed, in a correct proven manner, into an implementation-level description such as a VLSI regular array [1], imperative code [5], or a collection of loop nests aimed at a massively parallel architecture [4].

We present here an extension to the language allowing the parametrization and structuration of ALPHA programs into subsystems of AREs. In addition to the possibility of parametrizing ALPHA programs, we introduce in the language the usual structuration of a program into functions, but also more complex program structures in a very straightforward and regular manner. These poweful structures rely heavily on the *homomorphic dimension extension* of polyedra and affine functions, and thus are specific to systems of AREs, but they may be interpreted in terms of loop nests in imperative languages, or as an extension to the *map* operator of functional languages.

The paper is organized as follows : we first briefly introduce the language, then we present parametric ALPHA and its simplest structuration. We then describe the principles of structuration with homomorphic extension. The last section presents the detailed semantics of structured ALPHA.

## 1 The ALPHA language

The purpose of this section is only to introduce the principles and syntax of the language to the non-familiar reader. An extensive presentation may be found in [3], and [7] will provide the interested reader with a complete description of the language in its current version.

**Basics** Unlike most functional language, the dominant data structure in ALPHA is not the list but the *polyedral data array.* Such an array is a mapping from some *convex polyedron* of some integer space $\mathbb{Z}^n$ to a scalar value space (integer, boolean or real).

A convex polyedron $P$ is a convex subspace of $\mathbb{Z}^n$ bounded by a finite set of hyperplanes. In other terms, $P$ is the intersection of a finite family of closed linear halfspaces and may be specified by a system of constraints :

$$D = \{\mathbf{z} \in \mathbb{Z}^n \mid A\mathbf{z} \geq \mathbf{b}\}$$

**Systems** A standard ALPHA program has the overall syntax :

```
system <system-name> ( <input-var-declarations> )
          returns ( <output-var-declarations> ) ;
var <local-var-declarations> ;
let
   <equations>
tel;
```

The *system* is a mapping from *input variables* to *output variables* defined by a set of *equations* relating input, output and *local* variables. ALPHA is an equational language, and thus obeys *the single assignment rule* : there is at most one equation defining each variable.

**Variables** ALPHA is strongly typed, and each variable is declared with the syntax :

```
<var-name> : <domain> of <type> ;
```

The *type* may be either *integer*, *real* or *boolean*. The *domain* may be any convex integer polyedron. The syntax of a domain is :

```
{<indices-list> | <equations/inequations-list>}
```

where the identifiers appearing in the equations and inequations are those declared in the index list. Thus the index names are unsignificant, as their scope is restricted to the space between the curly braces.

*Example* : `B:{i,j| 1<=i<=10; i<=j} of boolean;` declares a boolean triangular matrix;

**Expressions** One may build expressions in ALPHA. Their terminals are either variables or constants (which are defined over the domain $\mathbb{Z}^0$). The operators are either *pointwise* operators (relating the values) or *spatial* ones (transforming the domains).

An expression also describes a polyedral data array, and thus has a type and a domain inherited from its subexpressions.

**Pointwise operators** They are the ALPHA generalization of the classical scalar operators. As an example, we describe the pointwise addition : if `E1` and `E2` are two expressions of the same dimension, then `E1+E2` is an expression of the same dimension whose domain is the intersection of the domains of `E1` and `E2`, and whose value in each point is the sum of the values of `E1` and `E2` at this point.

Other pointwise operators include most unary and binary arithmetic operators, comparison operators, logical connectors, plus the ternary `if then else` operator.

**Spatial operators** The *case* operator allows the piecewise definition of an expression by several subexpressions defined over disjoint domains. Its syntax is :

```
case <exp>; ...; <exp>; esac
```

It is an expression whose domain is the convex hull of the union of the domains of the subexpressions, and whose value at one point is the only value defined at this point.

