Semantical Equivalence of Process Functional and Imperative Programs

Ján Kollár

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Jan.Kollar@tuke.sk

Valerie Novitzká

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
Valerie.Novitzka@tuke.sk

Abstract. Source-to-source transformations play crucial role in weaving multiple aspects of computation in aspect languages. Except that expressing imperative programs in the uniform form of expressions simplifies these transformations, this form is useful from the viewpoint of recognizing different aspects of computation at any level of program structure. In this paper we present the relation between imperative language and PFL – a process functional language, which manipulate environment variables in a side-effect manner, still preserving a purely functional principle based on evaluating expressions. Using an example of an imperative structured program, we will show the semantical equivalence of process functional and imperative programs. As a result, fine grained PFL form for picking out potential join points in imperative programs is obtained.

Keywords. Programming paradigms, programming languages, side effects, process functional language, aspect oriented programming.

1 Introduction

Purely functional programs [4] support equational reasoning – the program synthesis and the proof of program correctness [4, 16, 17]. However, the complex systems are not functional [20]. They are executed using I/O, exceptions, interrupt handling, channel communication, etc. depending on the application to which they are proposed. The stateful computation/algorithm is such in which the state is significant. Since the state plays significant role in systems, an imperative language seems to be the best alternative for describing their functionality. On the other hand, using functional programming paradigm, a program is more tightly bound to the use of mathematical methods and more appropriate to transformations needed when weaving multiple aspects in aspect languages [2,3,7,15,23]. Let us introduce three approaches used in functional languages, able to express stateful computation. Using a functional language, the mechanism for updating a set of memory cells is required.

In Standard ML [5], the variable environment is used. For example, v is a memory cell in SML definition **val** v = **ref** 5. The value of variable v is accessed using operation ! : a **ref** $\rightarrow a$ in the form !v. The assignment $v := !v + 1$ increments the value of cell v by one. In SML, assignments are expressions of unit type and they may be used elsewhere in expressions by a programmer, i.e. explicitly as it is in an imperative language.

A pure, lazy functional language Clean [1] uses linear types again to perform the stateful computation, like Haskell. The asterisk in a type **World* designates that the type World is linear type. Since each function (process) may be of the type **World* $\rightarrow$ **World*, no single abstraction of monad is necessary to perform stateful computation.

No assignments are available to a programmer in Glasgow Haskell [19] – a purely functional language. They are hidden in application of processes, called state transformers [20]. The single abstraction of monads – mutable abstract type [22] is used to update the values of linear types, i.e. such that are accessed by a single pointer.

A monad is a triple (*M, unitM, bindM*), where M is a linear type, and the operations *unitM* and *bindM* are of the type:

$$unitM :: a \rightarrow M\ a \qquad \text{and} \qquad bindM :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$$

In contrast to SML, Haskell is not environment-based language. The disadvantage of monadic approach is (at least from the viewpoint of software engineering) that memory cells are invisible to a user.

Seemingly, we may decide either to hide variable environment not using assignments, or, making environment visible, we must use assignments explicitly. However, process functional paradigm is based on preventing assignments, at the same time making variable environment visible to a programmer. PFL – an experimental process functional language [8, 9, 10, 11, 12, 13, 14], which we have developed in the past, is closely related to process functional paradigm.

Syntactically, PFL is a reduced subset of Haskell language, extended in a uniform way to support object programming. Currently we have two generators from PFL -

a generator to Java and to Haskell languages. Originally, our aim was to develop a programming language, that is safe, in the sense that it manipulates the state by application of processes instead of assignments and, at the same time, it provides well-defined spatial information about memory data organization to a programmer.

Considering aspect oriented programming methodology [2,3,7,15,23], it seems that PFL may used as a general implementation bridge for any target imperative language, supporting this multi-aspect approach to programming. In this paper, we present the relation between imperative structures of a simple but representative imperative language and process functional expressions. Providing the translation scheme ***P***, and a simple imperative program expressing it in PFL, we prove informally the equivalence of process functional and imperative languages.

The essence of PFL is introduced in section 1. A representative subset of an imperative language (omitting procedures and functions) is defined in section 2. In section 3 we present the translation scheme ***P*** which maps imperative programs to PFL form. An example of the simple imperative program, as well as its equivalent form in PFL, is introduced in section 4.

