Acta Univ. M. Belii
Math no. 10(2003), pp. 53–64

THE CLASSIFICATION OF PROGRAMMING ENVIRONMENTS

Ján Kollár, Peter Václavík and Jaroslav Porubän

Abstract. A process functional paradigm prevents the use of assignments in programs, at the same time providing full power of both functional and imperative languages to a programmer. $\mathcal{PFL}$ – an experimental process functional language, originally developed as a programming language, seems to be promising to integrate the implementation requirements for any language aimed to von Neumann computer architectures. As we hope, $\mathcal{PFL}$ may serve as a unified implementation language in the future. That is why the formalized definitions of environments presented in this paper are useful and constructive for further development of $\mathcal{PFL}$ as a minimal superset of programming languages currently being used in a practice. In particular, we will classify the environments dividing them into two basic categories – external and internal environments, that may be defined in any scope of a program. Then we extend the notion of explicit and implicit environments to object and modular environments. Finally, we formulate the requirements for safe programming, which prevents the use of undefined values in programs.

Introduction

In the past, except well known imperative languages such as Pascal, C, Modula and Ada we have analyzed the languages that combines imperative and functional paradigms to exploit the benefits of both – the ability to manipulate the state preserving at the same time functional semantics, such as SML [13], Scheme [1], Clean [2] and Haskell [12,17,18], that either exploit the environments explicitly or hide them to a programmer [20,21]. Our aims were strongly practical – to develop a programming language that integrates the ideas promising to correct programming [14,15] and predicting the behavior of the systems [4,9] statically.

$\mathcal{PFL}$ – an experimental process functional language [5,6,7,8,9,10,11,19] was originally developed as a general purpose programming language based on a semantically minimal superset of known programming paradigms, such as imperative and functional, modular and object-oriented, deterministic and non-deterministic, sequential and parallel, message passing and shared memory, etc. At the same time, the goal was to provide a minimal set of syntactic constructs to a programmer.

2000 Mathematics Subject Classification. 68N15, 68N18.

Key words and phrases. Programming paradigms, applicative programming, process functional programming, explicit and implicit environments, object and modular environments, environmental application, referential transparency, side effects.

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

Received 10. 12. 2002; Accepted 16. 4. 2003

Syntactically, $\mathcal{PFL}$ is an extended subset of Haskell. Semantically, $\mathcal{PFL}$ integrates both functional and imperative programming paradigms, not however in monadic manner [20,21].

There are no assignments in $\mathcal{PFL}$ expressions provided to a user. All imperative actions are performed by applications of processes. On the other hand, all what is done is seen. It means that the state can be monitored even using graphic interface strongly bound to the source specification, visualizing the flow of data in environments. We do not want to hide the imperative actions to a user making them functional; we just want to separate them from functional grains, to provide the transparent and strong feedback to a user about mapping a problem to target architecture.

At present time, it seems that $\mathcal{PFL}$ is promising to be used as a unified implementation bridge between specification languages and computer architectures, since of its high abstraction coming out from the functional basis from one side, and its ability to affect computer architecture resources directly from the other side. Using $\mathcal{PFL}$, neither a target imperative language, such as C in pure functional languages able to manipulate state [17], nor a core language as for constrained functional languages [16] is required. Instead of that, the machine code can be generated directly.

However, $\mathcal{PFL}$ as a unified implementation language needs more detailed formalization of effects that arise from the state manipulation, since they affect the function of computation. This paper is not devoted to reduction strategies and/or the restrictions that must be taken into account when deterministic computation is considered. In this paper we concentrate just on the classification of environments manipulated in $\mathcal{PFL}$ that may be found in programming languages in general, and we will show, how they are affected, using process functional paradigm, which is materialized in $\mathcal{PFL}$.

We will define explicit and implicit environments first. Then we will use them to define object and modular environments. For the purpose of understanding the definitions of environments, we introduce both basics of process functional paradigm as well as a brief overview of $\mathcal{PFL}$ language constructs, related however just to the definitions of environments, less to the semantics of the language or its implementation.

As a result, we formulate the requirements for safe programming with respect to further development of $\mathcal{PFL}$. The definitions of programming environments introduced in this paper are useful, since they show the danger coming out from an undisciplined programming using imperative languages, and they are constructive, since they show the way, in which the use of undefined values can be prevented, still preserving full power of imperative languages. As we hope, this is solved systematically an transparently using process functional paradigm. On the other hand, integrating functional and imperative programming paradigms into the process functional paradigm is just the first step towards removing the gap between the specification and implementation.

