On deriving HPF code from Pei programs

Stéphane Genaud

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

Keywords: data-parallelism, HPF, specifications, refinement, compilation.

Abstract: The paper presents some solutions for the generation of HPF code from PEI specifications. We delimit the class of specifications that can be translated and we discuss the operational semantics given to these specifications. In conclusion, we outline that parallel programming can benefit from an underlying formalism if we can build gateways from such formalisms to real code.

1 Introduction

To use parallel computing effectively, many efforts have targeted the improvement of parallel languages in terms of readability, and their compilation to gain more performance. One major improvement certainly was the introduction of data-parallel languages such as C*[Thi90] or High Performance Fortran (HPF) [HPF93]. This programming model allow to write portable programs, much more rapidly and clearly than in the message passing paradigm. Some projects have also designed functional languages for that model (Nesl [Ble95] and Sisal [MSA+85] for example), to get rid of classical drawbacks of imperative languages. For all these languages, much attention is paid to the performance of the compilers (e.g. [AICK93]), which is the key to make this style of programming popular.

However, when it comes to program correctness, there still is a need for software tools to be able to formally state about programs meaning. Our contribution is a language called PEI, that can model most of the data-parallel programming model concepts. Our methodology for parallel program development can be seen as a specialized form of general formalisms such as the Bird-Meertens Formalism [Bir87,Ski93] : the language incorporates the *geometrical* primitives allowing the programmer to handle the placement of data, and focuses on the ways these geometrical aspects can be transformed so as to obtain new specifications from an initial one. PEI is close to ALPHA [Mau89] or CRYSTAL [CiCL91], because all were originally defined to model systems of affine recurrence equations. The main difference lies in the fact that data placement is part of our formalism, whereas in ALPHA for example, the mapping of data onto the computer architecture must be done through an external tool. Using the refinement rules provided with the PEI formalism, the user is thus able to transform its initial specification, to a new one in which the data placement has changed, and hopefully, fits better the computer architecture.

This paper addresses the problem of translating a high-level language into HPF code. We present here our effort to produce executable code from specifications written in PEI. The discussion is centered about what class of statements can be translated, and about the operational semantics implicitly given to PEI primitives when translating such statements to an imperative data-parallel program. We put forward the interest of having such an underlying formalism, by showing an example of program derivation and the translation in HPF code.

2 The Pei language

2.1 Pei objects

Generally speaking, we can consider a problem as a relation between input and output *multisets* of values. 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. Let us call this data field `V`. Obviously, the multiset might be mapped in a different manner, for example onto points whose indices (i, j) in $\mathbb{Z}^2$, are such that $(0 \le i, j \le 1)$. 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 are said to be *equivalent*. 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 denoted in PEI by the equation:

$$\begin{gathered}\mathtt{M} = align{::}\ \mathtt{V} \\ \text{where } align(i) = (0 \le i \le 3).(i \ mod \ 2, i \ div \ 2)\end{gathered}$$

A PEI *statement* is composed of such unoriented equations, each of them connecting two data field expressions, and thus connecting the input and output data fields. A statement is a *program*, if and only if the set of equations has at most one solution, i.e. from any set of input valued data fields there exists at most one set of resulting output valued data fields. Here is a program for computing the prefix-sum of n numbers:

Example 1 Prefix-Sum of n numbers $(n>1)$

$$\mathtt{PrefixSum\ :\ A} \mapsto \mathtt{X}$$
$$\begin{cases} \mathtt{A} = dom{::}\ \mathtt{A} \\ \mathtt{X} = (\mathtt{A} \lhd first)\ \mathtt{/\&/}\ (add \rhd\ (\mathtt{A}\ \mathtt{/\&/}\ \mathtt{X}\ \lhd pre)) \end{cases}$$

$$\begin{aligned} dom(i) &= (1 \le i \le n, n>1).(i) \\ pre(i) &= (1<i \le n).(i-1) \\ first(i) &= (i=1).(i) \\ add(a;b) &= (a+b) \end{aligned}$$

– the first equation defines the mapping of the input data field `A`: its values are placed onto a line segment $[1..n]$ of $\mathbb{Z}$,
– given the data field `A`, figure 1 intuitively shows that data field `X` is a solution of the second equation: its values are the prefix computations of the sums of the values in `A`. The expression (`X` $\lhd pre$) defines a data field resulting from `X` by shifting its values from left to right. They are then superimposed with the values of `A` in the expression (`A /&/ (X` $\lhd pre$)), and summations take place where pairs have been formed. The first element of `A` is then selected and superimposed with the summations[1].

