CASB: Common Aspect Semantics Base

ABSTRACT

This document gradually introduces formal semantic descriptions of aspect mechanisms.

Document Id	:	AOSD-Europe-INRIA-7
Deliverable No.	:	54
Work-package No.	:	8
Type	:	Integration
Status	:	Final
Version	:	1.0
Date	:	11 August 2006
Author(s)	:	Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, Didier Le Botlan
Contributor(s)	:	Jacques Noyé, Tom Staijen

Contents

1 Common Aspect Semantics Base

In a previous Milestone [DB05] we have reviewed the different formal semantics for AOP. All of them provide a semantics as a whole but do not isolate the different features of aspect languages. This document gradually introduces formal semantic descriptions of aspect mechanisms.

Most of previous semantics consider object oriented base programs [JJR03a, JJR03b, JJR05] [Läm02] [DT04] [WKD04] [CLW03]. Some other work also consider functional languages (call-by-value λ-calculus, ML, Scheme, ...) [WZL03, DWWW], as well as process calculi[BJJR04]. In this deliverable, 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 require any special mechanism: 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).

Most of related work are restricted to a subset of AspectJ's semantics. In our deliverable, 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, ...). As an illustration of our technique, we describe the semantics of an AspectJ-like core aspect language (around aspects + cflow + aspect association/instantiation) for a core Java language (Featherweight Java with assignments).

Most formal work are described in term of small step semantics (SOS). A few express semantics differently: big step (a.k.a. natural) [Läm02], denotational [WKD04], and finite state automata (and sos) [CF00]. We also use a small step semantics here because it precisely models the implementation and can be seen as an abstract machine or a compiler.

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:

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

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*

$$S ::= S_1; S_2 \mid f = S \mid call\ f$$

can be described in this setting as:

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

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*

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

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

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

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 pair of a function ϕ and a type *t* $(= before, after, around, \ldots)$ that denotes the kind of aspect. The function ϕ takes the state Σ as parameter and returns an advice *a* (supposed to be written in the same language as the base program) to be inserted

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

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 ε when the aspect does not match the current instruction. When there is no advice for the current state, the function ϕ returns ε if the kind of the aspect is *before* or *after* and returns *proceed* if the kind of the aspect is *around*.

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.

$$\textsc{NoAdvice}\quad \frac{\psi(i) = \varepsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma) \rightarrow (C', \Sigma')}$$

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

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

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

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.

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

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 (or the empty advice ε).

$$\textsc{Advice}\quad \frac{}{(\mathit{test}\ \phi : C, \Sigma) \rightarrow (\phi(\Sigma) : C, \Sigma)}$$

Note that the instructions of the advice are subject to be matched by an 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 $(\bar{i} : test\ \phi : C, \Sigma)$. The instruction i cannot be matched again and the next reduction step of the configuration will be done using $\rightarrow_b$.

$$\textsc{After}\quad \frac{\psi(i) = (\phi, after)}{(i : C, \Sigma) \rightarrow (\bar{i} : test\ \phi : C, \Sigma)}$$

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. As indicated before, if the dynamic part of the pointcut does not match the current state, the aspect is not applied and the current instruction is executed. This is formalized by the fact that, when it does not match the dynamic state, the function ϕ of an around aspect returns proceed (and not ε). This behavior (taken by AspectJ) implies that the dynamic checks in the pointcut (*e.g.* if $x = 0$) are not interpreted as a conditional expression in the advice code (*e.g.* if $x = 0$ then *advice* else skip). Another option (maybe less reasonable) could have been to interpret dynamic checks in the pointcut as a conditional expression in the advice. This would ensure the equivalence of identical advices with identical tests regardless of their location (pointcut or advice). To implement this option, the function ϕ should return ε when it does not match the current state.

In general, 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 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:*

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

Note that our rules for around aspects can be applied to any instructions, not only to calls. We also assumed also that the advice of an around aspect could be matched by another around aspect and that we could have imbricated proceeds. AspectJ prevents this case by syntactic restrictions. So our around aspects are more general than AspectJ's. To describe precisely AspectJ, we should restrict our rules to method calls. Assuming that proceeds cannot be imbricated would also permit to simplify the rule PROCEED (the instruction does not have to be removed and pushed at each proceed).

