Asynchronous Parallel Programming in PEI

E. Violard

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

Abstract. This paper presents a transformational approach for the derivation of asynchronous parallel programs. Transformation rules are based on a theory, called PEI. This theory includes the definitions of problems, programs and transformation rules. It is founded on the simple mathematical concepts of multiset and of an equivalence between their representations as data fields. Program transformations are founded on this equivalence and defined from a refinement relation. This paper is illustrated by the example of the shortest path problem.

Keywords multiset, transformation, parallelism, asynchrony, specification

1 Introduction

Reliable parallel programming techniques are founded on formal derivations of programs. The state of the art shows two complementary philosophies:

- the definition of a stepwise refinement calculus which requires a proof development,
- the definition of transformation rules which apply on successive statements.

If a program F' satisfies any specification that a program F satisfies, F' is called a *refinement* of F [7]. This approach requires to define a logic in order to reason on programs. UNITY [1] is a fundamental contribution in this domain, which allows to express specifications and solutions in the same formalism. It is associated with a non-deterministic computing model which allows to introduce sequentiality only when necessary.

The second approach lies on the definition of formal transformation rules. It supposes a convenient model to express successive statements and these rules. *Affine recurrences* on integral convex domains are an example of such an expression. They are mainly studied for systolic synthesis (see for example [9], [3], etc.). ALPHA [6] and CRYSTAL [2] are the main formalisms using this approach. These languages are founded on a deterministic computing model.

In our sense, parallel program derivation would benefit by extending the transformational approach to asynchrony, in order to reach a class of solutions as general as the one considered by refinement techniques. This supposes to introduce an unifying theory. This theory, called PEI (for Parallel Equations

Interpreter), is founded on the simple mathematical concept of multiset, represented as a *data field*. A program is thus a function on data fields and is denoted as a set of un-oriented equations. The two hand-sides of any equation define two equal data fields. This expression generalizes classical recurrence equations.

Moreover transformations on these equations are founded on a mathematical data fields equivalence referring to the same multiset. This equivalence induces an equivalence of programs which is defined from a refinement relation.

The paper is organized as follows: section 2 presents the theory PEI and its mathematical foundation. The equivalence of programs is defined in section 3. Section 4 is devoted to the transformation rules. The concept of asynchrony is presented in section 5 and applied to the derivation of an asynchronous program in section 6.

2 The theory PEI

PEI is a notation for some semantics domains, where all names refer to mathematical objects and all operations are mathematical operations. It is founded on the simple concepts of multiset and of equivalence between their representations as data fields.

2.1 PEI objects

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 computations 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 $(0 \le i, j \le 1)$ (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*.

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:

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

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.

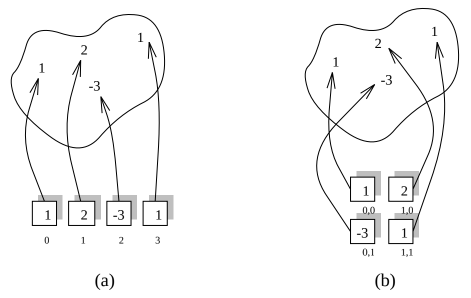

Fig. 1. Two different mappings of a multiset of values

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` $\lhd$ **`shift`**. The function **`shift`** shifts values one place cyclically to the right and is written in Pei as **`shift`**(i) $=$ $(0{\le}i{<}4)\,.\,(i{-}1)$ **`mod`** 4. If the function is not injective the operation expresses a broadcast. For instance `W` $=$ `V` $\lhd$ **`spread`**, where **`spread`**(i) $=$ $(0{\le}i{<}4)\,.\,0$ 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 $\rhd$ and performs an element-wise computation on the data field. For example, `W` $=$ **`inc`** $\rhd$ `V`, where **`inc`**$(a) = a + 3$, defines a data field whose values are computed from the `V` values having the same indices (figure 3).

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 4 shows the result of superimposing `X` and `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

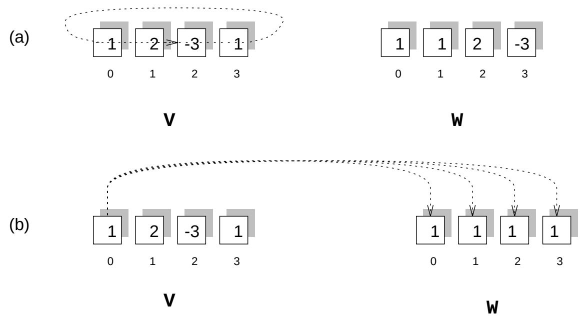

Fig. 2. Geometrical operations: (a) one-to-one relation (b) broadcast

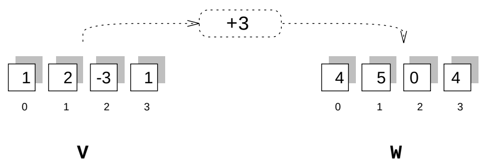

Fig. 3. `W` defined by a functional operation applied on `V`

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 `X`, `H`, `W`, *etc.* for data fields, whereas `f`, `g`, *etc.* denote functions. The general form of a Pei statement includes a header composed of the tuples `I` and `O` of input and output data fields, and an equation set `S`:

$$\begin{array}{l} \mathtt{P} : \mathtt{I} \mapsto \mathtt{O} \\ \{ \\ \mathtt{S} \\ \} \end{array}$$

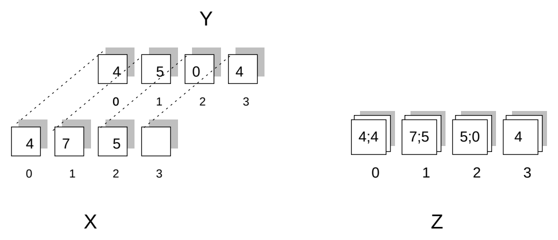

Fig. 4. Superimposition

We will also use the shorthand `P` { `S` } when `I` and `O` do not matter.

To improve the readability, only function names appear in the equations, their definition being reported outside the system. Function definitions are written using the following notation: a function f of domain $dom(f) = \{x \mid P(x)\}$ and image $img(f) = \{f(x) \mid P(x)\}$ which associates the expression $E(x)$ with x, is defined as $\texttt{f}(x) = P(x) \,.\, E(x)$ and we use `#` to separate alternatives in a definition by case.

2.4 Example

We present an example of PEI statement for the shortest path problem. This program is drawn from the UNITY program presented in [1], page 104 and will be used as our running example:

$F_0 : \texttt{W} \mapsto \texttt{D}$
{
$\texttt{onfirst}:: (\texttt{H} \lhd \texttt{first}) = \texttt{W}$
$\texttt{H} \lhd \texttt{next} = \texttt{min} \rhd ((\texttt{H} \lhd \texttt{pre}) \texttt{ /\&/ } (\texttt{H} \lhd \texttt{shift}) \texttt{ /\&/ } (\texttt{H} \lhd \texttt{move}))$
$\texttt{onlast}:: (\texttt{H} \lhd \texttt{last}) = \texttt{D}$
}

$$
\begin{aligned}
\texttt{first}(i,j,k) &= (0 \le i, j \le n-1, k=0) \,.\, (i,j,k) \\
\texttt{last}(i,j,k) &= (0 \le i, j \le n-1, k=n) \,.\, (i,j,k) \\
\texttt{onfirst}(i,j,k) &= (0 \le i, j \le n-1, k=0) \,.\, (i,j) \\
\texttt{onlast}(i,j,k) &= (0 \le i, j \le n-1, k=n) \,.\, (i,j) \\
\texttt{next}(i,j,k) &= (0 \le i, j \le n-1, 0 \le k \le n) \,.\, (i,j,k) \\
\texttt{pre}(i,j,k) &= (0 \le i, j \le n-1, 0 \le k \le n) \,.\, (i,j,k-1) \\
\texttt{shift}(i,j,k) &= (0 \le i, j \le n-1, 0 \le k \le n) \,.\, (i,k,k-1) \\
\texttt{move}(i,j,k) &= (0 \le i, j \le n-1, 0 \le k \le n) \,.\, (k,j,k-1) \\
\texttt{min}(d1; d2; d3) &= (d1 \le d2+d3) \,.\, d1 \texttt{ \#} \\
& (d1 > d2+d3) \,.\, (d2+d3)
\end{aligned}
$$

Considering a point (i, j, k) of `H`, the value in that point is the weight of the shortest path from i to j whose indices of intermediate vertices are smaller than k. We denote it $H(i, j, k)$ in section 6.

2.5 Semantics

The previous section points out that data fields are the central concept in PEI. A data field represents a multiset of values. It is characterized by a *drawing* of the multiset: a drawing associates a geometrical point on $\mathbb{Z}^n$ with each value of a multiset. Formally, assuming the values of the multiset are in V, a drawing of the multiset is a function $v : \mathbb{Z}^n \mapsto V$.

