PROCESS FUNCTIONAL PROPERTIES AND ASPECT LANGUAGE

Ján KOLLÁR
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic,
tel. +421 55 602 2577, E-mail: Jan.Kollar@tuke.sk

SUMMARY

In this paper we present essential characteristics of aspect-oriented approach to programming as provided in aspect programming languages. Then we de-modularize a programming structure of a process functional sample to a type definition module and the own definition module, using purely functional case. Adding environment variables to the type definition module we show that there are possible resources to the computational reflection using process functional paradigm in a well-defined variable environment. We also identify the weaknesses and possible directions in further development of object-oriented process functional language to extend it to an aspect oriented language.

Keywords: *Programming paradigms, process functional programming, aspect oriented programming, computational reflection, programming environments*

1. INTRODUCTION

Aspect oriented programming evolves from the fact that there exist some crosscutting concerns in systems that cannot be well modularized using traditional structured, object or component based software development methodologies. There is no formal proof but high deal of evidence that combination of different concerns of computation in complex software systems yields to scattered and tangled code, which is inappropriate to maintenance [2,3,4]. Sometimes, the appropriate modularization still can be reached, but the prize is too high – the run-time efficiency is decreased.

The other source for producing tangled code is adding a new concern of computation after a system has been developed. Then the situation, when manifold source code modifications are needed for the purpose of efficiency, is the nightmare of programmers. Scattering code manually clearly decreases the reliability of the system and its capability for the maintenance.

AspectJ [7,8] is a programming language, which provides the opportunity to a programmer for the modular description of crosscutting concerns via aspect declarations. The aspect declaration, similar to class declaration is a modular unit, which in addition to class declaration contains

- pointcut – the definition of a collection of join point – well defined points of computation in which advice is applied, and
- advice – a part of code, which is applied in join points, defined by pointcut designator.

AspectJ approach has evolved from Java – which is inherently object oriented imperative language. Therefore it seems that the subject of aspect language is applicable just to an object-oriented paradigm, but this is not true [1,16,35]. Crosscutting concerns can be taken into account also at the procedural level, excluding object paradigm, or at functional level, excluding an imperative paradigm. On the other hand, the crucial question is the usefulness of separated programming paradigms, for the development of large systems. Our mention is that better direction is to integrate them.

For example, object paradigm is without doubt the best-balanced basis for applying crosscutting concerns across classes because of systems complexity and their imperative nature.

However, the limits of AspectJ language are currently known [9]. The substance of these limits is as follows: Sometimes there is too strong interference between the function of computation and an aspect (specifically when parallel concerns are considered) and then the benefits of an aspect approach are not so high as expected. The reasons of this fact may be perhaps in strong binding of AspectJ to Java byte code. It may be noticed that AspectJ pointcut designators have their origins in Java language implementation, since AspectJ is an extension to Java.

In this paper we present our approach to possible incorporation of aspect programming paradigm into $\mathcal{PFL}$ - a process functional programming language that is based on application of processes, rather than statement sequences [10,11,12,13,14]. Although at the present time we have object $\mathcal{PFL}$ implemented [15,29,30,31,32] with both Haskell [22] and Java target code, it is not our aim to provide just a new programming language. The aim is to exploit the uniform and simple multi-paradigmatic structure of $\mathcal{PFL}$ integrating the functional, imperative [5,34], and object oriented paradigm [15] with the aspect paradigm. We have found it useful during experiments with profiling process functional programs [23,24,25] and mobile agents

This work was supported by VEGA Grant No. 1/1065/04: Specification and Implementation of Aspects in Programming.

programming [20]. In the following sections we present the essence of the aspect oriented conception and then, using simple tracing example, we will show the properties of process functional paradigm with respect to requirements to aspect extensions. Finally, we discuss the current state and possible directions in further research.

2. ASPECT ORIENTED CONCEPTION

Let us introduce the essential conception of the aspect approach to system development according to Fig.1. For the purpose of simplicity, let us consider incremental development of a system, considering first a functional aspect of computation and after that some tracing aspect. Let the functionality of a system is defined by the structure of two modules as illustrated by gray rectangles in the stage 1 of Fig.2.1.