Finally, in Conclusion, we summarize our results and briefly comment the directions of further research. We will use Bird‘s mathematical notation for PFL programs in this paper.

2 The Essence of PFL

In PFL, the assignments are performed by a process application implicitly [8]. The source form of a process definition (an equation designated by =) is seemingly purely functional. On the other hand, a variable environment is visible to a programmer in a process type definition (an equation designated by ::). Environment variables – memory cells – may be shared by two or more processes defined in the same scope. They are introduced as the attributes of the types of process arguments. There are no reference types in PFL. Each argument type (v T) introduces the variable v of the type T to the variable environment. Each function comprising at least one argument type in the form (v T) is a process. Except that, processes are functions defined in terms of unit types for arguments and/or values. Primitive and algebraic types are designated by identifiers starting with uppercase letters and the environment variables (memory cells) by identifiers starting with lowercase letters. It is just one exception from this rule, when the type T is a type variable. Then it is designated by single lowercase letter. For example (a a) means, that the first a is the environment variable and the second a is the type variable. The computational model for PFL based on control-driven data flow can be found in [9].

As an introductory example, let process *p* is defined as follows:

$p :: a\ Int \to b\ Int \to Int$

$p\ x\ y = x + y$

Then, the value of expression (*p* 2 3) is 5. Like the side effect of evaluation, the values of cells *a* and *b* (environment variables) will be *a* = 2 and *b* = 3.

Now, suppose that the values of *a*, *b* are *a* = 2 and *b* = 3.

Hence, the value of (*p* ()()) is 5 again, and the state of *a* and *b* (*a* = 2 and *b* = 3) remains unchanged. The arguments () are control (unit) values of unit types ().

Provided that an argument is of process type (*v T*), then applying the process on a control value (or an expression that yields control value), the process value is evaluated using the current value having been stored in *v* before the process is applied. Control values do not affect the function of computation directly, nevertheless, the order, in which the expressions of unit types are evaluated, affects the state. Notice, that the processes (in contrast to functions) are evaluated eagerly, following the principle of causality: *The value of a process is evaluated after the arguments are evaluated*. The order, in which the arguments are evaluated, may affect the function of computation. That is why source process definitions are purely functional just seemingly. As an example, let us define the process *q* as follows:

$q :: a\ Int \to a\ Int \to Int$

$q\ x\ y = x + y$

Evaluating (*q* 2 3), the result is 5, but the value assigned to variable *a* is either 2 or 3, depending on whether *q* is applied first to 3 or to 2. If the arguments (that may be complex expressions not just simple constants, such 2 and 3 above) are evaluated in parallel, the state change is non-deterministic. Clearly, the definition of process *q* is purely functional just if we omit its type definition (marked by ::), otherwise not. On the other hand, possible nontransparency is evidently separated from the definition itself and it is shifted to the type definition.

Let process *r* is defined as follows:

$r :: a\ Int \to ()$

$r\ x = ()$

The application (*r* (4 + 5)) evaluates the argument 9, which is assigned to *a*. The result is a control value () which does not allocate the stack at all. It may be noticed that PFL expression application (*r* (4 + 5)) corresponds to the assignment *a* := 4 + 5 in an imperative language.

In the last introductory example, let process *s* is defined as follows:

$s :: () \rightarrow ()$

$s\ () = ()$

Process *s* can be applied to each expression of unit type, yielding control value, for example, such as (*s* (*r* (4+5))). Side effect is the same as for (*r* (4+5)), which yields control value (). This value is used as the argument of *s*. In PFL, the conception of data and control values is well balanced, not hiding them to a programmer. We will suppose evaluating PFL expressions in leftmost innermost order, to guarantee correct semantics for imperative control structures – statements of a sequential imperative language.

