# PARTIAL MONADIC APPROACH IN PROCESS FUNCTIONAL LANGUAGE

Ján KOLLÁR
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, tel. 095/602 4179, E-mail: Jan.Kollar@tuke.sk

**SUMMARY**

*This paper is devoted to manipulating the state in process functional language (PFL) using monadic approach [13,19], preserving at the same time the visibility of memory cells. Instead of exploiting just pure functional expressions, as it is done in Haskell, balanced binding of functional and state aspects of computation in PFL using monads is presented. Monadic approach in PFL is partial in the sense that it may be exploited at any hierarchical level of computation, concentrating to the points in which state manipulation and pure functional evaluation are switched. From this point of view, this paper contributes to a systematic joining of function and state aspects of computation.*

***Keywords:*** *Programming paradigms, imperative functional programming, aspect oriented programming, implementation principles, programming environments, control driven dataflow, referential transparency, side effects*

## 1. INTRODUCTION

Process functional paradigm is based on evaluation of processes that affects the memory cells by their applications [8,10]. *PFL* - an experimental process functional language, aimed to von Neumann machines is a successor of a dataflow language used to modeling and development of dataflow architecture [18].

Combining different programming paradigms into a singleton language, such as functional and imperative in Haskell with monads [14], or logic and functional in Oz [12,16], is a way in which a programming language is more expressible and reliable. On the other hand, such a combination may result to complicated and/or over abstracted language constructs, as well as to the need for underlying core language, which manipulates the architecture resources. Moreover, since the programming language itself does not solve a problem of a gap between modeling the system and its programming [3], the necessity of integrating the computer architecture resources, programming, specification and modeling of a designed system is the task of high interest.

In contrast to Haskell with monads, *PFL* variables are visible memory cells in explicit, implicit and object environment [11,15]. *PFL* arrays are partial spatial processes manipulated using array comprehensions akin to list comprehensions. *PFL* exploits both parametric polymorphism and abstract typing [17].

*PFL* models any imperative language [9]. The difference between *PFL* as an implementation language and *PFL* as a programming language is as follows: while the programming language must be deterministic, the implementation language may be non-deterministic.

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

Similarly as the non-deterministic grammar may produce the deterministic language, *PFL* provides some degree of freedom, considering different aspects of computation. This, of course, is not to say that it covers all potential aspects.

In this paper we will concentrate just on two aspects of computation - the function and the state, joined synchronously. We will show that a *PFL* explicit environment variable - a memory cell - is visible, even in a monad. Monads comprising visible computational spaces, as we hope, are useful when a memory is the subject of the design.

## 2. HASKELL MONAD

A monad is a triple *(M,unitM,bindM)* consisting of a type constructor *M* and a pair of polymorphic functions [19].

```
unitM :: a -> M a
bindM :: M a -> (a -> M b) -> M b
```

These functions must satisfy three laws, as follows.

Left unit: *(unitM a) `bindM` k = k a*
Right unit: *m `bindM` unitM = m*

Associative:
*m `bindM` (λa. (k a) `bindM` (λb. h b))*
*= (m `bindM` (λa. k a)) `bindM` (λb. h* b)

As an introductory example, let us consider a pure functional solution to a problem of counting the number of operations `Add` in expression

```
Add (Mul (Num 3) (Num 4))
    (Add (Num 2) (Num 6))
```

in terms of algebraic data type as follows

```
data Etree = Num Int
           | Add Etree Etree
```

```
          | Mul Etree Etree
```

**Example 2.1** Pure functional solution

```
expr (Num x) = x
expr (Add x y) = expr x + expr y
expr (Mul x y) = expr x * expr y

count (Num x) = 0
count (Add x y) = count x + count y
                  + 1
count (Mul x y) = count x + count y

main = "Value: " ++ show (expr t)
       ++ "  State: "
       ++ show (count t)
  where t = Add (Mul (Num 3)
                     (Num 4))
                (Add (Num 2)
                     (Num 6))
```

Evaluating `main` the result `Value:20 State:2` is obtained, since the value of expression is 20 and `Add` occurs in this expression two times. The disadvantage is that the expression tree is traced two times and the separate (although structurally very similar) function `count` must be defined.

