A formal semantics of data parallel languages

Eric Violard

ICPS, Université Louis Pasteur, Strasbourg
Boulevard S. Brant, F-67400 Illkirch
violard@icps.u-strasbg.fr
Tel: (+33) 03 88 65 50 47, Fax: (+33) 03 88 65 50 61

Abstract. The data parallel programming model appears as a good framework in which to develop software. Nevertheless, studies have still to be carried out about this model, especially in theoretical foundations. This paper is intented to show that these foundations are met by the theory Pei. As an illustration, we propose a language based on this theory and whose formal description can serve as a sound semantics for data parallelism.

Keywords. Data-Parallel Languages, Semantics of programming languages, Shape for Parallel Computing.

Topic 11: Parallel Programming : Models, Methods and Languages

1 Introduction

Among the various parallel programming models that have recently emerged, – [13] gives an overview of existing ones – the data-parallel paradigm is of particular interest. Customized languages have been defined in this area, extending classical expressions such as C or Fortran. Typical data-parallel languages, such as C* [15] or HPF [6] include a similar core of data-parallel constructs, but they are associated with different semantics. Thus, it is interesting to carry out a formal semantic study about the data parallelism underlying concept, focussing on the essence independently to any language. We proceed then as follows. We first propose a theory that can integrate different implementations like C* or HPF. Then, we propose a language and its formal definition in order to serve as a base for illustrating our theory. The theory that we propose is the theory Pei [17, 18] born from a generalization of the so-called Polytope model [9]. Pei was introduced in order to address crucial questions of program design methodology and transformations [4], through a straightforward generalization of equational notations [11, 3].

This paper is organized in three sections: section 2 gives an overview of the language, section 3 is devoted to its formal definition and section 4 contains some more tricky examples that can be easily handled in the language.

2 The language PEI

2.1 Shaped data fields

PEI notations refer to mathematical objects that are its central concept: these objets are called *shaped data fields*. Shaped data fields, like sets or bags, are collections of elements which are *values* of a given data type. When operations use two or more elements possibly belonging to several bags, it is necessary to define the way each element is addressed whithin its bag. This can be done for example by associating an index with each element. In the literature such indexed collections are referred to as *data fields* [10, 3].

A natural and interesting way to index elements of such a collection is to use a kind of geometry: therefore each element is associated with a point in a discrete geometrical space, say $\mathbb{Z}^n$. The *index set* of a data field is the set of points where its values are placed.

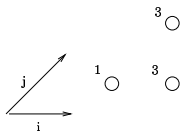

Fig. 1. Data field `A`

Example 1. Fig. 1 shows a data field `A` whose index set forms a triangle in $\mathbb{Z}^2$.

The way the values are combined can then be interpreted geometrically as *superimposing* values according to the points where they are placed.

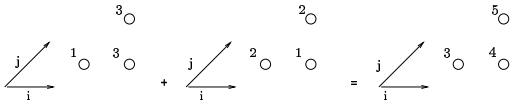

Fig. 2. Data field `A + B`

Example 2. For example, let us consider again the data field `A` and let `B` be another data field of same index set. Then, the expression `A + B` denotes the data field drawn on Fig. 2.

More generally, `A + B` (or `B + A`) denotes the data field whose values are the sum of values of `A` and `B` with the same index and whose index set is the intersection of `A` and `B` index sets. Superimposition defines another operation, we call *concatenation* and noted `[]`. The expression `A [] B` (or `B [] A`) denotes the data field whose values are the values of `A` or `B` on the union minus the intersection of their index sets.
Indexing is also natural to express *dependences* between the values of two objects as in a recurrence equation notation : dependence defines a partial function from indices to indices.

Example 3. Let `U` denote a data field whose index set is $\mathbb{N}$ and consider the data field, denoted as `V` whose values are shifted to the right. This relation between `U` and `V` can be denoted in PEI as: `V = U <|pre`
where `pre` is the name of a function which maps an index i to the index $i-1$. It means the value at index $i-1$ in `U` is moved to the index i in `V`. Such expressions meet data-flow approaches in declarative language like LUSTRE [5] and the data-parallel language $8_{1/2}$ [12].