Process Functional Paradigm

The form of a single argument "pure" function f in $\mathcal{PFL}$ script is as follows

$$f :: T_x \to T_e$$
$$f\ x = e$$

where the first line contains the type definition (TD), (sometimes called type signature) for f, expressing that f is a mapping from the values of the type T_x to the values of the type T_e. The second line contains the definition (D) of the function f; f is defined for each argument x by the expression e. Since a $\mathcal{PFL}$ function is just a specific process, which has no environment variable in TD, we will use a "pair" $PD = (TD; D)$ to designate a process definition PD including function definition. Then, whenever appropriate, instead of two lines of the definition above we will write PD in the form

$$PD = (f :: T_x \to T_e;\ f\ x = e)$$

Further, to minimize the space, we will use the form

$$PD;\ \ldots ; PD;$$

in which semicolons represent invisible newline characters causing all PD's above are indented to the same column given by first PD in the sequence.

The static semantics for the application ($f\ m$) is expressed by application rule, as follows:

(1)
$$\frac{f :: T_x \to T_e \qquad m : T_x}{f\ m :\ T_e}$$

According to (1), provided that f is of the type $T_x \to T_e$, and expression m is of the type T_x, then f is applied correctly to m, and then the application ($f\ m$) is of the type T_e.

The dynamic semantics of the application ($f\ m$) using semantic function $\mathcal{E}val$ and lambda form ($\lambda x.\ e$) for f is as follows,

(2)
$$\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!] = e[m/x]$$

where $e[m/x]$ (the value of application) is the expression e in which each occurrence of the lambda variable x is substituted by the expression m, or by the value of the expression m, depending on the reduction strategy.

The notion of variables in a purely functional language is mathematical; it means that all variables, such as lambda variable x or function f are values.

On the other hand, a variable in imperative languages is a memory cell used to store values. Process functional paradigm integrates both meanings using the theory of mutable abstract types [3]: process functional variable is a cell v containing (defined) data value $m \neq \perp$ (then we write $v[m]$), or undefined value $\perp$ ($v[\perp]$). At the same time, an environment variable v is an overloaded mapping, as follows.

(3)
$$v :\ \widetilde{T} \to T$$

where T is a data type and $\widetilde{T} = T \cup ()$ where () is the unit type.

The value of the application ($v\ m'$) depends on the argument m' as well as on the value m having been stored in the environment variable before.

According to definition 1, if ($m : ()$), then ($v\ m'$) is the access.

Definition 1.

$$(1.1)\qquad \frac{v[m] \vdash v : \widetilde{T} \to T \qquad m' : \ ()}{v\ m' : \ T \vdash m' : T}$$

where $(v\ m') = m'$.

$$(1.2)\qquad \frac{v[\bot] \vdash v : \widetilde{T} \to T \qquad m' : \ ()}{v\ m' : \ \Omega \vdash \bot : \Omega}$$

where $(v\ m') = \bot$.

According to definition 2, if $(m' : T)$, then $(v\ m')$ is the update of environment variable.

Definition 2.

$$(2.1)\qquad \frac{v[m] \vdash v : \widetilde{T} \to T \qquad m' : \ T}{v\ m' : \ T \vdash v[m'] \vdash m' : T}$$

where the state transition is $v[m] \Rightarrow v[m']$ and $(v\ m') = m'$.

$$(2.2)\qquad \frac{v[\bot] \vdash v : \widetilde{T} \to T \qquad m' : \ T}{v\ m' : \ T \vdash v[m'] \vdash m' : T}$$

where the state transition is $v[\bot] \Rightarrow v[m']$ and $(v\ m') = m'$.

Clearly, case (1.1) is the access of a data value, case (1.2) is the unwanted access of undefined value (prevented by the type checking), case (2.1) is the update called modification and case (2.2) is the update called initialization of the environment variable by the value m'. Hence, well-typed cases are (1.1), (2.1) and (2.2).