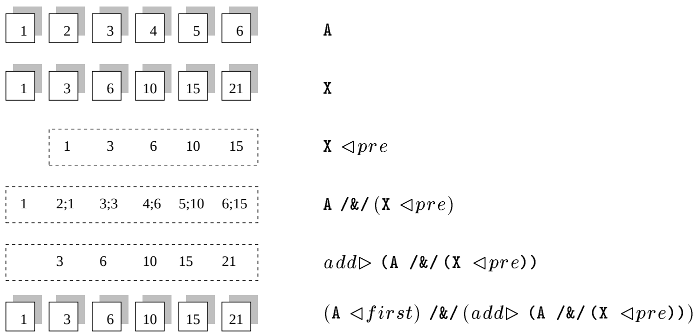

Fig. 1. `X` is a solution of the second equation

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

– operations that modify the reference domain onto which values are mapped (like *align*:: `V`) called *change of basis* and denoted by `::`
– operations that "move" values in the reference domain, called *geometrical* operations (or routings), and denoted by $\lhd$.
The geometrical operation has an inverse, called the *geometrical reduction* and denoted by $;\rhd$. 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` $= merge;\rhd$ `V`, where $merge(i) = (0 \le i < 4).(0)$ has all its values mapped at index point 0, listed in a sequence of arbitrary order.
– operations called *functional* operations, denoted by $\rhd$, perform element-wise computation on data field values.
– 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.

2.2 Notations

As typographic conventions, we will use `A`, `B`, `X`, *etc.* for data fields, whereas f, g, etc. denote functions, and we use the letter n to denote the size parameter of

[1] note that the problem is one-dimensional and therefore inherently sequential

a problem. To improve the readability, only function names appear in the equations, their definition being reported outside the system. A function $f : x \to e(x)$ of domain $dom(f) = \{x \mid P(x)\}$ and image $img(f) = \{e(x) \mid P(x)\}$ is denoted as $f(x) = P(x).(e(x))$. A function f defined on disjunctive sub-domains is denoted as $f_1 \sharp f_2$, 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.

2.3 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 values of the data field. For instance, consider `M` on figure 2 as a reference mapping representing a processor grid, then the value 2 mapped at index point 1 by `V` $= (v_{\mathtt{V}}{:}\sigma_{\mathtt{V}})$ would be computed by processor $\sigma_{\mathtt{V}}(1) = (1{,}0)$.

Definition 1. *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 $dom(v) \subseteq dom(\sigma)$.*

Operation semantics The data field definition enables us to formally describe the operations in PEI. In the following, `X` denotes a data field $(v{:}\sigma)$.

- Functional operation. Let f be a function. When applied on a `X` such that $img(v) \subseteq dom(f)$, the functional operation on f, denoted as $f\triangleright$, defines the element-wise application of f on `X`. It means that $f\triangleright$ `X` is $(f \circ v : \sigma)$.
- Geometrical operation. Let g be a function. When applied on a `X` such that $dom(g) \subseteq dom(\sigma)$ and $img(g) \subseteq dom(v)$, the geometrical operation on g, denoted as $\triangleleft g$, defines the data field `X` $\triangleleft g$ as $(v \circ g : \sigma)$.
  Conversely, let g be a function. When applied on a `X` such that $dom(g) \subseteq dom(v)$ and $img(g) \subseteq dom(\sigma)$, the geometrical reduction on g, denoted

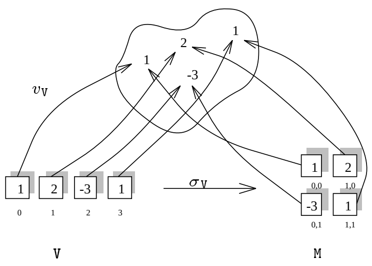

Fig. 2. Relation between two equivalent mappings

as $g;\triangleright$, defines the data field $g;\triangleright$ X as some data field $(w : \sigma)$ such that $dom(w) = g(dom(v))$ and $w(z)$ is a sequence of the values $v(y)$, y such that $g(y) = z$.
- Change of basis operation. Let h be a bijection. When applied on a X such that $dom(v) \subseteq dom(h)$, the change of basis on h, denoted as h ::, defines the data field h:: X as $(v \circ h^{-1} : \sigma \circ h^{-1})$.
- Superimposition. We say that a data field is *conform* with another if its bijection is a restriction of the other's bijection. Superimposition of $\mathtt{X}_1 = (v_1{:}\sigma_1)$ and $\mathtt{X}_2 = (v_2{:}\sigma_2)$ such that $\sigma_1 = \sigma_2\backslash_{dom(\sigma_1)}$ (conformity) defines $\mathtt{X}_1$/&/$\mathtt{X}_2$ as $(w : \sigma_2)$, where $w(z) = v_1(z); v_2(z)$ on $dom(v_1) \cap dom(v_2)$, $w(z) = v_1(z)$ on $dom(v_1) - dom(v_2)$ and $w(z) = v_2(z)$ on $dom(v_2) - dom(v_1)$.