Example 4. Let `M` denote a data field whose index set is $[1..4] \times [1..4]$ in $\mathbb{Z}^2$: it represents a 4×4-matrix, say M. Let us consider the 4×4-matrix formed of the first column of M replicated four times. It can be represented by the data field `L` defined as: `L = M <|spread_col`
where `spread_col` is the name of a partial function which maps any point (i, j) with $j \in [1..4]$ to the point $(i, 1)$. It is defined in PEI as:

```
spread_col(i,j) = (1<=j<=4) .(i,1).
```

Until now, it was the classical approach. Here is the PEI's one with the notion of "shaped" data field. The notion of *shape* expresses the difference between the locality of values on the architecture and the way they are indexed according to the problem terms. A shape is a function which associates indices with some others, called *locations*. A *shaped data field* associates a shape with a data field.

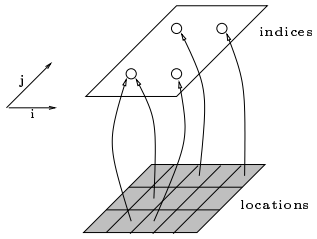

Fig. 3. A shape

Example 5. Fig. 3 shows a shape whose indices form a square in $\mathbb{Z}^2$: each of them is attached to some locations (grey painted). Fig. 4 shows a shaped data field which associates the data field `A` drawn on Fig. 1 with the previous shape. It contains three values. Its index set ($\{(i,j) \mid 0 \leq i+j \leq 1\}$) is included in the index set of the shape ($\{(i,j) \mid 0 \leq i,j \leq 1\}$).

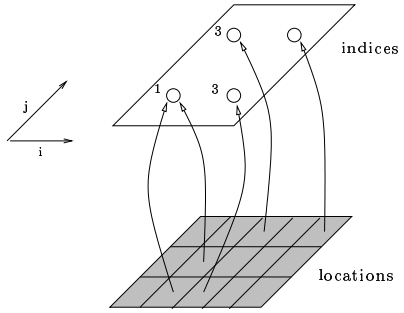

Fig. 4. A shaped data field

Let us consider the two shaped data fields drawn on Fig. 5. They are different data fields associated with different shapes, but their value at any location is the same : we say that they are equivalent. This relation is expressed in Pei through the equation: `X = align:: Y`
where `align` is the name of the partial function which maps any point (i,j) of the square $\{(i,j) \mid 0 \leq i,j \leq 1\}$ to $i+j$. This means the index $i+j$ in `Y` is the index (i,j) in `X`. The partial function named `align` is defined in Pei as:

```
align(i,j) = (0<=i,j<=1) .(i+j).
```

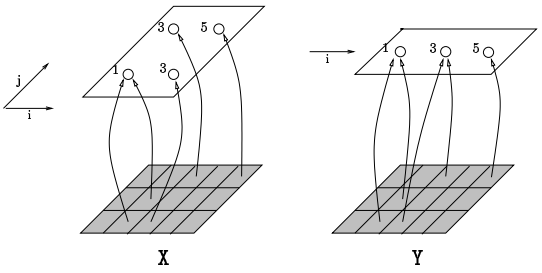

Fig. 5. Two equivalent shaped data fields

Thus, thanks to the association between locations and indices, data alignment on a *Virtual Processor Mesh* can be expressed. Hence, one can control and possibly transform the data placement on the machine in order to improve data locality. The goal is to make data local to computations and reach parallel code efficiency. Data alignment can be expressed in different manners in the data-parallel programming languages, according to how the association is modeled, whether as a function or as a bijection.
In languages like HPF, the association is a function: an array index can address several locations. Hence, if a value is moved from an index to another one at program level, it can involve any relation between locations: we could say communications are hidden. In other data parallel languages like C*, the association is a bijection. Each index refers thus to only one location. In that case, any move of a value between two indices at program level, involves a similar move between two locations at location level. Therefore, parallel variables are distributed on the mesh and communications are explicit.
The language $\mathcal{L}$ [1] was proposed as a formal semantics for languages with a bijection between indices and locations. We propose an extension to model both semantics.

2.2 PEI statements

A PEI statement is a set of equations which describes a relation between several input shaped data fields and an output one. Each equation connects two expressions which denote equal shaped data fields. Let us illustrate through a few examples what PEI statements look like:

Example 6. (The prefix sum of N numbers)

```
VecScan[n] : V -> S
/* Sum of the vector's elements */
{
S = (V <|fst) [] (V + (S <|pre))
}

fst(i) = (i=1) .(i)
pre(i) = (2<=i<=n) .(i-1)
```

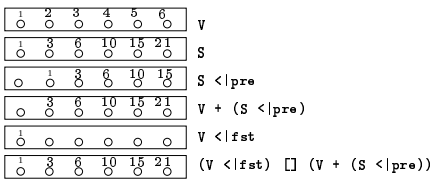

Fig.6. `S` is a solution of the equation

The input shaped data field `V` represents the vector whose values are assumed to be placed onto a line $[1..N]$ of $\mathbb{Z}$ (`n` denotes N). Fig. 6 intuitively shows that output shaped data field `S` is a solution of the equation: its values are the prefix computations of the sums of the values in `V`.