The dynamic semantics of the application $(v\ m')$ can be easily derived from the definitions 1 and 2. Using $\mathcal{E}val$, it is as follows:

Definition 3.

$$\begin{array}{ll}
(1.1) & \mathcal{E}val[\![\ v\ m'\ ]\!]\ v[m] = \ \mathcal{E}val[\![\ m\ \ ]\!]\ v[m], \quad \text{if } m' : () \\
(1.2) & \mathcal{E}val[\![\ v\ m'\ ]\!]\ v[\bot] = \ \mathcal{E}val[\![\ \bot\ \ ]\!]\ v[\bot], \quad \text{if } m' : () \\
(2.1) & \mathcal{E}val[\![\ v\ m'\ ]\!]\ v[m] = \ \mathcal{E}val[\![\ m'\ ]\!]\ v[m'], \quad \text{if } m' : T \\
(2.2) & \mathcal{E}val[\![\ v\ m'\ ]\!]\ v[\bot] = \ \mathcal{E}val[\![\ m'\ ]\!]\ v[m'], \quad \text{if } m' : T \\
 & \mathcal{E}val[\![\ \ m\ \ \ ]\!]\ v[m] = \ \mathcal{E}val[\![\ m\ \ ]\!] \\
 & \mathcal{E}val[\![\ \ \bot\ \ \ ]\!]\ v[\bot] = \ \bot
\end{array}$$

Since $(v\ m')$ may affect the state of computation by the side effect, this application is rather environmental than functional.

In $\mathcal{PFL}$ however, it is possible to perform environmental applications just indirectly via applications of processes. This approach prevents an undisciplined use of assignments, since there is no decision left to a programmer where to apply an environment variable in an expressions. It means that no environment variable occurs in source $\mathcal{PFL}$ expressions. Instead of that, environment variables are introduced in processes type definitions. This guarantees the systematic and disciplined use of hidden assignments in programs.

As shown below, it is the matter of the translation, to "remove" environment variables from type definitions and to "bring" them into expressions.

We will use mathematical form for $\mathcal{PFL}$ constructs, not numbering them. In this form we use $\rightarrow$, and $\Rightarrow$ instead of `->` and `=>`. We also use array curly brackets in the form {| and |} instead of { and } to precede the confusion with set brackets.

A $\mathcal{PFL}$ program consists of a set of multi-argument process definitions PD and main expression e in global scope $s = 0$, as follows,

$$\begin{array}{l}\textbf{program } M \textbf{ where}\\ PD;\ \ldots ; PD;\\ \textbf{main} = e\end{array}$$

where PD is a pair consisting of a type definition TD and the definition D, $PD = (TD; D)$

The form of PD is as follows.

$$\begin{array}{llll} f :: \overline{T_1} \rightarrow \ldots \rightarrow \overline{T_n} & & & \rightarrow \widetilde{T}\\ f\ \ p_1\ \ \ldots\ \ p_n & = & e & \\ & & \textbf{where} & \\ & & \quad PD;\ \ldots ; PD; & \end{array}$$

where PD's that follow keyword **where** are local processes definitions.

Using BNF, the syntax of type expressions is as follows:

$$\begin{array}{lll} \overline{T} & ::= & v\ T \mid v\ \{|R|\}\ T \mid \widetilde{T}\\ \widetilde{T} & ::= & () \mid T\\ T & ::= & a \mid T^P \mid T \rightarrow\ T \mid \{|R|\} \rightarrow T \mid T^D\ T_1 \ldots T_r \mid Cl\ T_1 \ldots T_u\\ \{|R|\} & ::= & \{|T_1^R, \ldots , T_d^R|\}, d \geq 1 \end{array}$$

where $r, u \geq 0$, v is an environment variable, $\{|R|\}$ is n-dimensional range, $()$ is unit type, a is a type variable, T^P is primitive type $(Char, Int, Float)$, $T \rightarrow\ T$ is function type, $\{|R|\} \rightarrow T$ is array type, T^D is algebraic data type, Cl is a class name and 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, such that $c_i^L, c_i^U : T_i^R$.

Attention must be paid to the type expressions $v\ T$ and $v\ \{|R|\}\ T$, since they are just syntactic shortcuts, which serve during compilation

- to derive the environment variable types $v : \widetilde{T} \rightarrow T$ and $v : \{|R|\} \rightarrow \widetilde{T} \rightarrow T$, transforming both to T in target process f type definition, and
- transforming each expression as follows:

– Provided that m is a process argument of source type $v\ T$, then it is translated into the form of environmental application $(v\ m)$, of the type T.