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, 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 the 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 $\mathtt{A} = add\triangleright$ $(size$:: B) is the boolean expression:

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

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

2.4 Refinement of Pei statements

We only summarize here the way statements are refined in our formalism and a complete definition of the refinement of PEI statements can be found in [VGP97].

Refinement consists in replacing some data field expressions in a program by an equal expression, or a refined one, that is an expression that involves more constraints on the values and reference functions. We have set up (and proved) a list of rewrite rules which form the basis for our refinement calculus. Table 1 shows some of these rules: the notation $E \rightarrow E'$, where E and E' are equations, indicates that substituting in a statement, E' for E gives a new refined statement. When a *with* clause appears after the rule, it means that E' has the same constraints than E but should also verify the conditions of the *with* clause.

(1)	$\mathtt{Y} = \mathtt{X} \longrightarrow h{::}\,\mathtt{Y} = h{::}\,\mathtt{X}$
	with $dom(\sigma_{\mathtt{X}}) \subseteq dom(h) \wedge dom(\sigma_{\mathtt{Y}}) \subseteq dom(h)$
(2a)	$\mathtt{Y} = \mathtt{X} \lhd (f_1 \circ f_2) \longrightarrow \mathtt{Y} = (\mathtt{X} \lhd f_1) \lhd f_2$
(2b)	$\mathtt{Y} = (\mathtt{X} \lhd f_1) \lhd f_2 \longrightarrow \mathtt{Y} = \mathtt{X} \lhd (f_1 \circ f_2)$
	with $img(f_2) \subseteq dom(f_1) \subseteq dom(\sigma_{\mathtt{X}})$
(3a)	$\mathtt{Y} = (f_1 \circ f_2){::}\,\mathtt{X} \longrightarrow \mathtt{Y} = f_1{::}\,(f_2{::}\,\mathtt{X})$
(3b)	$\mathtt{Y} = f_1{::}\,(f_2{::}\,\mathtt{X}) \longrightarrow \mathtt{Y} = (f_1 \circ f_2){::}\,\mathtt{X}$
	with f_1 and f_2 two bijections

Table 1. Examples of rewrite rules

3 HPF translation

Pei offers a fine control over expressions, and our aim is to translate Pei statements to a parallel imperative programming language to observe the impact of refinement on the data-parallel code obtained. We have chosen HPF because it offers the best facilities for data alignment, and also of course, because it is widely known.

Our idea is to associate to each equation of a Pei statement, a set of HPF instructions. Of course, the translation is not always possible because we cannot give an operational semantic to any Pei statement. In the next paragraphs we first present the class of translatable Pei programs, and we then explain what obstacles made us choose this limited set of equations. The translator is written in Caml and uses the Omega library [KMP+96] to perform the set theoretic operations.

3.1 Normalization in translatable form

Our translator only accepts a subset of Pei programs. This subset is presented in the table below and is called the translatable form. If a Pei program has all its equations in this form it is called a *translatable* program.

Translatable form
1. Geometrical operation : $\mathtt{Y} = \mathtt{X} \lhd g$
2. Functional operation : $\mathtt{Y} = f \rhd (\mathtt{X}_1 \ /\&/ \ldots /\&/ \mathtt{X}_p)$
3. Change of basis : $\mathtt{Y} = h{::}\ \mathtt{X}$
4. Data field property : $\mathtt{Y} = h{::}\ \mathtt{Y}$
5. Geometrical reduction : $\mathtt{Y} = f \rhd (g; \rhd \mathtt{X})$
6. Recursive definition : $\mathtt{Y} = \mathtt{X}\ /\&/\ f \rhd (\mathtt{X'}\ /\&/\ (\mathtt{Y} \lhd g_1)\ /\&/ \ldots /\&/ (\mathtt{Y} \lhd g_q))$

In section "Translation of equations" we give the HPF instructions associated with of each of these expressions, but we first explain our choice for this subset.

Semantic obstacles There are two main characteristics that make PEI statements not directly translatable into data-parallel code:

- PEI equations may not be considered as assignments. For instance an equation like $f{::}\ \mathtt{X} = g{::}\ \mathtt{Y}$ is not accepted by our translator, because the equation is not an expression of X or Y, which could be interpreted as an assignment of X or Y. The transformation of the equation so as to obtain an expression of X could be: apply f^{-1} on both sides (rule 1 and (3b) subject to conditions validity) and compute the new change of basis $f \circ g^{-1}$ to get the new equation $\mathtt{X} = (f \circ g^{-1}){::}\ \mathtt{Y}$.
- Recursive equations may not have an operational semantics. For a direct translation, we need to extract from such equations, a schedule for the computation, that is find which dimension(s) can represent the time, and which one(s) can represent the processors space. This problem has been widely addressed in the automatic parallelization literature ([Fea92] for example), and we use a space-time change of basis to transform a recursive definition so that it matches the previous requirement. The new PEI equation is obtained by applying a change of basis on both sides of the equation.

