Transformation techniques in Pei

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

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

Abstract. This article presents a few examples of program transformation strategies in the language Pei [Vio94]. Three strategies are developed : a simplification of the communications, the introduction of broadcasts by removing recursion from data field definitions, and the introduction of a reduction operator. These transformations emphasize the relationships between several programs solving a given problem, especially in the data parallelism area.

Keywords. Abstraction, Broadcast, Communication, Data-parallelism, Reduction, Refinement.

1 Introduction

Among the various parallel programming models that have recently emerged, the *data-parallel* paradigm is of particular interest and a few customized languages have been defined in this area, extending classical expressions such as C or Fortran. Hence, manipulation of regular data and control structures in the framework of a synchronous programming model only requires a small amount of effort to adapt from sequential programming to data-parallelism.

Nevertheless, studies have still to be carried out about this parallel programming model, especially in semantics, proof, program design methodologies or program transformations. This requires the definition of a theoretical foundation for data-parallel programming. Besides a few proposals (see for example [BL92]), Pei was defined [Vio94, VP94] in order to address the crucial question of program refinement through a straightforward generalization of equational notations ([Mau89, CCL91]), which were previously designed to implement nested loop parallelization methods in the so-called *Polytope model* [Len93].

Pei provides a formal framework to describe problem specifications and a means to reason on programs. Its semantics lies on a small but powerful set of mathematical issues since Pei statements describe operations on multisets of values. In order to access to these values, they are mapped onto discrete domains and form *data fields*, that are the basic objects of the language. 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, or express data dependencies, or redraw the discrete domains underlying data fields. These operations are bricks for a refinement calculus allowing program derivation or transformation. Programs themselves are sets of equations between data fields. Moreover, reduction in PEI is defined as the *inverse* of some operations and the refinements it involves can lead to different reduction or scan functions.

This article aims to emphasize the way transformations can apply to refine, or conversely abstract parallel programs. Some fundamental rules are presented and lead to the definition of transformation strategies. Starting from an initial statement, several programs can be deduced : they can be considered as different implementations of an original function, according to different execution models and to the control and communication primitives they involve. Section 2 recalls the main features of PEI. Section 3 presents the concept of program refinement which is the foundation to transform the programs. Three transformation strategies are then developed in section 4 : a simplification of the communications, the introduction of broadcasts and then of a reduction operator by removing recursion from data field definitions.

2 Definition of Pei

2.1 Specifications and programs

PEI specifies a problem as a relation between *multisets of values*, called its *inputs* and *outputs*. Values of a multiset are supposed to be mapped onto discrete domains and form a *data field*. A specification is expressed as a system of unoriented equations on data fields [1]. It is called a *program* if these equations define a function, i.e. the system has at most one solution.

Example 1. Summation of two matrices

$$\mathtt{MATSUM:(A,B)} \mapsto \mathtt{C}$$
$$\begin{cases} \mathtt{A} = \mathtt{matrix\ ::\ A} \\ \mathtt{B} = \mathtt{matrix\ ::\ B} \\ \mathtt{C} = \mathtt{add} \rhd \mathtt{(A\ /\&/\ B)} \end{cases}$$
$$\mathtt{matrix} = \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n)}$$
$$\mathtt{add} \quad = \lambda\mathtt{(a\cdot b).(a+b)}$$

– the two first equations define the input data fields `A` and `B`. They express that `A` and `B` are invariant by applying a so-called *change of basis* operation (notation `::`). This change of basis is the identity on the square domain `(0<=i,j<n)`[2],

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

[2] Without loss of generality, we assume that any vectorial data structure (vector, matrix, etc.) is expressed by a data field defined on the same geometry. So, the value A[i,j] is mapped on the index `(i,j)` of the data field `A` in $\mathbb{Z}^2$.

- the third equation defines the output data field `C`. The right hand side defines the *superimposition* of `A` and `B` (notation `/&/`), i.e. the set of couples `(a·b)`, with `a` in `A` and `b` in `B`. The *function* `add` is applied on every element of the resulting data field (notation $\triangleright$).

2.2 Objects and expressions

Basic objects in PEI are data fields and functions. We only need here an intuitive idea about what is a data field : as we said before, it is a set of values mapped onto some discrete domain. In the following `A`, `B`, `X`, etc. denote data fields, whereas `f`, `g`, etc. denote functions. The PEI notation for functions is derived from the lambda-calculus : any function `f` of domain dom(`f`) =`{x |P(x)}` is denoted as λ`x |(P(x)).f(x)`. Moreover, we denote λ`x.f(x)` for λ`x |(true).f(x)`, and λ`x |(P(x))` for λ`x |(P(x)).x`. Last, a function `f` defined on disjunctive subdomains is denoted as `f1 # f2`, and the domain of a composed function `f` $\circ$ `g` is `{x` $\in dom$(`g`) | `g(x)` $\in dom$(`f`) `}`.

