5

Refinement of data parallel programs in PEI

E. Violard, S. Genaud and G.-R. Perrin
ICPS - University Louis Pasteur, Strasbourg
Pôle API - Bd S. Brant, F-67400 Illkirch
{violard,genaud,perrin}@icps.u-strasbg.fr

Abstract

Parallel programs mainly differ from sequential ones in that they include geometrical aspects involved by the hardware architecture. We present in this paper the PEI formalism, which enables to take into account both the geometrical and functional aspects of programs. It provides a refinement calculus mainly used to transform the geometrical characteristics of parallel programs, and we show how it may apply on data parallel programs, in particular for data alignments.

Keywords

Data fields, Data-parallelism, Program transformation, Refinement.

1 INTRODUCTION

Parallel programming is a major challenge for handling efficient computations. It involves two technological issues: a *program* expressed in some dedicated language which supports a parallel programming model, and a *computer* and the parallel execution model it implements. Ideally the programming language should be architecture independent whereas the computations efficiency requires a strong-related architecture implementation.

The relationship between these issues is a matter of compiling techniques as well as programming capabilities: in scientific applications for example, on the one hand parallelizing compilers are able to automatically transform the iteration space, assuming conditions such as the affinity of dependencies and of loop bounds; on the other hand the programmer may help the compiler by using *directives* to indicate a "good" data alignment —that is a data placement that minimizes interprocessor communications. All these techniques are based on *geometrical* transformations either of the iteration space or of the index domains of arrays. Especially, they are of particular interest in the *data parallel programming* model, implemented by programming languages such as High Performance Fortran (HPF 1993) or C*(Thi 1990).

In some sense it shows that, beside a classical functional point of view on programs, geometrical issues in parallel programming or parallelizing compilation have

to be considered of main importance for the mastery of efficient computations. The geometrical approach entails an abstract manipulation of array indices, to define and transform:

- the data dependencies in programs,
- the way the data are, or are not, locally accessible,
- their expansion in a multidimensional space of virtual processors,
- *etc.*

The solution might be the use of some methodology for parallel program development, in the same spirit as what is done with the Bird-Meertens Formalism (Bird 1987, Skillicorn 1993). In addition, given the previous considerations on geometry, a specialised methodology might well be needed.

The PEI formalism (in some sense similar to ALPHA (Mauras 1989) or CRYSTAL (Chen, il Choo & Li 1991)) was originally defined (Violard & Perrin 1992, Violard & Perrin 1993) to describe and reason on parallel programs and their implementation. It includes a refinement calculus that makes possible to transform statements by associating algebraic laws and symbolic evaluation of functions with the classical geometrical foundations in this area. Using PEI, the programming activity may refer to a very small set of primitives to design programs. They are:

- the placement of value items in some discrete reference domain, which relates to data dependencies that possibly induce interprocessor communications,
- the definition or the change of the reference domain itself, which enables to change the program structure,
- the application of functions on the value items, which defines the operations carried out by a program.

The relationship between a PEI statement and a data parallel program is quite obvious since PEI expresses, in an abstract way, parallel variables, global operations and communications, and data alignment. Therefore PEI provides a good framework to design and transform data parallel programs (Genaud, Violard & Perrin 1995), and we illustrate how its refinement calculus may be used to deal with the crucial aspect of data alignments.

The paper is organized as follows: section 2 explains the PEI concepts and notations. Section 3 is concerned with the semantic definitions associated with PEI constructs, and section 4 presents how refinement is defined in the formalism. Finally, we illustrate most of the formalism issues, including design and refinement considerations on the *Gaxpy* example drawn from the numerical computation field.

2 DEFINITION OF THE FORMALISM PEI

2.1 PEI data fields

Generally speaking, we can consider a problem as a relation between input and output *multisets* of values. Of course, programming may imply to put these values in a convenient organized directory, depending on the problem terms. In scientific compu-

tations for example, items such as arrays are functions on indices: the index set, that is the reference domain, is a part of some $\mathbb{Z}^n$. In PEI such a multiset of value items mapped on a discrete reference domain is called a *data field.*

Let us consider the multiset $\{1, 2, -3, 1\}$. A possible way to map this multiset is to choose indices, for instance $\{0, 1, 2, 3\}$ of $\mathbb{Z}$, to refer to each of the values. This mapping is shown on figure 1(a) and defines the data-field called `V`. Obviously, this multiset might be mapped in a different manner, for example onto points whose indices (i, j) in $\mathbb{Z}^2$, are such that $(0 \leq i, j \leq 1)$ (figure 1(b)). Such a mapping thus defines another data field, let us say `M`. Although their mappings are different, the two data fields represent the same multiset of values and therefore we say they are *equivalent.*

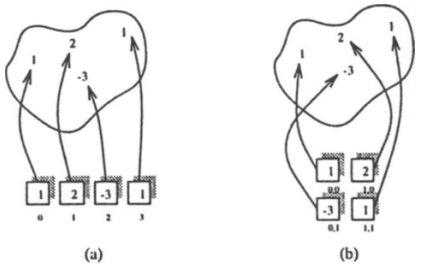

Figure 1 Two different mappings of a multiset of values

Formally, there exists a bijection from the first arrangement to the second one, namely $\sigma(i) = (i \ mod \ 2, i \ div \ 2)$. This relation is expressed in PEI through the equation:

```
        M = align::V
where   align = λ(i)|(0≤i≤3).(i mod 2,i div 2)
```

Any PEI program is composed of unoriented equations, each of them connecting two data field expressions. On the example, `M` and `align::V` have the same set of value items, placed in the same fashion in the same reference domain.

2.2 PEI operations

Expressions are defined by applying operations on data fields. The operations are second-order functions, and fall into three categories:

- the operation used in the expression `align::V` modifies the reference domain onto which values are mapped. It is called *change of basis* and is denoted by `::`

- another operation "moves" values in the reference domain. It is called *geometrical operation* (or routing), and is denoted by $\lhd$. Figure 2(a) shows the mapping of values of the data field `W` defined as `W = V ◁ shift`. The function `shift`

shifts values one place cyclically to the right and is written in PEI as `shift` = `λ(i)|(0<=i<4).((i-1) mod 4)`. If the function is not injective the operation expresses a broadcast. For instance `W` = `V` ◁ `spread`, where `spread` = `λ(i)|(0<=i<4).(0)` means the value mapped at index point 0 in `V` is broadcasted to index points 0 to 3, to form the data field `W` as shown on figure 2(b).

– the third operation computes the values of a data field, and is called *functional* operation. It is denoted by ▷ and performs an element-wise computation on the data field. For example, `W` = `inc` ▷ `V`, where `inc` = `λ(a).(a+3)`, defines a data field whose values are computed from the `V` values having the same indices (figure 3).

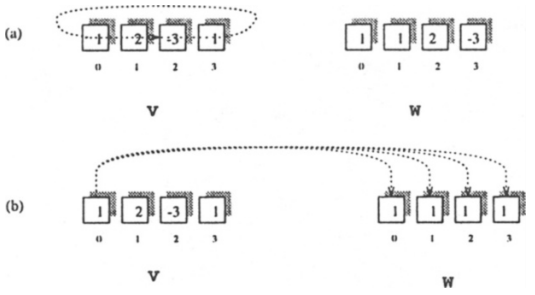

Figure 2 Geometrical operations: (a) one-to-one relation (b) broadcast

The geometrical operation has an inverse in PEI. It defines the *geometrical reduction*, denoted by ;▷ . It may be used for example, to gather all the values of `V` at a particular index point. For instance, the data field `W` verifying `W` = `merge` ;▷ `V`, where `merge` = `λ(i)|(0<=i<4).(0)` has all its values mapped at index point 0, listed in a sequence of arbitrary order. Figure 4 shows a possible solution for `W`.

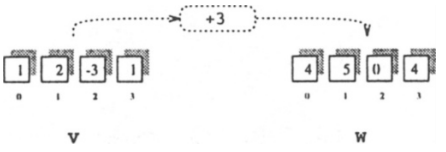

Figure 3 `W` defined by a functional operation applied on `V`

The geometrical reduction is defined as the inverse of the geometrical operation since if it applies on a bijective function it is equal to the geometrical operation applied on the inverse function.