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.

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

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.

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

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 $a'_i : \mathsf{proceed} : a''_i$ 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 $a'_1 : \mathsf{proceed} : a''_1 : \mathsf{proceed} : a'''_1$ 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 $around$. By default, in AspectJ for example, this tuple is sorted in the order $before, after$ and $around$ of t_i. But the programmer can modify it by the use of declare precedence.

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.* $test\ \phi$) 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, $test\ \phi$ at the beginning of the translated *before* aspects inspects the current state (see the rule ADVICE). In the translated *after* aspect, $test\ \phi$ 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.

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

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 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.

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

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

The pointcut *cflow*(B) selects all the join points which are in the control flow of the pointcut B. Thus, *cflow*(B) matches a signature of an instruction, if this instruction is in the control flow of B. Every signature is a pair (id, t). If the instruction is a procedure call, *id* is the name of the procedure and t its return type. For other instructions, *id* is the instruction and the type void will be returned. 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, a signature and a stack of signatures.

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

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

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

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

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

In this grammar, $(P_T)P_I$ is a pattern matching any signature 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:

$$\begin{array}{ll}
match_f(cflow(P), i, \varepsilon) & = False \\
match_f((P_T)P_I, i, F) & = match(P_T, t) \\
 & \quad \wedge match(P_I, id) \;\; if \; i = (id, t) \\
match_f(cflow(P), i, i' : F') & = match_f(P, i, i' : F') \\
 & \quad \vee match_f(cflow(P), i', F') \\
 & \\
match_f(cflowbelow(P), i, \varepsilon) & = False \\
match_f(cflowbelow(P), i, i' : F') & = match_f(cflow(P), i', F') \\
 & \\
match_f(P_1 \wedge P_2, i, F) & = match_f(P_1, i, F) \wedge match_f(P_2, i, F) \\
match_f(P_1 \vee P_2, i, F) & = match_f(P_1, i, F) \vee match_f(P_2, i, F) \\
match_f(\neg P, i, F) & = \neg match_f(P, i, F)
\end{array}$$

A join point is in the control flow of pointcut P if P matches either the signature of the current instruction or a procedure call which precedes this join point is in the control flow of P. A join point is below the control flow of a pointcut P is that a procedure call which precedes this join point is in the control flow of P.

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:

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

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.

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

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

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

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

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:

TRY

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

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

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}.

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

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 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 $(\varepsilon,\ afterthrowing)$. The rule 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).

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

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

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

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

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:

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

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.

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

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.

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

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

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:

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

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.

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

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 counter is defined as

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

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 per-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 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 per-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 per-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 12, 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:

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

We introduce a grammar for stateful aspects:

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

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 $\Box$ 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:

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

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 $(\phi, before, \psi')$ where ψ' is a kind of continuation for the aspect (*i.e.* the function to be applied to the next instruction).

7 Aspects for Java

In this section, we define the semantics of a core of Java and AspectJ. First, we give a semantics for a subset of Java, basically Featherweight Java with assignments (AFJ). Then, we define the semantics of an AspectJ-like language with around aspects, control flow pointcuts (cflow) as well as associations (pertarget, percflow) for the programming language AFJ.

7.1 Assignment Featherweight Java

Assignment Featherweight Java (AFJ) [MP05] is Featherweight Java (FJ), a purely functional subset of Java, extended with field assignments. The syntax of Featherweight Java is extended with a new kind of expression, $e.f = e$, representing assignments. Compared to AJF [MP05], our version does not have the syntactic category of constructor nor the return instruction which are useless for our purpose.

$$\begin{array}{lll} \mathit{Prog} & ::= & \overline{L};\ e \\ L & ::= & \mathsf{class}\ X\ \mathsf{extends}\ X\ \{\overline{X}\,\overline{f}\,\overline{M}\} \\ M & ::= & Xm(\overline{X}\overline{x})\{e\} \\ e & ::= & x \mid e.f \mid e.m(\overline{e}) \mid \mathsf{new}\ X(\overline{e}) \mid (X)e \mid e.f = e \end{array}$$