Expressions are defined by applying operations on data fields. PEI defines one internal operation on the data field set, called *superimposition* : it is denoted as `/&/` and builds the sequences of values of its arguments. Three external operations are defined too, and associate a data field with a function :

- *the functional operation*, denoted as $\triangleright$, applies a function `f` (which may be a partial function) on values of a data field `X` (notation `f` $\triangleright$ `X`),
- *the change of basis* (denoted as `::`). Let `h` be a bijection, `h :: X` defines a data field that maps the values of `X` onto an other discrete domain,
- *the routing operation*, denoted as $\triangleleft$, moves the values of a data field on its domain : the data field `X` $\triangleleft$ `g` is such that its value mapped on some index z, $z \in dom$(`g`), "comes from" `X` at index `g`(z).

Example 2. A vector rotation

$$\texttt{SHIFT:A} \mapsto \texttt{B}$$
$$\begin{cases} \texttt{A} = \texttt{vector :: A} \\ \texttt{B} = \texttt{A} \triangleleft \texttt{rotate} \end{cases}$$

`vector=`λ`i |(0<=i<n)`
`rotate=`λ`i.((i-1) mod n)`

The routing operation has an inverse in PEI. It defines the so-called *geometrical reduction*, denoted as `g` $\cdot \triangleright$ `X`, which "routes" at index z the sequence (in an arbitrary order) of all the values mapped in `X` at indices y, such that y $\in dom$(`g`) and `g`(y) $= z$.

Example 3. A global summation

$$
\begin{array}{l}
\texttt{GSUM:X} \mapsto \texttt{S} \\
\left\{ \begin{array}{l} \texttt{X} = \texttt{vector :: X} \\ \texttt{S} = \texttt{sum} \rhd \texttt{(reduce} \cdot \rhd \texttt{X)} \end{array} \right. \\
\texttt{vector} = \lambda \texttt{i} \mid \texttt{(0<=i<n)} \\
\texttt{reduce} = \lambda \texttt{i.(n-1)}
\end{array}
$$

An equation in PEI expresses that both left and right hand side expressions define the same data field. A program is a set of equations within function definitions. The next section presents the semantical aspects for these constructions.

2.3 Semantics

The semantics of PEI is founded on the notion of discrete domain associated with a multiset. We call *drawing* v of a multiset M of values in V, a partial function from $\mathbb{Z}^n$ in V whose image is M. Since several drawings of a given multiset can be deduced one from another by applying a bijection, we define a data field $\texttt{X}$ as the abstraction of any drawing :

Definition 1. A *data field* $\texttt{X}$ is a pair $(v : \sigma)$, composed of a drawing v of a multiset $M_{\texttt{X}}$ and of a bijection σ such that $dom(v) \subset dom(\sigma)$[3].

This formal definition means that a data field $\texttt{X} = (v : \sigma)$ can be interpreted as the function $[\![\texttt{X}]\!] = v \circ \sigma^{-1}$. External operations in PEI are defined as following :

Definition 2. Let $\texttt{f}$ be a partial function from V to W and $\texttt{X} = (v : \sigma)$ a data field of values in V. The *functional operation* defines the data field $\texttt{f} \rhd \texttt{X}$ of values in W as the data field $(\texttt{f} \circ v : \sigma)$.

Proposition 3. $[\![\texttt{f} \rhd \texttt{X}]\!] = \texttt{f} \circ [\![\texttt{X}]\!]$.

Definition 4. Let $\texttt{h}$ be a bijection from $dom(v)$ onto $\mathbb{Z}^p$ and $\texttt{X} = (v : \sigma)$ a data field drawn in $\mathbb{Z}^n$. The *change of basis* defines the data field $\texttt{h::X}$ as $(v \circ \texttt{h}^{-1} : \sigma \circ \texttt{h}^{-1})$.

Proposition 5. $[\![\texttt{h::X}]\!] = [\![\texttt{X}]\!]$.

Definition 6. Let $\texttt{g}$ be a partial function from $\mathbb{Z}^n$ to $dom(v)$ and $\texttt{X} = (v : \sigma)$ a data field drawn in $\mathbb{Z}^n$. The *routing operation* defines the data field $\texttt{X} \lhd \texttt{g}$ as $(v \circ \texttt{g} : \sigma)$.

Proposition 7. $[\![\texttt{X} \lhd \texttt{g}]\!] = [\![\texttt{X}]\!] \circ \sigma \circ \texttt{g} \circ \sigma^{-1}$.

