An Experiment with Aspect Programming Language

Ján Kollár, Marcel Tóth

Department of Computers and Informatics, Technical University of Košice, Slovakia
Jan.Kollar@tuke.sk, Marcel.Toth@tuke.sk

Abstract: In this paper we our result with the definition of advices in terms of PFL – a process functional language. We provide the comparison of AspectJ imperative programming and PFL functional programming considering a possible approach to advising using types. The reflection property of PFL – a process functional language is introduced, as promising for the systematic development of aspect oriented open language systeme. We also enumerate some problems associated with aspect programming language construction and present our results with aspect oriented analysis based on Kresl language. [1]

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

1 Introduction

Aspect oriented programming has evolved considering, that some concerns in computation are crosscutting over programs developed in structured or object oriented manner. There is high deal of evidence that combination of different concerns of computation in complex software systems yields to scattered and tangled code, which is inappropriate to maintenance and further development [1]. AspectJ [2], [3] is a programming language, in which crosscutting concerns are described using aspect declaration. Aspect is a module, which in addition to class contains

- pointcuts which define a collection of join points, and
- advices – parts of code, applied in join points defined in pointcuts.

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

Join points are points in original modules that are the subject of interest (such as tracing and/or profiling in debugging aspects, or an extension of functionality in development aspects). Pointcut is defined by the pointcut designator, which is a formula that selects a collection of picked out join points, using primitive pointcut designators and logical operations. This selection of join points does not affect the original modules. Instead of that, the advice – a part of code – is woven before, after or instead of each selected join point, using a source-to-source transformer called weaver [9], [19].The crucial problem is the mechanism of adding a new pointcut designator to an aspect language. It is strong feeling that some restrictions given by a concrete programming language must be preceded. Therefore we decided to study aspect oriented paradigm on the basis of a process functional paradigm which may be thought as a practical but still abstracted substance by imperative computation by expression evaluation. It means that although *PFL* – a process functional language [4], [5], [6], [8] is supplied by generators [15], [16], [17], [18] to both Java and Haskell, it still has a very simple syntax of a functional language. In particular, this paper is a sampled analysis and comparison of AspectJ imperative and *PFL* functional aspect oriented features, with respect to reflection property of *PFL*.

2 AspectJ Approach

Suppose Java program (package) consisting of Demo class printing the result value of constant function m (see Fig. 1) and Main class comprising the functions m and f (see Fig. 2).

```
public class Demo {
        public static void main(String[ ] args) {
                Main o = new Main();
                System.out.println(o.m());
        }
}
```

Figure 1
Demo.java – Demo class

```
public class Main {
        public int f(int x, int y) { return(x + y); }
        public int m() { return(f(3, 2)); }
}
```

Figure 2
Main.java – Main class

The execution Main and Demo in Fig. 2 and Fig. 1 yields the value 5. Now, let us require:

1 The value 3 – the first argument of *f* be assigned to a new environment variable (a field) *u*, and the value of *u* (which has been assigned, i.e. 3) is used as an argument of a function *p*, which value will be then used as a first argument of *f*.

2 The second argument of function f is the value of a function q which argument is previously assigned the value of environment variable u (i.e. 3 again).

Without aspect programming methodology, the requirements above are satisfied by modified class Main according to Fig. 3. The result of modified Java program is 22. Using aspect oriented methodology, Main class remains unchanged (in the form illustrated in Fig. 2, and the requirements are satisfied by new aspect *Extend* definition, illustrated in Fig. 4.

```
public class Main {
        private int u;
        private int p(int x)          { return(x * x); }
        private int q(int x)          { return(10 + x); }
        public int f(int x, int y){ return(x + y); }
        public int m()     { return(f(p(this.u = 3), q(this.u))); }
}
```

Figure 3
Main.java – modified Main class

```
aspect Extend {
        private int Main.u;
        private int Main.p(int x) { return(x * x); }
        private int Main.q(int x) { return(10 + x); }
        int around (Main o) :
        target(o) && call(public int m()) {
                return(o.f (o.p(o.u = 3), o.q(o.u)));
        }
}
```

Figure 4
Aspect.java – Aspect definition

In this aspect definition, we define new environment variable and the functions *p* and *q*, private to Main.class. The crucial part of aspect definition is around advice, which picks out all join points that are called as public integer constant functions named *m* (by primitive pointcut designator *call(public int m())* in target object *o*, of type *Main*, which value is exposed to the advice. Then the advice *return(o.f (o.p(o.u = 3), o.q(o.u)));*, where *o* is the object, is used instead of the body *return(f(3, 2));* of *m* in original *Main class*.

The advantage of defining the aspect *Extend* is poor in this case, since just one join point was selected. However, commonly it is possible to select multiple join points and then apply the advice to each of them. For example *call(* * m*(. .))* would select any function of any type, any number and types of arguments, such that names begin with the character *m*. Instead of anonymous pointcuts, as in around advice above, the pointcuts can be named, see Fig. 5.

```
pointcut extend(Main o) :
        target(o) && call(public int m());
int around (Main o) : extend(o) {
        return(o.f (o.p(o.u = 2), o.q(o.u)));
}
```

Figure 5
Definition and application of named pointcut

In pointcut *extend* definition, object value *o* is selected by (primitive) pointcut designator *target*, i.e. it is pulled right to left across ':' in pointcut definition.

Then, in around advice, the value *o* is pulled right to left across ':' from pointcuts to advice, and then to advice body.

An example above is based on pointcut, which defines static joint points that are the subject of compile time weaving. Opposite to static join points, dynamic joint points are such that are defined in dynamic context of program i.e. while execution. An example is *cflow* pointcut designator in *AspectJ*, which selects join points occurring in all methods called from a given method of a class. Then, instead static weaving, dynamic (i.e. runtime) weaving acts to perform advising in dynamic join points. The mechanism, which allows to identify run-time crosscutting is computational reflection [13], which is defined as the capability of a computational system to reason about itself and act upon itself, and adjust to changing conditions. A reflective system incorporates data representing static and dynamic aspects of it; this activity is called reification. This self-representation makes it possible for the system to answer questions about and support actions on it.

3 Reflection in *PFL*

Let us define a *PFL* Demo and Main modules, corresponding to Java program in Fig. 2 and Fig. 1. But since of separated class and multiple instance concept, here we have three modules: Demo in Fig. 6 and a pair: Main – class definition in Fig. 7, and Main – instance definition in Fig. 8.

```
main :: ()
main = print (Main => m)
```

Figure 6
PFL Demo module

```
class Main where
        f :: Int → Int →Int
        m :: Int
```

Figure 7
PFL Main class $C^{(0)}$

```
instance Main where
        f x y = x + y
        m = f 3 2
```

Figure 8
PFL Main instance $I^{(0)}$

The evaluation of process *main* in Demo module proceeds according to (1).

```
main = print (Main => m)
=> print (f 3 2)
=> print (3 + 2)                                    (1)
=> print 5
=> 5
```

where 5 is the number 5, displayed by built-in operation *print :: Int ! ()*.

In *PFL*, we provide powerful mechanism for manipulating environment variables, without introducing them in function bodies, but rather in type definitions of functions (such functions are then called processes).

Let us require each argument value of function *f* be reflected in two separated environment variables *u* and *v*. In *PFL*, this can be achieved by the modification of class *Main*, introducing u and *v* in the type definition for *f*, according to Fig. 9.

```
class Main where
        f :: u Int →v Int →Int
        m :: Int
```

Figure 9
PFL Main class $C^{(1)}$

If re-compiled by *W*, the result affects both class *Main* and all its instances (but in this case just one instance, because the class is monomorphic), as follows:

$$\mathbf{W(C^{(1)}, I^{(0)}) = (}\mathit{Env}\mathbf{, C^{(0)}_E , I^{(1)})} \qquad \mathbf{(2)}$$

It means, that new Env class is generated (see Fig.10), class $C^{(0)}{}_E$ (see Fig.11) is almost the same as original $C^{(0)}$, except that its context is a new generated *Env* superclass, while each instance $I^{(0)}$ is transformed to $I^{(1)}$, according to Fig. 12. The semantics has changed according to (3).

$$[\![W(C^{(1)}, I^{(0)})]\!] \neq [\![W(C^{(0)}, I^{(0)})]\!] \quad (3)$$

```
class (Env a Int) where
        env u^e :: Int
        env v^e :: Int
                u :: a → Int
                v :: a → Int
instance (Env Int Int) where
                u x = let u^e = x in u^e
                v x = let v^e = x in v^e
instance (Env () Int) where
                u () = u^e
                v () = v^e
```

Figure 10
Generated Environment class and Instances

```
class (Env a Int) => Main where
        f :: Int → Int → Int
        m :: Int
```

Figure 11
PFL Main class $C^{(0)}{}_E$ (in *Env* context)

```
instance Main where
        f x y = x + y
        m = f (u 3) (v 2)
```

Figure 12
Transformed *PFL* Main instance I(1)

However, the function of computation remains unchanged, since for all x and an environment variable u^e (and for v^e) it holds

$$x = \textbf{let } u^e = x \textbf{ in } u^e$$

This is evident from the evaluation (4), in which each reflection action re is enclosed in re box. Reflection is the most important property of an aspect programming language, since it enables to reason about 'hidden' properties of computation that exist but they may be invisible to a user. Such properties are for example as follows: timer values, register values, interrupt signals, message signals, etc.

main **= print (*Main* => *m*)**
=> print (*f* (*u* 3) (*v* 2))
=> print (*f* (let $u^e = 3$ in u^e) (*v* 2))
=> print (*f* 3 (*v* 2)) **(4)**
=> print (*f* 3 (let $v^e = 2$ in v^e))
=> print (*f* 3 2)
=> print (3 + 2)
=> print 5
=> 5

In *PFL*, we manipulate the state implicitly, generating new environment and two functions for accessing and updating the cells. It is over the scope of this paper, but this approach can be extended to any other cells, even residing at physical memory addresses. Moreover, since the source purely functional *PFL* definitions are defined in terms of expressions, hence each expression and its subexpression can be abstracted. That is why, reflection environments can be generated using also the type definitions of abstracted processes.

4 *PFL* Advices

To achieve the semantics of Java program defined by aspect *Extend* in Fig. 4 we must do two things. First, we must write two additional functions *p* and *q*, in the form (5)

p* :: *Int* → *Int
p x* = *x* * *x **(5)**
q* :: *Int* → *Int
q x* = 10 + *x

Second, we must rewrite the type definition for *f* in *PFL* Main class. Clearly, something like this

f* :: *u Int* → *u Int* → *Int

is not satisfactory, since *p* and *q* would never be applied, and the result is 5 again. Since *Env* contains just definitions for single variable u^e, both arguments (3 and 2) of *f* are reflected using the same variable u^e, nothing more.

However, considering Java woven form for *m* according Fig. 3, which is as follows:

public *int m*() { *return*(*f*(*p*(*this.u* = 3), *q*(*this.u*))); }

as well as the target form for *m* in Fig. 8, which is as follows:

***m* = *f* (*u* 3) (*v* 2)** **(6)**

we can take into account the environment variable access function *u* in the instance (*Env* () *Int*) in Fig. 10), and derive the woven definition of *m*, which is as follows:

***m* = *f* (*p* (*u* 3)) (*q* (*u* ())**

Using composition (○), this definition may be expressed in the form (7).

***m* = *f* ((*p* ○ *u*) 3)) ((*q* ○ *u*) ())** **(7)**

Hence, *PFL* concept of attributed types by environment variables is extended to attributing types by any processes, that are used as advices in aspect process functional programming. The solution is introduced in Fig. 13 which yields the required result of computation 22, as in AspectJ case (provided that *p* and *q* are defined), according to (5).

class *Main* where
f* :: (*p* ○ *u*) *Int* → (*q* ○ *u*) () → *Int
m* :: *Int

Figure 13
PFL Main class equivalent to AspectJ advice Extend in Fig. 4

5 Problems of Aspect Enhancement of *PFL*

The first step of introducing aspects to programming techniques was AspectJ, as seamless integration to object oriented language - Java. Since AspectJ was released quite long ago, it allowed everyone to check its properties in real projects.

Our intention is to do the same with the *PFL* language, which we have developed. For this purpose we have tested aspect oriented approach on simple object oriented language (Kresl) with its weaver, compiler and interpreter [14], briefly described in next chapter. This allowed us to check aspect principles on simple language and thus pick out potential problems and ways of future development (integration of AOP to *PFL*).

To effectively work with aspect ideas under *PFL* it is probably also needed to enhance *PFL* with language constructions, which will simplify functional notation of advices, aspects, pointcut designators and the others of aspect notions.

Joining aspect and *PFL* paradigms as well as enhancing another paradigms (object oriented, …) with aspect conceptions meets some hidden, but crucial problems, that have to be solved to reach destined effectiveness, namely:

- Strong dependency of aspect weaving on component source code. One small change in component source code (e.g. change of variable name) can cause weaving not to work as expected.
- Selecting join points using predefined pointcut designators (e.g. *call*, *within*, …). There is a strong feel, that more flexible system allowing self defining of pointcut designators would fit the needs better.

There is still the task of identifying potential problems and solving of listed ones. This is the aim of further work.

6 Experiment with Aspect Analysis

In this experiment we have created simple language (with its aspect extension) as well as its weaver, compiler and interpreter. All of them are stand alone modules, connected through windows application (integrated environment). The role of this experiment was to clearly illustrate the conception of aspect approach (and its syntactic analysis) on simplified language, that is easy to understand and whose interpretation produces visible results. Short description of each of developed modules follows:

Kresl language – Simple procedural language supporting majority of common procedural languages constructions (e.g. variables, declarations, functions, statements, arithmetical and logical expressions, loops …). Besides, the language contains keywords for working with pen on interpretation canvas (moving of pen, drawing geometric shapes, changing color, …).

Weaver of Kresl language – This is the module, where all actions related to aspect approach are taking place (aspect oriented syntactic analysis, weaving of input files, …). The module takes two input files (first with definition of aspects and the second with component code) and produces one weaved file in Kresl language.

Compiler of Kresl language – Compiler translates output of weaver to assembly language defined in [6], which is interpreted by interpreter.

Interpreter – The module draws results of Kresl program on interpretation canvas. Kresl language supports drawing of geometrical shapes (line, rectangles, ovals) and also drawing of text.

There is a support for several predefined pointcut designators in the language (call, set, get). The process of weaving, translating and interpreting of input file is visual, so the programmer can see changes of source code immediately after weaving, translating, etc. Although we used merely static weaving in this project, we were able to illustrate basic principles of aspect programming on simple language, which was our aim.

Conclusions

By attributing the argument types by compositions of processes and variables (such as (p ○ u) in Fig. 13 we can advise them to all applications of a function *f*. Notice, that *p* and *q* can be defined in a separate class (similarly as *Env*), which is the second superclass of the class *Main*. Clearly, this class, together with new type definition for *f*, in the form

f :: (p ○ u) Int → (q ○ u) () → Int

is *PFL* advice for *Main*. The result of weaving will not change *f* type definition in original *Main*, which must be taken into account while weaving, otherwise it would be impossible to check argument type *(q ○ u) () :: Int* whenever values of variables are used instead values of expressions.

It is also necessary to make syntactic distinction between functions and variables. Otherwise all undefined functions could be calmly expected to be variables, but this seems to be a very dangerous approach.

It is also not so clear if it is good idea to write around control advice in the form (), or in the form (*Int*), which provides opportunity for the additional type checking.

Many other questions have arised, such as the mutual relation of monadic [19] and aspect oriented programming, the effects of different sequences when adding new aspects, reasoning about aspect oriented systems while profiling them [10], [11], [12], the need for post-conditions, implementing event-based pointcut designators, etc. that are the subject of our current research.

References

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

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

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

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

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

[6] Kollár, J.: J - Pascal specification, , Research report DCI FEII TU Košice, May 2-4, 1997

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

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

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

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

[11] Porubän J.: Time and space profiling for process functional language, Proc. EMES'03, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 167-172, ISSN-1223-2106

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

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

[14] Tóth, M.: Aspect-oriented syntactic analysis. Masters thesis, 2004, DCI FEII TU Košice

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

[16] Václavík, P., Porubän, J.: Object Oriented Approach in Process Functional Language, Proc. ECI2002", October 10-11, 2002, Košice - Herlany, 2002, pp. 92-96, 80-7099-879-2

[17] Václavík, P.: The Fundamentals of a Process Functional Abstract Type Translation, Proc. EMES'03, May 29-31, 2003, Felix Spa - Oradea, University of Oradea, 2003, pp. 193-198, ISSN-1223-2106

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

[19] Wadler, P.: The essence of functional programming, In 19th Annual Symposium on Principles of Programming Languages, Santa Fe, New Mexico, January 1992, draft, 23 pp

[20] Wand, M.: A semantics for advice and dynamic join points in aspect-oriented programming. LNCS, 2196:45-57, 2001