A monadic (and purely functional) solution of the same problem in Haskell is introduced in the Example 2.2. It is based on the definition of monad *(S,unitS,bindS)* and on the systematic transformation of functions `expr` and `count` into a monadic form, in which they produce functions (of the type *Int -> (a, Int)*), instead of values.

**Example 2.2** Monadic solution in Haskell

```
type S a = Int -> (a, Int)

unitS :: a -> S a
unitS a = g where g s = (a,s)

bindS :: S a -> (a -> S b) -> S b
m `bindS` k = g
 where g s0 = k a s1
        where (a,s1) = m s0

expr (Num x)   = unitS (Num x)
expr (Add x y) = expr x `bindS` k
 where
  k (Num i) = expr y `bindS` h
   where
    h (Num j) = count `bindS` r
       where
        r () = unitS (Num (i+j))
expr (Mul x y) = expr x `bindS` k
 where
  k (Num i) = expr y `bindS` h
   where
    h (Num j) = unitS (Num (i*j))

count :: S ()
count = g where g s = ((),s+1)

showt (Num i) = show i

main = "Value: " ++ showt t ++
       "  State: " ++ show s
 where
  (t,s) = (expr
           (Add (Mul (Num 3)
                     (Num 4))
                (Add (Num 2)
                     (Num 6))
           ) ) 0
```

The solution above is adopted from [19], excluding lambda abstractions and let expressions, and replacing them by explicit local functions. Nevertheless, it is still not so easy to realize, in which cell (or cells) the count of operations `Add` is incremented. Monads in Haskell support stateful computation using purely functional approach, hiding memory cells to a programmer, and performing guaranteed sequencing of actions by application dependence. Since architecture resources in Haskell are affected indirectly via C routines calls, there is no need (and no opportunity) to affect them by a programmer explicitly.

In contrast to Haskell, *PFL* is a language, which associates the functional abstraction with the physical resources of computer architecture. Being the resources the subject of the system design, the visibility of memory cells is crucial.

In the next sections, we will define a monad *(M,unitM,bindM)* comprising a single visible memory cell in terms of *PFL*, but first let us introduce the essential concept of *PFL* memory variables.

## 3. *PFL* VARIABLES

The definitions introduced in this section are not a part of *PFL* script. To simplify the notation, we will use $v$ for a variable as a memory cell, and $v[m]$ for a variable containing a well-defined value $m$. If a variable is not initialized, it contains undefined value $\perp$, which is written as $v[\perp]$. A variable $v$ in figures is designated by large circle, containing either $\perp$ or defined values marked by small circles.

The crucial for understanding *PFL* approach is two-fold semantics of a variable $v$. From one point of view, it is a cell, as mentioned above. At the same time, the environment variable is a mapping

$$v[a_{\perp}] :: \tilde{a} \to a \qquad (1)$$

which may be read as follows: An environment variable $v$ is a cell comprising a value of any type $a$, including $\perp$ ($a\perp$), and it is mapping from values of the type $a$ including control value () ($\tilde{a}$) to the values of the type $a$. Corresponding to input arc for argument and output arc to the value of the mapping above, see Fig.1 and Fig.2 we distinct the situation,

when the argument is a data value and when it is the control value.

Let the argument is a data value. Then the definition of an environment variable is given by two equations (u.1), and (u.2).

$$v[\perp] \quad n \quad = \quad n[n] \qquad \text{(u.1)}$$

$$v[m] \quad n \quad = \quad n[n] \qquad \text{(u.2)}$$

According to (u.1), see the Fig. 1a), when $v$ containing undefined value is applied to a data value $n$, the output value is $n[n]$, which means that the value $n$ is produced as a result, and it is assigned to a variable $v$ before (expressed by $[n]$). It means that the value $n$ is assigned to yet not initialized memory cell $v$ and then pushed onto the stack. The equation (u.2), see Fig. 1b) differs just in that the incoming value replaces old value $m$, but both cases mean clearly *the update* of a cell $v$, changing the definition to $v[n]$.

According to the definition above, the update has its functional aspect - the identity as well as its state transition aspect - that may be expressed by $v[\perp] \Rightarrow v[n]$, and $v[m] \Rightarrow v[n]$, respectively.