Example 7. (The matrix product) The matrix product can be written in PEI using the previous parameterized statement `VecScan[n]` (modular programming is called *structuration* in PEI):

```
MatProd[n] : A,B -> C
/* n x n Matrix Multiplication */
   {
   P = (A <|align_row)*(B <|align_col)
   C = VecScan[n](P)
   }

align_row(i,j,k) = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k) = (1<=i,j,k<=n) .(k,j)
```

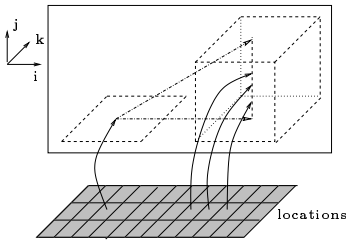

Fig.7. `A <|align_row`

The given matrices denoted by input shaped data fields `A` and `B` are broadcasted along axes j and i respectively in a cube of size N. Elements of the matrices are then superimposed in order to compute all products $a_{i,k} \times b_{k,j}$. Last, products are added using `VecScan[n]` considering their projection along the k-axis is a N-vector – detailed explanation about this semantic point is reported in appendix. The N^2 summations give the output `C`.
By definition, the shaped data fields `A` and `A <|align_row` in this statement, have the same shape (which maps indices of both matrix and cube onto locations).

Example 8. (The matrix product) (continued) Here is another Pei statement for the matrix product. It is in some sense, equivalent to the previous one. It describes the same algorithm, but dependences (involving communications) have been replaced by references to locations (involving data alignment).

```
MatProd[n] : A,B -> C
/* n x n Matrix Multiplication */
   {
   P = (align_row:: A)*(align_col:: B)
   C = VecScan[n](P)
   }

align_row(i,j,k) = (1<=i,j,k<=n) .(i,k)
align_col(i,j,k) = (1<=i,j,k<=n) .(k,j)
```

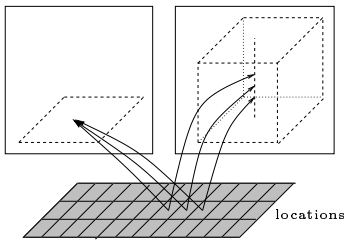

Fig.8. `A` and `align_row:: A`

Fig. 7 and 8 represent locations of the values of `A` for the statements of examples 7 and 8 respectively. In the first statement, an index is attached to only one location. In other terms, one may found a bijection between indices and locations. The statement can thus be interpreted in different intuitive semantics of data parallel languages like C* and HPF. On the contrary, in the second one, an index in `A` is attached to several locations: the association between indices and locations is not a bijection. It means this statement cannot be expressed in C* whereas it can be expressed in HPF.

These observations outline semantic differences between data parallel programming languages and we formalize them in the rest of the paper through the PEI language definition.

3 Towards a formal description

Previous section presented constructs of the language with an intuitive meaning and the relationship with data parallel languages. This section is intented to give a formal description of PEI itself, in order to have a sound semantics of data parallelism.

Preliminaries. Here are some mathematical notations and concepts that are assumed in the rest of the article. Let A and B be sets. Classically, we use $f : A \to B$ to indicate that f is a *function* with $\mathrm{dom}\, f = A$ (the *domain* of f) and $\mathrm{codom}\, f = B$ (the *co-domain* of f) and $f : A \rightharpoonup B$ to indicate f is a *partial function* ($f(a)$ can be $\perp$ (*undefined*) for some (or all or no) elements of A). Moreover, we use $\mathrm{def}\, f$ and $\mathrm{img}\, f$ to respectively denote the *domain of definition* of f i.e. $\{x \in \mathrm{dom}\, f \mid f(x) \neq \perp\}$ and its *image* i.e. $\{f(x) \mid x \in \mathrm{def}\, f\}$.