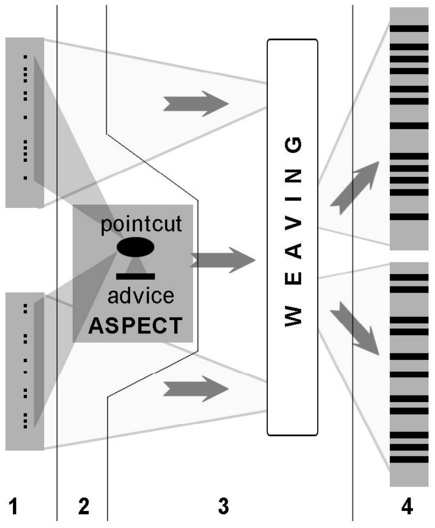

Figure 2.1 Aspect – oriented conception

Omitting the detailed function, the system of two modules can be compiled and executed. Suppose we need to include some tracing actions into modules. Instead of doing it manually, in aspect approach we write (in the stage 2) ASPECT module. This module consists of the pointcut and the advice. Pointcut is a collection of points in original modules that are the subject of interest (the subject of tracing, in our case). Such points are called join points. The pointcut is defined by the pointcut designator, i.e. a formula that identifies a collection of join points, marked by small dots in modules in Fig.2.1. In this manner join points are just identified, but the original modules are not affected.

The second part of the aspect is the advice - a part of code, which we want to place at join points. The pointcut is used in the definition of advice. The stage 2 is finished.

The stage 3 in Fig.2.1 illustrates weaving, which is an automated process of transforming original modules and defined aspect module, producing two modules, in which tracing actions are woven.

The result is a new system of consisting of two modules, in which the advice is applied, see stage 4 in Fig. 2.1. As can be seen, this new system has tracing code scattered across the original modules.

There are two main benefits of this aspect approach. First, a programmer need not scatter the advised tracing code manually and second, whenever needed, tracing aspect may be "removed" by re-compilation of original system to obtain the system with functionality as before its aspectizying.

Although tracing example yields scattered code, it is high deal of evidence, that combining other aspects can yield even tangled code, and it is not dependent on whether the system is developed incrementally or not.

Tracing above is based on pointcut, which defines static joint points that are the subject of compile time weaving. Opposite to static join points, dynamic joint points are such that are defined in dynamic context of program i.e. while execution. An example is `cflow` pointcut designator in AspectJ, which is used to define join points occurring in all methods called from a given method of a class.

Then, instead static weaving, dynamic (i.e. run-time) weaving must be used to perform crosscutting in dynamic join points.

The complication coming out from dynamic context of a program is as follows: The events during execution belong to a different abstraction levels, from such as input values of computation to those as architecture resources. The commonly accepted mechanism, which allows identify run-time crosscutting is computational reflection [26].

Computational reflection is the capability of a computational system to reason about itself and act upon itself, and adjust to changing conditions. The computational domain of a reflective system is the structure and the computations of the system itself. A reflective system incorporates data representing static and dynamic aspects of it; this activity is called *reification*. This self-representation makes it possible for the system to answer questions about and support actions on it.

Thus, the crucial task associated with dynamic context reasoning is to incorporate reflection data into a system, extracting them from original. In particular, we will show in this paper, how it can be solved using process functional program structure.

In the next section we will present the possible modularization of a purely functional program, starting with a simple purely functional case obtaining separate function type definition module and function definition module. In section 4 we will use the type module, aspectized by variable environment.

3. TYPE AND DEFINITION MODULE

Process functional paradigm is based on evaluation of processes that affect the memory cells by their applications. *PFL* - an experimental process functional language comes out from pure functional languages, including an imperative programming environments [15]. *PFL* environments are manipulated neither in monadic manner [34] nor in an assignment-based manner. Instead of this, source form of a process functional program has strongly separated visible sets of environment variables (in type definitions) and invisible side-effect operations (in definitions). In this section we will consider just (pure) functions `f` and `g` (not processes) and main expression `main`, as introduced in Fig. 3.1

```
f :: Int -> Int
f x  = 2*x

g :: Int -> Int -> Int
g x y = f x + f y

main :: Int
main = g 2 3
```

Figure 3.1 Purely functional program `P`

PFL form of purely functional program `P` is identical to that in Haskell, using currying in application of functions, for example `(g 2 3)`, instead of `g(2,3)` – the form usual in imperative languages. The evaluation of program `P` proceeds by the reduction as follows:

$$
\begin{aligned}
\texttt{main} = \texttt{g 2 3} \\
&\Rightarrow \texttt{f 2 + f 3} \\
&\Rightarrow \texttt{2*2 + 2*3} \\
&\Rightarrow \texttt{10}
\end{aligned}
\qquad (3.1)
$$