As it has been observed in section 2.1, many data fields can represent the same multiset of values and a bijection links any two of them. It is the reason why, besides its drawing, a bijection characterizes also a data field: it links the data field with a virtual reference domain and can be changed by a change of basis. In fact, the bijection of a data field is not explicit in PEI expressions and it only expresses the *conformity* of objects in such a way that two objects can be combined if and only if one of them conforms to the other. Formally, the bijection is denoted as σ, and it defines an other drawing $(v \circ \sigma^{-1})$ if $dom(v) \subseteq dom(\sigma)$.

$$\begin{tikzcd}
\mathbb{Z}^n \arrow[r, "v"] \arrow[d, "\sigma"'] & V \\
\mathbb{Z}^p \arrow[ur, "v \circ \sigma^{-1}"'] &
\end{tikzcd}$$

Definition 2. A data field is a pair, denoted as $(v : \sigma)$, composed of a drawing v and of a bijection σ such that $dom(v) \subseteq dom(\sigma)$.

This definition founds the formal definition of operations on data fields.

The *superimposition* combines the data fields in conformity. More precisely, we say that a data field conforms with an other one if its bijection is a restriction of the other's bijection. The drawing of the result is the union of the drawings and the operation builds sequences of values on the intersection. As a consequence, we consider all values are sequences built from an associative constructor ";". In the rest, we use classical notation $[\![.]\!]$ to associate syntax with semantics.

Definition 3. Let $[\![E_1]\!]$ and $[\![E_2]\!]$ be two data fields in conformity *i.e.* $\sigma_1 = \sigma_2 \backslash_{dom(\sigma_1)}$. The *superimposition* defines the data field $[\![E_1 \texttt{ /\&/ } E_2]\!]$ as $(w : \sigma_2)$, where $w(z) = v_1(z); v_2(z)$.

The other operations apply a function on a data field $[\![\texttt{X}]\!] = (v : \sigma)$ and form a new data field. We use two other notations on partial functions: composition: the domain of a composed function $[\![\texttt{f o g}]\!]$ is $\{x \in dom([\![\texttt{g}]\!]) \mid [\![\texttt{g}]\!](x) \in dom([\![\texttt{f}]\!])\}$ and inverse: $[\![\texttt{inv(h)}]\!]$ is the inverse of a bijection $[\![\texttt{h}]\!]$.

Definition 4. Let $[\![\texttt{f}]\!]$ be a partial function from V to W such that $img(v) \subseteq dom([\![\texttt{f}]\!])$. Let $[\![\texttt{g}]\!]$ be a partial function from $dom(\sigma)$ to $dom(v)$. Let $[\![\texttt{h}]\!]$ be a bijection from $dom(\sigma)$ to $\mathbb{Z}^p$ such that $dom(v) \subseteq dom([\![\texttt{h}]\!])$.

- The *functional operation* defines the data field $[\![\texttt{f} \triangleright\ \texttt{X}]\!]$ as $([\![\texttt{f}]\!] \circ v : \sigma)$.
- The *geometrical operation* defines the data field $[\![\texttt{X}\ \triangleleft \texttt{g}]\!]$ as $(v \circ [\![\texttt{g}]\!] : \sigma)$.
- The *change of basis* defines the data field $[\![\texttt{h}\text{::}\ \texttt{X}]\!]$ as $(v \circ [\![\texttt{h}]\!]^{-1} : \sigma \circ [\![\texttt{h}]\!]^{-1})$.

Naturally, equation $E_1 = E_2$ semantics where E_1 and E_2 denotes data fields is the data fields equality $[\![E_1]\!] = [\![E_2]\!]$ (that is $(v_1 = v_2) \wedge (\sigma_1 = \sigma_2)$) and the semantics of an equation set is the conjunction of the semantics of each equation.

3 Equivalence and refinement of PEI statement

Our definition of refinement is similar to Knapp's definition, applicable to UNITY programs. Let us consider a statement $\texttt{P} : \texttt{I} \mapsto \texttt{O}\ \{\texttt{S}\}$ and let $\texttt{T}$ denote intermediate data fields in $\texttt{S}$. According to the previous semantics definition, we can state what is specified by a PEI statement: we will say that $\texttt{P}$ specifies the relation, denoted as $\mathcal{R}(\texttt{P})$, between data fields $[\![\texttt{I}]\!]$ and $[\![\texttt{O}]\!]$. The relation is defined by the conjunction of all equations of the statement, the intermediate data fields are existentially quantified in order to only define the relation between $[\![\texttt{I}]\!]$ and $[\![\texttt{O}]\!]$. Formally:

$$\mathcal{R}(\texttt{P}) = \{([\![\texttt{I}]\!], [\![\texttt{O}]\!]) \mid \exists [\![\texttt{T}]\!] \,.\, [\![\texttt{S}]\!]\}$$

Definition 5. (refinement)
Let $\texttt{P}$ and $\texttt{P'}$ be two PEI statements. We say that $\texttt{P}$ is refined by $\texttt{P'}$ and we note $\texttt{P} \sqsubseteq \texttt{P}'$, iff $\mathcal{R}(\texttt{P}') \overset{\equiv}{\subseteq} \mathcal{R}(\texttt{P})$