To complete this semantical presentation, let us consider the *geometrical reduction*. Let $\texttt{g}$ be a partial function from $dom(v)$ to $\mathbb{Z}^n$ and $\texttt{X} = (v : \sigma)$ a data field drawn in $\mathbb{Z}^n$. The geometrical reduction defines $\texttt{g} \cdot \rhd \texttt{X}$ as some data field $(w : \sigma)$ such that $dom(w) = \texttt{g}(dom(v))$ and $w(z)$ is a sequence of the values $v(y)$, y such that $\texttt{g}(y) = z$.

[3] the function $v \circ \sigma^{-1}$ then defines another drawing of $M_{\texttt{X}}$.

3 Refinement

Refinement process [Mor90] is a powerful programming concept, that applies to parallelism in Unity [CM88] for example. In PEI, any equation whose arguments are only inputs defines *preconditions* on these input data fields. Any other one, whose arguments may be intermediates or outputs defines *postconditions* on these data fields. Intuitively, we will say that a statement S is refined by a statement S' if any solution of S' is equivalent to a solution of S for some *equivalent* parameters.

Definition 8. Two data fields $\mathtt{X}= (v : \sigma)$ and $\mathtt{Y}= (v' : \sigma')$ are said to be equivalent (denoted as $\mathtt{X} \equiv \mathtt{Y}$) if v and v' represent the same multiset of values and $Card(dom(\sigma)) = Card(dom(\sigma'))$.

Definition 9. Let S and S' be two statements. S is said refined by S', denoted as S $\sqsubseteq$ S' (or S abstracts S', denoted S' $\sqsupseteq$ S)[4]

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

$\mathtt{Y}= (\mathrm{f1} \circ \mathrm{f2}) \rhd \mathtt{X}$	$\equiv$	$\mathtt{Y}= \mathrm{f1} \rhd (\mathrm{f2} \rhd \mathtt{X})$	(1)
$\mathtt{Y}= \mathtt{X} \lhd (\mathrm{g1} \# \mathrm{g2})$	$\equiv$	$\mathtt{Y}= (\mathtt{X} \lhd \mathrm{g1})\ \mathtt{/\&/}\ (\mathtt{X} \lhd \mathrm{g2})$	(2)
$\mathtt{Y}= \mathtt{X} \lhd (\mathrm{g1} \circ \mathrm{g2})$	$\sqsubseteq$	$\mathtt{Y}= (\mathtt{X} \lhd \mathrm{g1}) \lhd \mathrm{g2}$	(3)
$\mathtt{Y}= (\mathrm{f} \rhd \mathtt{X}) \lhd \mathrm{g}$	$\sqsubseteq$	$\mathtt{Y}= \mathrm{f} \rhd (\mathtt{X} \lhd \mathrm{g})$	(4)
$\mathtt{Y}= \mathrm{h} :: (\mathtt{X} \lhd \mathrm{h}^{-1} \circ \mathrm{g} \circ \mathrm{h})$	$\sqsubseteq$	$\mathtt{Y}= (\mathrm{h} :: \mathtt{X}) \lhd \mathrm{g}$	(5)
$\mathtt{Y}= \mathrm{h} :: (\mathrm{f} \rhd \mathtt{X})$	$\sqsubseteq$	$\mathtt{Y}= \mathrm{f} \rhd (\mathrm{h} :: \mathtt{X})$	(6)
$\mathtt{Y}= (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2}) \lhd \mathrm{g}$	$\equiv$	$\mathtt{Y}= (\mathtt{X1} \lhd \mathrm{g})\ \mathtt{/\&/}\ (\mathtt{X2} \lhd \mathrm{g})$	(7)
$\mathtt{Y}= \mathrm{h} :: (\mathtt{X1}\ \mathtt{/\&/}\ \mathtt{X2})$	$\equiv$	$\mathtt{Y}= (\mathrm{h} :: \mathtt{X1})\ \mathtt{/\&/}\ (\mathrm{h} :: \mathtt{X2})$	(8)
$\mathtt{Y}= (\mathrm{g1} \# \mathrm{g2}) \cdot\!\rhd \mathtt{X}$	$\sqsubseteq$	$\mathtt{Y}= (\mathrm{g1} \cdot\!\rhd \mathtt{X})\ \mathtt{/\&/}\ (\mathrm{g2} \cdot\!\rhd \mathtt{X})$	(9)
$\mathtt{Y}= (\mathrm{g1} \circ \mathrm{g2}) \cdot\!\rhd \mathtt{X}$	$\sqsubseteq$	$\mathtt{Y}= \mathrm{g1} \cdot\!\rhd (\mathrm{g2} \cdot\!\rhd \mathtt{X})$	(10)

Notice that rule (2) holds provided functions `g1` and `g2` are defined on disjunctive sub-domains and rule (5) is an equivalence provided `h` is bijective on $dom(\mathtt{X})$ which denotes the drawing domain of `X`. The two last rules associated with the following proposition define transformations of a reduction in some composition of routing operations.