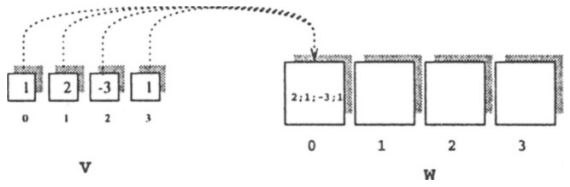

Figure 4 `W` is defined by a reduction operation applied on `V`

Last, an internal operation is defined on data fields. It is called *superimposition* and denoted by `/&/`. The superimposition of several data fields results in a new data field whose values are sequences. Each sequence is the concatenation of values mapped at the same indices. Figure 5 shows the result of superimposing `X` and `Y`.

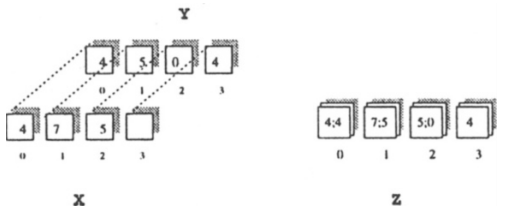

Figure 5 `Z` is defined by `Z = X/&/Y`

2.3 PEI programs and Syntactic issues

A PEI *statement* is a set of equations which expresses the relation between input and output data fields. It is a system of equations with the input data fields being the parameters and the output data fields being the unknowns.

A solution is a set of output data fields verifying the system. In the PEI theory, only some statements are programs.

Definition 1 *A* PEI *statement is a* program, *if for any given input data fields set, there exists at most one solution.*

The definition implies that we do not consider statements having non-deterministic solutions as programs.

Let us now make precise some of the syntactic features of the formalism. As typographic conventions, we will use `A`, `B`, `X`, *etc.* for data fields, whereas `f`, `g`, *etc.* denote functions. The general form of a PEI statement includes a header composed of the tuples `D` and `R` of input and output data fields, and an equation set S:

$$\begin{matrix} \mathtt{P} : \mathtt{D} \mapsto \mathtt{R} \\ \{ \ S \end{matrix}$$

We will also use the shorthand $\mathtt{P}\{S\}$ when D and R do not matter.

To improve the readability, only function names appear in the equations, their definition being reported outside the system. Functions are written using a notation derived from the lambda-calculus: a function f of domain $dom(f) = \{x \mid P(x)\}$ and image $img(f) = \{f(x) \mid P(x)\}$ is denoted as $\lambda(x) \mid P(x).f(x)$.

We write $f|_{dom(g)}$ for the part of f defined on the domain of g, so that $f|_{dom(g)} = g$ implies that f and g are equal on $dom(g)$. A function f defined on disjunctive sub-domains is denoted as $f1 \# f2$, and the domain of a composed function $f \circ g$ is $\{x \in dom(g) \mid g(x) \in dom(f)\}$.

We will also use the built-in function `id`, which only matches one-element sequences. The application of `id` on one element returns this element. It is frequently used to define recursive computations, like in the recursive summation of a sequence of numbers:

```
sum = id # λ(a;b).(a+sum(b))
```

The function `sum` returns the element for one-element sequences and for sequences made of several elements, it returns the first element plus the sum of the rest of the sequence.

2.4 Examples

Let us illustrate through a few examples what PEI programs look like:

Example 1 Addition of a matrix and its transpose

```
MatSum : A ↦ C
⎧ A = matrix::A
⎨ T = A ◁ transp
⎩ C = add ▷ (A /&/ T)
matrix  =  λ(i,j)|(0<=i,j<n).(i,j)
transp  =  λ(i,j)|(0<=i,j<n).(j,i)
add     =  λ(a;b).(a+b)
```

– The first equation expresses a pre-condition on the input data field A. It means that A is invariant by applying the change of basis defined by `matrix`. The change of basis is the identity on the square domain $\{(i,j), 0 \leq i,j < n\}$ of $\mathbb{Z}^2$: as we will see, it means that the discrete domain onto which the values of A are placed is bounded to the square.

– The second equation defines T from A by applying a geometrical operation. It means that the values of T mapped on points (i,j) are those of A mapped on (j,i).

- The third equation defines the output data field C. The right side defines the superimposition of A and T, *i.e.* the set of pairs (a;b), where a ∈ A and b ∈ T. The functional operation above applies the function add to each value of T.
◇

Example 2 Palindromes detection

A vector A of size n is a palindrome if $A_i = A_{n-1-i}$, $\forall i \in [0 \ldots n-1]$. The program below pairs the symmetric values respectively to the middle of the input vector A, and then compares equality of values in each of the pairs, resulting in data field B. To determine whether A is a palindrome or not, all comparison results defined by B must eventually be gathered. This is done through the definition of R. If all results are true then A is a palindrome. The way B values are put together can be written in at least two ways. Here is a first version:

```
Palind1 : A ↦ R
⎧ A = vector::A
⎨ B = equ ▷ (A /&/ A ◁ opp)
⎩ R = and ▷ (red ;▷ B)
 equ     =  λ(a;b).(a=b)
 vector  =  λ(i)|(0≤i<n).(i)
 opp     =  λ(i)|(0≤i<n).(n-1-i)
 red     =  λ(i)|(0≤i<n).(0)
 and     =  id # λ(a;b).(a&(and b))
```

In this version, we use a reduction operation that builds a sequence of all the boolean values of B and put it at index point 0 in R. All sequence elements are then logically ANDed. Reduction is thus a straightforward means to gather values, but it is sometimes necessary to indicate an explicit path for the values to be moved.

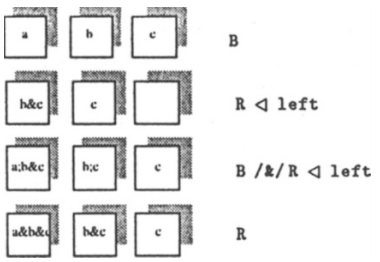

Figure 6 Given B, R is a solution of Palind2 last equation

```
Palind2 : A ↦ R
⎧ A = vector::A
⎨ B = equ ▷ (A /&/ A ◁ opp)
⎩ R = band ▷ (B /&/ R ◁ left)
equ    = λ(a;b).(a=b)
vector = λ(i)|(0≤i<n).(i)
opp    = λ(i)|(0≤i<n).(n-1-i)
left   = λ(i)|(0≤i<n-1).(i+1)
band   = id # λ(a;b).(a&b)
```

Here, R is recursively defined as the application of a logical AND on the superimposition of B and R shifted one position left. As we can see on figure 6, the data field R is a solution of the last equation because it maps at index point 0 the sequence of all elements of B, the sequence minus one element at index point 1, *etc.* We thus have a data field R very similar to the one in Palind1, in that the evaluation of all B values is in both cases mapped at point 0 (though the order of evaluation might be different). ◇

3 SEMANTICS

PEI semantics is founded on the notion of discrete domain associated with a multiset. As seen earlier, a data field maps the values of a multiset V onto a domain, such that each value can be indexed by a point of the domain. The mapping is therefore a function $v : \mathbb{Z}^n \mapsto V$.

In the previous section we pointed out that V, whose values $\{1, 2, -3, 1\}$ were mapped on the interval [0..3] of $\mathbb{Z}$, can be deduced from M using the σ bijection. Generally speaking, given a multiset V of values, and the mappings $v : \mathbb{Z}^n \mapsto V$ and $v' : \mathbb{Z}^p \mapsto V$, we have the relation $v = v' \circ \sigma$, where σ is a bijection which maps $dom(v)$ to $dom(v')$.

It means that any mapping of a given multiset is arbitrary and always can be deduced from a *reference mapping* (a given processor grid for example) by applying a bijection. By integrating this bijection into the definition of data fields, we express the relation between two mappings of a same multiset. Hence, a data field is a pair of functions $(v : \sigma)$. Moreover, we impose the following constraint: $dom(v)$, called *value domain* must be included in the domain of the bijection σ, called the *reference domain*, so that any transformation of a mapping apply to all the values of the data field. For instance, considering M on figure 7 as a reference mapping representing a processor grid, the value 2 mapped at index point 1 by $V = (v_V : \sigma_V)$ would be computed by processor $\sigma_V(1) = (1,0)$.

