Typechecking of PEI expressions

Eric Violard

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

1 Introduction

PEI is a formal framework for reasoning on programs. It was introduced [15, 16] to express and transform parallel programs [3]. It defines a language used to express statements, called PEI programs, and a refinement calculus to transform such programs.

PEI was clearly born of classical methods for the synthesis of systolic arrays [13, 10]. In this approach, a statement was a set of recurrence equations which are an abstract form for single assignment loop nests. Indices are the coordinates of points over a geometrical space and scan the *computation domain*. In this context, program transformations can be considered as pure geometrical transformations: they mainly consist in remodeling the computation domain by using isometries or partitions, in order to define a computation ordering in preserving the data *dependencies* within the computation domain. So, a parallel program results of a change of basis from the geometrical space into a new *space-time domain*: in this representation all dependencies are directed along the time axis, and successive computation fronts can be defined [7, 6].

These methods and all developments on automatic parallelization of DO loops apply when drastic constraints, such as linear constraints, are satisfied. The formal approach of PEI aims at overcoming these limitations and PEI proposes program transformations in a more general point of view. However, geometry and change of basis are features of main importance in PEI: objects expressed in the language are founded on these concepts and their type depends on them. This supposes to define a precise notion of type as definition domain of partial functions on $\mathbb{Z}^n$.

In this paper, we recall the main features of the language PEI and focus on the notion of *data field* in this language. Section 3 defines the type of such objects and section 4 presents an algorithm to infer their type. Last, section 5 is a survey of the implementation of this typechecker, by using the OMEGA library [5].

2 The language PEI

2.1 An introduction to PEI Programming

As a general point of view, let us consider that a problem can be specified as a relation between *multisets* of value items, roughly speaking its *inputs* and *outputs*. Of course, programming may imply to put these items 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*.

For example the multiset of integral items $\{1, -2, 3, 1\}$ can be expressed as a data field, say `A`, each element of `A` being recognized by an index in $\mathbb{Z}$ (e.g. from 0 to 3). Of course this multiset may be expressed as an other data field, say `M`, which places the items on points $(i, j) \in \mathbb{Z}^2$ such that $0 \le i, j < 2$. These two data fields `A` and `M` are considered equivalent in PEI since they express the same multiset. Formally, there exists a bijection from the first arrangement onto the second one, e.g. $\sigma(i) = (i \; mod \; 2, i \; div \; 2)$. This is denoted by the equation:

$$\mathtt{M} = \mathtt{align} :: \mathtt{A}$$

where $$\mathtt{align} = \lambda(i) \; | \; (0 \le i \le 3) \, . \, (i \; mod \; 2, i \; div \; 2)$$

Any PEI program is composed of such unoriented equations, each of them defining an expression of a *data field*. On the example, `M` and `align :: A` have the same set of value items, placed in the same fashion in the same reference domain.

A set of p input and q output data fields and a set of equations define a program in PEI if and only if the set of equations has at most one solution, i.e. from any set of p input valued data fields there exists at most one set of q resulting output valued data fields. Here is a classical example of prefix-sum of n numbers in PEI:

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

$$\begin{array}{l}
\mathtt{PrefixSum} \; : \; \mathtt{A} \mapsto \mathtt{X} \\
\{ \\
\mathtt{A} = \mathtt{dom} :: \mathtt{A} \\
\mathtt{X} = \mathtt{add} \rhd (\mathtt{A} \; /\&/ \, (\mathtt{X} \; \lhd \; \mathtt{pre})) \\
\} \\
\mathtt{dom} = \lambda(i) \; | \; (1 \le i \le n, n > 1) \, . \, (i) \\
\mathtt{pre} = \lambda(i) \; | \; (1 < i \le n) \, . \, (i-1) \\
\mathtt{add} = \mathtt{id} \; \# \; \lambda(a;b) \, . \, (a+b)
\end{array}$$

- the first equation defines the mapping of the input data field `A`: its values are placed onto a line segment $[1..n]$ of $\mathbb{Z}$,
- fig. 1 intuitively shows that data field `X` is a solution of the second equation: its value items are the prefix computations of the sums of the value items in `A`. The first expression $(\mathtt{X} \lhd \mathtt{pre})$ defines a data field resulting from `X` by shifting its values from left to right. They are then composed with the values of `A` in the expression $(\mathtt{A} \; /\&/ \, (\mathtt{X} \lhd \mathtt{pre}))$ and added one to another.

