Computing and Informatics, Vol. 24, 2005, 441–462

SEPARATING CONCERNS IN PROGRAMMING: DATA, CONTROL AND ACTIONS*

Ján Kollár, Jaroslav Porubän, Peter Václavík

Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
Letná 9, 042 00 Košice, Slovakia
e-mail: `jan.kollar@tuke.sk`

Manuscript received 19 October 2004; revised 21 September 2005
Communicated by Gianfranco Rossi

Abstract. A multiparadigm language provides an opportunity to a user for exploiting more programming methodologies. It simplifies the language syntax, and extends the application areas by the extended semantics. That is why multiparadigm languages can align a problem in wider application areas and more flexibly than that based on a single paradigm. In this paper, we present the idea of separating three essential concerns of programming currently being implemented in $\mathcal{PFL}$ – a process functional language. We separate data, control, and actions by the definition of a purely control structure. Then, by the structured application of a structure of actions to a purely control structure, we will express the computation of activated actions in a structured way, considering explicitly defined synchronization in computation.

Keywords: Programming languages, process functional programming, AspectJ, aspect-oriented programming, $\mathcal{PFL}$, computational reflection, programming environments

* This work was supported by VEGA Grant No. 1/1065/04 – Specification and implementation of aspects in programming

1 INTRODUCTION

Nowadays, many programming paradigms, such as functional, imperative, structured, object-oriented, component based, and aspect-oriented ones, are less or more combined when exploited by a programmer. In this sense, the paradigm means the pattern or approach when expressing an algorithmic problem in terms of a program, which is executable on the computer.

A programming paradigm can be exploited better if supported by corresponding programming language. Multiparadigm languages provide an opportunity to exploit multiple programming paradigms. In this sense, even Pascal can be seen as a multiparadigm language, since it integrates imperative paradigm and structured paradigm. Similarly, C++ or Java are multiparadigm languages, since they combine imperative, structured and object oriented paradigm.

In Haskell [13], both functional and imperative paradigms [12] are combined by the definition of monads [20, 21], as a uniform tool for expressing imperative paradigm in terms of functional paradigm. It means that a source program is written in a purely functional manner, and all imperative actions are performed via monads, hiding actions to a programmer.

Combining two or more paradigms directly by an extension of an existing programming language, such as Java, when extending it to AspectJ [3, 4], seems to be less efficient method for building multiparadigm languages, since of the rare but still occurring situations, in which the enveloped language (such as Java) does not conform with the new paradigm [5].

That is why it is better to extract the substances of multiple paradigms, then to define a new multiparadigm pattern and finally to define a new multiparadigm language, as it was done in Haskell.

We use the same approach in $\mathcal{PFL}$ – a process functional language development, in which we have combined imperative, functional, and object oriented paradigms [6, 7, 8, 9]. It was performed by unification of paradigms, followed by $\mathcal{PFL}$ definition, and the compiler to Java and Haskell construction [17, 18, 19].

As a result, we have obtained a systematic hierarchy of processes, global and local scopes, classes, instances and objects, free of mixed language concepts, such as the definitions of processes in classes, or concepts dealing with organization (such as public, private or static modifiers in Java). This is a good proposition for a transparent selection of join point, considering lexical scope, as required in pointcuts in aspect-oriented languages [1, 2, 11, 10, 22].

Except that, each value (including those produced by built-in operations) could be reflected into an external environment [9], if needed. This fact is meaningful when considering dynamic aspects of computation [16].

Based on an experiment with profiling of process functional programs [14, 15], we understood that three basic concerns of computation – data, control, and actions – must be clearly separated before unifying object paradigm with aspect paradigm.

In this paper we present the idea of separating data, control and actions as a proposition for expressing all of them explicitly. In this way, as we believe, a program is expressed in an accurate and concise source form, which is substantial for extending $\mathcal{PFL}$ to an aspect-oriented language. The work on aspect version of $\mathcal{PFL}$ goes on. Some remarks on the importance and advantages of separated concerns in implementing aspect languages conclude the paper.

First, in Section 2, we briefly summarize the current state in the development of $\mathcal{PFL}$. In Section 3, we introduce the notion of purely control structures and purely control functions. Then we define the operation of structured application in Section 4, as an application of a structure of actions to a control structure, supporting the idea of computation in the separated manner. In the section Examples, we compare the classical imperative approach and the functional approach using the structured application.

For $\mathcal{PFL}$ programs, we will use well known mathematical notation, in which $\rightarrow$ is used instead of `->`, $[\,]$ is used instead of `[]`, etc. To express 'is of the type', we will use double colon (::).

2 $\mathcal{PFL}$ – CURRENT STATE

Using $\mathcal{PFL}$ – an experimental process functional language, the following transformations are supported: Data-to-Data (DD), Data-to-Control (DC), Control-to-Data (CD), Control-to-Control (CC), Data-to-Environment data-to-Data (DED), Data-to-Environment data-to-Control (DEC) Control-to-Environment data-to-Data (CED), and Control-to-Environment data-to-Control (CEC). These transformations are shown in Table 1, in terms of applications of a single argument function $\lambda x.\ e$ to an argument m. In this table, T and T' are data types, $()$ is the unit type, $\rho[v = e_v]$ is an initial environment, comprising the variable v of the value e_v, both of the data type T, $\rho[v = m]$ is an environment with variable v of the value m, $e[m/x]$ is an expression in which each occurrence of lambda variable x is substituted by an expression m or its value, $e[e_v/x]$ is an expression in which each occurrence of lambda variable x is substituted by an expression e_v or its value, and ρv is the current value in environment variable v.

In $\mathcal{PFL}$, type definitions for all processes are obligatory, not optional, as it is for functions. This is true even for local processes that are defined using the keyword *where*, as it is in Haskell.

The transformations DD, DC, CD and CC are performed by application of pure functions ($\lambda x.\ e$), as in any functional language, provided that function body e is purely functional. However, since of environment affecting transformations DED, DEC, CED and CEC, $\mathcal{PFL}$ semantics is fully imperative, preserving the pure functional source style based on expressions, without statements and assignments.

An environment affecting transformation is determined by a "strange" argument type ($v\ T$), which is the source form of attributed type. Then the environment variable v with the stored value e_v is a proposition of this transformation ($\rho[v = e_v] \vdash$).