It usually needs the *restriction* operator, which restricts the inherited domain of an expression to a given subdomain. Its syntax is :

```
<domain> : <exp>
```

Finally, the *dependence* operator establishes an affine mapping from the points of a domain to the points of another domain. Its syntax is :

```
<exp> . <affine-function>
```

If we call $f$ the affine function, this is an expression whose domain is the preimage by $f$ of the domain of *exp*, and whose value at a point $\mathbf{z}$ is the value of *exp* at the point $f(\mathbf{z})$. The syntax of affine functions is :

```
( <index-list> -> <affine-expression-list> )
```

In most cases, the dependence operator may be rendered with an easier-to-read array notation (as demonstrated in the example below).

**Example** The following example program describes in ALPHA the classical matrix-matrix product defined as :

$$R_{i,j} = \sum_{k=1}^{8} a_{ik}.b_{kj}$$

```
system MatMat ( A,B: {i,j| 1<=i,j<=8} of real )
        returns( R: {i,j| 1<=i,j<=8} of real );
var c : {i,j,k|  1<=i,j,k<=8} of real;
let
  c[i,j,k] =
    case
      {|k=1}  : 0[];
      {|k>1}  : A[i,k] * B[k,j]  + c[i,j,k-1];
    esac;
  R[i,j] = c[i,j,8];
tel;
```

Remark that this example uses the array notation of ALPHA [7], where the referential locality of the indices is lost to improve readability. Here is the standard notation of the equation defining `c` :

```
c = case
    {i,j,k | k=1}: 0.(i,j,k-> );
    {i,j,k | k>1}: A.(i,j,k->i,k) * B.(i,j,k->k,j)
                   + c.(i,j,k->i,j,k-1);
    esac;
```

Also remark that the last equation, defining `R`, may be read as : *for all* $(i, j)$ *in the range defined at the declaration of* `R`, `R[i,j]` *equals* `c[i,j,8]`. There is no order of iteration on `i` or `j` specified in this equation.

Such an order will depend on the dependences present in the equation (see the equation defining `c`) and on the interpretation aimed at (loop nest for sequential or parallel machine, hardware).

**The ALPHA environment** Currently developped on top of the MATHEMATICA environment, it offers a wide set of automatic and semi-automatic transformations of programs in an interactive manner. It also provides tools to translate subsets of ALPHA into C and soon VHDL, and to derive VLSI designs of parallel regular architectures.

## 2 The extensions

### Parametric ALPHA

The first extension to the language allows us to have *size parameters* appearing in the language.

As an example, here is a general matrix-matrix product :

```
system MatMat:{M,N,P| M>0; N>0; P>0 }
              (A:{i,k| 1<=i<=M; 1<=k<=N} of real;
               B:{k,j| 1<=k<=N; 1<=j<=P} of real)
       returns(R:{i,j| 1<=i<=M; 1<=j<=P} of real);
var c:{i,j,k| 1<=i<=M; 1<=k<=N; 1<=j<=P} of real;
let
  c[i,j,k] =
    case
      {|k=1}  : 0[];
      {|k>1}  : A[i,k] * B[k,j]  + c[i,j,k-1];
    esac;
  R[i,j] = c[i,j,N];
tel;
```

We declare here in the header of the system three size parameters as a *parameter domain*. These parameters may then appear anywhere in the system where indices are allowed.

The meaning of such a parameter domain is simply to extend all the domains and all the dependences with these parameters. For example, in this system parametrized by

`{M,N,P| M>0; N>0; P>0}`

the declaration domain of `A`, written as

`{i,k| 1<=i<=M; 1<=k<=N}`

is actually a syntactic shortcut for

`{i,k,M,N,P| 1<=i<=M; 1<=k<=N; M>0; N>0; P>0}`

which is referentially closed. Similarly, the last dependance function, written `(i,j->i,j,N)`, is actually a shortcut for `(i,j,M,N,P->i,j,N,M,N,P)` which is standard ALPHA. This process is called *right homomorphic extension* of domains or affine functions and will be formalized in section 3.