– Provided that $(\{|e^R|\}m)$, (e^R is an index expression) is a process argument argument of source type $v\ \{|R|\}\ T$, then it is translated into the form $((v\ \{|e^R|\})\ m)$ which is of the type T again. Here $(v\ \{|e^R|\})$ selects a cell representing the item of an array, and then this cell – being the environment variable – is applied to m.

According to the transformation above, the source $\mathcal{PFL}$ script comprising environment variables just in type definitions is translated to an intermediate form that comprises the environment variables just in expressions.

Formal parameters p_i are either in the form of simple lambda variable x or in the form of patterns $(C\ p_1\ \ldots p_m)$ or $x@(C\ p_1\ \ldots p_m)$, such that $m \geq 0$, and C is a constructor of algebraic type. The form $x@(C\ p_1\ \ldots p_m)$ is extremely useful, since it allows updating the items of the structure $(C\ p_1\ \ldots p_m)$ in place returning its incoming value x.

$\mathcal{PFL}$ algebraic types are defined using data definitions, in the form as follows:

$$\begin{aligned}\textbf{data}\ T^D\ a_1 \ldots a_u = C_1\ T_{1,1} \ldots T_{1,n_1} \\ \mid C_2\ T_{2,1} \ldots T_{2,n_2} \\ \ldots\ldots\ldots\ldots \\ \mid C_m\ T_{m,1} \ldots T_{m,n_m}\end{aligned}$$

where a_k are type variables, $T_{i,j}$ are type expressions and C_i are constructors of algebraic type T^D. For our purposes it is sufficient to consider just product types $(m = 1)$ an then we may write the definition above in the form

$$\textbf{data}\ T^D\ a_1 \ldots a_u = C\ T_1 \ldots T_n$$

Provided that T^D is defined, and expressions $m_1, \ldots, m_n$ are such that $m_1 : T_1$, $\ldots$, $m_n : T_n$ holds, then

$$(C\ m_1 \ldots m_n)$$

is used in an expression to construct n-tuple of items $m_1, \ldots, m_n$, such that it is of monotype $T^D\ T_1^M \ldots T_u^M$, $T^D\ T_1^M \ldots T_u^M \subset T^D\ a_1 \ldots a_u$.

A dynamic array in $\mathcal{PFL}$ is created by an expression, called array creator, as follows.

$$\{\!|R^F|\!\} \rightarrow m$$

used in an expression, where $\{\!|R^F|\!\}$ is a finite subrange, $\{\!|R^F|\!\} \subseteq \{\!|R|\!\}$ and $m : \widetilde{T}$. If $m : T$, then $\{\!|R^F|\!\} \rightarrow m$ is the array of items, each initialized to m.

$\mathcal{PFL}$ type synonyms TS are defined using type definitions, in the form as follows

$$\textbf{type}\ TS\ a_1 \ldots a_u = \overline{T}$$

that use is appropriate especially when an environment variable v is shared by different processes, but also when a variable is associated with a memory address (for example 177746), such as follows

$$\textbf{type}\ TS\ a_1 \ldots a_u = v\ T\ \textbf{at}\ \#177746$$

Abstract type is implemented using class definition and corresponding set of instance definitions, in the form as follows,

$$\textbf{class } Cl\ a_1 \ldots a_u \textbf{ where } TD;\ \ldots ; TD;$$

$$\textbf{instance } Cl\ T_1 \ldots T_u \textbf{ where } D;\ \ldots ; D;$$

such that for each instance of a class Cl holds $(Cl\ T_1 \ldots T_n) \subset (Cl\ a_1 \ldots a_n)$. If a class is monomorphic, then it contains no type variables a_i $(u = 0)$ and then it is possible to define just one instance for this class.

Provided that TD's in a class definition contains at least one environment variable, i.e. this environment is not empty, new object for an instance of this class is created using the type expression in expressions as follows

$$Cl\ T_1 \ldots T_u$$

Then, if m is an expression of the type $Cl\ T_1 \ldots T_u$ and a process f is defined by D in **instance** $Cl\ T_1 \ldots T_u$, then f is selected using expression

$$(m \Rightarrow f)$$

which, when applied, affects the environment given by m. It may be noticed that if a class environment is empty, then it exists just one object for each instance, hence, in this case we have no opportunity for object programming using such a class.