DD	$\dfrac{(\lambda x.\ e) :: T \rightarrow T' \qquad m :: T}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!] = e[m/x]}$
DC	$\dfrac{(\lambda x.\ e) :: T \rightarrow () \qquad m :: T}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!] = e[m/x]}$
CD	$\dfrac{(\lambda x.\ e) :: () \rightarrow T \qquad m :: ()}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!] = e}$
CC	$\dfrac{(\lambda x.\ e) :: () \rightarrow () \qquad m :: ()}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!] = e}$
DED	$\dfrac{\rho[v = e_v] \vdash\ (\lambda x.\ e) :: v\ T \rightarrow T' \qquad m :: T}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ e)\ (\rho\ v)\ ]\!]\ \rho[v = m] = e[m/x]}$
DEC	$\dfrac{\rho[v = e_v] \vdash\ (\lambda x.\ e) :: v\ T \rightarrow () \qquad m :: T}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ e)\ (\rho\ v)\ ]\!]\ \rho[v = m] = e[m/x]}$
CED	$\dfrac{\rho[v = e_v] \vdash\ (\lambda x.\ e) :: v\ T \rightarrow T' \qquad m :: ()}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ e)\ (\rho\ v)\ ]\!]\ \rho = e[e_v/x]}$
CEC	$\dfrac{\rho[v = e_v] \vdash\ (\lambda x.\ e) :: v\ T \rightarrow () \qquad m :: ()}{\mathcal{E}val[\![\ (\lambda x.\ e)\ m\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ e)\ (\rho\ v)\ ]\!]\ \rho = e[e_v/x]}$

Table 1. Transformations supported by $\mathcal{PFL}$

Further, a kind of environment affecting transformation is determined by the type of argument. Provided that $m :: T$, then the transformation is data reflecting transformation, with either data (DED) or control (DEC) value. Provided that $m :: ()$, then the transformation is data accessing transformation, with either data (CED) or control (CEC) value.

Environment affecting transformations are implemented by source-to-source transformation, which, in terms of aspect-oriented languages, is a specific weaving considering each application of the process as a join point. The details of this transformation are introduced in Section 4.

Operationally, data affecting transformations are performed in two indivisible steps. An environment variable is either accessed by control argument or updated by data argument in the first step, and the value of environment variable is used as an argument in the application itself in the second step.

As an illustrating example, let us define a process p as follows:

$$\begin{aligned}&p\ ::\ v\ Int \rightarrow Int\\&p\ x = 2 * x.\end{aligned}$$

Suppose that the value of the environment variable v is 2, and let us consider the evaluation of the application $(p\ (p\ ()))$ in innermost order.

So we have $\rho[v = 2]$ as a proposition, and p in terms of typed lambda calculus as follows: $(\lambda x.\ 2 * x) :: v\ Int \rightarrow Int.$

Then $(p\ ()) = ((\lambda x.\ 2*x)\ ())$ is CED transformation as follows

$$\frac{\rho[v=2] \vdash\ (\lambda x.\ 2*x) :: v\ \mathit{Int} \to \mathit{Int} \qquad () :: ()}{\mathcal{E}val[\![\ (\lambda x.\ 2*x)\ ()\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ 2*x)\ (\rho\ v)\ ]\!]\ \rho = (2*x)[2/x] = 4}\ ,$$

which accesses the environment value $\rho v = 2$, yielding the result 4.

Notice that since of $(v\ \mathit{Int})$ argument type of p, the application $((\lambda x.\ 2*x)\ ())$ has quite different semantics than in an ordinary lambda calculus, since it is not reduced to $2*()$, but rather to $2*(\rho\ v) = 2*2$.

Since $p\ () = 4$, we have $(p\ (p\ ())) = (p\ 4) = ((\lambda x.\ 2*x)\ 4)$.

Then $((\lambda x.\ 2*x)\ 4)$ is DED transformation as follows:

$$\frac{\rho[v=2] \vdash\ (\lambda x.\ 2*x) :: v\ \mathit{Int} \to \mathit{Int} \qquad 4 :: \mathit{Int}}{\mathcal{E}val[\![\ (\lambda x.\ 2*x)\ 4\ ]\!]\ \rho = \mathcal{E}val[\![\ (\lambda x.\ 2*x)\ (\rho\ v)\ ]\!]\ \rho[v=4] = (2*x)[4/x] = 8}\ .$$

The application $((\lambda x.\ 2*x)\ 4)$ has the same value as when reduced directly, but value 4 of argument is reflected in v (see $\rho[v = 4]$ above). This semantics comes again from the attributed type $(v\ \mathit{Int})$.

Informally, CED and CEC use the value of v as an argument instead of the control (unit) value $()$, leaving this value unchanged. Transformations DED and DEC use the value of data argument, since this value is assigned to v before it is used in application.

As shown above, $\mathcal{PFL}$ conception is based rather on processes than on pure functions, since any grain of computation can be stateful. Considering a process of multiple arguments, data types T, the unit type $()$ and attributed types $(v\ T)$ may be freely mixed for arguments. Attributed types are never used for values of processes.

$\mathcal{PFL}$ supports type polymorphism using type variables $(a, b, \dots)$ in type expressions, algebraic types using *data* definitions, type synonyms using *type* definitions and abstract typing using **class** and *instance* definitions.

Our aim was to preserve the visibility of environment variables – the memory cells external to expressions in a program. That is why an environment variable is never used in expressions, but rather in type expressions. This is our mention that the visibility of memory cells is useful from the viewpoint of software engineering, and if supported by affecting them indirectly (this idea comes out from monadic approach), then their use is far more disciplined as in an imperative language.

This approach allows to exploit abstract typing in an object oriented manner. First, a $\mathcal{PFL}$ class C contains just the type definitions. All definitions of processes and functions are introduced in the instances, even if they are identical. In this matter we have separated the definitions of class members from their type definitions systematically.

Second, if type definitions in a class comprise environment variables using attributed types, and a class member is applied to expressions $e_1 \dots e_n$, in the form

($C \Rightarrow f\ e_1 \ldots e_n$), then the allocated environment together with all processes in derived instance form an object.

Moreover, an object can be shared by multiple objects since $\mathcal{PFL}$ supports multiple superclasses.

In case a class is monomorphic, then exactly one instance for this class is defined. If type definitions in a class comprise no environment variables, then the application of class members (or instantiated class members, in polymorphic case) does not allocate a memory. Then the situation is the same as in purely functional languages with abstract typing, in which overloaded functions are applied, but not in object-oriented manner.

We do not provide lambda abstraction ($\backslash x \to e$) to a programmer. From one point of view, abstracting of any expression is a powerful Haskell concept; on the other hand, its extensive use makes programs non-transparent.

We mention that the transparency of the source program is the proposition for transparent lexical scoping as required for aspect-oriented programming languages. That is why separating the definitions of processes from classes seems to be a good idea.

