Computing and Informatics, Vol. 25, 2006, 61–80

FROM EAGER $\mathcal{PFL}$ TO LAZY HASKELL*

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 21 January 2005; revised 30 November 2005
Communicated by Igor Walukiewicz

Abstract. The state of a system is expressed using $\mathcal{PFL}$, a process functional language, in an easily understandable manner. The paper presents $\mathcal{PFL}$ environment variable – our basic concept for the state manipulation in the process functional language. Then we introduce the style in which stateful systems are described using monads and state transformers in pure lazy functional language Haskell. Finally, we describe our approach to lazy state manipulation in $\mathcal{PFL}$ and correspondence between state manipulation in $\mathcal{PFL}$ and the one in a pure lazy functional language Haskell. The proposed translation from eager $\mathcal{PFL}$ to a lazy Haskell provides an opportunity to exploit laziness for process functional programs and furthermore for imperative programs. The approach described in this paper was used in implemented $\mathcal{PFL}$ to Haskell code generator.

Keywords: Process functional language, imperative functional programming, lazy state evaluation, environment variables, monads, state transformers, Haskell

1 INTRODUCTION

A purely functional language is concise, composable and extensible [14]. The reasoning about pure functional programs that are defined in terms of expressions and

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

evaluated without side effects is much simpler than the reasoning about imperative programs when describing stateful systems.

From the viewpoint of systems design, it seems more appropriate (at least to most of programmers) to describe systems using an imperative language, expressing the state explicitly by variables as memory cells. Although the reliability of an imperative approach may be increased using object oriented paradigm, it solves neither the problem of reasoning about the functional correctness of fine grains of computation, since they are still affected by subsequent updating the cells in a sequence of assignments, nor the problem of profiling the program to obtain the execution satisfying the time requirements of a user.

An imperative functional approach using monads [15, 18], implemented in Haskell [16], prevents the use of assignments still providing the opportunity to a programmer for manipulating the state in a disciplined and well-defined manner. Although the scripts written in Haskell are sometimes obscure, the idea of hiding the assignments in mutable abstract types and optimising the lazy evaluation is inspirative. On the other hand, impure languages, such as Standard ML [13], offer efficiency benefits and sometimes make a more compact mode of expression [18].

Regarding aspect oriented paradigm [1, 4, 12, 20], we are strongly interested in development of a multi-paradigmatic language, which preserves a software engineering approach to manipulating the visible environments as it is in imperative languages, at the same time providing the source form of expressions as it is in purely functional languages. As we feel, to be able to extend a language adding new aspect, it is necessary to have a highly reflective and a very simple source form of the language without assignments.

From the viewpoint of a user, the layout of $\mathcal{PFL}$ – an experimental process functional language is somewhere between imperative and pure functional languages, since the variable environment is visible to a user, and the source definitions of processes are purely functional, i.e. without assignments and without environment variables [5, 6]. This supports, as we believe, the simplification of the systems design and, at the same time, the simpler reasoning about the systems. The concept of $\mathcal{PFL}$ variable environment is presented in Section 2. Detailed classification of $\mathcal{PFL}$ variable environments can be found in [7].

It has been proved that imperative program constructions such as loops can be easily transformed into $\mathcal{PFL}$ program constructions [8]. That is why our approach shows the possibility of lazy evaluation of imperative programs.

The goal of this paper is twofold. In Section 4 we show the state manipulation using monads in Haskell. In Section 5 we present how the state can be manipulated lazily in $\mathcal{PFL}$ and we show informally the semantic equivalence of both approaches. Our progress is presented in context of simplified version of $\mathcal{PFL}$. Simplified $\mathcal{PFL}$ is an extended subset of $\mathcal{PFL}$ programming language but without the loss of generality. It is defined in Section 3. The presented approach was used in implemented $\mathcal{PFL}$ to Haskell code generator. Extended example in Section 6 concludes the paper showing the results of the proposed transformation from eager $\mathcal{PFL}$ to lazy Haskell.

2 $\mathcal{PFL}$ VARIABLE ENVIRONMENT

$\mathcal{PFL}$ type system comprises unit type () as it is in Haskell. This type comprises just control value, representing the control [5]. It means for example that it is impossible to mix data and control arguments for constructors of algebraic data types. Let T be a data type. Then the type $\widetilde{T}= T \cup ()$ ranges over a data type and unit type. The $\mathcal{PFL}$ process is similar to a function. Definition of a $\mathcal{PFL}$ process is the same as definition of a pure function. The only difference is in its type definition. Type definitions for processes are obligatory. The type definition of a process extends syntax and semantics of a pure function type definition. The type definition of a process comprises either () for an argument or the value type, or an argument type in the form $x\ T$, where x is an environment variable and T is a data type. Examples of process type definitions are provided later. The idea of incorporating some aspects in function type definitions is also presented in Clean with its uniqueness attributes [17].

Functions are first-class values in $\mathcal{PFL}$. On the other hand, processes are not. Process cannot be passed as an argument to a function or returned as the result of a function. There is no partial process application. This approach has been chosen to simplify identification of program parts manipulating the state. The $\mathcal{PFL}$ static analyzer finds parts of a program affected by the state – processes. Data gathered by $\mathcal{PFL}$ static analyzer are used in compiler (Section 5).

The well known and commonly accepted concept of the variable environment in both imperative and impure functional languages is as follows. The variable environment **Env** is a mapping from variables to their values. If $\rho\ ::\ \alpha\ \rightarrow\ \beta$ is environment and $a \in \alpha$, $b \in \beta$ then the update expression is as follows.

$$(\rho\ [\ a \mapsto b\ ])\ x = \begin{cases} b & \text{, if } x = a \\ \rho\ x & \text{, if } x \neq a \text{ and } \rho \neq \emptyset \\ \perp & \text{, if } x \neq a \text{ a and } \rho = \emptyset \end{cases}$$

Symbol $\emptyset$ is used to define empty environment. The variable environment **Env** is defined using the update expression.

$$\begin{array}{l} \mathbf{Env}\ =\ \mathbf{Var} \rightarrow \mathbf{Value} \\ access\ ::\ \mathbf{Var} \rightarrow \mathbf{Env} \rightarrow \mathbf{Value} \\ access\ x\ e\ =\ e\ x \\ update\ ::\ \mathbf{Var} \rightarrow \mathbf{Value} \rightarrow \mathbf{Env} \rightarrow \mathbf{Env} \\ update\ x\ v\ e\ =\ e\ [\ x \mapsto v\ ] \end{array}$$

In the type definition of **Env**, **Var** is a domain of environment variables and **Value** denotes a disjunctive unification of all $\mathcal{PFL}$ data types values.