A program is a sequence of classes ($\overline{L}$) followed by a main expression (e). The identifier X represents a class identifier. Every class has a sequence of fields associated with a type which is a class identifier ($\overline{X}\,\overline{f}$). A class also defines methods ($\overline{M}$). A method takes a sequence of objects as parameters ($\overline{X}\overline{x}$) and returns an object which is the result of the computation of the method body (e). An expression e can be a variable (x), an access to a field ($e.f$), a call to a method with a sequence of expression as parameters ($e.m(\overline{e})$), a construction of a class with a sequence of expressions as parameter ($\mathsf{new}\ X(\overline{e})$), a cast of an expression ($(X)e$) or an assignment ($e.f = e$). The evaluation of an assignment $e_1.f = e_2$ assigns to the field of the object obtained by evaluating e_1 the object obtained by evaluating e_2. Apart from that side effect, the result of the assignment $e_1.f = e_2$ is the result of the evaluation of e_2.

Expressions do not include sequences. A Java program with a sequence of Java statements can be transformed into a AFJ program with a longer list of method arguments. For instance, the following Java program

```
class Moo {
  Object o;

  Moo(Object o) {
    this.o = o;
  }
  Moo foo(Moo m) {
    this.o = m;
    return this.bar().bar();
  }
  Moo bar() {
    return (Moo)(this.o);
  }
  public static void main(String[] args) {
    new Moo(new Object()).foo(new Moo(new Moo(new Object()))).bar();
  }
}
```

can be transformed into the following AFJ program

```
class Moo extends Object {
  Object o;

  Moo foo(Moo m) {
    this.foo1(this.o=m,m)
  }
  Moo foo1(Object o1, Moo m) {
     this.bar().bar()
  }
  Moo bar() {
     (Moo)(this.o)
  }
}
new Moo(new Object()).foo(new Moo(new Moo(new Object()))).bar()
```

The semantics of AFJ updates a store component in order to take into account assignment. The original semantics [MP05] relies on congruence rules to define redexes. We slightly modify the semantics and replace the congruence rules by rules that sequentialise the computation using the continuation component C. The semantics of AFJ is defined as follows. First, we define some auxilary functions.

$$
\begin{array}{lcl}
\Sigma & : & \mathit{Object} \rightarrow X \times \mathit{Fd} \\
\mathit{Fd} & : & \mathit{Identifier} \rightarrow \mathit{Object} \\
\mathit{mbody} & : & \mathit{Identifier} \times X \rightarrow e \\
\mathit{FieldName} & : & X \rightarrow \overline{\mathit{Identifier} \times \mathit{Identifier}} \\
\mathit{init} & : & \overline{L} \times \mathit{mbody} \times \mathit{FieldName} \rightarrow \mathit{mbody} \times \mathit{FieldName}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{init}(\mathsf{class}\ X\ \mathsf{extends}\ B\ \{\overline{T}\,\overline{f}\,\overline{M}\}\ \overline{L'},\ \mathit{mbody},\ \mathit{FieldName}) & & \\
\qquad = & \mathit{init}(\overline{L'},\ \mathit{mbody}[(m_0,\ X) \mapsto e_0,\ \ldots,\ (m_n,\ X) \mapsto e_n], & \\
& \mathit{FieldName}[X \mapsto \overline{(T,\ f)} \cup \mathit{FieldName}(B)]) & \\
\mathit{init}(\varepsilon,\ \mathit{mbody},\ \mathit{FieldName}) & = & (\mathit{mbody},\ \mathit{FieldName})
\end{array}
$$

The function Σ represents the store (*i.e.* the heap, or memory). It takes an Object (i.e., a reference) as a parameter and returns an instance and its type. The function Fd takes a field identifier as a parameter and returns its current value (i.e., a reference). The function mbody takes the signature of a method and returns the list of its parameters and its body. The function $\mathit{FieldName}$ returns the list of fields and their types for a given class. The function init builds the initial environments mbody and $\mathit{FieldName}$ for a given program. The initial call is of the form $\mathit{init}(\overline{L}, \bot, \bot)$ with $\overline{L}$ the list of classes of the program.