It is easy to see, that modularity may be implemented in a similar manner. Except a main module, formed by program the additional set of modules can be defined each of them in the form

$$\textbf{module } M \textbf{ where } PD;\ \ldots ; PD;$$

Then a process f defined in module M is accessible in other modules using the expression

$$(M \Rightarrow f)$$

Let the global scope in each module and each object is the same as the global scope in the program, i.e. $s = 0$, and local scopes are such that $s > 0$. Provided that $PD = (TD; D)$ such that $v \in TD \wedge f \in D$, then both f and v are in the same scope. The names of all processes in the same scope s must be unique, and any environment variable name v in this scope must differ from the names of processes. On the other hand, $v \in TD_1 \wedge v \in TD_2$ means sharing this variable by processes defined by D_1 and D_2, which is allowed.

The scope boundary, as well as the visibility of lambda and pattern variables and process names is the same as in Pascal; if f is in the scope s then its formal parameters (lambda and pattern variables) and the names of local processes (as well as its body) are in the scope $s + 1$; if a name used in a scope s_3 is introduced in scopes s_1 and s_2 $(s_1 < s_2 < s_3)$ not however in scopes s, $s_2 < s \leq s_3$, then the used name is such that introduced in the scope s_2.

To prevent the obscure notation, we will consider just single argument processes f and g, such that f is defined in a scope s and g is defined in a scope s', $s' \geq s$.

Considering a relation between environments in scopes s and s' in a program, we distinct two essential kinds of environments:

- Explicit environment – formed introducing new names for environment variables that are different from the names used in patterns
- Implicit environment – formed using names introduced in patterns.

This means, that an environment variable in the scope s may belong to the explicit environment $E^{(s)}$, only if it does not belong to the implicit environment $I^{(s)}$.

An explicit environment for values is defined according to definition 4.

Definition 4. For all processes f in the scope s, defined by

$$PD = (f :: v\ T \to \widetilde{T_f};\ f\ x\ =\ e_f)$$

where T is primitive monotype or function polytype or dynamic array polytype, it holds:

- Lambda variable x is *the value* of v accessed in e_f by x, where $x : T$.
- If $\{v :\ \widetilde{T} \to T\} \not\subseteq I^{(s)}$ then $\{v :\ \widetilde{T} \to T\} \subseteq E^{(s)}$.
- Then, provided that g is a process defined by

$$D = (g\ y\ =\ e_g)$$

in the scope s', $s' \geq (s-1)$, such that f is accessible in e_g, an environment variable v is accessed/updated in e_g by application $(f\ m)$, translated to $f\ (v\ m)$, where v is *the address* and $m :\ \widetilde{T}$.

An explicit environment for static arrays is defined according to the definition 5.

Definition 5. For all processes f in the scope s, defined by

$$PD = (f :: v\ \{\!|R^F|\!\}\ T \to \widetilde{T_f};\ f\ x\ =\ e_f)$$

where T is polytype, $\{\!|R^F|\!\}$ is finite subrange $\{\!|R^F|\!\} \subseteq \{\!|R|\!\}$, it holds:

- Lambda variable x is *the value* of v $\{\!|e^R|\!\}$ ($\{\!|e^R|\!\}$-th item of the array v), $\{\!|e^R|\!\} \in \{\!|R^F|\!\}$, accessed in e_f by x, where $x : T$.
- If $\{v : \{\!|R|\!\} \to\ \widetilde{T} \to T\} \not\subseteq I^{(s)}$ then $\{v : \{\!|R^F|\!\} \to\ \widetilde{T} \to T\} \subseteq E^{(s)}$
- Then, provided that g is a function or a process defined by

$$D = (g\ y\ =\ e_g)$$

in the scope s', $s' \geq (s-1)$, such that f is accessible in e_g, and each environment variable – the $\{\!|e^R|\!\}$-th item of the array v is accessed/updated in e_g by application $f\ (\{\!|e^R|\!\}\ m)$, translated to $f\ (v\ \{\!|e_R|\!\}\ m)$, v is *the address*, where $m :\ \widetilde{T}$.

An explicit environment for algebraic structured data is defined according to the definition 6.