Manual transformations are therefore required to overcome these difficulties, and get a statement close to the translatable form. Practically, the user is assisted in the refinement phase, by a software tool called PEI, which enables to apply the rules and checks for the validity of the transformation.

Syntactic obstacles There are other obstacles to the translation, which relates to what can be expressed in HPF. Let us examine two difficulties:

- Consider the equation $\mathtt{Y} = (\mathtt{X} \lhd g) \lhd g'$. We could at a first glance, compute the indexing function $g \circ g'$ used to access elements of X. However, this transformation of the indexing operates a slight change in the semantics and requires to be cautious. That is precisely the purpose of the refinement rule (2b), which states under which conditions such a transformation can be done. In this case, either the user can decide to refine the equation, or let the translator decompose the equation into two equations, namely $\mathtt{X'} = \mathtt{X} \lhd g$ and $\mathtt{Y} = \mathtt{X'} \lhd g'$, that will be translated one after the other as simple array assignments.

– Another limitation comes from the reduction operation. For instance, the statement in which we have `Y` $= f;\triangleright$ `X` would not be a program if the function f was not injective, since its solution is not unique (the values of the sequence can be in any order). Yet, HPF does not allow such non-deterministic behavior since its reduction operators (`XXX_SCATTER` operators) implies an associative, commutative operator to reduce the sequence of values. We therefore reject non-deterministic data fields definitions from the translation process.

We overcome the previous obstacles with an automatic rewriting which splits equations to produce a program in translatable form. Two kinds of rewrite strategy are applied in this phase: the first rewrite scheme aims at bypassing the syntactic obstacles by replacing data-fields by their definitions, or introducing intermediate definitions when necessary. For example, if we find the two equations `W` = `X /&/ Y` and `Y` = $f_1\triangleright$ `(W /&/` $f_2\triangleright$ `(B /&/ C))`, then the rewrite will output `Z'` = $f_2\triangleright$ `(B /&/ C)` and `Z` = $f_1\triangleright$ `(X /&/ Z')` (introduction of `Z'` and removal of `W`).

The PEI equations also have to be sorted to generate the HPF instructions in the right order (the sequential order). We sort them with the order induced by the data fields definitions: we say that a data field `X` *depends on* a data field `Y` if the definition of `X` contains a least one occurrence of `Y`. From the input data fields, we setup a graph whose nodes represent the data fields and the edges represent the relation "depends on" — an exception is made for recursive definitions of type 6 for which no self-dependence is generated in the graph. We only translate PEI program whose graph has no cycle. With this graph, any data field `X` depending on a data field `Y` will appear after `Y` in the HPF program text.

This scheduling method is simple because it mimics the scheduling of data-parallel programs: in the data-parallel paradigm, the stream of execution is sequential. Here, the parallel variables are considered in their entirety, i.e. all the values of a parallel variable are evaluated in one parallel instruction.

3.2 Translation of translatable programs

In order to keep the translator as simple as possible, we have split the translation process in four phases, as sketched below. We have already discussed the two first phases, which have lead to a translatable program. The next phases associate HPF instructions to these equations.

semantic transf. (rewrite rules)	→	syntactic transf. (pre-processor)	→	array declarations (type-checker)	→	equations translation (translator)

Array declarations The type of a data field is its values and reference domains, and a typechecker has been developed [Vio97] to verify that the equation system does not have contradictory definitions. The verification is done by solving the system with the Omega library primitives. If there are data fields that can match the equation system, the typechecker delivers, for each expression, the candidate solution that has the biggest values domain. We then use this information to produce the Fortran array declarations. For each data field, we then simply compute a bounding box for its biggest values domain. This is achieved by projecting the domain on all its dimensions. For example for a data field whose $dom(v) = \{(i,j) \mid 1{\le}i{\le}n, 0{\le}j{\le}n, i{\ge}j\}$, we build the two projections functions $f_1(i,j) = i$ and $f_2(i,j) = j$ and get the two array sections $f_1(dom(v)) = \{i \mid 1{\le}i{\le}n\}$ and $f_2(dom(v)) = \{j \mid 0{\le}j{\le}n\}$, translated into the array declaration `(1:n,0:n)`.

Translation of equations In the following, we will assume data fields sequences have one element at most (of course we cannot check it at compile time), so that they can be represented as simple arrays elements in HPF.