Complex conditions (any affine equation or inequation) may be specified in the parameter domain and also between the parameters and the indices. It will be shown later that these condition can be statically verified, with warnings to the user when one of them is violated.

Since a parametrized ALPHA system may be straightforwardly rewritten as a standard ALPHA one (where all the dimensions are greater), we ensure that all the theoretical framework of the language remains untouched by this extension. Therefore, we may now focus the structuration of ALPHA into subsystems of AREs.

### Simple structuration

**An example** To introduce the syntax, we use the previous matrix product to perform the square of a matrix.

```
system SquareMat:{N | N>0}
                 (X: {i,j| 1<=i,j<=N} of real)
       returns   (S: {i,j| 1<=i,j<=N} of real);
  let
     use MatVect[N,N,N] (X,X) returns (S);
  tel;
```

The notation `MatVect[N,N,N]` gives the values of the size parameters `N,M,P` of the subsystem `MatVect` as an affine function of the size parameters of the caller (here the function $N \rightarrow N, N, N$).

**Substitution** We see that a **use** statement appears at the same level as an equation (of course it may be surrounded by equations or other **use** statements between the **let** and the **tel**). In fact, the semantics of such **use** statement is simply a straightforward substitution with the body of the sub), with some additional work to handle the input/output passing, as illustrated in the following program which is equivalent to the previous by definition of this semantics :

```
system SquareMat:{N | N>0}
                 (X: {i,j| 1<=i,j<=N} of real;
                  Y: {i,j| 1<=i,j<=N} of real)
         returns (Z: {i,j| 1<=i,j<=N} of real);
  var
   --variables added for substitution of MatVect
    MatVect_1_A:{i,j| 1<=i<=N; 1<=j<=N} of real;
    MatVect_1_B:{i,j| 1<=i<=N; 1<=j<=N} of real;
    MatVect_1_R:{i,j| 1<=i<=N; 1<=j<=N} of real;
    MatVect_1_c:{i,j,k| 1<=i<=N; 1<=j<=N;
```

```
                         0<=k<=N}  of real;
let
  -- Inputs
  MatVect_1_A = X;
  MatVect_1_B = X;
  -- Body of Matvect
  MatVect_1_c[i,j,k] =
    case
      {|k=0}: 0[];
      {|k>0}: MatVect_1_A[i,k]*MatVect_1_B[k,j]
                + MatVect_1_c[i,j,k-1];
    esac;
  MatVect_1_R[i,j] =  MatVect_1_c[i,j,N];
  -- Outputs
  Z = MatVect_1_R;
tel;
```

Note that, in addition to inlining the equations of the subsystem, its input, output and local variables have been declared as local, with some renaming to avoid name conflicts. Inputs and outputs passing are handled thanks to additional equations. Thus the actual inputs may be any ALPHA expression, but the actual output must be variables. Moreover, all the domains and affine functions appearing in the subsystem have been properly modified to take into account the parameter assignation. These modifications are based on basic domain operations (intersection, preimage) and affine function composition. They will be described in full detail in section 3.

### Extension of dimension

Now, in most practical cases, a system is not designed to be used with its declared I/Os, but with I/Os of greater dimension. We usually want to use *regular collections* of instances of a subsystem. For example, a matrix operation, when used in a real-time application, shall actually perform on collections of matrices indexed by the time : although the I/Os corresponding to a matrix are two-dimensional in the declaration of the system, the actual I/Os will be three-dimensional (the two dimensions of the matrix, and the time dimension). As another example, in an image-processing application, we will define a subsystem performing some computations over a window surrounding a pixel, and use this subsystem for each pixel of the image, that is with I/Os regularly extended by the two dimensions of the picture.

Thus, what is needed in real applications is an operator similar to the *map* operator of functional languages, i.e. applying to each point of a given collection the functionality of the subsystem. In classical functional languages, these collections are lists, for lists are the basic data structure of such languages. In ALPHA, on the other hand, they shall obviously be convex polyedra.