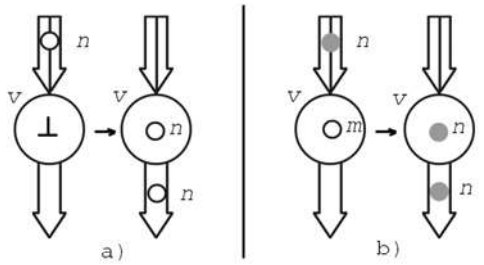


**Fig. 1** The update of environment variable $v$

Now, let the argument is the control value (), marked by a black dot in Fig. 2.

The definition of *the access* is given by the equations (a.1) and (a.2) as follows

$$v[\perp] \quad () \quad = \quad \perp[\perp] \qquad \text{(a.1)}$$

$$v[m] \quad () \quad = \quad m[m] \qquad \text{(a.2)}$$

In this case the value in a variable $v$ is just pushed onto the stack.

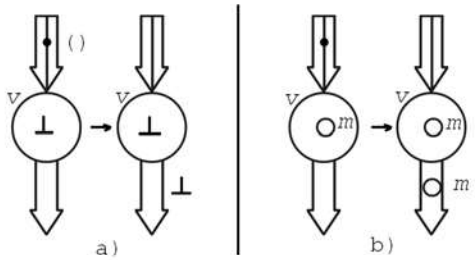


**Fig. 2** The access of environment variable $v$.

The functional aspect is expressed by a "constant", but, since $v$ may change its definition during computation, this constant is generally referentially non-transparent. On the other hand the state aspect is unaffected, since the state transitions are $v[\perp] \Rightarrow v[\perp]$, and $v[m] \Rightarrow v[m]$, respectively.

We attend that a variable never occurs in $\mathcal{PFL}$ source expressions. Instead of that, binding environment variables to pure functional arguments is the matter of $\mathcal{PFL}$ type definitions of processes and environmental applications are performed implicitly. This binding is discussed in the next section. It may be also noticed that the case (a.1) must be prevented, otherwise (see the Fig. 2a)) an expression is evaluated using undefined value. It is possible to detect the use of undefined values during the type checking of $\mathcal{PFL}$ programs, but a monadic approach is the second alternative. Although $\mathcal{PFL}$ variables are related to mutable abstract types [4], the above explanation in terms of control-driven dataflow [10] seems to be more "natural" to a user.

## 4. ENVIRONMENT BINDING

In $\mathcal{PFL}$ source script, the environment variables are bound to a set of processes by their type definitions. Suppose the next definition of `ua`, similar to identity function, which type definition however has argument type `v a`, instead of `a`. The type expression `v a` is a syntactic shortcut for type expression $v[a_{\perp}] :: \tilde{a} \rightarrow a$, binding the mapping $v[a_{\perp}]\,\tilde{a}$ to a purely functional argument of the type $a$. Hence, `ua` is not a simple identity but it is rather the composition of a memory cell `v` as a function and the identity `id`, see the Fig. 3.

```
ua :: v a -> a
ua x = x
```

Then the value of `(ua 3)` is 3, updating the cell $v$ by the value 3 before, and the value of `(ua ())` is a value having been assigned by the last update before. It means that *ua* represents an effect process which integrates both update and access actions depending on either data or control as an argument.

If `v` occurs in the type definitions of multiple processes, they share it, and then it is probably more appropriate to define type synonym, in the form

```
type  V a =  v a
```

changing the type definition as follows

```
ua  :: V a -> a
ua  x = x
```

It is even possible to associate a variable `v` with a physical memory location, in the form as follows:

```
type  V a =  v  a at #1277746
```

if needed, which means that the cell v is located at memory address #1277746.

Although the flexibility of such an approach may be criticized, potentially *PFL* does not require any target language except a machine language. Therefore *PFL* is a wide-range language, which benefits from well-disciplined functional approach in programming, not disqualifying architecture resources from the design.

A programmer may even think about the process ua, as defined internally as follows

```
ua  :: a -> a
ua  x = x
```

If ua above is applied to an argument, it invokes an invisible application of a memory variable to this argument. For example, the evaluation of source form (ua 3) is internally performed as (ua (v[m⊥] 3)) and a source application (ua ()), is internally performed as (ua (v[m] ())).