To give an operational semantics to translatable PEI equations is straightforward for simple geometrical or functional operations, and no extra-restrictions are needed to associate an HPF text with the two first kinds of translatable equations.

- To the **Geometrical** operation of type 1 is associated :

  $$\texttt{forall}\ (z \mid z \in dom(g))\ \texttt{Y}(z)\ \texttt{=}\ \texttt{X}(g(z))$$

  Note that the condition $img(g) \subseteq dom(v_{\texttt{X}})$ from the geometrical operation definition insures that indices $g(z)$ are valid for the array.
  The translation is actually complicated by the following HPF restriction: no index in a forall statement can be an expression of another index [KLS+94]. Therefore, to scan non rectangular areas, we have to set up a loop nest of foralls for which any index appearing in an inner forall only depends on indices appearing in outer foralls.

- For the **Functional** operation, the forall iteration space is the intersection of the domain of values of the computed data fields. On that domain, the superimposition gives sequences with n elements and the translator requires f to also have n arguments. Furthermore, the computations take place only where the predicate P of f is true. To the equation of type 2 is thus associated:

  $$\begin{array}{l}\texttt{forall}\ (z \mid z \in dom(v_{\texttt{X}_1}) \cap \ldots \cap dom(v_{\texttt{X}_n}), P(v_{\texttt{X}1}(z), \ldots, v_{\texttt{X}n}(z))) \\ \quad \texttt{Y}(z)\ \texttt{=}\ f\texttt{(X}_1(z)\texttt{,}\ldots\texttt{,X}_p(z)\texttt{)}\end{array}$$

The next two types of equations bring information on the placement of data fields. It shows that a PEI program contains all the information to arrange data on a processor grid.

- **Datafield property** An equation of the form $\texttt{X} = h{::}\,\texttt{X}$ says that the application of function h on the values and reference domains of `X` does not change `X`. This type of equation is used to constrain a data field: hence, from the change of basis definition, we must verify the equalities : $v_{\mathtt{X}} = v_{\mathtt{X}} \circ h^{-1}$ and $\sigma_{\mathtt{X}} = \sigma_{\mathtt{X}} \circ h^{-1}$. From the equation, the type-checker can deduce the biggest size of `X` since values of `X` must be in $dom(\sigma_{\mathtt{X}}) \subseteq dom(h)$. This equation is not translated but only used to produce array declarations.

- **Change of basis operation**:
  An equation of the form $\mathtt{Y} = h{::}\,\mathtt{X}$ denotes that any value mapped at index point z in `Y` is also mapped at index point $h(z)$ in `X`. Therefore, this type of equation indicates how data fields are placed on the discrete geometrical domain. In our translation, only one HPF template represents the discrete geometrical domain, and its dimensions are computed as a bounding box of all arrays.
  We place the arrays corresponding to the data fields on the template using the HPF `align` directive. Note that h must be an affine function to be translated as an alignment directive. For the previous equation, the following alignments are generated: `align X(`z`) with T(`z`)`
  `align Y(`z`) with T(`$h(z)$`)`
  Last, we generate instructions to copy the values of the first computed array into the new one so that the equality specified by the PEI equation is respected (the copy does not generate any communication since data fields are aligned).
  Let us illustrate the process with the alignment of two arrays such that one array is aligned one line up the other. Consider the following equations where `X` is an input data field:

  $$\begin{array}{lll} \mathtt{X} = matrix{::}\,\mathtt{X} & \text{with} & matrix(i,j) = (1{\le}i{\le}n, 1{\le}j{\le}n).(i,j) \\ \mathtt{Y} = shift{::}\,\mathtt{X} & & shift(i,j) \;\;\, = (1{\le}i{\le}n, 1{\le}j{\le}n).(i{+}1,j) \end{array}$$

  From the first equation the typechecker deduces that `X` values and reference domains are included in $dom(matrix^{-1}) = \{(i,j) \mid 1{\le}i,j{\le}n\}$. The second equation constrains $dom(v_{\mathtt{Y}})$ and $dom(\sigma_{\mathtt{Y}})$: for the value function for instance, $v_{\mathtt{Y}} = v_{\mathtt{X}} \circ shift^{-1}$ which implies $dom(v_{\mathtt{Y}}) \subseteq \{(i,j) \mid 2{\le}i{\le}n{+}1, 1{\le}j{\le}n\}$. Hence, the declarations are :

```
type X(1:n,1:n), type Y(2:n+1,1:n)
template T(n+1,n)
align X(i,j) with T(i,j)
align Y(i,j) with T(i+1,j)
forall (i=1:n,j=1:n) Y(i+1,j) = X(i,j)
```