A syntactic form of a variable attributed type $x\ T$ as an argument type of a process allows a user to consider the visible variable environment in role of input memory gate of process bodies, consisting of a subset of environment variables –

memory cells, that are possibly shared by multiple definitions of processes in the same scope.

The processes may be applied either to control values, and computed using values accessed from the environment variables, or to data values and computed using them, updating the environment variables by this value before.

Concluding, the state is defined by the environment that internally conforms to that used in imperative and impure functional languages, but for the reasons of its binding to process bodies, the $\mathcal{PFL}$ semantics is the same as the semantics of monadic approach, as we will show in Section 5. In $\mathcal{PFL}$, the access and the update of environment is uniform in each scope, by processes defined in the same scope as the environment – global, local or object one.

Suppose simple $\mathcal{PFL}$ process `sum` is defined in main scope, which has two environment variables `a` and `b` defined in a process type definition, as follows.

```
sum :: a Int -> b Int -> Int
sum x y = x + y
```

Suppose an application `(sum 3 4)` exists somewhere in an expression of a $\mathcal{PFL}$ program, such that `sum` is accessible (for example in a definition of a process in main scope). Then the result of the application will be updating the environment variables `a` by the value 3, updating the environment variables `b` by the value 4, as an additional side effect to the evaluation of pure function. It means the value of the application will be 7.

This is so because in the first stage of the translation the definition above is transformed to the form of pure function, as follows

```
sum :: Int -> Int -> Int
sum x y = x + y
```

while each application of `sum` is transformed to the form, in which environment variable is applied to corresponding argument. For example, `(sum 3 4)` is transformed to `(sum (a 3) (b 4))`.

On the other hand, if original argument is control value `()`, then the transformed application may be for example `(sum (a ()) (b 5))`, provided that the source form is `(sum () 5)`. Then the value of `y` will be 5 (updating `b` by 5), and the value of `x` will be the value accessed from environment variable `a` by the application `(a ())`. Provided that the value in `a` is 6, the value of `(sum (a ()) (b 5))` will be 11. If no value has been assigned to `a` before, then the value of the application is undefined.

Since the access and update instances are applied implicitly, i.e. they never occur in the process definitions, the state change strongly depends on the order in which the arguments of a process are evaluated. In Section 5 we present the transformation of eager $\mathcal{PFL}$ programs to the programs evaluated lazily with preserving the determinism of computation.

We attend that abstract syntax of simplified $\mathcal{PFL}$ is in the form after the source to source transformation mentioned above, which is something as weaving [1, 4, 12, 20] side-effect aspect of computation, implicitly into each application of a process.

3 SIMPLIFIED $\mathcal{PFL}$

For the purposes of the paper we present simplified $\mathcal{PFL}$ as an extended subset of $\mathcal{PFL}$ programming language, its abstract syntax and operational semantics. Using this $\mathcal{PFL}$ subset, our approach to imperative program lazy evaluation is presented and can be formalized. This approach can be extended to all $\mathcal{PFL}$ language constructs. The meta-variables and categories for simplified $\mathcal{PFL}$ language are as follows:

$$\begin{array}{llll} Prg \in \textbf{Program} & Def \in \textbf{Definition} & e \in \textbf{Expr} & x \in \textbf{Var} \\ f \in \textbf{FncName} & \oplus \in \textbf{Primitive} & y \in \textbf{EnvVar} & C \in \textbf{Constructor} \end{array}$$

The meta-variables can be primed or subscripted. The syntactic category **Primitive** defines strict primitive operations like elementary arithmetic operations. The syntactic category **Var** represents identifiers and syntactic category **EnvVar** represents the identifiers of environment variables. The syntactic category **Constructor** comprises constructors of algebraic types. Numbers, integer or real, may be viewed as nullary constructors and are included in the syntactic category **Constructor** ($\textbf{Int}, \textbf{Float} \subseteq \textbf{Constructor}$).

Program in simplified $\mathcal{PFL}$ consists of processes, functions and main expression which is evaluated during the program execution. Abstract syntax of simplified $\mathcal{PFL}$ is as follows.

$$\begin{array}{lcll} Prg & ::= & Def\ \textbf{main}\ =\ e & \\ Def & ::= & f\ x_1\ \ldots\ x_n\ =\ e\ \ Def & \\ & | & \varepsilon & \\ e & ::= & x & Variable \\ & | & f & Function \\ & | & e_1\ \oplus\ e_2 & Primitive \\ & | & y\ () & Access \\ & | & y\ e & Update \\ & | & C\ e_1\ \ldots\ e_n & Constructor \\ & | & e_1\ e_2 & Application \\ & | & \textbf{case}\ e\ \textbf{of}\ \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ e_i\}_{i=1}^{n} & Case \end{array}$$

Simplified $\mathcal{PFL}$ contains construction $y\ ()$ for accessing environment variable and construction $y\ e$ which is used to update variable with value of an expression. $\mathcal{PFL}$ processes application can be transformed to simplified $\mathcal{PFL}$ according to the transformation scheme $\mathcal{T}_a$. Let p be a $\mathcal{PFL}$ process with type definition

$$p\ ::\ \overline{T_1} \rightarrow \ldots \rightarrow \overline{T_i} \rightarrow \ldots \overline{T_n} \rightarrow \widetilde{T}$$

where $n \geq 1$ and $\overline{T_i} = y\ T_i \mid\ \widetilde{T_i}$ and $\widetilde{T_i},\ \widetilde{T}$ are $\mathcal{PFL}$ data type or unit type (), T_i is data type and y is environment variable. Then application of a process p in the form