Proposition 10. $\mathtt{g} \cdot\!\rhd \mathtt{X}$ *is equal to* $\mathtt{X} \lhd \mathtt{g}^{-1}$ *if and only if* `g` *is bijective.*

[4] If S and S' refine one each other then S $\equiv$ S'.

4 Transformation strategies

The refinement calculus is used to refine the initial specification accordingly to particular issues we want to achieve in the new solution. Conversely, the initial specification can be seen as an *abstract* version of the refined specification. In the following, we present strategic transformations, in one way or the other to reach particular goals such as minimization of data movements, introduction of broadcasts or reduction functions for instance.

These transformations are illustrated by the matrix-vector multiplication problem. First of all we consider a kind of systolic solution (cf. program `MATVECT1`) in which regular neighbour-to-neighbour communications "schedule" the execution. Assuming the same execution model, the equivalent program `MATVECT2` is deduced by simplifying the communications. This last solution allows to abstract a data-parallel solution (cf. program `MATVECT3`) by discovering a non-recursive definition of the data fields : this new definition yields broadcasts of the original matrix and vector. Assuming such a broadcast primitive is available in the execution model, the schedule of the solution can then be changed and leads to an elegant data-parallel expression (cf. program `MATVECT4`). This program can then be completed by introducing a step by step summation (cf. program `MATVECT5`) and then generalized in a global summation through a reduction operator (cf. program `MATVECT6`).

4.1 Communications simplification

The first transformation technique we describe aims at simplifying routing functions by finding a suitable change of basis which, when applied to equations results in new definitions with simpler communication schemes. The simplification is possible provided that all routing functions are of the general form : $f(p,z) = ((p+az+b)\ mod\ n, z-1)$ where n is a structure parameter of the problem, the dependency in z is uniform (the argument z could represent the time direction for instance) and the other dependencies are piecewise *quasi-uniform* refering to z. Although these conditions may appear to be strong, this refinement technique addresses a broad class of algorithms.

Definition 11. Let z be an argument of f. Any other argument p of f is said to be *quasi-uniform* refering to z if its dependency is an affine expression of z.

Example Let us consider a first program, which is a systolic version of $p = A \times b$, where b is a vector of length n, initially lying on the diagonal of matrix A. In the first part of the example we will only consider the products computations and we leave to the second part the products summations. At each step k, n computations $p_{i,i} = A_{i,i} \times b_i$ occur on the diagonal. All elements of A are then shifted one position left, the elements of b being moved up-left along the diagonal (all communications are processed toroïdally) before the next computation takes place (see figure 1). In the end we get all the desired products on the diagonal

plane ($i = j$) of the cube.
The algorithm described above is transcribed by the following PEI program : `P` is defined as being the superimposition of data fields `A` and `B`, both recursively defined using routing functions `left` and `upleft` and initial values `A0` and `B0` defined from data `M` and `V`.

$$
\begin{array}{l}
\texttt{MATVECT1:(M,V)} \mapsto \texttt{P} \\
\left\{
\begin{array}{l}
\texttt{M} = \texttt{aligna :: A0} \\
\texttt{V} = \texttt{alignb :: B0} \\
\texttt{A} = \texttt{A0 /\&/(A} \lhd \texttt{left)} \\
\texttt{B} = \texttt{B0 /\&/(B} \lhd \texttt{upleft)} \\
\texttt{P} = \texttt{prod} \rhd \texttt{(A} \lhd \texttt{diag /\&/B)}
\end{array}
\right. \\
\texttt{doma} = \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<n \& k=0)} \\
\texttt{domb} = \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<n \& i=j \& k=0)} \\
\texttt{aligna o doma} = \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<n \& k=0).(i,j)} \\
\texttt{alignb o domb} = \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<n \& i=j \& k=0).(i)} \\
\texttt{left} \quad = \lambda\texttt{(i,j,k)}\ |\texttt{(0<k<n \& 0<=i,j<n).(i,(j+1) mod n,k-1)} \\
\texttt{upleft} = \lambda\texttt{(i,j,k)}\ |\texttt{(0<k<n \& 0<=i,j<n).((i+1) mod n,(j+1) mod n,k-1)} \\
\texttt{prod} \quad = \lambda\texttt{(a}\cdot\texttt{b).(a*b)} \\
\texttt{diag} \quad = \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<=n \& i=j)}
\end{array}
$$

Routing decomposition Both `left` and `upleft` can be decomposed so that a common routing (bijective) function α appears in the expressions :

$$
\begin{array}{lll}
\texttt{left} = & \lambda\texttt{(i,j,k)}\ |\texttt{(0<=j<n).(i,(j+1) mod n,k) o} & \\
& \lambda\texttt{(i,j,k)}\ |\texttt{(0<k<n \& 0<=i,j<n).(i,j,k-1)} & = \alpha\ \texttt{o prev} \\
\texttt{upleft} = & \lambda\texttt{(i,j,k)}\ |\texttt{(0<=j<n).(i,(j+1) mod n,k) o} & \\
& \lambda\texttt{(i,j,k)}\ |\texttt{(0<=i,j<n).((i+1) mod n,j,k) o} & \\
& \lambda\texttt{(i,j,k)}\ |\texttt{(0<k<n \& 0<=i,j<n).(i,j,k-1)} & = \alpha\ \texttt{o up o prev}
\end{array}
$$

