From Weaving Schemes to Weaving Patterns[*]

JÁN KOLLÁR

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

Abstract. Coming out from the process functional paradigm and using $\mathcal{PFL}$ – a process functional language, a generalized approach to weaving at the micro-structural level is presented. Exploiting the application of processes and $\mathcal{PFL}$ reflection property, we develop a generalized weaving scheme and we express it in the form of weaving pattern. Different specializations and extensions of weaving patterns occurring in aspect oriented languages are discussed. Weaving patterns expressed in terms of weaving chains provide us with the flexibility inevitable for the aspect oriented evolution of software systems instead of aspect oriented programming. Presented abstraction in the form of patterns comes out from integrating imperative, purely functional and object paradigms in the process functional paradigm and it may contribute to the application of aspect oriented approach to the area of automatic evolution of software systems.

Keywords: Aspect oriented programming, weaving strategies, software architectures, systems evolution, implementation paradigms.

(Received March 06, 2005 / Accepted April 20, 2005)

1 Introduction

The separation-of-concerns principle is one of the essential principles in software engineering. It says that software should be decomposed in such a way that different concerns or aspects of the problem at hand are solved in well-separated modules or parts of the software [3, 20].

Aspect oriented programming [8, 14] offers a new paradigm for software development, which complements conventional programming paradigms with a higher degree of separation of concerns. The development of an aspect oriented application is commonly supported by an aspect language, such as AspectJ [8] to modularize crosscutting concerns as aspects; and the aspect weaver that instruments the component program with aspect programs to produce the final system.

AspectJ defines a set of new language constructs to modularize crosscutting concerns. An aspect module in AspectJ contains pointcuts and the associated advices. A pointcut construct denotes a collection of join points. AspectJ code can be executed before, after or in place of the program execution when a join point is reached. These actions are defined using AspectJ specific constructs *before*, *after*, and *around*. These constructs are called advices [8, 19], since they comprise advised code.

The aspect oriented approach is a software programming methodology, which makes programs more reliable, because mutual interconnections between original program modules and the aspect module are performed automatically, by weaving [8, 14, 21]. The reasons are in the application of specification principles, that are exploited using logical formulae for the selection (picking out) a set o join points, in which advises are applied.

Out motivation for the development of more abstract form of weaving patterns for weaving schemes comes from the following questions that are arising:

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

1. Can the methodology itself ensure non-existence of bugs in programs?

2. Is the set of pointcut designators in current aspect languages complete and/or is it extensible in a flexible manner?

3. Is it appropriate to use the names in pointcut designators, considering that then a programmer still must have a very detailed notion about the original modules, otherwise one mistake in advice module may yield unwanted woven programs?

4. Is the current aspect oriented approach applicable to all levels of granularity of systems, in the uniform way, with the same reliability? Or is it restricted, being just a coarse-grained extension of object approach?

5. Even, is aspect oriented approach applicable just to systems programming in a life-cycle? Or, would it not be possible to separate the specification and the implementation systematically, to provide an opportunity for systems evolution, such that makes them live at any time?

Especially with respect of the last question, we are interested in the uniform weaving mechanism, as a general systems evolution principle.

Since weaving transformations yield semantic changes [19], they must be inspected far more systematically, first from the structural point of view, as it was done up to now. That is why, we are focusing on the structural essence of one category of weaving transformations, and their semantic effects are mentioned just marginally, in discussion.

In the past, we have found that imperative, functional and object paradigms can be integrated in a process functional paradigm [9, 10], in which processes are defined in terms of expressions, as it is done in Haskell [17], but with memory cells visible, being all shifted to type definitions (type signatures) of processes. In this way, higher-order functions, parametric polymorphism and overloading are preserved.

Our approach is based on more abstract and still implementation level of $\mathcal{PFL}$ – a process functional language than provided by current imperative languages.

The application of processes is a single execution mechanism in $\mathcal{PFL}$. Control values are explicitly visible, opposite to imperative languages. The concerns of variable environment and code are well-separated [11, 12]. The source form of all $\mathcal{PFL}$ expressions is purely functional, and environments are associated with type expressions. $\mathcal{PFL}$ reflection means reflecting the properties of a system in the form of values computed during execution at the level of type expressions.

The essence of process functional paradigm is introduced in section 2, in which both outer and inner $\mathcal{PFL}$ reflection property is illustrated, as an assumption for the systematic approach to weaving.

In section 3, we will simply suppose, that a joint point in the form of the application is selected. Then we will be interested, how an advice (which is again in the form the application) can be woven into the original code, developing a weaving scheme.

In particular, we are interested in weaving scheme, which does not affect the function of original code, provided that advice is purely functional. As a result, we answer the question about the possibility of finding the original application after weaving.

Formal remarks and comments to this scheme are introduced in section 4. We also present more abstract form for weaving schemes, considering the structure of applications separately from semantics, in the form of chains. Weaving patterns are weaving schemes defined in terms of weaving chains.

We discuss the flexibility of weaving patterns as the abstraction of weaving schemes in section 5.

Related works are introduced in section 6.

In conclusion, we summarize our results.

In this paper, we omit the question, how to find a collection of join points. We are concentrated to weaving and its generalization, as the proposition for a systematic approach to the evolution of software architectures based on the specification of goals in terms of types and values, rather than names, introduced by a programmer. The names of functions, processes, lambda variables, etc., in this paper are introduced just for the purpose of explanation.