$p\ e_1 \ldots e_i \ldots e_n$ is transformed to simplified $\mathcal{PFL}$ as follows:

$$\mathcal{T}_a\ [\![p\ e_1 \ldots e_i \ldots e_n]\!] = p\ \mathcal{T}_a'\ [\![e_1]\!] \ldots \mathcal{T}_a'\ [\![e_i]\!] \ldots \mathcal{T}_a'\ [\![e_n]\!]$$

where

$$\mathcal{T}_a'\ [\![e_i]\!] = \begin{cases} e_i & \text{, if } \overline{T_i} = \widetilde{T_i} \\ y\ () & \text{, if } \overline{T_i} = y\ T_i \text{ and } e_i = () \\ y\ e_i & \text{, if } \overline{T_i} = y\ T_i \text{ and } e_i \neq () \end{cases}$$

The value $v \in \mathbf{Value}$ of an expression is either a lambda abstraction or a value of an algebraic type

$$\begin{array}{rcl} v & ::= & C\ v_1\ \ldots\ v_n \\ & | & \lambda x.e \end{array}$$

where $n \geq 0$.

The runtime state is defined by environments env_v, env_e. Environment env_f is created during the compile time and represents function/process environment. Environment env_v represents the heap for the values of lambda variables and env_e is a set of memory cells for storing values of environment variables.

$$\begin{array}{rcl} env_f \in \mathbf{Env_f} & = & \mathbf{FncName} \rightarrow \mathbf{Expr} \\ env_v \in \mathbf{Env_v} & = & \mathbf{Var} \rightarrow \mathbf{Value} \\ env_e \in \mathbf{Env_e} & = & \mathbf{EnvVar} \rightarrow \mathbf{Value} \\ (env_v, env_e) = s \in \mathbf{State} & = & \mathbf{Env_v} \times \mathbf{Env_e} \end{array}$$

The semantics of simplified $\mathcal{PFL}$ function/process definitions is in Figure 1 and the semantic rules for expressions of simplified $\mathcal{PFL}$ are defined in Figure 2. All rules are named corresponding to the abstract syntax. The predicate *matches* for pattern matching and operator *extract* for extracting the i^{th} item value of the structure constructed by $C\ v_1\ \ldots v_i \ldots\ v_n$ are defined as follows.

$$\begin{array}{l} \mathit{matches}\ v\ (C\ x_1\ \ldots\ x_n) \Leftrightarrow v = C\ v_1\ \ldots\ v_n \\ \mathit{extract}\ (C\ v_1\ \ldots v_i\ \ldots\ v_n)\ i = v_i\ \text{, where } 1 \leq i \leq n. \end{array}$$

The notation

$$env_f \vdash \langle e, s \rangle \rightarrow (v, s')$$

defines that expression e is evaluated in environment env_f considering the state s and produces the value v and new state s'. The state is defined by variable environments.

4 MONADS AND STATE TRANSFORMERS

In this section we illustrate the monadic approach to state manipulation in Haskell [3, 11, 15, 18, 19]. Both monads and state transformers have had big impact on functional programming in the last few years. State transformers and their theory are used in the paper as a basis for the transformation from eager $\mathcal{PFL}$ to lazy Haskell.

$$\mathcal{D} : \textbf{Definition} \rightarrow \mathbf{Env_f} \rightarrow \mathbf{Env_f}$$

$$\frac{\langle Def, env_f \rangle \rightarrow_D env'_f}{\langle f\ x_1\ \ldots\ x_n\ =\ e\ \ Def, env_f \rangle \rightarrow_D env'_f[f \mapsto \lambda x_1.\ \ldots\ .\lambda x_n.e]}$$

$$\langle \varepsilon, env_f \rangle \rightarrow_D env_f$$

Fig. 1. The semantics of definitions

A monad [11, 18] is a triple (M, *return*, *then*) consisting of a type constructor M and two polymorphic functions *return* and *then*.

The state transformer is a function which, given a state, produces a pair: a value and a new state [10, 11]. Using Haskell notation, let us define type synonym as follows:

$$\textbf{type}\ ST\ s\ a = s \rightarrow (a, s).$$

Using state transformer, the computation is the transformation of one state to the new state, which is constructed by modifying the old one.

Now, let us define the state transformer in terms of monad ($ST, returnST$, $thenST$) where operations *returnST* and *thenST* are defined as follows:

$$\begin{array}{l} returnST\ ::\ a \rightarrow ST\ s\ a \\ returnST\ a\ s = (a,\ s) \\ \\ thenST\ ::\ ST\ s\ a \rightarrow (a \rightarrow ST\ s\ b) \rightarrow ST\ s\ b \\ thenST\ m\ k\ s = k\ x\ s'\ \textbf{where}\ (x,\ s') = m\ s. \end{array}$$

Function *thenST* is defined by the expression using local definition in **where** clause. The definitions above are purely functional. Informally, function *returnST* takes a result of computation a and state s and produces pair (a,s), which can be used for the next computation. Function *thenST* is used for composition of functions in monadic form.

Although the monad semantics is well-defined and the state in Haskell is manipulated lazily [10], programs using monads in Haskell [16] become sometimes obscure even for an experienced programmer. That is why more expressive language construct, such as monad comprehensions, are provided to a user.

