Computing and Informatics, Vol. 22, 2003, 439–456

UNIFIED APPROACH TO ENVIRONMENTS IN A PROCESS FUNCTIONAL LANGUAGE*

Ján Kollár

Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
Letná 9, 041 20 Košice, Slovakia
e-mail: Jan.Kollar@tuke.sk

Manuscript received 12 July 2002; revised 19 March 2003
Communicated by Gianfranco Rossi

Abstract. A process functional paradigm is based on applications of processes and functions instead of assignments. The imperative computation and the functional evaluation are clearly separated in a well-balanced manner, providing the strong feedback about the implementation to a user. In this paper we present the unified approach to explicit and implicit environments in $\mathcal{PFL}$ – an experimental process functional language, as a uniform basis for approved implementation extensible by additional specification. $\mathcal{PFL}$ environmental concept is the generalized implementation substance, which can be found in many programming languages exploiting the large variety of programming paradigms. Environment variables do not occur in expressions, being still visible to a programmer either in $\mathcal{PFL}$ textual form or in an equivalent form of control driven data flow graphs. The approach is promising for reasoning about the functional correctness and predicting the behavior of systems.

Keywords: Programming paradigms, implementation principles, explicit and implicit environments, environmental application, control driven dataflow, referential transparency, side effects

* This work was supported by VEGA Grant No. 1/8134/01 — Binding Process Functional Language to MPI

1 INTRODUCTION

The role of the state in systems is crucial. In the past, combining imperative and functional paradigms to exploit the benefits of both — the ability to manipulate the state preserving at the same time functional semantics — has result to two categories of functional languages, able to manipulate the state [17, 1, 2, 3, 16, 21, 22, 27].

The first category comprises languages that have assignments used in expressions in an explicit form, such as SML [17] or Scheme [1]. Such languages are characterized as environment-based [23], since environment variables occur in expressions.

The second category comprises languages, such as Clean [2, 3] or Haskell [22], that hide assignments to a user, performing them implicitly, abstracting from environment variables. Clean and Haskell are imperative functional languages, since they are functional languages, manipulating the state in a hidden way to a user. They are often classified as being not environment-based languages, since they use the lazy evaluation mechanism based on the graph reduction.

$\mathcal{PFL}$ — an experimental process functional language [9, 10, 11, 12, 13] integrates both imperative and functional paradigms since we have found them essential as those being associated with the physical resources of underlying computer architectures. Therefore $\mathcal{PFL}$ origins are rather in imperative functional programming languages, than in some more abstract multi-paradigmatic languages, such as Oz [4, 20, 24].

Our essential idea is to make the strong boundary between the specification and implementation, concentrating at the first stage to balancing the abstraction with respect to the development of "the most abstract" implementation language. Rather than determining fixed strategy for a single programming language it is more interesting for us and it is far more flexible for constructing the systems to derive this strategy on the basis of both specification and architecture resources.

A $\mathcal{PFL}$ program consists of a set of recursive functions and processes and the main expression. A pure function is just a specific process, such that it does not affect a variable environment and both arguments and value of pure functions are of data types, not however of unit type () (as it is in Haskell, for example). On the other hand, a process is a function, such in which at least one argument is bound statically to an environment variable, or at least one argument or value is control (of unit type). Syntactically, there is no difference between process and function definitions. They differ just by their type definitions that either contain environment variables and/or unit types (in case of a process), or not (in case of a function). Hence, type definitions for processes are obligatory, while type definitions for pure functions are optional.

There are no assignments in $\mathcal{PFL}$ available. Evaluating the main expression, the program is executed, similarly as it is in functional languages Haskell and Gofer [8]. On the other hand, the variable environments comprising cells of memory are visible to a programmer via type definitions of processes.

We do not use in $\mathcal{PFL}$ source program neither low level functional constructs, such as LET strict or lambda non-strict expressions, nor such abstractions as mo-

nads [26, 27], since we do not want to hide the imperative actions to a user making them functional; we just want to separate them from purely functional grains to provide the transparent and strong feedback about mapping a problem to target architecture to a user.

The state is changed by applications of processes and may be monitored using graphic interface strongly bound to the source program visualizing the flow of data in environments. As we hope, the well-balanced structure of $\mathcal{PFL}$ is promising for reasoning about the functional correctness and predicting statically the behaviour of the systems using the approaches for Petri nets [6, 7] in the future.

This paper deals neither with the reasoning, nor with the detailed description of $\mathcal{PFL}$ language. Here we present the unified approach to explicit and implicit environments, using process functional paradigm, concentrating technically more on giving the $\mathcal{PFL}$ language shape than the semantics. For the purpose of explanation, the sequential evaluation of arguments of processes by value, i.e. the eager evaluation is supposed.

It is not to say that such categorization of environments is standard, we just have found them simply existing. The difference between them is as follows. Explicit environment is formed by a set of new variables (memory cells) defined by a programmer, i.e. explicitly in type definitions of processes. Implicit environment is formed by a selected subset of the set of memory cells, being comprised in a structured value (such as tuple, or array). Such value is accessed via lambda variable in a function body, and each sub-cell may be used as an environment variable for each local process. Thus, structured values provide potential opportunity for affecting their sub-cells by local processes in the same manner, as if these cells were explicitly defined. It is not so hard to see that restricting just to implicit environment, object concept of a language is impossible. Object environment is associated with abstract typing and with explicit environments [13]. The motivation for the categorization above is simple uniform structure of $\mathcal{PFL}$ as an implementation language.