2.2 Syntactic issues

The previous example points out that data fields are the central concept in PEI. Besides data fields, partial functions define operations on these objects. In the following `A`, `B`, `X`, etc. denote data fields, whereas `f`, `g`, etc. denote functions and `h`

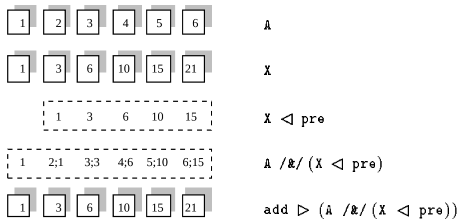

Fig. 1. `X` is a solution of the second equation

is used for bijections. The PEI notation for partial functions is derived from the lambda-calculus: any function $\mathtt{f}$ of domain $dom(\mathtt{f}) = \{x \mid P(x)\}$ and whose image is $img(\mathtt{f}) = \{\mathtt{f}(x) \mid P(x)\}$ is denoted as $\lambda x \mid P(x) \,.\, \mathtt{f}(x)$. Moreover, a function $\mathtt{f}$ defined on disjunctive sub-domains is denoted as a partition $\mathtt{f}_1 \,\#\, \mathtt{f}_2$ of functions, and the domain of a composed function $\mathtt{f} \circ \mathtt{g}$ is $\{x \in dom(\mathtt{g}) \mid \mathtt{g}(x) \in dom(\mathtt{f})\}$. Last, $inv(\mathtt{h})$ denotes the inverse of a bijection $\mathtt{h}$.

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. We use two operators on sequences: an associative constructor denoted as ";" and the function `id` which is the identity on sequences of one element. Four external operations are defined and associate a data field with a function: the *functional operation*, denoted as "$\triangleright$", the *geometrical operation*, denoted as "$\triangleleft$", the *geometrical reduction*, denoted as "$;\triangleright$" and the *change of basis*, denoted as "::".

2.3 Formal definition of data fields

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 $(v \circ inv(\sigma))$ if $dom(v) \subseteq dom(\sigma)$.

$$
\begin{tikzcd}
\mathbb{Z}^n \arrow[r, "v"] \arrow[d, shift right, "\sigma"'] & V \\
\mathbb{Z}^p \arrow[u, dashed, shift right] \arrow[ur, "v \circ inv(\sigma)"'] &
\end{tikzcd}
$$

Definition 1. A data field is a pair, denoted as $(v : \sigma)$, composed of a drawing v and of a bijection σ such that $dom(v) \subseteq dom(\sigma)$.

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.

Definition 2. Let $\mathtt{X}_1 = (v_1 : \sigma_1)$ and $\mathtt{X}_2 = (v_2 : \sigma_2)$ be two data fields such that $\sigma_1 = \sigma_2 \backslash_{dom(\sigma_1)}$. The *superimposition* defines the data field $\mathtt{X}_1 \mathtt{/\&/} \mathtt{X}_2$ as $(v : \sigma_2)$, where v is defined as:

$$\begin{aligned} v(z) &= v_1(z) \text{ if } z \in dom(v_1) - dom(v_2) \\ &= v_2(z) \text{ if } z \in dom(v_2) - dom(v_1) \\ &= v_1(z); v_2(z) \text{ if } z \in dom(v_1) \cap dom(v_2) \end{aligned}$$

The following operations apply a function on a data field $\mathtt{X} = (v : \sigma)$ and form a new data field:

Definition 3. Let $\mathtt{f}$ be a partial function from V to W such that $img(v) \subseteq dom(\mathtt{f})$. The *functional operation* defines the data field $\mathtt{f} \rhd \mathtt{X}$ (whose values are in W) as $(\mathtt{f} \circ v : \sigma)$.

Definition 4. Let $\mathtt{g}$ be a partial function from $dom(\sigma)$ to $dom(v)$. The *geometrical operation* defines the data field $\mathtt{X} \lhd \mathtt{g}$ as $(v \circ \mathtt{g} : \sigma)$.