5 LAZY STATE EVALUATION

$\mathcal{PFL}$ is a superset of a purely functional language. $\mathcal{PFL}$ purely functional program may comprise variable environment, not however the applications of processes to

$$\mathcal{E} : \mathbf{Expr} \to \mathbf{Env_f} \to \mathbf{State} \to \mathbf{Value} \times \mathbf{State}$$

$$env_f \vdash \langle x, (env_v, env_e)\rangle \to (env_v\ x, (env_v, env_e)) \qquad \textit{Variable}$$

$$\frac{env_f \vdash \langle env_f\ f, s\rangle \to (v, s')}{env_f \vdash \langle f, s\rangle \to (v, s')} \qquad \textit{Function}$$

$$\frac{\begin{array}{c} env_f \vdash \langle e_1, s\rangle \to (v_1, s_1) \\ env_f \vdash \langle e_2, s_1\rangle \to (v_2, s_2) \end{array}}{env_f \vdash \langle e_1 \ \oplus\ e_2, s\rangle \to (v_1 \oplus v_2, s_2)} \qquad \textit{Primitive}$$

$$env_f \vdash \langle y\ (), (env_v, env_e)\rangle \to (env_e\ y, (env_v, env_e)) \qquad \textit{Access}$$

$$\frac{env_f \vdash \langle e, (env_v, env_e)\rangle \to (v, (env'_v, env'_e))}{env_f \vdash \langle y\ e, (env_v, env_e)\rangle \to (v, (env'_v, env'_e[y \mapsto v]))} \qquad \textit{Update}$$

$$\frac{\begin{array}{l} env_f \vdash \langle e_1, s\rangle \to (v_1, s_1) \\ \ldots \\ env_f \vdash \langle e_i, s_{i-1}\rangle \to (v_i, s_i) \\ \ldots \\ env_f \vdash \langle e_n, s_{n-1}\rangle \to (v_n, s_n) \end{array}}{env_f \vdash \langle C\ e_1\ \ \ldots e_i \ldots\ \ e_n, s\rangle \to (C\ v_1\ \ \ldots v_i \ldots\ \ v_n, s_n)} \qquad \textit{Constructor}$$

$$\frac{\begin{array}{l} env_f \vdash \langle e_1, s\rangle \to (\lambda x.e, s') \\ env_f \vdash \langle e_2, s'\rangle \to (v_2, (env_v, env_e)) \\ env_f \vdash \langle e[\overline{x}/x], (env_v[\overline{x} \mapsto v_2], env_e)\rangle \to (v, s'') \end{array}}{env_f \vdash \langle e_1\ e_2, s\rangle \to (v, s'')} \qquad \textit{Application}$$

$$\frac{\begin{array}{l} env_f \vdash \langle e, s\rangle \to (v', (env'_v, env'_e)) \\ env_f \vdash \langle e_j, (env'_v[\overline{x_1} \mapsto extract\ v'\ 1] \ldots \\ \qquad\qquad \ldots [\overline{x_{m_j}} \mapsto extract\ v'\ m_j], env'_e)\rangle \to (v'', s'') \\ matches\ v'\ (C_j\ x_1\ \ \ldots\ \ x_{m_j}) \end{array}}{env_f \vdash \langle \mathbf{case}\ e\ \mathbf{of}\ \{C_i\ x_1 \ldots x_{m_i} \to\ \ e_i\}_{i=1}^{n}, s\rangle \to (v'', s'')} \qquad \textit{Case}$$

$$env_f \vdash \langle \lambda x.e, s\rangle \to (\lambda x.e, s) \qquad \textit{Lambda}$$

Fig. 2. The semantics of expressions

expressions of unit type. Then the environment does not affect the function of computation. For the reasons of strict semantics of variable updates, the eager evaluation for $\mathcal{PFL}$ programs is supposed, as a starting point for further optimisation. In this matter we proceed in backward direction, as it is in lazy languages, where a program is represented in a lazy form and then it is optimised using strictness analysis.

It may be noticed that one of the reasons for the different approaches is that the state in a lazy language is passed to evaluation explicitly via arguments while in $\mathcal{PFL}$ the state represented environment is affected implicitly, i.e. by the application of processes accessing and/or updating the environment variables.

In particular, we are interested in the semantical equivalence of $\mathcal{PFL}$ and monadic Haskell languages. We present its essential principle based on

- the transformation of a simplified $\mathcal{PFL}$ program into purely functional Haskell monadic form, and
- the transformation of eager representation of a $\mathcal{PFL}$ program into 'the most' lazy form.

The transformation can be also extended to local and object variable environments.

The source program in simplified $\mathcal{PFL}$ is designated by P, program after the transformation in Haskell language is designated by P':

$$P \leadsto P'.$$

5.1 State Transformer Definition

State manipulation in Haskell is performed by state transformers. In transformed program P', the state transformer is defined by the new algebraic type, as follows.

$$\textbf{data}\ StateTrans\ s\ a = ST\ (s \to (a, s)).$$

The instance of class $Monad$ for $StateTrans$ type is defined as follows:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{instance}\ Monad\ (StateTrans\ s)\ \textbf{where}} \\
\quad (ST\ m)\ >>=\ k & = & ST\ (\lambda s. & \textbf{let}\ (a, sp) = m\ s\ \textbf{in} \\
 & & & \textbf{let}\ (ST\ q) = k\ a\ \textbf{in}\ q\ sp) \\
\quad return\ a & = & ST\ (\lambda s. & (a, s)).
\end{array}$$

5.2 State Representation

State of the system in $\mathcal{PFL}$ is defined via current values of environment variables. The count and types of environment variables in a program are known during the compile time and do not change during the program runtime. Every environment variable is specified in the $\mathcal{PFL}$ source program explicitly. Let the program P in simplified $\mathcal{PFL}$ contain environment variables

$$v_1 :: T_1, v_2 :: T_2, \ldots, v_n :: T_n \qquad , n \geq 0$$

where v_i is environment variable and T_i is its type in P, i.e. v_i is included at least in one process definition of the program P. Type T_i is a data type, not the unit type. In transformed program P' in Haskell the variable environment is defined as n-tuple

$$\textbf{type}\ State\ =\ (T'_1, T'_2, \ldots, T'_n)$$

where $T'_i,\ 1 \leq i \leq n$ is Haskell type with the same domain as $\mathcal{PFL}$ T_i.

5.3 State Manipulation

In Haskell program P', two functions will be defined for every environment variable y_i located source $\mathcal{PFL}$ program P. The first one is the function $accessY_i$ for accessing value of the variable and the second one is $updateY_i$ for assigning the value into variable. These two functions manipulate the state and are defined as state transformers.

$$\begin{array}{lcl} accessY_i & :: & StateTrans\ State\ T'_i \\ accessY_i & = & ST\ (\lambda(y_1, \ldots, y_i, \ldots, y_n).\ (y_i, (y_1, \ldots, y_i, \ldots, y_n))) \\ \\ updateY_i & :: & T'_i \rightarrow StateTrans\ State\ T'_i \\ updateY_i\ val & = & ST\ (\lambda(y_1, \ldots, y_i, \ldots, y_n).\ (val, (y_1, \ldots, val, \ldots, y_n))) \end{array}$$

5.4 Process and Function Transformations

In the transformed program P', the order of the evaluation of arguments must be preserved, corresponding to leftmost innermost reduction strategy defined for eager $\mathcal{PFL}$ program evaluation.