Special remarks to sharing the environment variables are introduced in conclusion with respect to further work.

In Section 2 we present the essential concept of an environment variable as a mutable abstract type representing a memory cell, able to store and retrieve the values, when activated by process application.

Section 3 provides a brief introduction to static binding of an environment to a process (unshared case), as well as an informal operational semantics of process application with respect to state change, restricted however to sequential eager evaluation, as mentioned above.

The extension of the explicit environment concept to the implicit one is introduced in Section 4.

$\mathcal{PFL}$ examples would be more explanatory, if all supported by the equivalent form of control driven dataflow graphs. Unfortunately, this is impossible due to limited scope of the paper. Control driven dataflow [11] is rather akin to Petri nets execution [6, 7] than to data or demand driven dataflow, exploited in dataflow architectures [25]. $\mathcal{PFL}$ programs are introduced in mathematical form i.e. instead

of `->`, `if`, `x`, `[]`, ... the notation $\rightarrow$, **if**, x, [], ... is used. On the other hand, since curly brackets { and } are $\mathcal{PFL}$ language symbols, the brackets [and] are used for (mathematical) sets, whenever required.

2 ENVIRONMENTAL APPLICATION

The notion of a variable in a purely functional language is mathematical; a variable v is a value of the type T, i.e. $v : T$.

On the other hand, an imperative variable is a memory cell used to store values. The environment is the mapping from variables to values, as follows:

$$env = Var \rightarrow Val. \tag{1}$$

The value of an environment variable v is accessed by the application $(env\ v)$, and a new value m is assigned to v by $[v \mapsto m]$. According to definition (1), each environment variable is a passive element of the computation, which is accessed and/or updated by active elements, such as assignment operations.

Suppose now that the variable is not just a cell, but it is a "more active" entity, which may be applied, when defined as the overloaded mapping:

$$v : \widetilde{T} \rightarrow T. \tag{2}$$

Such mappings are related to the theory of mutable abstract types [5].

From the functional point of view, the mapping (2) has two instances (3); the identity, which means *the update* of an environment (1) variable, and a "constant", which means *the access* of a value in an environment (1) variable:

$$\text{the update} \quad \begin{cases} v : \ T \rightarrow T \\ v\ x \ = \ x \end{cases} \qquad \text{the access} \quad \begin{cases} v : \ () \rightarrow T \\ v\ () \ = \ env\ v. \end{cases} \tag{3}$$

Hence, the variable v may be applied either to expressions of data type T or unit type (), i.e. of the type $\widetilde{T}$. The application rule for an environment variable is as follows.

$$\frac{v : \widetilde{T} \rightarrow T \qquad m : \widetilde{T}}{v\ m : \ T} \tag{4}$$

However, the application (4) is not functional, but rather environmental, since the operational semantics is still related to an environment (1). Moreover, it may be performed just implicitly, i.e. by a process application, since this is the only way in von Neumann computers, how it may be activated. A single argument process is translated (see Section 3) into lambda abstraction $(\lambda x.e)$, and environmental application is its argument. Then the operational semantics of environmental application $(v\ m)$ is dependent on the type of an expression m. *The update* of a variable v by an expression m of data type T, is performed as follows:

$$\mathcal{E}val[\![(\lambda x.e)\ (v\ m)]\!]\ env\ =\ \mathcal{E}val[\![e[(env\ v)/x]]\!]\ [v \mapsto m]\ env,\ \text{if}\ m : T. \quad (5)$$

On the other hand, *the access* to an environment variable is performed, if an environment variable v is applied to an expression m of unit type:

$$\mathcal{E}val[\![(\lambda x.e)\ (v\ m)]\!]\ env\ =\ \mathcal{E}val[\![e[(env\ v)/x]]\!]\ env,\ \text{if}\ m : (). \quad (6)$$

According to (4), (5) and (6), if $v : \widetilde{T} \rightarrow T$, then $x : T$, i.e. lambda variable x bound to an environment variable v cannot be of the type (). In $\mathcal{PFL}$ we strongly distinct the control value () (known as unit value in functional languages), of unit type () from data values of the type T. When applying an environment variable to an expression evaluated to the control value (), provided that an environment variable was not initialized, then $v\ () = \perp$, i.e. $x = \perp$. The detection of such inappropriate applications of environment variables to control values is the matter of a sophisticated type checking, highly parametrised by the separated evaluation strategy, providing the transparent feedback to a user bound to the $\mathcal{PFL}$ source program.

Environmental applications are easily and efficiently implemented using built-in LETENV operation — a hidden access/update operation executed before an argument value is used in a process application [9].

As shown below, a variable environment is derived in the form of the set of mappings (2), instead of mappings (1).

3 DERIVING AND OPERATING THE ENVIRONMENT

Although internally the environment variables are applied to expressions, being arguments of processes, they never occur in expressions of $\mathcal{PFL}$ source form. Instead of that, environment variables are bound statically to processes by type definitions of processes. In this way the state is determined by the semantics of process applications exclusively, being not affected by an inapproriate occurence of environment variables elsewhere in expressions. We may even say that adding the semantics of process application is the only step needed when adopting a non-deterministic *implementation* language into a deterministic *programming* language.

For the purpose of explanation, as a sample of the process application semantics, below the leftmost innermost order of evaluation will be assumed, i.e. such in which the arguments of processes are evaluated sequentially and eagerly.