The evaluation is the same if the program is written without function type definitions, see Fig. 3.2, because the types are derivable from definitions in Milner type system. Let us designate this function module definition `D`. Then the semantics of P and D is the same, i.e.

$$[\![\texttt{P}]\!] = [\![\texttt{D}]\!] \qquad (3.2)$$

```
f x  = 2*x

g x y = f x + f y

main = g 2 3
```

Figure 3.2 Function definition module `D`

Since the mutual position of the type definition and the definition of a function in a program is not significant, we may write all type definitions in separate type definition module $\mathtt{T_M}$, illustrated in Fig. 3.3.

```
f :: Int -> Int

g :: Int -> Int -> Int

main :: Int
```

Figure 3.3 Function type definition module $\mathtt{T_M}$

If applying the composition W to module $\mathtt{T_M}$ and `D` the composed program $W(\mathtt{T_M},\mathtt{D})$ is the source program in Fig. 3.4, then the semantics of P is the same as $W(\mathtt{T_M},\mathtt{D})$:

$$[\![\texttt{P}]\!] = [\![W(\mathtt{T_M},\mathtt{D})]\!] \qquad (3.3)$$

```
f :: Int -> Int

g :: Int -> Int -> Int

main :: Int

f x  = 2*x

g x y = f x + f y

main = g 2 3
```

Figure 3.4 Composed program $W(\mathtt{T_M},\mathtt{D})$

If `D` is an original module and $\mathtt{T_M}$ is an advice, which is added at join point before the first definition in `D` by default, then, in terms of aspect programming, W is a trivial weaver. This weaver is an identity, since, as follows from (3,2) and (3,3), it holds:

$$[\![W(\mathtt{T_M},\mathtt{D})]\!] = [\![\texttt{D}]\!] \qquad (3.4)$$

Let us consider polymorphic function type definitions in separated module in Fig. 3.5. Instead of type constants `Int` type variables are used.

```
f :: a -> a

g :: a -> a -> a

main :: a
```

Figure 3.5 Polymorphic type module $\mathtt{T_P}$

The same weaver W is used to compose $\mathtt{T_P}$ and `D` obtaining woven program $W(\mathtt{T_M},\mathtt{D})$, according to Fig. 3.6.

```
f :: a -> a

g :: a -> a -> a

main :: a

f x  = 2*x

g x y = f x + f y

main = g 2 3
```

Figure 3.6 Composed program $\mathcal{W}(\mathrm{T_P},\mathrm{D})$

Since during type-checking phase the monomorphic types for all function are derived, as in P, we may conclude, as for monomorphic case, that it holds

$$[\![\mathcal{W}(\mathrm{T_P},\mathrm{D})]\!] = [\![\mathrm{D}]\!] \qquad (3.5)$$

Informally, including the `aspect' to a purely functional definition module in the form of function type definitions (both monomorphic and polymorphic) does not affect evaluation at all, since this is the same as introduced in (3.1).

It may be noticed that functional programming style is out of our interest (clearly the form in Fig. 3.1 is the most appropriate form from this viewpoint). Here we are extremely interested in separating concerns in $\mathcal{PFL}$ with respect to aspect programming paradigm.

The importance of separating concerns into different modules grows up when considering additional aspects of computation. As shown in the next section, we are able slightly modify the type module without any change of the definition module, and then weave them changing the semantics of program P, i.e. the definition D. This fact is crucial in aspect programming.

4. STATE ASPECT

Suppose now a "small" change of the type definition module $\mathrm{T_P}$, according to Fig. 4.1, where u, v, and w are the environment variables.

```
f :: u a -> a

g :: v a -> w a -> a

main :: a
```

Figure 4.1 State aspect $\mathrm{T_S}$

In this way we have defined the state aspect of computation, since by $\mathrm{T_S}$ we require two things:

1. For all applications of f in D: before f is applied to an argument e, assign e to u and then use e as an argument. This follows from (u a) in the type definition for f.
2. For all applications of g in D: before g is applied to the first argument e1, assign e1 to v and then use e1 as the first argument of g., and before (g e1) is applied to the argument e2, assign e2 to w and then use e2 as the second argument of g. This follows from the type definition for g.

For example, (f 2) will perform assignment u:=2 (using Pascal notation), and then (f 2) will be evaluated as in purely functional case. Considering (g 2 3), it is guaranteed, that assignments v:=2 and w:=3 are performed before (g 2 3) is evaluated continuing by f 2 + f 3 evaluation.