Definition 2 *Let V be a multiset of values. Let v be a mapping $\mathbb{Z}^n \mapsto V$ and σ a bijection $\mathbb{Z}^n \mapsto \mathbb{Z}^p$, then $(v : \sigma)$ defines a* data field *if $dom(v) \subseteq dom(\sigma)$.*

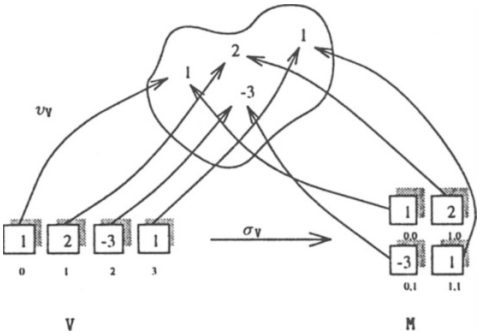

Figure 7 Relation between two equivalent mappings

3.1 Operation semantics

The data field definition enables us to formally describe the operations in PEI. Let us first consider the superimposition internal operation, denoted as /&/. Let $\mathtt{X}_1 = (v_1 : \sigma_1)$ and $\mathtt{X}_2 = (v_2 : \sigma_2)$ be two data fields. If $\sigma_1|_{dom(\sigma_2)} = \sigma_2$ or $\sigma_2|_{dom(\sigma_1)} = \sigma_1$, the superimposition defines $\mathtt{X}_1/\&/\mathtt{X}_2$ as:

$$\begin{cases} (v_1 : \sigma) & \text{on } dom(v_1) \backslash dom(v_2) \\ (w : \sigma) & \text{on } dom(v_1) \cap dom(v_2) \\ (v_2 : \sigma) & \text{on } dom(v_2) \backslash dom(v_1) \end{cases}$$

where $w(z) = v_1(z); v_2(z)$ and $\sigma = \begin{cases} \sigma_1 \text{ if } dom(\sigma_2) \subseteq dom(\sigma_1) \\ \sigma_2 \text{ if } dom(\sigma_1) \subseteq dom(\sigma_2). \end{cases}$

The other operations in PEI are second-order functions. In the following, f, g, *etc.* are functions and X denotes a data field $(v : \sigma)$.

– **Functional operation.** Let f be a function. When applied on a X such that $img(v) \subseteq dom(\mathtt{f})$, the functional operation on f, denoted as $\mathtt{f} \triangleright$, defines the element-wise application of f on X. It means that $\mathtt{f} \triangleright \mathtt{X}$ is $(\mathtt{f} \circ v : \sigma)$.

– **Geometrical operation.** Let g be a function. When applied on a X such that $dom(\mathtt{g}) \subseteq dom(\sigma)$ and $img(\mathtt{g}) \subseteq dom(v)$, the geometrical operation on g, denoted as $\triangleleft\, \mathtt{g}$, defines the data field $\mathtt{X} \triangleleft \mathtt{g}$ as $(v \circ \mathtt{g} : \sigma)$.

Conversely, let g be a function. When applied on a X such that $dom(\mathtt{g}) \subseteq dom(v)$ and $img(\mathtt{g}) \subseteq dom(\sigma)$, the **geometrical reduction** on g, denoted as $\mathtt{g};\triangleright$, defines the data field $\mathtt{g}\ ;\triangleright\ \mathtt{X}$ as some data field $(w : \sigma)$ such that $dom(w) = \mathtt{g}(dom(v))$ and $w(z)$ is a sequence of the values $v(y)$, y such that $\mathtt{g}(y) = z$. As a consequence, the next property links reductions and geometrical operations:

Property 1 *The operation* $\mathtt{g};\triangleright$ *is equal to the operation* $\triangleleft\ \mathtt{g}^{-1}$ *if and only if* g *is bijective.*

– **Change of basis operation.** Let h be a bijection. When applied on a X such that $dom(v) \subseteq dom(\mathtt{h})$, the change of basis on h, denoted as h::, defines the data field $\mathtt{h} :: \mathtt{X}$ as $(v \circ \mathtt{h}^{-1} : \sigma \circ \mathtt{h}^{-1})$.

Remark – The operations are very similar to the classical Bird-Meertens functions on lists, such as *map* or *reduce*. They differ in that PEI operations include some geometrical aspects in order to meet the data parallel paradigm.

3.2 Equation semantics

Let E denote a data field expression equal to $(v : \sigma)$ and E' denote a data field expression equal to $(v' : \sigma')$. Equation $E = E'$ semantics, denoted as $[\![E = E']\!]$, is the boolean expression $(v = v') \wedge (\sigma = \sigma') \wedge C$, where C is the conjunction of all the conditions involved by the operations appearing in the equation. If the expression equals true, we will say that data fields appearing in the equation *verify* the equation.

For example, given data fields $\mathtt{A} = (v_{\mathtt{A}} : \sigma_{\mathtt{A}})$ and $\mathtt{B} = (v_{\mathtt{B}} : \sigma_{\mathtt{B}})$, the semantics of the equation $\mathtt{A} = \mathtt{add} \triangleright (\mathtt{size} :: \mathtt{B})$ is the boolean expression:

$$\begin{aligned} & v_{\mathtt{A}} = \mathtt{add} \circ v_{\mathtt{B}} \circ \mathtt{size}^{-1} \wedge \sigma_{\mathtt{A}} = \sigma_{\mathtt{B}} \circ \mathtt{size}^{-1} \\ \wedge\ & dom(v_{\mathtt{B}}) \subseteq dom(\mathtt{size}) \wedge img(v_{\mathtt{B}}) \subseteq dom(\mathtt{add}) \end{aligned}$$

The definition can be extended to an equation set, since data fields must verify all the equations. The semantics of an equation set is therefore the conjunction of the semantics of each equation.

3.3 Data field equivalence

As mentioned in section 2.1, two data fields may be intuitively considered as *equivalent* if they represent the same multiset of values. In this case, according to semantics we will say that these data fields are *weakly equivalent* since only their value domains are considered. Moreover if the data fields have the same reference mapping, we will say that they are *strongly equivalent.*

Definition 3 *Let* X *and* Y *be two data fields. We say that* X *and* Y *are* strongly equivalent *if and only if there exists a bijection* h *such that* $\mathtt{Y} = \mathtt{h} :: \mathtt{X}$ *and* $dom(\sigma_{\mathtt{X}}) \subseteq dom(\mathtt{h})$.

The change of basis operation gives an intuitive idea about strong equivalence of data fields, because it modifies the mapping of the same multiset of values but

maintains the same reference mapping. For example, consider h as the permutation λ(i)|(0<=i<4).((i-1) mod 4)) and the data field W = h :: V, V being the data field presented on figure 7. We observe on figure 8 that the value 2 in W is associated with the same point in the reference mapping as for V: $v_W(0) = v_V \circ h^{-1}(0) = v_V(1) = 2$ and $\sigma_W(0) = \sigma_V \circ h^{-1}(0) = \sigma_V(1) = (1,0)$.

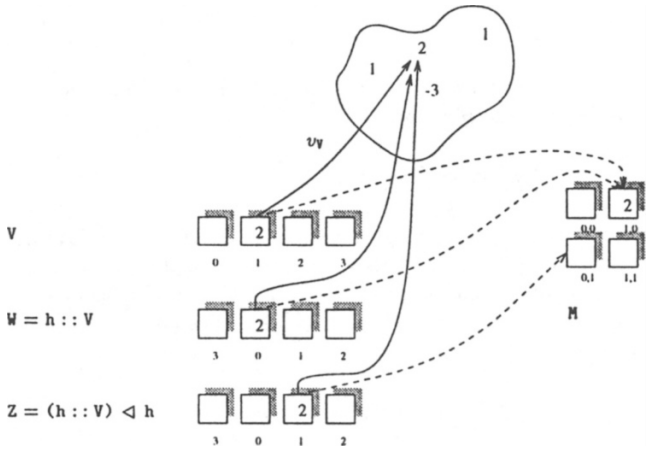

Figure 8 Weak and strong equivalence between data fields

Now, consider the definition Z = (h :: V) ◁ h in which Z and V are not strongly equivalent since $\sigma_Z \neq \sigma_V$ ($\sigma_Z = \sigma_V \circ h^{-1}$) though $v_Z = v_V$: they thus are weakly equivalent. By contrast to the previous case, the value 2 in Z ($v_Z(1) = v_V \circ h^{-1} \circ h\ (1) = v_V(1) = 2$) is associated with $\sigma_Z(1) = \sigma_V \circ h^{-1}(1) = \sigma_V(2) = (0,1)$.

Definition 4 *Let* X *and* Y *be two data fields. We say that* X *and* Y *are* weakly equivalent *if and only if they represent the same multiset of values.*

Weak equivalence of data fields can be characterized in PEI expressions, by the following property:

Property 2 *Let* X *be a data field and* Y *a data field expression depending on* X. X *and* Y *are* weakly equivalent *if and only if there exist three bijections* h, h' *and* g, *such that:*

- *either* (h :: X) ◁ g = h' :: Y *and* $h(dom(v_X)) = img(g)$,
- *or* h :: (X ◁ g) = h' :: Y *and* $dom(v_X) = img(g)$.