In contrast to a function definition, each process definition comprises the type definition. In general, an m-argument process definition is as follows.

$$\begin{array}{l} f :: \overline{T_1} \rightarrow \ldots \rightarrow \overline{T_m} \rightarrow \widetilde{T} \\ f\ x_1\ \ldots\ x_m\ =\ e \end{array} \quad (7)$$

where $m \geq 0$, $\overline{T} \ ::= v\ T \mid v\ \{R\}\ T \mid \ \widetilde{T}$, $\widetilde{T} ::= \ () \mid T$, $T ::= \ T \rightarrow \ T \mid \{R\} \rightarrow T \mid T^D \underbrace{T \ \ldots \ T}_{u} \mid T^P$, $\{R\}$ is n-dimensional range $\{R\} ::= \ \{T_1^R, \ldots, T_n^R\}$, T_i^R are range types — enumerated algebraic types, characters, integers, or their finite subranges in the form $c_i^L \,..\, c_i^U$, corresponding to lower and upper bounds of an array, $c_i^L, c_i^U : T_i^R$, T^D is an algebraic data type of the arity $u \geq 0$, T^P is a primitive data type or the type variable, v is an environment variable, and $()$ is unit type.

A process argument type is $\overline{T}$. A process value type is $\overline{T}$, such that does not contain an environment variable v, i.e. it is $\widetilde{T}$. A pure function arguments and/or value are of the type T.

To prevent misunderstanding, the source "type expression" $(v\ T)$ is a *syntactic shortcut* for $(v : \widetilde{T} \rightarrow T)$. Similarly, the "type expression" $(v\ \{R\}\ T)$ is a syntactic shortcut standing for $(v\ \{i\} : \widetilde{T} \rightarrow T)$, such that $\{i\} \in \{R\}$. Thus, if v is an array, then $(v\ \{i\})$ is an environment variable for an i-th item of this array.

Example 3.1 illustrates the translation step, in which the environment is derived. At the same time, we will show how the sequential process applications are operated using scalar environment variable u, algebraic data environment variable v and array environment variable w. As shown later, provided that u, v and w are not lambda variables, they all belong to explicit environment. For example, this comes into account if process f is defined in global (main program) scope marked by $s = 0$.

Example 3.1.
Let us define an algebraic type C and a $\mathcal{PFL}$ process f:

$$\textbf{data}\ C \ = \ \mathit{Red} \mid \mathit{Green} \mid \mathit{Blue}$$

$$\begin{aligned} f \ &:: \ a \rightarrow u\ a \rightarrow v\ C \ \rightarrow \ w\ \{\mathit{Bool}, 2\ ..\ 5\}\ a \rightarrow \ a \\ f\ x\ y\ c\ i \ &= \ x + y + i, \quad \textbf{if}\ c = \mathit{Red} \\ &= \ x + y - i, \quad \textbf{otherwise}. \end{aligned}$$

The control driven data flow graph for the above definition is introduced in Figure 1. Notice the different input arcs — a thick white arc is data arc, a thick gray arc is spatial arc addressing the array element in the form of an offset computed with respect of element type, and a thin black arc is control arc. Corresponding to the type $\widetilde{T}$, input arc of environment variables is either data or control arc, depending on data or unit type of incoming argument. In addition to the process definition in Example 3.1, the direct control inputs and control output of a process expression are allowed, which results to $\mathcal{PFL}$ ability for expressing any imperative program, as shown in [10].

At the first stage, the type environment is derived

$$\begin{aligned} &[u : \widetilde{a} \rightarrow a,\ v : \widetilde{C} \rightarrow C,\ w : \{\mathit{Bool}, 2\ ..\ 5\} \rightarrow (\widetilde{a} \rightarrow a), \\ &\quad f : a \rightarrow a \rightarrow C \ \rightarrow a \rightarrow \ a] \end{aligned} \tag{8}$$

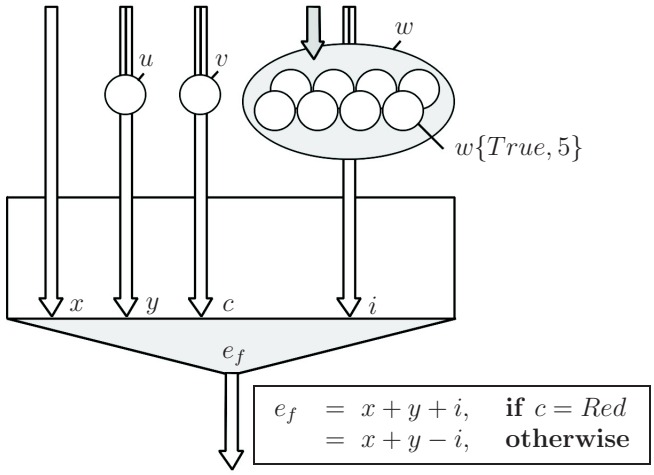

Fig. 1. The definition of the process f

marking u, v and w as the environment variables, and transforming the type definition of f. The aim of the derivation above is to disconnect environment variables and (seemingly) pure function definition. For example, source shortcut $(u\ a)$ for the second argument type implies environment variable u of the type $\widetilde{a}\rightarrow a$ and functional argument of the type a. Since w is an array, it is a spatial mapping of the type $\{Bool, 2\ ..\ 5\} \rightarrow (\widetilde{a}\rightarrow a)$, i.e. a variable cell affected such as $(w\ \{False, 3\})$ is of the type $(\widetilde{a}\rightarrow a)$. The derivation is more complicated, if an environment variable is shared by multiple processes, since then its type is derived by unification.