On the other hand, mixing the transformations freely is the weakness of both current implementation of $\mathcal{PFL}$ as well as of any imperative language (in which they are mixed even in a more hidden way). In this paper, we propose the structured application operation as a possible solution to this problem. This approach is less general than the monadic one, but tends to the separation of data and control in programs in a transparent way.

Since data manipulation using pure functions is a well known concept in purely functional languages, we will concentrate on the mutual relation of control and actions and on the style, in which actions are activated by control in a structured way.

3 CONTROL STRUCTURES AND CONTROL FUNCTIONS

By an analogy to data structures, we propose control structures be defined (but not constructed) by algebraic type constructors.

A new algebraic type in $\mathcal{PFL}$ is defined using the general form introduced in Definition 1. In this paper we use *algtype* keyword instead of *data* for the first time, expressing the fact that an algebraic type unifies quite different kinds of values – data and control ones.

Definition 1 (Algebraic type)**.** A new algebraic type T is defined as follows

$$\begin{array}{rcllll} \mathit{algtype}\ T\ a_1\ \ldots\ a_u & = & C_1 & T_{1,1} & \ldots & T_{1,n_1} \\ & \mid & C_2 & T_{2,1} & \ldots & T_{2,n_2} \\ & & \vdots & \vdots & \ddots & \vdots \\ & \mid & C_m & T_{m,1} & \ldots & T_{m,n_m}, \end{array}$$

where T is the new type, a_k, $k = 1, \ldots, u$ are type variables, C_i, $i = 1, \ldots, m$, are constructors, and $T_{i,j}$, $i = 1, \ldots, m$, $j = 1, \ldots, n_m$, are type expressions consisting of primitive types, type variables, or applications of algebraic types, that have been defined already.

According to Definition 1, each constructor C_i is a canonical function of the type

$$C_i :: T_{i,1} \to \ldots \to T_{i,n_i} \to T\ a_1\ \ldots\ a_u \tag{1}$$

and may be applied to expressions $e_1 \ldots e_{n_i}$.

The role of constructors is as follows:

1. They are introduced in algebraic type definitions to define a new algebraic type.
2. They are applied to data expressions to construct new data structures.
3. They are applied to control expressions to define new control structures.

Examples of some algebraic types are introduced in Figure 1.

Variant type:
$\quad$ *algtype* $Variant = C\ Char \mid I\ Int \mid F\ Float$

n-tuple types:
$\quad$ *algtype* $Pair\ a\ b\ =\ Pair\ a\ b$
$\quad$ *algtype* $Triple\ a\ b\ c\ =\ Triple\ a\ b\ c$

List type:
$\quad$ *algtype* $List\ a\ =\ Nil \mid Cons\ a\ (List\ a)$

Binary tree type:
$\quad$ *algtype* $Btree\ a\ =\ Tip\ a \mid Bin\ (Btree\ a)\ (Btree\ a)$

Binary search tree type:
$\quad$ *algtype* $Bstree\ a\ =\ Tips \mid Bins\ a\ (Bstree\ a)\ (Bstree\ a)$

Fig. 1. Examples of some frequently used algebraic types

While $Variant$ is the monomorphic data type, other types are polymorphic and then they can be either data or control types.

So, the data binary tree $(Bin\ (Bin\ (Tip\ 5)\ (Tip\ 2))\ (Tip\ 3))$ is of data type $(Btree\ Int)$, while the control binary tree $(Bin\ (Bin\ (Tip\ ())\ (Tip\ ()))\ (Tip\ ()))$ is of control type $(Btree\ ())$.

Similarly as for data expressions we may write control lists in the form $[() : () : () : []]$ or $[(), (), ()]$ instead of $(Cons\ ()\ (Cons\ ()\ (Cons\ ()\ Nil)))$, of control type $[()]$ instead of $(List\ ())$.

A syntactic support for n-tuples is also available. Then the form $((), (), ())$ instead of $(\mathit{Triple}\ ()\ ()\ ())$ may be used for control triples. In case of n-tuples, the same form is used for expressions and types.

To separate data and control in programming, we propose to suppress the mixed algebraic types, such as $(\mathit{Int}, ())$. This problem arises when a polymorphic type has more than one type variable, such as the pair (a, b). Then, if constructor *Pair* is applied to expressions e_1, e_2, both must be of (possibly different) data types, i.e. $(e_1, e_2) :: (T_1, T_2)$ or both are of control types such as $(e_1, e_2) :: ([()], ())$.

If constructors of algebraic types are applied to control expressions, their semantics is different than when they are applied to data expressions, as will be shown below.

3.1 Control Structures

The result of transformations DC, CC, DEC and CEC are unit values. The unit value represents primitive control, of primitive control type, called the unit type. At the same time, the unit value is the simplest control expression. If a constructor is applied to control expressions, the result will be control structure – a structure consisting of unit values.

In contrast to data structures that are constructed using data expressions, the meaning of control structures is different, since they just synchronize the computations that have yielded unit values.

Essentially, corresponding to the sequential evaluation order of arguments (currying), the evaluation of arguments in the form of control expressions is synchronized sequentially.

Both the unit value and constructor in a control expression are evaluated in zero time using no memory resources. Hence, even a complex control expression, such as

$$[((), [(), ()]),\ ((), [(), (), ()]),\ ((), [()])]$$

is also evaluated in zero time using no memory resources.

The definition of partial order of evaluation as defined by a control expression is introduced below.

Definition 2 (Partial order of evaluation)**.** A control expression is either of the unit type $()$, i.e. such that is evaluated to control value $()$, or it is in the form of the application of a constructor to control expressions (of control types), as follows:

$$C\ e_1\ e_2 \ldots e_n. \tag{2}$$

Then partial order of evaluation of control expressions in control structures is defined as follows:

1. For all expressions e_i, e_{i+1} in (2), such that they are both in the form of (2), $e_i \parallel e_{i+1}$ holds, i.e. e_i and e_{i+1} are evaluated in parallel.

2. If at least one of expressions e_i and e_{i+1} in (2) is of the unit type (), then $e_i \prec e_{i+1}$ holds, i.e. they are evaluated sequentially.
3. The priority of $\|$ operation is higher than that of sequential ordering operation $\prec$.

Hence, the effect of a constructor in control expression is to synchronize (theoretically in zero time) the evaluation of its arguments. According to Definition 2, control lists define sequential (dependent) execution order, and binary control trees define parallel (independent) execution order.

