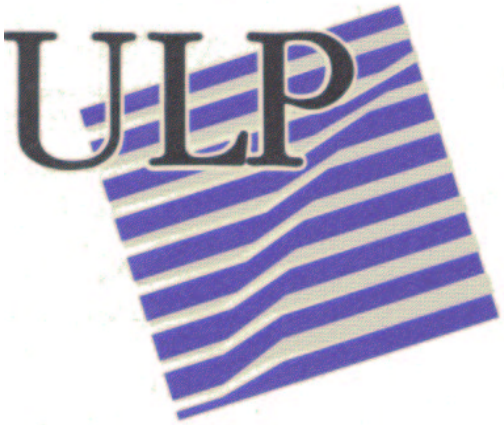

ICPS

Informatique et
Calcul Parallèle
de Strasbourg

Publication 94-05

PEI : a Theoretical Framework for Data Parallel Programming

G.-R. Perrin, E. Violard and S. Genaud

Published in French-british Workshop on Data Parallel Languages and Compilers, April 20-22, 1994, Lille (France)

ICPS - Université Louis Pasteur
Pôle API, Boulevard Sébastien Brant, F-67400 Illkirch

PEI : a Theoretical Framework for Data Parallel Programming

G.-R. Perrin, E. Violard and S. Genaud

ICPS, Université Louis Pasteur
F-67400 Illkirch
e-mail: perrin,violard,genaud@icps.u-strasbg.fr

Abstract. This article presents a theoretical framework, called PEI, which is a foundation for data parallel programming. This formalism lies on a small but powerful set of primitives : they are operations on data fields and inverse operations. They induce a clear refinement calculus to transform specifications in executable programs by ensuring a safe process of program design, program transformation or code optimization.

Keywords. Data Alignment, Data Parallelim, Environment, Language, Multiset, Recurrence Equation, Reduction, Refinement.

1 Introduction

Many theories have been proposed in order to derive and map programs onto target parallel architectures. Some of them (ALPHA [Mau89], CRYSTAL [CCL91], LACS [Raj93], etc.) define a program as a set of recurrence equations and propose synthesis techniques, which are founded on the *Polytope Model* [Len94]. Other ones (LINDA [Gel85], UNITY [CM88], GAMMA [BM90], etc.) are based on a refinement calculus : specifications are then expressed as predicates.

The theory PEI was defined [VP92, VP93, Vio94] in order to unify these two approaches and benefit from their advantages. PEI provides a formal frame to describe problem specifications and reason on programs. It lies on a small but powerful set of mathematical issues : first of all, the notion of multiset of values. In order to address these values they are mapped onto geometrical domains, and form *data fields*. From an operational point of view, such domains can abstract the mapping of calculations onto a mesh of virtual processors, whereas from a specification point of view, they can express the natural geometry of data structures such as arrays, for example. The set of data fields is supplied with three external operations, which either compute values of data fields *(functional operation)*, or express data dependencies *(geometrical operation)*, or else redraw a data field *(change of basis operation)*. These operations are bricks for a refinement calculus to derive or transform programs. Programs themselves are a set of equations between data fields : in that sense, PEI is a sort of abstract notation for systems of recurrence equations.

In order to overcome the intrinsic determinism of definitions expressed in term of recurrence equations, most languages include a non-deterministic notation, called *reduction operator*. It is one of the major issues in *data parallel* languages too. It means the definition of a n-ary operation from a binary one, for example to express the sum of a series of n numbers such as $s = \sum_{i \in 1..n} a_i$. A sort of program refinement then consists in reducing this non-determinism in order to generate an efficient reduce function code for some target architecture. In Pei reduction is defined as the *inverse operation* of a geometrical operation. The refinements it involves can lead to different reduction or scan functions.

This article aims to convince that all these issues ensure that Pei is a theoretical framework for data parallel languages : it supports the concepts of data alignment, global operation, regular communication, broadcast and reduction in a very general sense. Moreover, the simple mathematical issues it involves and its refinement calculus provide a powerful data parallel programming environment.