- **Geometrical reduction**:
  The `XXX_SCATTER` primitives of HPF are general enough to express PEI geometrical reductions combined with a functional operation that involves an associative, commutative operator $\otimes$. The HPF primitive indicates how values are to be gathered through indirection arrays.
  For data fields mapped onto $\mathbb{Z}^m$, we need to build m indirection arrays, let us say x_1 to x_m. We also need a function, called *index*, to select the kth element

of a m-uple $(i_1, \ldots, i_m)$. We can then build the arrays $x_k, \forall k \in [1 \ldots m]$ and fill them: for every element $(i_1, \ldots, i_m)$ such that it belongs to $dom(v_{\mathtt{X}})$, an array element $x_k(i_1, \ldots, i_m)$ takes the value $index(g(i_1, \ldots, i_m), k)$.
Thus, we have a mapping from the set of the tuples $(i_1, \ldots, i_m)$ to the set of the elements $(x_1(i_1, \ldots, i_m), \ldots, x_m(i_1, \ldots, i_m))$, as specified in [HPF93], p.89. If $e_\otimes$ is the neutral element of $\otimes$, the translation of the previous equation is $\mathtt{Y} = \otimes\mathtt{\_SCATTER(X,}e_\otimes\mathtt{,}x_1\mathtt{,}\ldots\mathtt{,}x_m\mathtt{)}$. For instance, the translation of the following equation:

$$\begin{array}{l} \mathtt{Y} = sum \triangleright\ (red; \triangleright \mathtt{X}) \\ red(i,j) = (1 \leq i \leq 3, 1 \leq j \leq 3).(i,i) \\ sum(id\ \sharp(a;b)) = (a + sum(b)) \end{array} \quad \text{is} \quad \mathtt{Y = SUM\_SCATTER(X,0,} \begin{bmatrix} 1\ 2\ 3 \\ 1\ 2\ 3 \\ 1\ 2\ 3 \end{bmatrix}, \begin{bmatrix} 1\ 2\ 3 \\ 1\ 2\ 3 \\ 1\ 2\ 3 \end{bmatrix})$$

An improvement could be the use of simpler primitives such as `XXX_SUFFIX` or `XXX_PREFIX` when we detect reductions along a canonical axis of the array.

– **Recursive definition**
Consider the recursive definition of type 6. Finding the solution `Y` of such an equation consists in finding a fix point for the equation. Some of our work is currently concerned with a general characterization of which recursive definitions accept a solution, following the framework of the Scott's theory [GS90]. It turns out that data fields can be seen as complete partial ordered sets if we impose the following restriction : the sequences of a data field must all be of the same length. With respect to this restriction, we have found a data fields ordering such that operations are continuous, and we can apply the fix-point theorem: we are guaranteed to have a least fix-point as solution for any recursive definition.
From an operational point of view, we also know how many recursions are needed to compute the solution since we require the equation to be schedulable. We restrict the translation to mono-dimensional schedulings. Provided the time axis, say t, the projection of the domain $dom(v_{\mathtt{Y}})$ on t gives a segment representing a super-set of the time steps (in the sense that some points in the segment may not be real instants of computation). The scanning of these time steps is translated as an outer `do` sequential loop. We call t_{min} and t_{max} the extrema of the segment. The cross-product of the others axes is the spatial domain, let us say S. The scanning of S, is obtained by generating parallel loops (a `forall` loop nest) for the domain $dom(v_{\mathtt{Y}})$ in which the variable t is considered as a parameter. Hence the loops scanning S depend on t and avoid unnecessary computations at possible dummy instants (see for example `Gauss2` in section 4).
The computation is initialized by copying all values of the data field `X` into `Y` — some values may not be used during the first step of computation but this method is simpler and does not require more time. The generated code is sketched below:

$$\begin{array}{l} \mathtt{forall}\ (z \mid z \in dom(v_{\mathtt{X}}))\ \mathtt{Y}(z) = \mathtt{X}(z) \\ \mathtt{do}\ t = t_{min}, t_{max} \end{array}$$

```
    forall (z | z ∈ S)
      Y(z) = f(X'(z),Y(g_1(z)),...,Y(g_q(z)))
    endforall
enddo
```

In the previous scheme, the data field `X'` is optional because in many cases (for instance in the examples of section 4) `X'` values are not involved in the computation. In the translation process, it is easy to include or exclude `X'` by simply counting the arguments of f: if f has q arguments and q variables `Y` in the right handside of equation, then `X'` does not appear in the HPF computation.