3.1 Syntax

Notations. The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter (`like this`). Non-terminal symbols are set in a upright italic font type (*like that*). Curly brackets $\{\ldots\}$ denote zero, one or several repetitions of the enclosed components. Square brackets $[\ldots]$ are used to denote optional components.

The expressions of the language are given by the following grammar:

E ::=	`-`E | E `+ 1` | E_0 `+` E_1 | ...	global operation
|	X	variable
|	E `<|`ʃ	dependence
|	ʃ`::` E	change of location
|	E_0 `[]` E_1	concatenation

where global operations are induced from classical n-ary operations ($-$, $+$ are representative examples) and ʃ is the name of some partial function on integer tuples: it is defined in the context of a statement. By convention, names of partial functions start with a lower case letter (ex: `roll`, `swap`, ...) and names of variables start with an uppercase one (ex: `A`, `B0`, `B1`, ...).

In PEI, the definition of a partial function is of the following form:

ʃ`(`pattern`) = (`pred`).(`expr`)`

where pattern is a tuple of integer names (ex: `(i,j,k)`) - an integer name is a lowercase letter `i`, `j`, ... - and denotes a formal argument of the function, pred is a boolean expression that denotes the domain where the function is defined, and expr is an integer tuple expression that denotes the result of the function on its domain. When the function is always defined, ʃ`(`pattern`) = (`expr`)` can be used as a shorthand for ʃ`(`pattern`) = (true).(`expr`)`.

In fact, only some of the expressions defined above are *well-formed*. Classically, we will say that an expression is well-formed only if a *type* can be associated with it. An elegant way to define the type of expressions consists in writing structural rules with formulas of the form $\mathrm{E} : \tau$ asserting that E is a well-formed expression of type τ [14]. Rather than introducing another language for types, mathematical types will be used. In addition, $\mathrm{df}[n, V]$ is used as a shorthand for the type of a shaped data field, that only depends on the dimension n of the geometrical space $\mathbb{Z}^n$ where values are placed, and on the data type V of its values. Well-formed expressions in PEI can then be defined by the following rules:

$$\frac{\mathrm{E} : \mathrm{df}[n, \mathbb{R}]}{\texttt{-}\mathrm{E} : \mathrm{df}[n, \mathbb{R}]}$$

$$\frac{\mathrm{E} : \mathrm{df}[n, \mathbb{R}] \quad 1 : \mathbb{R}}{\mathrm{E}\ \texttt{+}\ 1 : \mathrm{df}[n, \mathbb{R}]}$$

$$\frac{\mathrm{E}_0 : \mathrm{df}[n, \mathbb{R}] \quad \mathrm{E}_1 : \mathrm{df}[n, \mathbb{R}]}{\mathrm{E}_0\ \texttt{+}\ \mathrm{E}_1 : \mathrm{df}[n, \mathbb{R}]}$$

$$\mathrm{X} : \mathrm{df}[n, V]$$

$$\frac{\mathfrak{f} : \mathbb{Z}^{k'} \rightharpoonup \mathbb{Z}^{k} \quad \mathrm{E} : \mathrm{df}[n, V]}{\mathrm{E}\ \texttt{<|}\ \mathfrak{f} : \mathrm{df}[n-k+k', V]}\ n \geq k \qquad \text{(dependence)}$$

$$\frac{\mathfrak{f} : \mathbb{Z}^{k'} \rightharpoonup \mathbb{Z}^{k} \quad \mathrm{E} : \mathrm{df}[n, V]}{\mathfrak{f}\texttt{::}\ \mathrm{E} : \mathrm{df}[n-k+k', V]}\ n \geq k \qquad \text{(change of location)}$$

$$\frac{\mathrm{E}_0 : \mathrm{df}[n, V] \quad \mathrm{E}_1 : \mathrm{df}[n, V]}{\mathrm{E}_0\ \texttt{[]}\ \mathrm{E}_1 : \mathrm{df}[n, V]} \qquad \text{(concatenation)}$$

Let us consider the rules of dependence and change of location. Given the type of the partial function $\mathfrak{f}$, several types can be chosen for the expression E : all the types $\mathrm{df}[n, V]$, for any n such that $n \geq k$ where k is the dimension of the function $\mathfrak{f}$ co-domain. It means the definition is sufficient to define well-formed expressions, but not sufficient to have an unique type associated with an expression. Therefore we complete this definition by choosing the "least type" that can be found. This refers to an order relation between types which is defined as: $\mathrm{df}[n, V]$ is smaller than $\mathrm{df}[m, V]$ iff $n \leq m$. This precision allows expression types to be automatically inferred from function types.

