Publication 94-13

Reduction in PEI

E. Violard and G.-R. Perrin

Published in CONPAR'94, September 6-8, 1994, Linz (Austria), Lecture Notes in Computer Science 854, p. 112-123, Springer Verlag Ed.

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

Reduction in Pei

E. Violard and G.-R. Perrin

University of Franche-Comté,
F-25030 Besançon
Laboratoire d'Informatique
e-mail: {*violard, perrin*}*@comte.univ-fcomte.fr*

Abstract. Reduction is one of the major issues in data parallel languages : it can be defined as a rule of program refinement. This article presents a theoretical framework, called PEI, the foundation of a formalism for parallel programming, where this rule can easily be expressed and applied. This formalism is founded on a small but powerful set of primitives : they are three 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 design or optimization. We show how this approach allows to generalize the classical notion of reduction, by introducing a geometrical reduction and a functional one.

Keywords. Multiset, Parallel programming, Reduction operator, Refinement, Transformations.

1 Introduction

This article is concerned with program transformations. Many and various theories have been presented with the aim to derive correct programs on a target architecture. Some of them (ALPHA [Mau89], CRYSTAL [CCL91], LACS [Raj93], etc.) define a program as a set of recurrence equations. Other ones (GAMMA [BM90], LINDA [Gel85], UNITY [CM88], 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 in the same context. The goal is to benefit from the advantages of these two philosophies. PEI provides a formal frame, which permits to describe problem specifications and to reason on programs. It is founded on the notion of multisets of values. In order to address these values they are mapped onto geometrical domains, and form *data fields*. 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*).
In order to overcome the determinism of definitions expressed as recurrence equations, most of these languages (CRYSTAL or ALPHA [Lev91]) include a non-deterministic notation of operator, called *reduction operator*. Such an operator

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 $S = \sum_{i \in 1..n} a_i$. The computation of this sum can be done in different ways. Indeed, this operator is non-deterministic since it allows different choices in the calculation order. A sort of program refinement consists in reducing this non-determinism ("fan-in reduction" in CRYSTAL), in order to generate an efficient code for some target machine.
In PEI any such computation is defined through a *path*, which expresses the building of some *sequence* of all the values a_i. This characterizes the macro a compiler of any data parallel language would address in order to generate an efficient code for such a n-ary operation.
The aim of this article is to present the reduction in PEI, i.e. the way an efficient solution can be derived by a refinement process. Reduction operators are presented as *inverse operations* of basic operations. PEI defines two kinds of reduction : a *geometrical reduction*, which generalizes the notions of reduction in ALPHA and CRYSTAL, and a *dual* reduction, called *functional reduction*, whose interest is presented on a simple example.

2 Definition of the formalism Pei

2.1 Specifications and programs

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*, i.e. a geometrical representation of a multiset. These expressions are defined from data fields 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.

Example 1. Part (a) is the specification of a problem which calculates the square root of the values, between 0 and 9, of a given multiset :

```
SquareRoot : X -> Y                SquareRoot : X -> Y
    { X = dom |> X                     { X = dom |> X
      sqr |> Y = X                       Y = sqrt |> X
    }                                  }
   dom = \x |(0<=x<=9)                dom  = \x |(0<=x<=9)
   sqr = \y .y*y                      sqrt = \x .x^0.5

         (a)                                (b)
```

The first equation defines a precondition on the input data field `X` to select values between 0 and 9. The second one defines a postcondition on the output data field `Y` : it means that applying function `sqr` on `Y` results in `X`.

A specification is called a *program* if its system of equations defines a function, i.e. the system has at most one solution. Such a program may be derived by constraining the relation or by introducing an inverse operation in order to get an explicit definition of the outputs. For example, the previous specification may yield the program presented in part `(b)`.

2.2 Data fields

In this theory, objects called data fields, represent multisets in a geometrical way by associating a coordinate in Z^n with any of their values.

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

There are an infinity of geometrical representations for a given multiset. They differ one another by a bijection. This is expressed by the concept of *data field*, which recalls that a representation is always drawn within a bijection.

Definition 2. A *data field* `X` is a pair $(v : \sigma)$, composed of a geometrical representation 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 geometrical representation of M_X.