Conversely, let $\mathtt{g}$ be a partial function from $dom(v)$ to $dom(\sigma)$. The *geometrical reduction* defines the data field $\mathtt{g} \,;\!\rhd \mathtt{X}$ as some data field $(w : \sigma)$ such that $dom(w) = \mathtt{g}(dom(v))$ and $w(z)$ is the sequence of the value $v(y)$, y such that $\mathtt{g}(y) = z$. As a consequence, the next property links reductions and geometrical operations:

Property 5. $\mathtt{g} \,;\!\rhd \mathtt{X} = \mathtt{X} \lhd inv(\mathtt{g})$*, for all data field* $\mathtt{X}$ *iff* $\mathtt{g}$ *is bijective.*

Definition 6. Let $\mathtt{h}$ be a bijection from $dom(\sigma)$ to $\mathbb{Z}^p$ such that $dom(v) \subseteq dom(\mathtt{h})$. The *change of basis* defines the data field $\mathtt{h} :: \mathtt{X}$ as $(v \circ inv(\mathtt{h}) : \sigma \circ inv(\mathtt{h}))$.

As mentioned earlier, the conformity of objects or some assumptions on domains have to be checked in order to apply operations on objects, or more generally to transform programs. PEI was orginally defined to describe and reason on parallel programs and their implementation. It includes a refinement calculus [15, 17] that makes possible to transform statements by associating algebraic laws and symbolic evaluation of functions with the classical geometrical foundations in parallel programming or parallel compiling. In that context, typechecking is a particularly relevant issue. This point is detailed in the next section.

2.4 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 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 [11]: in PEI, feasibility checking is just typechecking.

Moreover, refinement rules are founded on algebraic properties of operations. More precisely, a refined statement is obtained by replacing one occurrence of a PEI expression by an other one in such a way that the conditions required for the new expression to be well-formed are stronger than the ones required for the old expression. In fact, refinement calculus in PEI is based on types: it preserves feasibility. Let us explain this last point with an example.

Example 2. Routing composition laws

Here is a classical law of refinement in PEI:

$$\mathtt{X} \lhd \mathtt{f1} \circ \mathtt{f2} \longrightarrow (\mathtt{X} \lhd \mathtt{f1}) \lhd \mathtt{f2}$$

This law says that the expression on the left can be replaced by the expression on the right without condition. Proof of this law directly lies on types: it demonstrates that if the expression on the right is well-formed, then the expression on the left will necessarily be well-formed and equal. Here is this proof:

> *Proof.* Let $\mathtt{X} = (v : \sigma)$ be a data field. Let us assume that the expression on the right is well-formed. Then, the subexpression $\mathtt{X} \lhd \mathtt{f1}$ is well-formed. By definition of the geometrical operation, this implies that $dom(\mathtt{f1}) \subseteq dom(\sigma)$ and $img(\mathtt{f1}) \subseteq dom(v)$. Moreover, since the whole expression is well-formed we have $dom(\mathtt{f2}) \subseteq dom(\sigma)$ (and $img(\mathtt{f2}) \subseteq dom(v \circ \mathtt{f1}) = dom(\mathtt{f1})$). Since $dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\mathtt{f2})$ and $img(\mathtt{f1} \circ \mathtt{f2}) \subseteq img(\mathtt{f1})$ always hold, we eventually have $dom(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(\sigma)$ and $img(\mathtt{f1} \circ \mathtt{f2}) \subseteq dom(v)$ which means that the expression $\mathtt{X} \lhd \mathtt{f1} \circ \mathtt{f2}$ is well-formed. Both expressions are equal to $(v \circ \mathtt{f1} \circ \mathtt{f2} : \sigma)$.

Here is the converse law that is also established:

$$(\mathtt{X} \lhd \mathtt{f1}) \lhd \mathtt{f2} \longrightarrow \mathtt{X} \lhd \mathtt{f1} \circ \mathtt{f2}$$
$$\text{with } img(\mathtt{f2}) \subseteq dom(\mathtt{f1}) \subseteq dom(\sigma)$$

It says that the expression on the left can be replaced by the expression on the right provided the condition $img(\mathtt{f2}) \subseteq dom(\mathtt{f1}) \subseteq dom(\sigma)$ can be infered from the new statement. In fact, it is an equivalence rule: the old and new statement are equivalent (or refine each other). The proof for this rule is very similar to the previous one. As seen previously, the condition is a type condition for expression $\mathtt{X} \lhd \mathtt{f1}$ to be well-formed. Type checking is needed to apply this rule, the type of $\mathtt{X}$ in the new statement has to be infered and has to match the type condition.

This shows that typing objects in PEI is an important issue for reasonning on PEI statements in a practical manner.