## 5. VISIBLE VARIABLE MONAD

In *PFL*, there is a natural boundary between purely functional evaluation and the referentially non-transparent imperative execution: pure functions have no environment variables used in their optional type definitions while processes have at least one environment variable used in their obligatory type definitions.

While *PFL* functions may be higher order, *PFL* processes are first order, which implies possible compile time transformations into the internal form.

In particular, monadic functions unitM and bindM prevent the need for considering call dependency in expressions, since they provide a uniform interface between purely functional evaluation and imperative computation. Also for this reason we are interested in an ability for monadic *PFL* form, preserving however the visibility of memory variables.

Let the type M is defined as follows

```
type  M a =  v a -> a
```

comprising variable v. Then a process ua may be defined, as follows

```
ua  :: M a
ua  x = x
```

which, when applied, performs imperative action - either the update or the access of a cell v as mentioned above. The result is of the type a.

Now, let c is a constant function, which takes any value of the type a and produces the process ua (i.e. the constant such as the entry address of ua body).

```
c :: a -> M a
c x = ua
```

The value of referentially transparent function c is referentially non-transparent action ua performed on the variable v.

The aim of unitM is to take a value into its corresponding representation in a monad [19]. In terms of *PFL* the aim of unitM is to assign a value to the variable v by application of ua, and then to produce the action ua. It is easy to see that the value of application c (ua x) for value x is the constant - the computation of the type M a. It is exactly what is needed for unitM. Then we have

```
unitM :: a -> M a
unitM x = c (ua x)
```

illustrated in the Fig. 3.

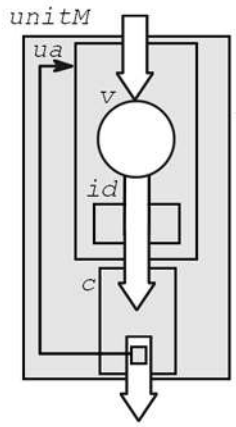


**Fig. 3** Function UnitM

A single argument function defined by an expression $e^k[x]$ on right hand side of its definition using the argument value x would evaluate to the value $e^k[x]$. The function k in monadic form is defined by the application (unitM $e^k[x]$), i.e. by the application (c(uae$^k$[x])) which means that the value of $e^k[x]$ (provided that it is of data type) is just assigned to v and the result is the computation ua.

If we define two monadic functions, say h and k as follows

```
h :: a -> M b
h x = unitM e^h[x]

k :: a -> M b
k x = unitM e^k[x]
```

it is substantial that applying them elsewhere in a script they affect the same variable v and they produce the same constant ua.

The last step is to define the function bindM, which, when applied, request the application of a monadic function, such as unitM, k, h, yielding its first argument m which is the constant ua. The second argument is not an application but rather a monadic function k as a value. In terms of Haskell, this function is applied to the monad M a producing

monad `M b`. In terms of *PFL*, the function `k` is applied to the old value of environment variable `v`, assigning the result to `v` and then producing the value of `m`, which is equal to `ua`. The old value is clearly selected by expression `m()`, (since `m()=ua()`).

In this way the referential transparency of `bindM` is preserved, since both arguments and value (`m=ua`) are constants. On the other hand we have the activation mechanism visible, since it is nothing more than the use of control value.

The function `bindM` is defined as follows and it is depicted in the Fig. 4.

```
bindM :: M a -> (a -> M b) -> M b
bindM m k = k (m ())
```

Formal proof that (M,unitM,bindM) is a monad is introduced below. In the proof of left and right unit we will consider the value changed in *v* in brackets.

**Proof 5.1**

Left unit law:

| | |
|---|---|
| (*unitM a*) `bindM` *k* | *v*[⊥] |
| = *k* ((*unitM a*) ()) | |
| = *k* ((*c* (*ua a*)) ()) | |
| ⇒ *k* ((*c a*) ()) | *v*[*a*] |
| ⇒ *k* (*ua* ()) | *v*[*a*] |
| ⇒ *k a* | *v*[*a*] |

It means that *k* is evaluated using the argument *a* stored in *v*, as required.