The names given to the new functions we obtain in this decomposition reflect the fact that moving points along the diagonal is equivalent to moving them left and then up. The idea is thus to demonstrate that a program that keeps the matrix `A` steady while the `B` vector is moved up is equivalent to the previous program, or in other words that we can simplify the routing functions by removing the α parts from both routing functions. Such a result comes from a general strategy presented hereunder.

Simplification strategy Let us consider the equation $\texttt{A} = \texttt{A0/\&/(A} \lhd \texttt{f)}$ and some change of basis `h` whose domain contains $dom(\texttt{A})$. Then, from the rules of the refinement calculus

$$
\begin{array}{lll}
(E1) & & \texttt{A} = \texttt{A0/\&/(A} \lhd \texttt{f)} \\
& \equiv & \texttt{h :: A} = \texttt{h::(A0/\&/(A} \lhd \texttt{f))} \\
& \equiv & \texttt{h :: A} = \texttt{(h :: A0)/\&/(h::(A} \lhd \texttt{f))} \qquad (8) \\
& \equiv & \texttt{h :: A} = \texttt{(h :: A0)/\&/((h :: A)} \lhd \texttt{f')} \quad (5 \text{ as } dom(\texttt{A}) \subseteq dom(\texttt{h})) \\
(E2) & \equiv & \texttt{A'} = \texttt{(h :: A0)/\&/(A'} \lhd \texttt{f')}
\end{array}
$$

where `f' = h o f o h`$^{-1}$ and `A' = h::A`

Therefore any bijection satisfying the previous conditions allows to transform $(E1)$ in $(E2)$; we now show a sufficient condition for it exists a particular bijection `h` such that `f =` α `o f'`.

Theorem 12. *Assume a definition of the form* $(E1)$ *where* $f = \lambda(p,z).((p + az + b) \ mod \ n, z-1)$*, and let* $\alpha = \lambda(p,z)|(0 \leq p < n).((p+b') \ mod \ n, z)$*, it then exists a bijection* h *built from* α*, such that* $(E1)$ *can be transformed into an equivalent definition* $(E2)$ *with* f' *such that* $f = \alpha \circ f'$*.*

Proof. Let us consider a set of bijections h_i defined on disjunctive domains varying with z (the uniform argument) and h_i is obtained by recursive compositions of the bijective function $\alpha = \lambda(p,z)|(0 \leq p < n).((p+b') \ mod \ n, z)$ with itself such that $h_i = \lambda(p,z)|(z = i).\alpha^i(p,z)$. We thus have

$$\begin{aligned}
h_1 &= \lambda(p,z)|(z = 1 \wedge 0 \leq p < n).((p + b') \ mod \ n, z) \\
&= \lambda(p,z)|(z = 1 \wedge 0 \leq p < n).((p + zb') \ mod \ n, z) \\
h_2 &= \lambda(p,z)|(z = 2 \wedge 0 \leq p < n).((p + 2b') \ mod \ n, z) \\
&= \lambda(p,z)|(z = 2 \wedge 0 \leq p < n).((p + zb') \ mod \ n, z) \\
&\vdots \\
h_r &= \lambda(p,z)|(z = r \wedge 0 \leq p < n).((p + rb') \ mod \ n, z) \\
&= \lambda(p,z)|(z = r \wedge 0 \leq p < n).((p + zb') \ mod \ n, z)
\end{aligned}$$

Let $h = h_1 \# h_2 \# \ldots \# h_r$. As $z \in [1 \ldots r]$, we find that $h = \lambda(p,z)|(0 \leq p < n).((p + zb') \ mod \ n, z)$ which is a bijection. Given

$$\begin{aligned}
f &= \lambda(p,z).((p + az + b) \ mod \ n, z - 1) \\
\alpha &= \lambda(p,z)|(0 \leq p < n).((p + b') \ mod \ n, z) \\
f' &= \lambda(p,z).((p + az + b - b') \ mod \ n, z - 1)
\end{aligned}$$

we have $f = \alpha \circ f'$, and h is a change of basis such that

$$\begin{aligned}
& h \circ f \circ h^{-1} \\
&= h \circ \lambda(p,z).((p + az + b) \ mod \ n, z - 1) \circ \lambda(p,z).((p - zb') \ mod \ n, z) \\
&= h \circ \lambda(p,z).((p - zb' + az + b) \ mod \ n, z - 1) \\
&= \lambda(p,z).((p + zb') \ mod \ n, z) \circ \lambda(p,z).((p - zb' + az + b) \ mod \ n, z - 1) \\
&= \lambda(p,z).((p + az + b - b') \ mod \ n, z - 1) \\
&= f'
\end{aligned}$$