Example 1. To introduce the example of control list, let us define processes as follows:

$$\begin{aligned} f &:: \ u\ \mathit{Int} \to () \\ f\ x &= () \\[1em] g &:: \ v\ \mathit{Int} \to () \\ g\ x &= () \\[1em] h &:: \ u\ \mathit{Int} \to () \\ h\ x &= (). \end{aligned}$$

The control expression $[f\ 2, g\ 3, h\ 4]$ yields the value $[(), (), ()]$ of the type $[()]$. Since it is equivalent to $(\mathit{Cons}\ (f\ 2)\ (\mathit{Cons}(g\ 3)\ (\mathit{Cons}\ (h\ 4)\ \mathit{Nil})))$, it defines the following set:

$$\{(f\ 2),\ \{(g\ 3),\ \{(h\ 4),\ \{\}\ \}\ \}\ \}.$$

Since $(f\ 2) \prec \{(g\ 3),\ \{(h\ 4), \{\}\ \}\}$, $(g\ 3) \prec \{(h\ 4), \{\}\ \}$, and $(h\ 4) \prec \{\}$, it follows:

$$(f\ 2) \prec (g\ 3) \prec (h\ 4).$$

The expression $[f\ 2, g\ 3, h\ 4]$ requires that $(f\ 2)$ terminates before $(g\ 3)$ starts and $(g\ 3)$ terminates before $(h\ 4)$ starts. Besides, we may be sure that final value of u will be 4.

The next example deals with the binary control tree.

Example 2. Let us consider the same processes as in Example 1.

Then a binary control tree

$$\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Tip}\ (f\ 2))\ (\mathit{Tip}\ (g\ 3))\ (\mathit{Bin}\ (\mathit{Tip}\ (f\ 4))\ (\mathit{Tip}\ (h\ 5))$$

yields the value

$$\mathit{Bin}\ (\mathit{Bin}\ (\mathit{Tip}\ ())\ (\mathit{Tip}\ ())\ (\mathit{Bin}\ (\mathit{Tip}\ ())\ (\mathit{Tip}\ ())$$

of the type Btree ().

The sets defined by binary control tree are as follows

$$\{\ \{\ \{(f\ 2)\}, \{(g\ 3)\}\ \},\ \{\ \{(f\ 4)\}, \{(h\ 5)\}\ \}\ \},$$

in that all applications are independent, i.e.

$$(f\ 2) \parallel (g\ 3) \parallel (f\ 4) \parallel (h\ 5)$$

and the execution terminates, when all applications are terminated.

3.2 Control Functions

Pure control functions are the transformations on control values, including control structures. As for constructors, just polymorphic functions can be used as control functions. As for constructors, the applications of control functions are executed in zero time using no memory resources.

For example, let us define the polymorphic functions (++) and *reverse* as follows:

$$\begin{array}{llcl}
\multicolumn{4}{l}{(+\!\!+) :: [a] \to [a] \to [a]} \\
[\,] & +\!\!+ \;\; ys & = & ys \\
(x : xs) & +\!\!+ \;\; ys & = & x \; : \; xs +\!\!+ ys
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{reverse} :: [a] \to [a]} \\
\mathit{reverse}\ [\,] & = & [\,] \\
\mathit{reverse}\ (x : xs) & = & \mathit{reverse}\ xs +\!\!+ [x].
\end{array}$$

In Example 3, we will compare the effect of the application of function *reverse* to data expression and its application to the control expression.

Example 3. According to the definition of function (++) above, it concatenates two lists. The value of the application of *reverse* to a list is reversed list.

For example, the value of an application (*reverse* $[1, 3, 5, 7]$) is the list $[7, 5, 3, 1]$ (data). In this case *reverse* was applied as a data function.

On the other hand, the value of (*reverse* $[f\ 2, g\ 3, h\ 4]$) may be thought as the list $[h\ 4, g\ 3, f\ 2]$, which implies the evaluation order as follows

$$(h\ 4) \prec (g\ 3) \prec (f\ 2). \tag{3}$$

3.3 Implementation Remarks

Although this paper is not about implementation, we expect that the code generation considering control structures and functions is not straightforward. To produce an efficient code, we must perform interpretation (partial evaluation) of control expressions in compile time.

First, each expression of the unit type is associated with the unit value. For the purpose of simplicity, let us mark unit values by indices. We will get control expression (*reverse* $[()_1, ()_2, ()_3]$) and the association set $\{f\ 2 \leftrightarrow ()_1, g\ 3 \leftrightarrow ()_2, h\ 4 \leftrightarrow ()_3\}$. Using interpreter, all control functions (such as *reverse*) are applied. The result of interpretation will be the set of canonical control values, expressed in terms of applications of constructors, (in our case it will be $[()_3, ()_2, ()_1]$). After that, the partial order according to Definition 2 is derived as follows: $()_3 \prec ()_2 \prec ()_1$. Finally, based on the association set, the target code for application in the derived order (3) is generated, which is executed in the run-time.

Since both the evaluation of control expressions and the transformation of canonical control expressions into the execution order is the matter of compile time, we can say that control expressions are executed in zero time using no memory resources in the run-time.

As we will show in the next section, canonical control values, such as $()$ or $[((), ()), ((), ()), ((), ())]$ (control list of control pairs), can be used to activate actions.

4 ACTIONS

Intuitively, each action must be activated. In the simplest case, action is activated by the unit value – the non-data constant representing control. In Table 1, an action is one of transformations CD, CC, CED, or CEC. As can be seen, each action can be applied to the argument in the form of the unit value, which is either initiating constant mentioned above, or the result of argument computation.

In contrast to transformations DD, DC, DED and DEC that are activated implicitly by argument data (by program), the actions are activated explicitly by control (by programmer). The role of control is increasing, if the evaluation is not purely functional, i.e. restricted just to DD and DED transformations. (In case of DED transformations, the values are reflected in an external environment, but the evaluation remains still purely functional, because no environment value is used in expressions.)

4.1 State Affecting Processes in $\mathcal{PFL}$

Currently we provide a general mechanism in which memory cells external to processes can be affected by applications of processes. Syntactically, this mechanism is supported by attributing the types of arguments of processes with the names of environment variables as follows

$$\begin{array}{l} p \ :: \ v_1 \, T_1 \rightarrow \ \ldots \ \rightarrow v_n \, T_n \rightarrow \overline{T} \\ p \, x_1 \ \ldots \ x_n \ = \ \overline{e}, \end{array} \tag{4}$$

where $T_1, \ldots, T_n$ are data types of arguments, $v_1, \ldots, v_n$ are environment variables – external memory cells, and $\overline{T}$ is either a data type T or the unit type $()$.

Data types in type definitions of processes are attributed with environment variables that, on the other hand, cannot be used in definitions (i.e. in expressions). The processes can be applied not just to data arguments but also to unit values. After some steps of the translation of the source form (4), the variable environment $\rho[v_1, \ v_2, \ \ldots, \ v_n]$ and two functions for each variable in ρ are generated as follows:

$$\begin{array}{ll} \lambda x.\ let\ v_k = x\ in\ v_k & \text{(the update function)} \\ \lambda().\ v_k & \text{(the access function)} \end{array} \tag{5}$$

for $1 \leq k \leq n$.