It means that except a purely functional evaluation according to the reduction (3.1), additional side effect actions (assignments) are performed. Or, from another viewpoint, argument values of functions f and g are traced using three environment variables: u, v, and w.

However, the selection of join points is weak. Our pointcut designator can be expressed just informally, as follows:

Join points are all arguments of functions defined by a user, (i.e. except built-in operations).

Our joint points are identified with a very low flexibility, since there are no designators able to use quantifiers and/or logical operations in $\mathcal{PFL}$.

In this paper we will concentrate on advices, as "a parts of code" being used at join points. In this matter it is substantial to understand the weaving

$$\mathcal{W}(\mathrm{T_S},\mathrm{D}) \qquad (4.1)$$

which, using the same weaver $\mathcal{W}$ and the same definitions D as above produces the program $\mathrm{P_S}$ which evaluates differently than program P. Hence, new aspect $\mathrm{T_S}$ affects the semantics. Hence it holds

$$[\![\mathcal{W}(\mathrm{T_S},\mathrm{D})]\!] \neq [\![\mathrm{D}]\!] \qquad (4.2)$$

The woven form of program $\mathrm{P_S}$ is in Fig. 4.2.

According to Fig. 4.2 we have introduced three environment variables in an (imperative) environment, we have defined three functions in a class Env, and we apply them to each argument of user-defined functions. Let us consider first these applications informally.

```
env
  u^c:: a
  v^c:: a
  w^c:: a

class (Env b a) where
  u:: b -> a
  v:: b -> a
  w:: b -> a

instance (Env a a) where
  u x = let u^c=x in u^c
  v x = let v^c=x in v^c
  w x = let w^c=x in w^c

instance (Env () a) where
  u x =u^c
  v x =v^c
  w x =w^c

f :: a -> a

g :: a -> a -> a

main :: a

f x  = 2*x

g x y = f (u x) + f (u y)

main = g (v 2) (w 3)
```

Figure 4.2 Program $P_S = W(T_S, D)$

Corresponding to our requirements to all applications of f and g, defined by our informal pointcut above, we require the result of evaluation to be the same as in (3.1). The function of computation is preserved, if it holds

```
u e = e, v e = e, w e = e
```

for all expression e of a data type. It means that environment variables in PFL are not just cells of memories, but they are identities, if their arguments are of a data type.

Next, before an environment variable is applied to argument e, the argument e is stored to the variable (since the environment variable is not just an identity, but also a memory cell). This state aspect corresponds to assignments

$$u^c := e,\ v^c := e,\ w^c := e$$

for all expression e of a data type, where variables as cells are marked by ^{c} to distinct them from variables as functions. Hence, the application, such as (v e) evaluates in two subsequent steps s and e, which we express by a pair

```
(s; e)
```

where s may be an assignment or empty action, i.e. state action and e is an expression, which defines the (functional) value of application.

Then the complete definition of a variable v in terms of two aspects is as follows:

$$v\ x = (v^c{:=}x;\ x),\qquad \text{if } x \neq ()$$
$$v\ x = (\varepsilon;\ v^c),\qquad \text{if } x = ()$$

Semantically equivalent definition to that above is as follows:

Definition 4.1. Informal definition of environment variable

$$v\ x = (v^c{:=}x;\ v^c),\qquad \text{if } x \neq ()$$
$$v\ x = (\varepsilon;\ v^c),\qquad \text{if } x = ()$$

The latter better expresses the argument data flow through the variable. The second equation is not used in our examples, since here we work just with data values. But notice, that if an argument of a function would be control value, designated by (), then state is not affected (since state action is empty), and the application v () yields the data value having been stored in cell v^c.

The definition of v above is informal, since the value of the application is not the pair on right hand side, just the second item, we use imperative sequencing (;) and imperative assignment in a pair on right hand side of informal definition. But looking at Fig.4.2 it is easy to see, that it holds

$$(v^c{:=}x;\ v^c) = \text{let } v^c{=}x \text{ in } v^c$$
$$(\varepsilon;\ v^c) = v^c$$

Using informal definition for environment variable the program P_S is evaluated as follows:

```
main = g (v:=2;2) (w:=3;3)
       ⇒ f (u:=2;2) + f (u:=3;3)
       ⇒ 2*2 + 2*3
       ⇒ 10                     (4.3)
```

To simplify notation, we designate cells by u, v, and w, not using u^c, v^c, and w^c anymore. Except the function of computation is evaluated (the value of (v:=2;2) is 2, the value of (w:=3;3) is 3, etc.), program P_S traces all argument values used in applications of user-defined functions storing them to variables – external memory cells that belong to variable environment env of computation.