2 Definition of the formalism Pei

2.1 Specifications and programs

Pei specifies a problem as a relation between multisets. Its graph is a set of pairs (D, R) where D and R are tuples of *multisets* of values, respectively called *input* and *output* multisets.

Such a specification is expressed as a system of unoriented equations[1], each of them defining two equal expressions of some *data field*, i.e. of a geometrical drawing of a multiset. These expressions are defined from data field identifiers and operations on data fields. A specification states the inputs and outputs identifiers : 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.

A specification is called a *program* if its system of equations defines a function, i.e. the system has at most one solution.

Example 1. $s = \sum_{i \in 1..n} a_i$.

```
sum : A -> S
  {A  =  dom :: A
   X  =  A <| first  /+/  add |> (A /;/ (X <| pre))
   S  =  X <| last
  }
```

[1] Pei means Parallel Equations Interpretor and pays hommage to the architect of the Pyramide du Louvre.

```
dom   =  \i |(1<=i<=n)
pre   =  \i |(1<i<=n) .i-1
first =  \i |(i=1)
last  =  \i |(i=n)
add   =  \(x;y) .x+y
```

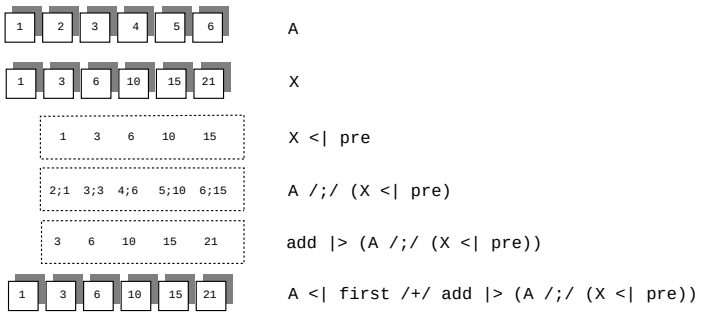

Fig. 1. X is a solution of the second equation

- the first equation defines an initial drawing of an input data field `A` : its values are mapped onto a line segment $[1..n]$,
- fig. 1 intuitively shows that data field `X` is a solution of the second equation : its values are the prefix computations of the sums of the values in `A`. The first expression `X <| pre` defines a data field resulting from `X` by shifting its values. They are then composed with the values of `A` in `A /;/ (X <| pre)` and added one another by `add |> (A /;/ (X <| pre))`. The expression `A <| first /+/ add |> (A /;/ (X <| pre))` links two parts to form a data field equal to `X`,
- the last equation defines the output `S`.

$\diamondsuit$

2.2 Mathematical issues

The mathematical issues of PEI are very simple. The basic notion is the notion of *multiset* of values, whose elements can be mapped onto a geometrical domain, in such a way that any element can be distinguished by integer coordinates.

This defines a *drawing* as a mapping from Z^n to a set of values. A major point to notice is that any *program transformation* consists in associating an other drawing with the considered multiset : for example, substitute a space-time

domain for an iteration space. It is then essential to remember that a drawing is always defined within a bijection : these potential drawing transformations are expressed by the concept of *data field*.

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

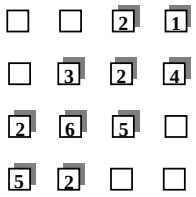

Fig. 2. An example of data field

Definition 2. A *data field* `X` is a pair $(v : \sigma)$, composed of a drawing v of a multiset M_X and of a bijection σ such that $dom(v) \subset dom(\sigma)$ and $v \circ \sigma^{-1}$ is an other drawing of M_X.

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. The values of the resulting data field, associated with the intersection of its arguments, are *sequences* of values. Restrictions of this operation to the disjunction and intersection of drawings are respectively called *sum* and *product*, and denoted as `/+/` and `/;/`.