2.5 Related works

In other works in the area of program transformations, such as ALPHA [9] or CRYSTAL [2], types are not infered: they are declared and defined as polyedral domains in $\mathbb{Z}^n$, in ALPHA for example. Typing variables consists in determining their exact domain and important restrictions on domains are made to insure operations can be applied. PEI is an attempt to overcome these limitations.

In the area of functional languages, Björn Lisper describes a formalism for data parallel programming [4, 8] where aggregate data are also called data fields (not to be confused with data fields defined here), which can be seen as the first part v of data fields in our sense. In this approach, "extend analysis" refers to a static analysis, applicable to data field expressions. "Extend" stands for index sets: extend analysis tries to find, for a data field f the points x where calls will be made to $f(x)$. The result of a successful analysis can be used to preschedule calls to f. This sort of analysis is strongly related to typechecking in our formalism.

3 Typing objects in PEI

As seen before, we are particularly interested in checking the type of the domains of a data field, but not the type of values in the data field. For sake of simplicity, the following definition does not take the type of values into account.

Definition 7. Let $(v : \sigma)$ be a data field, the pair $(dom(v), dom(\sigma))$ is called the *type* of this data field. Domains $dom(v)$ and $dom(\sigma)$ are called respectively the *value domain* and the *reference domain* of the data field.

We define a typechecker which aims to infer the type of any data field in a statement. It may happen that several types can be associated with a data field depending on some parameters. Then, the checker characterizes the set of all possible types.

3.1 Type system

Typechecking consists in solving a *type system*. The system contains all conditions on domains which ensure that every operation on data fields occurring in the program can be applied. Unknowns of the system denote either a value domain, or a reference domain of a data field `X`. We write them `val X` and `ref X` respectively. A type system is a set of equalities and inclusions. Each of them connects two domain expressions:

$$\begin{cases} \vdots \\ \langle D \rangle = \langle D \rangle \\ \vdots \\ \langle D \rangle \subseteq \langle D \rangle \\ \vdots \end{cases}$$

A domain expression is either a variable, or the union of two domains, or the image of a domain by a bijection, or a constant domain. Its syntax is defined in BNF-like notation as:

<D> ::= <V> | <D> ∪ <D> | *h*(<D>) | <cst>

3.2 Structural definition of types

Writing inference rules is an elegant way to define the type of expressions in a language [1]. Classically, formulae have the form:

$$\vdash \mathtt{E} : t$$

where $\mathtt{E}$ is an expression and t its type. We use this notation to define types in PEI. Let D, D_1 and D_2 be value domains and Δ, Δ_1 and Δ_2 be reference domains.

Superimposition

$$\frac{\vdash \mathtt{E}_1 : D_1 \times \Delta_1 \ \& \ \vdash \mathtt{E}_2 : D_2 \times \Delta_2}{\vdash \mathtt{E}_1\mathtt{/\&/}\mathtt{E}_2 : (D_1 \cup D_2) \times \Delta_2} \ \text{if} \ \Delta_1 \subseteq \Delta_2$$

Functional operation

$$\frac{\vdash \mathtt{E} : t}{\vdash \mathtt{f} \rhd \mathtt{E} : t}$$

Geometrical operation

$$\frac{\vdash \mathtt{E} : D \times \Delta}{\vdash \mathtt{E} \lhd \mathtt{g} : dom(\mathtt{g}) \times \Delta} \ \text{if} \ \begin{cases} img(\mathtt{g}) \subseteq D \\ dom(\mathtt{g}) \subseteq \Delta \end{cases}$$

Geometrical reduction

$$\frac{\vdash \mathtt{E} : D \times \Delta}{\vdash \mathtt{g} \ ;\!\rhd \mathtt{E} : img(\mathtt{g}) \times \Delta} \ \text{if} \ \begin{cases} dom(\mathtt{g}) \subseteq D \\ img(\mathtt{g}) \subseteq \Delta \end{cases}$$

Change of basis

$$\frac{\vdash \mathtt{E} : D \times \Delta}{\vdash \mathtt{h} :: \mathtt{E} : \mathtt{h}(D) \times img(\mathtt{h})} \ \text{if} \ \begin{cases} D \subseteq dom(\mathtt{h}) \\ dom(\mathtt{h}) \subseteq \Delta \end{cases}$$

Equation

$$\frac{\vdash \mathtt{E}_1 : t \ \& \ \vdash \mathtt{E}_2 : t}{\vdash \mathtt{E}_1 = \mathtt{E}_2}$$

where the conclusion of this rule means that the equation $\mathtt{E}_1 = \mathtt{E}_2$ is well-formed.