4 REFINEMENT AND ABSTRACTION

4.1 General definition

Refinement is a classical process for designing or transforming programs (Back 1988, Morgan 1990). It enables to build a program which satisfies any specification that an initial program satisfies. Refinement is usually defined in the following way, given a satisfaction property, denoted as sat :

Definition 5 *Let P and P' be two programs, we say that P is refined by P', and we note $P \sqsubseteq P'$, if and only if for any specification S*

$$P \,\mathsf{sat}\, S \Rightarrow P' \,\mathsf{sat}\, S$$

In imperative programming, the satisfaction property may be safety and termination for example. The *weakest pre-conditions* of Dijkstra, denoted $wp(P, post)$, where P is a statement and *post* a predicate can then be used: $wp(P, post)$ is the weakest predicate for which carrying out of P ends in a state which satisfies *post*. It allows to rewrite the previous definition as follows:

$$P \sqsubseteq P' \text{ if and only if } \forall r \cdot wp(P, r) \Rightarrow wp(P', r)$$

Refinement is a continuous process which allows successive statements to be built, from a specification until a program. It defines a partial order between statements, called *refinement relation*. Refinement is usually extended to specifications (Morgan 1990). Using pre- and post-specifications noted as [*pre*,*post*], two particular cases often arise:

$[p, q] \sqsubseteq [p', q]$ if $p \Rightarrow p'$ (pre-conditions weakening)
$[p, q] \sqsubseteq [p, q']$ if $q' \Rightarrow q$ (post-conditions strengthening)

Knapp (Knapp 1990) defines refinement in a different manner in order to apply it to UNITY programs (Chandy & Misra 1988). The definition is equivalent, but changes in its formulation because it adapts to specifications expressed in a modal logic. Let s and s' be two states, where $s(p)$ and $s'(q)$ mean that s and s' satisfies p and q respectively. Intuitively, a specification S has the following form:

$$S : \forall s(p) \cdot \exists s'(q) \cdot s \rightsquigarrow s' \qquad (1)$$

where the notation $s \rightsquigarrow s'$ means that s' is a reachable state from the initial state s. The definition of refinement becomes:

Definition 6 *Let S and S' be two specifications of the form* (1)*:*

$$S \sqsubseteq S' \text{ if and only if } S' \Rightarrow S$$

4.2 Refinement of PEI statements

Refinement relation.
The refinement in PEI is close to Knapp's definition, but reasoning is based on first order logic. Let us consider a statement $\mathtt{P}\{S\}$. The equations of S can be parsed into two exclusive types: the equations which only bind input data fields of P (they form a subset S_1 of S), and the equations which have intermediate or output data fields (these equations form the complementary subset S_2 of S). We call the equations of S_1 the *pre-equations* of P, and the equations of S_2 the *post-equations* of P.

According to the previous semantics definition we can state a property which is satisfied by a statement PEI. Let us call $sat(\mathtt{P})$ the property satisfied by statement P: it means that for all input data fields verifying S_1, the solution is a set of output data fields which verify S_2. Alternatively, we can write:

$$sat(\mathtt{P}) : [\![S_1]\!] \Rightarrow [\![S_2]\!]$$

For example, we associate with the following statement .

$$\mathtt{P1} : \mathtt{A} \mapsto \mathtt{B}$$
$$\begin{cases} \mathtt{A} = \mathtt{matrix} :: \mathtt{A} \\ \mathtt{B} = \mathtt{A} \lhd \mathtt{transp} \end{cases}$$
$$\mathtt{matrix} = \lambda(\mathtt{i},\mathtt{j})|(\mathtt{0<=i,j<n}).(\mathtt{i},\mathtt{j})$$
$$\mathtt{transp} = \lambda(\mathtt{i},\mathtt{j})|(\mathtt{0<=i,j<n}).(\mathtt{j},\mathtt{i})$$

the property $sat(\mathtt{P1}) : [\![(\mathtt{A} = \mathtt{matrix} :: \mathtt{A})]\!] \Rightarrow [\![(\mathtt{B} = \mathtt{A} \lhd \mathtt{transp})]\!]$
which means:

$$\begin{aligned} sat(\mathtt{P1}) : \quad & (v_{\mathtt{A}} = v_{\mathtt{A}} \circ \mathtt{matrix}^{-1}) \wedge (\sigma_{\mathtt{A}} = \sigma_{\mathtt{A}} \circ \mathtt{matrix}^{-1}) \\ & \wedge (dom(v_{\mathtt{A}}) \subseteq dom(\mathtt{matrix})) \\ \Rightarrow \quad & (v_{\mathtt{B}} = v_{\mathtt{A}} \circ \mathtt{transp}) \wedge (\sigma_{\mathtt{B}} = \sigma_{\mathtt{A}}) \\ & \wedge (dom(\mathtt{transp}) \subseteq dom(\sigma_{\mathtt{A}})) \wedge (img(\mathtt{transp}) \subseteq dom(v_{\mathtt{A}})) \end{aligned}$$

The constraints on domains arise from the operations definition : for example $\mathtt{h} :: \mathtt{X}$ is correctly defined only if $dom(v_{\mathtt{X}}) \subseteq dom(\mathtt{h})$.

Definition 7 *Let* P *and* P' *be two statements in* PEI. P *is said* refined by P' *—or* P abstracts P' *— denoted as* $\mathtt{P} \sqsubseteq \mathtt{P}'$, *if* $sat(\mathtt{P}') \Rightarrow sat(\mathtt{P})$.

Here is some classical examples of refinements in programming languages, such as the weakening of pre-conditions or the strengthening of post-conditions. In PEI, such transformations are expressed through the following property:

Property 3 *Let* $\mathtt{P}\{S_1, S_2\}$ *and* $\mathtt{P}'\{S_1', S_2'\}$ *be two statements, where* S_1, S_1' *are sets of pre-equations, and* S_2, S_2' *are sets of post-equations:*

$$\mathtt{P} \sqsubseteq \mathtt{P}' \quad \textit{if} \quad [\![S_2 = S_2']\!] \textit{ and } [\![S_1]\!] \Rightarrow [\![S_1']\!]$$
$$\textit{or} \quad [\![S_1 = S_1']\!] \textit{ and } [\![S_2']\!] \Rightarrow [\![S_2]\!]$$

Proof. Let us assume $[\![S_1]\!] \Rightarrow [\![S_1']\!]$, then $([\![S_1']\!] \Rightarrow [\![S_2]\!]) \Rightarrow ([\![S_1]\!] \Rightarrow [\![S_2]\!])$. Similarly, $[\![S_2']\!] \Rightarrow [\![S_2]\!]$ implies $([\![S_1]\!] \Rightarrow [\![S_2']\!]) \Rightarrow ([\![S_1]\!] \Rightarrow [\![S_2]\!])$. □

Example 3 Weakening pre-equations

P2 : X ↦ Y
{ X = vector : : X
{ Y = X

specified by *sat*(P2) :
[X = vector : : X] ⇒ [Y = X]

P3 : X ↦ Y
{ Y = X

specified by *sat*(P3) :
true ⇒ [Y = X]

Let us demonstrate *sat*(P3) ⇒ *sat*(P2): *sat*(P3) can be simplified to (Y = X) and the property [Y = X] ⇒ ([X = vector : : X]) ⇒ [Y = X]) is true.
◇

Example 4 Strengthening post-equations

P4 : X ↦ Y
{ vector : : Z = X
{ Y = vector : : Z

specified by *sat*(P4) : true ⇒ [vector : : Z = X] ∧ [Y = vector : : Z]

We have P3 ⊑ P4 since ([vector : : Z = X] ∧ [Y = vector : : Z]) ⇒ [Y = X].
◇

Refinement calculus.
In a more operational way, we define a *refinement calculus* allowing to build a refined statement from a given one. In PEI, the calculus consists in replacing one equation in the initial statement with another one. This can be done

- either by replacing *equal by equal* data fields expressions (cf. *Algebraic laws* hereunder),
- or by changing the reference domain of the equation (cf. *Change of reference domain*),
- or by substituting an equivalent data field for a given one. This requires the definition of a more general refinement relation including data fields equivalence (cf. *Refinement within the equivalence*).

Algebraic laws.
Some algebraic laws on operations can be proved. Assuming some hypotheses, they allow to replace a data field expression by another one in equations. In table 1, we note as $E \longrightarrow E'$ the rewriting of equation E in equation E' by applying such a law. The laws are the basis for the refinement calculus, as stated in the following property. We note as $S[E\backslash E']$ the equation set S in which an equation E has been rewritten as E', according to the laws.

(1a)	$\mathtt{Y = f1 \rhd (f2 :: X)} \longrightarrow \mathtt{Y = f2 :: (f1 \rhd X)}$
(1b)	$\mathtt{Y = f2 :: (f1 \rhd X)} \longrightarrow \mathtt{Y = f1 \rhd (f2 :: X)}$
(2a)	$\mathtt{Y = (f1 \rhd X) \lhd f2} \longrightarrow \mathtt{Y = f1 \rhd (X \lhd f2)}$
(2b)	$\mathtt{Y = f1 \rhd (X \lhd f2)} \longrightarrow \mathtt{Y = (f1 \rhd X) \lhd f2}$
(3a)	$\mathtt{Y = (f2 :: X) \lhd f1} \longrightarrow \mathtt{Y = f2 :: (X \lhd f2^{-1} \circ f1 \circ f2)}$ with $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq img(\mathtt{f2}) \wedge dom(v_X) \subseteq dom(\mathtt{f2})$
(3b)	$\mathtt{Y = f2 :: (X \lhd f1)} \longrightarrow \mathtt{Y = (f2 :: X) \lhd f2 \circ f1 \circ f2^{-1}}$ with $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq dom(\mathtt{f2})$
(3c)	$\mathtt{Y = f1 \;;\rhd (f2 :: X)} \longrightarrow \mathtt{Y = f2 :: (f2^{-1} \circ f1 \circ f2 \;;\rhd X)}$ with $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq img(\mathtt{f2}) \wedge dom(v_X) \subseteq dom(\mathtt{f2})$
(3d)	$\mathtt{Y = f2 :: (f1 \;;\rhd X)} \longrightarrow \mathtt{Y = f2 \circ f1 \circ f2^{-1} \;;\rhd (f2 :: X)}$ with $dom(\mathtt{f1}) \cup img(\mathtt{f1}) \subseteq dom(\mathtt{f2})$
(4a)	$\mathtt{Y = (f1 \circ f2) \rhd X} \longrightarrow \mathtt{Y = f1 \rhd (f2 \rhd X)}$
(4b)	$\mathtt{Y = f1 \rhd (f2 \rhd X)} \longrightarrow \mathtt{Y = (f1 \circ f2) \rhd X}$
(5a)	$\mathtt{Y = X \lhd (f1 \circ f2)} \longrightarrow \mathtt{Y = (X \lhd f1) \lhd f2}$
(5b)	$\mathtt{Y = (X \lhd f1) \lhd f2} \longrightarrow \mathtt{Y = X \lhd (f1 \circ f2)}$ with $img(\mathtt{f2}) \subseteq dom(\mathtt{f1}) \subseteq dom(\sigma_X)$
(6)	$\mathtt{Y = (f1 \circ f2) \;;\rhd X} \longrightarrow \mathtt{Y = f1 \;;\rhd (f2 \;;\rhd X)}$
(7a)	$\mathtt{Y = (f1 \circ f2) :: X} \longrightarrow \mathtt{Y = f1 :: (f2 :: X)}$
(7b)	$\mathtt{Y = f1 :: (f2 :: X)} \longrightarrow \mathtt{Y = (f1 \circ f2) :: X}$ with f1 and f2 two bijections
(8a)	$\mathtt{Y = (X1 \;/\&/\; X2) \lhd f} \longrightarrow \mathtt{Y = (X1 \lhd f) \;/\&/\; (X2 \lhd f)}$
(8b)	$\mathtt{Y = (X1 \lhd f) \;/\&/\; (X2 \lhd f)} \longrightarrow \mathtt{Y = (X1 \;/\&/\; X2) \lhd f}$ with $dom(\mathtt{f}) \subseteq dom(\sigma_{X1}) \cap dom(\sigma_{X2}) \wedge img(\mathtt{f}) \subseteq dom(v_{X1}) \cap dom(v_{X2})$
(9a)	$\mathtt{Y = X \lhd (f1 \,\#\, f2)} \longrightarrow \mathtt{Y = (X \lhd f1) \;/\&/\; (X \lhd f2)}$
(9b)	$\mathtt{Y = (X \lhd f1) \;/\&/\; (X \lhd f2)} \longrightarrow \mathtt{Y = X \lhd (f1 \,\#\, f2)}$ with $dom(\mathtt{f1}) \cap dom(\mathtt{f2}) = \emptyset$
(10)	$\mathtt{Y = (f1 \,\#\, f2) \;;\rhd X} \longrightarrow \mathtt{Y = (f1 \;;\rhd X) \;/\&/\; (f2 \;;\rhd X)}$
(11a)	$\mathtt{Y = (f :: X1) \;/\&/\; (f :: X2)} \longrightarrow \mathtt{Y = f :: (X1 \;/\&/\; X2)}$
(11b)	$\mathtt{Y = f :: (X1 \;/\&/\; X2)} \longrightarrow \mathtt{Y = (f :: X1) \;/\&/\; (f :: X2)}$

Table 1 Rewriting of equations

Property 4 *Let E and E' be two equations in a* PEI *statement* $\mathtt{P}\{S\}$*:*
- *if* $E \longrightarrow E'$ *and* E, E' *are pre-equations, then* $\mathtt{P}\{S[E\backslash E']\} \sqsubseteq \mathtt{P}$
- *if* $E \longrightarrow E'$ *and* E, E' *are post-equations, then* $\mathtt{P} \sqsubseteq \mathtt{P}\{S[E\backslash E']\}$

Proof. We only report the proof for two rules in this paper, assuming they apply in a post-equation.

- Let us consider rule (5a) and prove $p' \Rightarrow p$, where:

$$\begin{aligned} p \;=\;& (v_Y = v_X \circ \mathtt{f1} \circ \mathtt{f2}) \wedge (\sigma_Y = \sigma_X) \\ & \wedge (dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma_X)) \wedge (img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v_X)) \\ p' \;=\;& (v_Y = v_X \circ \mathtt{f1} \circ \mathtt{f2}) \wedge (\sigma_Y = \sigma_X) \\ & \wedge (dom(\mathtt{f1}) \subseteq dom(\sigma_X)) \wedge (dom(\mathtt{f2}) \subseteq dom(\sigma_X)) \\ & \wedge (img(\mathtt{f1}) \subseteq dom(v_X)) \wedge (img(\mathtt{f2}) \subseteq dom(v_X \circ \mathtt{f1})) \end{aligned}$$

  The proposition reduces to:

$$(2) \begin{cases} v_Y = v_X \circ \mathtt{f1} \circ \mathtt{f2} \\ \sigma_Y = \sigma_X \\ img(\mathtt{f1}) \subseteq dom(v_X) \\ img(\mathtt{f2}) \subseteq dom(v_X \circ \mathtt{f1}) \end{cases} \Rightarrow (1) \begin{cases} img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v_X) \\ dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma_X) \end{cases}$$

  Let us assume (2) holds. Since $dom(v_X \circ \mathtt{f1}) \subseteq dom(\mathtt{f1})$, we have $img(\mathtt{f2}) \subseteq dom(\mathtt{f1})$ and from function composition, $img(\mathtt{f1} \circ \mathtt{f2}) = img(\mathtt{f1})$. So $img(\mathtt{f1}) \subseteq dom(v_X)$ can be rewritten $img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v_X)$. Moreover, it implies $dom(\mathtt{f1} \circ \mathtt{f2}) = dom(v_X \circ \mathtt{f1} \circ \mathtt{f2})$ which equals $dom(v_Y)$. From data field definition we have $dom(v_Y) \subseteq dom(\sigma_Y)$ and, since $\sigma_Y = \sigma_X$, $dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma_X)$.

- Concerning rule (7a), let us prove $p' \Rightarrow p$, where:

$$\begin{aligned} p \;=\;& (v_Y = v_X \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge (\sigma_Y = \sigma_X \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \\ & \wedge (dom(v_X) \subseteq dom(\mathtt{f1} \circ \mathtt{f2})) \\ p' \;=\;& (v_Y = v_X \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \wedge (\sigma_Y = \sigma_X \circ \mathtt{f2}^{-1} \circ \mathtt{f1}^{-1}) \\ & \wedge (dom(v_X) \subseteq dom(\mathtt{f2})) \wedge (dom(v_X \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1})) \end{aligned}$$

  The proposition reduces to:

$$(2) \begin{cases} dom(v_X) \subseteq dom(\mathtt{f2}) \\ dom(v_X \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1}) \end{cases} \Rightarrow (1) \; \{dom(v_X) \subseteq dom(\mathtt{f1} \circ \mathtt{f2})$$

  Let us assume (2) holds. Since $dom(v_X) \subseteq dom(\mathtt{f2})$ we have:

$$\begin{aligned} dom(v_X \circ \mathtt{f2}^{-1}) \subseteq dom(\mathtt{f1}) &\Rightarrow \mathtt{f2}(dom(v_X)) \subseteq dom(\mathtt{f1}) \\ &\Rightarrow \mathtt{f2}^{-1}(\mathtt{f2}(dom(v_X))) \subseteq \mathtt{f2}^{-1}(dom(\mathtt{f1})) \\ &\Rightarrow dom(v_X) \subseteq dom(\mathtt{f1} \circ \mathtt{f2}) \end{aligned}$$

□

Change of reference domain.
An other way to refine a PEI statement is to change the reference domain of an equation by applying a change of basis on both sides. This fact is expressed by the following rules on which property 4 also applies:

$$\mathtt{Y} = \mathtt{X} \longrightarrow \mathtt{h} :: \mathtt{Y} = \mathtt{h} :: \mathtt{X} \quad \text{with } dom(\sigma_X) \subseteq dom(\mathtt{h}) \wedge dom(\sigma_Y) \subseteq dom(\mathtt{h})$$

$$\mathtt{h::Y = h::X} \longrightarrow \mathtt{Y = X} \quad \text{with } dom(v_X) \subseteq dom(\mathtt{h})$$

Proof. Let us prove the first rule. Let p be the property deduced from the equation Y = X (the left part of the rule) and let p' be the one deduced from the right part. We have:

$$\begin{aligned} p &= (v_Y = v_X) \wedge (\sigma_Y = \sigma_X) \\ p' &= (v_Y \circ \mathtt{h}^{-1} = v_X \circ \mathtt{h}^{-1}) \wedge (\sigma_Y \circ \mathtt{h}^{-1} = \sigma_X \circ \mathtt{h}^{-1}) \\ &\quad \wedge (dom(v_X) \subseteq dom(\mathtt{h})) \wedge (dom(v_Y) \subseteq dom(\mathtt{h})) \\ &\quad \wedge (dom(\sigma_X) \subseteq dom(\mathtt{h})) \wedge (dom(\sigma_Y) \subseteq dom(\mathtt{h})) \end{aligned}$$

Let us show that $p' \Rightarrow p$: from $v_Y \circ \mathtt{h}^{-1} = v_X \circ \mathtt{h}^{-1}$ and $\sigma_Y \circ \mathtt{h}^{-1} = \sigma_X \circ \mathtt{h}^{-1}$, we deduce $v_X|_{dom(\mathtt{h})} = v_Y|_{dom(\mathtt{h})}$ and $\sigma_X|_{dom(\mathtt{h})} = \sigma_Y|_{dom(\mathtt{h})}$ respectively, and since $dom(v_X)$, $dom(v_Y)$, $dom(\sigma_X)$ and $dom(\sigma_Y)$ are subsets of $dom(\mathtt{h})$, it follows $v_X = v_Y$ and $\sigma_X = \sigma_Y$.

The proof of the second rule is quite similar. □

Refinement within the equivalence.

A more general way to refine a PEI statement consists in substituting an equivalent data field for a given one, according to the weak or strong equivalences which were introduced in section 3.3. This can be done by generalizing the previous definition of refinement (definition 7), in order to take equivalence into account.

In the following, we note $S[\mathtt{X}\backslash\mathtt{X}']$ the equation set S in which one occurrence X has been replaced with another expression X′ of the same data field, and $S[\mathtt{X}\backslash\mathtt{X}']^*$ when all occurrences of X have been replaced by X′.

Definition 8 *Let* P *and* P' *be two statements, and* S *be the equation set of* P. *We say that* $sat(\mathtt{P}')$ implies $sat(\mathtt{P})$ within data field equivalence, *denoted as* $sat(\mathtt{P}') \overset{\equiv}{\Rightarrow} sat(\mathtt{P})$ *if and only if* $sat(\mathtt{P}') \Rightarrow sat(\widehat{\mathtt{P}})$ *with:*

- *either* $\widehat{\mathtt{P}} = \mathtt{P}\{S[\mathtt{X}\backslash e(\mathtt{X})]^*\}$, *where* $e(\mathtt{X})$ *is an expression depending on* X *and satisfies: for any data field denoted as this* X, *there exists a strongly equivalent data field, say* X′, *such that* $e(\mathtt{X}')$ *and* X *denote the same data field,*
- *or* $\widehat{\mathtt{P}} = \mathtt{P}\{S[\mathtt{E}\backslash e(\mathtt{E})]\}$, *where* $e(\mathtt{E})$ *is an expression depending on* E *and satisfies: for any data field denoted as this expression* E, *there exists a weakly equivalent data field expression, say* E′, *such that* $e(\mathtt{E}')$ *and* E *denote the same data field.*

It allows to generalize the previous definition 7 of refinement in PEI as follows:

Definition 9 *Let* P *and* P' *be two statements.* P *is said* refined *by* P' *if and only if* $sat(\mathtt{P}') \overset{\equiv}{\Rightarrow} sat(\mathtt{P})$.

Example 5 Refinement within the equivalence

Let us consider the two programs P5 and P6, and show that P5 $\sqsubseteq$ P6 using data fields equivalence.

$$\begin{array}{ll} \texttt{P5} : \texttt{X} \mapsto \texttt{Y} & \texttt{P6} : \texttt{X} \mapsto \texttt{Y} \\ \left\{ \begin{array}{l} \texttt{X} = \texttt{square}::\texttt{X} \\ \texttt{Y} = \texttt{X} \end{array} \right. & \left\{ \begin{array}{l} \texttt{X} = \texttt{square}::\texttt{X} \\ \texttt{Z} = \texttt{X} \\ \texttt{transp}::\texttt{Y} = \texttt{Z} \end{array} \right. \end{array}$$

$$\begin{array}{l} \texttt{square} = \lambda(\texttt{i},\texttt{j})|(0<=\texttt{i},\texttt{j}<\texttt{n}).(\texttt{i},\texttt{j}) \\ \texttt{transp} = \lambda(\texttt{i},\texttt{j})|(0<=\texttt{i},\texttt{j}<\texttt{n}).(\texttt{j},\texttt{i}) \end{array}$$

Let us consider $\widehat{\texttt{P5}}$ obtained from P5 by replacing Y with (transp::Y') and then by renaming Y' as Y (since Y and Y' are strongly equivalent). The properties we consider are then:

$sat(\widehat{\texttt{P5}})$: $[\texttt{X} = \texttt{square}::\texttt{X}] \Rightarrow [\texttt{transp}::\texttt{Y} = \texttt{X}]$
$sat(\texttt{P6})$: $[\texttt{X} = \texttt{square}::\texttt{X}] \Rightarrow ([\texttt{Z} = \texttt{X}] \wedge [\texttt{transp}::\texttt{Y} = \texttt{Z}])$

It is straightforward that $sat(\texttt{P6}) \Rightarrow sat(\widehat{\texttt{P5}})$ because the equality on data fields is transitive. Then $sat(\texttt{P6}) \overset{\equiv}{\Rightarrow} sat(\texttt{P5})$ and P5 $\sqsubseteq$ P6.
$\diamond$

5 CASE STUDY: *GAXPY*

The example studied in this section is a variant of the *Gaxpy* operation (Golub & Loan 1989), which is merely a computation of the form: $c = Ax + y$, $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. We only consider in this article the particular case where $y = x$, which operation is often iterated to compute $A(\ldots(A(Ax+x)+x)\ldots)+x$, for example in the conjugate gradient method.

Due to its extensive usage in numerical computations, many implementations of the operation have been proposed for parallel computers. We show two data parallel versions of the algorithm: the first version takes into account hardware restrictions that are sometimes encountered (for instance, diagonal values cannot be broadcasted along a dimension on *CM-200* and *CM-5* computers (Petiton & Emad 1996)), whereas the second version does not consider this restriction. Thus the two algorithms differ in their data placement, and we show that one algorithm can be deduced from the other.

The initial algorithm is described on figure 9(a) and is transcribed by the program Gaxpx1: it consists in aligning x (represented by data field X) on the first row of the matrix (B0). Then, x is replicated on all the matrix rows to form the data field B. Such an alignment enables to compute all the products at one time (P). All P values have then to be added row-wise to obtain Ax, the result being arbitrarily placed on last

column (C0). The addition of Ax with x requires C0 to be transposed (C1); eventually the result $Ax + x$ lies on first row of the matrix (C).

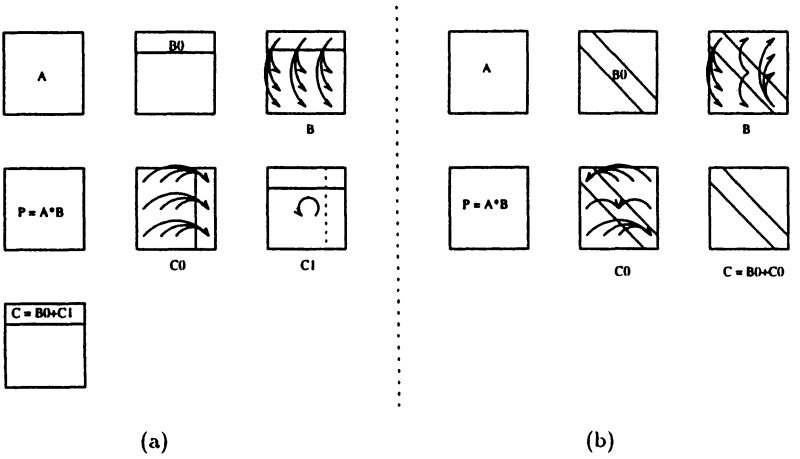

Figure 9 (a) Initial algorithm (b) Improved algorithm

```
Gaxpx1 : (A,X) ↦ C
{ A = matrix::A
  X = align::B0
  B = B0 ◁ spread
  P = prod ▷ (A /&/ B)
  C0 = sum ▷ (red ;▷ P)
  C1 = C0 ◁ col2row
  C = add ▷ (B0 /&/ C1)

matrix   =  λ(i,j)|(0≤i,j<n).(i,j)
align    =  λ(i,j)|(i=0&0≤j<n).(j)
spread   =  λ(i,j)|(0≤i,j<n).(0,j)
red      =  λ(i,j)|(0≤i,j<n).(i,n-1)
col2row  =  λ(i,j)|(i=0&0≤j<n).(j,n-1)
add      =  λ(a;b).(a+b)
prod     =  λ(a;b).(a*b)
sum      =  id # λ(a;b).(a+sum b)
```

If the hardware requirements support broadcast of diagonal values, then we can design a more efficient program, as sketched on figure 9(b). Hence, the alignments of x and Ax on the diagonal makes the transposition phase useless. On parallel computers, such an operation involves a costly interprocessor communication so that removing this phase results in an important performance gain.

We show that the new program can be obtained from **Gaxpx1** using the refinement calculus. The idea is to realign B0 and C0 by mapping them onto the diagonal.

5.1 Vector realignments

First, we change the definition of B0 relatively to X (we change X alignment) by applying the following change of basis (equal to its inverse), that swaps diagonal and first row points:

$$\begin{aligned}\texttt{row2diag} = \quad & \lambda(i,j)|(i{=}0 \,\&\, 0{\le}j{<}n).(j,j) \;\#\\ & \lambda(i,j)|(i{\neq}0 \,\&\, i{=}j \,\&\, 0{\le}j{<}n).(0,j) \;\#\\ & \lambda(i,j)|(i{\neq}0 \,\&\, i{\neq}j \,\&\, 0{\le}i,j{<}n).(i,j)\end{aligned}$$

From definitions 8 and 9, a refined program is obtained by replacing all occurrences of B0 by (row2diag :: B0):

$$\texttt{Gaxpx1'} : (\texttt{A},\texttt{X}) \mapsto \texttt{C}$$
$$\left\{\begin{array}{ll}\texttt{A} = \texttt{matrix} :: \texttt{A} & \\ \texttt{X} = \texttt{align} :: (\texttt{row2diag} :: \texttt{B0}) & (i)\\ \texttt{B} = (\texttt{row2diag} :: \texttt{B0}) \lhd \texttt{spread} & (ii)\\ \texttt{P} = \texttt{prod} \rhd (\texttt{A} \;/\&/\; \texttt{B}) & \\ \texttt{C0} = \texttt{sum} \rhd (\texttt{red} \;;\rhd\; \texttt{P}) & \\ \texttt{C1} = \texttt{C0} \lhd \texttt{col2row} & \\ \texttt{C} = \texttt{add} \rhd (\texttt{row2diag} :: \texttt{B0} \;/\&/\; \texttt{C1}) & \end{array}\right.$$

We obtain a new alignment from the previous statement, by applying some algebraic rules:

$$\begin{aligned}(i) \longrightarrow \quad & \text{"}\textit{rewriting rule (7b)}\text{"}\\ & \texttt{X} = \texttt{align} \circ \texttt{row2diag} :: \texttt{B0}\\ \longrightarrow \quad & \text{"}\textit{function renaming}\text{"},\\ & \text{let } \texttt{align2} = \texttt{align} \circ \texttt{row2diag} = \lambda(\texttt{i},\texttt{j})|(\texttt{i}{=}\texttt{j} \,\&\, 0{\le}\texttt{j}{<}\texttt{n}).(\texttt{j})\\ & \texttt{X} = \texttt{align2} :: \texttt{B0}\end{aligned}$$

Intuitively, the data field B has the same values in both versions, but their origins are different; it is a typical case of weak equivalence. From definition 8 and 9, (*ii*) may be replaced by

$$\texttt{B} = (\texttt{row2diag} :: ((\texttt{row2diag} :: \texttt{B0}) \lhd \texttt{spread})) \lhd \texttt{row2diag}$$

The expression can be successively transformed into the following equations where i denotes the identity on $\mathbb{Z}^2$:

```
      B = (row2diag :: ((row2diag :: B0) ◁ spread)) ◁ row2diag
```

⟶ "*rewriting rule (3a)*",
as *img*(spread) ⊆ *img*(row2diag)
and *dom*(v_{B0}) ⊆ *dom*(row2diag) from (*i*),
and "*function renaming*",

```
let spread2    = row2diag ∘ spread ∘ row2diag
               = λ(i,j)|(0≤i,j<n).(j,j)
B = (row2diag :: (row2diag :: (B0 ◁ spread2))) ◁ row2diag
```

⟶ "*rewriting rule (7b)*"

```
B = (i|dom(row2diag) :: (B0 ◁ spread2)) ◁ row2diag
```

⟶ "*simplification*", since *dom*(spread2) ⊆ *dom*(row2diag)

```
B = (B0 ◁ spread2) ◁ row2diag
```

⟶ "*rewriting rule (5b)*", since *dom*(spread2) ⊆ *dom*(σ_{B0})

```
B = B0 ◁ spread2 ∘ row2diag
```

⟶ "*definition of* spread2", spread2 = spread2 ∘ row2diag

```
B = B0 ◁ spread2
```

The following refined program summarizes the previous refinements:

```
Gaxpx2 : (A,X) ↦ C
⎧ A = matrix :: A
⎪ X = align2 :: B0
⎪ B = B0 ◁ spread2
⎨ P = prod ▷ (A /&/ B)
⎪ C0 = sum ▷ (red ;▷ P)
⎪ C1 = C0 ◁ col2row
⎩ C = add ▷ (row2diag :: B0 /&/ C1)
```

Similarly, we change the alignment of Ax (C0) using the change of basis:

```
col2diag =  λ(i,j)|(j=n-1&0≤i<n).(i,i) #
            λ(i,j)|(j≠n-1&i=j&0≤i<n).(i,n-1) #
            λ(i,j)|(j≠n-1&i≠j&0≤i,j<n).(i,j)
```

From definitions 8 and 9, we can refine Gaxpx2 in:

```
Gaxpx2' : (A,X) ↦ C
⎧ A = matrix :: A
⎪ X = align2 :: B0
⎪ B = B0 ◁ spread2
⎪ P = prod ▷ (A /&/ B)
⎨ col2diag :: C0 = sum ▷ (red ;▷ P)    (iii)
⎪ C1 = (col2diag :: C0) ◁ col2row       (iv)
⎩ C = add ▷ (row2diag :: B0 /&/ C1)
```

As shown on figure 9(b), P values are gathered on the diagonal to form C0. The data field P and (col2diag :: P) ◁ col2diag are weakly equivalent. We obtain a new reduction by applying some rewriting rules on the following expression:

col2diag :: C0 = sum ▷ (red ;▷ ((col2diag :: P) ◁ col2diag))