Then, the source form of each application of process p in the form

$$p\ \overline{e}_1\ \overline{e}_2\ \ldots \overline{e}_n \tag{6}$$

where $\overline{e}_k$, $k = 1 \ldots n$ are either data expressions or unit values are transformed, substituting arguments by application of one from two functions above to argument as follows:

1. If $\overline{e}_k$ is data expression of the type T_k then it is substituted by

$$(\lambda x.\ let\ v_k = x\ in\ v_k)\ \overline{e}_k.$$

2. If $\overline{e}_k$ evaluates the unit value of the unit type, then it is substituted by

$$(\lambda().\ v_k)\ \overline{e}_k.$$

Since both applications above affect the environment variable, let us explain two possible effects, supposing two extreme cases.

First, let all arguments be of data types. Then the application (6) (representing p by lambda abstraction) is evaluated as follows:

$$\frac{\begin{array}{c} \rho[v_1 = e'_1,\ \ldots,\ v_n = e'_n] \vdash \\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e}) :: v_1\ T_1 \rightarrow\ \ldots\ \rightarrow v_n\ T_n \rightarrow \overline{T} \\ e_1 :: T_1\ \ldots\ e_n :: T_n \end{array}}{\begin{array}{c} \mathcal{E}val[\![\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e})\ e_1\ \ldots\ e_n\ ]\!]\ \rho = \\ \mathcal{E}val[\![\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e})\ (\rho\ v_1) \ldots (\rho\ v_n)]\!]\ \rho[v_1 = e_1,\ \ldots,\ v_n = e_n] = \\ \overline{e}[e_1/x_1, \ldots, e_n/x_n] \end{array}} \ . \tag{7}$$

The values of data arguments were reflected in environment variables in the order corresponding to currying, i.e. as follows:

$$v_1 = e_1 \prec \ldots \prec v_n = e_n.$$

Second, if all arguments are unit values (or expressions of unit types), then the application of p is evaluated as follows:

$$\frac{\begin{array}{c} \rho[v_1 = e'_1,\ \ldots,\ v_n = e'_n] \vdash \\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e}) :: v_1\ T_1 \rightarrow\ \ldots\ \rightarrow v_n\ T_n \rightarrow \overline{T} \\ e_1 :: ()\ \ldots\ e_n :: () \end{array}}{\begin{array}{c} \mathcal{E}val[\![\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e})\ e_1\ \ldots\ e_n\ ]\!]\ \rho = \\ \mathcal{E}val[\![\ (\lambda x_1.\ \ldots\ \lambda x_n.\ \overline{e})\ (\rho\ v_1) \ldots (\rho\ v_n)]\!]\ \rho = \\ \overline{e}[e'_1/x_1, \ldots, e'_n/x_n] \end{array}} \ . \tag{8}$$

Then the values in environment variables remain unchanged and they are used in the evaluation of the body $\overline{e}$.

In $\mathcal{PFL}$, the access of an environment data is the action, the update is just reflective. In both cases, if $\overline{e}$ is of a data type, then the application (6) yields data value. On the other hand, if $\overline{e}$ is of the unit type, then the application (6) yields the unit value.

4.2 Activation of Actions

Clearly, the function of an action expressed by the expression $\overline{e}$ may be very complex; but the action is still *elementary*, if it is activated by its application to the unit value. It follows that elementary action is lambda abstraction, as follows

$$(\lambda().\ \overline{e}) \tag{9}$$

where $\overline{e} : \overline{T}$, and $(\lambda().\ \overline{e}) : () \rightarrow \overline{T}$.

Then, the activation of an elementary action is the application of lambda abstraction as follows:

$$\mathcal{E}val[\![\ (\lambda().\ \overline{e})\ ()\ ]\!] \ = \ \mathcal{E}val[\![\ \overline{e}\ ]\!]. \tag{10}$$

After the action is activated, it is executed by evaluation $\mathcal{E}val[\![\ \overline{e}\ ]\!]$, yielding the value of expression of the type $\overline{T}$.

Understanding the difference between the executed action $\overline{e}$ and its abstraction $(\lambda().\ \overline{e})$ is crucial for expressing more powerful transformations than those in an imperative language.

In an imperative language, this difference is not so visible. Seemingly, the same statements are executed as those written in programs. However, source program statements are abstractions, and executed statements are the applications of these abstractions, i.e. the expressions of the unit type.

To illustrate the difference between abstractions and expressions, it is possible to introduce the application of fully abstracted form of the expression $(x+3)$, which is as follows:

$$(\ \lambda().((\lambda().x)\ () + (\lambda().3)\ ())\ )\ ().$$

While the flow of control in the expression $(x+3)$ is hidden, the application of its abstraction above expresses all possible activations of actions during the evaluation of this expression.

4.3 Structured Application

Let us consider the activation of a set of actions forming a structured action. We are interested in the way, in which "more powerful activation" of structured actions than the activation of an elementary action can be expressed. By an analogy to (10), the expected form of structured application is as follows:

$$\mathcal{A}\ \mathcal{C},$$

in which $\mathcal{A}$ is a representation of structured action, consisting of the set of elementary actions, as follows

$$(\lambda().\ \overline{e}_1),\ (\lambda().\ \overline{e}_1),\ \ldots,(\lambda().\ \overline{e}_n), \tag{11}$$

and $\mathcal{C}$ is a purely control expression, which activates all actions comprised in the structured action $\mathcal{A}$.

The solution can be found, based on the function *zipWith* for lists, slightly re-defined as follows:

$$\begin{array}{llllll}
zipWith :: (a \to b \to c) \to [a] \to [b] \to [c] \\
zipWith & \oplus & [\,] & [\,] & = & [\,] \\
zipWith & \oplus & (x : xs) & (y : ys) & = & (x \oplus y)\ :\ zipWith\ \oplus\ xs\ ys.
\end{array}$$

Clearly, the value of the application (*zipWith* (+) $[1, 2, 4]$ $[10, 20, 30]$) is [11, 22, 34], and the value of (*zipWith* (+) [] []) is empty list []. On the other hand, the evaluation of (*zipWith* (+) $[1, 2, 4]$ $[10, 20]$) fails, since we do not allow argument lists of different length.

A non-structured application $(f\ e)$ is the source form for $(f@e)$, in which the application operation @ of the type $(T_0 \to T_1) \to T_0 \to T_1$ is applied to a function f, of the type $(T_0 \to T_1)$ and to an expression e, of the type T_0.