□

Example (continued) In order to apply the previous strategy and find a new program with simplified routing functions we search h as shown previously. Since in our example $\alpha = \lambda(i,j,k)|(0 \leq j < n).(i, (j+1) \ mod \ n, k)$ then we deduce $h = \lambda(i,j,k)|(0 \leq j < n).(i, (j+k) \ mod \ n, k)$. Note that theorem 12 fully applies on `MATVECT1` since $dom($`A`$)$, $dom($`B`$)$ and $dom($`P`$) \subseteq dom($`h`$)$, and `h :: A0` reduces to `A0` and `h :: B0` reduces to `B0`. We thus get the following `MATVECT2` program:

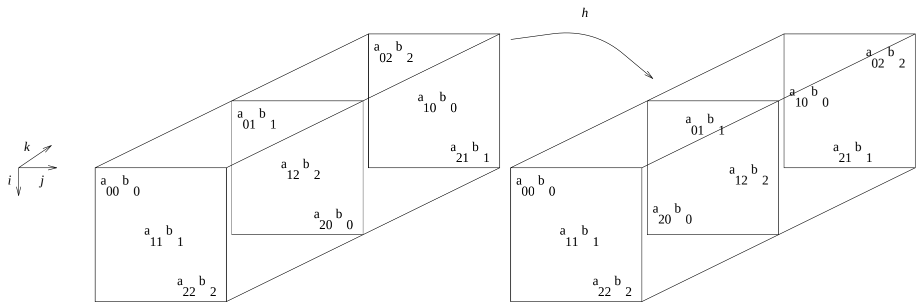

Fig. 1. Transformation of `MATVECT1` in `MATVECT2` for $n = 3$

```
MATVECT2:(M,V) ↦ P'
 ⎧ M = aligna :: A0
 ⎪ V = alignb :: B0
 ⎨ A' = A0 /&/ (A' ◁ prev)
 ⎪ B' = B0 /&/ (B' ◁ up o prev)
 ⎩ P' = prod ▷ (A' ◁ diag' /&/ B')
prev  =λ(i,j,k) |(0<k<n & 0<=i,j<n).(i,j,k-1)
up    =λ(i,j,k) |(0<=i,j<n).((i+1) mod n,j,k)
diag' =λ(i,j,k) |(i=(j+k) mod n)
```

4.2 Removing recursion from data fields definition

The previous program is a typical systolic program since all its routing functions are point-to-point communications. If we assume a broadcasting communication operator is available the communications can be rewritten in a data-parallel style. The relationship between recursive definitions of data fields and broadcasting can be stated as follows :

Theorem 13. *Let g be a function defined by the recursion $g = i \# g \circ s$, where i is the identity and s a function. The* PEI *equation* $\mathtt{Y} = \mathtt{X} \lhd g$ *is refined by the following recursive definition of* `Y`*:* $\mathtt{Y} = \mathtt{X} \ \lhd \ i \ \mathtt{/\&/} \ \mathtt{Y} \ \lhd \ s$

Proof. From rules of the refinement calculus :
$\mathtt{Y} = \mathtt{X} \lhd g$
$\equiv \mathtt{Y} = \mathtt{X} \lhd (i \# g \circ s)$
$\sqsubseteq \mathtt{Y} = \mathtt{X} \lhd i \ \mathtt{/\&/} \ \mathtt{X} \lhd (g \circ s) \qquad (rule\ 2)$
$\sqsubseteq \mathtt{Y} = \mathtt{X} \lhd i \ \mathtt{/\&/} \ (\mathtt{X} \lhd g) \lhd s \qquad (rule\ 3)$
$\equiv \mathtt{Y} = \mathtt{X} \lhd i \ \mathtt{/\&/} \ \mathtt{Y} \lhd s$
□

This is the only clue we need to write another program `MATVECT3` refined by the `MATVECT2` program. Note that the new program uses broadcasting of data, which is a main feature of the data-parallel paradigm.

$$\texttt{MATVECT3:(M,V)} \mapsto \texttt{P'}$$
$$\begin{cases} \texttt{M} = \texttt{aligna :: A0} \\ \texttt{V} = \texttt{alignb :: B0} \\ \texttt{A'} = \texttt{A0} \lhd \texttt{spreada} \\ \texttt{B'} = \texttt{B0} \lhd \texttt{spreadb} \\ \texttt{P'} = \texttt{prod} \rhd \texttt{(A'} \lhd \texttt{diag' /\&/ B')} \end{cases}$$
$$\texttt{spreada} = \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n).(i,j,0)}$$
$$\texttt{spreadb} = \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n \& 0<=i<n).((i+k) mod n,j,0)}$$