Fig. 3. Superimposition operation

External operations either define the computations of the values of a data field, or express data dependencies, or 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 lambda-calculus : any function f of domain $dom(f) = \{x \mid P(x)\}$ is denoted as $\backslash x \,|\, P(x) \,.\, f(x)$. Moreover, we denote $\backslash x \,.\, f(x)$ for $\backslash x \,|\, \texttt{true} \,.\, f(x)$, and $\backslash x \,|\, P(x)$ for $\backslash x \,|\, P(x) \,.\, x$. Last, a function f defined on disjunctive sub-domains is denoted as $f_1 + f_2$, and the domain of a composed function $f \circ g$ is $\{x \in dom(g) \mid g(x) \in dom(f)\}$.

Definition 3. Let $(v : \sigma)$ be a data field whose values are in V and f a partial function from V to W. The *functional operation* defines data field f `|>` $(v : \sigma)$ whose values are in W as

$$f \quad \texttt{|>} \quad (v : \sigma) \stackrel{def}{=} (f \circ v : \sigma)$$

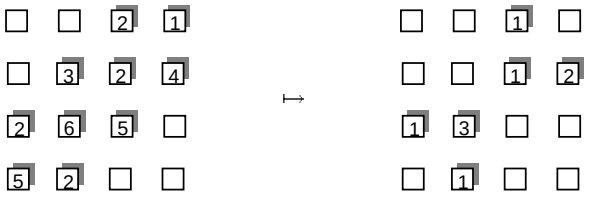

Fig. 4. Functional operation : `\x |(x mod 2 =0) .x/2 |> X`

Definition 4. Let $(v : \sigma)$ be a data field drawn on Z^n and g a partial function from Z^n to $dom(v)$. The *geometrical operation*, or *routing*, defines the data field $(v : \sigma)$ `<|` g as

$$(v : \sigma) \quad \texttt{<|} \quad g \stackrel{def}{=} (v \circ g : \sigma)$$

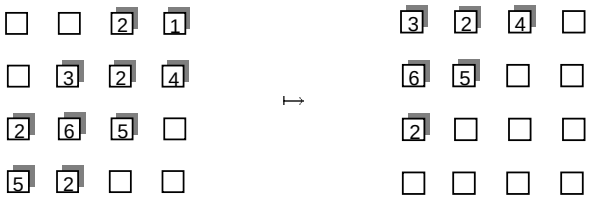

Fig. 5. Geometrical operation : `X <| \(i,j) |(i<j) .(i+1,j-1)`

Definition 5. Let $(v : \sigma)$ be a data field drawn in Z^n and h a bijection from $dom(v)$ onto Z^p. The *change of basis* defines the data field h `::` $(v : \sigma)$ as

$$h \texttt{ :: } (v : \sigma) \stackrel{def}{=} (v \circ h^{-1} : \sigma \circ h^{-1})$$

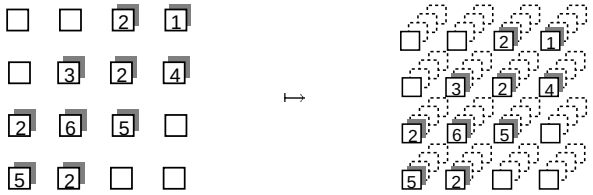

Fig. 6. Change of basis : `\(i,j) .(i,j,1) :: X`

Notice that both data fields `X` and `h :: X` draw the same multiset : they are said to be *equivalent data fields*.

3 Refinement

3.1 Definition

Refinement of specification is a powerful programming concept [Mor90]. In the theory Pei it consists in defining some operational order from an initial specification, in the following sense.

- As said in section 2.1, a program characterizes a function between input and output data fields. In order to get an operational definition of this function, i.e. the set of computations it involves, it is necessary to introduce a partial order on the data field elements. This order is defined by using the bijection σ from Z^n to Z^m of the considered data field.