Provided that $T_0 = ()$, we have $(@) :: (() \to T) \to () \to T$, which is exactly what is needed; the first argument is an action of the type $() \to T$ and the second is an expression of the unit type.

Then the value of application (*zipWith* (@) $[f_1, f_2, f_3]$ $[(), (), ()]$) is $[f_1@(), f_2@(), f_3@()]$ or simply $[f_1\ (), f_2\ (), f_3\ ()]$. We have obtained the list of activated actions.

Suppose now, we define the overloaded operation (#), of the type

$$(\#) :: (T\ () \to T\ a) \to T\ () \to T\ a \tag{12}$$

where T is a polymorphic algebraic type.

For lists, we have (#) = (*zipWith* @), such that $(\#) :: ([()] \to [a]) \to [()] \to [a])$.

For binary trees, (#) is defined as follows:

$$\begin{array}{lllll}
(\#) :: (Btree\ () \to Btree\ a) \to Btree\ () \to Btree\ a \\
(Tip\ x) & \# & (Tip\ y) & = & Tip\ (x\ y) \\
(Bin\ tx1\ tx2) & \# & (Bin\ ty1\ ty2) & = & Bin\ (tx1\#ty1)\ (tx2\#ty2).
\end{array}$$

For pairs, (#) is defined as follows:

$$\begin{array}{lllll}
(\#) :: (((), ()) \to (a, b)) \to ((), ()) \to (a, b) \\
(x1, x2) & \# & (y1, y2) & = & (x1\#y1, x2\#y2).
\end{array}$$

Then, the general form for the structured application – the application of a structure of actions to the same structure of controls is as follows

$$\#\ (C\ f_1\ \ldots f_n)\ (C\ e_1\ \ldots\ e_n),$$

where f_k are abstractions of the type $() \to T$, and e_k are expressions of the unit type.

Prefix form of structured application above can be written in the infix form as follows:

$$(C\ f_1\ \dots f_n)\ \#\ (C\ e_1\ \dots\ e_n).$$

Provided that both (@) and all instances (#) are overloaded, it is even possible to use (@) instead of (#) and then we get the structured application in the form

$$(C\ f_1\ \dots f_n)\ (C\ e_1\ \dots\ e_n).$$

It means that $\mathcal{A} = (C\ f_1\ \dots f_n)$ is a (data) structure of actions (abstracted expressions) and $\mathcal{C} = (C\ e_1\ \dots\ e_n)$ is the same but control structure of control expressions, as expected.

4.4 Abstracting Expressions to Actions

We propose the source form $\backslash e$ to express $(\lambda().\ e)$.

This form is useful whenever a programmer requires the evaluation of the expression e be controlled. As we have shown, the evaluation of e yields the same result as the controlled evaluation $\backslash e\ ()$.

More general abstraction of e by $(\lambda x.\ e)$ which may be applied to a data expression is clearly redundant, since each process in the source program is defined in terms of this abstraction.

5 EXAMPLES

Instead of complex application examples we will introduce the comparison of a classical imperative approach and the approach to programming using more powerful structured application with respect of separated data, control and actions.

5.1 Imperative Approach

The execution of the assignment of an expression e to a variable v is expressed by the application $(assignv\ e)$, where $e :: T$ (DEC transformation), and the process $assignv$ is defined as follows:

$$\begin{array}{l} assignv :: v\ T \to () \\ assignv\ x\ =\ () \end{array}$$

Then $(assignv\ e) :: ()$, and abstracted assignment is $\backslash(assignv\ e)$, of the type $() \to ()$. Then assignment may be executed by the application

$$\backslash(assignv\ e)\ () \qquad \text{instead of} \qquad (assignv\ e).$$

In general, if the application s is an executed statement, then $s :: ()$ and its abstracted form is $\backslash s$, such that $\backslash s :: () \to ()$. Then, as for assignment, the statement s may be executed by the application

$$\backslash s\ () \qquad \text{instead of} \qquad s.$$

Notice that this application itself can be very complex, and the assignment execution ($assignv\ e$) is just the simplest (but still complex) case.

Then, the execution of the sequence of the statements $s_1, \ldots, s_n$ is the application, as follows:

$$(\backslash s_n)\ (\ldots((\backslash s_1)\ ())\ldots) \qquad \text{or} \qquad (\backslash s_n \circ \ldots \circ \backslash s_1)\ ().$$

Hence, the abstraction of a statement sequence can be expressed in terms of composition operation $(\circ)$ (defined by $(f \circ g)\ x\ =\ f\ (g\ x)$), as follows:

$$\backslash s_n \circ \ldots \circ \backslash s_1.$$

Then *if-then-else* statement is executed by the application as follows

$$\backslash(\mathit{if}\ \backslash e\ (\backslash s^T_{n_T} \circ \ldots \circ \backslash s^T_1)\ (\backslash s^F_{n_F} \circ \ldots \circ \backslash s^F_1))\ ()$$

or

$$\mathit{if}\ \backslash e\ (\backslash s^T_{n_T} \circ \ldots \circ \backslash s^T_1)\ (\backslash s^F_{n_F} \circ \ldots \circ \backslash s^F_1),$$

provided that *if* is defined as follows:

$$\begin{array}{lll} \mathit{if}\ ::\ (() \to \mathit{Bool}) \to (() \to ()) \to (() \to ()) \to () & & \\ \mathit{if}\ b\ x\ y & \mid b\ () & =\ x\ () \\ & \mid \mathit{otherwise} & =\ y\ (). \end{array}$$

Clearly, the abstraction of *if-then-else* statement is as follows:

$$\backslash(\mathit{if}\ \backslash e\ (\backslash s^T_{n_T} \circ \ldots \circ \backslash s^T_1)\ (\backslash s^F_{n_F} \circ \ldots \circ \backslash s^F_1)).$$

By an analogy, the execution of *while* statement is defined by the application as follows:

$$\backslash(\mathit{while}\ \backslash e\ (\backslash s_n \circ \ldots \circ \backslash s_1))\ () \quad \text{or} \quad \mathit{while}\ \backslash e\ (\backslash s_n \circ \ldots \circ \backslash s_1).$$

The definition of *while* is as follows:

$$\begin{array}{lll} \mathit{while}\ ::\ (() \to \mathit{Bool}) \to\to (() \to ()) \to () & & \\ \mathit{while}\ b\ x & \mid b\ () & =\ \backslash(\mathit{while}\ b\ x)\ (x\ ()) \\ & \mid \mathit{otherwise} & =\ (). \end{array}$$

Notice that both arguments – Boolean expression b and the statement sequence x are abstractions, i.e. constants. After $(x\ ())$ is executed in an iteration step, its value is used as an argument for *while* abstraction again, which, when applied, yields the application (*while* b x) of the same constant arguments as in the previous iteration step. Then, the same context (on the stack) can be used for all iterations, and *jump-to-subroutine* instruction can be replaced by simple *jump* instruction, when implementing *while* call. Or even, as in imperative languages, instead of defining all *while*'s separately using different names, *while* (and also *if*) can be defined as built-in operations, an then their bodies are included into the code (as macros), instead of calling them as functions.

