The ALPHA language

Doran Wilde

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The ALPHA Language

Doran K. Wilde

N° 2295

May 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

The ALPHA Language

Doran K. Wilde*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n ° 2295 — May 1994 — 21 pages

Abstract: This report is a formal description of the ALPHA language, as it is currently implemented. ALPHA is a strongly typed, functional language which embodies the formalism of systems of affine recurrence equations. In this report, ALPHA language constructs are described, and denotational and type semantics are given. The theorems which are the basis for doing transformations on an ALPHA program are stated. And finally, the syntax and semantics of ALPHA are given.

Key-words: recurrence equations, systolic arrays, functional languages, hardward design languages

(Résumé : tsvp)

*email: wilde@irisa.fr This work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632 and by NSF Grant No. MIP-910852.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)

Le Language ALPHA

Résumé : Ce rapport traite de la description formelle de l'implémantation courante du langage ALPHA . ALPHA est un langage fonctionnel fortement typé et basé sur le formalisme des équations récurrentes affines. Sont présentés ici, les constructions du langage ainsi que sa sémantique dénotationnelle et sa sémantique des types. Un point est également consacré aux théorèmes constituant la base des transformations de programmes ALPHA . Ces différents aspects sont complétés par la présentation de la syntaxe et des sémantiques du langage.

Mots-clé : équations récurrentes, réseaux systoliques, langages fonctionnel, langage de description de matériel.

The ALPHA language was developed by Mauras [7] at IRISA in Rennes, France. A large part of this document is my interpretation of Mauras' thesis, both in a linguistic sense and in a technical sense.

ALPHA was a product of research in regular parallel array processors and systolic arrays. The ALPHA language is able to formally represent algorithms which have a high degree of regularity and parallelism such as systolic algorithms as defined by Kung [4]. It is based on the formalism of recurrence equations which have been often used in various forms by several authors [10, 2, 8] all of which were based on the introduction of the notion of uniform recurrence equations by Karp, Miller and Winograd [3].

1 Systems of Affine Recurrence Equations

ALPHA is based on the formalism of systems of affine recurrence equations. The definitions in this section review the basic concepts of systems of affine recurrence equations and are taken primarily from the work of Rajopadhye and Fujimoto [10], Yaacoby and Cappello [13], Delosme and Ipsen [1], and Quinton and Van Dongen [9].

In describing systems of affine recurrence equations, there are two equally valid points of view which can be taken. The first is a purely functional point of view in which every identifier is a *function*. A recurrence equation defines a function on the left hand side in terms of the functions on the right hand side. Alternately, each identifier can be thought of as a *single assignment variable* and equations equate the variable on the left hand side to a function of variables on the right. In the following presentation, I take the second point of view.

Definition 1 *(Recurrence Equation)*
A **Recurrence Equation** *over a domain $\mathcal{D}$ is defined to be an equation of the form*

$$f(z) = g(\ f_1(I_1(z)), f_2(I_2(z)), \cdots, f_k(I_k(z))\ )$$

where

- *$f(z)$ is a variable indexed by z and the* left hand side *of the equation.*
- *$z \in \mathcal{D}$, where $\mathcal{D}$ is the (possibly parameterized) domain of variable f.*
- *$f_1, \cdots, f_k$ are variables found on the* right hand side *of the equation. They may include the variable f, or multiple instances of any variable.*
- *I_i are index mapping functions which map $z \in \mathcal{D}$ to $I_i(z) \in \mathcal{D}_i$, where $\mathcal{D}_1, \cdots \mathcal{D}_k$ are the (possibly parameterized) domains of variables $f_1, \cdots, f_k$, respectively.*
- *g is a strict single-valued function whose complexity is $\mathcal{O}(1)$ defining the* right hand side *of the equation.*

A variation of an equation allows f to be defined in a finite number of disjoint "cases" consisting of convex subdomains each having the same left hand side as follows:

$$f(z) = \begin{cases} z \in D_1 & \Rightarrow & g_1(\ldots f_1(I_1(z)) \cdots) \\ z \in D_2 & \Rightarrow & g_2(\ldots f_2(I_2(z)) \cdots) \\ \quad \vdots & & \end{cases} \tag{1}$$

where the domain of variable f is $\mathcal{D} = \bigcup_i D_i$ and $(i \neq j) \rightarrow (D_i \cap D_j = \emptyset)$

Definition 2 *(Dependency)*
For a system of recurrences, we say that a variable f_i at a point $p \in D_i$ (directly) **depends on** *variable f_j at q, (denoted by $p_i \mapsto q_j$), whenever $f_j(q)$ occurs on the right hand side of the equation defining $f_i(p)$. The transitive closure of this is called the* **dependency relation**, *denoted by $p_i \rightarrow q_j$.*

Definition 3 *(Dependency Function)*
Given a dependency $p_i \mapsto q_j$, and a function I mapping p_i to q_j then I is called the **dependency function**. *The dependency can be rewritten in terms of the mapping function in the form $p_i \mapsto I(p_i)$.*

Definition 4 *(Affine Recurrence Equation)*
A recurrence equation of the form defined above is called a **Uniform Recurrence Equation** *(URE) if all of the dependency functions (index mapping functions) I_i are of the form $I(z) = z + \mathrm{b}$, where b is a (possibly parameterized) constant n-dimensional vector. It is called an* **Affine Recurrence Equation** *(ARE) if $I(z) = \mathrm{A}z + \mathrm{b}$, where A is a constant matrix, and b is a (possibly parameterized) constant n-vector.*

Definition 5 *(System of Affine Recurrence Equations, or SARE)*
A **system** *of recurrence equations is a set of m such equations, defining the functions $f_1 \ldots f_m$ over domains $\mathcal{D}_1 \cdots \mathcal{D}_m$ respectively. The equations may be (mutually) recursive. Variables are designated as either input, output, or local variables of the system. Each variable (which is not a system input) appears on the left hand side of an equation once and only once. Variables may appear on the right hand sides of equations as often as needed.*

Since there is a one to one correspondence between (non-input) variables and equations, the two terms are often used interchangeably. Such equations serve as a purely functional definition of a computation, and are in the form of a **static program**— a program whose dependency graph can be statically determined and analyzed (for any given instance of the parameters). Static programs require that all g_i be strict functions and that any conditional expressions be limited to linear inequalities involving the indices of the left hand side variable. By convention, it is assumed that, boundary values (or input values) are all specified whenever needed for any function evaluation.