Let f be $\mathcal{PFL}$ process or function with definition

$$\begin{array}{l} f\ ::\ \overline{T_1} \rightarrow \overline{T_2} \rightarrow \ldots \rightarrow \overline{T_n} \rightarrow \widetilde{T} \\ f\ x_1\ x_2\ \ldots x_n = e \end{array}$$

where $n \geq 0$ and $\overline{T_i} = y\ T_i \mid\ \widetilde{T}_i$, where $\widetilde{T}_i,\ \widetilde{T}$ is $\mathcal{PFL}$ data type or unit type.

Let us suppose that there is equivalent Haskell data type T'_i and T' for $\mathcal{PFL}$ data type T_i and T.

Type definition of $\mathcal{PFL}$ process or function can be transformed from P to P' as follows:

$$\begin{array}{l} \mathcal{T}\ [\![f\ ::\ \overline{T_1} \rightarrow \overline{T_2} \rightarrow \ldots \rightarrow \overline{T_n} \rightarrow \widetilde{T}\ ]\!] = \\ \quad f\ ::\ T'_1 \rightarrow T'_2 \rightarrow \ldots \rightarrow T'_n \rightarrow StateTrans\ State\ T'. \end{array}$$

Process/function definition is transformed as follows:

$$\mathcal{D}\ [\![f\ x_1\ x_2\ \ldots x_n = e\ ]\!] = f\ x_1\ x_2\ \ldots x_n = \mathcal{E}\ [\![e]\!].$$

Let us assume that a process or a function application exist somewhere in $\mathcal{PFL}$ source program, as follows:

$$f\ e_1\ e_2\ \ldots\ e_n.$$

The application above is transformed to Haskell using the monadic operations as follows

$$\mathcal{E}\ [\![f\ e_1\ e_2\ \ldots\ e_n]\!] = \begin{array}{l} \mathbf{do}\ \{ \\ \qquad \mathcal{E}'[\![e_1]\!]; \\ \qquad \mathcal{E}'[\![e_2]\!]; \\ \qquad \ldots \\ \qquad \mathcal{E}'[\![e_n]\!]; \\ \qquad f\ v_1\ v_2\ \ldots\ v_n \\ \} \end{array}$$

where

$$\mathcal{E}'[\![e_i]\!] = \begin{cases} y_i \leftarrow \mathcal{E}\ [\![e_i]\!] & ,\ \text{if}\ \overline{T_i} = \widetilde{T_i} \\ y_i \leftarrow \mathit{accessY} & ,\ \text{if}\ \overline{T_i} = y\ T_i \wedge e_i = () \\ y_i \leftarrow \mathcal{E}\ [\![e_i]\!];\ \mathit{updateY}\ y_i & ,\ \text{if}\ \overline{T_i} = y\ T_i \wedge e_i \neq () \end{cases}$$

The transformation scheme has two drawbacks.

- It is not possible to transform all functions into monadic form, such as the library functions. In addition to this, it is not necessary to transform functions which do not manipulate the state.
- Program is always defined as a sequence of statements even, but it is not necessary – in case of program grains that do not manipulate the state.

Considering the above-mentioned facts, the transformation scheme can be improved. The improved transformation scheme is defined for simplified $\mathcal{PFL}$ presented in Section 3. Let us define the predicate $(aff\ e)$. If $(aff\ e)$ holds, then the evaluation of the expression e depends on the state or it changes the state, otherwise the state is not affected by the e. The predicate $(aff\ e)$ is calculated during the static analysis of a program in the compiler. Two sets are constructed for all expressions in a program. AV(e) is the set of environment variables which can be accessed during the evaluation of expression e. UV(e) is the set of environment variables which can be updated during the evaluation of expression e.

$$
\begin{array}{lcl}
\mathrm{AV}(x) & = & \emptyset \\
\mathrm{AV}(f) & = & \mathrm{AV}(env_f\ f) \\
\mathrm{AV}(e_1\ \oplus\ e_2) & = & \mathrm{AV}(e_1) \cup \mathrm{AV}(e_2) \\
\mathrm{AV}(y\ ()) & = & \{y\} \\
\mathrm{AV}(y\ e) & = & \mathrm{AV}(e) \\
\mathrm{AV}(C\ e_1\ \ldots\ e_n) & = & \mathrm{AV}(e_1) \cup \ldots \cup \mathrm{AV}(e_n) \\
\mathrm{AV}(e_1\ e_2) & = & \mathrm{AV}(e_1) \cup \mathrm{AV}(e_2) \\
\mathrm{AV}(\mathbf{case}\ e\ \mathbf{of}\ \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ e_i\}_{i=1}^{n}) & = & \mathrm{AV}(e) \cup \mathrm{AV}(e_1) \cup \ldots \cup \mathrm{AV}(e_n) \\
\mathrm{AV}(\lambda x.e) & = & \mathrm{AV}(e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{UV}(x) & = & \emptyset \\
\mathrm{UV}(f) & = & \mathrm{UV}(env_f\ f) \\
\mathrm{UV}(e_1\ \oplus\ e_2) & = & \mathrm{UV}(e_1) \cup \mathrm{UV}(e_2) \\
\mathrm{UV}(y\ ()) & = & \emptyset \\
\mathrm{UV}(y\ e) & = & \{y\} \cup \mathrm{UV}(e) \\
\mathrm{UV}(C\ e_1\ \ldots\ e_n) & = & \mathrm{UV}(e_1) \cup \ldots \cup \mathrm{UV}(e_n) \\
\mathrm{UV}(e_1\ e_2) & = & \mathrm{UV}(e_1) \cup \mathrm{UV}(e_2) \\
\mathrm{UV}(\mathbf{case}\ e\ \mathbf{of}\ \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ e_i\}_{i=1}^{n}) & = & \mathrm{UV}(e) \cup \mathrm{UV}(e_1) \cup \ldots \cup \mathrm{UV}(e_n) \\
\mathrm{UV}(\lambda x.e) & = & \mathrm{UV}(e)
\end{array}
$$