These rules express the conditions we have associated with the definition of PEI operations and define well-formed phrases in PEI. As said in 2.1, PEI equations are unoriented: they cannot be considered as definitions as a general rule. For that reason, the inference system associated with these rules cannot be carried out: these rules only enable to formally define the type of a data field and to build the type system associated with a PEI statement.

Moreover, note that for any data field $\mathtt{X}$, the assertion $\mathtt{val\ X} \subseteq \mathtt{ref\ X}$, deduced from data field definition, is implicitly added to this system.

Example 3. Data replication

Let us consider the following PEI program:

$\mathtt{Broadcast} : \mathtt{B} \mapsto \mathtt{A}$

$\{$

$\mathtt{project} :: \mathtt{T} = \mathtt{B}$

$\mathtt{A} = \mathtt{T} \lhd \mathtt{column}$

$\}$

$$\mathtt{project} = \lambda(i,j)\ |\ (i{=}1, 1{\leq}j{\leq}n)\,.\,(j)$$
$$\mathtt{column}\ = \lambda(i,j)\ |\ (1{\leq}i,j{\leq}n)\,.\,(1,j)$$

It describes the replication of the first row of a matrix on the whole matrix as this loop nest says:

```
do i=1,n
 do j=1,n
  a(i,j)=b(j)
 enddo
enddo
```

The type system associated with the PEI program is:

$$\left\{\begin{array}{ll}
\left.\begin{array}{l}
\mathtt{val\ A} \subseteq \mathtt{ref\ A} \\
\mathtt{val\ B} \subseteq \mathtt{ref\ B} \\
\mathtt{val\ T} \subseteq \mathtt{ref\ T} \\
\mathtt{val\ T} \subseteq dom(\mathtt{project}) \\
dom(\mathtt{project}) \subseteq \mathtt{ref\ T} \\
img(\mathtt{column}) \subseteq \mathtt{val\ T} \\
dom(\mathtt{column}) \subseteq \mathtt{ref\ T}
\end{array}\right] & \text{conditions to define the data fields} \\
\left.\begin{array}{l}
\mathtt{project}(\mathtt{val\ T}) = \mathtt{val\ B} \\
\mathtt{project}(\mathtt{ref\ T}) = \mathtt{ref\ B} \\
\mathtt{val\ A} = dom(\mathtt{column}) \\
\mathtt{ref\ A} = \mathtt{ref\ T}
\end{array}\right] & \text{equations}
\end{array}\right.$$

4 Solving the type system: a type algorithm

The rest of the paper proposes an algorithm to solve a type system. The correctness of this algorithm lies on the equivalence of successive type systems obtained step by step from the initial one.

Example 4. Data replication (continued)

From the previous type system, the typecheck algorithm returns the following result:

$$\begin{cases} \texttt{val A} = \{(i,j) \mid 1 \le i, j \le n\} \\ \texttt{ref A} = \texttt{ref T} \\ \texttt{val B} = \{j \mid 1 \le j \le n\} \\ \texttt{ref B} = \{j \mid 1 \le j \le n\} \\ \texttt{val T} = \{(1,j) \mid 1 \le j \le n\} \\ \{(1,j) \mid 1 \le j \le n\} \subseteq \texttt{ref T} \end{cases}$$

Warning: our algorithm presents the following limitations:

- The following condition, say (C), is required:

  "$\texttt{h}_1 :: \texttt{X}$ and $\texttt{h}_2 :: \texttt{X}$ can be connected in an equation" iff $\texttt{h}_1 \preceq \texttt{h}_2$ (or $\texttt{h}_2 \preceq \texttt{h}_1$) where $\preceq$ denotes the relation "is a restriction of" between bijections.

  It means that all data fields resulting from a change of basis applied on the same data field must be in conformity inside an equation. Note that this is a rather weak limitation.
- In order to be eventually implemented, the algorithm also requires some operations to be available:
  . $dom(f)$ (domain), $img(f)$ (range),
  . $h_1 \preceq h_2$, $inv(h)$ (inverse), $h(dom)$ (image set),
  . $\subseteq$, $\cup$, $\cap$, $-$ (set difference).

  where h, h_1 and h_2 denote bijections from $\mathbb{Z}^n$ to $\mathbb{Z}^p$, f denotes a function from $\mathbb{Z}^n$ to $\mathbb{Z}^n$ and sets are subsets of $\mathbb{Z}^n$.