2 ALPHA : An applicative equational language

An equational language is a natural way to describe a SARE. When thinking about algorithms such as those used in signal processing or numerical analysis applications, a person naturally thinks in terms of mathematical equations. Mathematical notation has evolved over the centuries and obeys certain basic rules. 1. Given a function and an input, the same output must be produced each time the function is evaluated. If the function is time varying, then time must be a parameter to the function. Turner [11] uses the term *static* to describe this property. 2. Consistency in the use of names: a variable stands for the same value throughout its scope. This is called *referential transparency*. An immediate consequence of referential transparency it that

equality is substitutive — equal expressions are always and everywhere interchangeable. This property is what gives mathematical notation its deductive power [11]. ALPHA shares both of these properties with mathematical notation: it is static and it is referentially transparent.

Using a language that shares the properties of mathematical notation eases the task of representing an algorithm as a program. Furthermore, such a method of describing algorithms has some interesting properties, as has been discovered by users of ALPHA . An equation specifies an assertion on a variable which must always be true. Reasoning about programs can thus be done in the context of the program itself, and relies essentially on the fact that ALPHA programs respect the **substitution principle**. This principle states that an equation $X = Expression$ specifies a total synonymy between the variable on the left hand side of the equation and the expression on the right hand side of the equation. Thus any instance of a variable on the left hand side of any equation may be replaced with the right hand side of its definition. Likewise, any sub-expression may be replaced with a variable identifier, provided that an equation exists, or one is introduced, in which that variable is defined to be equal to that sub-expression.

Alpha has other properties which should be mentioned.

- ALPHA equations are unordered. The only ordering of computations is that which is implied by data dependencies.
- ALPHA is a single assignment language. Each variable element can only ever hold a single value which is a function of system inputs.
- Every ALPHA program is a *static program*. This makes ALPHA statically analyzable.
- ALPHA does not support any notion of global variables. An execution of a system defined in ALPHA only affects the outputs of the system— there are never any side effects. This is a necessary characteristic of a referentially transparent language. A system without outputs is without purpose, and is dead code.
- ALPHA is strongly typed. Each variable must be predeclared with both a domain and data type attributed to that variable.

At this point, it can be stated that ALPHA adopts the classical principles of a functional language which is structured and strongly typed. An ALPHA program defines a function between domain based input and output variables. This notion of a function is embedded in the definition of the ALPHA *system* construct.

3 Alpha System Declarations

At the top level, an ALPHA program consists of a *system declaration* consisting of:

1. A system name
2. A list of input variable declarations
3. A list of output variable declarations

which is followed by the *system definition*, consisting of:

1. A list of local variable declarations

2. A list of equations defining output and local variables.

This information appears in the following format in ALPHA syntax.

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

Declaration

Definition

4 Variable Declarations

ALPHA is strongly typed and each variable is declared with a type of the form (< `domain` > $\longrightarrow$ < `datatype` >). As in a classical typed system, each variable must be declared with its type and then usages of that variable must conform to its declared type.

A variable can be thought of as a function mapping integer points in a domain to values in the data type set:

$$X \ : \ z \in < \texttt{domain} > \longmapsto X[z] \in < \texttt{datatype} >$$

The observation that a variable is also a function ties together the variable and functional views of recurrence equations as discussed in section 1. The type of a variable may thus be thought of as a prototype of the function of the variable, giving both the domain and range of the function.

The `<datatype>` is one of the three base data types: `integer`, `boolean`, and `real`. All data types are assumed to be infinite precision and include the special value *error* which means a value which is uncomputable.

Each variable in ALPHA is associated with a fixed domain and has a value associated with each point in that domain. Scalar variables may also be declared, and hold single value associated with the point in the trivial domain $\mathcal{Z}^0$. The following syntax forms for a variable declaration are supported in ALPHA , (the last three are scalar declarations):

```
<var-list> : <domain> of integer;
<var-list> : <domain> of boolean;
<var-list> : <domain> of real;
<var-list> : integer;
<var-list> : boolean;
<var-list> : real;
```

4.0.1 Type Checking

An important advantage of being a strongly typed language is that certain static analysis methods can be performed which detect inconsistencies in the way a variable is declared and the way it is used. ALPHA extends the classical type check to a much more powerful check which uses polyhedral computation and permits the verification that a variable is defined and used in manner consistent with its declaration and that each and every element in the variable has exactly one value associated with it. These syntactic checks are able to detect a large class of programming and logic errors.

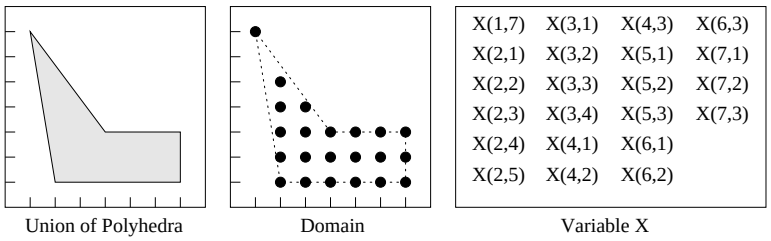

Figure 1: Comparison of Polyhedra, Domain, and Variable

5 Domain variables

Each variable used in an ALPHA program is a function over a domain in $\mathcal{Z}^n$. When specifying a system of affine recurrence equations, unions of convex polyhedra are used to describe the domains of computation of system variables.

Definition 6 *A* **polyhedron**, $\mathcal{P}$ *is a subspace of* $\mathcal{Q}^n$ *(rational space*[1]*) bounded by a finite number of hyperplanes.*

Alternate definition:
$\mathcal{P}$ *= intersection of a finite family of closed linear halfspaces* $\{x \mid ax \geq c\}$ *where* a *is a non-zero row vector,* c *is a scalar constant.*

The set of solution points which satisfy a mixed system of linear constraints form a polyhedron $\mathcal{P}$ and serve as the *implicit definition* of the polyhedron

$$\mathcal{P} = \{x \mid Ax = b,\ Cx \geq d\} \tag{2}$$

given in terms of equations (rows of A, b) and inequalities (rows of C, d), where A, C are matrices, and b, d and x are vectors. All polyhedra are convex.