The predicate $(aff\ e)$ is defined using the calculated sets $\mathrm{AV}(e)$ and $\mathrm{UV}(e)$:

$$(aff\ e) \Leftrightarrow \mathrm{AV}(e) \cup \mathrm{UV}(e) \neq \emptyset.$$

Let f be a $\mathcal{PFL}$ process or function with definition

$$
\begin{array}{l}
f\ ::\ \overline{T_1} \rightarrow \overline{T_2} \rightarrow \ldots \rightarrow \overline{T_n} \rightarrow \widetilde{T} \\
f\ x_1\ x_2\ \ldots x_n = e.
\end{array}
$$

Then modified transformation scheme has the form

$$
\begin{array}{l}
\mathcal{T}\ [\![f\ ::\ \overline{T_1} \rightarrow \overline{T_2} \rightarrow \ldots \rightarrow \overline{T_n} \rightarrow \widetilde{T}\ ]\!] = \\
\quad f\ ::\ T_1' \rightarrow T_2' \rightarrow \ldots \rightarrow T_n' \rightarrow T',\ \text{if } not\ (aff\ e) \\
\quad f\ ::\ T_1' \rightarrow T_2' \rightarrow \ldots \rightarrow T_n' \rightarrow StateTrans\ State\ T',\ \text{otherwise.}
\end{array}
$$

$$
\mathcal{D}\ [\![f\ x_1\ x_2\ \ldots x_n = e\ ]\!] = \begin{cases} \boxed{f\ x_1\ x_2\ \ldots x_n = \mathcal{E}_{expr}[\![e]\!]} & \text{if } \ not\ (aff\ e) \\ \boxed{f\ x_1\ x_2\ \ldots x_n = \mathbf{do}\ \{\ \mathcal{M}[\![e]\!]\ \}} & \text{otherwise.} \end{cases}
$$

$$
\mathcal{M}\ [\![e]\!] = \begin{cases} \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ return\ \mathcal{E}_{expr}[\![e]\!]; \end{array}} & \text{if } \ not\ (st\ e) \\ \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ \mathcal{E}_{expr}[\![e]\!]; \end{array}} & \text{otherwise.} \end{cases}
$$

Just processes are transformed to monadic form, not the functions. The transformation scheme $\mathcal{E}$ for $\mathcal{PFL}$ expressions produces a pair. The first item of the pair is a Haskell expression containing state transformers *sts*. The state changes are made by these transformers before the expression is evaluated. The second item of the pair is a Haskell expression *expr* which should be evaluated producing the value of an expression. The expression *expr* can be a state transformer.

$$\mathcal{E}[\![e]\!] = \begin{array}{|c|}\hline sts \\ \hline expr \\ \hline \end{array}$$

$$\begin{aligned} \mathcal{E}_{sts} &= fst \circ \mathcal{E} \\ \mathcal{E}_{expr} &= snd \circ \mathcal{E} \end{aligned}$$

Let us define predicate $(st\ e)$. If $(st\ e)$ holds, then the transformation scheme $\mathcal{E}[\![e]\!]$ produces expression *expr* in the form of a state transformer, otherwise not. Predicate *st* is defined as follows:

$$\begin{array}{lcl}
st\ x & = & false \\
st\ f & = & aff\ f \\
st\ \oplus & = & false \\
st\ (y\ ()) & = & false \\
st\ (y\ e) & = & false \\
st\ C & = & false \\
st\ (e_1\ e_2\ \ldots\ e_n) & = & st\ e_1 \\
st\ (\textbf{case}\ e\ \textbf{of}\ \{C_i\ x_1 \ldots x_{m_i} \to\ e_i\}_{i=1}^{n}) & = & aff\ e_1 \vee \ldots \vee aff\ e_n.
\end{array}$$

It can be seen from the definition above that $(st\ e)$ holds only if $(aff\ e)$ holds. Also, if *not* $(aff\ e)$ holds then *not* $(st\ e)$ holds. The transformation scheme for expressions is presented on Figure 3 and Figure 4. Symbol $\overline{\triangledown}$ used in transformation scheme represents unique lambda variable. If the symbol is used in both items of a pair (in *sts* and *expr*), then it denotes the same variable. The symbol $\emptyset$ used in transformation denotes empty *sts* item of a pair.

5.5 Properties of Transformation

This section concludes the transformation presenting the properties of the transformation with sketch of proofs.

Property 1. Expressions that do not manipulate the state are not transformed to state transformer form by the presented scheme.

If the expression e does not manipulate the state then *not* $(aff\ e)$ and *not* $(st\ e)$ holds. According to the definition of the predicate $(aff\ e)$ it is clear that any subexpression within the expression e does not manipulate the state. Expressions that do not manipulate the state surely do not contain the constructions for accessing $y\ ()$ or updating $y\ e$ the variable environment. It can be seen from the transformation

$$\mathcal{E}\llbracket x \rrbracket = \begin{array}{|c|}\hline \emptyset \\ \hline x \\ \hline \end{array} \qquad \mathcal{E}\llbracket f \rrbracket = \begin{array}{|c|}\hline \emptyset \\ \hline f \\ \hline \end{array} \qquad \mathcal{E}\llbracket y\ () \rrbracket = \begin{array}{|l|}\hline \overline{\mathrm{v}} \leftarrow accessY; \\ \hline \overline{\mathrm{v}} \\ \hline \end{array}$$