We use mathematical notation for $\mathcal{PFL}$ programs, since it is more appropriate for our purposes. For example, instead of concrete types, such as `Int` or `Float` we use types in general form, such as T_1, S_2, etc. All $\mathcal{PFL}$ function and/or process definitions, introduced in this paper are not numbered.

2 Reflection in $\mathcal{PFL}$

$\mathcal{PFL}$ reflection property enables to access and update the values in environment variables outside the definition of a function, whenever the function is applied to its arguments. Such functions are called processes – hence the name *process functional* paradigm. A $\mathcal{PFL}$ (pure) function is defined in terms of the type definition (see the equation comprising ::) and the definition (the equation comprising $=$). For example, function f

of two arguments, which sums them, is defined as follows:

$$f :: T_1 \to T_2 \to T$$
$$f\ x\ y = x + y$$

The definition of function above is illustrated in Fig.1.

Provided that environment variables occur in the type definition as the attributes of the argument types, such function is called the process.

Since of two arguments, function f may be rewritten or transformed easily, obtaining four possible kinds of processes, see Fig.1 a), b), c), and d), corresponding to the following cases of definitions.

Case a):

$$f :: u\,T_1 \to T_2 \to T$$
$$f\ x\ y = x + y$$

Case b):

$$f :: T_1 \to v\,T_2 \to T$$
$$f\ x\ y = x + y$$

Case c):

$$f :: u\,T_1 \to v\,T_2 \to T$$
$$f\ x\ y = x + y$$

Case d):

$$f :: u\,T_1 \to u\,T_2 \to T$$
$$f\ x\ y = x + y$$

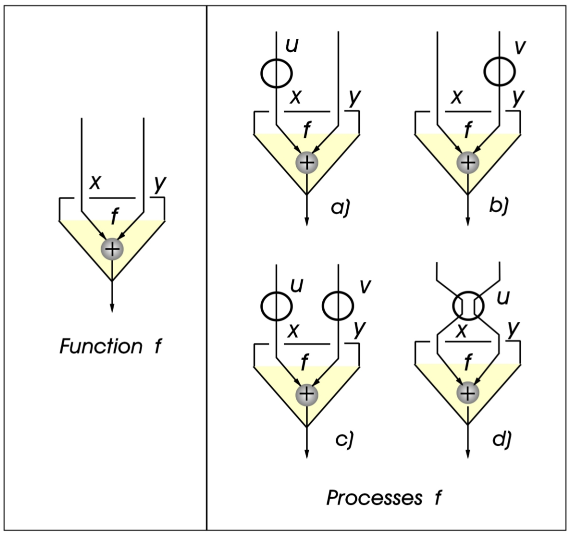

Figure 1: Function and processes

Binding the environment variables u and v in processes is transparent, since it is static – exclusively via type definitions. Notice, process body $x + y$ remains unchanged.

The same environment variable (such as u in case d) may be shared by multiple arguments of a process. Environment variables may be even shared by different processes; but this case is not illustrated in this section.

2.1 Outer Reflection

Outer reflection property enables to update the environment cells by data values of arguments, and to access them by the unit value arguments. Both effects are reached by the applications of processes.

Let us consider case b) of process f in Fig.1.

The application $(f\ 3\ 2)$ is evaluated by subsequent parameter passing, and the evaluation of f body, according to Fig.2.

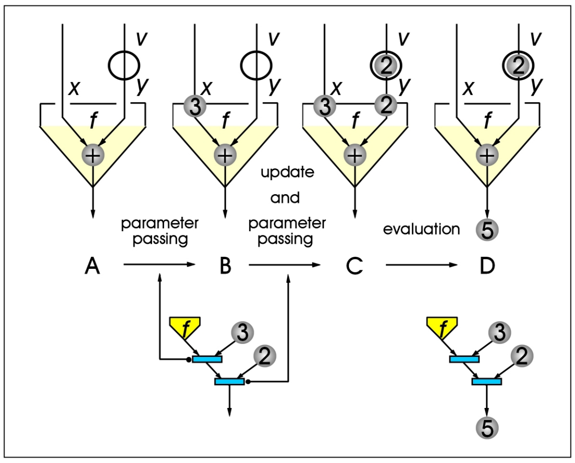

Figure 2: Application to data values - example of the update

Starting with the undefined value in environment variable v, the crucial is the second parameter value passing, since of assigning (or reflecting) this value to v by the side effect. The value of the application is 5, and it will be the same, even for pure function (case a), or processes in cases b) or d). However, the reflected value in case b) would be $u = 3$ and in case d) $u = 2$.

Provided that the value of v is 2, for example as a side effect of application $(f\ 3\ 2)$, the other application, see Fig.3, may access the value stored in v, using the unit value as the second argument.

The unit value and its (unit) type, are designated in $\mathcal{PFL}$ by $()$, as in Haskell. In Fig.3, a small ball marks it, while big balls mark data values.