Right unit law:

Suppose *m*=*c*(*ua a*), i.e. *v*[*a*]

| | |
|---|---|
| *m* `bindM` *unitM* | *v*[*a*] |
| = *unitM* (*m* ()) | *v*[*a*] |
| ⇒ *unitM a* | *v*[*a*] |
| ⇒ *m* | *v*[*a*] |

It means that $unitM$ does not change the monad, as required.

Associative law:

L = *m* `bindM` (λ*a*. (*k a*) `bindM` (λ*b*. *h b*))
= (λ*a*. (*k a*) `bindM` (λ*b*. *h b*)) (*m* ())
= (λ*a*. (λ*b*. *h b*) ( (*k a*) ())) (*m* ())
⇒ (λ*b*. *h b*) ( (*k* (*m* ())) ())
⇒ *h* ((*k* (*m* ())) ())

R = (*m* `bindM` (λ*a*. *k a*)) `bindM` (λ*b*. *h b*)
= (λ*b*. *h b*) ((*m* `bindM` (λ*a*. *k a*)) ())
= (λ*b*. *h b*) (((λ*a*. *k a*) (*m* ())) ())
⇒ *h* (((λ*a*. *k a*) (*m* ())) ())
⇒ *h* ((*k* (*m* ())) ())

Proving associative law, both its sides are reduced to the same application dependence - the expression *h* ((*k* (*m* ())) ()). It means that the order in which the arguments of *bindM* are evaluated is not significant.

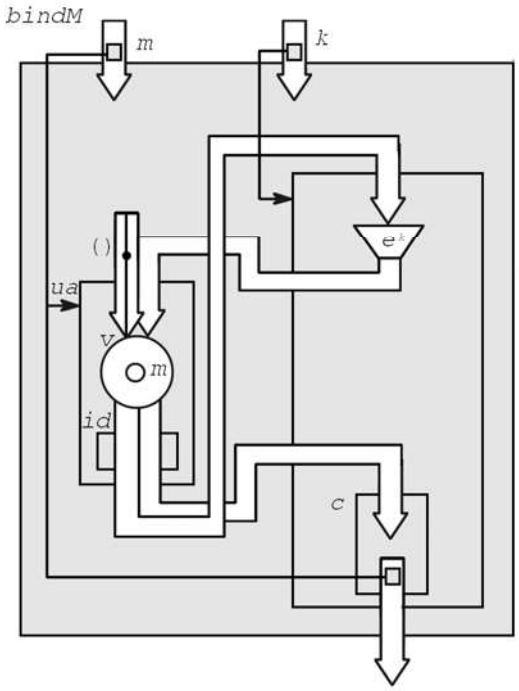


**Fig. 4** Function bindM

## 6. PARTIAL MONADIC APPROACH

In this section we present *PFL* partial monadic solution and "pure" *PFL* solution of the same task as in Examples 2.1 and 2.2, respectively.

Our goal is to perform minimum changes in `expr` defined in Example 2.1. In Example 6.1 and 6.2, the visible environment variable `s` in the type definition `S` has crucial role, since it is memory cell, in which the number of `Adds` is incremented. Since the operation `Add` affects the state, and is performed by operation `(+)`, it is sufficient to change its definition `` `iadd` ``, as can be seen below.

**Example 6.1** Partial monadic *PFL* solution

```
type S a = s a -> a

ua x = x
c x = ua

unitS :: Int -> S Int
unitS x = c (ua x)

bindS :: S Int -> (Int -> S Int)
                  -> S Int
bindS m k = k (m ())

incr :: Int -> S Int
```

```
incr x = unitS (x + 1)

expr (Num x)   = x
expr (Add x y) = expr x
                 `iadd`
                 expr y
 where
  iadd  = add(unitS 5 `bindS` incr)
  add _ = (+)
expr (Mul x y) = expr x * expr y

e _ = expr (Add (Mul (Num 3)
                     (Num 4))
           (Add (Num 2) (Num 6)))

st x = (x, unitS () ())

main = "Value: " ++ show t ++
       "  State: " ++ show s
  where
    (t,s) = (st (e (unitS 0)))
```

In contrast to Haskell form, the appropriate sequencing must be guaranteed by explicit application dependence. It means, that `unitS 0` must be evaluated first initializing the variable to 0, then the expression is evaluated `e(unitS 0)` and then the pair containing both the value of expression and the state is constructed (by `st (e (unitS 0))`, see the pair `(t,s)` definition in `main`.