The symbol $\overset{\equiv}{\subseteq}$ denotes an inclusion relation which takes data field equivalence into account. It is defined as follows:

Definition 6. (inclusion modulo data field equivalence)
Let $\texttt{P}$ and $\texttt{P'}$ be two PEI statements. $\mathcal{R}(\texttt{P}') \overset{\equiv}{\subseteq} \mathcal{R}(\texttt{P})$ iff $\mathcal{A}(\mathcal{R}(\texttt{P}')) \subseteq \mathcal{A}(\mathcal{R}(\texttt{P}))$ where $\mathcal{A}$ is the application which returns the multisets pair represented by a given data fields pair.

Note 7. Of course, if $\mathcal{R}(\texttt{P}') \subseteq \mathcal{R}(\texttt{P})$, then $\mathcal{R}(\texttt{P}') \overset{\equiv}{\subseteq} \mathcal{R}(\texttt{P})$. We will speak about *strong refinement* in that particular case.

The equivalence of two statements is defined from refinement:

Definition 8. (equivalence)
Let $\texttt{P}$ and $\texttt{P'}$ be two PEI statements. We say that $\texttt{P}$ and $\texttt{P'}$ are equivalent and we note $\texttt{P} \equiv \texttt{P}'$, iff $\texttt{P}$ and $\texttt{P}'$ refine each other.

3.1 Relevance of typechecking in PEI

As seen in the previous section, PEI operations are not allowed on any data fields: this means that some phrases are forbidden according to some type constraints. In other words, if the constraints do not hold, then we say that no semantics is associated with such phrases. Note that this is not absolutely necessary and we could decide to associate a specific meaning with such phrases. But this addresses the following crucial question: is a given PEI specification, feasible or not ? It is important to be able to check for feasibility at any step of the refinement process [7]: in PEI, feasibility checking is just typechecking.

We presented in [11] an algorithm that can infer the type of PEI expressions defined as the pair $(dom(v), dom(\sigma))$. Our algorithm presents weak limitations. Based on this algorithm, a typechecker for PEI statements has been implemented. It uses the OMEGA library [5, 8] for evaluating set expressions. This means that the algorithm is decidable if the functions used for geometrical operations or change of basis inside a statement can be coded into an OMEGA relation.

4 Transformation rules

Transformation rules are partitioned in three sets: the first rules are derived from operation properties, the following ones are derived from equation systems and the last ones are equivalence rules.

4.1 Operation properties rules

These rules are founded on algebraic properties of operations. More precisely, the refined statement is obtained by replacing one occurrence of a PEI expression by an other one that can be proved equal from some operation property or from the mathematical structure of the data fields set.

The rules derived from operation properties fall into two categories: some are unoriented and we obtain an equivalent statement by replacing an expression by the other:

$$\begin{array}{rcl}
(\texttt{f o f'}) \rhd E & = & \texttt{f} \rhd (\texttt{f'} \rhd E) \\
(\texttt{h o h'}) :: E & = & \texttt{h} :: (\texttt{h'} :: E) \\
(\texttt{h} :: E_1) \ \texttt{/\&/} \ (\texttt{h} :: E_2) & = & \texttt{h} :: (E_1 \ \texttt{/\&/} \ E_2) \\
(\texttt{f} \rhd E) \lhd \texttt{g} & = & \texttt{f} \rhd (E \lhd \texttt{g}) \\
\texttt{f} \rhd (\texttt{h} :: E) & = & \texttt{h} :: (\texttt{f} \rhd E)
\end{array}$$

Others are oriented because the conditions required for the expression on the right to be well-formed are stronger than the ones required for the expression on the left to be well-formed. If, when substituting the right expression for the left one, the new statement is well-formed, then it strongly refines the old one. Conversely, if we substitute the left expression for the right one, then the new statement is well-formed but it is an abstracted statement: it is equivalent only if both statements have the same type constraints.

$$\begin{array}{rcl}
E \lhd (\mathtt{g\ o\ g'}) & \longrightarrow & (E \lhd \mathtt{g}) \lhd \mathtt{g'} \\
(E_1\ \mathtt{/\&/}\ E_2) \lhd \mathtt{g} & \longrightarrow & (E_1 \lhd \mathtt{g})\ \mathtt{/\&/}\ (E_2 \lhd \mathtt{g}) \\
(\mathtt{h}{::}\, E) \lhd \mathtt{g} & \longrightarrow & \mathtt{h}{::}\,(E \lhd (\mathtt{inv(h)\ o\ g\ o\ h})) \\
\mathtt{h}{::}\,(E \lhd \mathtt{g}) & \longrightarrow & (\mathtt{h}{::}\, E) \lhd (\mathtt{h\ o\ g\ o\ inv(h)})
\end{array}$$

4.2 Equations systems rules

These rules are more general than the preceding ones. They permit to modify not only an expression, but one or more equations of the system which defines a statement. The transformed system is a new one whose solutions set is the same: these rules maintain the equality of statements. Among these rules let us cite classical substitution and the application of a non-singular function to both sides of an equation.