  Definition 6. Let $<$ be any partial order on Z^m and $(v : \sigma)$ a data field whose bijection σ is a function from Z^n to Z^m. The relation $\vdash$ defined as

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

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

  The choice of an order $<$ on Z^m predetermines the operational definition of a program, for example by 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 σ which introduces a convenient operational order. These transformations lie on the change of basis operation.

– 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, as said before, this equivalence means that equivalent data fields draw the same multiset. Such a refinement defines a stepwise process going from a relation towards a function. It decreases the number of solutions of a system of equations, until only one solution remains : the system then defines a program and can define some operational order.

Definition 7. 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 `h :: X` for all occurrences of `X`.

3.2 Refinement rules

Since it is monotonic, the refinement induces a *refinement calculus* which is founded on the following rules on functional, geometrical, change of basis or superimposition operations :

```
Y = f1 o f2 |> X      ≡    Y = f1 |> (f2 |> X)
Y = X <| g1 o g2      ⊑    Y = (X <| g1) <| g2
Y = h1 o h2 :: X      ≡    Y = h1 :: (h2 :: X)
Y = (f |> X) <| g     ⊑    Y = f |> (X <| g)
Y = (h :: X) <| g     ⊑    Y = h :: (X <| h^-1 o g o h)
Y = f |> (X /+/ X')   ≡    Y = (f |> X) /+/ (f |> X)
Y = (X /;/ X') <| g   ⊑    Y = (X <| g) /;/ (X' <| g)
```

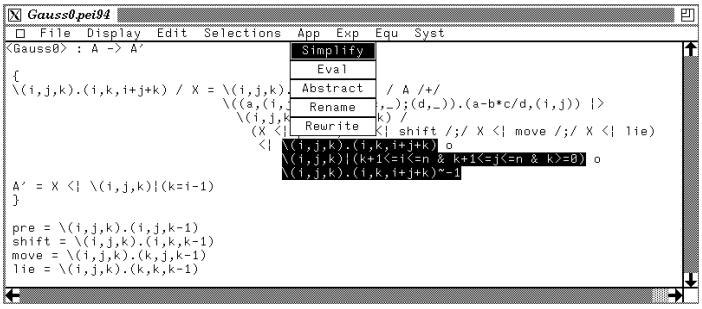

Fig. 7. A session in the PEI environment

In order to assist the end user in his programming task, PEI expressions and transformation rules can be carried out using the interactive PEI environment.

This tool implements equations and applies transformation rules in a CENTAUR session. It allows definition folding, relabeling, etc., and uses Maple for symbolic evaluations, such as function compositions, for example. Fig. 7 and 8 show snapshots of such a session to refine the Gaussian elimination program.

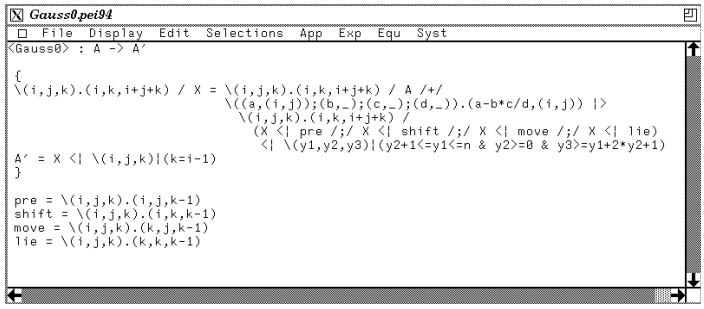

Fig. 8. The resulting snapshot in the Pei environment

4 Reduction operator

Previous issues clearly suggested some analogy between Pei and data parallelism concepts such as data alignment, communications or global operations on parallel variables. Moreover, the refinement calculus induces transformations which suit operational data parallel programs. Last concept in Pei, but not least, is the notion of *reduction* [VP94], which is one of the major issues in data parallel languages too.

Let us consider again the problem $s = \sum_{i \in 1..n} a_i$. It solution introduced a routing of the values of X by shifting from left to right, step by step. As we will see it later on, this is a particular way to implement the sum of n numbers, using a scan function : this is an example of refinement of a general non deterministic definition.