In general, for all applications of the process f, the argument types are unified. As a result, the type environment (8) is derived being specialized to a monomorphic one. Omitting the detailed representation of such monomorphic types, let the derived monomorphic type environment be as follows:

$$\begin{array}{l}[u : \widetilde{Int}\rightarrow Int,\ v : \widetilde{C}\rightarrow C,\ w : \{Bool, 2\ ..\ 5\} \rightarrow (\widetilde{Int}\rightarrow Int),\\ \ \ f : Int \rightarrow Int \rightarrow C\ \rightarrow Int \rightarrow\ Int]\end{array}$$

Then the variable environment E^s in the scope s, equal to the scope of the process f, is derived, in the form:

$$E^s\ =\ [u : \widetilde{Int}\rightarrow Int,\ v : \widetilde{C}\rightarrow C, w : \{Bool, 2\ ..\ 5\} \rightarrow (\widetilde{Int}\rightarrow Int)].$$

Suppose now the application $(f\ 3\ 5\ Red\ (\{False, 3\}\ 6))$ occurs somewhere in an expression, such that f is accessible. Notice that it contains no environment variable. This application is translated into the following form:

$$(f\ 3\ (u\ 5)\ (v\ Red)\ (w\ \{False, 3\}\ 6)).$$

The result of the application in the run time is as follows:

$$[u \mapsto 5, v \mapsto Red, w\{False, 3\} \mapsto 6]\ 14,$$

where the environmental mappings preceding the evaluation are enclosed in brackets, and the result of purely functional evaluation is 14.

Suppose the source form of application application $(f\ 2\ ()\ Blue\ (\{False, 3\}\ ()))$ – translated into $(f\ 2\ (u\ ())\ (v\ Blue)\ (w\ \{False, 3\}\ ()))$ is evaluated subsequently to the preceding application. It yields the result

$$[u \mapsto 5,\ v \mapsto Blue,\ w\{False, 3\} \mapsto 6]\ 1,$$

as shown in Figure 2.

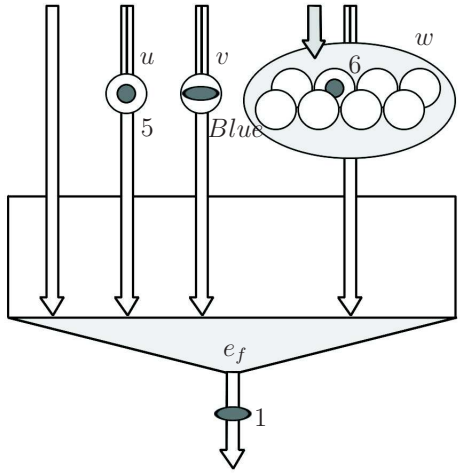

Fig. 2. The result of application $(f\ 2\ ()\ Blue\ (\{False, 3\}\ ()))$

While the fourth argument is evaluated in the Example 3.1, two subsequent expressions were internally evaluated: $(w\ \{False, 3\}\ 6)$ and $(w\ \{False, 3\}\ ())$. Constant index expression $\{False, 3\}$ was used here to illustrate the update and the subsequent access of the same $(\{False, 3\}$-th item of an array w. Considering more general form $(w\ \{e^{Bool}, e^{2\ \cdot\ \cdot\ 5}\}\ m)$, such that $\{e^{Bool}, e^{2\ \cdot\ \cdot\ 5}\} : \{Bool, 2\ .\ .\ 5\}$, and $m : \widetilde{Int}$, the expression is evaluated in two steps as follows:

$$\frac{\dfrac{w : \{Bool, 2\,..\,5\} \to (\widetilde{Int} \to Int) \quad \{e^{Bool}, e^{2\,\cdot\,\cdot\,5}\} : \{Bool, 2\,..\,5\} \quad m : \widetilde{Int}}{w\ \{e^{Bool}, e^{2\,\cdot\,\cdot\,5}\} : \widetilde{Int} \to Int \quad m : \widetilde{Int}}}{w\ \{e^{Bool}, e^{2\,\cdot\,\cdot\,5}\}\ m : Int},$$

where the first application ($w\ \{e^{Bool}, e^{2\,\cdot\,\cdot\,5}\}$) is not environmental, but rather spatial, addressing the spatial position of an array item, while the second ($w\ \{e^{Bool}, e^{2\,\cdot\,\cdot\,5}\}\ m$) is environmental application evaluated to an item value (of the type Int) used in an expression. The array w of the type $\{Bool, 2\,..\,5\} \to (\widetilde{Int} \to Int)$ is *static array*, because it cannot be used in expressions in a higher order manner. Lambda variable representing the array item value is of the type Int. There is no way however for static arrays to obtain lambda variable of the type $\{Bool, 2\,..\,5\} \to (\widetilde{Int} \to Int)$.

Deriving the environment and reasoning about undefined values which results in simplified notion of lambda variables type in the form T instead of seemingly more correct $T_\perp$, are based on the proposition of first order processes, i.e. such that may be applied just in the form $(f\ e_1 \ldots e_n)$, where f is the name of a process, not in higher order form $(e_0\ e_1)$, available for functions. It is easy to see that allowing processes in the form e_0 computable in the run time, the selection of a set of accessed and/or updated environment variables would be run time dependent, i.e. it could not be reasoned statically.

A new purely functional thread of evaluation starts by application of a process to argument of the type (). This thread may cause side effects, affecting the state of computation, being alone not affected. Purely functional threads (or grains) of computation may be detected in the compile time, and they are strongly related to the source program.