Let us now describe the general features of such a dimension extension process with the help of the following toy example. We define a subsystem which performs the sum of the elements of a vector, with one parameter which is the size of the vector, and we use it in another system which performs a linear collection of of such sums, for all the rows of some non-rectangular matrix (figure 1)

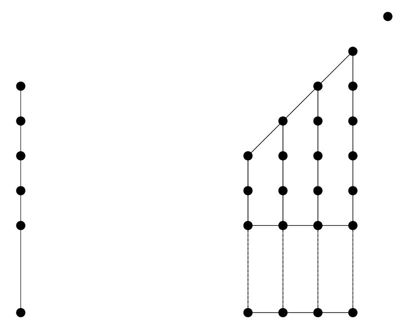

Figure 1: Toy exemple of subsystem dimension extension

```
system Sum:{N| N>0} ( V : {j | 1<=j<=N} of real)
            returns ( R : real);
var S:{j | 1<=j<=N} of real;
let
  S[j] =
    case
        {| j=1} : V[j];
        {| j>1} : V[j] + S[j-1];
    esac;
  R = S[N]
tel;
```

```
system Toy (A: {i,j|1<=i<=4; 1<=j<=i+2} of real)
   returns (X: {i|1<=i<=4} of real);
let
  use {i|1<=i<=4} Sum[i+2] (A) return(X);
tel;
```

Consider the line containing the **use**. The *extension domain*, here {i|1<=i<=4}, may be any convex polyedron. All the actual I/Os shall have their dimension greater than the formal ones by the dimension of this extension domain. The corresponding indices are the leftmost ones (*leftmost homomorphic extension*), and the projections of the actual I/Os on these indices must of course contain this domain. The parameter assignation may be any affine function of the callers' parameter, but also of the extension indices, as illustrated in this example. This feature is very powerful in real applications.

The semantics is also a substitution, but with all the domains and affine functions processed according to the extension of dimension. We briefly explain here this process; a complete and formal description will be given is section 3.

- The dimensions of all the domains appearing within the subsystem have to be increased. In our example, the extension domain is 1-dimensional, therefore the 1-dimensional input variable `V` of `Sum` become 2-dimensional, and its zero-dimensional output variable becomes 1-dimensional. The same process is applied to all the domains appearing in the subsystem.
- The affine functions appearing in the subsystem are similarly modified to handle the extra dimensions. Here there is only one affine function hidden behind the array notation : `(j->N)`, and it shall be transformed into `(i,j->i,N)`

Here is our `Toy` system after inlining of its subsystem :

```
system Toy (A: {i,j|1<=i<=4; 1<=j<=i+2} of real)
   returns (X: {i|1<=i<=4} of real);
var
    Sum_1_V: {i,j| 1<=i<=4; 1<=j<=i+2} of real);
    Sum_1_R: {i| 1<=i<=4} of  real);
    Sum_1_S: {i,j| 1<=i<=4; 1<=j<=i+2} of  real;
let
  Sum_1_V = A;
  S[i,j] =
    case
        {| j=1} : V[i,j];
        {| j>1} : V[i,j] + S[i,j-1];
    esac;
  R[i] = S[i,N]
  Sum_1_X = R;
tel;
```

# 3 The semantics of system inlining

We present here the detailed semantics of system **use**.

## Notations

Let us define a few notations :

- let $P_{caller}$ be the parameter space of the caller, $d_{caller}$ its dimension.
- let $P_{sub}$ be the parameter space of the subsystem, $d_{sub}$ its dimension.
- let $e$ be the number of extension indices, and $D_{ext}$ be the extension domain. Note that this domain is parametrized by $P_{caller}$, thus in the case of a simple **use** $D_{ext} = P_{caller}$, whereas in case of dimension extension $dim(D_{ext}) = d_{caller} + e$.
- let $f : D_{ext} \rightarrow P_{sub}$ be the affine parameter assignation.