Some important polyhedral operators are used in the semantics of ALPHA. They are defined here:

$$\begin{aligned} Image(\mathcal{P},T) &= \{x \mid x = Ty, y \in \mathcal{P}\} \\ Preimage(\mathcal{P},T) &= \{x \mid \exists y \in \mathcal{P} : y = Tx\} \\ \mathcal{P}_1 \cap \mathcal{P}_2 &= \{x \mid x \in \mathcal{P}_1 \text{ and } x \in \mathcal{P}_2\} \\ \mathcal{P}_1 \cup \mathcal{P}_2 &= \{x \mid x \in \mathcal{P}_1 \text{ or } x \in \mathcal{P}_2\} \\ \mathcal{P}_1 \backslash \mathcal{P}_2 &= \{x \mid x \in \mathcal{P}_1 \text{ and not } x \in \mathcal{P}_2\} \end{aligned}$$

These operations are all implemented in the polyhedral library [12] which is used extensively in the ALPHA system.

Whereas a polyhedron is a region containing an infinite number of rational points, a *polyhedral domain*, as the term is used in this report, refers to the lattice of integral points $\mathcal{Z}^n$ which are inside a polyhedron (or union of polyhedra). Figure 1 illustrates this difference.

Definition 7 *A polyhedral* **domain** *of dimension* n *is defined as*

$$\mathcal{D} : \{z \mid z \in \mathcal{Z}^n, z \in \mathcal{P}\} = \mathcal{Z}^n \cap \mathcal{P} \tag{3}$$

[1] polynomials may be defined in real space also

where $\mathcal{P}$ is a union of convex polyhedra of dimension n.

In affine recurrence equations of the type considered here and in the ALPHA language, every variable is declared over a domain as just described. Elements of a variable are in a one-to-one correspondence with points in a domain. Again, figure 1 illustrates this. Here, we formalize the definition of a variable.

Definition 8 *A* **variable** X *of type* "datatype" *declared over a domain* $\mathcal{D}$ *is defined as*

$$X = \{\ X[z]\ :\ X([z]) \in \text{datatype},\ z \in \mathcal{D}\ \} \tag{4}$$

where $X[z]$ is the element of X corresponding to the point z in domain $\mathcal{D}$ and "datatype" *is either* `integer`*,* `boolean`*, or* `real`*.*

The ALPHA syntax for representing a domain which consists of a single polyhedron is as follows:

```
{ <index-list> | <constraint-list> }
```

More complicated domains can be built up using the three domain operators: union, intersection, and difference. Union is written by combining two domains with a vertical bar: `{ ... } | { ... }`, intersection is written with an ampersand: `{ ... } & { ... }`, and difference with an ampersand-tilde: `{ ... } &~{ ... }`

Some examples of domains written in the ALPHA syntax are given below :

```
{ x,y,z | -5 <= x-y <= 5; -5 <= x+y <= 5; z = 2x - 3y }
{ i,j,N | 0 <= i <= N-1; N <= j <= 2N-1 }  -- parameterized domain
{ i,j,k | k = 5 }                          -- a plane in 3 space
{ i,j | 1>=0 }                    -- 2 dimensional universe domain
{ i | 1 = 0 }                     -- 1 dimensional empty domain
{ i,j | i=1; 0<=j<=2} | { i,j | i=3; 1<=j<=5 }  -- union of domains
```

6 Denotational and type semantics

In the following sections, the syntax and semantics of each ALPHA construct is described. For each construct, both the denotational and type semantics are given in terms of functions which take as an argument a syntactic entity which is in the form of an abstract syntax tree. These functions are described in this section.

The type semantics are defines by the two functions Domain(), and Type(). The functions Domain(), and Type() both take a single argument, which is a syntactic entity representing an ALPHA expression, and they return the `<domain>` and `<datatype>` of their arguments respectively. Using these two functions, a static type checker can be made which checks the type semantics of a program.

The denotational semantics are given by the Eval() function which takes as its first argument a syntactic entity, and as its second argument, a point in the domain space of that entity. It then returns the value of the first argument evaluated at the point given by the second argument. The function is usually given recursively in terms of other Eval() functions, the *error* value, and the input() function. The input() function retrieves values from the "system" in an unspecified

manner. (The parameter passing mechanism is unspecified). Using the denotational semantics, an interpreter which "executes" a program, that is, finds values for all of the outputs given the outputs, may be written.

These functions are used to prove theorems relating to the semantic equivalence of ALPHA expressions. Many of these theorems are stated in this report. Their proofs are based the the definition of equivalent ALPHA expressions, which is given here.

Definition 9 *(Equivalent Expressions)*
Two ALPHA *expressions* $\exp_1$ *and* $\exp_2$ *are equivalent if and only if they are of the same type and denotation, as established by the following three equalities :*

$$\begin{aligned} \text{Domain}(exp_1) &= \text{Domain}(exp_2) = \mathcal{D} \\ \text{Type}(exp_1) &= \text{Type}(exp_2) \\ \forall\, z \in \mathcal{D} : \text{Eval}(exp_1, z) &= \text{Eval}(exp_2, z) \end{aligned}$$

7 Equations and Expressions

All local and output variables are defined by a set of equations.

```
var
   X : <domain> of <type>;
   ...
let
   X = <expression_1>;
   ...
   X = <expression_n>;
tel;
```

where :
`X : <domain> of <type>;` is called the *declaration* of X, and
`X =< expression >`$_1$;$\cdots$`X =< expression >`$_n$`;` is called the *definition* of X. The semantics of an equation are as follows:

$$\forall z \in \texttt{<domain>} : \text{Eval}(X, z) = \begin{cases} \text{Eval}(\texttt{<expression>}_1, z) & \text{if } z \in \text{Domain}(\texttt{<expression>}_1) \\ \cdots & \\ \text{Eval}(\texttt{<expression>}_n, z) & \text{if } z \in \text{Domain}(\texttt{<expression>}_n) \\ \textit{error} & \text{otherwise} \end{cases}$$

A procedure, GetInfo, can be defined which retrieves the type and definition information for a variable:

$$\begin{aligned} \text{GetInfo}(X) &= \{< \texttt{domain} >, < \texttt{type} >, < \texttt{kind} >, < \texttt{expression} >\} \\ \text{where} \quad < \texttt{domain} > &\quad \text{is from the declaration of } X \\ < \texttt{type} > &\quad \in \{\text{integer}, \text{boolean}, \text{real}\} \text{ from the declaration of } X \\ < \texttt{kind} > &\quad \in \{\text{input}, \text{output}, \text{local}\} \text{ from the declaration of } X \\ < \texttt{expression} > &\quad \text{is from the definition of } X \end{aligned}$$

The following static type checks can be made

$$\bigcup_i \text{Domain}(\texttt{<expression>}_i) \supseteq \texttt{<domain>}$$
$$\forall i \ : \ \text{Type}(\texttt{<expression>}_i) = \texttt{<type>}$$

Other static checks may also be made:

1. *Declaration rule.* Every variable has one declaration as either an input, output, or local variable.
2. *Definition rule.* Every output and local variable has a definition. Input variables have no definition.
3. *Usage rule.* Every local and input variable appears in the definition of some other variable.