$$\mathcal{E}\llbracket y\ e \rrbracket = \begin{cases} \begin{array}{|l|}\hline \mathcal{E}_{sts}\llbracket e \rrbracket \\ \overline{\mathrm{v}} \leftarrow updateY\ \ \mathcal{E}_{expr}\llbracket e \rrbracket; \\ \hline \overline{\mathrm{v}} \\ \hline \end{array} & ,\ \text{if}\ \ not\ (st\ e) \\ \\ \begin{array}{|l|}\hline \mathcal{E}_{sts}\llbracket e \rrbracket \\ \overline{\mathrm{v}} \leftarrow \mathcal{E}_{expr}\llbracket e \rrbracket; \\ updateY\ \ \overline{\mathrm{v}}; \\ \hline \overline{\mathrm{v}} \\ \hline \end{array} & ,\ \text{otherwise} \end{cases}$$

$$\mathcal{E}\llbracket e_1\ e_2\ \ldots\ e_n \rrbracket = \begin{array}{|l|}\hline \mathcal{E}_{sts}\llbracket e_1 \rrbracket \\ \mathcal{E}'_{sts}\llbracket e_2 \rrbracket \\ \ldots \\ \mathcal{E}'_{sts}\llbracket e_n \rrbracket \\ \hline \mathcal{E}_{expr}\llbracket e_1 \rrbracket\ \mathcal{E}'_{expr}\llbracket e_2 \rrbracket\ \ \ldots\ \ \mathcal{E}'_{expr}\llbracket e_n \rrbracket \\ \hline \end{array}$$

$$\mathcal{E}\llbracket C\ e_1\ \ldots\ e_n \rrbracket = \begin{array}{|l|}\hline \mathcal{E}'_{sts}\llbracket e_1 \rrbracket \\ \ldots \\ \mathcal{E}'_{sts}\llbracket e_n \rrbracket \\ \hline C\ \mathcal{E}'_{expr}\llbracket e_1 \rrbracket\ \ \ldots\ \ \mathcal{E}'_{expr}\llbracket e_n \rrbracket \\ \hline \end{array}$$

$$\mathcal{E}\llbracket e_1\ \oplus\ e_2\ \rrbracket = \begin{array}{|l|}\hline \mathcal{E}'_{sts}\llbracket e_1 \rrbracket \\ \mathcal{E}'_{sts}\llbracket e_2 \rrbracket \\ \hline \mathcal{E}'_{expr}\llbracket e_1 \rrbracket\ \ \oplus\ \ \mathcal{E}'_{expr}\llbracket e_2 \rrbracket \\ \hline \end{array}$$

$$\mathcal{E}'\llbracket e \rrbracket = \begin{cases} \begin{array}{|l|}\hline \mathcal{E}_{sts}\llbracket e \rrbracket \\ \hline \mathcal{E}_{expr}\llbracket e \rrbracket \\ \hline \end{array} & ,\ \text{if}\ \ not\ (st\ e) \\ \\ \begin{array}{|l|}\hline \mathcal{E}_{sts}\llbracket e \rrbracket \\ \overline{\mathrm{v}} \leftarrow \mathcal{E}_{expr}\llbracket e \rrbracket; \\ \hline \overline{\mathrm{v}} \\ \hline \end{array} & ,\ \text{otherwise} \end{cases}$$

Fig. 3. The transformation scheme for expressions

$$\mathcal{E}[\![\textbf{case } e \textbf{ of } \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ e_i\}_{i=1}^{n}]\!] =$$

$$= \begin{cases} \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ \hline \textbf{case } \mathcal{E}_{expr}[\![e]\!] \textbf{ of } \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ \mathcal{E}_{expr}[\![e_i]\!]\}_{i=1}^{n} \end{array}} & , \begin{array}{l} \text{if } \mathit{not}\ (\mathit{st}\ e) \wedge \\ \wedge\ \mathit{not\ aaf} \end{array} \\ \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ \hline \textbf{case } \mathcal{E}_{expr}[\![e]\!] \textbf{ of } \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ \mathcal{M}[\![e_i]\!]\}_{i=1}^{n} \end{array}} & , \begin{array}{l} \text{if } \mathit{not}\ (\mathit{st}\ e) \wedge \\ \wedge\ \mathit{aaf} \end{array} \\ \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ \overline{\mathrm{v}} \leftarrow \mathcal{E}_{expr}[\![e]\!]; \\ \hline \textbf{case } \overline{\mathrm{v}} \textbf{ of } \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ \mathcal{E}_{expr}[\![e_i]\!]\}_{i=1}^{n} \end{array}} & , \text{ if } \mathit{st}\ e \wedge \mathit{not\ aaf} \\ \boxed{\begin{array}{l} \mathcal{E}_{sts}[\![e]\!] \\ \overline{\mathrm{v}} \leftarrow \mathcal{E}_{expr}[\![e]\!]; \\ \hline \textbf{case } \overline{\mathrm{v}} \textbf{ of } \{C_i\ x_1 \ldots x_{m_i} \rightarrow\ \mathcal{M}[\![e_i]\!]\}_{i=1}^{n} \end{array}} & , \text{ if } \mathit{st}\ e \wedge \mathit{aaf} \end{cases}$$

$$\mathit{aaf} = \mathit{aff}\ e_1 \vee \ldots \vee \mathit{aff}\ e_n$$

Fig. 4. The transformation scheme for expressions – continued

scheme for expressions that $\mathcal{E}_{sts}[\![e]\!]$ is empty and $\mathcal{E}_{expr}[\![e]\!]$ is not a Haskell state transformer.

Property 2. Pure $\mathcal{PFL}$ function is transformed to a pure non-monadic Haskell function.

Pure $\mathcal{PFL}$ function is a function which does not manipulate state. Let f be a pure function with definition $f\ x_1\ x_2\ \ldots x_n = e$, then $\mathit{not}\ (\mathit{aff}\ e)$ holds. According to the property 1 and transformation scheme for definitions the function is transformed to a non-monadic form.

Property 3. In the transformed program P', the order of the evaluation of arguments is preserved, corresponding to leftmost innermost reduction strategy defined for eager $\mathcal{PFL}$ program evaluation.

Property 4. The Haskell program P' is semantically equivalent to $\mathcal{PFL}$ program P.

Informally, the last two properties were essentials for the transformation scheme definition. Coming out from the semantics of Haskell monad and state transformers and $\mathcal{PFL}$ variable environment semantics we have defined the presented transformation. Formally, they can be proved using the semantics of Haskell programming language and simplified $\mathcal{PFL}$ semantics, showing the semantical equivalence of program P in $\mathcal{PFL}$ and the Haskell program P'.

It means that $\mathcal{PFL}$ programs can be evaluated lazily as those written in purely functional languages even the process functional programs have side effects.

6 EXTENDED EXAMPLE

The final example presents results from translation in already implemented $\mathcal{PFL}$ compiler. A fragment of $\mathcal{PFL}$ program P

```
incrX :: x Int -> Int -> Int
incrX x y = x+y

setXY :: x Int -> y Int -> ()
setXY x y = ()

swapXY :: x Int -> y Int -> ()
swapXY x y = setXY y x
...
swapXY () ()
...
incrX () 5
...
```

is transformed to Haskell form P', as follows.

```
data StateTrans s a = ST (s -> (a,s))

instance Monad (StateTrans s) where
  (ST m) >>= k = ST (\s -> let (a,sp)=m s in let (ST q)=k a in q sp)
  return a = ST (\s -> (a,s))

type State = (Int,Int)

accessX :: StateTrans State Int
accessX = ST (\(x,y) -> (x,(x,y)))

updateX :: Int -> StateTrans State Int
updateX xn = ST (\(x,y) -> (xn,(xn,y)))

accessY :: StateTrans State Int
accessY = ST (\(x,y) -> (y,(x,y)))

updateY :: Int -> StateTrans State Int
updateY yn = ST (\(x,y) -> (yn,(x,yn)))

incrX :: Int -> Int -> Int
```

```
incrX x y = x+y

setXY :: Int -> Int -> ()
setXY x y = ()

swapXY :: Int -> Int -> StateTrans State ()
swapXY x y = do {
                v1 <- updateX y;
                v2 <- updateY x;
                return (setXY v1 v2)
             }
...
do {
   v1 <- accessX;
   v2 <- accessY;
   swapXY v1 v2
}
...
do {
   v1 <- accessX;
   return (incrX v1 5)
}
...
```

7 CONCLUSION

In this paper, we have presented the way in which visible variables are bound to expressions in $\mathcal{PFL}$ – a process functional language. Introducing the state manipulation using monads in Haskell, we have shown that the state manipulation by the application of $\mathcal{PFL}$ processes is equivalent to that using monads. As a result of our transformation of $\mathcal{PFL}$ programs can be expressed in terms of Haskell monads, which means that process functional paradigm supports the laziness, providing the opportunity for transformations needed when profiling $\mathcal{PFL}$ programs in both sequential and parallel environments.

Application of functional programming languages looks very promising even in real time and embedded systems. Mostly functional language, like Hume [2], is suitable for design of embedded systems because of its time and space predictable behavior.

The subject of our current research is to exploit the process functional paradigm for integrating functional, imperative and aspect oriented methodology, using simple, uniform and still practical basis, appropriate for source-to-source transformations, reasoning on the behavior and verification experiments. Currently we have implemented the compiler from simplified $\mathcal{PFL}$ to both Java and Haskell languages. The

Haskell target code is about of the same length as source $\mathcal{PFL}$ code, while Java's code is about six times longer. Using $\mathcal{PFL}$, the level of abstraction has increased, preserving all abilities of imperative languages, including the visibility of all environments, providing a single tool for affecting the state in the form of the application of processes.

REFERENCES

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

[2] Hammond, K.—Michaelson, G. J.: Hume: A Domain-Specific Language for Real-Time Embedded Systems. Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, Erfurt, Germany, September 2003, Springer-Verlag Lecture Notes in Computer Science, ISBN 3-540-20102-5, 2003, pp. 37–56.

[3] Hudak, P.: Mutable Abstract Data Types – or – How to Have Your State and Munge It Too. Research Report YALEU/DCS/RR-914, Yale Universtity, Department of Computer Science, December 1992.

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

[5] Kollár, J.: Process Functional Programming. Proc. 33rd Spring International Conference MOSIS '99 – ISM '99, Information Systems Modeling, Rožnov pod Radhoštěm, Czech Republic, April 27–29, ACTA MOSIS No. 74, 1999, pp. 41–48.

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

[7] Kollár, J.—Václavík, P.—Porubän, J.: The Classification of Programming Environments. Acta Universitatis Matthiae Belii, Math No. 10, 2003, pp. 53–64.

[8] Kollár, J.—Novitzká, V: Semantical Equivalence of Process Functional and Imperative Programs, Acta Polytechnica Hungarica, Vol. 1, 2004, No. 2, pp. 113–124.

[9] Kollár, J.—Porubän, J: Static Evaluation of Process Functional Programs. Proc. of EMES 2001 – The International Conference on Engineering of Modern Electric Systems, May 24–26, 2001, Oradea, Romania, pp. 93–98

[10] Launchbury, J.: Lazy Imperative Programming. Proceedings ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIGPL '92.

[11] Launchbury, J.—Peyton Jones, S. L.: State in Haskell. In Lisp and Symbolic Computation 8, 1995, pp. 293–342.

[12] 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.

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

[14] Peyton Jones, S. L.: The Implementation of Functional Programmnig Languages. Prentice-Hall, 1987, 445 pp.

[15] Peyton Jones, S. L.—Wadler, P.: Imperative Functional Programming. 20th Symposium on Principles of Programming Languages, ACM Press, Charlotte, North Carolina, January 1993.

[16] Peyton Jones, S. L. ed.: Haskell 98 Language and Libraries, The Revised Report. Cambridge University Press, April 2003, 270 pp.

[17] Plasmeijer, R.—Eekelen, M.: Concurrent Clean Language Report Version 2.1. University of Nijmegen, November 2002.

[18] Wadler, P.: Monads for Functional Programming. Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.

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

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

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 has been 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 months 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 implementation of programming languages. Currently he is working in the area of multi-paradigmatic languages, with respect of aspect paradigm. He is the author of $\mathcal{PFL}$ – a process funtional 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 tools for process functional programming language. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming.

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 was implemented object oriented version of $\mathcal{PFL}$ language. Currently the subject of his research is the application of process functional paradigm in aspect oriented programming.