3 An Imperative Language

We will start with the syntactic domain of a simple imperative language, in which the program *pr* comprises variable declarations *vd* and block *bl* consisting of statement sequence, according to the Fig.1. A statement *st* may be assignment *as*, if statement *if* , or while statement *wh*.

$$
\begin{array}{lll}
pr & ::= & vd\ bl \\
vd & ::= & \textbf{var}\ v_1 : T_1 ; \ldots v_n : T_n; \\
bl & ::= & \textbf{begin}\ st_1 ; \ldots ; st_m\ \textbf{end} \\
st & ::= & as \mid if \mid wh \\
as & ::= & v := e \\
if & ::= & \textbf{if}\ e\ \textbf{then}\ bl_T\ \textbf{else}\ bl_F \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ bl_T \\
wh & ::= & \textbf{while}\ e\ \textbf{do}\ bl
\end{array}
$$

Fig. 1: An imperative language

The detailed syntax of expression *e* is not substantial for our purposes. In Fig.1, T_k are types, *n* is the number of variables ($n \geq 1$), *m* is the number of statements ($m \geq 1$) of a block, and *e* is an expression. Instead of extended BNF form ($st\ (; st)^*$) we rather express non-empty statement sequence by $st_1 ; \ldots ; st_m$ ($m \geq 1$). Block bl_T of if statement is executed when the value of boolean expression *e* is true, and block bl_F is executed when *e* is false.

4 Expressing Imperative Programs in PFL

In this section we will present the scheme ***P*** for translation of imperative language in Fig. 1. For the purpose of simplicity, we will consider just variables declared in variable declarations *vd* are used in block *bl.*

Program *pr* is equivalent to PFL expression $\boldsymbol{P}\ [\![\ vd\ bl\ ]\!]$, such that:

$$\boldsymbol{P} [\![\; vd \; bl \;]\!] = \boldsymbol{P} [\![\; bl \;]\!] \; \mathbf{A}$$

where $\mathbf{A} = \{ v_1 : T_1 ; \ldots v_n : T_n \}$ is the set of associations defined in variable declarations *vd*

such that v_k are used in *bl*.

For example, the set of associations for imperative program in Fig. 2 is as follows:

$$\mathbf{A} = \{x : Int, y : Int, s : Int\}$$

Expression *e* in an imperative language is equivalent to PFL expression $\boldsymbol{P} [\![\; e \;]\!]$ $\mathbf{A}_e$, as follows:

$$\boldsymbol{P} [\![\; e \;]\!] \; \mathbf{A}_e = ep \; ()_1 \ldots ()_r$$

provided that $v_1, \ldots, v_r$ are (imperative) variables used in *e*, $\mathbf{A}_e = \{v_1 : T_1, \ldots, v_r : T_r\}$, such that $\mathbf{A}_e \subset \mathbf{A}$, and process *ep* is a new process defined as follows:

$$ep :: v_1 \; T_1 \rightarrow \ldots \rightarrow v_r \; T_r \rightarrow T$$

$$ep \; x_1 \ldots x_r = e[x_1/v_1, \ldots, x_r/v_r]$$

where *T* is a type of expression *e*, and its form $e[x_1/v_1, \ldots, x_r/v_r]$ means that in this expression are variables v_k substituted by lambda variables x_k.

For example, let us consider the expression $s + x$ on the right hand side of assignment

$s := s + x$. This expression is translated using associations $\mathbf{A}_e = \{x : Int, s : Int\}$ into the application of a new PFL process, say *sxp*, in the form *sxp* () (), where that *sxp* is defined as follows:

$$sxp :: s \; Int \rightarrow x \; Int \rightarrow Int$$

$$sxp \; p \; q = p + q$$

Block *bl* is equivalent to PFL expression $\boldsymbol{P} [\![\; bl \;]\!] \; \mathbf{A}$, as follows:

$$\boldsymbol{P} [\![\; \mathbf{begin} \; st_1 ; \ldots ; st_m \; \mathbf{end} \;]\!] \; \mathbf{A} = blp \; \boldsymbol{P} [\![\; st_1 \;]\!] \; \mathbf{A}_1 \ldots \boldsymbol{P} [\![\; st_m \;]\!] \; \mathbf{A}_m$$

where the variables of associations $\mathbf{A}_i$ are used in statements st_i. It holds $\mathbf{A} = \mathbf{A}_1 \cup \ldots \cup \mathbf{A}_m$. The new process *blp* applied in PFL expression above, is defined as follows:

$$blp :: ()_1 \rightarrow \ldots \rightarrow ()_m \rightarrow ()$$

$$blp \; ()_1 \ldots ()_m = ()$$

For example, if top-level block contains three assignments, one while statement and one write statement, then corresponding PFL definition of process *blp* is as follows:

$blp :: () \rightarrow () \rightarrow () \rightarrow () \rightarrow () \rightarrow ()$

$blp\ ()\ ()\ ()\ ()\ () = ()$

Statement *st* is equivalent to PFL expression $\boldsymbol{P} [\![\ st\ ]\!]\ \mathbf{A}$, which is equal to PFL expressions for assignment, if statement, or while statement.