4.3 Equivalence rules

These rules are still more general than the preceding ones. They consist in substituting a data field for an equivalent one. One of these rules is classical. It allows to change representation of a data field. This is its formal definition:

Theorem 9. *Let* P{S} *be a* PEI *statement and* T *a data field name in* S.
If P{S[T/(h:: T)]} *is well-formed, then it refines* P.

where $E[E_1/E_2]$ denote the result of replacing all occurences of E_1 by E_2 in E.

4.4 Operational aspects

Operational aspects define the set of computations associated with any data field definition. This means the definition of an order on the data field elements. This order is a partial one for parallel computations. We define this order, denoted as $\vdash$, for some given partial order $<$ on $\mathbb{Z}^m$, by considering the bijection σ from $\mathbb{Z}^n$ to $\mathbb{Z}^m$ which characterizes a data field.

Definition 10. Let $(v : \sigma)$ be a data field where σ is a bijection from $\mathbb{Z}^n$ to $\mathbb{Z}^m$,

$$\forall z, z' \in dom(v) \,.\, v(z) \vdash v(z') \Leftrightarrow \sigma(z) < \sigma(z')$$

The choice of the order relation $<$ on $\mathbb{Z}^m$ predetermines the operational definition of a program. In fact, the aim of the transformations is to explicit or to build a "nice" bijection σ which introduces the "convenient" order to define a "nice" operational behaviour of the program. These transformations lie on the change of basis operation.

Examples

1. Let us consider a bijection σ from $\mathbb{Z}^n$ to $\mathbb{Z}^m$ such as $\sigma(z) = (p(z), t(z))$, where p is a function from $\mathbb{Z}^n$ to $\mathbb{Z}^{m-1}$ and t a function from $\mathbb{Z}^n$ to $\mathbb{N}$. Note that such a definition is a classical way to define a scheduling and a mapping of the computations on a processor set.
   - Let $<$ be an order on $\mathbb{Z}^m$ such that $\sigma(z) < \sigma(z')$ iff $t(z) < t(z')$ on $\mathbb{N}$. The induced operational definition only defines computations scheduling.
   - Let $<$ be an order on $\mathbb{Z}^m$ such that $\sigma(z) < \sigma(z')$ iff $p(z) = p(z') \wedge t(z) < t(z')$. The induced operational definition defines computations mapping and the scheduling of the processors.
2. **The shortest path problem continued**: an obvious synchronous operational definition of the programm F_0 is obtained by identifying index k with time t. In this solution, indices i, j can be identified with coordinates x, y of the computation point on the processor array. By considering the bijection σ of data field H, this solution consists in defining σ as the bijection from $\mathbb{Z}^3$ to $\mathbb{Z}^3$ such that $\sigma(z) = (p(z), t(z))$, where p is the function from $\mathbb{Z}^3$ to $\mathbb{Z}^2$, defined as $p(i, j, k) = (i, j)$ and where t is the function from $\mathbb{Z}^3$ to $\mathbb{N}$, defined as $t(i, j, k) = k$.

5 About asynchrony

Asynchronous programs are defined from a function denoted as pack for packing function: when applied to a data field X whose drawing domain $dom(v)$ is a subset of $\mathbb{Z}$, it defines a bijection which packs $dom(v)$ in an interval of $\mathbb{N}$, whose length is the cardinality of $dom(v)$. The change of basis applied on pack(X) and X defines a packed data field such that $dom(v) = dom(\sigma)$. Here is the formal definition of the packing function which is generalized to be applied to any data field:

Definition 11. The function pack associates with any data field X of drawing v such that $dom(v) \subseteq \mathbb{Z}^n, n{>}0$, the bijection pack(X) defined on $dom(v)$ which maps $(i_1, \ldots, i_n)$ to $(i_1, \ldots, i_{n-1}, h(i_n))$ where h is the bijection defined on domain $\{i_n \mid (i_1, \ldots, i_n) \in dom(v)\}$ as:

- $img(h) = [0..\text{card}\,(dom(h))[$
- h strictly increases.