The semantics of a program is given by a transition system and summarized by the following equation:

$[\![\overline{L};\ e]\!] = (e : \varepsilon,\ \varepsilon,\ \bot,\ \varepsilon) \rightarrow_b^* (\varepsilon,\ v : \varepsilon,\ \Sigma, \varepsilon)$ *with* $init(\overline{L},\ \bot,\ \bot) = (mbody,\ FieldName)$

The transition system itself is defined by the following inference rules:

$$CAST1\ \frac{}{((X)e : C,\ S,\ \Sigma,\ F)\ \rightarrow_b\ (e : CAST_X : C,\ S,\ \Sigma,\ F)}$$

$$CAST2\ \frac{\Sigma(v) = (X,\ Fd)\ \ X <: D}{(CAST_D : C,\ v : S,\ \Sigma,\ F)\ \rightarrow_b\ (C,\ v : S,\ \Sigma,\ F)}$$

$$GET1\ \frac{}{(e.f_i : C,\ S,\ \Sigma,\ F)\ \rightarrow_b\ (e : get\ f_i : C,\ S,\ \Sigma,\ F)}$$

$$GET2\ \frac{\Sigma(v) = (X,\ Fd)\ \ Fd(f_i) = v_2}{(get\ f_i : C,\ v : S,\ \Sigma,\ F)\ \rightarrow_b\ (C,\ v_2 : S,\ \Sigma,\ F)}$$

$$SET1\ \frac{}{(e_0.f_i = e : C,\ S,\ \Sigma,\ F)\ \rightarrow_b\ (e : e_0 : set\ f_i : C,\ S,\ \Sigma,\ F)}$$

$$SET2\ \frac{\Sigma(v_0) = (X,\ Fd)}{(set\ f_i : C,\ v_0 : v : S,\ \Sigma,\ F) \quad \rightarrow_b\ (C,\ v : S,\ \Sigma[v_0 \mapsto (X,\ Fd[f_i \mapsto v])],\ F)}$$

$$CALL1\ \frac{}{(e_0.m(e_1,\ \ldots,\ e_n) : C,\ S,\ \Sigma,\ F) \quad \rightarrow_b\ (e_1 : \ldots : e_n : e_0 : call\ m^n : C,\ S,\ \Sigma,\ F)}$$

$$CALL2\ \frac{\Sigma(v_0) = (X,\ Fd)\ \ mbody(m,\ X) = (x_1,\ \ldots,\ x_n).e}{\begin{array}{l}(call\ m^n : C,\ v_0 : v_1 : \ldots : v_n : S,\ \Sigma,\ F) \\ \rightarrow_b\ (e[x_1/v_1,\ \ldots,\ x_n/v_n,\ this/v_0] : \{C\},\ S,\ \Sigma, ((X)m, v_0) : F)\end{array}}$$

$$NEW1\ \frac{}{(\mathsf{new}\ X(e_1,\ \ldots,\ e_n) : C,\ S,\ \Sigma,\ F) \quad \rightarrow_b\ (e_1 : \ldots : e_n : New_X^n : C,\ S,\ \Sigma,\ F)}$$

$$NEW2\ \frac{v \notin dom(\Sigma)\ \ FieldName(X) = (T_1,\ f_1), \ldots, (T_n,\ f_n)\ \ Fd = [f_1,\ \ldots,\ f_n \mapsto v_1,\ \ldots,\ v_n]}{(New_X^n : C,\ v_1 : \ldots : v_n : S,\ \Sigma,\ F) \quad \rightarrow_b\ (v : C,\ S,\ \Sigma[v \mapsto (X,\ Fd)],\ F)}$$

$$RET\ \frac{}{(\{C\},\ S,\ \Sigma,\ ((X)m, v_0) : F) \quad \rightarrow_b\ (C,\ S,\ \Sigma,\ F)}$$

$$PUSHOBJ\ \frac{}{(v:C,\ S,\ \Sigma,\ F)\ \rightarrow_b\ (C,\ v:S,\ \Sigma,\ F)}$$

In all rules, S represents the evaluation stack which contains the result of the evaluation of an expression. The stack F contains the signatures of the methods calls which have not returned yet (like in the rules CALL and RET of page 11). Here, every element of F is not only the signature but a pair $((t)id, r)$ which contains also the receiver r needed to bind variables (needed by the AspectJ-like pointcut target).