In the simplest form, input operation *#input* is built-in operation of the type:

$$\#\mathit{input} :: T \tag{13}$$

Whenever applied, in the form *#input*, it reads a value of the type T from an external world of computation. Being a constant function, it is still referentially non-transparent, since the values of two applications may be different.

If abstracted into the form \*#input*, of the type $() \rightarrow T$, the application (\*#input*) () is controlled.

Output operation *#output* is the other black box in computation, of the following type:

$$\#\mathit{output} :: T \rightarrow () \tag{14}$$

Then, if abstracted into the form \\(*#output* e) of the type $() \rightarrow ()$, the application \\(*#output* e) () is controlled.

Notice that *#output* and *assignv* are very similar; *#output* assigns the value of the argument e to an external device and *assignv* assigns it to environment variable v. That is why the attributed type (v T) used for *assign* is just a source tool, which identifies, where the argument value is assigned. But the types of both *#output* and *assignv* are $T \rightarrow ()$.

5.2 Functional Approach Using Structured Application

Compound assignment *compoundAssign* assigns the values of arguments subsequently to environment variables $v_1, \ldots, v_n$, provided that it is defined as follows:

$$\begin{array}{l} \mathit{compoundAssign} :: v_1\ T_1 \rightarrow \ldots \rightarrow v_n\ T_n \rightarrow () \\ \mathit{compoundAssign}\ x_1\ \ldots\ x_n\ =\ (). \end{array}$$

Replacing the unit value () by an expression e in the definition, the assigned values can be immediately used in its evaluation since they are represented by lambda variables $x_1, \ldots, x_n$.

Replacing the unit type () by a data type T in the type definition e yields the data value, possibly even in a purely functional manner, which, however, is out of our interest in this paper.

As a consequence of separating data, control and actions, programs with visible environments can be written very concisely and expressively using functional programming approach.

Instead of writing the statement sequence in reversed order using composition ($\circ$), the execution of statement sequence is in the form of list of executed statements, as follows

$$[\ s_1, \ldots, s_n\ ],$$

or in the form of the structured application of the list of abstracted statements to the control list of the same length, as follows:

$$[\ \backslash\ s_1, \ldots, \backslash\ s_n\ ]\ [\ (), \ldots, ()\ ].$$

In both cases, the result is control list $[\ (), \ldots, ()\ ]$ of the type $[()]$.

The transformation from control lists to the unit value may be defined as follows

$$\begin{array}{l} unit :: [()] \to () \\ unit\ x\ =\ (), \end{array}$$

to obtain the form equivalent to $\backslash s_n \circ \ldots \circ \backslash s_1$, as follows

$$\backslash(unit\ [\ s_1, \ldots, s_n\ ]),$$

in which s_k are executions of statements, not their abstractions.

Of course, the abstraction above would be used as an argument of *while* or *if-then-else* statements representing a block of imperative statements.

On the other hand, there is no need to restrict a programmer to this imperative approach. The sequences of actions can be concatenated, reversed, mapped, and transformed by user-defined transformers using well known functions for lists, such as (++), *reverse*, *map*, etc.

It is beyond the scope of this paper to enumerate the variety of applications to illustrate the advantages of this approach. Instead, we will introduce just a simple example of the definition of a process f, in which s_k are the statements as any applications yielding the unit value. The example below is not very transparent, and it is far from good programming style, since the aim is to illustrate the effect of separated control rather than to present programming methodology.

Example 4. For the purpose of simplicity, suppose we have a set of processes defined, that applications s_1, s_2, etc., are all of the types (). So, their abstractions $\backslash s_1$, $\backslash s_2$, etc. are all of the types $() \to ()$. We will use them in the next program:

```
f :: Int → Bool → [()]
f n e =  [ s1 | i ← [1 . . n] ] ++
         branch e
           \([s2, s3, s4] ++ reverse [s5, s6])
           \(reverse [\s7, \s8] [s4, s5])
         where
             branch :: Bool → (() → [()]) → (() → [()]) → [()]
             branch e x y  | e          =  x ()
                           | otherwise  =  y ()
main :: Int → Bool → [()]
main  =  f 3 True
```

Then *main* evaluates to the control list of eight unit values, yielding control list of the type [()] as its value. Nothing will be displayed. During the evaluation the actions are performed in the following sequence:

$$s_1 \prec s_1 \prec s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_6 \prec s_5.$$

In particular, as a result of partial evaluation, the list

$$[for\ \backslash s1\ \backslash 1\ \backslash n, s_2, s_3, s_4, s_6, s_5]$$

is formed, in which ($for\ \backslash s1\ \backslash 1\ \backslash n$) of the type () is the application of *for* which implements the list comprehension [$s_1 \mid i \leftarrow [1\ .\ .\ n]$] iteratively. (The *for* implementation is beyond the scope of this paper; nevertheless, the approach is very similar to that used for *while*.)

On the other hand, if the definition of *main* will be changed to $main = f\ 3\ False$, the result will be the sequence of seven actions, as follows:

$$s_1 \prec s_1 \prec s_1 \prec s_4 \prec s_5 \prec s_8 \prec s_7.$$

In this case, *reverse* is applied to the (data) list of actions $[\backslash s_7, \backslash s_8]$, producing the list $[\backslash s_8, \backslash s_7]$, while $[s_4, s_5]$ is evaluated before used as the argument in structured application $[\backslash s_8, \backslash s_7]\ [s_4, s_5]$. That is why partially evaluated list will be as follows:

$$[for\ \backslash s1\ \backslash 1\ \backslash n, s_4, s_5, s_8, s_7]$$

Clearly, the expressions of unit types must be evaluated eagerly, in contrast to expressions of data types, that may be evaluated either eagerly or lazily.

In both cases, the first three actions are produced iteratively by list comprehension [$s_1 \mid i \leftarrow [1\ .\ .\ n]$] representing *for* statement (except that it is of the type [()], not of the type () as in imperative languages).