Since then functions affect the variable environment, they are rather processes than functions. That is why we call this paradigm process functional. However, in framework of this paper is more substantial, that weaving the module T_S and D the semantics of original module `D` will change, according to (4.2).

Notice that our "weaver" W performs compile time transformation, when producing $W(T_S, D)$. But the same W acts as identity when producing $W(D)$. In each case, the type checking is performed after weaving.

Further, as follows from evaluation of $W(T_S, D)$ we can say, that arguments of user-defined functions *are reflected* in variable environment performing the next sequence of assignments.

```
v:=2; w:=3; u:=2; u:=3;
```

The sequence above is true if all arguments are evaluated in the leftmost order and + is left associative operation. Some comments on this, and other problems associated with maintaining reflective information are introduced in the following section.

5. DISCUSSION

In this section we identify some problems coming out from the current state of process functional programming language, which is aimed to be adapted to an aspect programming language.

Currently we have developed a compiler from object-oriented *PFL* to both Haskell and Java languages. The purpose of *PFL* project was to provide a programming language, which would make open view to variable environment to a user as it is in imperative languages, and at the same time to preserve the approach coming out from purely functional languages, that the evaluation is defined by application of processes and functions, excluding the sequences of statements. As a result, *PFL* is a simple and an expressive language, and still more relaxed than Haskell, since function of computation can be affected by evaluation order.

The weaknesses of *PFL* language and its perspectives, from the viewpoint of aspect programming paradigm are as follows:

- The order of evaluation is fixed and it is supposed to be known to a programmer. Then aspect of evaluation order, which is associated with parallelism, cannot be defined separately. Since this aspect is highly dependent on target architecture, sometimes even at the level of built-in operations [6,33], it must be expressible explicitly.
- Nothing has been said about the use of reflected values in this paper. But *PFL* is capable for the definition of multi-threaded programs and the mechanism for accessing the values in environments is defined by application of an environment variable to control value. The updates can be performed in one thread and the accesses in the second thread.
- Using control values is possible but wrong programming praxis. One possible solution is to "tear" of purely functional programs is monadic approach. This is well disciplined but still just programming methodology, so including control values as a new control aspect seem to be more perspective.
- In this paper the mechanism of application of environment variables is used just to reflect the values of arguments. But it may be noticed, that the mechanism is very strong, because we may reflect not just values coming from computation, but also from an external environment, such as architecture resources.
- Or, it is possible to use the single variable for many points of a program. Then, if we use `v` instead of both `u` and `w` in T_s we would obtain the following tracing

```
v:=2; v:=3; v:=2; v:=3;
```

- Although *PFL* arrays are over the scope of this paper, process functional paradigm can be applied in backward direction. It means that it is possible to generate an application of a new generated variable to each expression instead of this expression, and then compose the set of variables into an array that "application" to a type substitutes this type in a function type definition. Then we would obtain something like this

```
v:=2; w:=3; u{0}:=2; u{1}:=3;
```

- Using *PFL*, the reflection interface is still not flexible enough, since of using just environment variables in type definitions. Extensions are the subject of our current research.
- At the time it is strong feeling that fixed number of abstraction levels is not sufficient enough to provide a general purpose aspect language, open to new aspects that can arise in the future.
- Currently no pointcuts can be defined in *PFL*. It is however clear that pointcuts must be defined rather over abstraction levels than according user requirements. Providing the appropriate syntax and semantics of pointcuts is crucial task, since they affect compile-time pre-weaving, and are related to reflection information when performing run time weaving.

6. CONCLUSION

In this paper we use the principle of composing multiple modules into target program by source-to-source transformation. Using simple tracing example we have shown the principle of the reflection of values in purely functional evaluation, to an external variable environment.

We also discuss briefly the use of values coming from external environment variables. It may be noticed that our type system unifies data and control types just for arguments of environment variables (the types are unified just in the type variable `b` in a generated `class Env b a`, otherwise not). This is the difference between *PFL* and Haskell.

Opposite to the specification approaches oriented to the correctness of programs [17,18,19], or specialized tools for time-critical systems [27,28], our approach supports the computational environments of the systems in a more open way. We take into account different levels of abstraction, working still at programming language level and, at the same time, at the level of programming paradigm.

Considering the aspects are crosscutting concerns of computation, pointcut designators must specify lexical, syntactic and semantic levels of an aspect language, the environmental properties and run-time events of computation. But this is still not sufficient, since it is necessary to prevent the situation, when adding a new aspect fails since of language restrictions.