4 IMPLICIT ENVIRONMENT

The environment variables form a set of typed places (possibly shared by different processes and different arguments of the same process), in the role of an input memory gate of a process. In this way the "dangerous" imperative state affecting actions are separated from the "safe" evaluation of expressions. At the same time, imperative assignments are closely related to process applications. Both actions and memory cells can be expressed in the form of control driven dataflow graphs.

So far we have considered variables comprised in an explicit environment, since they were introduced using new names, different from lambda variables, in the type definitions of processes. As shown below, the extension of this concept to implicit variable environment defined by a selected subset of lambda variables is gentle; a variable may be a member of an explicit environment, if it is not a member of an implicit environment.

Let $x@(p,q)$ be a pair x of two items p, and q, that are either values in expressions, or memory cells in type definitions of local processes, as shown below.

Let us consider the definition of a "pure" function in the form as follows:

$$f\ x@(y@(p,q),r)\ =\ e_f$$

in the scope s, and suppose the application $(f\ ((e_p,e_q),e_r))$, such that e_p, e_q and e_r are expressions, e_p evalutes to control value $(e_p : ())$ and both e_q and e_r evaluate to data values. Let the value x of the argument, as illustrated in Figure 3, be partially defined.

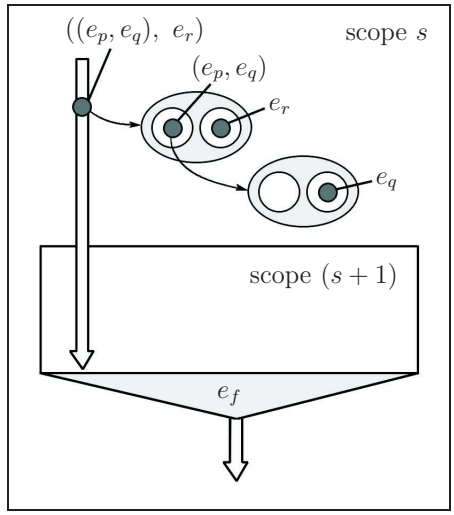

Fig. 3. The application $f\ ((e_p,e_q),\ e_r)$

The lambda variable x is the constant which cannot change in the context of f definition; this guarantees the referential transparency of the expression evaluation.

According to the pattern matching rule

$$\frac{x@(y@(p,q),r) :\ ((T_1,T_2),T_3)}{x :\ ((T_1,T_2),T_3)\quad y :\ (T_1,T_2)\quad p :\ T_1\quad q :\ T_2\quad r :\ T_3}$$

the selection mechanism guarantees the variables x, y, p, q, and r are (old) values that may be potentially used in the expression e_f. The type checking however excludes p from this set, since p value is undefined, as shown in Figure 4.

On the other hand, the value represented by lambda variable x can be redefined partially, updating one or more of variables y, p, q, and r that are implicit environment variables — potential environment variables for any process defined in the

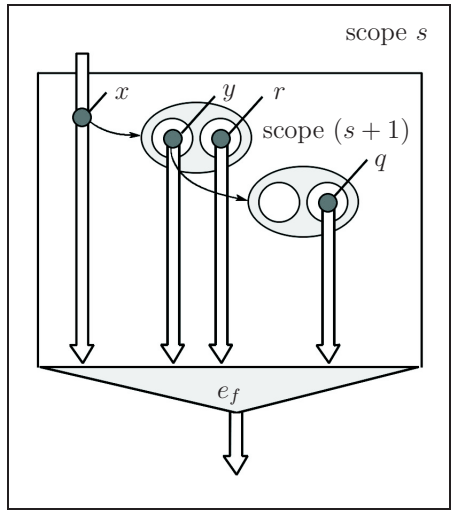

Fig. 4. The use of old values x, y, q, and r in e_f

scope $(s+1)$, local to the scope s. Pattern matching rule for implicit environment variables is as follows:

$$\frac{x@(y@(p,q),r) :: ((T_1,T_2),T_3)}{y : (T_1,T_2) \tilde{\to} (T_1,T_2) \quad p : T_1 \tilde{\to} T_1 \quad q : T_2 \tilde{\to} T_2 \quad r : T_3 \tilde{\to} T_3}$$

and y, p, q and r may belong to the implicit environment $I^{(s+1)}$ — a set of environment variables for all processes defined in the scope $(s+1)$ local to scope s provided that these processes use them in their type definitions.

For instance, let y, p, q and r be all used in the type definition of a process g in Example 4.1.

Example 4.1.

$$\begin{aligned} f\ x@(y@(p,q),r) \ &= e_f \\ &\quad \textbf{where} \\ &\quad\ g :: y\ (T_1,T_2) \to r\ T_3 \to p\ T_1 \to q\ T_2 \to T_g \\ &\quad\ g\ a\ b\ c\ d \ = e_g \end{aligned}$$

corresponding to Figure 5.