The word “asynchronous” characterizes a non-deterministic aspect of the program computing model. To illustrate what is asynchrony, let us consider the following trivial example. Let X be a data field whose drawing domain is a subset of $\mathbb{Z}$. The values of X are ordered by their coordinates in $\mathbb{Z}$. Let us define:

```
Sync : X ↦ Y
{
Y = X  ◁ odd
}
odd(i) = ((i mod 2)=1) . i
```

As said in section 4.4, this program defines the instants where the values of Y are computed, for the bijection σ of Y is the same as the bijection of X. So, this program can be considered as synchronous. Let us consider now the following equivalent program:

```
Async : X ↦ Y
{
Z = X ◁ odd
pack(Y):: Y = pack(Z):: Z
}
odd(i) = ((i mod 2)=1) . i
```

The packed drawings of data fields Y and Z are the same. So, the sequences of values in Y and Z are the same. But, the drawings of Y and Z are different. This means that the computing instants of these two sequences can be different. This program can then be considered as asynchronous and the bijection pack(Y) defines the computation delays in Y.

6 Programs derivation

Programs derivation, by using transformation rules, is illustrated here under with the shortest path problem. We present the first steps to design an asynchronous solution for the problem whose initial statement was previously given.

Step 1 : distribution An asynchronous solution can be obtained by dissociating index k from time t, by convincing that a value in the point (i, j, k) is computed before the computation of a value in the point $(i, j, k+1)$. In order to reach this goal, we apply a transformation which "distributes the index k on each point (i, j, k)" and leads to the following new statement:

```
F₁ : W ↦ D
{
onfirst:: (H ◁ first) = join₀ ▷ W
H ◁ next = min' ▷ ((H ◁ pre) /&/ (H ◁ shift) /&/ (H ◁ move))
onlast:: (H ◁ last) = joinₙ ▷ D
}
first(i, j, k)   = (0≤i, j≤n−1, k=0) . (i, j, k)
last(i, j, k)    = (0≤i, j≤n−1, k=n) . (i, j, k)
onfirst(i, j, k) = (0≤i, j≤n−1, k=0) . (i, j)
onlast(i, j, k)  = (0≤i, j≤n−1, k=n) . (i, j)
next(i, j, k)    = (0≤i, j≤n−1, 0<k≤n) . (i, j, k)
pre(i, j, k)     = (0≤i, j≤n−1, 0<k≤n) . (i, j, k−1)
shift(i, j, k)   = (0≤i, j≤n−1, 0<k≤n) . (i, k, k−1)
move(i, j, k)    = (0≤i, j≤n−1, 0<k≤n) . (k, j, k−1)
```

$$\texttt{min}'((d1,k1);(d2,k2);(d3,k3)) = \begin{array}{l}(d1{\le}d2{+}d3)\,.\,(d1,k1{+}1)\ \texttt{\#} \\ (d1{>}d2{+}d3)\,.\,(d2{+}d3,k1{+}1)\end{array}$$

$\texttt{join}_0(d) = (d, 0)$
$\texttt{join}_n(d) = (d, n)$

This new program is obtained by considering a new intermediate data field `H'` whose any point (i, j, k) is a pair of values: the value of data field `H` in this point and the value k:

$$\texttt{H'} = \underset{K}{(\texttt{/\&/})}\ (\texttt{join}_K \rhd (\texttt{H} \lhd \texttt{select}_K))$$

where: $\texttt{select}_K(i,j,k) = (0{\le}i, j{\le}n{-}1, k{=}K)\,.\,(i,j,k)$ and $\texttt{join}_K(d) = (d, K)$
`H'` can be substituted for `H` in any equation of the program F_0, according to an equation system rule which allows to apply an operation on the two hands of an equation. Let us consider, for example, the following equation in F_0: $\texttt{H} \lhd \texttt{next} = \texttt{min} \rhd ((\texttt{H} \lhd \texttt{pre})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{shift})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{move}))$
Using properties of the superposition, this equation can be rewritten as:

$$\underset{K}{(\texttt{/\&/})}((\texttt{H} \lhd \texttt{next}) \lhd \texttt{select}_K) =$$
$$\underset{K}{(\texttt{/\&/})}((\texttt{min} \rhd ((\texttt{H} \lhd \texttt{pre})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{shift})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{move}))) \lhd \texttt{select}_K)$$

Then, for any K:
$(\texttt{H} \lhd \texttt{next}) \lhd \texttt{select}_K =$
$(\texttt{min} \rhd ((\texttt{H} \lhd \texttt{pre})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{shift})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{move}))) \lhd \texttt{select}_K$
By using the equation system rule, we can apply functional operation $\texttt{join}_K$ on the two hands of this equation:

$\texttt{join}_K \rhd ((\texttt{H} \lhd \texttt{next}) \lhd \texttt{select}_K) = \texttt{join}_K \rhd$
$((\texttt{min} \rhd ((\texttt{H} \lhd \texttt{pre})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{shift})\ \texttt{/\&/}\ (\texttt{H} \lhd \texttt{move}))) \lhd \texttt{select}_K)$
Now, by using properties of operations and by applying an other equation system rule to substitute $(\texttt{H'} \lhd \texttt{select}_K)$ for $(\texttt{join}_K \rhd (\texttt{H} \lhd \texttt{select}_K))$, the following equation comes:

$(\texttt{H'} \lhd \texttt{next}) \lhd \texttt{select}_K =$
$(\texttt{min'} \rhd ((\texttt{H'} \lhd \texttt{pre})\ \texttt{/\&/}\ (\texttt{H'} \lhd \texttt{shift})\ \texttt{/\&/}\ (\texttt{H'} \lhd \texttt{move}))) \lhd \texttt{select}_K$

where: $\texttt{min}'((d1,k1);(d2,k2);(d3,k3)) = \begin{array}{l}(d1{\le}d2{+}d3)\,.\,(d1,k1{+}1)\ \texttt{\#} \\ (d1{>}d2{+}d3)\,.\,(d2{+}d3,k1{+}1)\end{array}$
For this equation is valid for any K and by renaming `H'` as `H` and $\texttt{min}'$ as `min`, we deduce the corresponding equation of the program F_1.

Step 2 : scheduling This step allows to explicit a scheduling of the computations and leads to the following new statement:

F_2 : `W` $\mapsto$ `D`

```
{
onfirst':: (H ◁ first') = join₀ ▷ W
H ◁ next' = min ▷ ((H ◁ pre') /&/ (H ◁ shift') /&/ (H ◁ move'))
onlast':: (H ◁ last') = joinₙ ▷ D
}
```

$\texttt{first}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, r{=}0)\,.\,(i,j,t)$
$\texttt{last}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, r{=}n)\,.\,(i,j,t)$
$\texttt{onfirst}'(i,j,t) = (0{\leq}i,j{\leq}n{-}1, r{=}0)\,.\,(i,j)$
$\texttt{onlast}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, r{=}n)\,.\,(i,j)$
$\texttt{next}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, 0{<}r{\leq}n)\,.\,(i,j,t)$

$\texttt{pre}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, 0{<}r{\leq}n)\,.\,(i,j,k(i,j)^{-1}(r{-}1))$
$\texttt{shift}'(i,j,t) = (0{\leq}i,j{\leq}n{-}1, 0{<}r{\leq}n)\,.\,(i,r,k(i,j)^{-1}(r{-}1))$
$\texttt{move}'(i,j,t) \;= (0{\leq}i,j{\leq}n{-}1, 0{<}r{\leq}n)\,.\,(r,j,k(i,j)^{-1}(r{-}1))$

$$\begin{aligned}\texttt{min}((d1,k1);(d2,k2);(d3,k3)) = \;& (d1{\leq}d2{+}d3)\,.\,(d1,k1{+}1)\ \#\\ & (d1{>}d2{+}d3)\,.\,(d2{+}d3,k1{+}1)\end{aligned}$$

where $r = k(i,j)(t)$

The transformation consists in applying the change of basis `inv(pack(H))` on the data field `H` of the previous program. By definition of the packing function, we can write:

$$\texttt{pack(H)}(i,j,t) = (i,j,k(i,j)(t))$$

where $k(i,j)$ is a bijection defined from a subset of $\mathbb{N}$ to the drawing domain of `H`. First, we use an equation system rule to apply the change of basis operation `inv(pack(H))` on the two hands of any equation of the program F_1. Then, we use operation properties in order to apply this change of basis on `H`. Last, we rename `inv(pack(H))`:: `H` as `H` and use the equivalence rule, which allows to change of representation in order to obtain the equivalent program F_2.

Step 3 : domain expanding The following statement is obtained by considering the function $\overline{k}(i,j)$ defined as:

$$\begin{aligned}\overline{k}(i,j) :\ & \mathbb{N} \to [0..n]\\ & t \ \mapsto k(i,j)(t) \text{ if } t \in \mathit{dom}(k(i,j))\\ & \qquad \overline{k}(i,j)(t{-}1) \text{ else}\end{aligned}$$