Most semantics rules comes as a pair. Rules indexed by 1 sequentialize the computation by decomposing the current expression into a sequence of continuation expressions. Rules indexed by 2 perform the actual computation. For instance,

- The rule CAST1 builds a continuation that first evaluates the expression e, then performs the cast. The rule CAST2 actually performs the cast by verifying subtyping constraints on the dynamic type of the value v.
- The rules GET evaluate the receiver and access one of its fields.
- The rules SET evaluate the right hand side, the left hand side, and performs an assignment.
- The rules CALL evaluate the arguments of a method from left to right, then the receiver. The call itself pushed te signature and receiver in F, substitutes the parameters and evaluates the body.
- The rules NEW evaluate the arguments of a constructor from left to right and build an instance using a fresh reference.
- The single rule RET which represents the return instruction (end of a block) pops a pair from F. The rule PUSHOBJ pops a reference from the continuation stack and pushes it on the value stack S.

EXAMPLE **8** *In this example, we consider the previous AFJ program (page 22) that we call Prog. For the clearness, the stack of more than one elements $x_1,\ldots,x_n$ is noted $x_1:\ldots:x_n$, of one element x is $x:\varepsilon$ and the empty stack is ε.*

The execution of Prog according to our semantic rules begins by the execution of init which initializes the program environments:

$$init(\textsf{class}\ Moo\ \textsf{extends}\ Object\ \{\ldots\}\varepsilon,\ \bot,\ \bot)$$
$$= (\varepsilon,\ mbody[(foo,Moo) \mapsto this.foo1(this.o = m, m), (foo1, Moo) \mapsto this.bar().bar(),$$
$$(bar,Moo) \mapsto (Moo)(this.o)],\ FieldName[Moo \mapsto (Object, o)])$$

$$= (mbody[(foo,Moo) \mapsto this.foo1(this.o = m, m), (foo1, Moo) \mapsto this.bar().bar(),$$
$$(bar,Moo) \mapsto (Moo)(this.o)],\ FieldName[Moo \mapsto (Object, o)])$$

Then

$[\![Prog]\!]$
$= (new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar() : \varepsilon,\ \varepsilon,\ \bot,\ \varepsilon)$
$\rightarrow_b (new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object())))) : call\ bar^0,\ \varepsilon,\ \bot,\ \varepsilon)$
$\rightarrow_b (new\ Moo(new\ Moo(new\ Object()))) : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \bot,\ \varepsilon)$
$\rightarrow_b (new\ Moo(new\ Object()) : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \bot,\ \varepsilon)$
$\rightarrow_b (new\ Object() : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \bot,\ \varepsilon)$
$\rightarrow_b (New^0_{Object} : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \bot,\ \varepsilon)$

$\rightarrow_b (v_0 : New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \Sigma[v_0 \mapsto (Object,\ \bot)],\ \varepsilon)$
$\rightarrow_b (New^1_{Moo} : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ v_0 : \varepsilon,\ \Sigma[v_0 \mapsto (Object,\ \bot)],\ \varepsilon)$
$\rightarrow_b (v_1 : New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \Sigma[v_1 \mapsto (Moo,\ o \mapsto v_0)],\ \varepsilon)$
$\rightarrow_b (New^1_{Moo} : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ v_1 : \varepsilon,\ \Sigma[v_1 \mapsto (Moo,\ o \mapsto v_0)],\ \varepsilon)$
$\rightarrow_b (v_2 : new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ \varepsilon,\ \Sigma[v_2 \mapsto (Moo,\ o \mapsto v_1)],\ \varepsilon)$