The following theorems apply to equations:

$$\begin{aligned} Var = Exp_1;\ \cdots\ Var = Exp_n; &\iff Var = case(Exp_1, \cdots, Exp_n); \\ Var_1 = Exp_1;\ Var_2 = \cdots Var_1 \cdots; &\iff Var_2 = \cdots(\text{Domain}(Var_1)\ :\ Exp_1)\cdots; \\ \text{if Domain}(Var) \cap \text{Domain}(Exp) = \emptyset &\implies \\ Var = Exp; &\iff < \text{removed} > \end{aligned}$$

An expression in ALPHA is composed of variables, constants, and operations defined in the language. These constructs are described in the following sections.

7.1 Variables

All variables must have a **<domain>** and **<type>** given in the variable declaration and a defining **<expression>** given on the RHS of the variable definition. (Exception: input variables do not have a definition). When a variable is used, the following semantics apply:

$$\begin{aligned} \text{GetInfo}(X) &= \{< \texttt{domain} >, < \texttt{type} >, < \texttt{kind} >, < \texttt{expression} >\} \\ \text{Eval}(X,z) &= \begin{cases} \text{Eval}(\texttt{<expression>}, z) & \text{if } z \in \texttt{<domain>} \text{ AND } < \texttt{kind} > = \text{local or output} \\ \text{input(X,z)} & \text{if } z \in \texttt{<domain>} \text{ AND } < \texttt{kind} > = \text{input} \\ \textit{error} & \text{otherwise} \end{cases} \\ \text{Domain}(X) &= \texttt{<domain>} \\ \text{Type}(X) &= \texttt{<type>} \end{aligned}$$

A second evaluation procedure, Eval2, stores precomputed values for points in X in a table which has been preinitialized with the value *undefined*. The first time each point is evaluated, it is stored in this table. Should a point ever need to be re-evaluated, it is not recomputed, but obtained from the table. This method is more complicated, but also more efficient. The table is accessed using three procedures: Store(X,z,v) which stores value v in the X-table at point z, Get(X,z) which retrieves and returns the value in the X-table at point z, and Valid(X,z) which returns true if the value in the X-table at point z is not *undefined*.

$$\text{Eval2}(X,z) = \begin{cases} \text{Get}(X,z) & \text{if Valid}(X,z) \text{ is True} \\ v = \text{Eval}(X,z), Store(X,z,v) & \text{if Valid}(X,z) \text{ is False} \end{cases}$$

7.2 Constants

Constants (integer, boolean, and real) are defined over the scalar domain $\mathcal{Z}^0$, a zero dimensional domain consisting of a single value. This domain can be extended to a domain of any dimension.

$$\begin{aligned} \text{Eval}(constant, z) &= constant.value \\ \text{Domain}(constant) &= \mathcal{Z}^0 \\ \text{Type}(constant) &= constant.type \end{aligned}$$

Both the value and the type of a constant are extracted from the syntax.

7.3 Pointwise operators

A pointwise operator is an operator which takes arguments defined over some common domain and returns a result defined over this same domain. These operations are a spatial generalization of classical scalar operators which are applied to the domains of their arguments element by element. For example, if

$$X = \begin{matrix} X_{11} & X_{21} & X_{31} \\ X_{12} & X_{22} & \\ X_{13} & & \end{matrix} \qquad Y = \begin{matrix} Y_{11} & Y_{21} & Y_{31} \\ Y_{12} & Y_{22} & \\ Y_{13} & & \end{matrix}$$

then the expression $X - Y$ is computed as:

$$X - Y = \begin{matrix} X_{11} - Y_{11} & X_{21} - Y_{21} & X_{31} - Y_{31} \\ X_{12} - Y_{12} & X_{22} - Y_{22} & \\ X_{13} - Y_{13} & & \end{matrix}$$

Even if the domains of the arguments are not exactly the same, the definition of a scalar binary operator can still be extended to a operate over two domain variables. In general, a pointwise operator *op* is defined over the intersection of the domains of its argument variables as follows for binary operations :

$$\begin{aligned} \text{Eval}(binop(op, Exp_1, Exp_2), z) &= \begin{cases} \text{Eval}(Exp_1, z)\ op\ \text{Eval}(Exp_2, z) \\ \qquad : if\ z \in (\text{Domain}(Exp_1) \cap \text{Domain}(Exp_2)) \\ error\ :\ \text{otherwise} \end{cases} \\ \text{Domain}(binop(\ op\ , Exp_1, Exp_2)) &= \text{Domain}(Exp_1) \cap \text{Domain}(Exp_2) \\ \text{Type}(binop(\ op\ , Exp_1, Exp_2)) &= \text{TypeTable}(op, \text{Type}(Exp_1), \text{Type}(Exp_2)) \end{aligned}$$

and as follows for unary operations :

$$\begin{aligned} \text{Eval}(unop(\ op\ , Exp), z) &= \begin{cases} op\ \text{Eval}(Exp, z) & :\ \text{if } z \in \text{Domain}(Exp) \\ error & :\ \text{otherwise} \end{cases} \\ \text{Domain}(unop(\ op\ , Exp)) &= \text{Domain}(Exp) \\ \text{Type}(unop(\ op\ , Exp)) &= \text{TypeTable}(op, \text{Type}(Exp)) \end{aligned}$$

The type semantics of all of the pointwise operators are defined in table 1.