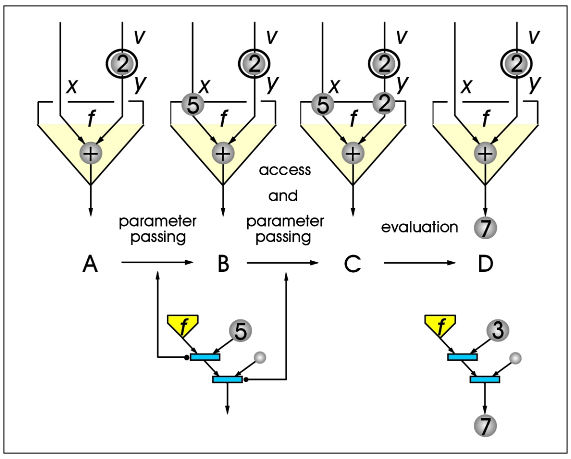

Figure 3: Application to data and control values - example of the access

The application operation $@$, represented by the space in a program, is left associative operation, i.e. it holds $(f\ 5\ ()) = ((f@5)@())$. As can be seen, the operation $@$ is marked by boxes in all figures.

For the purpose of explanation, we have used just constant arguments of the process. Of course, the arguments of processes may be any complex expressions in general.

It would be possible to designate the environment variables u and v by the same names as lambda variables, i.e. by x and y, because of different positions of lambda variables and environment variables in a memory. In process functional paradigm, lambda variables are not just holes, that represents values used in function (lambda abstraction) body, as it is in lambda calculus, but they designate stack memory cells, containing (as a result of application) actual parameter values, similarly as it is in imperative languages.

Environment variables may reside on the stack, in global memory, or in object record, corresponding to imperative and object paradigm.

On the other hand, the values of processes are defined by expressions, in terms of lambda variables, not using environment variables, corresponding to purely functional paradigm.

2.2 Inner Reflection

Suppose a function/process h and its local process g.

The inner reflection enables to access and to update one or more environment variables of process g, that are matched with lambda variables of a function/process h.

Let $\mathcal{PFL}$ function h with a local process g be defined as follows:

$$\begin{aligned}
&h :: T_1 \to T_2 \to T\\
&h\ \boxed{x}\ y = g\ x\ y * g\ ()\ 4\\
&\qquad \textbf{where}\\
&\qquad\quad g :: \boxed{x}\ T_1 \to z\ T_2 \to T\\
&\qquad\quad g\ x\ y = x + y
\end{aligned}$$

Since of the same name x (in the box) used for the environment variable of local process g and for lambda variable of pure function h, the same stack cell (allocated for the first parameter of h) is used for both of them.

The environment variable z is the environment variable of g, local to h. The definition of function h is shown in Fig.4.

The result of application $(h\ 3\ 2)$ is 35. The substantial is the fact, that after passing 3 by $(h\ 3)$ to lambda variable x of h, this value is accessible as the value of environment variable x of g in application $(g\ ()\ 4)$.

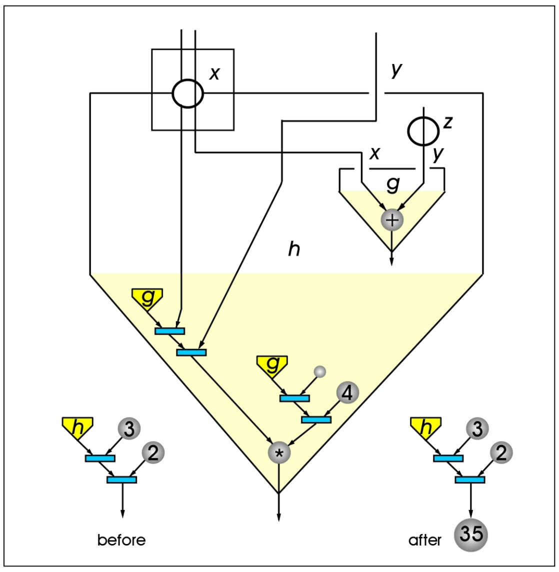

Figure 4: Inner reflection as a result of sharing $\boxed{x}$ variable

We introduce local environment variable z to show that it is updated two times. Because multiplication $(*)$ is left associative operation, z is updated by the value 2, and after that by the value 4.

Now, let us discuss the effect of different modifications of the body of function h. Replacing the expression $g\ x\ y * g\ ()\ 4$ by expression $g\ x\ y * g\ ()\ ()$, the result of $(h\ 3\ 2)$ is 25.

Provided that the body is $g\ ()\ y * g\ ()\ ()$, the result is again 25.

The result of $g\ y\ x{*}g\ ()\ ()$ body is 16. After passing the value 2 by $(g\ y)$, this value is stored in environment variable x of g, i.e. it replaces the argument value x of h. It means that $(g\ y\ x)$ evaluates to $(g\ 2\ \mathbf{2})$, not to $(g\ 2\ \mathbf{3})$, as one might think. This case illustrates the ability for substitution of the value of argument x of h by the value of argument y. Mathematically, computing $(h\ 2\ 2)$ instead of $(h\ 3\ 2)$ is a nonsense. Computationally, in aspect programming, changing $(h\ 3\ 2)$ to $(h\ 2\ 2)$ may be useful.

The variations on different bodies above illustrate the flexibility and simplicity of transformations based on process functional paradigm in $\mathcal{PFL}$, yielding different semantic effects.

Exploiting process functional paradigm in $\mathcal{PFL}$, full imperative semantics is reached considering just applications and application dependencies, what clearly yields the significant simplification of source-to-source transformations, as required for weaving.

3 Weaving Requirements

In this section we define our particular task and the result, which will be obtained by the weaving of original and advice – both in the form of applications.