Then it holds

$$[y : (T_1,T_2) \tilde{\to} (T_1,T_2),\ p : T_1 \tilde{\to} T_1,\ q : T_2 \tilde{\to} T_2,\ r : T_3 \tilde{\to} T_3] \subseteq I^{(s+1)}.$$

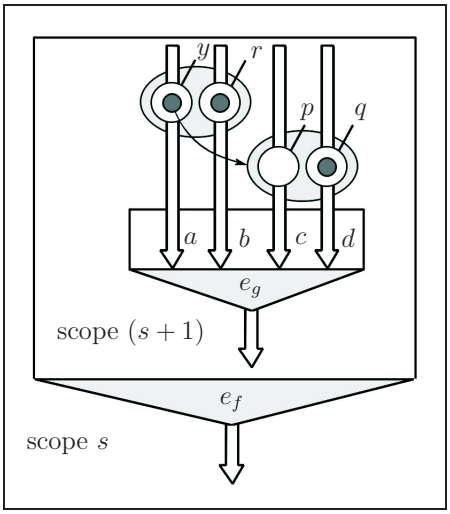

Fig. 5. The use of implicit environment by a process g local to f

The types of arguments of processes having been defined in the scope $(s+1)$, local to the scope s may be derived automatically. Thus, it is possible to use underscore _ instead of explicit type $\widetilde{T}$, designating "any" type, i.e. such that can be derived.

The state may be manipulated even excluding the explicit environment from the computation as it is done in imperative functional languages, by the application of the local process, such as g when defining a new value e_p. In $\mathcal{PFL}$, however, the previous value in p may be undefined.

The same situation arises when processes or functions are applied to *dynamic* arrays. A $\mathcal{PFL}$ array is an extensionally defined mapping from the space of (n-grid) immaterial positions to values, which is created either fully undefined or fully defined, and it can change its definition during the computation. To create n-dimensional array with undefined items we provide the aggregated array creator, similar to the type expression, but used in an expression, as follows:

$$\{R^F\} \to (),$$

where $\{R^F\}$ is a finite subrange, $\{R^F\} \subseteq \{R\}$, taking into account that the array is created being first of the type $\{R^F\} \to \widetilde{T} \to T$, and then, when each item is applied to (), it is of the type $\{R^F\} \to T$.

Except creating fully undefined arrays, $\mathcal{PFL}$ array creator $\{R^F\} \to m$, where $m : T$, serves to create a fully defined array comprising iteratively evaluated expression m, storing the evaluated values subsequently in a memory chunk allocated to an array. Such aggregated computation is implemented using built-in loop comprehension [12].

Let us have the array created as follows

$$\{R_1^F\} \to \{R_2^F\} \to ()$$

and suppose a process f is applied to $\{R_1^F\} \to \{R_2^F\} \to ()$. Then, the corresponding type of an argument in the process f type definition may occur just in the form $w\ (\{R_1\} \to \{R_2\} \to T)$, where w is either implicit or explicit environment variable, since it is commonly impossible to define finite ranges in source program, that are computed in the run time. Then, the derived type for an environment variable w is as follows:

$$w:\ \overbrace{(\{R_1\} \to \{R_2\} \to T)}^{\sim} \to (\{R_1\} \to \{R_2\} \to T).$$

Notice however that it is not so much substantial whether the argument type of f comprises the environment variable w, as introduced above, or not. In both cases, either according to the environmental application rule (4), or directly, the corresponding lambda variable x of either the process or function f is of the type

$$x:\ \{R_1\} \to \{R_2\} \to T.$$

Lambda variable x can be used in the definition of f in the next forms: x, which means the array of the array of item values, $x\ \{e^{R_1}\}$ means the array of item values (of the type $\{R_2\} \to T$), and $x\ \{e^{R_1}\}\ \{e^{R_2}\}$ means the value of item, of the type T, still being $\perp$, if the array was not initialised. The lambda variable x itself cannot be an environment variable, since it is a constant — a pointer to a contiguous chunk of memory representing the array of array of items. On the other hand, the chunks of memory consist of environment variables. Therefore, if f is defined in the scope s, it holds:

$$[x\ \{R_1\} :\ \overbrace{(\{R_2\} \to T)}^{\sim} \to (\{R_2\} \to T),\ x\ \{R_1\}\ \{R_2\} : \widetilde{T} \to T] \subseteq I^{(s+1)}.$$

Then it is possible to define a process g in the scope $(s+1)$ local to f using argument type $x\ \{R_1\}\ (\{R_2\} \to T)$, or $x\ \{R_1\}\ \{R_2\}\ T$. But, since both range types and the item type are derivable from the type of lambda variable x, the argument types can be defined using any type $\_$, in the form $(x\ \{\}\ \_)$, or $(x\ \{\}\ \{\}\ \_)$.

Explicit environment variables names must differ not just from the names of processes and functions, but also from the names of variables forming implicit environment at the same scope. The last check is easily performed using the condition

$$v : \widetilde{T} \to T \in E^s,\ \text{if } v : \widetilde{T} \to T \notin I^s$$

for all $v\ T$ attempting to be an argument type of a process in the scope s, such that v be an explicit environment variable. Clearly, argument type expressions comprising "any" type cannot be used for explicit environment variables, since their types cannot be derived.

Manipulating static and dynamic arrays is illustrated in Examples 4.2 and 4.3. While explicit static array of the size $maxT$ in Example 4.2 represented by a variable *table* is initialised by process *init*, which uses loop comprehension applies *tab* iteratively; the dynamic array of the same size is created and initialised in Example 4.3 by application of $init'$ defined by array creator.

According to the definition of a process *acup* the k-th item of *table* is either accessed and/or updated by an application $acup\ (\{k\}\ m)$, depending on the type of m (() or T), which is decidable statically.