$\rightarrow_b (new\ Moo(new\ Object()) : call\ foo^1 : call\ bar^0,\ v_2 : \varepsilon,\ \Sigma[v_2 \mapsto (Moo,\ o \mapsto v_1)],\ \varepsilon)$
$\rightarrow_b (new\ Object() : New^1_{Moo} : call\ foo^1 : call\ bar^0,\ v_2 : \varepsilon,\ \Sigma[v_2 \mapsto (Moo,\ o \mapsto v_1)],\ \varepsilon)$
$\rightarrow_b (New^0_{Object} : New^1_{Moo} : call\ foo^1 : call\ bar^0,\ v_2 : \varepsilon,\ \Sigma[v_2 \mapsto (Moo,\ o \mapsto v_1)],\ \varepsilon)$
$\rightarrow_b (v_3 : New^1_{Moo} : call\ foo^1 : call\ bar^0,\ v_2 : \varepsilon,\ \Sigma[v_3 \mapsto (Object,\ \bot],\ \varepsilon)$
$\rightarrow_b (New^1_{Moo} : call\ foo^1 : call\ bar^0,\ v_3 : v_2,\ \Sigma[v_3 \mapsto (Object,\ \bot],\ \varepsilon)$
$\rightarrow_b (v_4 : call\ foo1 : call\ bar^0,\ v_2 : \varepsilon,\ \Sigma[v_4 \mapsto (Moo,\ o \mapsto v_3],\ \varepsilon)$

$\rightarrow_b (call\ foo^1 : call\ bar^0,\ v_4 : v_2,\ \Sigma,\ \varepsilon)$
$wiht\ \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_3)]$
$\rightarrow_b (v_4.foo1(v_4.o\ =\ v_2,\ v_2) : \{call\ bar^0\},\ \varepsilon,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (v_4.o\ =\ v_2 : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\},\ \varepsilon,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (v_2 : v_4 : set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\},\ \varepsilon,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (v_4 : set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\},\ v_2 : \varepsilon,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (set\ o : v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\},\ v_4 : v_2,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (v_2 : v_4 : call\ foo1^2 : \{call\ bar^0\},\ v_2 : \varepsilon,\ \Sigma[v_4 \mapsto (Moo,\ o \mapsto v_2],\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (v_4 : call\ foo1^2 : \{call\ bar^0\},\ v_2 : v_2,\ \Sigma[v_4 \mapsto (Moo,\ o \mapsto v_2],\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b (call\ foo1^2 : \{call\ bar^0\},\ v_4 : v_2 : v_2,\ \Sigma[v_4 \mapsto (Moo,\ o \mapsto v_2],\ ((Moo)foo, v_4) : \varepsilon)$

$\rightarrow_b (v_4.bar().bar() : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ ((Moo)foo1, v_4) : F)$
$wiht\ \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$
$F = ((Moo)foo, v_4) : \varepsilon$
$\rightarrow_b (v_4.bar() : call\ bar^0 : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ ((Moo)foo1, v_4) : F)$
$\rightarrow_b (v_4 : call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ ((Moo)foo1, v_4) : F)$

$\rightarrow_b$ $(call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ ((Moo)foo1, v_4) : F)$
$\rightarrow_b$ $((Moo)v_4.o : \{call\ bar^0 : \{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)$
$\quad$ $with\ F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow_b$ $(v_4.o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)$
$\rightarrow_b$ $(v_4 : get\ o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)$
$\rightarrow_b$ $(get\ o : CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\},\ v_4 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)$
$\rightarrow_b$ $(CAST_{Moo} : \{call\ bar^0 : \{\{call\ bar^0\}\}\},\ v_2 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)\ \ with\ \Sigma(v_4) = (\_,\ o \mapsto v_2)$

$\rightarrow_b$ $(\{call\ bar^0 : \{\{call\ bar^0 :\}\}\},\ v_2 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_4) : F)\ with\ \Sigma(v_2) = (Moo,\ \_) \wedge Moo <: Moo$
$\rightarrow_b$ $(call\ bar^0 : \{\{call\ bar^0\}\},\ v_2 : \varepsilon,\ \Sigma,\ F)$
$\rightarrow_b$ $((Moo)v_2.o : \{\{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)$
$\rightarrow_b$ $(v_2.o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)$
$\rightarrow_b$ $(v_2 : get\ o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)$
$\rightarrow_b$ $(get\ o : CAST_{Moo} : \{\{\{call\ bar^0\}\}\},\ v_2 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)$
$\rightarrow_b$ $(CAST_{Moo} : \{\{\{call\ bar^0\}\}\},\ v_1 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)\ with\ \Sigma(v_2) = (\_,\ o \mapsto v_1)$