## Validity of a use statement

Defining the semantics of the **use** in terms of substitution has the advantage that all the type checking over the inputs and outputs is defined by the semantics of non-structured ALPHA.

Except for this input/output type checking, the validity of a **use** statement only resumes to checking that :

- The subsystem exists.
- The number of actual inputs (resp. outputs) appearing in the **use** statement is equal to the number of formal inputs (resp. outputs) declared in the header of the subsystem.
- Parameter assignation compatibility : the dimensions of $f$ are $d_{caller} + e \times d_{sub}$.
- The values given to parameters by $f$ belong to the parameter space of the subsystem, that is :

  $$f(D_{ext}) \subset P_{sub}$$

  This check is difficult and is currently not implemented[1]

Note that the strong static semantics of the language, which allowed to analyze a program in order to check that all the variables are well defined over their full domain, now makes it possible to check indiretly that the conditions on the parameters stated within the parameter domains are fullfilled by the **use** statements. This feature as already proven invaluable in developping large applications (an adaptive Singular Values Decomposition algorithm is curently beeing

[1] The best approach seems to perform this check only in the special case of a caller system without parameter (physical implementation are modelized by such systems). In this case, this check reduces to checking that a point belongs to a domain, which is very simple, and may be performed recursively over all the subsystems of this system.

implemented in ALPHA). The parameter domains, if restricted enough, are in practice similar to *preconditions* over iteration loops and their bounds in the corresponding sequential program, as far as these conditions are expressible in an affine way.

There is a real interface problem, however, concerning the output of the static analysis : it is sometimes very difficult to infer the cause of the error from the error message, and there are intrisic reasons for it (which won't be detailed here because of the lack of space).

## Substitution

An ALPHA system containing a valid **use** statement is equivalent to the system in which this statement has been replaced with :

- Additional declarations of all the input, output and local variables of the subsystem (with renaming if needed). The declaration domains are transformed according to the parameter assignation (see below).
- Input equations of the type :
  `FormalInput = ActualInput`
- The body of the subsystem. Restriction domains and affine functions appearing in this body are transformed according to the parameter assignation (see below)
- Output equations of the type :
  `ActualOutput = FormalOutput`

Remark that the added input and output equations carry all the type checking over inputs and outputs.

## Domain processing

Let $D$ be a domain appearing either in the declarations or in the body of the subsystem, $n + d_{sub}$ its dimension. Let

$$n \otimes f \quad : \quad \mathbb{Z}^{n+e+d_{caller}} \longrightarrow \mathbb{Z}^{n+d_{sub}}$$

the affine function defined by the following block matrix (left homomorphic extension of affine functions) :

$$n \otimes f = \left( \begin{array}{c|c} I_n & 0 \\ \hline 0 & f \end{array} \right)$$

($I_n$ is the identity of size $n$.)

Let us also define the left homomorphic extension of a domain :

$$n \otimes \{i_1, ..i_p \mid I_1, ..I_k\} = \{j_1, .., j_n, i_1, ..i_p \mid I_1, ..I_k\}$$

The domain $D'$ appearing in the inlined system is then :

$$D' \quad = \quad (n \otimes f)^{-1}(D) \ \cap \ (n \otimes D_{ext})$$

**Remarks**

- $D'$ is the preimage of an ALPHA domain, and as such is a domain. Its dimension is $n + e + d_{caller}$.
- Remembering that the parameter space within $D$ was the universe polyhedron of dimension $d_{sub}$, this formula ensures that the parameter space appearing in $D'$ is also the universe polyedron

## Affine function processing

Let $a$ be an affine dependence function appearing in the body of the subsystem. There are two cases, due to the fact that void dependence functions, typically used along with scalar constants, don't carry the parameter identity :