Example 4.2.

$$
\begin{aligned}
&tab\ ::\ table\ \{0\ ..\ maxT-1\}\ [Char] \to ()\\
&tab\ \_\ =\ ()
\end{aligned}
$$

$$
\begin{aligned}
&init\ ::\ ()\\
&init\ =\ (tab\ (\{i\}\ [])\mid i \leftarrow \{1\ ..\ maxT\})
\end{aligned}
$$

$$
\begin{aligned}
&acup\ ::\ table\ \{0\ ..\ maxT-1\}\ [Char] \to [Char]\\
&acup\ item\ =\ item
\end{aligned}
$$

In Example 4.3, while the k-th item of a dynamic array t (having been created by $init'$) can be accessed (not however updated) by the application $access\ t\ k$, the application $(tab'\ t\ i)$ means either access or update by the value m, depending on the type of m (decidable statically again). Using the implicit environment formed by the lambda variable *table* of tab' the access or update action is performed here by application of local process $acup'$.

Example 4.3.

$$
\begin{aligned}
&init'\ ::\ \{0\ ..\ maxT-1\} \to [Char]\\
&init'\ =\ \{0\ ..\ maxT-1\} \to []
\end{aligned}
$$

$$
\begin{aligned}
&access\ ::\ (\{0\ ..\ maxT-1\} \to [Char]) \to Int \to [Char]\\
&access\ table\ i\ =\ table\{i\}
\end{aligned}
$$

$$
\begin{aligned}
&tab\ ::\ (\{0\ ..\ maxT-1\} \to [Char]) \to Int \to [Char]\\
&tab\ table\ index\ =\ acup'\ (\{index\}\ m)\\
&\qquad\qquad\textbf{where}\\
&\qquad\qquad\quad acup'\ ::\ table\ \{\}\ \_ \to \_\\
&\qquad\qquad\quad acup'\ item\ =\ item
\end{aligned}
$$

In both cases array items are initialised to the empty string [], but initialising expression is not limited to a constant in general. Omitting the styles in which the applications are bound to form a program in the whole, as well as the form of derivable types in the type definition of $acup'$, there is no difference in manipulating the static array by application of *acup* and the dynamic array by application of $acup'$.

Concluding, the following explicit and implicit environments are recognized:

- Explicit environment for values
- Explicit environment for static arrays
- Explicit environment for structured data
- Implicit environment formed by a dynamic array
- Implicit environment formed by structured data.

Furthermore, except that explicit environments are appropriate to be bound to physical architecture resources, they form object environment, as shown in [13].

Notice also that sharing a subset of explicit environments by multiple modules is a perfect basis for the implementation of a modular language systems.

The description of architecture dependent, object and modular $\mathcal{PFL}$constructs is over the scope of this paper.

At the present time, object oriented $\mathcal{PFL}$ is bound to MPI — a message passing interface [18, 19]. In particular, extending explicit environment concept to implicit one enables exploiting MPI routines such as MPI BSend (buffered send). The physical notion of memory in $\mathcal{PFL}$ is necessary for exploiting new MPI datatypes, such as contiguous, strided vector, indexed and structure datatypes. Object paradigm is necessary to solve some imbalances introduced in MPI standard [15].

5 CONCLUSION AND FURTHER WORK

A $\mathcal{PFL}$ environmental concept is the generalized implementation substance, which occurs in programming languages exploiting large variety of programming paradigms. The language structure is ballanced with respect of giving strong boundary separating the specification, which is associated with a problem, and implementation, which is associated with the architecture resources while a program is executed.

In this paper, the concepts of explicit and implicit environments are unified.

No implicit environment for values can be defined, since lambda variables are values, not environment variables. On the other hand, a lambda variable is well defined, if it is defined partially.

That is why imperative records and functional tuples are semantically unified using a single concept of $\mathcal{PFL}$ tuples; each data structure may be partially defined, not however just for the reason that an item was not yet evaluated, as it is in purely functional languages. For example, the value of $(1, ())$ is a pair $(1, \perp)$, i.e. with the second item undefined. The construction of partially defined values corresponds to an imperative programming paradigm. Using process functional paradigm, the use of undefined values may be checked far simpler and it is far more tightly bound to source program than it is in an imperative language, since $\mathcal{PFL}$ program is an expression. The $\mathcal{PFL}$ ability for variant records is illustrated in Example 3.1, where the values of algebraic type C are stored in the same memory cell of external environment.

$\mathcal{PFL}$ arrays are extensional partial processes, defined as mappings from space to values that either exist (static arrays) fully undefined or they are created (dynamic arrays) either fully undefined or fully defined. The definition of arrays may change during computation.

The future is to use $\mathcal{PFL}$ in the role of a high level, typed and still non-deterministic implementation language which semantics is determined by the additional specification. The sequencing of updates and/or accesses, which guarantees the required level of determinism is the main problem. Usually the sequencing mechanism is defined by appropriate constructs incorporated into a general purpose programming language, fixing its semantics, either combining multiple paradigms as it is done in Oz [4, 20, 24], or determining the evaluation strategy explicitly in imperative and/or functional languages.

However, deriving the "most appropriate" sequences of "process application firings" by additional specification seems to be more flexible. Moreover, as we hope, it might result in approved implementation of the systems. Our idea comes out from the results reached in [6, 7]. The analogies between statically structured Petri nets and dynamically evolving control driven dataflow nets can be found. These are promising for static reasoning about *the evolution of computation*, which would provide the approved profile for the systems [14], i.e. such which guarantees the functional correctness as well as their expected behavior, minimizing at the same time the computational overheads.