The typechecking algorithm consists in removing the unknowns of the system, one after the other. It works on a *normalized system*.

4.1 System normalization

A normalized system has the following form:

$$\begin{cases} \vdots \\ \langle \text{L} \rangle \;\subseteq\; \langle \text{R} \rangle \\ \vdots \end{cases}$$

with <L> ::= <V> | <cst> and <R> is a domain expression (<D>) in which any variable occurs only once and is different from <L>.

Example 5. Data replication (continued)

By definition, only the equational part has to be normalized. The normal form of the previous system is obtained as follows: the last two equations are easy to normalize, by replacing each of them with two inclusions. For example, $\texttt{val A} = dom(\texttt{column})$ leads to $\texttt{val A} \subseteq dom(\texttt{column})$ and $dom(\texttt{column}) \subseteq \texttt{val A}$.

The same idea applied to the former two equations leads to two new inclusions which must be normalized:

$$\begin{cases} \mathtt{project}(\mathtt{val\ T}) \subseteq \mathtt{val\ B} & (1) \\ \mathtt{project}(\mathtt{ref\ T}) \subseteq \mathtt{ref\ B} & (2) \end{cases}$$

- Since `val T` is a value domain and by definition of the change of basis operation, $\mathtt{val\ T} \subseteq dom(\mathtt{project})$ is already checked. So the inclusion (1) is equivalent to $\mathtt{val\ T} \subseteq inv(\mathtt{project})(\mathtt{val\ B})$ which has a normal form.
- Since `ref T` is a reference domain and by definition of the change of basis operation, $dom(\mathtt{project}) \subseteq \mathtt{ref\ T}$. So the inclusion (2) is equivalent to $img(\mathtt{project}) \subseteq \mathtt{ref\ B}$.

The normal form of a type system can be obtained by applying the following rules **R1** to **R7**:

R1	$<D>_{.1} = <D>_{.2}$	$\rightarrow$	$<D>_{.1} \subseteq <D>_{.2},\ <D>_{.2} \subseteq <D>_{.1}$	
R2	$<D>_{.1} \cup <D>_{.2} \subseteq <D>$	$\rightarrow$	$<D>_{.1} \subseteq <D>,\ <D>_{.2} \subseteq <D>$	
R3	$h(<D>_{.1} \cup <D>_{.2})$	$\rightarrow$	$h(<D>_{.1}) \cup h(<D>_{.2})$	
R4	$h_{.1}(h_{.2}(<D>))$	$\rightarrow$	$(h_{.1} \circ h_{.2})(<D>)$	
R5	$h(\mathtt{val\ X}) \subseteq <D>$	$\rightarrow$	$\mathtt{val\ X} \subseteq inv(h)(<D>)$	
R6	$h_{.1}(\mathtt{val\ X}) \cup \ldots \cup h_{.n}(\mathtt{val\ X})$	$\rightarrow$	$max\{h_{.1} \ldots h_{.n}\}(\mathtt{val\ X})$	$(n>1)$
R7	$\mathtt{val\ X} \subseteq <D> \cup h(\mathtt{val\ X})$	$\rightarrow$	$true$	

By applying rules **R1** to **R5**, any equality is rewritten as a set of inclusions of the form $<L> \subseteq <R'>$, where the right side $<R'>$ has still to be simplified because it can contain several occurrences of a same variable or $<L>$ can be a variable which occurs in $<R'>$.

Let us consider the first case: by the structural definition of the type system in 3.2, the variable denotes necessarily a value domain and the right side matches thus the following form: $<D> \cup h_{.1}(\mathtt{val\ X}) \cup h_{.2}(\mathtt{val\ X}) \cup \ldots \cup h_{.n}(\mathtt{val\ X}) \qquad (3)$

Remark. when `val X` is a term of this union (no bijection is applied on it), we replace it with `i(val X)` where `i` is the identity.

The condition (C) is required here to reduce the expression (3) to $<D> \cup h(\mathtt{val\ X})$ where h is the greatest of the functions $h_{.i}$ according to the order $\preceq$. This is expressed by the rule **R6**.