Assignment *as* is equivalent to PFL expression $\boldsymbol{P} [\![\ as\ ]\!]\ \mathbf{A}$ as follows:

$$\boldsymbol{P} [\![\ v := e\ ]\!]\ \mathbf{A} = asp\ \boldsymbol{P} [\![\ e\ ]\!]\ \mathbf{A}_e$$

where the variables of associations $\mathbf{A}_e$ are used in expression e. It holds $\mathbf{A} = \{v : T\} \cup \mathbf{A}_e$. The new process *asp* is defined as follows:

$asp :: v\ T \rightarrow ()$

$asp\ x = ()$

and $\boldsymbol{P} [\![\ e\ ]\!]\ \mathbf{A}_e$ is a PFL expression.

If statement *if* is equivalent to PFL expression $\boldsymbol{P} [\![\ if\ ]\!]\ \mathbf{A}$, as follows:

$$\boldsymbol{P} [\![\ \mathbf{if}\ e\ \mathbf{then}\ bl_T\ \mathbf{else}\ bl_F\ ]\!]\ \mathbf{A} = ifp$$

The new process *ifp* is defined as follows:

$ifp :: ()$

$ifp \quad |\ \boldsymbol{P} [\![\ e\ ]\!]\ \mathbf{A}_e \quad = \boldsymbol{P} [\![\ bl_T\ ]\!]\ \mathbf{A}_T$

$\qquad\ \ |\ \mathbf{otherwise} \quad = \boldsymbol{P} [\![\ bl_F\ ]\!]\ \mathbf{A}_F$

where $\mathbf{A} = \mathbf{A}_e \cup \mathbf{A}_T \cup \mathbf{A}_F$.

If statement without bl_F block is expressed as follows:

$$\boldsymbol{P} [\![\ \mathbf{if}\ e\ \mathbf{then}\ bl_T\ ]\!]\ \mathbf{A} = ifp$$

In this case the new process *ifp* is defined as follows:

$ifp :: ()$

$ifp \quad |\ \boldsymbol{P} [\![\ e\ ]\!]\ \mathbf{A}_e \quad = \boldsymbol{P} [\![\ bl_T\ ]\!]\ \mathbf{A}_T$

$\qquad\ \ |\ \mathbf{otherwise} \quad = ()$

where $\mathbf{A} = \mathbf{A}_e \cup \mathbf{A}_T$.

While statement *wh* is equivalent to PFL expression $\boldsymbol{P} [\![\ wh\ ]\!]\ \mathbf{A}$, as follows:

$$\boldsymbol{P} [\![\ \mathbf{while}\ e\ \mathbf{do}\ bl\ ]\!]\ \mathbf{A}_e = whp\ ()$$

New process *whp* is defined as follows:

$$whp :: () \rightarrow ()$$

$$whp\ () \quad |\ \boldsymbol{P}\ [\![\ e\ ]\!]\ \mathbf{A}_e \quad = whp\ \boldsymbol{P}\ [\![\ bl\ ]\!]\ \mathbf{A}_w$$

$$|\ \textbf{otherwise} \quad = ()$$

where $\mathbf{A} = \mathbf{A}_e \cup \mathbf{A}_w$.

PFL form for boolean expression *e* in if and while statements is the same as for general expresions *e* being shown above.

5 An Example

As an example, let us consider a simple imperative program in Fig. 2, which reads two integers x and y, and computes the sum *s* of absolute values in the range

(x . . . y).

It is supposed, that primitive function **read** (referentially non-transparent) and primitive **write**, of the types **read** :: *Int*, and **write** :: *Int* → (), are built-in.

```
var x, y, s : integer;
begin
  x := read; y := read; s := 0;
  while x <= y do begin
    if x > 0 then begin
      s := s + x
    end else begin
      s := s – x
    end;
    x := x + 1
  end;
  write(s)
end
```

Fig. 2: Imperative program

As we will see, except that the basic association is $\mathbf{A} = \{x : Int, y : Int, s : Int\}$, after expressing the program in PFL form, each process uses a subset of this association, which is defined by its type definition.