Definition 6. For all processes f in the scope s, defined by

$$PD = (f :: \ v \ T^D \rightarrow \widetilde{T_f}; \ f \ p \ = \ e_f)$$

where $p ::= x \mid x@(C \ x_1 \ \ldots x_n) \mid (C \ x_1 \ \ldots x_n)$, T^D is algebraic monotype, C is a constructor ($C :: \ T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T^D$), it holds:

- Lambda variable x is *the address* of v. Then x, x_1, ... , x_n, are used in e_f as *values*, where $x : T$, $x_1 : T_1$, ... , $x_n : T_n$.
- If $\{v : \widetilde{T^D} \rightarrow T^D\} \not\subseteq I^{(s)}$ then $\{v : \widetilde{T^D} \rightarrow T^D\} \subseteq E^{(s)}$.
- Then, provided that g is a function or a process defined by

  $$D = (g \ y \ = \ e_g)$$

  in the scope s', $s' \geq (s-1)$, such that f is accessible, an environment variable v is accessed/updated in e_g by application $f \ (C \ m_1 \ \ldots \ m_n)$, translated to $f \ (v \ (C \ m_1 \ \ldots \ m_n))$, v is *the address*, where $m_i : \ \widetilde{T}_i$, for $i = 1 \ldots n$.

The extension to algebraic polytype is straightforward – by the substitution of T^D by type application $(T^D \ T_1^a \ldots T_u^a)$, where T_u^a are type expressions comprising type variables.

According to the definition 5, static arrays belong to an explicit environment and they cannot be used in a higher order manner. According to the definition 6, static data structures such as Pascal records in global memory or on the stack are manipulated.

Opposite to the explicit environment, the implicit environment is defined considering that the items of data structures and arrays are values, but at the same time they are cells, used as implicit environment variables for local processes. Since non-structured values are represented by lambda variables that are values, not cells, it follows that they cannot form an implicit environment.

An implicit environment formed by a lambda variable of a dynamic array type is defined according to definition 7.

Definition 7.

Let f is a process or function in the scope s, defined by

$$D = (f \ x \ = \ e_f)$$

(type definition is out of interest here), such that $x : \{\!|R|\!\} \rightarrow T$, where T is a polytype, and R is infinite range.

Let us consider the application $(f \ (\{\!|R^F|\!\} \rightarrow m))$, such that $R^F \subseteq R$ and $m : T$. Then it holds:

- Lambda variable x is a dynamic array value and $x\{\!|e^R|\!\}$ the array item value, both accessible in e_f, $x : \{\!|R^F|\!\} \rightarrow T$ and $x\{\!|e^R|\!\} : T$.
- $\{x : \{\!|R|\!\} \rightarrow \widetilde{T} \rightarrow T\} \subseteq I^{(s')}$, $s' > s$, provided that $\exists \ g$ in the scope s', defined by

  $$PD = (g :: \ x \ \{\!|\!\} \ \_ \rightarrow \widetilde{T_g}; \ g \ y \ = \ e_g)$$

  where $x \ \{\!|\!\} \ \_ = x \ \{\!|R|\!\} \ T$.
- Then, $\{\!|e^R|\!\}$-th item of a dynamic array x is accessible/updatable in scopes s'', $s'' \geq s'$, if g is accessible, by the application $g \ (\{\!|e^R|\!\} \ m')$, translated to $g \ (x \ \{\!|e^R|\!\} \ m')$, where x is *the value*, and $m' : \ \widetilde{T}$.

A possible type definition for a dynamic array argument of f in the definition 7 is $(\{\!|R|\!\} \to T)$, or $v\ (\{\!|R|\!\} \to T)$, where v is an explicit environment variable, and $v\ (\{\!|\,|\!\} \to \_)$, if v is an implicit environment variable.

An implicit environment formed by a lambda variable of an algebraic structured data type is defined according to definition 8.

Definition 8.

Let f is a process or a function in the scope s, defined by

$$D = (f\ (C\ x_1\ \ldots x_n)\ =\ e) \quad \text{or} \quad D = (f\ x@(C\ x_1\ \ldots x_n)\ =\ e)$$

such that $x : T^D\ T_1^a \ldots T_u^a$, $x_1 : T_1$, $\ldots$, $x_n : T_n$, $x : T$, $C : T_1 \to \cdots \to T_n \to T^D\ T_1^a \ldots T_u^a$, T^D is an algebraic data polytype and T_1, $\ldots$, T_n are polytypes.