Finally, let us consider the case where $<L>$ is a variable which occurs in $<R'>$. It involves that $<L>$ denotes a value domain. The whole inclusion has the form: $\mathtt{val\ X} \subseteq <D> \cup h(\mathtt{val\ X})$. From condition (C), h is necessarily a restriction of identity. Then, it follows $h(\mathtt{val\ X}) = \mathtt{val\ X} \cap dom(h) = \mathtt{val\ X}$ (since $\mathtt{val\ X} \subseteq dom(h)$). Therefore the rule **R7** holds.

Property 8. *The rule system* **R1** *to* **R7** *forms a terminating system.*

This property is obvious since every rule can be applied at most once.

4.2 Unknown removing

Assuming a type system is under normal form, any unknown $\mathtt{X}$ is removed by applying this algorithm: let $S_{\mathtt{X}}$ be the set of inclusions where $\mathtt{X}$ occurs. The system can be reduced to:

. two inclusions which bound $\mathtt{X}$:

$$e \subseteq \mathtt{X} \subseteq E$$

where e and E are set expressions which do not use $\mathtt{X}$

. and a set of inclusions in the normal form which do not contain $\mathtt{X}$.

The details of the simplification are given hereunder:

- Let us consider the inclusions of $S_{\mathtt{X}}$ whose form is $\mathtt{X} \subseteq <\mathrm{D}>_{.i}$. Since $S_{\mathtt{X}}$ has a normal form, $<\mathrm{D}>_{.i}$ does not contain $\mathtt{X}$. These inclusions can be grouped into a single one of the form $\mathtt{X} \subseteq E$ with $E = \cap_i <\mathrm{D}>_{.i}$.
- Let us consider the other inclusions of $S_{\mathtt{X}}$. Since $S_{\mathtt{X}}$ has a normal form, the unknown $\mathtt{X}$ occurs in the right side. We obtain an inclusion of the form $<\mathrm{D}>_{.i} \subseteq \mathtt{X}$ by isolating $\mathtt{X}$. The new inclusions can be grouped into a single one of the form $e \subseteq \mathtt{X}$ with $e = \cup_i <\mathrm{D}>_{.i}$.

To show how to isolate $\mathtt{X}$ on the right side of an inclusion of $S_{\mathtt{X}}$, let us consider the most general form:

$$<\mathrm{L}> \ \subseteq \ <\mathrm{D}> \cup \ h(\mathtt{X})$$

This inclusion is equivalent to:

$$\begin{cases} inv(h)(<\mathrm{L}>) - inv(h)(<\mathrm{D}>) \ \subseteq \ \mathtt{X} & (4) \\ <\mathrm{L}> \ \subseteq \ <\mathrm{D}> \cup \ img(h) & (5) \end{cases}$$

where (4) has the required form and (5) does not use $\mathtt{X}$. Moreover, we note that (5) has a normal form.

So, we obtain bounds for $\mathtt{X}$ and a system in the normal form where $\mathtt{X}$ does not appear anymore and from which another unknown can be removed.

4.3 Ending of the algorithm

The unknowns of the type system are removed one after the other. At the end the resulting system does not contain any unknowns: the inclusions can then be evaluated by computing constant domain expressions. If all inclusions are evaluated to true, then PEI expression is well typed. In this case, the algorithm returns a type range for all data fields in the program.

In the other case, there exists a data field without a type. Some informations about the type error source can be obtained, but this point is out of the scope of the paper.

Complexity of the algorithm depends on two factors: the number of operations and the number of identifiers in the statement: rules of normalization can be applied while building the type system using a structural scanning. Unknowns removing phasis is clearly linear relatively to the number of identifiers in the statement.

5 Implementation

Based on this algorithm, a typechecker for Pei programs has been written in Caml [18]. It uses the Omega [5, 12] library for evaluating set expressions: the notion of *relation* in Omega is based on Presburger formula and allows to represent functions and domains.

The Omega library allows to handle subsets of $\mathbb{Z}^n \times \mathbb{Z}^m$. When $m \neq 0$, these subsets define *relations* which connect n-tuples to m-tuples. When $m=0$, it define subsets of $\mathbb{Z}^n$. A relation has the following syntax:

{[<InputList>] [-> [<OutputList>]] [: <formula>]}

where <InputList>, <OutputList> are tuples. Tuples relations and sets are described by using *Presburger formulae* a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$.

To link these tools, we developed a syntactic analyzer of Pei in Caml and we extended an interface for using the functions of the Omega library in Caml.