⟶ "*rewriting rule (3a)*", since $dom(v_P) \subseteq dom(\text{cold2diag})$
col2diag :: C0 = sum ▷ (red ;▷ (col2diag :: (P ◁ col2diag)))

⟶ "*rewriting rule (3c) and function renaming*",
let red2 = col2diag ∘ red ∘ col2diag = $\lambda(i,j)|(0 \leq i,j < n).(i,i)$
col2diag :: C0 = sum ▷ (col2diag :: (red2 ;▷ (P ◁ col2diag)))

⟶ "*rewriting rule (1a)*"
col2diag :: C0 = col2diag :: (sum ▷ (red2 ;▷ (P ◁ col2diag)))

⟶ "*change of reference domain*"
as $dom(v_{C0}) \subseteq dom(\text{col2diag})$ from equation (*iv*)
C0 = sum ▷ (red2 ;▷ (P ◁ col2diag))

The following steps allows the routing col2diag to be removed in the last expression:

⟶ "*definition of* red2", red2 = red2 ∘ col2diag
C0 = sum ▷ (red2 ∘ col2diag ;▷ (P ◁ col2diag))

⟶ "*rewriting rule (6) and property 1*", col2diag equals to its inverse
C0 = sum ▷ (red2 ;▷ ((P ◁ col2diag) ◁ col2diag))

⟶ "*rewriting rule (5b)*", since $dom(\text{col2diag}) \subseteq dom(\sigma_P)$
C0 = sum ▷ (red2 ;▷ (P ◁ $i|_{dom(\text{col2diag})}$))

⟶ "*simplification*", since $dom(\text{col2diag}) \subseteq dom(v_P)$
C0 = sum ▷ (red2 ;▷ P)

We thus have:

```
Gaxpx3 : (A,X) ↦ C
⎧ A = matrix :: A
⎪ X = align2 :: B0
⎪ B = B0 ◁ spread2
⎨ P = prod ▷ (A /&/ B)
⎪ C0 = sum ▷ (red2 ;▷ P)
⎪ C1 = (col2diag :: C0) ◁ col2row      (vi)
⎩ C = add ▷ (row2diag :: B0 /&/ C1)
```

5.2 Transposition removing

Intuitively, C0 and B0 are both aligned on the diagonal, making the transposition defined by C1 useless. As B0 is aligned using row2diag, we have to introduce row2diag in the expression of C0. The idea is to decompose col2diag using row2diag. Since col2diag realigns B0 onto the last column, we have to complete row2diag by a change

of basis that swaps the last column and the first row. Let `row2col` be this change of basis:

```
row2col =  λ(i,j)|(j=n-1&0≤i<n).(0,i) #
           λ(i,j)|(j≠n-1&i=0&0≤j<n).(j,n-1) #
           λ(i,j)|(j≠n-1&i≠0&0≤i,j<n).(i,j)
```

Since `col2diag` and `row2col` ∘ `row2diag` are equal on the value domain of `C0`, the data fields `col2diag` : : `C0` and `row2col` ∘ `row2diag` : : `C0` are weakly equivalent. Moreover, `row2col` ∘ `row2diag` : : `C0` can be rewritten as `row2col` : : (`row2diag` : : `C0`) from rewriting rule (7a), and `col2row` is equal to `row2col` on the first row (*i.e.* the value domain of `row2diag` : : `C0`). Therefore we can conclude that data fields `C1` and `row2diag` : : `C0` are weakly equivalent. This allows the new statement `Gaxpx3` to be obtained by substituting `row2diag` : : `C0` for `C1` definition, using weak equivalence implication (definition 8):

```
Gaxpx3' : (A,X) ↦ C
⎧ A = matrix : : A
⎪ X = align2 : : B0
⎪ B = B0 ◁ spread2
⎪ P = prod ▷ (A /&/ B)
⎪ C0 = sum ▷ (red2 ;▷ P)
⎩ C = add ▷ (row2diag : : B0 /&/ row2diag : : C0)
```

We obtain the final statement from the previous one by applying the change of basis defined by `row2diag` on `C` and by using rules (11a), (1a) and the change of reference domain in the last equation.

```
Gaxpx4 : (A,X) ↦ C
⎧ A = matrix : : A
⎪ X = align2 : : B0
⎪ B = B0 ◁ spread2
⎪ P = prod ▷ (A /&/ B)
⎪ C0 = sum ▷ (red2 ;▷ P)
⎩ C = add ▷ (B0 /&/ C0)
```

6 CONCLUSION

We have presented in this paper a formalism designed for the construction and derivation of parallel programs. In particular, it has shown to be well suited to data parallel programs specifications, and the case study in last section showed a specific but nonetheless crucial kind of transformation for that model, which mainly concerns data alignments. In addition to the PEI theory, some tools have been developed to ease the use of PEI:

- a type-checker that infers the domains of the data fields in a statement, and therefore controls the data fields consistency. Furthermore, the inference of domain information allows conditions required by refinement rules to be easily checked,
- the refinement calculus can be done through a dedicated tool. It browses a statement, in which any data field expression may be selected and refined, the appropriate rewriting rule being automatically applied,
- some other prototype tools, such as a PEI-CAML and PEI-HPF compiler, should also enlarge the potential use of PEI.

REFERENCES

Back, R. J. (1988), 'A calculus of refinements for program derivations', *Acta Informatica* **25**, 593–624.

Bird, R. S. (1987), Introduction to the theory of lists, *in* M. Broy, ed., 'Logic of Programming and Calculi of Discrete Design', Springer-Verlag, pp. 3–42.

Chandy, K. M. & Misra, J. (1988), *Parallel Program Design : A foundation*, Addison Wesley.

Chen, M., il Choo, Y. & Li, J. (1991), *Parallel Functional Languages and Compilers*, ACM Press.

Genaud, S., Violard, E. & Perrin, G.-R. (1995), Transformations techniques in PEI, *in* P. Magnusson, S. Haridi & A. Khayri, eds, 'LNCS', EURO-PAR95 Parallel Processing, Springer-Verlag, Stockholm, Sweden, pp. 131–142.

Golub, G. H. & Loan, C. F. V. (1989), *Matrix Computations*, 2nd edn, The John Hopkins University Press.

HPF (1993), *High Performance Fortran Language Specification*, 1.0 edn.

Knapp, E. (1990), 'An exercise in the formal derivation of parallel programs: Maximum flows in graphs', *ACM Transactions on Programming Languages and Systems* **12**, 203–223.

Mauras, C. (1989), ALPHA : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones, PhD thesis, Université de Rennes I.

Morgan, C. (1990), *Programming from specifications*, C.A.R. Hoare, Prentice Hall Ed., Endlewood Cliffs, N.J.

Petiton, S. & Emad, N. (1996), *A data parallel scientific computation introduction*, Vol. LNCS 1132, Springer Verlag, pp. 45–62.

Skillicorn, D. B. (1993), The Bird-Meertens formalism as a parallel model, *in* J. Kowalik & L. Grandinetti, eds, 'NATO ARW "Software for Parallel Computation"', Springer-Verlag.

Thi (1990), *C* Programming Guide*.

Violard, E. & Perrin, G.-R. (1992), 'PEI : a language and its refinement calculus for parallel programming', *Parallel Computing* **18**, 1167–1184.

Violard, E. & Perrin, G.-R. (1993), PEI : a single unifying model to design parallel programs, *in* A. Bode, M. Reeve & G. Wolf, eds, 'PARLE 93, LNCS', Springer-Verlag, pp. 500–516.

7 BIOGRAPHIES

ERIC VIOLARD has been an assistant professor at the University of Besancon (France) since 1993 where he submitted a doctorate in 1992. He is currently working at the ICPS laboratory in Strasbourg. His research work is concerned with design of parallel programs by transformation techniques. His current works concern the foundations of PEI.

STÉPHANE GENAUD received a PhD in computer science from Louis Pasteur University of Strasbourg (France) in 1997. He is currently a lecturer at IUT d'informatique de Strasbourg. His research interests are in languages and methods for parallel programming.

GUY-RENÉ PERRIN received his PhD in 1976 and his higher doctorate in 1985. Since this date he has been a professor in Computer Science at the University of Besançon (France) where he was head of the Computer Science Department and of the Postgraduate Research Courses. Since 1994 he is a professor at the University Louis Pasteur at Strasbourg, where he is head of the ICPS research team. He is currently engaged in research on automatic parallelization techniques and data parallel programming.