Since the values of variables *s*, *x* and *y* are used in expressions, the accessing processes *s*, *x* and *y* (for the purpose of simplicity we use for them the same names as for variables) are identities, defined in Fig. 3a.

```
s :: s Int → Int
s p = p
```

$$x :: x\ Int \rightarrow Int$$
$$x\ p = p$$

$$y :: y\ Int \rightarrow Int$$
$$y\ p = p$$

Fig. 3a: Accessing processes

As we will see below, the weakness of imperative programs is that the argument of accessing processes is just unit value (), not a complex expression evaluated to unit value. The values in environment variables *s*, *x*, and *y* are accessed using applications *s*(), *x*(), and *y*(), respectively.

The top level block of a program in Fig. 2 consists of the sequence of three assignments, followed by two statements (while and write). So, we can integrate the transformation of block, sequence and assignment combining all of them by the definition of single process *bsa* – block-sequence-assignment compound process. Since different compound process expresses the body of while statement they are designated by different names: *bsaA* for the top level block, and *bsaB* for the while block in Fig. 3b. On the other hand, both blocks in if statement consist of assignment to the same variable *s*. Such blocks can be integrated using one process *bsaC* for both blocks, as shown in Fig. 3b.

$$bsaA :: x\ Int \rightarrow y\ Int \rightarrow s\ Int \rightarrow () \rightarrow () \rightarrow ()$$
$$bsaA\ \ p\ q\ r\ ()() = ()$$

$$bsaB :: () \rightarrow x\ Int \rightarrow ()$$
$$bsaB\ \ ()\ p = ()$$

$$bsaC :: s\ Int \rightarrow ()$$
$$bsaC\ \ p\ \ = ()$$

Fig. 3b: Compound processes

Now we are ready to built up the structure of our program in Fig. 2 in the whole, defining *while* process using the scheme for while statement, *if* process using the scheme for if statement and finally, main as a constant expression – the application of *bsaA*, yielding unit value, as can be seen in Fig. 3c.

```
while :: () → ()
while () | x()<= y()      = while ( bsaB  if  (x() + 1))
                  | otherwise      = ()

if :: ()
if        | x() > 0  = bsaC (s() + x())
          | otherwise       = bsaC (s() – x())

main  :: ()
main = bsaA read read 0 (whp ()) (write (s()) )
```

Fig. 3c: The structure of the program

The PFL program in Figures 3a, 3b, and 3c including built-in **read** and **write**, is semantically equivalent to the imperative program in Fig. 2; the evaluation of *main* is the same as the execution of program in Fig. 2.

Conclusions

Using PFL, a program is expressed without assignments and statement sequences, preserving at the same time the visibility of environment variables – memory cells. An imperative computation is performed by the evaluation of an expression with side effects.

We have illustrated structured imperative style of programming, building PFL program in botton-up manner. It is not to argue that process functional programming is better than when an imperative language is used. But we can see the following facts. First, exploiting the application dependency in imperative languages is poor, if any. Omitting functions, lambda variables (designated by *p*, *q*, and *r* in our example in Fig. 3a, 3b, and 3c) are not used in expressions at all. Second, the use of variables in imperative languages is far less disciplined, as when they are associated with PFL processes in their type definitions and shared by process applications.

The arguments of processes in this paper are supposed to be evaluated sequentially and eagerly. Essentially, this is the simpliest way how to guarrantee the required degree of determinism in computation. On the other hand, the application dependency is other alternative, exploited using monads [22]. Although this is over the scope of this paper, monadic programming style is not excluded using process functional language. Using monadic style in PFL, in contrast to Haskell, memory cells remain still visible, as we have shown in [12].

Considering multi-paradigmatic approaches, such as combining logic and functional programming [18, 21] or object oriented and logic programming in aspect programming methodology, the aim of both is to increase the semantical power of the language, using less or more uniform language syntax. Especially in aspect programming, the underlying language such as Java in AspectJ [7] determines the transparency of programs, because Java constructs are used in pointcut designators. Except that, renaming a metod in an original module after adding aspect module may affect the semantics the program inappropriately.