Suppose the application $f\ (C\ m_1\ \ldots\ m_n)$, where $m_i : T_i$. Then it holds:

- Lambda variable x is the data value and x_1, $\ldots$, x_n are its item values, accessible in e_f, $x : T^D\ T_1^a \ldots T_u^a$, and $x_i : T_i$, for all $x_i \in \{x_1, \ldots, x_n\}$.
- $\{x_i : \widetilde{T_i} \to T_i\} \subseteq I^{(s')}$, $s' > s$, provided that $\exists\ g$ in the scope s', defined by

$$PD = (g ::\ x_i\ \_\ \to \widetilde{T_g};\ g\ y\ =\ e_g)$$

  where $(x_i\ \_) = (x_i\ T_i)$.
- Then a structured data item x_i is accessible/updatable in e_f (including scopes s'', $s'' \geq s'$), if g is accessible, by the application $g\ m_i'$, translated to $g\ (x_i\ m_i')$, where x_i is *the address*, and $m_i' : \widetilde{T_i}$.

Definition 9.

Object environment is the explicit environment $E^{(0)}$, defined by a class definition in global scope ($s = 0$), and allocated by class application $Cl\ T_1 \ldots T_u$. Its size is dependent on argument types T_1, $\ldots$, T_u. Different objects have mutually disjunctive environments. All object environments are also disjunctive with explicit and implicit environments of a program and modules.

Definition 10.

Since a program is a main module of a modular sequential language, it is sufficient to consider just a set of modules $M_1, \ldots, M_p$, one of them being a program.

Let $E_{M_i}^{(0)}$ is an explicit environment in the global scope $s = 0$ defined by type definitions of global processes in module M_i. Let $(M_j \Rightarrow f_k)$ is a process f_k imported from module M_j being applied in module M_i. Then $\mathbf{I}_{M_i}^{(0)}$ is a set of subsets of explicit environments $E_{M_j}^{(0)}$ formed by environment variables affected by all imported processes, which is defined as follows.

$$\mathbf{I}_{M_i}^{(0)} = \{S \mid S \subseteq E_{M_j}^{(0)},\ v \in S \iff v \in TD, (TD(M_j \Rightarrow f_k), D(M_j \Rightarrow f_k)) \in M_j\}$$

where $TD(M_j \Rightarrow f_k)$ is type definition, and $D(M_j \Rightarrow f_k)$ is the definition of a process f_k in a module M_j.

Then modular environment for a module M_i is defined by the unification of all subsets of explicit environments affected by all imported processes applied in M_i and the explicit environment in global scope of module M_i.

$$\bigcup_{\forall(M_j \Rightarrow f_k) \in M_i} \mathbf{I}^{(0)}_{M_i} \cup E^{(0)}_{M_i}$$

Clearly, modular environments may be disjunctive, i.e. non-overlapping, but also shared.

Conclusion

In this paper, programming environments are defined in terms of $\mathcal{PFL}$ – a process functional language. We introduced the essence of process functional paradigm – programming by application of processes instead of assignments providing the ability for imperative semantics of $\mathcal{PFL}$ programs being described seemingly by a purely functional source script, since no environmental applications occur in source expressions. This approach, from one point of view, does not disqualify the language to the role of a purely functional but still macro language that affects architecture resources indirectly, using an underlying imperative target language. On the other hand, defining the environments in its abstracted nature, we may see a danger of the use of undefined values, inherent to each imperative language.

Without doubt, the implicit environments are initialized to defined values, since algebraic data are constructed by $(C\ m_1 \ldots m_n)$ and arrays are created by $(\{\!|R^F|\!\} \rightarrow m)$, using m_i and m that cannot be of unit type $()$. To preserve the safeness in the use of defined values of arrays we must guarantee that all items are defined also for arrays created using loop comprehensions [8] that are over the scope of this paper.

However, since explicit environments are not initialized as may be seen from their definitions, it is quite possible to use the undefined values when applying processes to expressions of unit type. Therefore, to guarantee the use of defined values in programs, all explicit environment variables, except those associated with a memory addresses we must think about their initialization. But this approach decreases the run-time efficiency, especially for local explicit environments ($s > 0$). Moreover, since modular environment for a module is formed by unification of global explicit environment of this module and all subsets of explicit environments affected by imported processes from other modules, the modular environment may be shared. Then, the initialization of shared environments possesses the question, which of potential initializations occurring in the different modules determines the initial state. Clearly, this question has no satisfactory answer for parallel modules, since multiple initialization of shared environments in parallel would result to non-deterministic initial state. On the other hand, it is reasonable to initialize object environments, since they are disjunctive. (Notice, that we do not need static methods, such as in C here, since our "static" processes are defined in program modules.)