In a general way, reduction means some implementation of a n-ary operation from a binary one. At a specification level, or a high-level programming, the user should not mention the way this implementation operates. On this example, since addition is commutative and associative, the only thing to say is that the n values must be collapsed, whatever the sequence order is, to form a data field with a single value. This sort of routing is not a bijective function, but it must be defined from bijective ones in order to be implemented : such an implementation expresses some geometrical operation in Pei. So, a reduction in

Pei is a *geometrical inverse*, i.e. the inverse of a geometrical operation, denoted as g `;>` $(v : \sigma)$, which routes in any point z any sequence of values $v(y)$ such that $g(y) = z$. These issues are precised in the following definition and property.

Definition 8. Let $(v : \sigma)$ be a data field drawn in Z^n and g a partial function from $dom(v)$ to Z^n. A data field $(w : \sigma)$, denoted as g `;>` $(v : \sigma)$, is said to be a *geometrical inverse* of $(v : \sigma)$ by g iff

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

Property 1 *The data field* `g ;> X` *is equal to* `X <|` g^{-1} *iff* `g` *is bijective.*

Other properties of this geometrical inverse lead to these refinement rules, which define the reduction process :

```
Y = (g1+g2) ;> X     ⊑    Y = g1 ;> X /&/ g2 ;> X
Y = (g1 o g2) ;> X   ⊑    Y = g1 ;> (g2 ;> X)
```

Example 2. Let us consider again the summation of n numbers. This problem is expressed by the following specification :

```
sum : A -> S
  {A  =  dom :: A
   S  =  rec_add |> (route ;> A)
  }
  dom     =  \i |(1<=i<=n)
  route   =  \i |(1<=i<=n) .n
```

The geometrical inverse in the second equation maps some sequence of all the values of `A` at index `n`. These values are then added by `rec_add`, which is not specified here.

$\diamondsuit$

This example was just an introduction for the reduction. Of course this approach is similar in other favourite examples, such as the matrix product. It leads to a very elegant and concise data parallel form, which describes virtual broadcasts of the matrices and a reduction of the summation to compute each result. This form is achieved by a concise specification in Pei too, as following :

Example 3. The matrix product.

```
prod : (A, B) -> C
  {A =  cube :: A
   B =  cube :: B
   T =  mult |> (A <| spread_j /;/ B <| spread_i)
   C =  rec_add |> (reduce_k ;> T)
  }
```

```
cube      = \(i,j,k) |(0<=i<n & 0<=j<n & 0<=k<n)
spread_j  = \(i,j,k) .(i,0,k)
spread_i  = \(i,j,k) .(0,j,k)
reduce_k  = \(i,j,k) .(i,j,0)
```

The square matrices `A` and `B` are multisets aligned in Z^3, respectively on the ceil and on the side of a cube. The definition of `C` emphasizes

- broadcasts of `A` and `B`, respectively along the dimensions `j` and `i`,
- n^3 scalar multiplications,
- n^2 summations by reduction along the dimension `k`.

From this specification, executable programs can be refined, as routing and reducing macros do in data parallel languages. Such programs can be formally designed in Pei by applying the previous refinement rules. For example, the reduction can be transformed by introducing a geometrical operation which shifts the products along `k` :

```
prod : (A, B) -> C
  {A =  cube :: A
   B =  cube :: B
   T =  mult |> (A <| spread_j /;/ B <| spread_i)
   R =  (T <| last_k)  /+/  (T /;/ R <| succ_k)
   C =  rec_add |> (R <| first_k)
  }
  cube      = \(i,j,k) |(0<=i<n & 0<=j<n & 0<=k<n)
  spread_j  = \(i,j,k) .(i,0,k)
  spread_i  = \(i,j,k) .(0,j,k)
  last_k    = \(i,j,k) |(k=n-1)
  first_k   = \(i,j,k) |(k=0)
  succ_k    = \(i,j,k) |(k+1>0) .(i,j,k+1)
```

♢

5 Pei and Data Parallelism

The main features in Pei lie on a very little set of mathematical concepts. They express abstract geometrical domains for multisets of values. These domains are defined and manipulated by using very few formal mappings. It seams clear that all these issues meet the foundations of the data parallel programming model :

- data alignments look like change of basis and superimposition,
- global operations are defined by functional operations applied on data fields,
- broadcast and global communications are geometrical operations,
- reduction is an intrinsic feature of Pei.