For example, the prefix-sum specification would be translated into the code below. Note that the first instruction has been introduced to obtain a translatable form, and simply enables `A'` to represent `A` $\lhd first$.

```
forall (i=1:1) A'(i) = A(i)
forall (i=1:n) X(i) = A'(i)
do i = 2,n
  X(i) = A'(i) + X(i-1)
enddo
```

4 Example

The next PEI program is the triangularizing part of the Gaussian elimination used to solve the linear problem $Ax = b$, where A is an $(n \times n)$ matrix and b is a vector of size n. The input data fields `A` and `B` are placed on the first face of a cube represented by `X` and `Y`. The triangularization itself is obtained in computing `C`. In the end the upper triangular matrix and solution vector are on the plane $i = k-1$. It is easy to see that the `Pei` program is already in translatable form (`X` and `Y` could be replaced by one data field). In particular, an obvious scheduling for the recursion is along the k axis.

$$\texttt{Gauss1 : (A,B)} \mapsto \texttt{C}$$
$$\begin{cases} \texttt{A} = align1 :: \texttt{X} \\ \texttt{B} = align2 :: \texttt{Y} \\ \texttt{C} = \texttt{X /\&/ Y /\&/ } elim \rhd \texttt{((C } \lhd pre\texttt{) /\&/ (C } \lhd cp\texttt{) /\&/ (C } \lhd lp\texttt{) /\&/ (C } \lhd piv\texttt{))} \end{cases}$$

$$\begin{array}{ll} align1(i,j,k) & = (1{\le}i{\le}n, 1{\le}j{\le}n, k{=}0).(i,j) \\ align2(i,j,k) & = (1{\le}i{\le}n, j{=}n{+}1, k{=}0).(i) \\ pre(i,j,k) & = (k{<}i{\le}n, k{<}j{\le}n{+}1, 0{<}k).(i,j,k{-}1) \\ piv(i,j,k) & = (k{<}i{\le}n, k{<}j{\le}n{+}1, 0{<}k).(k,k,k{-}1) \\ cp(i,j,k) & = (k{<}i{\le}n, k{<}j{\le}n{+}1, 0{<}k).(i,k,k{-}1) \\ lp(i,j,k) & = (k{<}i{\le}n, k{<}j{\le}n{+}1, 0{<}k).(k,j,k{-}1) \\ elim(a;b;c;d) & = (a{-}b * c/d) \end{array}$$

The translation Gauss1 is simple because all dependences in the geometrical operations are along k. Hence the HPF code is very close to the PEI specification. The two first `forall` loops are just parallel assignments to transfer the values from input data to the three dimensional space of computation.

```
      real C(1:n,1:n+1,0:n),X(1:n,1:n,0:0),Y(1:n,n+1:n+1,0:0),
           A(1:n,1:n),B(1:n)
!hpf$  template T(1:n,1:n+1,0:n)
!hpf$  align C(i,j,k) with T(i,j,k)
!hpf$  align A(i,j) with T(i,j,0)
!hpf$  align B(i) with T(i,n+1,0)
       forall (i=1:n,j=1:n) X(i,j,0)=A(i,j)
       forall (i=1:n) Y(i,n+1,0)=B(i)
c recursion initialization
       forall (k=0:0,i=1:n,j=1:n) C(i,j,k) = X(i,j,k)
       forall (k=0:0,i=1:n,j=n+1:n+1) C(i,j,k) = Y(i,j,k)
c recursion
       do k=1,n-1
          forall (i=k+1:n,j=k:n+1)
            C(i,j,k) = C(i,j,k-1)-C(i,k,k-1)*C(k,j,k-1)/C(k,k,k-1)
       enddo
```

Let us now make the assumption that routing primitives cannot broadcast values (e.g. an architectural constraint). In other words, only neighbors array elements can be accessed, and hence a new scheduling and mapping are necessary for the computations to be able to access the pivot element for example. Among the possible new schedulings and mappings are $t(i,j,k) = i{+}j{+}k$ for the scheduling, and $p(i,j,k) = (i,k)$ for the mapping on the processors space (this result comes from a dependences analysis tool). PEI allows to transform the initial specification by applying the corresponding change of basis, namely $h(i,j,k) = (i,k,i{+}j{+}k)$. We obtain the following equivalent specification after the application of some refinement rules:

$$\texttt{Gauss2 : (A,B)} \mapsto \texttt{C}$$
$$\begin{cases} \texttt{A} = align1 :: \texttt{X} \\ \texttt{B} = align2 :: \texttt{Y} \\ \texttt{C} = \texttt{X /\&/ Y /\&/ } elim \triangleright \texttt{((C } \triangleleft pre\texttt{) /\&/ (C } \triangleleft cp\texttt{) /\&/ (C } \triangleleft lp\texttt{) /\&/ (C } \triangleleft piv\texttt{))} \end{cases}$$

$$\begin{aligned}
align1(i,j,k) &= (1{\le}i{\le}n, 1{\le}k{-}i{-}j{\le}n, j{=}0).(i,k{-}i{-}j) \\
align2(i,j,k) &= (1{\le}i{\le}n, k{-}i{-}j{=}n{+}1, j{=}0).(i) \\
pre(i,j,k) &= (j{<}i{\le}n, j{<}k{-}i{-}j{\le}n{+}1, 0{<}j).(i,j{-}1,k{-}1) \\
piv(i,j,k) &= (j{<}i{\le}n, j{<}k{-}i{-}j{\le}n{+}1, 0{<}j).(j,j{-}1,3j{-}1) \\
cp(i,j,k) &= (j{<}i{\le}n, j{<}k{-}i{-}j{\le}n{+}1, 0{<}j).(i,j{-}1,i+2j{-}1) \\
lp(i,j,k) &= (j{<}i{\le}n, j{<}k{-}i{-}j{\le}n{+}1, 0{<}j).(j,j{-}1,j{+}k{-}i{-}1)
\end{aligned}$$

The translation of Gauss2 seems less obvious. The typechecker returns $dom(v_{\texttt{C}})$ $= \{(i,j,k) \mid 0{\le}j{<}i{\le}n, k{\le}1{+}n{+}i{+}j, i{+}2j{<}k\}$. A projection of $dom(v_{\texttt{C}})$ on k returns the set of time steps $\{k \mid 5{\le}k{\le}3n\}$, and hence the `do` loop in the following

HPF program. For the scanning of the spatial axes, we generate `forall` loops to scan the set $\{(i,j) \mid 0 \leq j < i \leq n, k \leq 1+n+i+j, i+2j < k\}$, where k is constant, and therefore we have loop expressions depending on k. We only give here the translation of the recursive definition of `C`. The final results are the values of the array elements $\{(i,k) | 1 \leq i \leq n, 3i-1 \leq k \leq 2i+n\}$.

```
 c recursion initialization
       forall (i=1:n)
         forall (k=i+1:n+i)
             C(i,0,k) = X(i,0,k)
       forall (i=1:n) C(i,0,n+i+1) = Y(i,0,n+i+1)
c recursion
       do k=5,3*n
           forall (i=max(((k-n+1)/2),2):min(n,k-3))
             forall (j=max(k-i-n-1,1):min(i-1,((k-i-1)/2)))
               C(i,j,k) = C(i,j-1,k-1)-C(i,j-1,i+2*j-1)*
      &                       C(j,j-1,j+k-i-1)/C(j,j-1,3*j-1)
             endforall
           endforall
```

5 Conclusion

We have outlined in this paper, some problems and possible solutions to the production of parallel imperative code derived from a high-level language. A prototype of translator has been developed on this basis, which enables to examine the effects of formal transformations on real code. Especially, the change of basis operation, which appears in the specification itself and has a strong influence on data alignments.

Many problems still have to be studied to broaden the class of translatable PEI programs (for example mutual recursive definitions). If we overcome some of the current limitations, we believe PEI can be used as a formal framework to produce safer HPF code, its typechecking system being really tight and the control over transformations well mastered.

References

[AICK93] Corinne Ancourt, François Irigoin, Fabien Coelho, and Ronan Keryell. A linear algebra framework for static hpf code distribution. In *CPC'93*, November 1993.

[Bir87] Richard S. Bird. Introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[Ble95] Guy E. Blelloch. NESL *: A Nested Data-Parallel Language*. Carnegie Mellon University, Pittsburgh, PA 15213, September 1995.

[CiCL91] Marina Chen, Young il Choo, and Jingke Li. *Parallel Functional Languages and Compilers*. ACM Press, 1991.

[Fea92] Paul Feautrier. Some efficient solution to the affine scheduling problem, part ii, multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992.

[GS90] C.A. Gunter and Dana S. Scott. Semantic domains. In Jan Van Leeuwen, editor, *Handbook of Theoritical Computer Science, Formal Models and Semantics*, pages 634–674. Elsevier, 2nd edition, 1990.

[HPF93] HPF Forum. *High Performance Fortran Language Specification*, 1.0 edition, May 3 1993.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, TatianaShpeisman, and David Wonnacott. *The Omega Library - Version 1.00*, April 1996. Interface Guide.

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

[MSA+85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and iterations in a single assignement language: Reference Manual version 1.2*. Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Ski93] David B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *NATO ARW "Software for Parallel Computation"*. Springer-Verlag, 1993.

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

[VGP97] Eric Violard, Stéphane Genaud, and Guy-René Perrin. Refinement of data-parallel programs. In R. Bird and L. Meertens, editors, *Proceedings of the IFIP Working Conference on Algorithmic Language and Calculi*, pages 107–131. Chapman & Hall, February 1997.

[Vio97] Eric Violard. Typechecking of PEI programs. In *LNCS 1300, Proceedings of EUROPAR'97*, pages 521–529. Springer-Verlag, August 1997.