We suppose that a join point in the form of original application has been picked out in $\mathcal{PFL}$ function p, which is defined as follows:

$$\begin{array}{l} p :: T_1 \rightarrow \ldots \rightarrow T_p \rightarrow T \\ p\ x_1\ \ldots x_p = \sigma\ (e_1\ e_2\ \ldots\ e_n)\ \omega \end{array}$$

It means, that p is defined by expression

$$\sigma\ (e_1\ e_2\ \ldots\ e_n)\ \omega$$

of the type T, which comprises the original application

$$(e_1\ e_2\ \ldots\ e_n)$$

i.e. join point. Prefix and postfix parts σ and ω are out of our interest, since they do not contain the application $(e_1\ e_2\ \ldots\ e_n)$.

For the purposes of more transparent description, we will use the abbreviated form for argument variables and types below, and we designate $T_1 \rightarrow \ldots \rightarrow T_p$ by $\mathbf{T}^p$, and $x_1\ \ldots x_p$ by $\mathbf{X}^p$. Using this shortcuts, the abbreviated form of p definition above is as follows:

$$\begin{array}{l} p :: \mathbf{T}^p \rightarrow T \\ p\ \mathbf{X}^p = \sigma\ (e_1\ e_2\ \ldots\ e_n)\ \omega \end{array}$$

Designating the types of expressions used in the original application, the application rule is defined by (1).

$$\frac{\begin{array}{c} e_1 :: Q_1 \rightarrow Q_2 \rightarrow \ldots \rightarrow Q_{n-1} \rightarrow Q_n \\ e_2 :: Q_1 \quad e_3 :: Q_1\ \ldots\ e_n :: Q_{n-1} \end{array}}{e_1\ e_2\ \ldots\ e_n :: Q_n} \tag{1}$$

Considering the application order given by currying – a mechanism that guarantees a subsequent application of arguments – it holds

$$e_1\ e_2\ \ldots\ e_n = (\ldots((e_1)\ e_2)\ \ldots)\ e_n \tag{2}$$

Supposing eager evaluation of expressions, the application order given by (2) yields the precedence (time order) for expressions in the original application, as follows:

$$e_1 \prec e_2 \prec \ldots \prec e_n \tag{3}$$

Now, we will define the advice, again in the form of an application.

This advised application evaluates corresponding to the application rule (4), with evaluation order (5).

$$\frac{\begin{array}{c} a_0 :: S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow S_n \\ a_1 :: S_0 \quad a_2 :: S_1\ \ldots\ a_n :: S_{n-1} \end{array}}{a_0\ a_1\ \ldots\ a_n :: S_n} \tag{4}$$

$$a_0\ a_1\ a_2\ \ldots\ a_n = (\ldots(((a_0)\ a_1)\ a_2)\ \ldots)\ a_n \tag{5}$$

The application order (5) yields the precedence of advice expressions, as follows:

$$a_0 \prec a_1 \prec a_2 \prec \ldots \prec a_n \tag{6}$$

Since the advised application may contain free variables, which we want to be bound by parameters of original function p in woven form, it is reasonable to express the advice in the form of the definition of function, as follows:

$$\begin{array}{l} advice :: R_0 \rightarrow \ldots \rightarrow R_m \rightarrow S_n \\ advice\ x_1\ \ldots x_m = a_0\ a_1\ \ldots\ a_n \end{array}$$

in which all, originally free variables can be seen now as lambda variables $x_1, \ldots, x_m$.

The abbreviated form for $advice$ is as follows:

$$\begin{array}{l} advice :: \mathbf{R}^m \rightarrow S_n \\ advice\ \mathbf{X}^m = a_0\ a_1\ \ldots\ a_n \end{array}$$

The original process p and the function $advice$ are illustrated in Fig.5.

The task of weaving is as follows:

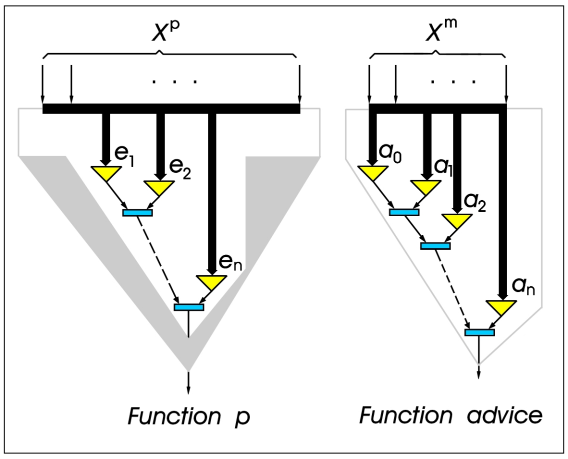

Figure 5: Original function p and advice defined by function $advice$

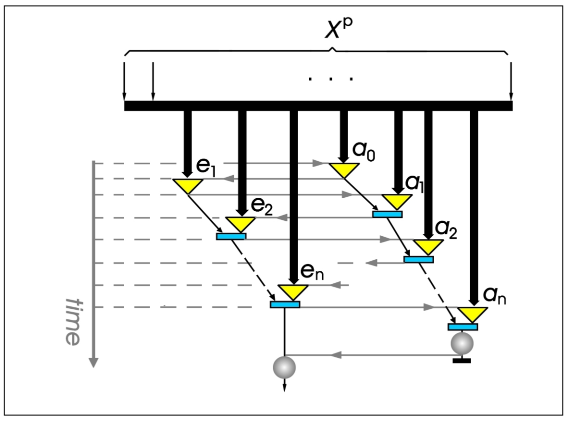

Figure 6: Requirements to woven form

1. Advised code defined by $advice$ must be woven into function p, using the values of arguments of p of the same type.

2. By weaving, it is necessary obtain evaluation precedence, as follows:

$$a_0 \prec e_1 \prec a_1 \prec e_2 \prec a_2 \prec \ldots \prec e_n \prec a_n$$

3. The value of function p must not be affected by weaving. As we will see later, since of flexibility of the developed scheme, it is easy to substitute this requirement by other.

The execution of original application and advised application in time, as well as sharing parameter values of p by advised application are illustrated in Fig.6. The irrelevant σ and ω parts of p body are omitted.

Instead of concrete input arcs that represent the flow of values, the "buses of arcs" are used, since of general form of weaving scheme.

4 Weaving Scheme and Weaving Pattern

The woven form of function p is the value of weaving scheme $\mathcal{W}$, which is introduced in Fig.7.

In woven form, both original and advised applications are step-by-step synchronized.

In Fig.7, the shortcut $()^{s_k}$ stands for s_k unit values used as arguments, i.e. for $\underbrace{()\ ()\ \ldots\ ()}_{s_k}$.

Local processes ad_k and local functions ad'_k are generated using original names for variables in expressions e_k and a_k.

The shortcut $(S^{k\rightarrow n})$ stands for type expression $(S_k \rightarrow \ldots S_n)$ and the form $(Q^{k\rightarrow n})$ stands for type expression $(Q_k \rightarrow \ldots Q_n)$. In both cases, the parentheses are relevant, since parameters x_a and x_e are functions.

The shortcut $\mathbf{X}^{s_k}$ is used for s_k lambda variables, identical to free variables used in a_k on the right hand side of ad_k definition. By other words, lambda variables can be generated from the set of free variables used in a_k.

Sharing the subset of parameters $\mathbf{X}^p$ of p is given by attributed types $\mathbf{XT}^{s_k}$ of a local process ad_k, in which the expression a_k is evaluated. The shortcut $\mathbf{XT}^{s_k}$ is used instead of $x_{u_1}\ T_1 \rightarrow \ldots \rightarrow x_{u_{s_k}}\ T_{s_k}$, such that $\{x_{u_1}, \ldots x_{u_{s_k}}\} \subseteq \mathbf{X}^p$.

Except that original and advised application are synchronized, we may conclude, that the function p is not changed, provided that the advice does not affect environment variable which is used by the original application. If this is true (and it may be detected based on the application dependence analysis), then, for example, we are sure, that a potential bug in advice does not infect the original program and vice versa, since both are executed using disjunctive computational spaces. On the other hand, if this is not the case, it is possible to detect statically where and when the original function is affected by the side effect caused by advice, and vice versa.

The variables x_{u_1}, ..., $x_{u_{s_k}}$ from $\mathbf{X}^p$ may be generated based on matching types, by no means by matching parameter names. This, however, is over the scope of this paper. But careful reader may notice, that the accurate, deterministic and non-redundant solution is not so trivial, as might seem at the first sight.

It may be also noticed, that the definition of an advice by the constant application would simplify the weaving scheme significantly, but it is still possible to share

$$\mathcal{W}\left\llbracket \begin{array}{l} p :: \mathbf{T}^p \to T \\ p\ \mathbf{X}^p = \sigma\ (e_1\ e_2\ \ldots\ e_n)\ \omega \end{array} \right\rrbracket\ advice =$$

$$\boxed{\begin{array}{l}
p :: \mathbf{T}^p \to T \\
p\ \mathbf{X}^p = \sigma\ (ad_0\ ()^{s_0})\ \omega \\
\quad \mathbf{where} \\
\qquad ad_0 :: \mathbf{XT}^{s_0} \to Q_n \\
\qquad ad_0\ \mathbf{X}^{s_0} = ad'_0\ a_0 \\
\\
\qquad ad'_0 :: (S^{0\to n}) \to Q_n \\
\qquad ad'_0\ x_a\ = ad_1\ x_a\ e_1\ ()^{s_1} \\
\\
\qquad ad_1 :: (S^{0\to n}) \to (Q^{1\to n}) \to \\
\qquad\qquad \mathbf{XT}^{s_1} \to Q_n \\
\qquad ad_1\ x_a\ x_e\ \mathbf{X}^{s_1} = ad'_1\ (x_a\ a_1)\ x_e \\
\\
\qquad ad'_1 :: (S^{1\to n}) \to (Q^{1\to n}) \to Q_n \\
\qquad ad'_1\ x_a\ x_e = ad_2\ x_a\ (x_e\ e_2)\ ()^{s_2} \\
\\
\qquad ad_2 :: (S^{1\to n}) \to (Q^{2\to n}) \to \\
\qquad\qquad \mathbf{XT}^{s_2} \to Q_n \\
\qquad ad_2\ x_a\ x_e\ \mathbf{X}^{s_2} = ad'_2\ (x_a\ a_2)\ x_e \\
\\
\qquad ad'_2 :: (S^{2\to n}) \to (Q^{2\to n}) \to Q_n \\
\qquad ad'_2\ x_a\ x_e = ad_3\ (x_e\ e_3)\ x_a\ ()^{s_3} \\
\qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\qquad ad_{n-1} :: (S^{(n-2)\to n}) \to (Q^{(n-1)\to n}) \to \\
\qquad\qquad\quad \mathbf{XT}^{s_{n-1}} \to Q_n \\
\qquad ad_{n-1}\ x_a\ x_e\ \mathbf{X}^{s_{n-1}} = ad'_{n-1}\ (x_a\ a_{n-1})\ x_e \\
\\
\qquad ad'_{n-1} :: (S^{(n-1)\to n}) \to (Q^{(n-1)\to n}) \to Q_n \\
\qquad ad'_{n-1}\ x_a\ x_e\ = ad_{n-1}\ x_a\ (x_e\ e_n)\ ()^{s_{n-1}} \\
\\
\qquad ad_n :: (S^{(n-1)\to n}) \to Q_n \to \\
\qquad\qquad \mathbf{XT}^{s_n} \to Q_n \\
\qquad ad_n\ x_a\ x_e\ \mathbf{X}^{s_n} = ad'_n\ (x_a\ a_n)\ x_e \\
\\
\qquad ad'_n :: S_n \to Q_n \to Q_n \\
\qquad ad'_n\ x_a\ x_e\ = x_e
\end{array}}$$

Figure 7: General Weaving Scheme

common space of environment variables and hence to affect the original function.

The required time order is guaranteed by the application order of generated local processes ad_k and local functions ad'_k. However, the variables in both e_k and a_k remain unchanged, after weaving.

Provided that we designate the original application $e_1\ e_2 \ldots e_n$ by the chain $\bullet\bullet\ldots\bullet$ and the advised application $a_0\ a_1\ a_2 \ldots a_n$ by the chain $\circ\circ\circ\ldots\circ$, the weaving pattern, which corresponds to weaving scheme $\mathcal{W}$ in Fig.7, can be expressed as follows.

$$\mathcal{W}\llbracket \bullet\bullet\ldots\bullet \rrbracket\ \circ\circ\circ\ldots\circ = {}^{\circ}{}_{\bullet}{}^{\circ}{}_{\bullet}{}^{\circ}\cdots{}_{\bullet}{}^{\circ} \tag{7}$$

The value of weaving pattern above expresses that two independent applications are evaluated (in horizontal direction) while the precedence of evaluation in time of all expressions is determined in vertical direction.

Considering just the chains in weaving patterns, we may abstract from function of computation, but it is still possible to reason about the types. We may conclude, that types in circle chain are independent from types in bullet chain, but such that application rules (4) and (1) hold.

Let us discuss now the specialization and possible extensions of weaving patterns.

5 Discussion

It is possible to specialize weaving pattern (7), to obtain *before* advice by $\mathcal{W}_1$ and *after* advice by $\mathcal{W}_2$

$$\mathcal{W}_1\llbracket \bullet \rrbracket \circ = {}^{\circ}{}_{\bullet} \qquad \mathcal{W}_2\llbracket \bullet \rrbracket \circ = {}_{\bullet}{}^{\circ} \tag{8}$$

In weaving patterns, (7) and (8), it is supposed the value for original is produced by bullet chain, since of its bottom position in pattern. We may notice the flexibility of the scheme $\mathcal{W}$.

Changing the definition of the last function in $\mathcal{W}$ to $ad'_n\ x_a\ x_e\ = x_a$, and its type to $S_n \to Q_n \to S_n$, the advised application value is used instead of original, which vice-versa will be computed by the side effect. Then the value of function p will be changed, even if the advice is purely functional.

This corresponds to the weaving schemes $\mathcal{W}'$, $\mathcal{W}'_1$, and $\mathcal{W}'_2$, according to (9) and (10).

$$\mathcal{W}'\llbracket \bullet\bullet\ldots\bullet \rrbracket\ \circ\circ\circ\ldots\circ = {}_{\circ}{}^{\bullet}{}_{\circ}{}^{\bullet}{}_{\circ}\cdots{}^{\bullet}{}_{\circ} \tag{9}$$

$$\mathcal{W}'_1\llbracket \bullet \rrbracket \circ = {}_{\circ}{}^{\bullet} \qquad \mathcal{W}'_2\llbracket \bullet \rrbracket \circ = {}^{\bullet}{}_{\circ} \tag{10}$$

In addition, the condition $S_n = Q_n$ must be satisfied, otherwise the advice could not be used instead of the original. It means that $\mathcal{W}'$ is valid, provided that type checking rule (11) holds, and $\mathcal{W}'_1$ and $\mathcal{W}'_2$ are valid, provided that type checking rule (12) hold, as follows.

$$\bullet \bullet \ldots \bullet : T \Rightarrow \circ \circ \circ \ldots \circ : T \tag{11}$$

$$\bullet : T \Rightarrow \circ : T \tag{12}$$

The *instead* advices defined by weaving schemes $\mathcal{W}'$, $\mathcal{W}'_1$, and $\mathcal{W}'_2$ are associated with the evaluation of original expressions by the side-effect. An alternative is the weaving scheme that replaces original application by advised application, according to (13).

$$\mathcal{W}_3[\![\bullet \bullet \ldots \bullet]\!] \ \circ \circ \circ \ldots \circ = \circ \circ \circ \ldots \circ \tag{13}$$

In this way, the chain $\bullet \bullet \ldots \bullet$ is forgotten forever. For such advising, $\bullet \bullet \ldots \bullet : T \Rightarrow \circ \circ \circ \ldots \circ : T$ must hold.

The weaving above is trivial for a constant in the role of advised application, because constants do not exploit lambda variables of original function or process, in which they are woven. Otherwise it is impossible simply to substitute an original represented by bullet chain by the advice represented by circle chain in function/process body. Instead of that, it is necessary to use the similar approach as for the scheme $\mathcal{W}$ defined in Fig.7. However, this weaving belongs to the different category, which is not the subject of this paper.

The same holds for weaving pattern (14), in which $\diamond$ designates the (empty) position in which $\circ \circ \circ$ will be substituted.

$$\mathcal{W}_3[\![\diamond \bullet]\!] \ \circ \circ\circ = \circ \circ \circ \bullet \tag{14}$$

Provided that it holds

$$\bullet = e_1 \quad \text{and} \quad \circ \circ \circ = \textbf{if}\ a_1\ a_2 \tag{15}$$

we obtain

$$\circ \circ \circ \bullet = \textbf{if}\ a_1\ a_2\ e_1 \tag{16}$$

which means, that instead original evaluation of expression e_1, the application of operation **if** to arguments evaluates, yielding either a_2 or e_1, depending on boolean value of expression a_1. In this way, the evaluation may depend on values in external variable environment accessible via processes applied in a_1. In this way, dynamic weaving is enabled, while static weaving defined by (14) is just inevitable preliminary step.

Finally, we discuss the simplest form of *instead* and *around* weaving. The Wand's [19] *instead* advising can be expressed in terms of weaving pattern, defined by (17).

$$\mathcal{I}[\![\bullet]\!] \circ = \circ \tag{17}$$

The simplest and correct *around* weaving is defined by the pattern $\mathcal{A}_1$ or $\mathcal{A}_2$ in (18).

$$\mathcal{A}_1[\![\bullet]\!] \ \circ \circ = {}^{\circ}{}_{\bullet}{}^{\circ} \qquad \mathcal{A}_2[\![\bullet]\!] \ \circ \circ = {}_{\circ}{}^{\bullet}{}_{\circ} \tag{18}$$

On the other hand, the patterns defined by (19) are wrong.

$$\mathcal{A}'_1[\![\bullet]\!] \circ = {}^{\circ}{}_{\bullet}{}^{\circ} \qquad \mathcal{A}'_2[\![\bullet]\!] \circ = {}_{\circ}{}^{\bullet}{}_{\circ} \tag{19}$$

This is so because applying the weaving pattern to single $\circ$, it holds $\circ : T$. But, at the same time, $\circ : T$ is used in pattern twice in application $\circ\circ$, first occurrence being of type $T_1 \rightarrow T_2$, and the second occurrence of the type T_1. Formalizing this, we obtain the condition (20).

$$(\circ : T) \wedge (\circ : T_1 \rightarrow T_2) \wedge (\circ : T_1) \tag{20}$$

But this is a contradiction, since the unification of types fails on equation $T_1 \rightarrow T_2 = T_1$ (or $T \rightarrow T_2 = T$).

6 Related Work

In practice, the principle of separation of concerns is not always that easy to achieve. As it turns out, no matter how well an application is decomposed into modular entities, some functionality always crosscuts this modularization. This phenomenon is known as the tyranny of the dominant decomposition. As a consequence, such crosscutting functionality (often called a concern) cannot be evolved separately, as it affects all other entities in the application [5].

Many security experts feel uneasy about trying to isolate security-related concerns, because security is such a pervasive property of a piece of software. The implementation convolution problem refers to the phenomenon that, for a large number of non-trivial functionalities, although their semantics are distinctive, their implementations do not have clear modular boundaries within the (middleware) code space and, more seriously, often tangle with one another. This prohibits these functionalities from being pluggable [20]. For example, the principles of orthogonal and weakly orthogonal aspects instruct in the design of aspects that are included in some system configurations, but not in others [3].

Aspect mining and static refactoring techniques are proposed in [5], to detect and separate the cross-cutting concerns respectively. In a second step, the well-modularized application should be controlled at the metalevel by a monitor with full reflective capabilities.

On the other hand, to achieve new semantics of woven programs, novel-programming constructs can be found in aspect languages, that are the subject of formal analysis. This analysis is complicated, since the restrictions are given by the complexity of implementation language, such as Java.

Simplifying the structure of language, better results are achieved, and the analysis is more complex and valuable. For example, taking as a basis MiniMAO [2], practically all constructs of AspectJ analyzed, in contrast to previous works.

Strategies that are used to aid in the rapid construction of new domain-specific weavers and an adoption of generative programming approaches with respect to constructing a weaver [7] require non-trivial source-to-source program transformations.

The design of a generic framework to express aspects as syntactic transformations as well as a generic weaver requires the semantic properties for the definition of aspects be used [6]. However, an approach to generic weaving based on repeated program transformations might fail using imperative assignments and statement sequence.

There is strong need for formalizing aspects [18] as well as for manipulating them using more formal languages as implementation languages, see for example μabc [1].

To be able to provide models describing goals and strategies for reaching the properties of software systems, not just models or meta-models for software architectures, such as in [13], we must think about incremental evolution instead of incremental programming [15], and formalize not just design patterns [16] but also implementation patterns.

Inferring the grammar for the language, from fragments of programs written in different languages is possible [4]. Our task is different. We want to determine an abstracted general weaving pattern, and then to use it as a parameter, while software architecture is inferred, i.e. automatically generated from the model, which describes the substantial properties of the system.

7 Conclusions

Presented abstraction in the form of weaving patterns may contribute to the theory of aspects [1, 18], as well as to the application of aspect oriented approach to the automatic evolution of software systems in the future.

Introducing the essence of process functional paradigm including the reflection property, we have developed the scheme for weaving two applications. The functionality of transformed function p, as a result of weaving given by the value of $\mathcal{W}$ in Fig.7 remains unchanged, except that its computational time has increased, as a result of advice computation by the side effect. Although the transformed p is an executable $\mathcal{PFL}$ function with generated names, it may be noticed, that our approach is not oriented to considering the names by a user.

Moreover, the form of original application is transformed to the form such that cannot be supposed to be found as a join point anymore. This follows us to conclude, that repeated weaving decreases the transparency of woven programs, hence there is no benefit from the fact that it is source-to-source transformation.

Of course, it is not critical for a very coarse aspect oriented programming at the level of classes, using a very simple advising patterns in aspect oriented programming languages. Some of them we have analyzed in discussion.

At the same time, expressions e_k (marked by $\bullet$), and a_k (marked by $\circ$) (and of course σ and ω parts) may be the subject of collecting new join points in the form of applications.

To be able to apply weaving as a general composition method for generating software architectures by the specification of goals of evolution, we have introduced the abstracted representation of weaving schemes in the form of weaving patterns, using chains. It could be impossible without a single execution engine – the application of processes, as a result of process functional paradigm.

The main contribution of this paper is the separation of structural and semantic aspects of weaving, providing a new idea of generalized approach to weaving on more abstract level. Our general weaving scheme in Fig.7 and its pattern (7) are still general just with respect of one category of patterns.

How to generate, to mutate and to combine weaving patterns, that are some kind of genetic information affecting the systems generation (or evolution) by aspect manner, is the future, and the systematic study of categories of weaving chains, associated with semantic rules is the subject of our current research.

References

[1] Bruns, G., Jagadeesan, R., Jeffrey, A. and Riely, J. *μabc: A minimal aspect calculus*. In Proceedings of the 2004 International Conference on Concurrency Theory, Springer-Verlag, p.209–224, 2004.

[2] Clifton, C. and Leavens, G. T. *MiniMAO: Investigating the semantics of proceed*. FOAL 2005 Proceedings, Foundations of Aspect-Oriented Languages Workshop at AOSD 2005, p.51–61, 2005.

[3] Colyer, A., Rashid, A. and Blair, G. *On the Separation of Concerns in Program Families*. Technical Report, Computing Department, Lancaster University, 11 p., 2004.

[4] Črepinšek, M., Mernik, M., Bryant, B. R., Javed, F. and Sprague A. *Inferring context-free grammars for domain-specific language.* Electronic notes in theoretical computer science, No.141, p.99–116, 2005.

[5] Ebraert, P. and Tourwe, T. *A Reflective Approach to Dynamic Software Evolution.* In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04), p.37–43, 2004.

[6] Fradet, P. and Sudholt, M. *Towards a Generic Framework for Aspect-Oriented Programming.* Third AOP Workshop, ECOOP'98 Workshop Reader, LNCS, v.1543, p.394-397, 1998.

[7] Gray, J., Bapty, T., Neema, S. and Tuck, J. *Handling crosscutting constraints in domain-specific modeling.* Communications of the ACM, v.44, No.10, p.87–93, 2001.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. *An Overview of AspectJ.* ECOOP'01, LNCS, v.2072, p.327–355, 2001.

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

[10] Kollár, J. *Object Modelling using Process Functional Paradigm.* Proc. ISM'2000, Rožnov pod Radhoštěm, Czech Republic, May 2-4, p.203–208, 2000.

[11] Kollár, J. *Unified Approach to Environments in a Process Functional Programming Language.* Computing and Informatics, 22, 5, p.439–456, 2003.

[12] Kollár, J., Porubän, J. and Václavík, P. *Separating Concerns in Programming: Data, Control and Actions.* Computing and Informatics, 24, 5, p.441–462, 2005.

[13] Ledeczi, Á., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. *The Generic Modeling Environment.* Proc. of WISP'2001, May, Budapest, p.34–42, 2001.

[14] Lieberherr, K., Lorenz, D. H. and Ovlinger, J. *Aspectual Collaborations: Combining Modules and Aspects.* The Computer Journal, v.46(5), p.542–565, 2003.

[15] Mernik, M. and Zumer, V. *Incremental programming language development.* Computer languages, Systems and Structures, v.31, p.1–16, 2005.

[16] Mikkonen, T. *Formalizing Design Patterns.* In Proc. ICSE'98, p.115–124, 1998.

[17] Peyton Jones, S. L. and Hughes, J. [editors] *Report on the Programming Language Haskell 98 – A Non-strict, Purely Functional Language.*, 163 p., 1999.

[18] Walker, D., Zdancewic, S. and Ligatti, J. *A theory of aspects.* In Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, Uppsala, Sweden, ACM Press, p.127–139, 2003.

[19] Wand, M. *A Semantics for Advice and Dynamic Join Points in Aspect–Oriented Programming.* LNCS, 2196, p.45–57, 2001.

[20] De Win, B., Piessens, F., Joosen, W. and Verhanneman, T. *On the importance of the separation-of-concerns principle in secure software engineering.* Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, p.62–76, 2002.

[21] Wu, H., Gray, J. G., Roychoudhury, S. and Mernik, M. *Weaving a debugging aspect into domain-specific language grammars.* Proceedings of the 2005 ACM symposium on applied computing, p.1370–1374, 2005.