- if $a$ is a void function :

  $$a \quad : \quad \mathbb{Z}^{n+d_{sub}} \longrightarrow \mathbb{Z}^0$$

  The dependence function appearing after inlining is then

  $$a' \quad : \quad \mathbb{Z}^{n+e+d_{caller}} \longrightarrow \mathbb{Z}^0$$

  built as follows :

  $$a' = a \circ (n \otimes f)$$

- in the general case, $a$ is a dependence function with a parameter identity :

  $$a \quad : \quad \mathbb{Z}^{n+d_{sub}} \longrightarrow \mathbb{Z}^{m+d_{sub}}$$

  $$a = \left( \begin{array}{c|c} \multicolumn{2}{c}{a_r} \\ \hline 0 & I_{d_{sub}} \end{array} \right)$$

  The dependence function appearing after inlining is then

  $$a' \quad : \quad \mathbb{Z}^{n+e+d_{caller}} \longrightarrow \mathbb{Z}^{m+e+d_{caller}}$$

  build as follows : first, we build

  $$a_r \quad : \quad \mathbb{Z}^{n+d_{sub}} \longrightarrow \mathbb{Z}^m$$

  by removing this parameter identity from $a$. Then we build

  $$a'_r = a_r \circ (n \otimes f) \quad : \quad \mathbb{Z}^{n+e+d_{caller}} \longrightarrow \mathbb{Z}^m$$

  and $a'$ is just $a'_r$ extended with an identity of the rightmost indices, which are the the $e$ extension indices and the $d_{caller}$ parameters.

## Conclusion

The extensions to the language presented here mainly adress two needs :

The first need is the *reuseability* of a system, and corresponds to

- *function application* in functional languages,
- *procedure calls* in Pascal-like imperative language,
- *component instanciation* in hardware description languages like VHDL.

In terms of recurrence equations, we want a means to give a name to a set of equations, which we then call a *system*, and then use this name as a shortcut for the whole system. We don't need to enumerate here the advantages of structured programming in terms of readability and development costs.

Such structuration has to include some kind of *parametrization* of the systems.

The second need is specific to the world of recurrence equations. We want to be able to express a collection of instances of a system in one simple expression.

- If the subsystem is viewed as a function of a functional language, this corresponds to the *map* operator. It is however more powerful : where the *map* operator applies a function to all the terms of a list, we may apply it to all the terms of any integer polyedron.
- If the subsystem is viewed as a procedure in an imperative language, this corresponds to a procedure call within an affine loop nest [5, 4].
- If we compare ALPHA to VHDL, this correspond to nested `for` and `generate` loops.

The formalism of AREs allows to express such complex structuration in a very simple and uniform manner.

These power of description and interpretation has still to be exploited. Future work include the definition of a subset of ALPHA, called ALPHARD, to describe structured hardware. This will allow to design tools converting high-level ALPHA into ALPHARD, and ALPHARD into silicon or FPGA, through VHDL and the MadMacs tool for synthesis of regular arrays developped at IRISA. On the language side, we shall try to improve the interpretability of the outputs from the static analysis tools, and complete the process of upgrading all the existing program transformations to handle structured ALPHA.

## References


[1] C. Dezan, E. Gautrin, H. Le Verge, P. Quinton, and Y. Saouter. "Synthesis of systolic arrays by equation transformations." In *ASAP'91*, Barcelona, Spain, September 1991. IEEE.

[2] R.M. Karp, R.E. Miller, and S. Winograd. "The organization of computations for uniform recurrence equations." *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[3] C. Mauras. "Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones". Thèse de l'Université de Rennes 1, IFSIC, December 1989.

[4] P. Quinton, S. Rajopadhye, and D. Wilde. "Derivation of data parallel code from a functional program." In *IPPS*, Santa Barbara, USA, April 1995.

[5] S. Rajopadhye and D. Wilde. "The naive execution of affine recurrence equations." In *ASAP*, Strasbourg, France, July 1995.

[6] H. Le Verge, C. Mauras, and P. Quinton. "The ALPHA language and its use for the design of systolic arrays." *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[7] Doran K. Wilde. "The alpha language." Internal Report 999, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Januar 1994.