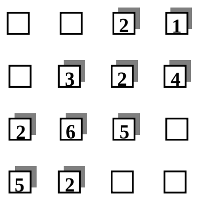

Fig. 1. An example of data field

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 *sequences* of values. Restrictions of this operation to the disjunction and intersection of drawings are respectively called *sum* and *product*, and denoted as `/+/` and `/;/`. In the following we will use three operators defined on sequences : an associative constructor denoted as ; and the two classical functions `tail` and `head`.

2.3 External operations on data fields

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 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 $\backslash x \,|\, \mathtt{true} \,.\, f(x)$, and $\backslash x \,|\, P(x)$ for $\backslash x \,|\, P(x) \,.\, x$.
Let us recall a few useful concepts about functions :

- An *inverse function* can be associated with any function f. If f is bijective, its inverse, denoted as f^{-1}, satisfies $f^{-1}(f(x)) = x$ for any $x \in dom(f)$. If f is not injective, it is still possible to specify a function g by the requirement that $f(g(y)) = y$ for any y such that $g(y) \in dom(f)$. In general such an inverse function is not unique, and in the absence of any more deterministic constraints, each definition is satisfactory : this will be used later as a method of specification, which gives no hint as to how a deterministic executable definition might be formulated. The way such a program can meet the specification is the matter of a refinement calculus.
- A function f can be defined as a partition f_1+f_2+...+f_n of functions where the f_i are defined on disjunctive sub-domains.
- 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

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

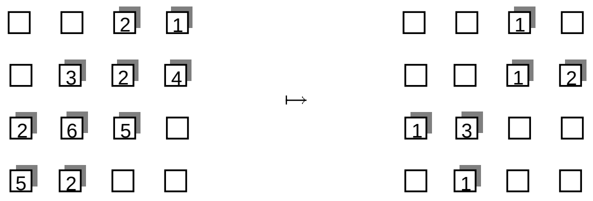

Fig. 2. Functional operation : `\x |(x mod 2 =0) .x/2 |> X`

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

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

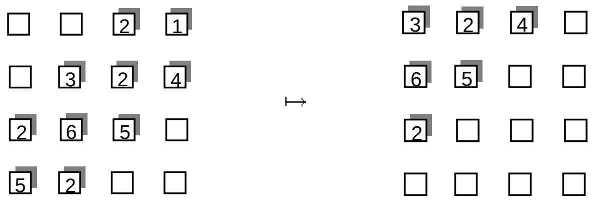

Fig. 3. Geometrical operation : `X <| \(i,j) |(i<j) .(i+1,j-1)`

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

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

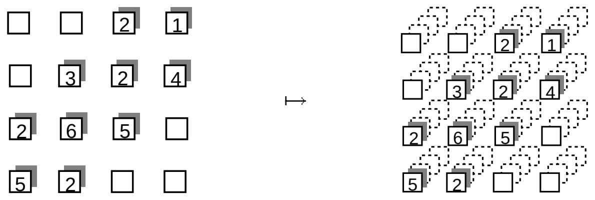

Fig. 4. Change of basis : `\(i,j) .(i,j,1) :: X`

3 Inverse operations

As previously said, an elegant expression for specifications may use inverse functions, to be refined in a deterministic executable program. This leads to introduce inverse operations in PEI. Of course the inverse of a change of basis operation is a change of basis defined from the inverse of its bijection h. Other inverse operations are crucial since they define *reduction operators*. Their definition requires more explanations because they are non-deterministic operations if associated partial functions are not bijective.

Definition 6. Let $(v : \sigma)$ be a data field whose values are on V and f a partial function from W to V. A data field $(w : \sigma)$, denoted as $(v : \sigma)$ `<;` f, is said to be a *functional inverse* of $(v : \sigma)$ by f iff

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

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

Definition 7. 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

- $dom(w) = \{z \in Z^n \mid \exists y \in dom(v) \cdot g(y) = z\} = g(dom(v))$
- $w(z)$ is any sequence formed with the values of $\prec v(y)\ ,\ g(y) = z \succ$

Property 2 *The data field* g `;> X` *is equal to* `X <|` g^{-1} *iff* g *is bijective.*

4 Refinement

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.
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 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 σ from Z^n to Z^m of the considered data field.

Definition 8. 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 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 σ which introduces a convenient operational order. These transformations lie on the change of basis operation.