This transformation allows to expand the domain of data field `H` on $\{(i,j,t) \mid 0{\leq}i,j{\leq}n{-}1 \wedge t{\geq}0\}$. It is an *uniformization* of the function $k(i,j)$.

```
$F_3$ : W ↦ D
{
onfirst:: (H ◁ first) = join$_0$ ▷ W
H ◁ next = min ▷ ((H ◁ pre) /&/ (H ◁ shift) /&/ (H ◁ move))
onlast:: (H ◁ last) = join$_n$ ▷ D
}
first$(i,j,t)$   = $(0{\le}i,j{\le}n{-}1, t{=}0)\,.\,(i,j,t)$
last$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{=}T)\,.\,(i,j,t)$
onfirst$(i,j,t)$ = $(0{\le}i,j{\le}n{-}1, t{=}0)\,.\,(i,j)$
onlast$(i,j,t)$  = $(0{\le}i,j{\le}n{-}1, t{=}T)\,.\,(i,j)$
next$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,j,t)$
pre$(i,j,t)$     = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,j,t{-}1)$
shift$(i,j,t)$   = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,r,t{-}1)$
move$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(r,j,t{-}1)$
min$((d1,k1);(d2,k2);(d3,k3))$ =
    $(k2{=}k1 \wedge k3{=}k1 \wedge d1{\le}d2{+}d3)\,.\,(d1,k1{+}1)$ #
    $(k2{=}k1 \wedge k3{=}k1 \wedge d1{>}d2{+}d3)\,.\,(d2{+}d3,k1{+}1)$ #
    $(k2{\ne}k1 \vee k3{\ne}k1)\,.\,(d1,k1)$
where $r = \overline{k}(i,j)(t)$ and $T = k(i,j)^{-1}(n)$.
```

Step 4 : abstraction From the preceding program, we make permissible to have one process race ahead of another by using property $H(i,j,k{+}m){\le}H(i,j,k)$, $m{>}0$. This leads to the following equivalent program that is presented in [1], p. 109:

```
$F_4$ : W ↦ D
{
onfirst:: (H ◁ first) = join$_0$ ▷ W
H ◁ next = min' ▷ ((H ◁ pre) /&/ (H ◁ shift) /&/ (H ◁ move))
onlast:: (H ◁ last) = join$_n$ ▷ D
}
first$(i,j,t)$   = $(0{\le}i,j{\le}n{-}1, t{=}0)\,.\,(i,j,t)$
last$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{=}T)\,.\,(i,j,t)$
onfirst$(i,j,t)$ = $(0{\le}i,j{\le}n{-}1, t{=}0)\,.\,(i,j)$
onlast$(i,j,t)$  = $(0{\le}i,j{\le}n{-}1, t{=}T)\,.\,(i,j)$
next$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,j,t)$
pre$(i,j,t)$     = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,j,t{-}1)$
shift$(i,j,t)$   = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(i,r,t{-}1)$
move$(i,j,t)$    = $(0{\le}i,j{\le}n{-}1, t{>}0)\,.\,(r,j,t{-}1)$
min'$((d1,k1);(d2,k2);(d3,k3))$ =
    $(k2{\ge}k1 \wedge k3{\ge}k1 \wedge d1{\le}d2{+}d3)\,.\,(d1,k1{+}1)$ #
    $(k2{\ge}k1 \wedge k3{\ge}k1 \wedge d1{>}d2{+}d3)\,.\,(d2{+}d3,k1{+}1)$ #
    $(k2{<}k1 \vee k3{<}k1)\,.\,(d1,k1)$
```

7 Conclusion

The theory we have presented in this paper is founded on the simple mathematical concepts of multiset and of an equivalence between their representations as data fields. A detailed presentation can be found in [10, 4]. Program transformations are founded on this equivalence and defined from a refinement relation. Due to the unifying aspect of this theory, solutions that can be reached by these transformations are relevant to various synchronous or asynchronous computing models. The point we have focused in this paper concerns asynchronous computations and was illustrated by the algebraic path problem.

References

1. K.M. Chandy and J. Misra. *Parallel Program Design :* A foundation. Addison Wesley, 1988.
2. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. chapter 7.
3. P. Clauss, C. Mongenet, and G.-R. Perrin. Synthesis of size-optimal toroidal arrays for the algebraic path problem : A new contribution. *Parallel Computing*, 18:185–194, 1992.
4. S. Genaud, E. Violard, and G.-R. Perrin. Transformation techniques in PEI. In *EUROPAR'95*, Stockholm, Sweden, August 1995.
5. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library - Version 1.00*, April 1996. Interface Guide.
6. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, Université de Rennes 1, Décembre 1989.
7. C. Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.
8. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.
9. P. Quinton. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989.
10. E. Violard. *Une théorie unificatrice pour la construction de programmes parallèles par des techniques de transformations.* PhD thesis, Université de Franche-Comté, Octobre 1992.
11. E. Violard. Typechecking of PEI expressions. In *LNCS, EUROPAR'97*, volume 1300, pages 521–529, Passau, 1997.

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