Operators	Source1 Type	Source 2 Type	→ Destination Type
$+, -, *, div, mod, min, max$	int	int	int
$+, -, *, min, max/$	real	real	real
$-$ (negate)	int/real		int/real
$<, \leq, >, \geq$	int/real	int/real	boolean
$=, \neq$	int/real/boolean	int/real/boolean	boolean
and,*or*, *xor*	int/boolean	int/boolean	int/boolean
not	int/boolean		int/boolean

And for the *if* — *then* — *else* — operation :

Source1 Type	Source 2 Type	Source 3 Type	→ Destination Type
boolean	int/real/boolean	int/real/boolean	int/real/boolean

Table 1: Type Table for Pointwise Operations

8 Domain operators

The domain operators explicitly manipulate the domains of domain based variables. In this section, the *case* operator is presented which allows the piecewise definition of equations. In connection with the *case* operator, the restrict operator is presented which permits the discrimination between different parts of a variable domain. And finally, the dependence operator is presented, which establishes a relation between the points of one domain and the points of another.

8.1 Case operator

The case operator pieces together a set of disjoint subexpressions. The expression :

$$\begin{array}{l} case \\ \qquad \begin{array}{c} Exp_1 \\ \vdots \\ Exp_n \end{array} \\ esac \end{array}$$

is defined as:

$$\begin{aligned} \text{Eval}(case(Exp_1, \cdots, Exp_n), z) &= \begin{cases} \text{Eval}(Exp_1, z) & : \text{ if } z \in \mathcal{D}_1 = \text{Domain}(Exp_1) \\ \qquad\vdots & \\ \text{Eval}(Exp_n, z) & : \text{ if } z \in \mathcal{D}_n = \text{Domain}(Exp_n) \\ error & : \text{ otherwise} \end{cases} . \\ \text{Domain}(case(Exp_1, \cdots, Exp_n)) &= \text{Domain}(Exp_1) \cup \cdots \cup \text{Domain}(Exp_n) \\ \text{Type}(case(Exp_1, \cdots, Exp_n)) &= \text{Type}(Exp_i),\ i = 1 \cdots n \end{aligned}$$

with static checks

$$i \neq j \quad \rightarrow \quad \begin{cases} \text{Domain}(Exp_i) \cap \text{Domain}(Exp_j) = \{\} \\ \text{Type}(Exp_i) = \text{Type}(Exp_j) \end{cases}$$

Theorems

$$\begin{aligned}
case(Exp) &\iff Exp \\
case(case(Exp_1, \cdots, Exp_n)) &\iff case(Exp_1, \cdots, Exp_n) \\
Exp\ op\ case(Exp_1, \cdots, Exp_n) &\iff case(Exp\ op\ Exp_1, \cdots, Exp\ op\ Exp_n) \\
case(Exp_1, \cdots, Exp_n)\ op\ Exp &\iff case(Exp_1\ op\ Exp \cdots, Exp_n\ op\ Exp) \\
case(Dom_1 : Exp, Dom_2 : Exp, \cdots) &\iff case(Dom_1 \cup Dom_2\ :\ Exp, \cdots) \\
Var = Exp_1; \cdots; Var = Exp_n; &\iff Var = case(Exp_1, \cdots, Exp_n) \\
\text{if Domain}(Var) \cap \text{Domain}(Exp_k) = \emptyset &\implies \\
Var = case(\cdots, Exp_k, \cdots) &\iff Var = case(\cdots, Exp_{k-1}, Exp_{k+1}, \cdots)
\end{aligned}$$

For each point $z \in \text{Domain}(case)$, one and only one expression should ever exist such that $z \in \text{Domain}(Exp_i)$. The value of the case statement is not defined for values of z which are not found in the domains of any of the subexpressions, and is defined to be an error z is found in two or more subexpression domains. This error can be found with a static check.

8.2 Restrict operator

The case statement is usually used in combination with the restriction operator. The restriction operator takes a subset of an expression and is written as:

$$Dom\ :\ Exp$$

and is defined as:

$$\begin{aligned}
\text{Eval}(restrict(Dom, Exp), z) &= \begin{cases} \text{Eval}(Exp, z) & : \text{if } z \in Dom\ \cap\ \text{Domain}(Exp) \\ error & : \text{otherwise} \end{cases} \\
\text{Domain}(restrict(Dom, Exp)) &= Dom\ \cap\ \text{Domain}(Exp) \\
\text{Type}(restrict(Dom, Exp)) &= \text{Type}(Exp)
\end{aligned} .$$

The restrict operator if often used in connection with the case operator as follows:

$$\begin{array}{ll} case & \\ & \begin{array}{l} Dom_1\ :\ Exp_1 \\ \vdots \\ Dom_n\ :\ Exp_n \end{array} \\ esac & \end{array}$$

in which the restriction operators help to define the subdomains of the case expressions. The following theorems follow from the definition of the restrict operator.

$$\begin{aligned}
\text{Domain}(Exp) \;:\; Exp &\iff Exp \\
Dom \;:\; Exp &\iff Exp \quad \text{iff } Dom \supseteq \text{Domain}(Exp) \\
Dom_1 \;:\; (Dom_2 \;:\; Exp) &\iff (Dom_1 \cap Dom_2) \;:\; Exp \\
Dom \;:\; (Exp_1 \; op \; Exp_2) &\iff (Dom \;:\; Exp_1) \; op \; (Dom \;:\; Exp_2) \\
Dom \;:\; case(Exp_1, \cdots, Exp_n) &\iff case(Dom \;:\; Exp_1, \cdots, Dom \;:\; Exp_n) \\
Dom_1 \cap Dom_2 \;:\; (Exp_1 \; op \; Exp_2) &\iff (Dom_1 \;:\; Exp_1) \; op \; (Dom_2 \;:\; Exp_2) \\
Dom \;:\; (Var_1 = Exp_1; \cdots; Var_n = Exp_n;) &\iff Dom \;:\; Var_1 = Exp_1; \cdots; Dom \;:\; Var_n = Exp_n; \\
Dom \;:\; Var = Exp; &\iff Var = Dom \;:\; Exp;
\end{aligned}$$