3.2 Semantics

Objects PEI notations refer to objects that are called *shaped data fields*. This section is intented to give a formal definition of these objects. Let us first define the related notions of *data field* and *shape*.

A *data field* maps values of a given data type V onto its *index set*. It is therefore a partial function from indices to V. Its domain of definition is called the *index set* of the data field. An *index* is an element of any geometrical space. Formally, indices are elements of the set I defined as the union of all $\mathbb{Z}^n$, $n \in \mathbb{N}$. The index set of a data field is a subset of I, such that each element belongs to the same $\mathbb{Z}^n$.

Definition 1. *(data field) We call* data field*, any partial function X from I to V, whose domain of definition is a subset of some $\mathbb{Z}^n$.*

$$X : I \rightharpoonup V$$
$$\exists n \in \mathbb{N} \,.\, def\, X \subseteq \mathbb{Z}^n$$

A *shape* associates *locations* with some indices. It is therefore a partial function from locations to indices. Mathematically, a location and an index are both elements of I. We call location any element of the domain of a shape.

Definition 2. *(shape) We call* shape*, any partial function σ from I to I.*
$$\sigma : I \rightharpoonup I$$

The image of a shape is the set of indices which are attached to at least one location. A *shaped data field* is a data field X associated with a shape σ, such that the index set of the data field is included in the image of the shape i.e. $\mathrm{def}\, X \subseteq \mathrm{img}\, \sigma$. It could be defined as the pair (σ, X). This would assume to give a constructive definition of the part σ for every operations on shaped data fields, and would impose unnecessary restriction on operations. This was our original approach in [16, 4]. A shaped data field will be rather formally defined as a total function $\mathcal{X}$ which returns X given the shape σ and the data field nowhere defined, given any other shape.

Definition 3. *(shaped data field) We call* shaped data field*, any total function $\mathcal{X}$ from shapes to data fields, that returns for one shape σ, a data field $\mathcal{X}(\sigma)$ denoted as X such that $def\, X \subseteq img\, \sigma$, and for any other shape, the data field nowhere defined.*
$$\begin{array}{rl} \mathcal{X} : (I \rightharpoonup I) & \to (I \rightharpoonup V) \\ \sigma & \mapsto \mathcal{X}(\sigma) \end{array}$$

Expressions Semantics of well-formed phrases in Pei are defined by semantic functions, denoted as $[\![.]\!]^\tau$ for any phrase of type τ. Given some *environment* which may depend on the type τ, the function $[\![.]\!]^\tau$ maps any well-formed phrase of type τ to its mathematical meaning.

Any semantic function is defined within an environment, denoted ϵ which gives values to some global *parameters*. It may also depend on a more specific environment, denoted as ρ, that depends on the type τ. If there is no ambiguity about the type of a phrase x, the result of the semantic function is denoted as $[\![x]\!]_\rho$, or even $[\![x]\!]$ if there is no ambiguity about the specific environment.

Let us consider the well-formed expressions defined in the previous section. Their semantic function is defined given an environment ρ which maps any variable in the statement to its value. Classically, ρ can also be described as a set of pair $(\mathrm{X}, \mathrm{value})$ and we note $\rho.\mathrm{X}$ the value of the variable X in the environment ρ. The semantic function of well-formed expressions is defined as follows:

For any expression E of type $\mathrm{df}[n, V]$, $[\![\mathrm{E}]\!]$ is the shaped data field defined for any shape σ and any index z by the following semantic definitions, where we admit $-\bot = \bot$ and $\bot + a = a + \bot = \bot$, for any a in $\mathbb{R}$:

$$[\![\text{-E}]\!](\sigma)(z) = -[\![\text{E}]\!](\sigma)(z)$$

$$[\![\text{E + 1}]\!](\sigma)(z) = [\![\text{E}]\!](\sigma)(z) + 1$$

$$[\![\text{E}_0 \text{ + } \text{E}_1]\!](\sigma)(z) = [\![\text{E}_0]\!](\sigma)(z) + [\![\text{E}_1]\!](\sigma)(z)$$

$$[\![\text{X}]\!]_\rho = \rho.\text{X}$$

$$[\![\text{E < | f}]\!](\sigma)(z) = \begin{cases} [\![\text{E}]\!](\sigma)(F(z)) & \text{if } F(z) \neq \bot \wedge z \in \operatorname{img}\sigma, \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\text{f : : E}]\!](\sigma)(z) = \begin{cases} [\![\text{E}]\!](F \circ \sigma)(F(z)) & \text{if } F(z) \neq \bot \wedge z \in \operatorname{img}\sigma, \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\text{E}_0 \text{ [] } \text{E}_1]\!](\sigma)(z) = \begin{cases} [\![\text{E}_0]\!](\sigma)(z) & \text{if } [\![\text{E}_1]\!](\sigma)(z) = \bot, \\ [\![\text{E}_1]\!](\sigma)(z) & \text{if } [\![\text{E}_0]\!](\sigma)(z) = \bot, \\ \bot & \text{otherwise.} \end{cases}$$

where the function F is called the *cartesian extension* to $\mathbb{Z}^n$ of the function $[\![\text{f}]\!]$. Cartesian extension allows objects to be structured (see appendix A for a complete definition). The semantics of functions ($[\![\text{f}]\!]$) is not detailed (it is straightforward to give a mathematical meaning to the function definitions, presented in section 3.1). The meaning of functions does not depend on a specific environment.

This semantics description is *compositional* since each operation semantics is defined in terms of its arguments semantics only. In other words, the meaning of every composite phrase is expressed as a function of its intermediate subphrases. For instance, $[\![\text{E < | f}]\!]$ only depends on $[\![\text{E}]\!]$ and $[\![\text{f}]\!]$. As a consequence, the meaning of a phrase remains unchanged when a part of the phrase is replaced by an expression having the same meaning.

Statements A PEI statement is mainly a set of (unoriented) equations of the form: E_0 = E_1 connecting two shaped data field expressions. Such an equation describes the equality of two shaped data fields: it can be true or false. Since its meaning is defined from the meaning of the two expressions, the semantic function for equations depends on the same environment ρ than expressions. It returns a boolean value defined as: $[\![\text{E}_0 \text{ = } \text{E}_1]\!]_\rho = ([\![\text{E}_0]\!]_\rho = [\![\text{E}_1]\!]_\rho)$.

In PEI, an equation and a set of equations have the same type (boolean type). When applied to several equations, the associated semantic function naturally returns the conjunction of each equation semantics.

A whole PEI statement includes a header composed of a statement name P, a list I of input variables (of the form X{,X}) and an output variable O. In addition to these variables, the equation set S of the statement possibly use some intermediate variables (later referred by T):

```
P : I -> O
  { S }
```

Function definitions are reported under the system. Intuitively, a PEI statement denotes a relation between input and output shaped data fields that satisfy the equation set. The relation holds provided that, for some other intermediate shaped data fields, the equation set denotes true when the values of the variables inside are set to all these shaped data fields.
Formally, the semantic function for statements does not depend on a specific environment and returns a relation which is expressed as a set of pairs of shaped data fields. In the particular case where there is only one input and only one intermediate variable, it is defined as:

$$[\![\mathrm{P}]\!] = \{\ (\mathcal{I}, \mathcal{O}) \mid \exists \mathcal{T} \,.\, [\![\mathrm{S}]\!]_\rho = \mathrm{true} \ \text{ where } \rho = \{(\mathrm{I}, \mathcal{I}), (\mathrm{O}, \mathcal{O}), (\mathrm{T}, \mathcal{T})\}\ \}$$

where the environment ρ is described as a set. Of course, this definition can easily be generalized to any number of input or intermediate variables.

4 Further examples

We illustrate previous definition of the PEI language with the classical Cannon's algorithm [2] for the matrix product. It can be expressed in PEI as:

```
MatProd[n] : A,B -> C
/* n x n Matrix Multiplication */
   {
   A0 = face_cube:: A
   B0 = face_cube:: B
   A1 = A0 <|pipe
   B1 = B0 <|roll
   P = A1 * B1
   C = VecScan[n](P)
   }

face_cube(i,j,k) = (1<=i,j<=n & k=1) .(i,j)
pipe(i,j,k) = (1<=i,j,k<=n) .(i,((i-1)+(k-1))%n+1,1)
roll(i,j,k) = (1<=i,j,k<=n) .(((i-1)+(k-1))%n+1,j,1)
```

Using structuration, the previous statement can be generalized to the block-matrix multiplication, as:

```
BlockMatProd[n,h] : A,B -> C
/* n x n Block Matrix Multiplication */
/* with h-by-h blocks */
     {
     A0 = by_block:: A
     B0 = by_block:: B
     A1 = A0 <|pipe
     B1 = B0 <|roll
     P = MatProd[h]((elt_wise:: A1), (elt_wise:: B1))
     C = VecScan[n](step_wise:: P)
     }
```

```
by_block(x,y,i,j,k) = (1<=i,j<=n & 1<=x,y<=h & k=1) .(h(i-1)+x,h(j-1)+y)
step_wise(i,j,k) = (1<=i,j<=n*h & 1<=k<=n)
                          .((i-1)/h+1,(j-1)/h+1,k,(i-1)%h+1,(j-1)%h+1)
pipe(i,j,k) = (1<=i,j,k<=n) .(i,((i-1)+(k-1))%n+1,1)
roll(i,j,k) = (1<=i,j,k<=n) .(((i-1)+(k-1))%n+1,j,1)
elt_wise(i,j,k,x,y) = (x,y,i,j,k)
```

Matrix blocks of matrix `A` are broadcasted row-wise by dependence "`pipe`" whereas matrix blocks of matrix `B` are shifted column-wise by dependence "`roll`". Blocks are multiplied using the previous statement `MatProd[n]`.

5 Conclusion

We define a compositional semantics for data parallel languages. Some associated tools have been developed including:

- a type-checker for PEI programs, which infers the domains of shaped data fields and uses the OMEGA library [8] for evaluating set expressions.
- an interpretor from PEI to CAML programs, based on a functional interpretation of shaped data fields,
- a translator from PEI to HPF.

Our theory mainly lies on the notion of shape. Some languages refer to similar notion of shape thought of as a structure with holes or positions, into which data elements (stored in a list for example) can be inserted. We advocate semantics studies about this shape notion based on ideas from category theory ([7] as example), that could bridge different models.

Acknowledgments

I would like Stéphane Genaud for clarifying some points about data-parallel programming languages during long hours of discussions, and Philippe Gerner for our constructive brain-storming considerations about semantics.

References

1. Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.
2. L. Cannon. *A cellular computer to implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, 1969.
3. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.
4. Stéphane Genaud, Eric Violard, and Guy-René Perrin. Transformations techniques in PEI. *EUROPAR'95, LNCS*, 966:131–142, August 1995.
5. N. Halbwachs, P. Caspi, D. Pilaud, and J.A. Plaice. LUSTRE / a declarative language for programming synchronous systems. *P.O.P.L.*, 215:178–188, 1967.

6. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.
7. C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
8. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library - Version 1.00*, April 1996. Interface Guide.
9. C. Lengauer. Loop parallelization in the polytope model. *Parallel Processing Letters*, 4(3), 1994.
10. B. Lisper. *Data Parallelism and Functional Programming.* School on Data Parallelism, Les Ménuires (France). LNCS 1132-Tutorial Series, 1996.
11. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.
12. O. Michel, D. De Vito, and J.-P. Sansonnet. 8 1/2 : data-parallelism and data-flow. *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming. World Scientific*, May 1996.
13. D.B. Skillicorn and D. Talia. Models and languages for parallel computation. Technical report, Queen's University, October 1996. also published in Computing Surveys.
14. R.D. Tennent. *Semantics of Programming Languages.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1991.
15. Thinking Machines Corp. *C* Programming Guide*, November 1990.
16. E. Violard. Typechecking of PEI expressions. *EUROPAR'97, LNCS*, 1300:521–529, 1997.
17. E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.
18. E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, June 1993.

A Structuration

Structuration in PEI is based on both equation set definition and what we call *cartesian extension* of partial functions. These concepts respectively involve statements and objects.

A.1 Structuration of statements

As seen previously, a statement expresses a relation. Since an equation expresses an equality which is a relation too, a statement can be called from another statement via an equation of the form: E_0 = P(E_1) where P is the name of the statement, and E_0 and E_1 are shaped data field expressions. Intuitively, it means that the shaped data fields verify the relation denoted by the statement. This meaning is formally defined by the semantic function for equation, as:

$$[\![E_0 = P(E_1)]\!]_\rho = \begin{cases} \text{true} & \text{if } ([\![E_1]\!]_\rho, [\![E_0]\!]_\rho) \in [\![P]\!], \\ \text{false} & \text{otherwise} \end{cases}$$

A classical equation (of the form E_0 = E_1) is just a particular case where the statement (of name) P denotes the identity relation. Of course, this definition is generalized so that E_1 can be a list of expressions.

A statement call can modify the general environment ϵ by associating a value with some parameter used in the statement. The parameter must appear in the statement name of the form: P ::= root[[param]] where root is a sequence of letters and param is a list of formal parameters of any phrase type: for example, a formal parameter can be an integer name or a function name. Classically, a statement call indicates a value that is substituted for the parameter in its definition. Statement call with parameters is defined hereunder (with only one parameter):

$$[\![E_0 = \text{root}[\text{value}](E_1)]\!]_\epsilon = \begin{cases} \text{true} & \text{if } ([\![E_1]\!], [\![E_0]\!]) \in [\![\text{root}[\text{param}]]\!]_{\epsilon'} \\ & \text{where } \epsilon' = (\epsilon \mid \text{param} \mapsto [\![\text{value}]\!]_\epsilon), \\ \text{false} & \text{otherwise.} \end{cases}$$

where $(\epsilon \mid \text{param} \mapsto [\![\text{value}]\!]_\epsilon)$ is the environment which results from substituting the value $[\![\text{value}]\!]_\epsilon$ for the value of the parameter param in the environment ϵ. This definition generalizes to any number of parameters.

A.2 Structuration of objects

As seen in section 3.2, shaped data fields represent data fields, that is values mapped onto some index set which is a part of $\mathbb{Z}^n$. These objects are mainly characterized by "geometrical" operations that are dependence, change of location and concatenation that do not involve values but only indices in $\mathbb{Z}^n$. In some sense, it means that these objects are not characterized by values but rather by their structure which is defined by their index set.

Example 9. Let us consider a $N \times N$-matrix which is a particular case of data field whose values are placed on a square of $\mathbb{Z}^2$: In PEI, it is represented by a shaped data field, say `A`. Let us consider the operation that consists in shifting the values one column right. The matrix can be considered as formed of N^2 elements that are its values: In that case, the operation is written in PEI: `A <|shift` with `shift(i,j) = (i,j-1)`. In fact, the same operation can be written differently considering that it only involves the columns of the matrix. We can write: `A <|shift'` with `shift'(j) = (j-1)`. In that case, the matrix is considered as formed of N columns that define another type of values. These values are placed on indices from 1 to N in $\mathbb{Z}$: the index set of `A` is considered as a line in $\mathbb{Z}$.

As seen in section 3.2, the meaning of the second expression is defined from the notion of cartesian extension which is formally defined hereunder:

Cartesian extension refers to the cartesian structure of $\mathbb{Z}^n$: for any $k \leq n$, $\mathbb{Z}^n$ can be expressed as the cartesian product $\mathbb{Z}^{n-k} \times \mathbb{Z}^k$. It means that the cartesian coordinates of any index z of $\mathbb{Z}^n$ can be split into two parts so that z can be denoted $(x{:}y)$ where $x \in \mathbb{Z}^{n-k}$ and $y \in \mathbb{Z}^k$.

Definition 4. *(cartesian extension) Let $f : \mathbb{Z}^k \rightharpoonup \mathbb{Z}^{k'}$ with $k \leq n$. We call the* cartesian extension *of f to $\mathbb{Z}^n$, the partial function $F : I \rightharpoonup I$ defined as: $F(z) = (x{:}f(y))$ if $z \in \mathbb{Z}^n$ and $f(y) \neq \perp$, and $\perp$ otherwise. Its image is included in $\mathbb{Z}^{n-k+k'}$.*

In a geometrical point of view, the point of index y is the *projection* of the point z on the space defined by the last k axes. This intuitively means that any object in PEI can be viewed through such projection.

Of course, it can be useful to organize the objects differently in order to express some other operation. This is naturally achieved in PEI by using a change of location.

Example 10. Let us consider a $(2N) \times (2N)$-matrix. It can be seen as a $N \times N$ block matrix whose "values" are blocks of size 2×2. Assuming the shaped data field `A` represents the matrix, it can be block-organized by applying the change of location defined by the bijection `by_block`:

```
by_block(x,y,i,j) = (1<=i,j<=n & 1<=x,y<=2) .(2(i-1)+x,2(j-1)+y)
```

Let `B` be the shaped data field which verifies `B = by_block:: A`. Then, `B` can be used as a matrix whose index set is $[1..N] \times [1..N]$, any index identifying a 2×2 block. For example, this new layout of the matrix can be used to write the block-matrix product presented in section 4.