The openness to dynamic aspects is the crucial property of an aspect language. In this paper we have presented the systematic manipulation with environments provided by process functional paradigm as a proposition for the development of an aspect process functional language considering computational reflection.

REFERENCES

[1] Andrews, J.: Process-algebraic foundations of aspect oriented programming.

http://citeseer.nj.nec.com/andrews01processalgebraic.html, 2001.

[2] 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.

[3] Avdicausevic, E., Mernik, M., Lenic, M., Zumer, V.: Experimental aspect-oriented language - AspectCOOL. Proceedings of 17th ACM symposium on applied computing, SAC 2002, pp. 943-947.

[4] Filman, R. E., Friedman, D. P.: Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[5] Hudák, 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] Jelšina, M., Vokorokos L., Sobota, B.: Parallel Computer Architecture of the MIMD Paradigm, Proc. of the III. Internal Scientific Conference of the Faculty of Electrical Engineering and Informatics, May 2003, Košice, pp. 35-36, ISBN 80-89066-65-8

[7] Kiczales, G. et al: An overview of Aspect J. Lecture Notes in Computer Science, 2072:327-355, 2001.

[8] 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.

[9] Kienzle, J. and Guerraoui, R.: Aspect oriented software development AOP: Does it make sense? The case of concurrency and failures. In B. Magnusson, editor, Proc. ECOOP 2002, pages 37-61. Springer Verlag, June 2002.

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

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

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

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

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

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

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

[17] Novitzká, V.: Computer Programming and Mathematics, Fifth International Scientific Conference „Electronics Computers and Informatics´2002“, 10.-11.10.2002, Košice-Herľany, Technická univerzita v Košiciach, 2002, 5, pp. 31-36, ISBN 80-7099-879-2

[18] Novitzká, V.: About the theory of correct programming. February 2003, Elfa s.r.o, Košice, 117pp. (in Slovak)

[19] Novitzká, V.: Mathematical language in programming, Acta Electrotechnica et Informatica, 3, 3, 2003, pp. 31-35, ISSN 1335-8243

[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. [editors]: Report on the Programming Language Haskell 98 - A Non-strict, Purely Functional Language. February 1999, 163 p.

[23] Porubän, J.: Profiling process functional programs. Research report DCI FEII TU Košice, 2002, 51.pp, (in Slovak)

[24] Porubän, J.: Time and space profiling for process functional language, Proceeding of the 7th Scientific Conference with International Participation: Engineering of Modern Electric '03 Systems, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 167-172, ISSN-1223-2106

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

[26] Sullivan, G. T.: Aspect-oriented programming using reflection and meta-object protocols. *Comm. ACM*, 44(10):95–97, Oct. 2001.

[27] Šimoňák, S., Hudák, Š.: Using Petri Nets and Process Algebra in FDT Interfacing, the Fifth International Scientific Conference „Electronic Computers and Informatics´2002“, October 2002, Košice - Herľany, 2002, pp. 8-13, 80-7099-879-2

[28] Šimoňák, S., Hudák, Š.: APC - Algebra of Process Components, EMES '03, May 29.-31. 2003., Felix Spa, Oradea, 2003, pp. 57-63, ISSN 1223 – 2106

[29] Václavík, P.: Abstract types and their implementation in a processs functional programming language. Research report DCI FEI TU Košice, 2002, 48.pp, (in Slovak)

[30] Václavík, P., Porubän, J.: Object Oriented Approach in Process Functional Language, Proceedings of the Fifth International Scientific Conference „Electronic Computers and Informatics´2002“, October 10.-11. 2002, Košice - Herľany, 2002, pp. 92-96, 80-7099-879-2

[31] Václavík, P.: The Fundamentals of a Process Functional Abstract Type Translation, Proceeding of the 7th Scientific Conference with International Participation: Engineering of Modern Electric '03 Systems, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 193-198, ISSN-1223-2106

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

[33] Vokorokos, L.: Data flow computing model: Application for parallel computer systems diagnosis, Computing and Informatics, **20**, (2001), 411-428.

[34] 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.

[35] Wand, M.: A semantics for advice and dynamic join points in aspect-oriented programming. Lecture Notes in Computer Science, **2196**: 45-57, 2001.

BIOGRAPHY

Ján Kollár 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 University of P.J. Šafárik 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, the dataflow systems, the educational systems, and the implementation of functional programming languages. Currently the subject of his research is the implementation of multi-paradigmatic languages.