As has been shown, PFL form of programs allows (and requires) to designate the structures and substructures of a program systematically. Then we may think about more fine grained aspects, as those when an imperative language is used. In the past, we have PFL-to-Java and PFL-to-Haskell generators developed. The subject of our current research is integrating aspect and process functional paradigm of programming. In this framework, with respect of complexity of software systems and the need to precede their behavior [6], we are interested especially in methods for replacing the „programming style“ (in which PFL implementation structures are named) by the „specification style“ in which they are derived [16, 17] and affected by new static and dynamic aspects, in the way that run-time can be still monitored, corrected, profiled and optimized according to user requirements.

Acknowledgement

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

References

[1] P. Achten, R. Plasmeijer: Interactive Functional Objects in Clean. In: Clack et al. (Ed.): IFL'97, LNCS 1467, 1998, pp. 304–321.

[2] J. H. Andrews: Process-algebraic foundations of aspect-oriented programming, In Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), LNCS, Springer-Verlag, 2001, Vol. 2192, pp. 187–209.

[3] E. Avdicausevic, M. Lenic, M. Mernik, and V. Zumer: AspectCOOL: An experiment in design and implementation of aspect-oriented language. ACM SIGPLAN not., December 2001, Vol. 36, No.12, pp. 84–94.

[4] R. S. Bird: Algebraic Identities for Program Calculation, The Computer Journal, Vol.32, No.2, 1989, pp. 122–126

[5] R. Harper, D. MacQueen, and R. Milner: Standard ML. ECS-LFCS-86-2, LFCS Report Series, University of Edinburgh, Department of Computer Science, 1986, 35pp.

[6] Š. Hudák, S. Šimoňák: FDT Interfacing, Analele Universitatei din Oradea, Romania, pp. 53-59.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold: An overview of AspectJ, In Proceedings European Conference on Object-Oriented Programming, LNCS, Springer-Verlag, Vol. 2072, 2001, pp. 327–353

[8] J. Kollár: Process Functional Programming, In: Proc. of Int. Conf. MOSIS' 99 Conference, Rožnov pod Radhoštěm, Czech Republic, April 27–29, 1999, pp. 41–48.

[9] J. Kollár: Control-driven Data Flow, Journal of Electrical Engineering, Vol. 51. No. 3-4, 2000, pp. 67–74

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

[11] J. Kollár: Object Modelling using Process Functional Paradigm, Proc. 34th Spring International Conference MOSIS 2000 - ISM 2000 Information Systems Modelling , Rožnov pod Radhoštěm, Czech Republic, May 2–4, 2000, ACTA MOSIS No. 80, pp. 203–208

[12] J. Kollár: Partial Monadic Approach in Process Functional Language, Acta Electrotechnica et Informatica, No. 1, Vol. 3, 2003, TU Košice, Slovakia, pp. 36–42

[13] J. Kollár: Unified Approach to Environments in a Process Functional Language, Computing and Informatics, Vol 22, 2003, pp. 439-456

[14] J. Kollár, P. Václavík, and J. Porubän: The Classification of Programming Environments, Acta Universitatis Matthiae Belii, 2003, 10, 2003, pp. 51-64

[15] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer: A reusable object-oriented approach to formal specification of programming languages, L'Objet, 1998, Vol. 4, No. 3, pp. 273-306

[16] V. Novitzká: Structures of Algebraic Specification Languages, Proc. of the Int. Conf. ECI'98, Košice–Herl'any, Slovakia, October 8–9, pp. 1–6.

[17] V. Novitzká: Systems for Deriving Correct Implementations. In:Proc. of Int. Conf. MOSIS'99 Conference, Rožnov pod Radhoštěm, Czech Republic, April 27–29, 1999, pp. 201–207.

[18] M. Paralič: Mobile Agents Based on Concurrent Constraint Programming, Joint Modular Languages Conference, JMLC 2000, September 15 6-8, 2000, Zurich, Switzerland. In: Lecture Notes in Computer Science, 1897, pp. 62–75.

[19] J. Peterson, K. Hammond (Ed.): Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.3. Yale University, May 1996. 164pp.

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

[21] G. Smolka: 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.

[22] P. Wadler: The essence of functional programming. In: 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, pp. 47-70.

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