At this point it may be seen the advantage of process functional approach: Since $\mathcal{PFL}$ programs are expressions, the use of undefined explicit environment variables are identifiable during the type checking, providing strong feedback to a user about incorrectness, since they are related to arguments of unit types used improperly in applications of processes.

Concluding, the safe programming is such that uses either initialized environments or provide source-to-target feedback about the use of undefined values that yield potential incorrect execution of programs.

References

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

[2] Achten P., and R. Plasmeijer, *The implementation of interactive local state transition systems in Clean, In: Koopman et al. (Ed.): IFL'99* **LNCS 1868** (1999), 115–130.

[3] 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; rev. May 1993).

[4] Hudák Š., and K. Teliopoulos, *Formal Specification and Time Analysis of Systems, Proc. of International Scientific Conf. ECI'98, Košice-Herľany, Slovakia* (October 8–9 1998), 7–12.

[5] Kollár J., *Process Functional Programming, Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic* (April 27–29, 1999), 41–48.

[6] Kollár J., *PFL Expressions for Imperative Control Structures, Proc. Scient. Conf. CEI'99, Herľany, Slovakia* (October 14–15, 1999), 23–28.

[7] Kollár J., *Control-driven Data Flow, Journal of Electrical Engineering, No.3–4* **51** (2000), 67–74.

[8] Kollár J., *Comprehending Loops in a Process Functional Programming Language, Computers and Artificial Intelligence* **19** (2000), 373–388.

[9] Kollár, J., and Porubän, J., *Static Evaluation of Process Functional Programs. Analele Universitatii din Oradea, Proc. of the 6'th Scientific Conference with International participation EMES'01 Engineering of Modern Electric Systems, Felix-Spa, Oradea, Romania* (May 24–26, 2001), 93–98.

[10] Kollár, J., Porubän, J., Václavík, P., and Vidiščak, M., *Lazy State Evaluation of Process Functional Programs, ISM'02, Proceedings of the Conference "Information Systems Modelling", Rožnov pod Radhoštěm, Czech Republic* (April 22–24, 2002), 165–172.

[11] Kollár, J., *Imperative Expressions using Implicit Environments. Proc of ECI'2002, the 5-th Int. Scient. Conf. on Electronic Computers and Informatics, Herľany, Slovakia* (Oct. 10-11, 2002,), 86–91.

[12] Launchbury J., and S.L. Peyton Jones, *"Lazy Functional State Threads", Computing Science Department, Glasgow University, 1994, 17 pages.*

[13] Milner R., Mads Tofte, Robert Harper, and David MacQueen, *The Definition of Standard ML - Revised*, The MIT Press, May 1997.

[14] Novitzká, V., *Systems for Deriving Correct Implementations, Proc. ISM'99, Rožnov pod Radhoštěm, Czech Republic* (April 27–29, 1999), 201–207.

[15] Novitzká, V., *Formal Foundations of Correct Programming*, elfa, Košice, Slovakia, 1999.

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

[17] Peyton Jones S.L., and P. Wadler, *Imperative functional programming, In 20th Annual Symposium on Principles of Programming Languages, Charleston, South Carolina* (January 1993), 71–84.

[18] Peyton Jones S.L., and John Hughes [editors], *"Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language"* (February 1999), 163 pages.

[19] Václavík, P., and Porubän, J., *Object Oriented Approach in Process Functional Language. Proc of ECI'2002, the 5-th Int. Scient. Conf. on Electronic Computers and Informatics, Herľany, Slovakia* (Oct. 10-11, 2002), 92–96.

[20] Wadler P., *The essence of functional programming, In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico (draft)* (January 1992), 23 pages.

[21] Wadler P., *The marriage of effects and monads, In ACM SIGPLAN International Conference on Functional Programming* (1998), ACM Press, 63–74.

Department of Computers and Informatics; Technical University of Košice; Letná 9; SK–041 20 Košice; Slovakia
E-mail: `Jan.Kollar@tuke.sk`