Example 2. The previous program may be transformed as following, to define an operational order :

```
SquareRoot : X -> Y
    { X = \(p,t) |(t=0) :: X
      Y = (sqrt |> X) <| \(p,t) .(p,t-1)
    }
   sqrt = \x .x^0.5
```

The first equation applies a change of basis which defines the drawing of the input data field `X` onto a set of points $(p, 0)$ in Z^2 : p expresses a virtual processor index and t means a computation instant. The second equation tells that all the output values are computed at the same instant, from the initial values of `X`. Such a program may be translated in a common parallel language as :

```
x,y : array (...) of real;
forall p do
   y(p) := x(p)^0.5
end forall;
```

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 different data fields represent the same multiset. We define two kinds of refinement. They form the links of the derivation chain $\mathcal{S}^0 \sqsubseteq \mathcal{S}^1 \sqsubseteq , \ldots , \sqsubseteq \mathcal{S}^n$ where $\mathcal{S}^0$ is a specification and $\mathcal{S}^n$ a program :

- *Denotational refinement.* It defines a stepwise process going from a relation towards a function. It decreases the number of solutions of a system of equations, until there is only one solution : then, the system defines a program. Such a refinement consists in inversing some geometrical or functional operations in order to make the specification executable. Reduction of operators is an example of such a refinement.
- *Operational refinement.* It defines a schedule and a mapping of the computations with regard to some target architecture. It consists in choosing a convenient geometrical representation by using a change of basis. Such a refinement defines equivalent data fields.

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

These two kinds of refinement may apply whatever the step in the design of a solution. The previous example `SquareRoot` emphasized first a denotational refinement and then an operational one. In fact, the last statement results from a denotational refinement too, since the precondition was weakened.
Since it is monotonic, the denotational refinement definition induces a *refinement calculus* which is founded on the following rules on functional, geometrical, change of basis or inverse operations :

```
Y = f1 o f2 |> X        ≡    Y = f1 |> (f2 |> X)                 (1)
Y = X <| g1 o g2        ⊑    Y = (X <| g1) <| g2                 (2)
Y = h1 o h2 :: X        ≡    Y = h1 :: (h2 :: X)                 (3)
Y = f |> (X /+/ X')     ≡    Y = (f |> X) /+/ (f |> X)           (4)
Y = (X /;/ X') <| g     ⊑    Y = (X <| g) /;/ (X' <| g)          (5)
Y = (f |> X) <| g       ⊑    Y = f |> (X <| g)                   (6)
Y = (h :: X) <| g       ⊑    Y = h :: (X <| h^-1 o g o h)        (7)
Y = (g1+g2) ;> X        ⊑    Y = g1 ;> X /&/ g2 ;> X             (8)
Y = (g1 o g2) ;> X      ⊑    Y = g1 ;> (g2 ;> X)                 (9)
Y = X <; (f1+f2)        ⊑    Y = X <; f1 /&/ X <; f2             (10)
Y = X <; (f1 o f2)      ⊑    Y = (X <; f1) <; f2                 (11)
```

5 Geometrical and functional reductions

Reduction operator is one of the major issues in data parallel languages : it expresses, in a global way, a n-ary operation from a binary one. In our formalism, it means an operation defined on a sequence of n values, whatever the sequence order if the original binary operation is commutative and associative, i.e. what we called an inverse operation. As we introduced two kinds of inverses, we define two kinds of reduction : the *functional reduction* and the *geometrical* one.
The classical notion of reduction is the geometrical one. We introduce this reduction first with an example, after this short explanation : let g be a non-injective function and let us consider a geometrical inverse data field g ;> $(v : \sigma)$. One can define an inverse function g_i such that $g(g_i(z)) = z$ for $g_i(z) \in dom(g)$. So, there are as many functions g_i as there are different $x_1, ..., x_k \in dom(g)$ nailed as a sequence on the same point by g. A binary operation can apply step by step on such a sequence by defining a path in Z^n, going from these $x_1, ..., x_k$ to z. Such a path can be defined by introducing geometrical operations, i.e. as Property 2 says, by decomposing or partitioning g in bijective components (using refinement rules (8) or (9)). This is illustrated with the classical example of the n-ary summation.

5.1 An example of geometrical reduction

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

```
Sum : A -> S
     { A  =  \i |(1<=i<=n) :: A
       S  =  sum |> (\i |(1<=i<=n) .0 ;> A)
     }
     sum =   \x |(x = tail(x)) .x
           + \x |(x<> tail(x)) .head(x)+sum(tail(x))
```

where the first equation defines a precondition on the input data field `A` by defining its drawing onto exactly n points. The geometrical inverse of the second equation puts some sequence of all the values of `A` at index `0`. These values are then added by using the operation `sum`.
Other equivalent specifications can be written by choosing an other drawing for `A` and an other target index for the sequence. For example :

```
Sum : A -> S
     { A  =  \(i,j) |(1<=i<=n & j=0) :: A
       S  =  sum |> (path ;> A)
     }
     path = \(i,j) |(1<=i<=n & j=0) .(n,n)
```

The first equation applies a change of basis to redraw the input data field `A` in Z^2. The second one moves the target index to `(n,n)`.
Neither this last specification, nor the previous one, defined a deterministic way to form the sequence of values of `A` : hence, function `path` is not bijective,

so the definition of the geometrical inverse of `A` by `path` is non-deterministic. This is a way to mean that operator $\sum$ is non-deterministic. Reduce this non-determinism, in order to implement this specification, consists in applying some refinement rules on inverse operations : these rules consist in decomposing `path` to make a *reduction path* explicit by introducing bijective functions. Since function `path` can be decomposed as

```
path = lowdiag o ... o lowdiag o \(i,j) |(1<=i<=n & j=0)
```

with `lowdiag = \(i,j) |(i=j) .(i+1,j+1) + \(i,j) |(i>j) .(i,j+1)`, the refinement rule (9) gives :

```
S  =  sum |> (lowdiag ;> ... (lowdiag ;>
                   (\(i,j) |(1<=i<=n & j=0) ;> A)) ...)
```

Notice that this definition of `path` can be expressed in a recursive style as :

```
path     = rec-path o \(i,j) |(1<=i<=n & j=0)
rec-path =   \(i,j) |(j=n)
           + \(i,j) |(j<n) . rec-path o lowdiag (i,j)
```

This statement defines a reduction path from points `(i,0)` to point `(n,n)`. This path is still non-deterministic since each superimposition defines a sequence in an undefined order. The next step will precise this order by partitioning the domain and applying rule (8) :

$$\texttt{Y = lowdiag ;> X} \sqsubseteq \texttt{Y = low ;> X /\&/ diag ;> X}$$

where
```
low  = \(i,j) |(i>j) .(i,j+1)
diag = \(i,j) |(i=j) .(i+1,j+1)
```

Since these functions are bijective, the recursive definition of `path` leads to a recursive definition of an intermediate data field `T`. This yields the following specification :

```
Sum : A -> S
    { A  =  \(i,j)|(1<=i<=n & j=0) :: A
      T  =  (low ;> T /&/ diag ;> T) /+/ A
      S  =  sum |> (T <| top)
    }
   low  = \(i,j) |(i>j) .(i,j+1)
   diag = \(i,j) |(i=j) .(i+1,j+1)
   top  = \(i,j) |(i=n & j=n)
```

This specification is now a deterministic program because the inverse operations are bijective. Then, from Property 2, we can write :

```
Sum : A -> S
    { A  =  \(i,j) |(1<=i<=n & j=0) :: A
      T  =  (T <| up /&/ T <| antidiag) /+/ A
      S  =  sum |> (T <| top)
    }
   up       = \(i,j) |(i>j-1) .(i,j-1)
   antidiag = \(i,j) |(i=j) .(i-1,j-1)
   top      = \(i,j) |(i=n & j=n)
```

Finally, the following program is obtained by distributing addition on the path. This completes the reduction :

```
Sum : A -> S
     { A  =  \(i,j) |(1<=i<=n & j=0) :: A
       T  =  add |> (T <| up /&/ T <| antidiag) /+/ A
       S  =  T <| top
     }
    up        = \(i,j) |(i>j-1) .(i,j-1)
    antidiag = \(i,j) |(i=j) .(i-1,j-1)
    top       = \(i,j) |(i=n & j=n)
    add       = \(a;b) .a+b
```

This example was just an introduction for geometrical reduction. Of course this approach is similar in other favourite examples, such as the matrix product. This problem 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 :

```
Prod : (A, B) -> C
   {
   A =  matrix :: A
   B =  matrix :: B
   C =  sum |> (reduce-k ;> (mult |> (A <| spread-j /;/ B <| spread-i)))
   }
matrix   = \(i,j,k)|(1<=i<=n & 1<=j<=n & k=0)
spread-j = \(i,j,k) .(i,k,k)
spread-i = \(i,j,k) .(k,j,k)
reduce-k = \(i,j,k) .(i,j,0)
mult     = \(a,(i,k));(b,(k,j)) .(a*b,(i,j))
sum      =   \x |(x= tail(x)) .x
           + \x |(x<>tail(x)) .
                 (car(head(x))+car(sum(tail(x))),cdr(head(x)))
car      = \(x,y) .x
cdr      = \(x,y) .y
```

The square matrices `A` and `B` are multisets whose values are pairs of the form $(a_{i,j},(i,j))$ [CM88, Cre91], whose elements are accessed by functions `car` and `cdr`. A change of basis aligns `A` and `B` in Z^3. The definition of `C` emphasizes the broadcasts of `A` and `B`, respectively along the dimensions `j` and `i`, and the summations by reduction along the dimension `k`.
From this specification, executable programs can be refined, as routing and reducing macros do for a data parallel language. Such programs can be formally designed in PEI by applying the previous technique.

5.2 An example of functional reduction

Previous examples dealt with classical reduction, i.e. geometrical reduction. Thanks to the orthogonality of the concepts we introduced in PEI, a functional reduction can be defined too. Here is an example : the problem is to search a root of a discrete function f in an interval (a, b), assuming convenient hypotheses. The well-known partitioning algorithm can be specified as following :

```
Root : I0 -> R
     { I = I0 /+/ inter |> (I <| pre /;/ F <| pre)
       F = \(a,x,b) .f(x) |> T
       T = head |> (I <; \(a,x,b) |(a<x<b) .(a,b))
       R = \(a,b) |(a=b) .a |> I
     }
    pre   = \i |(i>0) .i-1
    inter =   \((a,x,b);fx) |(fx> 0) .(x,b)
            + \((a,x,b);fx) |(fx<=0) .(a,x)
```

A data field `I` is defined from an input data field `I0` : its values are successive intervals `(a,b)` begining with the initial one, until a last one, such that `a=b`. These intervals are defined by applying function `inter` on a sequence composed of a triplet `(a,x,b)` and the value `f(x)` of the function `f` applied on `x` : this `x` may be any point such that `a<x<b`. The functional inverse in the equation defining data field `T` expresses this non-deterministic aspect : the value of `T` is any of these triplets, e.g. the first one, which is accessed by the function `head` on the sequence of all the `(a,x,b)`.
From this specification, one can define this partition of the functional inverse :

```
\(a,x,b) |(a<x<b) .(a,b) =   \(a,x,b) |(x=(a+b)/2) .(a,b)
                           + \(a,x,b) |(a<x<b & x<>(a+b)/2) .(a,b)
```

The refinement rule (10) then applies on the definition of `T` :

```
T = head |> (I <; \(a,x,b) |(x=(a+b)/2) .(a,b)  /&/
             I <; \(a,x,b) |(a<x<b & x<>(a+b)/2) .(a,b))
```

which can be simplified in `T = I <; \(a,x,b) |(x=(a+b)/2) .(a,b)`, i.e. from Property 1, `T = \(a,b) .(a,(a+b)/2,b) |> I`, which defines a program.

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 : for example, this paper emphasizes reduction. It presents the reduction as a program refinement, which involves inverse operations. This approach is more clear than different concepts in many languages, such as predefined reduction operators in programming languages, or associative and commutative reductions in ALPHA or CRYSTAL : for example, the notion of graph in CRYSTAL expresses the notion of reduction path in PEI. An other point in favour of PEI is the unifying approach of concepts such as fan-in and fan-out reduction in CRYSTAL[1]. Last,

[1] CRYSTAL defines a "fan-out reduction" which relies on the concept of broadcast. This is expressed by the refinement rule (2) in PEI.

the orthogonality of the concepts introduced in PEI, induced the original and powerful notion of functional reduction.
All these concepts and the refinement rules they induce are based on properties of the data field operations. These rules are syntactic rules which can be included in a transformational environment, by using any formal calculus motor such as Maple [Map89] for example. Such an environment can also benefit from the geometrical aspect of the objects in PEI : our objective is to specify, design and transform programs by using graphic tools, in order to generate and optimize code 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.

[Cre91] C. Creveuil. *Techniques d'analyse et de mise en oeuvre des programmes* GAMMA. PhD thesis, U. Rennes, 1991.

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

[Lev91] H. Leverge. Reduction operators in ALPHA. Technical report, IRISA, November 1991.

[Map89] Distribution INRIA - Rocquencourt. *Maple Reference Manual, 4th Edition*, March 1989.

[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.

[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.

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