Proof. From theorem 13, the last statement is obtained by writing functions `spreada` and `spreadb` in the following way :

$$\begin{aligned} \texttt{spreada} &= \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n).(i,j,0)} \\ &= \lambda\texttt{(i,j,k)} \,|\texttt{(k=0)} \;\#\; \lambda\texttt{(i,j,k)} \,|\texttt{(0<k<n).(i,j,0)} \\ &= \lambda\texttt{(i,j,k)} \,|\texttt{(k=0)} \;\#\; \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n).(i,j,0)} \\ &\quad \texttt{o}\ \lambda\texttt{(i,j,k)} \,|\texttt{(0<k<n).(i,j,k-1)} \\ &= \mathrm{i} \;\#\; \texttt{spreada o prev} \\ \texttt{spreadb} &= \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n \& 0<=i<n).((i+k) mod n,j,0)} \\ &= \lambda\texttt{(i,j,k)} \,|\texttt{(k=0 \& 0<=i<n).((i+k) mod n,j,0)} \;\# \\ &\quad \lambda\texttt{(i,j,k)} \,|\texttt{(0<k<n \& 0<=i<n).((i+k) mod n,j,0)} \\ &= \lambda\texttt{(i,j,k)} \,|\texttt{(k=0 \& 0<=i<n)} \;\# \\ &\quad \lambda\texttt{(i,j,k)} \,|\texttt{(0<=k<n \& 0<=i<n).((i+k) mod n,j,0)} \\ &\quad \texttt{o}\ \lambda\texttt{(i,j,k)} \,|\texttt{(0<=i<n).((i+1) mod n,j,k-1)} \\ &= \mathrm{i} \;\#\; \texttt{spreadb o up o prev} \end{aligned}$$

□

From definition of data field `P'`, we come to the conclusion that products are only defined on the diagonal plane of the cube and that these results could be packed in a two-dimensional data field. The transformation is formally described by applying an appropriate change of basis h such that $h \circ \texttt{diag'} = \lambda\texttt{(i,j,k)} \,|\texttt{(i=(j+k) mod n).(i,j)}$. Definition of `P'` becomes :

$$\texttt{h :: P'} = \texttt{prod} \rhd \texttt{(h :: (A'} \lhd \texttt{diag') /\&/ h :: B')}$$

If we let $\texttt{A}'' = \texttt{h :: (A'} \lhd \texttt{diag')}$, `B''= h::B'`

$$\begin{aligned} &\text{then} \quad \texttt{A}'' = \texttt{h :: ((A0} \lhd \texttt{spreada)} \lhd \texttt{diag')} \\ &\sqsupseteq \texttt{A}'' = \texttt{h :: (A0} \lhd \texttt{spreada o diag')} && (3) \\ &\equiv \texttt{A}'' = \texttt{(h :: A0)} \lhd \texttt{h o spreada o diag' o h}^{-1} && (5) \\ &\equiv \texttt{A}'' = \texttt{M} \lhd \lambda\texttt{(i,j)} \,|\texttt{(0<=i,j <n)} = \texttt{M} \\ &\text{and} \quad \texttt{B}'' = \texttt{h :: B'} \\ &\equiv \texttt{B}'' = \texttt{h :: (B0} \lhd \texttt{spreadb)} \\ &\equiv \texttt{B}'' = \texttt{(h :: B0)} \lhd \texttt{h o spreadb o h}^{-1} && (5) \\ &\equiv \texttt{B}'' = \texttt{(h :: B0)} \lhd \lambda\texttt{(i,j)} \,|\texttt{(0<=i,j<n).(j,j)} \end{aligned}$$

If we rename $\texttt{V}' = \texttt{h :: B0}$ and $\texttt{P}'' = \texttt{h :: P}'$ then we get the abstract version :

$$\texttt{MATVECT4:(M,V')} \mapsto \texttt{P"}$$
$$\begin{cases} \texttt{A"} = \texttt{M} \\ \texttt{B"} = \texttt{V'} \lhd \texttt{spreab'} \\ \texttt{P"} = \texttt{prod} \rhd \texttt{(A" /\&/ B")} \end{cases}$$
$$\texttt{spreadb'} = \lambda\texttt{(i,j)}\ |\texttt{(0<=i,j<n).(j,j)}$$

4.3 Introduction of a reduction operator

We have only focused so far on the first phase of the algorithm, and we shall now complete the previous program with the summation of products. Once again it is a "systolic" version where we explicitly describe the communication path for the points to be added, to eventually reach a "data-parallel" version in which a reduction operator is used.