REFERENCES

[1] Abelson, H. et al.: Revised Report on the Algorithmic Language Scheme. Higher-Order and Symbolic Computation, Vol. 11, 1998, pp. 7–105.

[2] Achten, P.—Plasmeijer, R.: Interactive Functional Objects in Clean. In: Clack et al. (Ed.): IFL'97, LNCS 1467, 1998, pp. 304–321.

[3] Achten, P.—Plasmeijer, R.: The Implementation of Interactive Local State Transition Systems in Clean. In: Koopman et al. (Ed.): IFL'99, LNCS 1868, 1999, pp. 115–130.

[4] Henz, M.: Objects for Concurrent Constraint Programming. The Kluwer International Series in Engineering and Computer Science, Volume 426. Kluwer Academic Publishers, Boston, 1998.

[5] Hudak, P.: Mutable abstract datatypes – or – How to have your state and munge it too. Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-914, December 1992, revised May 1993.

[6] Hudák, Š.—Teliopoulos, K.: Formal Specification and Time Analysis of Systems. Proc. of International Scientific Conf. ECI'98, Košice-Herľany, Slovakia, October 8–9, pp. 7–12.

[7] Hudák, Š.—Teliopoulos, K.: Loop Spectral Analysis of Time Reachability Problem. Proc. the 2-nd Int. Conf. RSEE'98, Oradea, Romania, May 27–30, 1998, pp. 51–61.

[8] Jones, M. P.: An Introduction to Gofer — Functional programming environment. Version 2.20, draft, 1991.

[9] Kollár, J.: Process Functional Programming. Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic, April 27–29, 1999, pp. 41–48.

[10] Kollár, J.: PFL Expressions for Imperative Control Structures, Proc. Scient. Conf. CEI'99, October 14–15, 1999, Herľany, Slovakia, pp. 23–28.

[11] Kollár, J.: Control-driven Data Flow, Journal of Electrical Engineering, Vol. 51, 2000, Nos.3–4, pp. 67–74.

[12] Kollár, J.: Comprehending Loops in a Process Functional Programming Language. Computers and Artificial Intelligence, Vol. 19, 2000, pp. 373–388.

[13] Kollár, J.: Object Modelling Using Process Functional Paradigm. Proc. ISM'2000, Rožnov pod Radhoštěm, Czech Republic, May 2–4, 2000, pp. 203–208.

[14] Kollár, J.—Porubän, J.: Static Evaluation of Process Functional Programs. Proceeding of the 6-th International Conference: Engineering of Modern Electric '01 Systems, Oradea, Romania, May 24–26, 2001.

[15] Kollár, J.—Václavík, P.: The Layout of MPI Routines in Object Oriented Process Functional Language. Proceeding of the 6-th International Conference: Engineering of Modern Electric '01 Systems, Oradea, Romania, May 24–26, 2001.

[16] launchbury, J.—Peyton Jones, S. L.: Lazy Functional State Threads. Computing Science Department, Glasgow University, 1994, 17 pp.

[17] Milner, R.—Tofte, M.—Harper, R.—MacQueen, D.: The Definition of Standard ML — Revised. The MIT Press, May 1997, 128 pp.

[18] MPI: A Message-Passing Interface Standard. Message Passing Interface Forum, June 12, 1995, University of Tennessee, Knoxville, Tennessee, 231 pp.

[19] MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum, July 18, 1997, University of Tennessee, Knoxville, Tennessee, 362 pp.

[20] Paralič, M.: Mobile Agents Based on Concurrent Constraint Programming. Joint Modular Languages Conference, JMLC 2000, September 6–8, 2000, Zurich, Switzerland. In: Lecture Notes in Computer Science, 1897, pp. 62–75.

[21] Peyton Jones, S. L.—Wadler, P.: Imperative Functional Programming. In 20th Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993, pp. 71–84.

[22] Peyton Jones, S. L.—Hughes, J. (Ed.): Report on the Programming Language Haskell 98 — A Non-strict, Purely Functional Language. February 1999, 163 pp.

[23] Reade, C.: Elements of Functional Programming. Prentice-Hall, 1989.

[24] Smolka, G.: The Oz Programming Model. In Jan van Leeuwen (Ed.), Computer Science Today, Lecture Notes in Computer Science 1000, Springer-Verlag, Berlin, 1995, pp. 324–343.

[25] Vokorokos, L.: Data Flow Computing Model: Application for Parallel Computer Systems Diagnosis. Computing and Informatics, Vol. 20, 2001, pp. 411–428.

[26] Wadler, P.: The Essence of Functional Programming. In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, draft, 23 pp.

[27] Wadler, P.: The Marriage of Effects and Monads. In ACM SIGPLAN International Conference on Functional Programming, ACM Press, 1998, pp. 63–74.

Ján Kollár(Assoc. Prof.) was born in 1954. He received his MSc. summa cum laude in 1978 and his PhD. in Computing Science in 1991. In 1978-1981 he was with the Institute of Electrical Machines in Košice. In 1982-1991 he was with the Institute of Computer Science at the P. J. Šafárik University in Košice. Since 1992 he is with the Department of Computers and Informatics at the Technical University of Košice. In 1985 he spent 3 months in the Joint Institute of Nuclear Research in Dubna, Soviet Union. In 1990 he spent 2 month at the Department of Computer Science at Reading University, Great Britain. He was involved in the research projects dealing with the real-time systems, the design of (micro) programming languages, image processing and remote sensing, dataflow systems, educational systems, and the implementation of functional programming languages. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming.