# Data-parallelism versus Functional programming: the contribution of PEI

Eric Violard
ICPS, Université Louis Pasteur, Strasbourg
Bd Sébastien Brant
F-67400 Illkirch (France)
e-mail: violard@icps.u-strasbg.fr

**Abstract**

A lot of research works have been done to examine connections between data-parallel and functional programming, as [8, 2] for example, who define the denotational semantics of a data-parallel language. Some others show how interesting it is to keep or associate a geometry with data-parallel objects: for example, in the automatic parallelization area, a change of basis makes a space-time mapping explicit. PEI is a minimal formalism to transform parallel programs [17]: it gives a framework for the main issues of data parallelism [16, 18]. The purpose of this article is to show that PEI is a link between the last two approaches as it offers a geometrical point of view via a change of basis operation and a simple denotational semantics of its programs.

## 1 Introduction

A few years ago, the design of SIMD massively parallel architectures induced a new parallel programming model, so called *data-parallelism*. This model, very closed to the underlying computation, was expressed through some dedicated data-parallel languages, such as C* or PML.

These languages were first designed in order to program these architectures efficiently. Nowadays, they can be considered as a parallel programming standard, besides the control-parallel languages. So, this tends to change the problem into its converse: general MIMD architecture design has to ensure convenient capabilities, such as routing performances, cache prefetching or synchronization barriers, in order to implement this programming model. Architectures depend then on languages as it was the case for Lisp machines, $\mu$FP or APL ones.

In a programming point of view, the question is to appreciate how a programming model is closed to a computation one, and conversely how a program matches a specification. Since it lies on synchronized computations on regularly arranged data arrays, data-parallelism seems to be a better candidate than control-parallelism. But like any other one in the sequential or parallel programming areas, this particular model recalls a major issue: how imperative languages can safely describe problems?

As Lisper says in [8], "it is hard to use these languages to express algorithms in the problem domain in an abstract machine-independent way". Obviously these languages seem not abstract enough yet, for reasoning on programs.

Nevertheless a few studies focus on abstract specifications to derive computable programs: Morgan [13] introduced a reasoning on programs founded on a refinement calculus for imperative sequential programs. In the classical functional programming area, equational reasoning is founded on the fixpoint theorem hypotheses. In [11] Misra proposed an extension of this model for non-deterministic programs.

All these approaches confirm that program reasoning should not depend on some particular computation model, or at least that such a model should not be imposed as a general rule. Indeed languages should be associated with denotational semantics issues which do not impose a given computation model, a priori.

For example, the concept of program, considered as a reasoning support, involves the definition of objects: the abstraction degree of programs then depends on the abstraction degree of these objects. This point was taken into account in the past by defining complex abstract data structures, or hierarchical classes in object oriented languages. Conversely, we consider a very abstract, unstructured, but powerful point of view: the mathematical notion of *multiset* of values. Similar approaches were also developped in the languages LINDA [7] or GAMMA [1]. Of course, the programming activity may then imply to put these values in a convenient organized directory, depending on the problem terms.

In scientific computations, for example, objects of a problem are incidentally functions on indices: the index set, that is the problem domain, is a subset of $Z^n$. In fact, this domain is only a geometrical abstraction of an other one: the computation domain. These domains characterize different ways of mapping a multiset of values. Their geometrical definition means that programming consists in discovering some change of basis. Many contributions in the automatic parallelization area have enforced this thesis till now [15, 14, 5, 12]. The geometry specifies a computation schedule and a location number which can be interpreted as a processor coordinates in a virtual architecture. These works made increasing the interest to keep or associate a geometry with a problem since the target architecture has a geometrical framework, such as a grid, a mesh, or a hypercube-like connection topology.

Concerning programming languages, some of them (ALPHA [10], or CRYSTAL [4] for example) were born of this

approach. In these languages, since objects are associated with geometrical properties, they meet parallel variables in the data-parallel languages. PEI [17, 16] comes from this approach too. It is founded on geometrical representations of multisets and on the equivalence of these representations. It defines a minimal formal framework in which specifications and programs are expressed.

In this paper, we show how a functional interpretation of PEI programs can be defined. The natural data-parallel meaning of PEI establishes then a connection between two programming paradigms: the functional one and the data-parallel one.

Section 2 intuitively presents this theory and its geometrical point of view, with the classical example of the convolution sum. Sections 3 and 4 are formal contributions to define the issues of PEI and their functional interpretation. A refinement calculus is presented in Section 5: it allows some operational transformations which yield a practical interpretation in terms of mapping and schedule, and some other denotational ones, to derive functional programs. Both operational and denotational refinements are illustrated with the convolution sum.

The second part of the paper (Sections 6 and 7) focuses on functional interpretations: a first one, which maintains some geometrical issues for a sort of data-parallel computation, and a second one which rubs these aspects to derive a standard function, whose parallel evaluation can be elaborated with a classical parallel graph reduction technique. Scan and reduction illustrate these points.

## 2 A short presentation

Languages which are founded on the recurrence equations concept, like ALPHA or CRYSTAL, define the denotational semantics of expressions as functions which map index sets onto sets of values. Function domains are then built from integral convex polyhedra in $Z^n$. Domains and values are deduced from equations which define data dependencies.

> Example : The convolution sum. Let $b$ be a series defined on $\{k \mid 1 \leq k \leq p\}$ and $c$ a series defined on $N^*$. The convolution sum $a$ of $b$ and $c$ is defined for any $n \geq 1$ as:
>
> $$a_n = \sum_{k=1}^{p} b_k \times c_{k+n-1}$$
>
> In a recurrence form, any $a_n, n \geq 1$, can be defined on the domain $\{k \mid 1 \leq k \leq p\}$ of Z by:
>
> $$\begin{cases} s_{n,1} = b_1 \times c_n & (1) \\ s_{n,k} = s_{n,k-1} + (b_k \times c_{k+n-1}) \quad 1 < k \leq p & (2) \\ a_n = s_{n,p} & (3) \end{cases}$$
>
> where $s_{n,k}$ are intermediate results. The recurrence equation (2) emphasizes uniform dependencies $(0, 1)$ for the calculation of $s_{n,k}$.

PEI is a more general notation since it expresses abstract domains called *data fields*. These domains allow to address the values of a multiset by mapping them onto a subset of $Z^n$: they can be considered as a geometrical *drawing* of multisets. Of course, any drawing of a multiset can be changed by applying a bijection: the so called *change of basis* is then a major operation on a data field. It results in an equivalent data field associated with the considered multiset.

A program PEI is a set of equations on data fields. The two sides of any equation `E1 = E2` are expressions defining the same data field.

> Example : The convolution sum (continued). In PEI, the previous definition can be expressed in the following way. Series $a$ and $c$ are expressed as data fields `A` and `C`, whose drawings are $N^*$. Series $b$ is expressed as the data field `B`, whose drawing is $\{k \mid 1 \leq k \leq p\}$. This means that for any $k$, the values $b(k)$ and $c(k)$ are located at point $k$[1]. Of course, any program which lies on an other drawing for $b$ or $c$ defines an equivalent program, provided its operations result in an equivalent data field.
>
> `Convolution:(B,C)` $\mapsto$ `A`
>
> $$\begin{cases} \texttt{align :: (B' } \triangleleft \texttt{ first)} = \texttt{B} \\ \texttt{align :: (C' } \triangleleft \texttt{ first)} = \texttt{C} \\ \texttt{B"} = \texttt{B'} \triangleleft \texttt{ spread1} \\ \texttt{C"} = \texttt{C'} \triangleleft \texttt{ spread2} \\ \texttt{S} = \texttt{addprod} \triangleright (\texttt{S} \triangleleft \texttt{ pre /\&/ B" /\&/ C"}) \\ \texttt{A} = \texttt{project :: (S} \triangleleft \texttt{ last)} \end{cases}$$
>
> ```
> align   = λ(n,k) |(n=1).k
> first   = λ(n,k) |(n=1)
> project = λ(n,k) |(k=p).n
> last    = λ(n,k) |(k=p)
> spread1 = λ(n,k) |(k<=p & n>=1).(1,k)
> spread2 = λ(n,k) |(k<=p & n>=1).(1,k+n-1)
> pre     = λ(n,k) |(1<k).(n,k-1)
> addprod = λ(b·c).(b*c) +
>           λ(s·b·c).s+b*c
> ```
>
> Data fields `B'`, `C'`, `B"` and `C"` are intermediate notations to simplify the definition of `S`.
>
> The recurrence defining $s$ suggests to draw the data field `S` in $Z^2$. So, the values of `B` and `C` are drawn in $Z^2$ too, by a sort of *data alignment*: this change of basis forms the data fields `B'` and `C'`. Their values are then broadcasted to *localize* the right values onto the right locations (data fields `B"` and `C"`) in order to compute the *recurrence* steps.
>
> For sake of simplicity, this intuitive explanation referred to the data-parallel programming issues. Nevertheless this imperative presentation must not confuse the reader: PEI is a declarative language which expresses equations on data fields. So, here is a more complete comment on every equation:
>
> The first equation implicitely defines `B'`. The function `align`, defined in $Z^2$, applies a change of basis (notation `::`) which expresses that the projection of its argument (`B'` $\triangleleft$ `first`, supposed to be a row) on an unidimensional domain, is equal to `B`. The argument `B'` $\triangleleft$ `first`, itself, applies a *geometrical operation* (notation $\triangleleft$) which selects the first row of `B'` (see Fig. 1).

[1] Notice that this matches the *natural drawing* for series. Obviously this simplification hypothesis can be done for any data field which expresses a *vectorial* data structure (vector, matrix, and so on) as in data-parallel programs, without loss of generality.

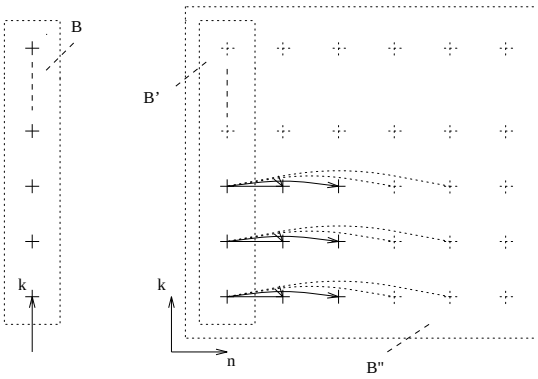


Figure 1: The data fields B, B' and B"

All these functions define the context of the program: they are expressed as $\lambda$-expressions, in which the separator `|` allows to define the domain of the function (the constant predicate `true` may be omitted), and the "`.`" begins the function body (the function identity may have an empty body).

The second equation similarly defines `C'` from `C` (see Fig. 2).

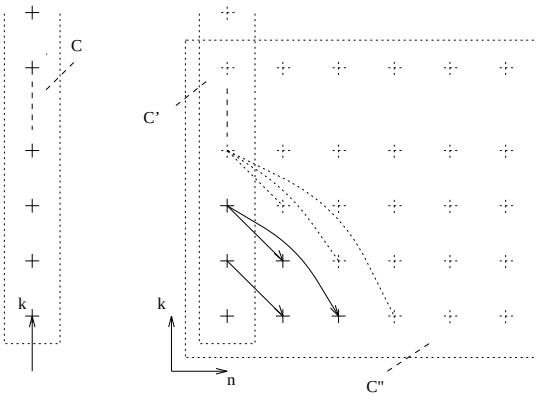


Figure 2: The data fields C, C' and C"

The data field `B"` results of the application of the geometrical operation `spread1` on `B'`. This operation broadcasts the values of the data field `B'` in the direction $(1,0)$ in $\mathrm{Z}^2$. Similarly, the values of `C'` are broadcasted by using `spread2` to form the data field `C"`. This operation broadcasts the values of the data field `C'` in the direction $(1,-1)$ in $\mathrm{Z}^2$.

In the fifth equation, the data field `S` results from the application of a so called *functional operation* (notation $\triangleright$) `addprod` on three data fields. The first one results of a geometrical operation on `S` which expresses the dependency $(0,1)$ in $\mathrm{Z}^2$. The other ones are the data fields `B'` and `C'`.

The last equation defines the data field `A`, by a change of basis: it projects the part of `S` whose drawing is $\{(n,p) \mid n \geq 1\}$ in Z.

## 3 Data fields and functional interpretation

In the theory Pei, objects called data fields, represent multisets in a geometrical way by associating a coordinate in $\mathrm{Z}^n$ with any of their values. In a data-parallel point of view, the coordinate defines the location of the PE (processor element) which computes the value, and the schedule of the computation.

**Definition 1** *A* drawing *$v$ of a multiset $M$ of values in $V$, is a partial function from $\mathrm{Z}^n$, $n \in \mathrm{N}$, in $V$ such that $M = \prec v(z), z \in dom(v) \succ$.*

This notion of drawing seems to define a natural functional interpretation of a data field `X`: it can be denoted as $[\![\mathtt{X}]\!] = v$. Since drawings of a given multiset can be deduced one another from a bijection, we consider in fact a data field `X` as the abstraction of any drawing, i.e. a data field is a drawing within some bijection. A change of basis operation is then defined in Pei: it allows to redraw a data field `X` by applying a bijection $h$.

From the previous definition, we have $[\![h \texttt{ :: } \mathtt{X}]\!] = v \circ h^{-1}$. Of course, the equivalent data fields `X` and $h$ `:: X` should have the same functional interpretation: so, this first natural interpretation is not sound. A relevant one is given hereunder: it is founded on the following definition of a data field.

**Definition 2** *A* data field `X` *is a pair $(v : \sigma)$, composed of a drawing $v$ of a multiset $M_{\mathtt{X}}$ and of a bijection $\sigma$ such that $dom(v) \subset dom(\sigma)$*[2]*.*

**Definition 3** *The functional interpretation of a data field $\mathtt{X} = (v : \sigma)$ is defined as $[\![\mathtt{X}]\!] = v \circ \sigma^{-1}$.*

The type of the values of any data field is the type *sequence*, i.e. for any set $T$ the cpo of sequences of elements in $T$ with prefix order and the empty sequence as least element. In the following we will use two operators on sequences: an associative constructor denoted as "$\cdot$" and the function `id` which is the identity on sequences of one element.

## 4 Operations on data fields

Data fields are built from an internal associative operation, called *superimposition* and denoted as `/&/`. The drawing of any data field obtained by this operation is the union of its arguments drawings. The values of the resulting data field, associated with the intersection of its argument drawings, are the sequences of their values.

External operations either define the computations of the values of a data field, or express data dependencies, or else redraw a data field. These operations apply a partial function on a data field. According to the way the function is applied, the operation is called a *functional* operation, or a *geometrical* one, or else a *change of basis* operation.

The notation Pei for partial functions is derived from the lambda-calculus: any function $f$ of domain $dom(f) = \{x \mid P(x)\}$ is denoted as $\lambda x \,|(P(x))\,.\,f(x)$. Moreover, we

[2] the function $v \circ \sigma^{-1}$ defines then an other drawing of $M_{\mathtt{X}}$.

denote $\lambda x\,.\,f(x)$ for $\lambda x\,|(true)\,.\,f(x)$, and $\lambda x\,|(P(x))$ for $\lambda x\,|(P(x))\,.\,x$.

A function $f$ can be defined as a partition $f_1$ + $f_2$ of functions where the $f_i$ are defined on disjunctive sub-domains.

**Definition 4** *Let $f$ be a partial function from $V$ to $W$ and $\mathtt{X} = (v : \sigma)$ a data field of values in $V$. The* functional operation *defines the data field $f \rhd \mathtt{X}$ of values in $W$ as the data field $(f \circ v : \sigma)$.*

**Proposition 1** $[\![f \rhd \mathtt{X}]\!] = f \circ [\![\mathtt{X}]\!]$.

**Definition 5** *Let $g$ be a partial function from $\mathrm{Z}^n$ to $dom(v)$ and $\mathtt{X} = (v : \sigma)$ a data field drawn in $\mathrm{Z}^n$. The* geometrical operation *defines the data field $\mathtt{X} \lhd g$ as $(v \circ g : \sigma)$.*

**Proposition 2** $[\![\mathtt{X} \lhd g]\!] = [\![\mathtt{X}]\!] \circ \sigma \circ g \circ \sigma^{-1}$.

**Definition 6** *Let $h$ be a bijection from $dom(v)$ onto $\mathrm{Z}^p$ and $\mathtt{X} = (v : \sigma)$ a data field drawn in $\mathrm{Z}^n$. The* change of basis *defines the data field $h$ :: X as $(v \circ h^{-1} : \sigma \circ h^{-1})$.*

**Proposition 3** $[\![h :: \mathtt{X}]\!] = [\![\mathtt{X}]\!]$.

Proof : $$\begin{aligned}[\![h :: \mathtt{X}]\!] &= v \circ h^{-1} \circ (\sigma \circ h^{-1})^{-1} \\ &= v \circ h^{-1} \circ h \circ \sigma^{-1} \\ &= v \circ \sigma^{-1} \\ &= [\![\mathtt{X}]\!]\end{aligned}$$

This last result shows the soundness of the functional interpretation of a data field. It emphasizes the difference between a denotational semantics and an operational one:

- in the context of data-parallelism, equivalent data fields may define different geometrical implementations, such as different data alignments or computation schedules, which lead to different program executions,
- whereas, in a functional interpretation domain, these programs define the same function.

To complete the presentation of Pei, let us observe that an elegant expression for specifications may use inverse functions. These functions have then to be *refined* to design deterministic executable programs. This leads to introduce inverse operations in Pei. Of course the inverse of a change of basis operation is a change of basis too, which is defined from the inverse of its bijection $h$. Other inverse operations are crucial since they define so called *reduction operators* in data-parallel languages [18].

**Definition 7** *Let $f$ be a partial function from $W$ to $V$ and $\mathtt{X} = (v : \sigma)$ a data field of values in $V$. A data field, denoted as $\mathtt{X} \lhd \cdot f$, is said to be a* functional inverse *of X by $f$ iff its functional interpretation $[\![\mathtt{X} \lhd \cdot f]\!]$ is a partial function $w$ such that:*

- $dom(w) = \{z \in dom(v) \mid \exists u \in dom(f) \cdot f(u) = v(z)\}$
- *$w(z)$ is a sequence of the values of $\prec u, f(u) = v(z) \succ$*

**Definition 8** *Let $g$ be a partial function from $dom(v)$ to $\mathrm{Z}^n$ and $\mathtt{X} = (v : \sigma)$ a data field drawn in $\mathrm{Z}^n$. A data field, denoted as $g \cdot \rhd \mathtt{X}$, is said to be a* geometrical inverse *of X by $g$ iff its functional interpretation $[\![g \cdot \rhd \mathtt{X}]\!]$ is a partial function $w$ such that:*

- $dom(w) = g(dom(v))$
- *$w(z)$ is a sequence of the values of $\prec v(y), g(y) = z \succ$*

Example : The convolution sum (continued). In Pei, the convolution can also be expressed in the following more abstract way by using a geometrical inverse operation.

```
Convolution:(B,C) ↦ A
```

$$\begin{cases} \texttt{align :: (B' } \lhd \texttt{ first)} = \texttt{B} \\ \texttt{align :: (C' } \lhd \texttt{ first)} = \texttt{C} \\ \texttt{B"} = \texttt{B'} \lhd \texttt{spread1} \\ \texttt{C"} = \texttt{C'} \lhd \texttt{spread2} \\ \texttt{P} = \texttt{prod} \rhd \texttt{(B" /\&/ C")} \\ \texttt{A} = \texttt{project :: (sum} \rhd \texttt{(reduce} \cdot \rhd \texttt{P))} \end{cases}$$

```
align   = λ(n,k) |(n=1).k
first   = λ(n,k) |(n=1)
project = λ(n,k) |(k=p).n
reduce  = λ(n,k).(n,p)
spread1 = λ(n,k) |(k<=p & n>=1).(1,k)
spread2 = λ(n,k) |(k<=p & n>=1).(1,k+n-1)
prod    = λ(b·c).(b*c)
sum     = id + λ(a·b).a+sum b
```

The intermediate data field P contains all the products $b_k * c_{n+k-1}$. The geometrical inverse operation λ(n,k).(n,p) puts some sequence of all the values of points $(n,k)$ of P at the index $(n,p)$. These values are then added by using the functional operation sum. This program can be *refined* and leads to the previous one. This concept of refinement is presented in the next section.

## 5 Refinement

Refinement of specification is a powerful programming concept [13]. In the previous sections, we have discussed Pei programs. More generally, a statement Pei defines a problem P as a relation between multisets. Its graph is a set of pairs $(D, R)$ where $D$ and $R$ are tuples of multisets, respectively called *input* and *output* multisets. Such a relation is specified as a system of unoriented equations, each of them defining two equal expressions of some data field. A specification states the inputs and outputs: inputs are the parameters of the system and the outputs are its unknowns. Any equation whose arguments are only inputs defines *preconditions* on these input data fields. Any other one, whose arguments may be intermediates or outputs defines *postconditions* on these data fields.

Intuitively, we will say that a specification S is refined by a specification S' if any solution of S' is equivalent to a solution of S for some equivalent parameters, where this equivalence means that equivalent data fields are associated with the same multiset, as it was said in Section 2.

**Definition 9** *Let $S$ and $S'$ be two specifications. $S$ is said refined by $S'$, denoted as $S \sqsubseteq S'$*

- *either if $Pre \Rightarrow Pre' \wedge Post' \Rightarrow Post$, where $Pre$, $Pre'$ and $Post$, $Post'$ are the predicates associated with pre- and postconditions of $S$ and $S'$,*
- *or if $S$ is identical to $S'$ by substituting the data field $h$ :: X for all occurrences of X.*

These two kinds of refinement, respectively called the *denotational refinement* and the *operational* one, may apply whatever the step in the design of a solution. Since it is monotonic, this refinement definition induces a *refinement calculus* which is founded on the following rules:

$$\begin{array}{lcl}
\mathtt{Y} = f1 \circ f2 \rhd \mathtt{X} & \sqsubseteq & \mathtt{Y} = f1 \rhd (f2 \rhd \mathtt{X}) \\
\mathtt{Y} = \mathtt{X} \lhd g1 \circ g2 & \sqsubseteq & \mathtt{Y} = (\mathtt{X} \lhd g1) \lhd g2 \\
\mathtt{Y} = h1 \circ h2 \ \mathtt{::}\ \mathtt{X} & \sqsubseteq & \mathtt{Y} = h1 \ \mathtt{::}\ (h2 \ \mathtt{::}\ \mathtt{X}) \\
\mathtt{Y} = (f \rhd \mathtt{X}) \lhd g & \sqsubseteq & \mathtt{Y} = f \rhd (\mathtt{X} \lhd g) \\
\mathtt{Y} = (h \ \mathtt{::}\ \mathtt{X}) \lhd g & \sqsubseteq & \mathtt{Y} = h \ \mathtt{::}\ (\mathtt{X} \lhd h^{-1} \circ g \circ h) \\
\mathtt{Y} = (g1 \ \mathtt{+}\ g2) \cdot \rhd \mathtt{X} & \sqsubseteq & \mathtt{Y} = g1 \cdot \rhd \mathtt{X} \,\mathtt{/\&/}\, g2 \cdot \rhd \mathtt{X} \\
\mathtt{Y} = (g1 \circ g2) \cdot \rhd \mathtt{X} & \sqsubseteq & \mathtt{Y} = g1 \cdot \rhd (g2 \cdot \rhd \mathtt{X}) \\
\mathtt{Y} = \mathtt{X} \lhd \cdot (f1 \ \mathtt{+}\ f2) & \sqsubseteq & \mathtt{Y} = \mathtt{X} \lhd \cdot f1 \,\mathtt{/\&/}\, \mathtt{X} \lhd \cdot f2 \\
\mathtt{Y} = \mathtt{X} \lhd \cdot (f1 \circ f2) & \sqsubseteq & \mathtt{Y} = (\mathtt{X} \lhd \cdot f1) \lhd \cdot f2
\end{array}$$

Associated with the following propositions, the last four rules define denotational transformations of reductions in some compositions of functional or geometrical operations. This kind of transformation is used to refine the convolution example in its first form.

**Proposition 4** *The data field* $\mathtt{X} \lhd \cdot f$ *is equal to* $f^{-1} \rhd \mathtt{X}$ *iff* $f$ *is bijective.*

**Proposition 5** *The data field* $g \cdot \rhd \mathtt{X}$ *is equal to* $\mathtt{X} \lhd g^{-1}$ *iff* $g$ *is bijective.*

On the other hand, an operational refinement consists in defining some operational order from an initial specification, in the following sense: a program is a particular specification, which defines a function between input and output data fields. In order to get an operational definition of this function, i.e. the set of computations of an abstract computer it envolves, it is necessary to introduce a partial order on the data field elements. This order is defined by using the bijection $\sigma$ of the considered data field.

**Definition 10** *Let* $<$ *be any partial order on* $\mathrm{Z}^m$ *and* $(v : \sigma)$ *a data field whose bijection* $\sigma$ *is a function from* $\mathrm{Z}^n$ *to* $\mathrm{Z}^m$. *The relation* $\vdash$ *defined as*

$$\forall z, z' \in dom(v) \cdot v(z) \vdash v(z') \text{ iff } \sigma(z) < \sigma(z')$$

*is a partial order on* $dom(v)$, *called an* operational order.

The choice of an order $<$ on $\mathrm{Z}^m$ predetermines the operational definition of a program, for example in defining a schedule and maybe a mapping of the computations onto a set of virtual processors. The aim of program transformations, indeed, is to make explicit a bijection $\sigma$ which introduces a convenient operational order. These transformations lie on the change of basis operation.

The following example illustrates the result of an operational refinement, applied on the first statement of the convolution sum.

Example : The convolution sum (continued). The inital statement can be refined to the following one, which explicits an operational order:

$\mathtt{Convolution:(B,C)} \mapsto \mathtt{A}$

$$\begin{cases}
\mathtt{B"} = \mathtt{B} \,\mathtt{/\&/}\, \mathtt{B"} \lhd \mathtt{init} \\
\mathtt{C"} = \mathtt{C} \,\mathtt{/\&/}\, \mathtt{C"} \lhd \mathtt{left} \\
\mathtt{S} = \mathtt{addprod} \rhd (\mathtt{S} \lhd \mathtt{right} \,\mathtt{/\&/}\, \mathtt{B"} \,\mathtt{/\&/}\, \mathtt{C"}) \\
\mathtt{A} = \mathtt{S} \lhd \lambda\mathtt{(x,t)} \,|\mathtt{(x=p)}
\end{cases}$$

```
init    = λ(x,t) |(t>x+1).(x,t-2)
left    = λ(x,t) |(t>x+1).(x+1,t-1)
right   = λ(x,t) |(1<x<=p).(x-1,t-1)
addprod = λ(b·c).(b*c) +
          λ(s·b·c).s+b*c
```

Refinement steps for this program can be found in [16]. This statement determines a computing schedule and a regular mapping, which describe a systolic array (drawn for $p = 5$ in Fig. 3). The geometrical operations `left` and `right` define the processor links. The geometrical operation `init` defines a memory cell. The systolic array topology is defined by the set $\{x \mid 1 \leq x \leq p\}$ which is deduced from the data field drawing.

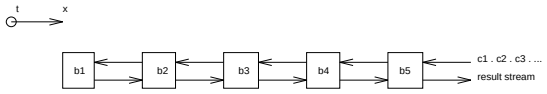


Figure 3: A processor array for the convolution sum

This powerful refinement concept, and its dual called *program abstraction*, are founded on the intrinsic geometry of data fields : it yields transformations which abstract classical functional transformations.

## 6 A functional interpretation of PEI Programs

Since data fields are interpreted as functions and sequences of elements can be interpreted as lists, PEI programs, such as the previous one, can be interpreted as functional programs associated with graph reduction.

In order to express such functional solutions, we introduce a MIRANDA-like notation [9].

The notation for function application is only a juxtaposition, as in `f x`. Function application is left-associative, so `f x y = (f x) y`. This assumes the currying of any function of two or more arguments. A list is written in square brackets: `[]` is the empty list, and `[a,b,c,d]` contains four elements. `x:xs` denotes the list `xs` with `x` attached to the front. The enumeration syntax is just a shortland for constructing the list with ":", so `[a,b,c,d] = a:b:c:d:[]`. Lists may be appended by the "`++`" infix operator. Any function `f` is defined on lists, assuming that `f [] = []`.

Now, let us come back to the convolution sum, to propose a functional interpretation of the previous systolic solution.

Example : The convolution sum (continued). The following program expresses the functional interpretation of the previous PEI program:

```
Convolution1 (b,c) = a
  where
  b" [(x,t)] = superimpose b
                 (compose b" init) [(x,t)]
  c" [(x,t)] = superimpose c
                 (compose c" left) [(x,t)]
  s  [(x,t)] = compose addprod (superimpose
                 (compose s right)
                 (superimpose b" c")) [(x,t)]
  a  [(x,t)] = compose s last [(x,t)]
```

Functions `compose` and `superimpose` are higher functions which are the interpretation of mapping an external or internal Pei operation. They can be defined as following:

```
compose f g x = f (g x)
superimpose f g x = f x ++ g x
```

Functions `addprod`, `init`, `left`, `right` and `last` can be defined as:

```
addprod [b,c]   = [b*c]
addprod [s,b,c] = [s+b*c]
addprod a:b:c:d:xs = []
addprod [a] = []

init  [(x,t)] = [(x,t-2)], t>x+1
              = [], otherwise
left  [(x,t)] = [(x+1,t-1)], t>x+1
              = [], otherwise
right [(x,t)] = [(x-1,t-1)], 1<x<=p
              = [], otherwise
last  [(x,t)] = [(x,t)], x=p
              = [], otherwise
```

Let us consider a new example, which is significant in the data-parallel approach.

Example : The partial sums of $n$ numbers. This Pei program computes all partial sums of a series $(x_i)_{i=1..n}$ of $n$ numbers. It is a *scan operation* in data-parallel languages.

`PartialSums:X` $\mapsto$ `S`

$\{$ `S = add` $\triangleright$ `(X /&/ S` $\triangleleft$ `pre)`

```
pre = λi |(1<i<=n).i-1
add = id + λ(s,x).s+x
```

This Pei program can be translated as following, in a functional style by introducing data fields as functions and sequences as lists.

Example : The partial sums of $n$ numbers (continued).

```
PartialSums1 x = s
  where
  s [i] = (compose add
          (superimpose (compose s pre) x)) [i]
```

Functions `add` and `pre` can be defined this way:

```
add [a] = [a]
add [a,b] = [a+b]
add a:b:c:xs = []

pre [i] = [i-1], 1<i<=n
        = [], otherwise
```

This example shows how the functional interpretation of Pei programs defines functional programs. The dual problem of determining a parallel computation associated with a functional program can find several solutions:

- it may consist in implementing its reduction graph onto a parallel architecture,
- or in coming back to the original Pei program and transforming it by using the refinement calculus in order to explicit an operational order.

These first functional solutions can be considered as hybrid solutions since the functions refer to geometrical elements such as indices `x`, `t` or `i` on the previous examples. Of course, the natural meaning of these elements may induce data-parallel solutions.

Conversely, a pure functional solution consists in substituting some abstract data types, such as lists or trees for functions: this leads to rub any geometrical reference for a parallel computation.

## 7 Functional programming

In order to emphasize the relationship between functional and data-parallel programming, this section illustrates the way the functional interpretation of Pei programs leads to pure functional programs. As a first illustration, let us come back to the previous example `PartialSums1`.

Example : The partial sums of $n$ numbers (continued). In functional programming this problem may be solved by representing functions as lists. The following program explicits the previous solution:

```
PartialSums2 [] = []
PartialSums2 x:xs = flat_map add
          (combine2 (PartialSums2 xs) (x:xs))
```

where `flat_map` and `combine2` implement composition and superimposition:

```
flat_map f x:xs = f x ++ flat_map f xs
flat_map f [] = []

combine2 x1:x1s x2:x2s = [x1,x2]:combine2 x1s x2s
combine2 x1:x1s [] = [x1]:combine2 x1s []
combine2 [] x2:x2s = [x2]:combine2 [] x2s
combine2 [] [] = []
```

and assuming that list `[xn,...,x2,x1]` represents series $(x_i)_{i=1..n}$.

Note that this functional program is a particular way to implement the classical scan-from-right function[3] `scanr (+)` which collects partial results into a list. Here is its definition for $n = 3$:

```
scanr (+) [x3,x2,x1] = [x3+x2+x1,x2+x1,x1]
```

The functional program `PartialSums2` is obtained from the program `PartialSums1` by representing functions `x` and `s` by lists: let `lx` and `ls` be these lists. `PartialSums1 x = s` can be rewritten as `PartialSums2 lx = ls`.

Let `List` denotes the representation function, such that `List lx = x` and `List ls = s`. It is defined as follows:

```
List [] [i]   = []
List x:xs [i] = [x], i=(length xs)+1
              = List xs [i], otherwise
```

[3] Functions `scanr` or `scanl` are generally considered as primitives in data-parallel languages.

Conversely, the functional program `PartialSums1` can be obtained from the program `PartialSums2` by using the function `List` in order to represent lists as functions: this representation introduces a geometry again, as it is presented in [8].

The same remark can be made for the convolution sum. The last statement emphasized a functional interpretation of the systolic solution. Assuming $c$ is a finite series, we can rewrite this first functional program by substituting lists for functions.

Example : The convolution sum (continued).

```
Convolution2 (lb,[]) = []
Convolution2 (lb,xlc:xlcs) =
     (sum lb xlc:xlcs):Convolution2 (lb,xlcs)

  where
  sum lb [] = []
  sum [] lc = []
  sum xlb:xlbs xlc:xlcs = flat_map addprod
              (combine3 (sum xlbs xlcs)
                        xlb:xlbs xlc:xlcs)
```

Let us consider a last example, from a PEI program until a pure functional one.

Example : The sum of $n$ numbers. The following PEI program can be obtained by performing a reduction via some refinement rules. It computes the sum of $n$ numbers in the particular case where $n = 2^k$.

$$\texttt{Sum:X} \mapsto \texttt{S}$$

$$\begin{cases} \texttt{Y} = \texttt{add} \triangleright (\texttt{X /\&/ Y} \triangleleft \texttt{low /\&/ Y} \triangleleft \texttt{diag}) \\ \texttt{S} = \texttt{Y} \triangleleft \texttt{root} \end{cases}$$

$$\begin{aligned} \texttt{low} &= \lambda(\texttt{i,j}) \,|\, (\texttt{j>0}).(\texttt{i,j-1}) \\ \texttt{diag} &= \lambda(\texttt{i,j}) \,|\, (\texttt{j>0}).(\texttt{i-2}\tilde{}\texttt{(j-1),j-1}) \\ \texttt{root} &= \lambda(\texttt{i,j}) \,|\, (\texttt{i=n \& j=k}) \\ \texttt{add} &= \texttt{id + } \lambda(\texttt{a}\cdot\texttt{b}).\texttt{a+b} \end{aligned}$$

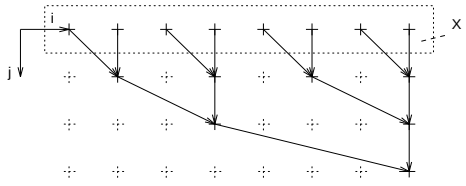


Figure 4: The data field Y drawn on a tree

The data field `X` represents a multiset composed of $n$ numbers. It is drawn on plane in the interval $[1..n, 0]$. Here is the functional interpretation of this program:

```
Sum1 x = s
  where
  y [(i,j)] = compose add
              (superimpose x (superimpose
                (compose y low)
                (compose y diag))) [(i,j)]
  s [(i,j)] = compose y root [(i,j)]
```

Functions `low`, `diag` and `root` can be defined this way:

```
low  [(i,j)] = [(i,j-1)], j>0
             = [], otherwise
diag [(i,j)] = [(i-2^(j-1),j-1)], j>0
             = [], otherwise
root [(i,j)] = [(i,j)], i=n & j=k
             = [], otherwise
```

A classical functional interpretation of this program may consist in representing the function `x` as a binary tree with numbers on the leaves. Such a tree can be formally defined as the following structured type:

```
tree ::= LEAF num | BRANCH tree tree
```

A tree is either a `LEAF`, which contains a number, or a `BRANCH`, which connects two smaller trees: `LEAF` and `BRANCH` are then constructors for the type tree, like ":" is a constructor for the lists[4]. This leads to the following functional program, on which only a parallel graph reduction can apply.