$$\texttt{MATVECT5:(M,V)} \mapsto \texttt{R}$$
$$\begin{cases} \texttt{A} = \texttt{M} \\ \texttt{B} = \texttt{V} \lhd \texttt{spreadb'} \\ \texttt{P} = \texttt{prod} \rhd \texttt{(A /\&/ B)} \\ \texttt{S} = \texttt{P /\&/ S} \lhd \texttt{pre} \\ \texttt{R} = \texttt{sum} \rhd \texttt{(S} \lhd \texttt{last)} \end{cases}$$
$$\texttt{pre} = \lambda\texttt{(i,j)}\ |\texttt{(0<=i<n \& 0<j<n).(i,j-1)}$$
$$\texttt{last} = \lambda\texttt{(i,j)}\ |\texttt{(0<=i<n \& j=n-1)}$$
$$\texttt{sum} = \texttt{id}\ \#\ \lambda\texttt{(a}\cdot\texttt{b).(a+sum(b))}$$

Note that the field `S` is the result of an operator called *scan* in the data-parallel style. As previously, the recursive definition of `S` in the last statement can be abstracted to a non-recursive one (from theorem 14).

Theorem 14. *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* $\texttt{Y} = f \cdot \rhd \texttt{X}$ *is refined by the following statement, which involves the recursive definition of an intermediate data field* `T` *:*
$$\texttt{T} = \texttt{X /\&/ T} \lhd s$$
$$\texttt{Y} = \texttt{T} \lhd i$$

Proof. From rules of the refinement calculus:

$\texttt{Y} = f \cdot \rhd \texttt{X}$
$\equiv \texttt{Y} = (i \# f \circ s^{-1}) \cdot \rhd \texttt{X}$
$\sqsubseteq \texttt{Y} = i \cdot \rhd \texttt{X /\&/ } (f \circ s^{-1}) \cdot \rhd \texttt{X}$ (*rule* 9)
$\sqsubseteq \texttt{Y} = i \cdot \rhd \texttt{X /\&/ } f \cdot \rhd (s^{-1} \cdot \rhd \texttt{X})$ (*rule* 10)
$\equiv \texttt{Y} = \texttt{X} \lhd i \texttt{ /\&/ } f \cdot \rhd (\texttt{X} \lhd s)$ (*proposition* 10)
$\sqsubseteq \texttt{Y} = \texttt{X} \lhd i \texttt{ /\&/ } (\texttt{X} \lhd s) \lhd i \texttt{ /\&/ } f \cdot \rhd ((\texttt{X} \lhd s) \lhd s)$
$\equiv \texttt{Y} = \texttt{T} \lhd i$

□

It leads to the following statement expressing a reduction in the data-parallel programming model:

$$\mathtt{MATVECT6:(M,V)} \mapsto \mathtt{R}$$
$$\begin{cases} \mathtt{A} = \mathtt{M} \\ \mathtt{B} = \mathtt{V} \lhd \mathtt{spreadb'} \\ \mathtt{P} = \mathtt{prod} \rhd \mathtt{(A\ /\&/\ B)} \\ \mathtt{R} = \mathtt{sum} \rhd \mathtt{(red}\ \cdot \rhd \mathtt{P)} \end{cases}$$
$$\mathtt{red} = \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n).(i,n-1)}$$

Proof. From theorem 14, function `red` can be rewritten as :

$$\begin{aligned} \mathtt{red} &= \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n).(i,n-1)} \\ &= \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i<n\ \&\ j=n-1).(i,n-1)}\ \# \\ &\quad \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n\ \&\ j<n-1).(i,n-1)} \\ &= \mathtt{last}\ \#\ \mathtt{red\ o}\ \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n\ \&\ j<n-1).(i,j+1)} \\ &= \mathtt{last}\ \#\ \mathtt{red\ o}\ \lambda\mathtt{(i,j)}\ |\mathtt{(0<=i,j<n).(i,j-1)}^{-1} \\ &= \mathtt{last}\ \#\ \mathtt{red\ o\ pre}^{-1} \end{aligned}$$

□

5 Conclusion

This article was an attempt to use a refinement calculus and its reverse abstraction to transform programs within some target constraints of the execution model. Examples of such transformations were developed in this article, all founded on routing operations rewriting. Note that abstraction might be seen as a *reverse engineering* method to prove the correctness of refined programs since the most abstract version is the closest from the specification. PEI offers a convenient foundation for all these concepts because it enables these formal transformations. Further work could consist in the implementation of these strategies and would benefit from recent work like reduction detection([Red95]).

References

[BL92] L. Bougé and J.-L. Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.

[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.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. June 1993.

[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.

[Red95] Xavier Redon. *Détection et exploitation des récurrences dans les programmes scientifiques en vue de leur parallélisation.* PhD thesis, U. Paris 6, Jan 1995.

[Vio94] E. Violard. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4(3):313–328, 1994.

[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