A representation function translates a Pei function into an Omega relation. Note that this representation is not always possible because Pei expressions are not necessary limited to Presburger formulae.

A Pei statement can be typed if the geometrical operations, reductions or change of basis inside can be coded into an Omega relation. They must have the following form:

<func> ::= λ(<tuple>$_{.1}$) [|(<pred>)] [.(<tuple>$_{.2}$)]

<tuple> ::= <expr> | <expr>,<tuple>

<expr> ::= <expr> (+ | -) <expr>
| (+ | -) <expr>
| <expr> (**div** | **mod**) <cst>
| <cst>×<expr>
| <expr>×<cst>
| (<expr>)
| <index> | <param> | <cst>

<pred> ::= <expr> (= | <> | < | > | ≤ | ≥) <expr>
| <pred> (|| | &&) <pred>
| !<pred>
| (<pred>)
| **true** | **false**

where <cst> is either an integer, or a constant expression whose syntax is <expr>, and <param> denotes a parameter.

Example 6. Data replication (continued)

When applied to the program `Broadcast`, the typechecker provides the following result:

```
Type-check succeeded ...

ref B equalsto (=) {[j]: 1 <= j <= n}
val B equalsto (=) {[j]: 1 <= j <= n}
ref A equalsto (=) ref T
val A equalsto (=) {[i,j]: 1 <= i <= n && 1 <= j <= n}
ref T supsetof (>) {[i,j]: 1 <= i <= n && 1 <= j <= n}
val T equalsto (=) {[1,j]: 1 <= j <= n}
```

Example 7. Prefix-Sum of n numbers (continued)

When applied to the program **PrefixSum**, the typechecker provides the following result:

```
Type-check succeeded ...

ref A equalsto (=) {[i]: 1 <= i <= n && 2 <= n}
val A supsetof (>) {[1]: 2 <= n}
val A subsetof (<) {[i]: 1 <= i <= n && 2 <= n}
ref X supsetof (>) {[i]: 1 <= i <= n && 2 <= n}
val X equalsto (=) {[i]: 1 <= i <= n && 2 <= n}
```

6 Conclusion

This paper defined the notion of type associated with the objects used in the language PEI and presented an algorithm that can infer the type of PEI expressions. Based on this algorithm, a typechecker has been implemented using OMEGA, it allows conditions required by refinement rules to be easily checked. Beyond this practical approach, this allows to determine well-formed phrases of the language [14] and further research may consist in associating classical semantics with programs. This notion of type is of great importance to prove correctness of parallel programs as well as to determine allowable program transformations.

References

1. Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:2, April 1987. AT&T Bell Laboratories.
2. M. Chen, Y. Choo, and J. Li. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991. Chapter 7.
3. Stéphane Genaud, Eric Violard, and Guy-René Perrin. Transformations techniques in PEI. *EUROPAR'95, LNCS*, 966:131–142, August 1995.
4. P. Hammarlund and B. Lisper. On the relation between functional and data parallel programming languages. *FPCA93, ACM Press*, pages 210–222, 1993.
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. L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.
7. C. Lengauer. Loop parallelization in the polytope model. *Parallel Processing Letters*, 4(3), 1994.

8. B. Lisper. *Data Parallelism and Functional Programming.* School on Data Parallelism, Les Ménuires (France). LNCS 1132-Tutorial Series, 1996.
9. C. Mauras. ALPHA *: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* PhD thesis, U. Rennes, 1989.
10. C. Mongenet, P. Clauss, and G.-R. Perrin. Geometrical tools to map systems of affine recurrence equations on regular arrays. *Acta Informatica*, 31:137–160, 1994.
11. C. Morgan. *Programming from specifications.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1990.
12. William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.
13. P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, 1989.
14. R.D. Tennent. *Semantics of Programming Languages.* C.A.R. Hoare. Prentice Hall Ed., Endlewood Cliffs, N.J., 1991.
15. E. Violard and G.-R. Perrin. PEI : a language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.
16. E. Violard and G.-R. Perrin. PEI : a single unifying model to design parallel programs. *PARLE'93, LNCS*, 694:500–516, June 1993.
17. Eric Violard, Stéphane Genaud, and Guy-René Perrin. Refinement of data parallel programs in PEI. *IFIP TC2 Workshop on Algorithmic Languages and Calculi, Chapman & Hall*, February 1997.
18. Pierre Weis and Xavier Leroy. *Le langage* CAML. Interéditions - iia, 1993.

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