6 CONCLUSION

Instead of considering execution order of actions directly, we have used structured approach to building control structures by applying constructors of control types to control expressions. This allows us to separate the control from the function of computation systematically. As we have shown, this yields more powerful programming methodology than mixing function and control in step–by–step manner using an imperative language. Although a programmer never manipulates the environment variable directly, all environment variables are visible and able to reflect the data in computation expressed in the form of expression.

Seemingly, this paper is weakly related to aspect-oriented paradigm, but as we believe, all the above mentioned supports more transparent and expressive definitions of pointcut designators than when they are based on an imperative language.

Moreover, control constructors are named points in a program that semantics is defined by the definition of a control type. For example, control triple $((), (), ())$ defines sequential order of arguments, and control triple $((()), (()), (()))$ would define parallel (i.e. independent by control) evaluation of arguments. Notice that $(())$ is just a shortcut for the type *Unit* $()$, provided that *algtype Unit a* $=$ *Unit a* is defined.

Pipeline processing can be expressed using $(([()]), ([()]), ([()])$ as parallel evaluation of lists, each being evaluated sequentially. Expressing farm (trivial) parallelism is left to a reader. On the other hand, it is impossible to express massive parallelism using algebraic types, since this type of parallelism is based on $\mathcal{PFL}$ arrays, that are beyond the scope of this paper.

The unification of programming languages on a multiparadigm basis, in the way which makes their connections to the specification methods precisely defined, methods of behavioral analysis, etc., would be great contribution to the stability, extensibility, an reliability of software production in the future. In this sense, our research in integrating process functional and aspect-oriented paradigm is just a particular experiment, which provides an opportunity to understand the needs for a systematic development of multiparadigm languages.

Originally, the development of $\mathcal{PFL}$ language was not related to aspect paradigm at all. The aim was just to provide an experimental language integrating purely functional syntax and fully imperative semantics, to make environment visible to a user, as a software engineering concept, opposite to monadic 'more mathematical' approach.

On the other hand, as shown in this paper, using process functional paradigm any grain of computation can be affected using additional control, and any data can be reflected in an external environment in a very systematic way. The reflection and control evidence are clearly the properties inevitable for adding new aspects to a system. The contribution of this paper is in making them exploitable in large scale of graining, which is a proposition for the mutual connection of the program structure and events that are related to both static and dynamic aspects of computation. Mechanisms needed for both static and dynamic weaving at fine grains of computation are currently under development, and the selection of events related to dynamic aspects and their specification using temporal logic is the future.

REFERENCES

[1] Avdicausevic, E.—Lenic, M.—Mernik, M.—Zumer, V.: AspectCOOL: An Experiment in Design and Implementation of Aspect-Oriented Language. ACM Sigplan Not., December 2001, Vol. 36, No. 12, pp. 84–94.

[2] Filman, R. E.—Friedman, D. P.: Aspect-Oriented Programming is Quantification and Obliviousness. In Workshop on Advanced Separation of Concerns (OOPSLA 2000), Oct. 2000.

[3] Kiczales, G. et al: An Overview of AspectJ. Lecture Notes in Computer Science, Vol. 2072, 2001, pp. 327–355.

[4] Kiczales, G. et al: Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, 11th Europeen Conf. Object-Oriented Programming, Volume 1241 of LNCS, pp. 220–242, 1997.

[5] Kienzle, J.—Guerraoui, R.: Aspect Oriented Software Development AOP: Does It Make Sense? The Case of Concurrency and Failures. In B. Magnusson, editor, Proc. ECOOP 2002, pp. 37–61. Springer Verlag, June 2002.

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

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

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

[9] Kollár, J.—Václavík, P.—Porubän, J.: The Classification of Programming Environments. Acta Universitatis Matthiae Belii, Vol. 10, 2003, pp. 51–64, ISBN 80-8055-662-8.

[10] Lämmel, R.: Adding Superimposition to a Language Semantics. Foundations of Aspect-Oriented Langauges Workshop at AOSD 2003, pp. 61–70.

[11] Mernik, M.—Kosar, T.—Zumer, V.: A Note on Aspect, Aspect–oriented and Domain–Specific Languages. Acta Electrotechnica et Informatica, FEII TU Košice, Slovakia, Vol. 5, 2005, No. 1, pp. 1–8.

[12] 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.

[13] Peyton Jones, S. L.—Hughes, J. editors: Report on the Programming Language Haskell 98 – A Non-Strict, Purely Functional Language. February 1999, 163 pp.

[14] Porubän J.: Time and Space Profiling for Process Functional Language. Proc. EMES '03, May 29–31, 2003, Felix Spa – Oradea, University of Oradea, 2003, pp. 167–172, ISSN-1223-2106.

[15] Porubän, J.: Functional Programs Profilation. Ph. D. Thesis, March 2004, DCI FEII TU Košice, 87 pp. (in Slovak).

[16] Sullivan, G. T.: Aspect-Oriented Programming Using Reflection and Meta-Object Protocols. Comm. ACM, Vol. 44, 2001, No. 10, pp. 95–97.

[17] Václavík, P.—Porubän, J.: Object Oriented Approach in Process Functional Language, Proc. ECI 2002, October 10–11, 2002, Košice – Herľany, 2002, pp. 92–96, ISBN 80-7099-879-2.

[18] Václavík, P.: The Fundamentals of a Process Functional Abstract Type Translation. Proc. EMES '03, May 29–31, 2003, Felix Spa – Oradea, University of Oradea, 2003, pp. 193–198, ISSN-1223-2106.

[19] Václavík, P.: Implementation of Abstract Types in a Process Functional Programming Language. Ph. D. Thesis, March 2004, DCI FEII TU Košice, 108 pp. (in Slovak).

[20] 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.

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

[22] Wand, M.: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. LNCS, Vol. 2196, 2001, pp. 45–57.

Ján Kollár (Assoc. Prof., M. Sc., Ph. D.) received his M. Sc. summa cum laude in 1978 and his Ph. D. 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, United Kingdom. He was involved in the research projects dealing with real-time systems, the design of (micro) programming languages, image processing and remote sensing, dataflow systems, implementation of programming languages. Currently he is working in the field of multi-paradigmatic languages, with respect of aspect paradigm. He is the author of $\mathcal{PFL}$ – a process functional language.

Jaroslav Porubän (M. Sc., Ph. D.) received his M. Sc. summa cum laude in 2000 and his Ph. D. in computing science in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling based on process functional language. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming and program profiling systems.

Peter Václavík (M. Sc., Ph. D.) received his M. Sc. summa cum laude in 2000 and his Ph. D. in Computing Science in 2004. Since 2003 he is with the Department of Computers and Informatics at Technical University of Košice. He has implemented the object oriented version of $\mathcal{PFL}$ language. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming and program profiling systems.