Let us precise these points in the following sections.

5.1 Data alignments

The previous example of the matrix product has shown how the change of basis operation is used to align arrays on a virtual mesh. The following Pei equations

```
A =  cube :: A
B =  cube :: B
```

and the geometrical operations `A <| spread_j` and `B <| spread_i` with

```
spread_j = \(i,j,k) .(i,0,k)
spread_i = \(i,j,k) .(0,j,k)
```

lead to the following declarations of *template* and alignment directives [Hig93] :

```
!HPF$     TEMPLATE  CUBE (N,N,N)
          DIMENSION (N,N) :: A, B
!HPF$     ALIGN  A (I,K)  WITH  CUBE (I,0,K)
!HPF$     ALIGN  B (J,K)  WITH  CUBE (0,J,K)
```

Moreover, if two data fields have the same drawing, the superimposition operation in Pei defines a data field which expresses collections of data on the same *shape*.

Example 4. Summation of two matrices.

```
sum_mat : (A, B) -> C
  {A =  matrix :: A
   B =  matrix :: B
   C =  add |> (A /;/ B)
  }
  matrix  = \(i,j) |(1<=i<=n & 1<=j<=n)
  add   =  \(a;b) .a+b
```

The data field `(A /;/ B)` represents a grid of virtual processors storing local values of `A` and `B`. The result `C` is aligned on the same shape and n^2 scalar additions can run in parallel, as this data parallel code says [Thi90] :

```
shape [n][n] matrix;
float:matrix A, B, C;
C = A+B;
```

$\diamondsuit$

5.2 Global operations

A functional operation in Pei clearly defines a global operation on all the elements of a data field : it applies some function f, defined as $\backslash x \,|\, P(x) . f(x)$, on these elements, whose type is scalar or sequence of scalars if it results of a superimposition. The domain of f, defined by the predicate $P(x)$, expresses a *selection* statement, such as a `where` instruction in a data parallel language.

Example 5.

```
sup : (A, B) -> C
  {A =  dom :: A
   B =  dom :: B
   C =  test |> (A /;/ B)
  }
  dom   =  \i |(1<=i<=n)
  test  =  \(a;b) |(a<>0 & b<>0) .max(a,b)
```

Assuming `A` and `B` are aligned on a one-dimensional array, this statement is expressed in a data parallel language by :

```
where ((A!=0) && (B!=0))
       C = max(A,B);
```

♢

5.3 Communications and broadcasts

Previous examples emphasized *routings* in PEI : some of them defined regular dependencies, such as uniform translations. Other ones introduced broadcasts through a non injective function. Such geometrical operations have occurred in the second statement presented in Example 3 :

```
T =  ...  A <| spread_j /;/ B <| spread_i
R =  ...  R <| succ_k
```

with

```
spread_j    = \(i,j,k) .(i,0,k)
spread_i    = \(i,j,k) .(0,j,k)
succ_k      = \(i,j,k) |(k+1>0) .(i,j,k+1)
```

Such non injective routings are expressed by broadcast primitives in data parallel languages. This example would use notations like [Thi91] :

```
SPREAD (A, DIM=2, NCOPIES=N)
SPREAD (B, DIM=1, NCOPIES=N)
```

Conversely, uniform translations in PEI are expressed by implicit communications in a data parallel language, such as

```
R = [.][.][.+1] R;
```

for the considered example.

5.4 Scan and reduction functions

A powerful issue in data parallel languages leads to generate optimal code for reduction or scan functions. Considering the example of the summation of n numbers, we show how these features are supported by PEI.

The expression `route ;> A` in the statement of Example 2, where the function `route` is `\i |(1<=i<=n) .n`, can be refined by introducing a recursive definition of this function on the domain `1<=i<=n` :

```
route  =   \i |(i=n)
         + route o \i |(1<=i<n) .i+1
```

Since `\i |(1<=i<n) .i+1` is bijective and its inverse is `\i |(1<i<=n) .i-1`, we have the following refinement steps (from rules in Section 4 and Property 1) :

```
route ;> A
        ⊑  \i |(i=n) ;> A /&/ (route o \i |(1<=i<n) .i+1) ;> A
        ⊑  \i |(i=n) ;> A /&/ route ;> (\i |(1<=i<n) .i+1 ;> A)
        ≡  A <| \i |(i=n) /&/ route ;> (A <| \i |(1<i<=n) .i-1)
```

Let `pre = \i |(1<i<=n) .i-1` and `last = \i |(i=n)`, the last expression is

```
A <| last /&/ route ;> (A <| pre)
```

The same reasoning, leads to the following expression :

```
A <| last /&/
(A <| pre) <| last  /&/
route ;> ((A <| pre) <| pre)
```

and so on. We recognize then a recursive definition for this data field, following this property :

Property 2 *Let f be a function defined by the recursion $f = i + f \circ s^{-1}$, where i is the identity and s is bijective. The* PEI *equation* `Y = f ;> X` *is refined by the following statement, which involves the recursive definition of an intermediate data field* `T` *:*

```
T  =  X /&/ (T <| s)
Y  =  T <| i
```

Example 6. Refinement of a geometrical inverse by a reduction function.
This last property leads to the following statement in PEI, for the summation :

```
sum : A -> S
  {A  =  dom :: A
   T  =  A  /&/  (T <| pre)
   S  =  rec_add |> (T <| last)
  }
  dom   =  \i |(1<=i<=n)
  pre   =  \i |(1<i<=n) .i-1
  last  =  \i |(i=n)
```

Since for any data fields `X` and `Y`, `X /&/ Y = (X /+/ Y) /+/ (X /;/ Y)`, the second equation can be rewritten as

```
T = (A /+/ (T <| pre)) /+/ (A /;/ (T <| pre))
```

Since `A` is drawn on the domain `1<=i<=n` and `(T <| pre)` is drawn on `1<i<=n`, this expression can be simplified as

```
T = (A <| \i |(i=1)) /+/ (A /;/ (T <| pre))
```

Last, a scalar addition applied on every `(A /;/ (T <| pre)` can be substituted for the recursive function `rec_add` defined on the sequence mapped onto the point `i=n`. This leads to the program first presented in Example 1.

♢

This completes the refinement process to transform the previous program by introducing a reduction function : this development describes the way a compiler could generate code from such a macroscopic operation. Of course, other refinements can be proposed, which can express other parallel efficient implementations of the reduction, by defining other drawings and other recursive definitions of the geometrical inverse.

We focused on this example because this particular refinement shows another classical feature in data parallel languages : the prefix computations of *scan sets* by a `scan` function. Indeed, the data field `X` hereunder defines the prefix computations of the summation, as shown in Figure 1 :

```
prefix_sum : A -> X
       {A  =  dom :: A
        X  =  A <| first  /+/  add |> (A /;/ (X <| pre))
       }
       dom   =  \i |(1<=i<=n)
       pre   =  \i |(1<i<=n) .i-1
       first =  \i |(i=1)
       add   =  \(x;y) .x+y
```

5.5 Visual Programming in Pei

The equational approach of Pei and its refinement calculus offer a convenient framework to write specifications and derive or transform operational programs, especially data parallel programs.

In order to achieve in data parallel programming, Pei supports a visual interface called V.Pei, which supplies an interactive programming tool. An example of V.Pei session for the matrix product is shown in Figure 9.

Top window visualizes the data fields defined in the program, second window is used to monitor operation results on data fields and lower window receives the automatically generated PEI equations.

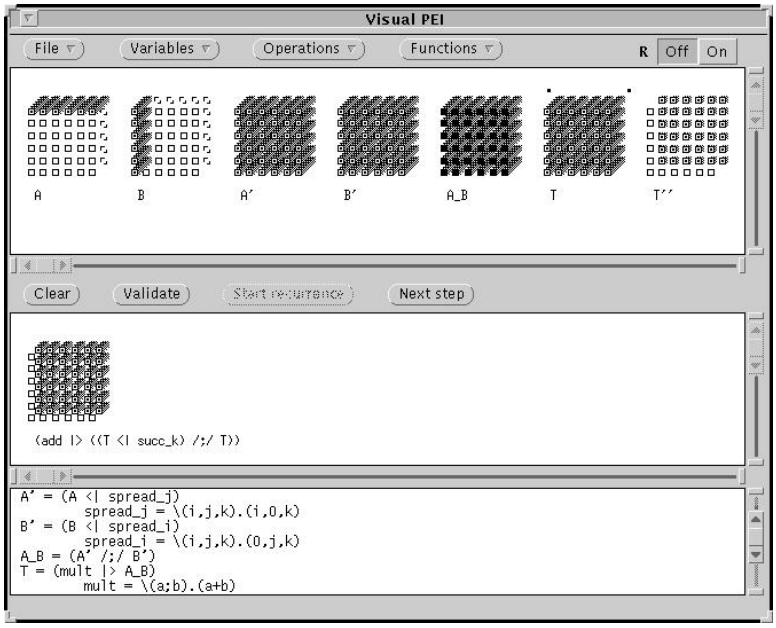

Fig. 9. A session V.PEI for the matrix product

This session shows `A` and `B` as being two data fields drawn in the plane, and then aligned on a cube. They are spread through their cube in order to perform the products : these two templates are aligned to form the pairs `a;b`, and the products operate. This data field `T` is transformed by a recursion whose definition appears in the draft window. The whole symbolic execution can be simulated step by step before validating the definition, and Figure 9 shows one step of the data field motion towards the rear of the cube. Once the definition validated the resulting PEI equation is produced and put in the lower text editor. The program we obtain is the second one presented hereabove in Example 3.

6 Conclusion

In the context of data parallelism, main issues of programming languages concern the data alignment for suitable global operations, such as broadcasts or

efficient reductions. PEI offers a convenient framework for all these concepts. These concepts and the refinement rules they induce are based on data field operation properties. These rules are syntactic rules which are included in the PEI transformational environment.

Such an environment can also benefit from the geometrical aspect of the objects in PEI : our current work is to specify, design and transform programs by using PEI and its visual interface V.PEI, in order to generate and optimize code in a data parallel language. This is the way we are working, in order to integrate three approaches :

- PEI itself and its refinement calculus,
- visual programming using V.PEI,
- effective code generation in a data parallel language.

References

[BM90] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–79, 1990.

[CCL91] M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design :* A foundation. Addison Wesley, 1988.

[Gel85] D. Gelernter. Generative communication in LINDA. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, January 1993.

[Len94] C. Lengauer. Loop parallelization in the polytope model. *to appear in PPL*, 1994.

[Mau89] C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.

[Mor90] C. Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.

[Raj93] S. Rajopadhye. LACS : A language for affine communication structures. Technical report, IRISA Rennes, 1993.

[Thi90] Thinking Machines Corp. *C* Programming Guide*, November 1990.

[Thi91] Thinking Machines Corp. *CM Fortran Programming Guide*, January 1991.

[Vio94] E. Violard. A mathematical theory and its environment for parallel programming. *to appear in PPL*, 1994.

[VP92] E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.

[VP93] E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, 1993.

[VP94] E. Violard and G.-R. Perrin. Reduction in PEI. *CONPAR'94, LNCS*, 1994.

This article was processed using the LaTeX macro package with LLNCS style