8.3 Dependence operator

An *affine dependence function* is a function that maps each point z in a domain $\mathcal{D}$ to a point $A.z + b$ in a domain $\mathcal{E}$, $dep : \mathcal{D} \longmapsto \mathcal{E}$. The dependence operator composes an expression Exp with an affine dependence function defined by $A.z + b$, where A is a constant matrix and b is a constant vector, and is written as:

$$Exp.(z \rightarrow A.z + b)$$

and is defined as:

$$\begin{aligned}
matrix &= \left(\begin{array}{c|c} A & b \\ \hline 0 & 1 \end{array}\right) \\
\text{Eval}(dep(Exp, matrix), z) &= \begin{cases} \text{Eval}(Exp, A.z + b) \;: \\ \qquad \text{if } z \in \text{Preimage}(\text{Domain}(Exp), matrix) \\ error \qquad\qquad : \text{otherwise} \end{cases} \\
\text{Domain}(dep(Exp, matrix)) &= \text{Preimage}(\text{Domain}(Exp), matrix) \\
\text{Type}(dep(Exp, matrix)) &= \text{Type}(Exp)
\end{aligned}$$

Here are a few examples of dependency operations written in ALPHA syntax:

`X.(i,j->j,i)`	The transpose of a 2-dimensional variable X
`X.(i->i,i)`	The diagonal vector of a 2-dimensional variable X
`X.(i,j->2*i+j+3)`	A 1-dimensional variable X being indexed by $2 * i + j + 3$

The following theorems follow from the definition of the dependence operator:

$$\begin{aligned}
Exp.(z \rightarrow z) &\iff Exp \\
(Exp.(y \rightarrow Ay + b)).(z \rightarrow Cz + d) &\iff Exp.(z \rightarrow (AC)z + (Ad + b)) \\
(Exp_1 \; op \; Exp_2).(z \rightarrow Az + b) &\iff (Exp_1.(z \rightarrow Az + b)) \; op \; (Exp_2.(z \rightarrow Az + b)) \\
case(Exp_1, \cdots, Exp_n).(z \rightarrow Az + b) &\iff case(Exp_1.(z \rightarrow Az + b), \cdots, Exp_n.(z \rightarrow Az + b)) \\
(dom \;:\; Exp).(z \rightarrow Az + b) &\iff dom.(z \rightarrow Az + b) \;:\; Exp.(z \rightarrow Az + b) \\
dom.(z \rightarrow Az + b) &\iff \text{Preimage}(dom, matrix)
\end{aligned}$$

8.4 Reduction Operator

The *reduction operator* is a high level construct that allows for a more abstract expression of an algorithm and enlarges the design space for the implementation of the algorithm. Many basic regular array algorithms can be very simply expressed using this operator. It was introduced into the ALPHA language by Le Verge [5] who showed that this construct preserved referential transparency, the substitution principle, and normalization.

A reduction operator performs a many to one projection of an ALPHA expression, combining values mapped to a common result point with an associative and commutative binary operator. Its syntax is

$$\text{reduce}(\oplus, (z \rightarrow A.z + b), \textit{expression})$$

The operator $\oplus$ is any associative and commutative binary operator defined in the ALPHA language and *expression* is any ALPHA expression. The function $(z \rightarrow A.z + b)$ is a *projection function* $f : \mathcal{Z}^n \mapsto \mathcal{Z}^m$ which is written like a dependence operator, where the matrix A is of dimension $n \times m, n > m$. Not all projection functions are valid in a reduction operator. The following conditions qualify a valid function $(z \rightarrow A.z + b)$:

1. A is an $n \times m$ matrix, where $n > m$ and the Domain(*expression*) is of dimension n.
2. The matrix A has an integer right inverse. This is true of A can be written as $A = [\text{Id } 0]U$ where [Id 0] is the hermite normal form of A and U is a square unimodular matrix.
3. Given $\mathcal{D}$ =Domain(*expression*), for all $T = \left(\begin{array}{c|c} \text{Id}^{n\times n} & t \\ \hline 0 & 1 \end{array}\right)$, where $At = 0$,
   the domain $\mathcal{E} = \mathcal{D} \cup \text{Image}(\mathcal{D}, T)$ must be a convex domain over $\mathcal{Z}^n$. This can be tested by the checking that $(\text{convex}(\mathcal{E}) \backslash \mathcal{E}) \cap \mathcal{Z}^n = \emptyset$. This test guarantees that all integer points in the projection of $\mathcal{D}$ by the function $(z \rightarrow A.z + b)$ have at least one antecedent in $\mathcal{D}$. This is proved in [6, page 41].

Examples of projection functions are: $(i, j, k \rightarrow i + j + k)$ and $(i, j \rightarrow j)$. The semantics of the reduction operator are defined as:

$$\begin{aligned}
\textit{matrix} &= \left(\begin{array}{c|c} A^{n\times m} & b \\ \hline 0 & 1 \end{array}\right), n > m \\
\text{Eval}(\textit{reduce}(\oplus, (z \rightarrow A.z + b), \textit{Exp}), z) &= \begin{cases} \bigoplus(\{\text{Eval}(\textit{Exp}, y) \mid A.y + b = z, y \in \text{Domain}(\textit{Exp})\}) & : \\ \qquad \text{if } z \in \text{Image}(\text{Domain}(\textit{Exp}), \textit{matrix}) & \\ \textit{error} & : \\ \qquad \text{otherwise} & \end{cases} \\
\text{Domain}(\textit{reduce}(\oplus, (z \rightarrow A.z + b), \textit{Exp})) &= \text{Image}(\text{Domain}(\textit{Exp}), \textit{matrix}) \\
\text{Type}(\textit{reduce}(\oplus, (z \rightarrow A.z + b), \textit{Exp})) &= \text{Type}(\textit{Exp}) \\
\bigoplus(S) &= \begin{cases} \text{Identity}(\oplus) & : \text{ if } S = \emptyset \\ x \ \oplus \ \bigoplus(S \backslash \{x\}) & : \text{ if } x \in S \end{cases}
\end{aligned}$$

An example of a reduction operation to do matrix multiplication, written in ALPHA syntex is:

```
reduce( +, (u,v,w->u,v), a(i,j,k->i,j)*b(i,j,k->k,j) )
```

The following theorems follow from the definition of the dependence operator:

$$reduce(\oplus, f, Exp) \iff reduce(\oplus, f.g, Exp.g)$$

$$Dom = \{x | Cx + d >= 0\},\ f = (z \rightarrow A.z + b),\ \forall t : (At = 0 \Rightarrow Ct = 0) \implies$$
$$reduce(\oplus, f, Dom : Exp) \iff \text{Image}(Dom, f) : reduce(\oplus, f, Exp)$$

$$reduce(\oplus, f, case(Exp_1, Exp_2) \iff reduce(\oplus, f, case(\text{Domain}(Exp_1) : \text{Identity}(\oplus), Exp_2) \oplus reduce(\oplus, f, case(Exp_1, \text{Domain}(Exp_2) : \text{Identity}(\oplus))$$

$$reduce(\oplus, f, case(Exp_1, Exp_2)) \iff case(\mathcal{D}_{12} : reduce(\oplus, f, Exp_1) \oplus reduce(\oplus, f, Exp_2), \mathcal{D}_1 : reduce(\oplus, f, Exp_1), \mathcal{D}_2 : reduce(\oplus, f, Exp_2))$$
where $\mathcal{D}_{12} = \text{Image}(\text{Domain}(Exp_1), f) \cap \text{Image}(\text{Domain}(Exp_2), f)$
$\mathcal{D}_1 = \text{Image}(\text{Domain}(Exp_1), f) \backslash \mathcal{D}_{12}$
$\mathcal{D}_2 = \text{Image}(\text{Domain}(Exp_2), f) \backslash \mathcal{D}_{12}$

$$reduce(\oplus, f, Exp).g \iff reduce(\oplus, f', Exp.g')$$
where f, g, f', and g' are affine integer functions
f and f' are right invertible
$g \circ f' = f \circ g'$

$$reduce(\oplus, f, Exp) \iff case(\text{Image}(\mathcal{D}, T) \backslash \mathcal{D} : \text{Identity}(\oplus), \mathcal{D} : Y.T \ \oplus \ Exp).g$$
where $\mathcal{D} = \text{Domain}(Exp)$,
$f : \mathcal{Z}^n \mapsto \mathcal{Z}^{n-1} = (z \rightarrow Az + b)$,
$T = (z \rightarrow z - t)$, where $At = 0$,
$\forall y \in \mathcal{D} \backslash \text{Image}(\mathcal{D}, T) \ : \ f \circ g(y) = y$

if $f = f_1 \circ f_2 \circ \cdots \circ f_{n-m}$, where $f : \mathcal{Z}^n \mapsto \mathcal{Z}^m,\ n > m,\ f_i : \mathcal{Z}^{n+1-i} \mapsto \mathcal{Z}^{n-i} \implies$
$$reduce(\oplus, f, Exp) \iff reduce(\oplus, f_1, reduce(\oplus, f_2, \cdots reduce(\oplus, f_{n-m}, Exp) \cdots))$$

9 Example

This section presents an ALPHA program for a simple convolution filter. Given a sequence x_i, where $i \geq 1$, and a set of coefficients a_i, where $i = 1 \cdots 4$, the convolution filter computes an output sequence $y_i = \sum_{j=1}^{4} a_j x_{i-j+1}$ for $i \geq 4$. This function is expressed in its most general form using reduction and a parameter as a simple equation in the ALPHA language as shown in figure 2. The reduction statement can be serialized and replaced with a set of recurrence equations as shown in figure 3. The parameter can be fixed to a constant as shown in figure 4.

This same program in array notation is shown in figure 5. The expression `0 .(i,j->)` means that the constant 0 defined over the domain $\mathcal{Z}^0$ (a single point) is projected on to $\mathcal{Z}^2$ and becomes a 2-dimensional infinite array of constant 0's.

```
system convolution (N : { N | N>=0 } parameter;
                    a : { j | 1<=j<=N } of integer;
                    x : { i | i>=1    } of integer)
       returns     (y : { i | i>=N    } of integer);
  var
    Y : { i,j | 0<=j<=N; i>=1; i>=j }  of integer;
  let
    y = reduce(+, (i,j->i), a.(i,j->j)*x.(i,j->i-j+1) );
  tel;
```

Figure 2: Parameterized Alpha Program for a Convolution Filter using Reduce

```
system convolution (N : { N | N>=0 } parameter;
                    a : { j | 1<=j<=N } of integer;
                    x : { i | i>=1    } of integer)
       returns     (y : { i | i>=N    } of integer);
  var
    Y : { i,j | 0<=j<=N ; i>=N }  of integer;
  let
    Y = case
         { i,j | j=0 }     : 0.(i,j->);
         { i,j | 1<=j<=N } : Y.(i,j->i,j-1)
                           + a.(i,j->j) * x.(i,j->i-j+1);
        esac;
    y = Y.(i->i,N);
  tel;
```

Figure 3: Parameterized Alpha Program for a Convolution Filter

```
system convolution (a : { j | 1<=j<=4 } of integer;
                    x : { i | i>=1    } of integer)
       returns    ( y : { i | i>=4    } of integer);
  var
    Y :  { i,j | 0<=j<=4 ; i>=1; i>=j }  of integer;
  let
    Y = case
         { i,j | j=0 }     : 0 .(i,j->);
         { i,j | 1<=j<=4 } : Y .(i,j->i,j-1)
                             + a .(i,j->j) * x .(i,j->i-j+1);
        esac;
    y = Y .(i->i,4);
  tel;
```

Figure 4: Alpha Program for a Convolution Filter with 4 Stages

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | 1<=i} of integer)
       returns (y : {i | 4<=i} of integer);
var
  Y : {i,j | (1,j)<=i; 0<=j<=4} of integer;
let
  Y[i,j] =
      case
        {|j=0} : 0[];
        {|1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
      esac;
  y[i] = Y[i,4];
tel;
```

Figure 5: Alpha Program for a Convolution Filter in Arrray Notation

10 ALPHA Syntax

This section specifies the syntax of ALPHA . It was extracted from the `yacc` file of the actual ALPHA parser.

Comments in ALPHA start with a double dash (– –) and terminate with the end of line (like C++).

10.1 Meta Syntax

*phrase** === zero or more repetitions of *phrase*

phrase1 / *phrase2* === Alternation, either *phrase1* or *phrase2*

(···) === syntactic grouping

[···] === optional semantic phrase

`bold` === a terminal

italic === a non-terminal

10.2 Systems

system :: `system` *system-name* `(` *[input-declaration-list]* `)`
`returns (` *output-declaration-list* `);`
[`var` *local-declaration-list]*
equation-block

system-name :: *id*

input-declaration-list :: *var-declaration-list*

output-declaration-list :: *var-declaration-list*

local-declaration-list :: *var-declaration-list*

10.3 Declarations

var-declaration :: *id-list* `:` *[domain* `of] /`
id-list `:` *domain* `parameter (` `integer` / `boolean` / `real)`

var-declaration-list :: *[var-declaration-list]* `;` *var-declaration*

10.4 Domains

domain :: `{` *index-list* | *constraint-list* `} /`
domain | *domain /*
domain `&` *domain /*
domain `.` *function /*
domain `.` `convex /`
`~` *domain /*
`(` *domain* `)`

index-list :: *[index-list* `,` *] id*

constraint-list :: *[constraint-list* `;` *] constraint*

constraint :: *increasing-sequence* / *decreasing-sequence* / *equality-sequence*

increasing-sequence :: (*increasing-sequence* / *index-expression-list*) (`<` / `<=`) *index-expression-list*

decreasing-sequence :: (*decreasing-sequence* / *index-expression-list*) (`>` / `>=`) *index-expression-list*

equality-sequence :: (*equality-sequence* / *index-expression-list*) `=` *index-expression-list*

10.5 Equations

equation-block :: `let` *equation-list* `tel;`

equation-list :: [*equation-list*] *equation*

equation :: *id* `=` *expression* `;` /
id `[` [*index-list*] `]` `=` *expression* `;` /
equation-block /
domain `:` *equation*

10.6 Expressions

expression :: `case` *expression-list* `;` `esac` /
`if` *expression* `then` *expression* `else` *expression* /
domain `:` *expression* /
expression `.` *function* /
expression `[` [*index-expression-list*] `]` /
expression binary-op expression /
unary-op expression /
binary-op `(` *expression* `,` *expression* `)` /
`reduce` `(` *commutative-op* `,` *function* `,` *expression* `)` /
`(` *expression* `)` /
id /
constant

expression-list :: [*expression-list* `;`] *expression*

binary-op :: *commutative-op* / *relative-op* / `-` / `/` / `div` / `mod`

commutative-op :: `+` / `*` / `and` / `or` / `xor` / `min` / `max`

relative-op :: `=` / `<>` / `<` / `<=` / `>` / `>=`

unary-op :: `-` / `not`

constant :: *integer-constant* / *real-constant* / *boolean-constant*

10.7 Dependence Functions and Index Expressions

function :: `(` [*index-list*] `->` [*index-expression-list*] `)`

index-expression-list :: [*index-expression-list* `,`] *index-expression* / `(` *index-expression-list* `)`

index-expression :: *index-expression* (`+` / `-`) *index-term* / [`-`] *index-term*

index-term :: *integer-constant id* / *integer-constant* / *id*

10.8 Terminals

integer-constant :: *[* `-` *]* *number*

real-constant :: *[* `-` *]* *number* `.` *number*

boolean-constant :: `true` / `false`

number :: *digit digit**

digit :: `0` / `1` / ... / `9`

id :: *letter (letter* / *digit)**

letter :: `a` / ... / `z` / `A` / ... / `Z` / `_`

11 ALPHA semantics

In this section, the semantics of ALPHA are specified.

1. ALPHA is a single assignment language. The definition of a variable must be unique at each point in its domain.

2. Only single system programs are supported, no subsystems are supported at this time.

3. Variables may be declared over domains consisting of unions of polyhedra. Polyhedra are defined with both equalities and inequalities.

4. Scalar variables are declared over domains of dimension zero.

5. All data types are assumed to be infinite precision and include the special value *error* which means an uncomputable value. Some operational semantics also require a value *undefined* to represent a value which is not (yet) computed.

6. A `parameter` declaration, if it exists, must be the first declaration in the input declaration section of a system. The parameter domain defines the range of values that parameters can take in that system.

7. The precedence of expression operations is defined in the following table :

Expression Operation	Associative	Precedence
reduce(*commutative-op*, *projection-function*, *Expression*)		11
binary-op(*Expression*, *Expression*)		11
Expression.affine-function (change of basis)		10
Expression[*index-expression**] (value selection)		10
– (negation)		9
* (multiplication)	Y	8
/ (division), div (integer division), mod (modulo)		8
+ (addition)	Y	7
– (subtraction)		7
$<$, $\leq$, $=$, $\geq$, $>$, $\neq$		6
not		5
and, min, max	Y	4
or, xor	Y	3
Domain : *Expression* (restriction)		2
if *Expression* then *Expression* else *Expression*		1
case (*Expression*;)* esac		1

8. The precision of arithmetic is unspecified (assumed infinite).

9. The domains of expressions within a case statement do not intersect each other.

10. The precedence of domain operations is defined in the following table :

Domain Operation	Precedence
~ (inversion)	4
Domain.affine-function (change of basis)	3
Domain.convex (convex hull)	3
& (intersection)	2
| (union)	1

References

[1] J. M. Delosme and I. C. F. Ipsen. *Systolic Array Synthesis: Computability and Time Cones*, pages 295–312. Elsevier Science Publishers B. V. (North-Holland), 1986.

[2] Concettina Guerra. A unifying framework for systolic design. In F. Makedon, T. Melhorn, T. Papatheodorou, and P. Spirakis, editors, *VLSI Algorithms and Architectures: Aegean Workshop on Computing*, Springer-Verlag, Loutraki, Greece, 1986.

[3] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, Jul 1967.

[4] H. T. Kung. Why systolic architectures. *Computer*, 15(1):37–46, January 1982.

[5] H. Le Verge. Reduction operators in alpha. In D. Etiemble and J-C. Syre, editors, *Parallel Algorithms and Architectures, Europe*, pages 397–411, Springer Verlag, Paris, June 1992. See also [6].

[6] Hervé Le Verge. *Un environnement de tranformations de programmes pour la synthèse d'architectures régulières*. PhD thesis, Université de Rennes 1, Rennes, France, Oct 1992.

[7] Christophe Mauras. *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, Rennes, France, Dec 1989.

[8] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. *Proceedings 11th Annual International Symposium on Computer Architecture, Ann Arbor*, 208–214, June 1984.

[9] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, 1989.

[10] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing 14*, 14:163–189, 1990.

[11] D. A. Turner. *Recursion Equations as a Programming Language*, pages 1–28. Cambridge University Press, 1982.

[12] D. Wilde. *A Library for Doing Polyhedral Operations*. Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993.

[13] Yoav Yaacoby and Peter R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *Journal of VLSI Signal Processing*, 1(2):115–125, 1989.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399