As a result, the variable `s` is visible in monad `(S,unitS,bindS)`, and minimal change of `expr` has been obtained.

The *PFL* solution not using monad is introduced in the Example 6.2. This solution is equivalent to monadic *PFL* one, except that we may use the value of `ua 0` application instead of `c (ua 0)` when evaluating the arguments for functions `add` and `e` that argument value, marked by underscore character, is not significant.

**Example 6.2** Pure *PFL* solution

```
ua :: s a -> a
ua x = x

expr (Num x)   = x
expr (Add x y) = expr x
                 `iadd`
                 expr y
  where
    iadd  = add (ua () + 1)
    add _ = (+)
expr (Mul x y) = expr x * expr y

e _ = expr (Add (Mul (Num 3)
                     (Num 4))
           (Add (Num 2) (Num 6)))

st x = (x, ua ())

main = "Value: " ++ show t ++
       "  State: " ++ show s
  where
    (t,s) = (st (e (ua 0)))
```

In both cases above the state is changed at two different hierarchical levels of computation - the initialization of variable `s` and reading the final state is performed on the top level and the incrementing is performed inside a purely functional grain defined by pure function `expr` (since `` `iadd` `` is referentially transparent).

## 7. CONCLUSION

A partial monadic approach in *PFL* is not so restrictive to a programmer as Haskell approach, since it may be applied at any level of computation separately. The environment variables are visible and the memory organization may be the subject of software design. For the limited scope of this paper it was however not possible to introduce here more advanced computational spaces.

Nevertheless, two aspects of computation - the function and the state - are bound in a very systematic way. In terms of aspect oriented programming (AOP) [6], `` `iadd` `` is a primitive and synchronous case of a join point which has two-fold semantics - it is an addition operation considering functional aspect and it is count incrementing operation considering state aspect.

In AOP, instead of manipulating different aspects of computation in a single program producing tangled code, each aspect is described separately, considering join points in which the aspects cross-cuts. Then, in principle, the scripts are composed by automatic tool, called weaver, producing the implementation with respect to all aspects having been considered. AspectJ, as an extension to Java [5], supports the ideas of AOP in praxis. There are still many problems open. It is argued, that AOP using AspectJ fails for transition systems [7]. On the other hand, formal descriptions for aspect programs using process algebras [1] as well as denotational semantics for recursive procedure calls [20] have been published. Considering a set of domain specific languages for different aspects in AspectCool [2] seems to be better alternative for AOP, but multi-language AOP is not so attractive for a user as single language concept.

Our idea to exploiting *PFL* as aspect oriented modeling language is as follows. Having a single language L in *PFL* experimental form, it is necessary to derive its subsets $\{L_i \mid i=1\ldots n\}$, adding a minimum new syntactic constructs, such that all languages $L_i(A_i)$ ($L_i$ with respect of aspect $A_i$) are deterministic, while $L_j(A_k)$ are non-deterministic for all $j \neq k$. Then, instead of weaving scripts we may think about weaving partially deterministic subsets of a single language producing its deterministic implementation.

From this point of view, this paper contributes to a systematic joining of function and state aspects of computation in a synchronous way.

## 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] Havlice, Z.: The Integrated CASE System Based on the Modeling Tools Description Language. Proc. Scient. Conf. CEI'99, October 1999, Herľany, Slovakia, pp. 68-73.

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

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

[6] 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. Springer Verlag, 1997.

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] Peton Jones, S.L., Hughes, J. **[**editors**]**: Report on the Programming Language Haskell 98 - A Non-strict, Purely Functional Language. February 1999, 163 p.

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

[16] Smolka, G.: The Oz programming model, In Jan van Leeuwen, editor, Computer Science Today, Lecture Notes in Computer Science 1000, Springer-Verlag, Berlin, 1995, pp. 324-343.

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

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

[19] 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.

[20] 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** (Assoc. Prof.) was born in 1954. He received his MSc. summa cum laude in 1978 and his PhD. in Computing Science in 1991. In 1978-1981 he was with the Institute of Electrical Machines in Košice. In 1982-1991 he was with the Institute of Computer Science at the 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 process functional paradigm and its application in the high performance computing.