Example : The sum of $n$ numbers (continued).

```
Sum2 (LEAF num)     = num
Sum2 (BRANCH t1 t2) = (Sum2 t1) + (Sum2 t2)
```

A geometrical representation of a tree can be determined by the following function `Tree`:

```
Tree (LEAF num) [(i,j)] = [num], i=1 & j=0
Tree (BRANCH t1 t2) [(i,j)]
          = Tree t1 [(i,j)], i<=2^(depth t1)
          = Tree t2 [(i-2^(depth t2),j)], otherwise
```

By applying this representation function of a tree, the previous program `Sum1` can be obtained, which introduces a convenient geometry again for a parallel computation.

## 8 Related works

The purpose of Lisper in [8] is to define a denotational semantics for some common concepts of data-parallel languages. The main objective is to guide the design of data parallel languages with a higher level of abstraction: this supposes to capture these different concepts in a more general model.

Conversely, PEI is an abstract model based on the geometrical representation of multisets. The operations which are defined in this model generalize data parallel primitives, such as data alignment, global operations, scans and reductions, communications, etc. PEI claims that a formalism should not a priori impose a computation model: a computation is only determined from the geometry of the objects.

Lisper view is close to the concepts underlying the language CRYSTAL. ALPHA [10] and CRYSTAL [4] were originally founded on the recurrence equations concept. Only the "spatial" properties of the objects defined in these languages meet the data parallel primitives. As an example, data alignment or reduction are external operations in these languages.

PEI is born of the influence of an other complementary approach. The refinement of non-determinstic programs

[4] A similar abstract parallel tree architecture is defined in [6], in order to reason formally about the parallel scan algorithm.

which defines step by step an executable program from a specification. UNITY [3] and GAMMA [1] follow this approach. The main problem for these formalisms is to define efficient solutions.

Geometry is the foundation for our transformations. The previous examples show the interest to keep or associate a geometry with objects in order to define an operational semantics of programs, due to the geometrical properties of parallel architectures.

An other point of view is the one of O'Donnell in [6], which consists in defining an architecture via an algebraic data type. Architecture and algorithm are thus considered as two functions, and computation consists in applying the algorithm on the architecture. This view is a sort of abstraction of the classical graph reduction model associated with functional programming. Our view is different since the specification of the problem is transformed to match a target architecture. These transformations explicit algorithm and architecture from the problem specification.

## 9 Conclusion

The theory PEI is founded on the simple mathematical concepts of multiset and of an equivalence between their representations as data fields. Program transformations are founded on this equivalence and defined from a refinement relation. Due to the unifying aspect of this theory, solutions that can be reached by these transformations are relevant to various parallel programming models, as systolic processing or data-parallelism.

This article shows that a specification PEI can be refined until a program can be described in a functional language. To parallelize a program for a particular machine consists then in implementing its reduction graph or directly interpret it as a parallel algorithm on an architecture. This last point of view is clearly captured in PEI.

As a future development, refinement rules can be used in the reverse order to abstract a specification and detect reduction: any recursive definition in PEI can be abstracted with a non-recursive one by using a reduction. Abstraction increases non-determinism and leads to perform multiset operations. This new issue may show equivalence between functional programs. In this sense, PEI is also a contribution to derive parallel implementation from functional programs.

## References


[1] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–79, 1990.

[2] L. Bouge and J.-L. Levaire. Control structures for data-parallel simd languages : semantics and implementation. *Future Generation Computer Systems*, 8:363–378, 1992.

[3] K.M. Chandy and J. Misra. *Parallel Program Design :* A foundation. Addison Wesley, 1988.

[4] M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.

[5] A. Darte and Y. Robert. *Séquencement des nids de boucles.* Algorithmique parallèle. Masson, 1992.

[6] J.T. O Donnell. A correctness proof of parallel scans. *Parallel Processing Letters*, 4, 1994.

[7] D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[8] P. Hammalund and B. Lisper. On the relation between functional and data parallel programming languages. *FPCA*, pages 210–222, June 1993.

[9] S.L. Peyton Jones. *The implementation of Functional Programming Languages.* Prentice-Hall, c.a.r. hoare edition, 87.

[10] C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.

[11] J. Misra. Equational reasoning about nondeterministic processes. *Formal Aspects of Computing*, 2:167–195, 1990.

[12] C. Mongenet, P. Clauss, and G.-R. Perrin. Geometrical tools to map systems of affine recurrence equations on regular arrays. *Acta Informatica*, 31:137–160, 1994.

[13] C. Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.

[14] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989.

[15] S.K. Rao. *Regular iterative algorithms and their implementations on processor arrays.* PhD thesis, Stanford University, 1986.

[16] E. Violard. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4(3):313–328, 1994.

[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. Reduction in PEI. *CONPAR 94*, 854:112–123, 1994.