CASB: Common Aspect Semantics Base

ABSTRACT

This document gradually introduces formal semantic descriptions of aspect mechanisms.

Document Id	:	AOSD-Europe-INRIA-7
Deliverable No.	:	41
Work-package No.	:	8
Type	:	Integration
Status	:	Final
Version	:	1.0
Date	:	10 February 2006
Author(s)	:	Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, Didier Le Botlan
Contributor(s)	:	

Contents

1 Common Aspect Semantics Base

This document gradually introduces formal semantic descriptions of aspect mechanisms. To begin with, we present minimal requirements on the base language semantics. Then, we consider the weaving of a single aspect, in particular *before*, *after* and *around* aspects. We extend the model with multiple aspects, cflow pointcuts, aspects on exceptions, aspect deployment, aspect instantiation and stateful aspects.

We do our best to describe aspects as independently as possible from the base language. For each aspect feature, we introduce the minimal constructions of the base language necessary to plug aspects in. For example, a *before* aspect does not requires any special mechanisms: the base language semantics should only respect the common requirements of Section 2. Aspects using cflow-like pointcuts assume the base language to have a call & return instructions (*e.g.* procedures, functions or methods).

Many features and examples are inspired from AspectJ but our descriptions are usually more general. For example, our description of around aspects applies to a larger class of instructions than just method calls. Usually we introduce only the minimum constraints on the base language so that the aspectual feature described makes sense. In many cases, our descriptions could be applied to many different types of programming languages (object-oriented, imperative, functional, logic, assembly, ...).

2 Hypotheses on the base language semantics

The base language semantics must be described in terms of a small-step semantics (aka SOS), formalized through a binary relation $\rightarrow_b$ on configurations made of a program and a state (C, Σ).

A program C is a sequence of basic instructions i terminated by the empty instruction ε:

$$C ::= i : C \mid \varepsilon$$

We will abuse the notation and write, for example, $C_1 : C_2$ to denote the concatenation of two programs. The operator ":" is supposed associative and, implicitly, programs are supposed to be of the form $i_1 : (i_2 : \ldots : (i_n : \varepsilon) \ldots)$.

States Σ are kept as abstract as possible. They may contain environments (*e.g.* associating variables to values, procedure names to code, etc.), stacks (*e.g.* evaluation stack), heaps (*e.g.* dynamically allocated memory), etc.

A single reduction step of the base language semantics is written

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$

Intuitively, i represents the current instruction and C the continuation. The component $i : C$ can be seen as a control stack. The operator ":" sequences the execution of instructions. We rely on this property to define before and after aspects. Final configurations are of the form (ε, Σ).

EXAMPLE **1** *The semantics of the small arithmetic language*

$$E ::= k \mid E_1 + E_2$$

can be described in this setting as:

$$\begin{array}{lll}(E_1+E_2:C,S) & \rightharpoonup_b & (E_1:E_2:+:C,S)\\ (+:C,k_1:k_2:S) & \rightharpoonup_b & (C,k_1+k_2:S)\\ (k:C,S) & \rightharpoonup_b & (C,k:S)\end{array}$$

The state is made of an evaluation stack. Evaluating an expression E_1+E_2 amounts to evaluating E_1 and E_2 before performing the addition. The three instructions corresponding to these tasks are placed into the control stack. The evaluation of an integer pushes it onto the evaluation stack. An addition replaces the top integers on top of the evaluation stack by their sum.

EXAMPLE **2** *The semantics of the small imperative language*

$$S ::= S_1;S_2 \mid f=S \mid \mathit{call}\ f$$

can be described in this setting as:

$$\begin{array}{llll}(S_1;S_2:C,\rho) & \rightharpoonup_b & (S_1:S_2:C,\rho) & \\ (f=S:C,\rho) & \rightharpoonup_b & (C,\rho[f\mapsto S]) & \\ (\mathit{call}\ f:C,\rho) & \rightharpoonup_b & (C':C,\rho) & \mathit{if}\ \rho(f)=C'\end{array}$$

The state is made of an environment (a function) ρ associating identifiers (f) to their code ($\rho(f)$). Evaluating a sequence $S_1;S_2$ amounts to evaluate S_1 and S_2 in turn. A definition $f=S$ updates the environment so that it associates the name f to the code S. A call to f pushes the associated code in the control stack.

Another possibility could be to compile the language by translating every sequence ";" (a source language sequencing operator) by ":" (the semantic constructor representing sequencing). The first semantics rule would disappear.

Most instructions executes without deleting nor referring to their continuation. We say that such instructions *respect sequencing*. Formally:

DEFINITION **3** *An instruction i respects sequencing if*

$$(i:\varepsilon,\Sigma)\rightharpoonup_b(C',\Sigma')\Rightarrow(i:C,\Sigma)\rightharpoonup_b(C':C,\Sigma')$$

All instructions seen in the examples above respect sequencing whereas jumps, call/cc or exceptions would not.

It is sometimes useful to retain some structure within the program being evaluated. We extend programs with the notion of block to represent sub-programs.

$$C ::= i:C \mid \{C_1\}:C_2 \mid \varepsilon$$

With this extension, an instruction can be a block of instructions. such as $\{i_1:\ldots:i_n:\varepsilon\}$. The reduction rule for blocks is just

$$(\{C_1\}:C_2,\Sigma)\rightharpoonup_b(C_1:C_2,\Sigma)$$

EXAMPLE 4 *Blocks can be useful to distinguish return addresses. If we consider again the small imperative language above then the reduction of the program*

$$(f = call\ g; i_3); (g = i_1; i_2); call\ f$$

will lead to the configuration $(i_1 : i_2 : i_3 : \varepsilon, \rho)$ *where it is impossible to distinguish the continuations of calls to* f *and* g*. If we use blocks in the rule for calls as follows*

$$(call\ f : C, \rho) \rightarrow_b (C' : \{C\}, \rho) \ \ if\ \rho(f) = C'$$

then, the previous configuration will be $(i_1 : i_2 : \{i_3 : \varepsilon\}, \rho)$ *which makes clear that* $i_1 : i_2$ *are instructions of the current function and* i_3 *is a return address.*

3 Weaving a single aspect

The semantics represents an *aspect* as a function ψ to be applied to the current instruction i that returns a function ϕ. This second function takes the state Σ as parameter and returns a pair (a,t):

$$\psi(i) = (\phi, t) \qquad \text{and} \qquad \phi(\Sigma) = a$$

where a denotes the advice (supposed to be written in the same language as the base program) to be inserted, and t $(= before, after, around, \ldots)$ denotes the kind of aspect.

These two functions ψ and ϕ can be seen as two steps to decide which joinpoints are woven. The first function ψ takes only static information (*e.g.* syntax) into account, while the second one ϕ uses dynamic information (*e.g.* runtime values). The function ψ returns $\emptyset$ when there is no advice for the current instruction. The function ϕ returns ε when there is no advice for the current state. In AspectJ, the function ψ can be interpreted as the compiler that instruments the code with an advice that starts with dynamic checks (*i.e.* the function ϕ).

The semantics of weaving is described in terms of a relation $\rightarrow$ on configurations. The rule NOADVICE below executes the current instruction i if no advice is to be executed.

$$\text{NOADVICE} \quad \frac{\psi(i) = \emptyset \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma) \rightarrow (C', \Sigma')}$$

In order, to prevent an instruction i to be matched, we introduce the notion of tagged instructions (written $\bar{i}$). A tagged instruction $\bar{i}$ has exactly the same semantics as i except that it is not subject to weaving. Formally

$$\forall (i, C, \Sigma) \ \ (i : C, \Sigma) \rightarrow_b (C', \Sigma') \Rightarrow (\bar{i} : C, \Sigma) \rightarrow (C', \Sigma')$$

$$\forall i \ \ \psi(\bar{i}) = \emptyset$$

Some aspect oriented languages consider only the weaving of the base program and rule out the weaving of advice code. This can be represented by tagging all advice instructions.

In the following subsections we present the semantics rules for *before*, *after* and *around* aspects. Our semantic descriptions always consider that advice is subject to weaving.

3.1 *Before* aspect

When a before aspect matches the current instruction, its advice is executed before reducing this instruction. If the before aspect ψ matches the current instruction, the rule BEFORE tags the current instruction and inserts *test* ϕ before.

$$\text{BEFORE}\quad \frac{\psi(i) = (\phi, \mathit{before})}{(i : C, \Sigma) \to (\mathit{test}\ \phi : \bar{i} : C, \Sigma)}$$

When *test* ϕ is the current instruction, the rule ADVICE applies ϕ it to the current state Σ in order to insert the corresponding advice.

$$\text{ADVICE}\quad \frac{}{(\mathit{test}\ \phi : C, \Sigma) \to (\phi(\Sigma) : C, \Sigma)}$$

Note that the instructions of the advice can be matched by the aspect.

3.2 *After* aspect

The intuition behind an after aspect is to execute the advice after the current instruction has completed. To make sense, it should be applied to instructions which respect sequencing.

The rule AFTER inserts the advice function after the current instruction and tags the current instruction if the after aspect ψ matches the current instruction. If an advice has to be executed after i, the configuration $(i : C, \Sigma)$ is transformed into $(\bar{i} : \mathit{test}\ \phi : C, \Sigma)$. The instruction i cannot be matched again and the next reduction step of the configuration will be done using $\to_b$.

$$\text{AFTER}\quad \frac{\psi(i) = (\phi, \mathit{after})}{(i : C, \Sigma) \to (\bar{i} : \mathit{test}\ \phi : C, \Sigma)}$$

If the instruction does not respect sequencing (*e.g.* it can throw exceptions) then the advice might not be executed. If the instruction is a procedure call, the advice will be executed when the procedure returns.

3.3 *Around* aspect

In order to accommodate *around* aspect, the base language must contain an additional instruction (proceed) which can be used in the code of an *around* advice.

Typically, an around aspect starts by executing its advice before the current instruction. The advice code may proceed by executing the instruction matched by the *around* aspect (using the instruction proceed). The advice may also terminate without executing the current instruction: the advice has completely replaced the instruction.

In AspectJ, execution of around advices may be more complex than this. For example, an around advice may contain several proceed resulting in multiple executions of the instruction matched by the *around* aspect. The advice of an around aspect may be matched by another around aspect and one has to keep track to which instruction each proceed is referring to.

To represent the behavior of AspectJ-like around aspects we introduce the following additional semantic components:

- ◦ a special stack P called the *proceed stack*,
- ◦ the semantic function $push_p\ i$ which pushes the instruction i in the proceed stack,
- ◦ the semantic function pop_p which removes the top of the proceed stack.

The rule AROUND inserts the advice function followed by pop_p and pushes the current instruction in the proceed stack so that it can be possibly executed by a proceed.

The rule PROCEED executes the instruction placed on top of the proceed stack. This instruction is removed (i may be the code of an enclosing around aspect whose proceed would refer to the top of the stack P not i) and replaced after completion (using $push_p\ i$) since the advice may contain other proceeds.

The rule POP terminates the current advice by removing the instruction on top of the proceed stack.

$$\textsc{Around}\quad \frac{\psi(i) = (\phi, around)}{(i : C, \Sigma, P) \rightarrow (test\ \phi : pop_p : C, \Sigma, \bar{i} : P)}$$

$$\textsc{Proceed}\quad \frac{}{(\mathsf{proceed} : C, \Sigma, i : P) \rightarrow (i : push_p\ i : C, \Sigma, P)}$$

$$\textsc{Pop}\quad \frac{}{(pop_p : C, \Sigma, i : P) \rightarrow (C, \Sigma, P)}$$

EXAMPLE **5** *Let us consider the previous small imperative language and an aspect ψ such that*

$$\psi(call\ foo) = (\phi, around)$$

$$\phi(\Sigma) = call\ bar; proceed; call\ baz$$

This aspect inserts a call to bar (resp. baz) before (resp. after) each call to foo. An example of reduction is:

$$\begin{array}{llr}
(call\ foo : \varepsilon, \rho, \varepsilon) & & \\
\rightarrow\ (test\ \phi : pop_p : \varepsilon, & \rho, & \overline{call\ foo} : \varepsilon) \\
\rightarrow\ (call\ bar; proceed; call\ baz : pop_p : \varepsilon, & \rho, & \overline{call\ foo} : \varepsilon) \\
\rightarrow^*\ (\underline{proceed} : call\ baz : pop_p : \varepsilon, & \rho', & \overline{call\ foo} : \varepsilon) \\
\rightarrow\ (\overline{call\ foo} : push_p(call\ foo) : call\ baz : pop_p : \varepsilon, & \rho', & \varepsilon) \\
\rightarrow^*\ (push_p(\overline{call\ foo}) : call\ baz : pop_p : \varepsilon, & \rho'', & \varepsilon) \\
\rightarrow^*\ (call\ baz : pop_p : \varepsilon, & \rho'', & \overline{call\ foo} : \varepsilon) \\
\rightarrow^*\ (pop_p : \varepsilon, & \rho''', & \overline{call\ foo} : \varepsilon) \\
\rightarrow\ (\varepsilon, & \rho''', & \varepsilon)
\end{array}$$

Note that our semantics of around aspects is not limited to calls but can be applied to any instruction.

4 Weaving several aspects

We now consider the weaving of several aspects at the same join point. We first consider the weaving of several aspects of the same kind. Then we consider the general case of weaving *before*, *after* and *around* aspects at the same join point.

4.1 Aspects of the same kind

The aspects matching an instruction i are represented by a tuple of advice functions and a kind

$$\psi(i) = ((\phi_1 \dots \phi_n), t)$$

with $t = before, after$ or $around$.

The order of execution of the advices is made by the function ψ. It is the order of occurrence of the advice functions in the tuple.

Before aspects

When *before* aspects match an instruction, their advices are executed before reducing this current instruction. As the rule BEFORE, the rule BEFORE* tags the current instruction to prevent matching it again and inserts the advice functions before.

$$\textsc{Before*} \quad \frac{\psi(i) = ((\phi_1 \dots \phi_n), \mathit{before})}{(i : C, \Sigma) \rightarrow (\mathit{test}\ \phi_1 : \dots : \mathit{test}\ \phi_n : \bar{i} : C, \Sigma)}$$

After aspects

When *after* aspects match an instruction, their advices are executed after reducing the current instruction. As the rule AFTER, the rule AFTER* inserts the advice functions after this current instruction and tags this current instruction.

$$\textsc{After*} \quad \frac{\psi(i) = ((\phi_1 \dots \phi_n), \mathit{after})}{(i : C, \Sigma) \rightarrow (\bar{i} : \mathit{test}\ \phi_1 : \dots : \mathit{test}\ \phi_n : C, \Sigma)}$$

Around aspects

The rule AROUND* inserts the first function and pushes all the other advice functions and the current instruction in the *proceed stack*. As before, advice can perform 0, 1, or several proceeds. If n advices $(a_1, \dots, a_n)$ match a instruction i and each advice is of the form a'_i : proceed : a''_i then the execution will be of the form

$$a'_1 \rightarrow a'_2 \rightarrow \dots \rightarrow a'_n \rightarrow i \rightarrow a''_n \rightarrow \dots \rightarrow a''_2 \rightarrow a''_1$$

If we change a_1 to a'_1 : proceed : a''_1 : proceed : a'''_1 the execution will look like

$$a'_1 \rightarrow \dots \rightarrow a'_n \rightarrow i \rightarrow a''_n \dots a''_2 \rightarrow a''_1 \rightarrow a'_2 \dots \rightarrow a'_n \rightarrow i \rightarrow a''_n \dots a''_2 \rightarrow a'''_1$$

If we further remove the proceed of a_2 the reduction will look like

$$a_1' \rightarrow a_2 \rightarrow a_1'' \rightarrow a_2 \rightarrow a_1'''$$

The rule AROUND* inserts the first advice function followed by $pop_p\ n$ which is responsible to remove the other advice functions and the instruction after completion.

The rule PROCEED* executes the next advice or instruction placed on top of the proceed stack. It is the same rule as before. If the instruction execution is an advice it will possibly execute the next instruction in the proceed stack and will eventually terminate by reintroducing itself in the proceed stack.

The rule POP* terminates the current advice by removing the top n instructions (the $n-1$ advices and the matched instruction) of the proceed stack.

$$\text{AROUND*}\ \frac{\psi(i) = ((\phi_1 \ldots \phi_n), around)}{(i : C, \Sigma, P) \rightarrow (test\ \phi_1 : pop_p\ n : C, \Sigma, test\ \phi_2 : \ldots : test\ \phi_n : \bar{i} : P)}$$

$$\text{PROCEED*}\ \frac{}{(\mathsf{proceed} : C, \Sigma, x : P) \rightarrow (x : push_p\ x : C, \Sigma, P)}$$

$$\text{POP*}\ \frac{}{(pop_p\ n : C, \Sigma, x_1 : \ldots : x_n : P) \rightarrow (C, \Sigma, P)}$$

4.2 *Before*, *After* and *Around* aspects

We now consider that the more general case of several aspects of different kinds matching a join point. The aspects matching an instruction i are represented by a function ψ returning a tuple of pairs made of an advice function and a kind:

$$\psi(i) = ((\phi_1, t_1) \ldots (\phi_n, t_n))$$

with $t_i = before, after$ or $around$.

The function γ translates such a tuple in an equivalent tuple of around only aspects using the two following rules:

$$(\phi, before) \longmapsto (\lambda\Sigma.(test\ \phi : \mathsf{proceed}), around)$$

$$(\phi, after) \longmapsto (\lambda\Sigma.(\mathsf{proceed} : test\ \phi), around)$$

A *before* aspect is translated in a *around* aspect that possibly inserts the advice (*i.e. test* ϕ) before it proceeds with the next aspect. Symmetrically, an *after* aspect is translated in a *around* aspect that possibly inserts the advice *after* the next aspect is executed. Remember the function ϕ takes the state as a parameter, so the translations have to start with $\lambda\Sigma$. Here, *test* ϕ at the beginning of the translated *before* aspects inspects the current state (see the rule ADVICE). In the translated *after* aspect, *test* ϕ inspects the state after the other aspects execution (*i.e.* proceed).

The new AROUND* rule is similar to the previous one, but it calls γ.

$$\text{AROUND*}\ \frac{\psi(i) = ((\phi_1, t_1)\dots(\phi_n, t_n)) \qquad \gamma((\phi_1, t_1)\dots(\phi_n, t_n)) = ((\phi'_1, around)\dots(\phi'_n, around))}{(i : C, \Sigma, P) \rightarrow (test\ \phi'_1 : pop_p\ n : C, \Sigma, test\ \phi'_2 : \dots : test\ \phi'_n : \bar{i} : P)}$$

4.3 Option

Our semantics is based on two functions ψ and ϕ. We could also use only one function that combines both. For instance, let us consider a new version of ψ that takes both the current instruction i and the current state Σ as parameters and returns a tuple of pairs made of an advice and a kind:

$$\psi(i, \Sigma) = ((a_1, t_1)\dots(a_n, t_n))$$

In this case, we use the same state Σ to decide which aspects match the current instruction i (*i.e.* an advice execution does not influence whether or not the following aspects will be woven). This option makes it simpler to predict when more than one advice can be woven at the same instruction.

5 Pointcuts

An aspect is made of a pointcut selecting some join points, an advice (*i.e.* a code to execute) and a kind (*e.g.* *before*, *after*, *around*). Until now, we have abstracted aspects in a function ψ. In this section, we make more precise the structure of this function and consider several types of pointcuts.

If we represent pointcuts by patterns, the function ψ can be written as follow:

$$\psi(i) = if\ match(P, i)\ then\ (\sigma(\phi), type)\ else\ \varepsilon$$

$$with\ \sigma\ such\ that\ \sigma(P) = i$$

The aspect selects a join point i by matching it against a pattern (pointcut) P. We represent the matching process by the function *match* which takes a pattern, an instruction and returns a boolean.

$$match : P \times Instruction \rightarrow bool$$

where *Instruction* is the set of instructions. In case of a match, the advice and its type is returned. Information (names, types, etc.) can be passed from the instruction to the advice using the substitution (σ) unifying the pattern with the instruction.

The pattern P can be a term with variables matching an instruction, or disjunction, conjunction and negation of patterns. Thus, during the execution of the program, an aspect matches an instruction, if the pattern of that aspect matches this instruction. Standard patterns are described by the following grammar:

$$P ::= T_i \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P$$

The term T_i follows the grammar of instructions (left unspecified here) but includes pattern variables to match arbitrary instructions. The boolean function *match* is defined as follows:

$$\begin{array}{lcl} match(T_i, i) & = & true \quad \text{if } \exists\, \sigma \text{ such that } \sigma(T_i) = i \\ & = & false \quad \text{otherwise} \\ match(P_1 \wedge P_2, i) & = & match(P_1, i) \wedge match(P_2, i) \\ match(P_1 \vee P_2, i) & = & match(P_1, i) \vee match(P_2, i) \\ match(\neg P, i) & = & \neg match(P, i) \end{array}$$

Boolean operators (especially the negation) may lead to complications. For example, a pattern can match an instruction but according to several substitutions (consider for example the pointcut $\neg call\ x$ matching all instructions different from a call). In these cases, no pattern variables should occur in the advice.

5.1 *Cflow(below)* pointcuts

cflow(B), is a pointcut which intuitively represents all the join points which are in the control flow of a method/procedure call B including the join point represented by B. *cflowbelow*(B) is similar but excludes the the join point represented by B. To describe the semantic of such pointcuts we introduce new instructions, namely method definition and call:

$$Prog ::= (T\ \mathsf{id}()\{S\})^*\ S$$

$$T ::= \mathsf{void} \mid \mathsf{int} \mid \ldots$$

$$S ::= \mathsf{call\ id}() \mid \ldots$$

In this grammar, $T\mathsf{id}()\ \{S\}$ represents the declaration of the procedure/method id and T (void, int, etc) its return type. A program consists in a collection of procedures/methods declarations followed by a main command. Commands include instructions call id() which are calls to id. The semantic of those instructions are expressed by the rules below. Configurations are extended with an environment ρ and a stack F. The environment ρ is a function which associates to each *id* of a procedure its body and return type. The stack F contains the signature of all the calls which have not returned yet. The program is in the control flow of all calls whose signatures are contained in F. A call to a procedure inserts a block representing its return address and the body of the procedure contained in the environment. It also pushes the signature corresponding of the procedure call on F. Blocks, which represent a return instruction, will remove that signature on exit.

$$\textsc{Call}\quad \frac{\rho(id) = (C',t)}{(\textsf{call id()} : C, \Sigma, \rho, F) \rightarrow_b (C' : \{C\}, \Sigma, \rho, (t)id : F)}$$

$$\textsc{Ret}\quad \frac{}{(\{C\}, \Sigma, \rho, (t)id : F) \rightarrow_b (C, \Sigma, \rho, F)}$$

The pointcut *cflow*(B) selects all the join points which are in the control flow of the pointcut B. Thus, *cflow*(B) matches an instruction, if this join point is in the control flow of B. The semantic of *cflow* and *cflowbelow* is described by extending the matching function *match* to take into account the stack F. The boolean function $match_f$ takes a pattern, an instruction and a stack of signatures.

$$match_f : P \times Instruction \times Sig^* \rightarrow bool$$

where P is the set of pointcuts, $Instruction$ is the set of instructions and Sig^* is a stack of signatures corresponding to procedure calls. The pointcuts have the following syntax:

$$P ::= T_i \mid cflow((P_T)P_I) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P$$

$$P_T ::= x \mid void \mid int \mid \ldots$$

$$P_I ::= x \mid id$$

In this grammar, $(P_T)P_I$ is a pattern matching any signature corresponding to a procedure call whose identifier is matched by P_I. The optional type pattern P_T matches the return type. The patterns P_T (resp. P_I) are either a type (resp. an identifier) or a pattern variable x. The matching function $match_f$ taking into account *cflow*$(below)$ pointcuts is defined as follows:

$$match_f(cflow((P_T)P_I), i, F) = \begin{cases} ((match(P_T,t) \wedge match(P_I,id) \wedge\ i = \textsf{call id} \quad with\ \rho(id) = (\_,t)) \\ \vee \\ (\exists\ (t)id \in F. match(P_T,t) \wedge match(P_I,id)) \end{cases}$$

$$\begin{array}{lcl} match_f(cflowbelow(((P_T)P_I), i, F) & = & \exists\ (t)id \in F. match(P_T,t) \wedge match(P_I,id) \\ match_f(P_1 \wedge P_2, i, F) & = & match_f(P_1,i,F) \wedge match_f(P_2,i,F) \\ match_f(P_1 \vee P_2, i, F) & = & match_f(P_1,i,F) \vee match_f(P_2,i,F) \\ match_f(\neg P, i, F) & = & \neg match_f(P,i,F) \end{array}$$

A join point is in the control flow of a pointcut $(P_T)P_I$ if $(P_T)P_I$ matches either the signature of the current instruction if this instruction is call id or $(P_T)P_I$ matches a signature in the stack F. A join point is below the control flow of a pointcut $(P_T)P_I$ is that $(P_T)P_I$ matches a signature in the stack F. Boolean combination of patterns are treated as before.

6 Aspects on specific linguistic features

In this section, we describe several aspectual features taken from AspectJ: aspects on exceptions (around throws, after throwing and handler) and aspect instantiation. They involve to introduce special instructions on the base language. For example, to specify aspects on exceptions, we introduce exception mechanisms (try-catch blocks and a throw instruction). For the sake of clarity, aspectual features are described in isolation. We believe that these descriptions provide hints useful enough to establish the semantics of complete AO language.

6.1 *Exceptions*

We introduce exceptions on the base language using the following instructions:

$$S ::= \mathsf{try}\ S_1\ \mathsf{catch}\ ex\ S_2\ \mid\ \mathsf{throw}\ ex\ \mid\ \ldots$$

The instruction try S_1 catch $ex\ S_2$ declares a new exception ex which can be thrown within S_1 and is handled by S_2. The instruction throw ex raises an exception ex.

The store remains as abstract as possible but we need to introduce a stack E recording the exceptions declared. Every element of E is a pair of type $I \times C$ where I represents an exception identifier and C a code. A pair (ex, C) of E provides the code C to execute when the exception ex is raised. These pairs are pushed in the order of the try-catch block declarations in the program. When an exception ex is thrown, the current continuation is replaced by the code associated with ex in E. If an exception cannot be found in E it is a dynamic error "*uncaught exception*".

The semantic of exceptions in the base program is described in terms of the relation $\rightarrow_b$ on configurations extended with the stack E. The execution of a try S_1 catch $ex\ S_2$ block pushes in E the pair constituted of the exception name and the code to execute in that case (*i.e.* $(ex, S_2 : C)$), execute the block S_1. The instruction pop_e removes the pair from E after completion of S_1. When an exception ex is raised, the current continuation C is replaced by the code C' associated with (the first occurrence of) ex in E. All the exceptions stacked after ex are removed from E; indeed the exception escapes from all the try catch blocks encountered between its declaration and raise.

$$\text{TRY}\ \frac{}{(\mathsf{try}\ S_1\ \mathsf{catch}\ ex\ S_2 : C,\ \Sigma,\ E)\ \rightarrow_b\ (S_1 : pop_e : C,\ \Sigma, (ex,\ S_2 : C) : E)}$$

$$\text{POP}_e\ \frac{}{(pop_e : C,\ \Sigma,\ X : E)\ \rightarrow_b\ (C,\ \Sigma,\ E)}$$

$$\text{THROW}\ \frac{}{\begin{array}{c}(\mathsf{throw}\ ex : C,\ \Sigma,\ (ex_0, C_0) : \ldots : (ex_k, C_k) : (ex, C') : E) \\ \rightarrow_b\ (C',\ \Sigma,\ E)\ with\ \mathsf{ex}_i \neq \mathsf{ex}\ \wedge\ 0 \leq i \leq k\end{array}}$$

$$\text{UNCAUGHT}\ \frac{}{\begin{array}{c}(\mathsf{throw}\ ex : C,\ \Sigma,\ (ex_0, C_0) : \ldots : (ex_k, C_k) : \varepsilon) \\ \rightarrow_b\ \textit{Uncaught exception}\ \text{with}\ \mathsf{ex}_i \neq \mathsf{ex}\ \wedge\ 0 \leq i \leq k\end{array}}$$

We now present the semantics rules of aspects and pointcuts taking exceptions into account. We consider three aspectual features inspired from AspectJ: around throws, after throwing and handler.

Around throws Aspects

The aspect around throws $P\ P_{ex}$ matches an instruction which can match P and which *can* also raise an exception matching the pattern P_{ex}. The execution is an around aspect but the definition of pointcut has to be adapted. We have to define patterns matching exceptions. For example, we can used

$$P_{ex} ::= * \mid id$$

where $*$ represents any exceptions and id is a specific exception identifier. We use the function $excep$ which takes the current instruction and returns the list of exceptions lex this join point might raise. Then, the function $match_{ex}$ returns true if it exists at least one exception in the list of exception matching the pattern of exception ($match_{ex}(lex,*) = true$). Therefore, around throws aspects are taken into account by redefining ψ to associate instructions with the list of exceptions they may raise. The aspect around throws $P\ P_{ex}$ is defined by a function ψ of the form:

$$\psi(i) = if\ match(P,i) \wedge match_{ex}(excep(i),P_{ex})\ then\ (\sigma(\phi), around)\ else\ \varepsilon$$

$$with\ \sigma\ such\ that\ \sigma(P) = i$$

After throwing Aspects

After throwing aspects apply on procedure returning by propagating an exception. We assume that calls and returns are formalized using the stack F in configurations as in section 5.1. The stack F contains the signatures corresponding to the calls which have not returned yet. First to in order to find which calls propagate an exception the current stack F must be memorized with the exception and the current continuation when entering a try - catch block. The two rules TRY and POP$_e$ are refined as follows:

$$\textsc{Try}\quad \frac{}{(\text{try } S_1 \text{ catch } ex\ S_2 : C,\ \Sigma,\ F,\ E)\ \rightarrow_b\ (S_1 : pop_e : C,\ \Sigma,\ F, (ex,\ S_2 : C, F) : E)}$$

$$\textsc{Pop}_e\ \frac{}{(pop_e : C,\ \Sigma,\ F,\ X : E)\ \rightarrow_b\ (C,\ \Sigma,\ F,\ E)}$$

When an exception is thrown, the program is replaced by the code C' associated with this exception and the exception stack is reset as before. Instead of replacing immediately F by the stack recorded with the exception, this will be done iteratively by the instruction Ret_{id}.

$$\textsc{Throw}\ \frac{}{\begin{array}{c}(\text{throw } ex : C,\ \Sigma,\ F,\ (ex_0, C_0, F_0) : \ldots : (ex_k, C_k, F_k) : (ex, C', F') : E) \\ \rightarrow_b\ (Ret_{id}\ ex\ F' : C',\ \Sigma,\ F,\ E)\ with\ ex_i \neq ex\ \wedge\ 0 \leq i \leq k\end{array}}$$

The function $Ret_{id}\ ex\ F'$ recursively pops the signatures of the stack F until it is the same as during the try-catch corresponding to the exception *ex*. Each instruction popped corresponds to a return propagating the exception therefore a candidate for inserting an afterthrowing advice. The rule $\textsc{Ret}^1_{id}$ pops the top signature of F and tries to match call id(), the call instruction denoting a return propagating the exception *ex*. In that case, the advice corresponding to the afterthrowing aspect is inserted. We consider here that if no afterthrowing aspect matches the instruction, the function ψ will return $(\emptyset,\ \mathit{afterthrowing})$. The rule $\textsc{Ret}^2_{id}$ ends this process when the F stack is back to its correct state. The execution proceeds with the exception continuation (*i.e.* the handler).

$$\textsc{Ret}^1_{id}\ \frac{(t)id : F \neq F' \ \wedge\ \psi(\mathsf{call\ id()}) =\ (\phi,\ \mathit{afterthrowing})}{(Ret_{id}\ ex\ F' : C, \Sigma, (t)id : F, E)\ \rightarrow\ (Ret_{id}\ ex\ F' : test\ \phi : C, \Sigma, F, E)}$$

$$\textsc{Ret}^2_{id}\ \frac{}{(Ret_{id}\ ex\ F : C, \Sigma, F, E)\ \rightarrow\ (C, \Sigma, F, E)}$$

EXAMPLE **6** *Consider the program Prog and the aspect ψ defined as follows:*

$$\begin{array}{lcl} Prog & = & \mathsf{try}\ call\ foo()\ \mathsf{catch}\ ex\ \varepsilon \\ void\ foo()\ ex & = & call\ goo() \\ void\ goo()\ ex & = & throw\ ex \\ \psi(*,*) & = & (\phi,\ afterthrowing) \\ \phi(\Sigma) & = & call\ baz \\ void\ baz() & = & \varepsilon \end{array}$$

The declaration of procedures is extended to include the exceptions they might throw (or propagate). Prog call the procedure foo in a try - catch *block. The procedure foo, which can propagate the exception* ex*, calls the procedure goo which raises the exception ex. The aspect ψ matches any return exiting abruptly by throwing any exception. It inserts a call to the procedure baz.*

The execution of Prog proceeds as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{(\mathsf{try}\ call\ foo()\ \mathsf{catch}\ ex\ \varepsilon : \varepsilon, \Sigma, \varepsilon, \varepsilon)} \\ \rightarrow & (call\ foo() : pop_e : \varepsilon, \Sigma, \varepsilon, (ex, \varepsilon, \varepsilon) : \varepsilon) \\ \rightarrow & (call\ goo() : \{pop_e : \varepsilon\}, \Sigma, (void)\ foo : \varepsilon, (ex, \varepsilon, \varepsilon) : \varepsilon) \\ \rightarrow & (\mathsf{throw}\ ex : \{\{pop_e : \varepsilon\}\}, \Sigma, (void)\ goo : (void)\ foo : \varepsilon, (ex, \varepsilon, \varepsilon) : \varepsilon) \\ \rightarrow & (Ret_{id}\ ex\ \varepsilon : \varepsilon, \Sigma, (void)\ goo : (void)\ foo : \varepsilon, \varepsilon) \\ \rightarrow & (Ret_{id}\ ex\ \varepsilon : test\ \phi : \varepsilon, \Sigma, (void)\ foo : \varepsilon, \varepsilon) \\ \rightarrow & (Ret_{id}\ ex\ \varepsilon : test\ \phi : test\ \phi : \varepsilon, \Sigma, \varepsilon, \varepsilon) \\ \rightarrow & (test\ \phi : test\ \phi : \varepsilon, \Sigma, \varepsilon, \varepsilon) \\ \rightarrow & (call\ baz : test\ \phi : \varepsilon, \Sigma, \varepsilon, \varepsilon) \\ \rightarrow & (\{test\ \phi : \varepsilon\}, \Sigma, (void)\ baz : \varepsilon, \varepsilon) \\ \rightarrow & (test\ \phi : \varepsilon, \Sigma, \varepsilon, \varepsilon) \\ \rightarrow & (call\ baz : \varepsilon, \Sigma, \varepsilon, \varepsilon) \\ \rightarrow & (\{\varepsilon\}, \Sigma, (void)\ baz : \varepsilon, \varepsilon) \\ \rightarrow & (\varepsilon, \Sigma, \varepsilon, \varepsilon) \end{array}$$

When the exception is thrown, the current continuation is replaced by the handler code (here ε*) and the instruction* Ret_{id} *ex. It removes iteratively the two signatures in the stack F inserting each time a call to the procedure baz.*

Handler

The pointcut handler P_{ex} matches any join point which catches an exception ex matching the pattern P_{ex}. It is supported only by aspects of kind *before*. Since the entry of the handler is not distinguished in our semantics of exceptions, we model the rule HANDLER when the exception is thrown.

$$\textsc{Handler}\quad \frac{\psi(\mathsf{throw\ ex}) = (\phi,\ \mathit{beforehandler})}{\begin{array}{c}(\mathsf{throw}\ ex : C,\ \Sigma,\ (ex_0, C_0) : \ldots : (ex_k, C_k) : (ex, C') : E) \\ \rightarrow\ (\mathit{test}\ \phi : C',\ \Sigma,\ E) \quad \mathit{with}\ ex_i \neq ex\ \wedge\ 0 \leq i \leq k\end{array}}$$

6.2 Aspect deployment

Like classes, aspects can also be instantiated dynamically. For example, an instance can be activated on entry in a block and deactivated on exit. This is a dynamic aspects deployment which is opposed to the static aspect deployment where the aspects are instantiated once and for all. We describe here the semantic of a dynamic aspect deployment similar to the feature deploy of CaesarJ.

We consider the instruction deploy id S in the base language. By this instruction, the aspect named id is activated within the block S and deactivated after the execution of S. We introduce in the configurations a stack Ψ recording all the current active aspects. The stack Ψ contains the aspects which are dynamically activated by the instruction deploy but also those which are statically instantiated. These global aspects are supposed to be at the bottom of the stack. When the instruction deploy id S is executed, the new aspect ψ_{id} is pushed on Ψ and the block S followed by the instruction pop_Ψ are executed. After the execution of S, the instruction pop_Ψ deactivates the aspect which is on the top of Ψ by removing it.

$$\textsc{Deploy}\quad \frac{}{(\mathsf{deploy\ id}\ S : C,\ \Sigma,\ \Psi)\ \rightarrow\ (S : pop_\Psi : C,\ \Sigma,\ \psi_{id} : \Psi)}$$

$$Pop_\Psi\quad \frac{}{(pop_\Psi : C,\ \Sigma,\ \psi_{id} : \Psi)\ \rightarrow\ (C,\ \Sigma,\ \Psi)}$$

During the execution, trying to match a join point i amounts to apply the stack of active aspects to i. As usual, the application of each aspect to i returns pair made of an advice and a kind. These pairs must be sorted with respect to their relative priorities. We suppose that such priorities are given by the global function *priority*. This function can be explicitly defined by the programmer using declarations such as declare precedence in AspectJ. So, matching a join point i by a stack of aspects $(\psi_1 : \ldots : \psi_n : \varepsilon)$ is described as follows:

$$\begin{array}{rcl}(\psi_1 : \ldots : \psi_n : \varepsilon)(i) & = & \mathit{priority}(\psi_1(i), \ldots, \psi_n(i)) \\ & = & ((\phi_{j_1}, t_{j_1}), \ldots, (\phi_{j_n}, t_{j_n}))\ \ \mathit{with}\ 1 \leq j_i \leq n\end{array}$$

Aspect deployment can be seen as a simple and restricted form of aspect instantiation that we consider in the following section.

6.3 Aspect Association

Aspect association designates the mechanism that associates a peculiar aspect to an instruction. In our model, this association is performed by a function ψ taking an instruction as an argument and returning a dynamic test function ϕ with a type (before, after, around), see Rule BEFORE page 5 for instance. In this section, we refine the model so that aspects can be associated to dynamic entities, along the lines of perTarget and variants in AspectJ. In order to keep the presentation simple, we consider the single aspect case. Generalization to multiple aspects is orthogonal, and the technique exposed in Section 4 can be used.

In opposition to aspects that are instantiated once and for all (also called *singleton* aspects), some aspects are meant to be associated to dynamic entities, such as objects. Thus, several instances of the same aspect may exist at run-time, for example one aspect being associated to each object instance of a given class. Each instance has its own private state, stored in Σ, that may evolve over time. Since in general the number of instances of a given class is not known statically, neither is (in general) the number of instances of a given aspect. This is why aspects have to be instantiated dynamically.

Aspect Instantiation

Since new aspect instances may be generated at run-time, the association function ψ has to evolve dynamically to take account of all aspect instances. To this end, we decompose ψ into elementary association functions ψ_{id}, where *id* is a unique aspect identifier. This leads to the definition of an *aspect environment*:

DEFINITION **7** *The* aspect environment *is a mapping* Ψ *that maps aspect identifiers (id) to elementary aspects* ψ_{id}.

At a given time, the domain of Ψ, written $\mathsf{dom}(\Psi)$, is the set of identifiers of all existing aspects. The composition of all elementary aspects ψ_{id} in Ψ is written $(\circ\Psi)$ and formally defined as $\psi_{id_1} \circ .. \circ \psi_{id_n}$ for $\mathsf{dom}(\Psi) = \{id_1, .., id_n\}$.

We modify the semantics so that the aspect environment Ψ may evolve over reductions. Thus, the general form of a reduction is now the following:

$$(C, \Sigma, \Psi) \rightarrow (C', \Sigma', \Psi')$$

Like pointcuts, aspect instantiation is governed by instructions. More precisely, each time an instruction is executed, the function Ψ is possibly updated with new aspects, thanks to a function update. We show a new version of Rule BEFORE that takes aspect instantation into account. Other rules can be updated likewise, in particular rules NOADVICE, AFTER, and AROUND.

$$\textsc{Before}\ \frac{\mathsf{update}(\Psi, i, \Sigma) = (\Psi', \Sigma') \qquad (\circ\Psi')(i) = (\phi, before)}{(i : C, \Sigma, \Psi) \rightarrow (test\ \phi : \bar{i} : C, \Sigma', \Psi')}$$

Intuitively, the function update checks if the instruction i should trigger an aspect instantiation. If this is the case, and if the aspect does not already exist, it is created. Since the state of the new aspect instance is stored in Σ, it also takes Σ as an argument and returns a new state Σ'. As a possible effect, update can also remove aspect instances from the context once their associated entity has disappeared (garbage collection).

The function update must be idempotent: if $\mathsf{update}(\Psi,i,\Sigma) = (\Psi',\Sigma')$, then $\mathsf{update}(\Psi',i,\Sigma') = (\Psi',\Sigma')$. Additionally, tagged instructions must not introduce new aspects: $\mathsf{update}(\Psi,\bar{i},\Sigma) = (\Psi,\Sigma)$.

Example (perTarget)

We illustrate aspect instantiation by associating an aspect counter to each instance of a given class Point, like perTarget in AspectJ. Each aspect counter advises the calls to a method m in Point, by counting the number of times the method is invoked. At first, we describe only the association and instantation mechanisms. Then, we provide the details of updating the value of the counter.

We have two options: either we create new instances of counter each time a new object of class Point is created, either instances of counter are created by need, that is, only when the method m is invoked on an object for the first time. We consider the second case, although the other option fits in our model as well.

New aspect instances are created using an aspect template, called a generator, and written G. In the perTarget case, a generator is a function expecting an object x and a store Σ and returning a new elementary aspect as well as a new store Σ' which holds the private state of the elementary aspect. The elementary aspect is a function ψ_{id}. By convention, in the perTarget case, the identifier id is of the form $\mathsf{aspectName}_x$, that is, the name of the aspect tagged by a reference to the object it is associated to. In short, $G(x,\Sigma)$ is a pair (ψ_{id},Σ').

To pursue the example, update is defined as follows, where $x.m()$ is the invocation of method m on object x.

$$\begin{array}{lcll}
\mathsf{update}(\Psi,\mathsf{x.m}(),\Sigma) & \overset{\Delta}{=} & (\Psi',\Sigma') & \text{if } \mathsf{counter}_x \notin \mathsf{dom}(\Psi) \\
 & & & \text{with } G(x,\Sigma) = (\psi_{id},\Sigma') \\
 & & & \text{and } \Psi' = \Psi\{id \to \psi_{id}\} \\
\mathsf{update}(\Psi,i,\Sigma) & \overset{\Delta}{=} & (\Psi,\Sigma) & \text{otherwise}
\end{array}$$

In the first case, an aspect instance is created by invoking the generator G and adding ψ_{id} to the current environment Ψ. The global state Σ is extended with private aspect state and becomes Σ'. In the second case, the instruction does not trigger aspect instantiation, and so Ψ is not modified.

The generalization to other classes and aspects is immediate.

To exemplify the introduction of new state in Σ, we now provide the details of counter. To this end, we assume that Σ is an environment associating a value (such as an integer) to identifiers (variables). We assume given a function *incr* that increments its argument, which must be a variable. Formally, the generator G associated to the aspect $\mathsf{counter}$ is defined as

$$\begin{array}{llcll}
 & G(x,\Sigma) & \stackrel{\Delta}{=} & (\psi_{id},\Sigma') & \text{with } id \stackrel{\Delta}{=} \mathsf{counter}_x \\
\text{where} & \psi_{id} & \stackrel{\Delta}{=} & \left\{ \begin{array}{lcll} x.m() & \mapsto & ((\lambda\Sigma.incr\ z), before) & \text{if } \mathsf{classOf}(x) = \mathsf{Point} \\ i & \mapsto & \emptyset & \end{array} \right. & \\
\text{and} & \Sigma' & \stackrel{\Delta}{=} & \Sigma\{z \mapsto 0\} & \text{with } z \notin \mathsf{dom}(\Sigma)
\end{array}$$

The new environment Σ' is defined as $\Sigma\{z \mapsto 0\}$, that is, the environment Σ extended with a new variable z initialized with 0 (note the side-condition $z \notin \mathsf{dom}(\Sigma)$ that avoids capture of existing variables). The association function ψ_{id} is defined as $x.m() \mapsto ((\lambda\Sigma.incr\ z), before)$.

Example of per-control-flow aspects (percflow)

Let P_e be a pointcut definition, as defined in Section 5.1. In this example, we formalize the meaning of $\mathsf{per\text{-}cflow}(P_e)$, which states that the corresponding aspect is instantiated each time a "new cflow" is considered.

Let us first describe more precisely the intuitive meaning. In Section 5.1, a stack of enclosing calls written F was introduced in the program state. Thus, each time a procedure (or function, method) call occurs, its signature $(t)id$ is pushed onto the stack. Conversely, each time a procedure calls ends, the last element of the stack is popped. The pointcut $\mathsf{cflow}(P_e)$ matches the current joinpoint if and only if a signature satisfying P_e is in the stack or if the current instruction is a call to a method whose signature matches P_e. Informally, we say that the program is in the cflow of P_e.

When the program steps from a state where it is not in the cflow of P_e into a state where it is in the cflow of P_e, we say that the program *enters* the cflow of P_e. Conversely, the program may *leave* a cflow of P_e. The qualifier $\mathsf{per\text{-}cflow}(P_e)$ states that a new aspect instance must be created each time the program enters the cflow of P_e.

Formally, we assume given an aspect generator G that takes two arguments: the state Σ and the instruction i that caused the program to step into the cflow of P_e. The generator returns an elementary aspect ψ_{id}, where id is the (unique) aspect name, and a possibly new state Σ'.

$$G(i,\Sigma) = (\psi_{id},\Sigma')$$

New aspects are created each time the program enters the cflow of P_e, which happens only when the current instruction i is a call to a method whose signature is matched by P_e while the program was not in the cflow of P_e. Conversely, aspects are deleted each time the program leaves the cflow of P_e, which occurs in Rule RET defined in Section 5.1. In order to take the call stack into account, the function update, defined in the previous section, gets F as an extra argument.

The function update that implements $\mathsf{per\text{-}cflow}(P_e)$ is defined as follows (the name id is the unique name of the aspect being considered):

$$\begin{array}{lcl}
\mathsf{update}(\Psi,i,\Sigma,F) & \triangleq & (\Psi,\Sigma') \\
 & & \quad \text{if } match_f(P_e,i,F) \text{ and } id \notin \mathsf{dom}(\Psi) \\
 & & \quad \text{with } G(i,\Sigma) = (\psi_{id},\Sigma') \\
 & & \quad \text{and } \Psi' = \Psi\{id \to \psi_{id}\} \\
\mathsf{update}(\Psi,\{C\},\Sigma,F) & \triangleq & (\Psi - id,\Sigma) \\
 & & \quad \text{if not } match_f(P_e,\{C\},F) \\
\mathsf{update}(\Psi,i,\Sigma,F) & \triangleq & (\Psi,\Sigma) \\
 & & \quad \text{if not } match_f(P_e,i,F) \text{ or } id \in \mathsf{dom}(\Psi)
\end{array}$$

The first case correspond to a program entering the cflow of P_e. The function $match_f$ is formally defined page 11, in the definition of cflow. The second case captures the return instruction (Rule RET). It removes the aspect instance id from the aspect environment Ψ if the reduction steps out of the cflow of P_e.

Note that it is also acceptable to consider per cflow for individual threads. To this end, it suffices to mark the identifier of the aspect (id in ψ_{id}) with the thread identifier. Thus, a new aspect instance will be created each time a thread enters the cflow of P_e. This requires that the current thread id is made explicit in the reduction rules.

6.4 Stateful Aspects

An aspect inserts an advice when it matches an instruction. A stateful aspect inserts an advice when it matches a *sequence* of instructions. So, it has a state that evolves and specifies the next instruction to be matched. Our semantics rules must take into account the evolutions of ψ as the weaving progresses. For instance, the rule BEFORE becomes:

$$\textsc{Before}\quad \frac{\psi(i) = (\phi, before, \psi')}{(i : C, \Sigma, \psi) \to (test\ \phi : \bar{i} : C, \Sigma, \psi')}$$

We introduce a grammar for stateful aspects:

$$\begin{array}{lcl}
A & ::= & \mu a.(P_1 \triangleright a_1;A_1 \square \ldots \square P_n \triangleright a_n;A_n) \\
 & | & a
\end{array}$$

The base case $P \triangleright a;A$ inserts the advice a when the pattern P matches the current instruction, then it weaves A. The choice operator $\square$ defines branches in sequences of instructions. Finally, the recursion operator enables sequences of arbitrary length. Such an aspect definition can be translated into a function ψ as follows:

$$\begin{array}{l}
T_1 :: A \to \psi \\
T_1[\![\mu a.(P_1 \triangleright a_1;A_1 \square \ldots \square P_n \triangleright a_n;A_n)]\!] = \mu a.\lambda i. \\
\quad \textit{if } match(P_1,i) \textit{ then } (\lambda\Sigma.a_1, before, T_1[\![A_1]\!]) \textit{ else} \\
\quad \ldots \\
\quad \textit{if } match(P_n,i) \textit{ then } (\lambda\Sigma.a_n, before, T_1[\![A_n]\!]) \textit{ else} \\
\quad \text{a} \\
T_1[\![a]\!] = a
\end{array}$$

The recursion is translated directly. A function ψ takes the current instruction in parameter (*i.e.* $\lambda i.$). Then it performs pattern matching and returns the corresponding advice (or check the next instruction if no pattern matches). The function returns a triplet $(\phi, \mathit{before}, \psi')$ where ψ' is a kind of continuation for the aspect (*i.e.* the function to be applied to the next instruction).

7 Conclusion

In this note, we have defined a small step semantics for the major mechanisms of AspectJ, EAOP and Caesar. They have been defined in isolation. So, the next step could be to mix them. It is quite likely these mechanisms are not fully orthogonal. For instance, both aspect association and stateful aspects update the function ψ. We also plan to use our semantics in order to explore the analysis and/or verification of aspect oriented programs. In particular, we would like to verify that a specific aspect ensures some properties in the woven program or preserves some properties of the base program.