$\rightarrow_b$ $(\{\{\{call\ bar^0 : \varepsilon\}\}\},\ v_1 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_2) : F)\ with\ \Sigma(v_1) = (Moo,\ \_)\ \wedge\ Moo <: Moo$
$\rightarrow_b$ $(\{\{call\ bar^0 : \varepsilon\}\},\ v_1 : \varepsilon,\ \Sigma,\ F)$
$\quad$ $with\ F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow_b$ $(\{call\ bar^0 : \varepsilon\},\ v_1 : \varepsilon,\ \Sigma,\ ((Moo)foo, v_4) : \varepsilon)$
$\rightarrow_b$ $(call\ bar^0 : \varepsilon,\ v_1 : \varepsilon,\ \Sigma,\ \varepsilon)$
$\rightarrow_b$ $((Moo)v_1.o : \{\varepsilon\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)$
$\rightarrow_b$ $(v_1.o : CAST_{Moo} : \{\varepsilon\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)$
$\rightarrow_b$ $(v_1 : get\ o : CAST_{Moo} : \{\varepsilon\},\ \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)$
$\rightarrow_b$ $(get\ o : CAST_{Moo} : \{\varepsilon\},\ v_1 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)$
$\rightarrow_b$ $(CAST_{Moo} : \{\varepsilon\},\ v_0 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)\ with\ \Sigma(v_1) = (\_,\ o \mapsto v_0)$
$\rightarrow_b$ $(\{\varepsilon\},\ v_0 : \varepsilon,\ \Sigma,\ ((Moo)bar, v_1) : \varepsilon)\ with\ \Sigma(v_0) = (Object,\ \_) \wedge Object <: Moo$
$\rightarrow_b$ $(\varepsilon,\ v_0 : \varepsilon,\ \Sigma,\ \varepsilon)$
$wiht\ \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$

7.2 Featherweight AspectJ

The grammar below represents a subset of AspectJ. In this grammar, an aspect is represented by a set of fields, a pointcut P, an advice Ad and its identifier Y. The advice, of the around kind, takes a sequence of objects as parameters and returns an object. An advice body is an expression e like in AFJ in which proceed can be a subexpression. The pointcut P is a pattern which can represent either

- a call to a method $(P_T)P_I$ where P_T or P_I are either an identifier (class or method) or a wildcard $*$,

- the reference of the receiver $target(x)$ where x is a variable which will be bound to the receiver,
- the *cflow* pointcut,
- a disjunction, conjunction or negation of these patterns.

An aspect can optionally be either a percflow or a pertarget aspect. In the case of a percflow, an instance of the aspect is created each time the program enters the cflow of $(P_T)P_I$ and for a pertarget, an instance of the aspect is created each time $(P_T)P_I$ is accessed by a new object.

$$
\begin{array}{lcl}
A & ::= & \mathsf{aspect}\ [percflow((P_T)P_I) \mid pertarget((P_T)P_I)]\ Y\ \{\bar{X}\,\bar{f}\,P\,Ad\} \mid \\
Ad & ::= & X\ \mathsf{around}(\bar{X}\bar{x})\{e\} \\
P & ::= & (P_T)P_I \mid target(x) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P \mid cflow(P) \\
P_T & ::= & id \mid * \\
P_I & ::= & id \mid *
\end{array}
$$

In the previous sections, we used Ψ as aspect environment. In definition 7 (page 17), Ψ maps aspect identifiers to elementary aspects ψ_{id} and $(\circ\Psi)$ represents the composition of all elementary aspects in Ψ.

Here we define the $(\circ\Psi)$ function returning advice of matching aspects. We suppose to have the following functions:

$$
\begin{array}{l}
Fields:\ id \rightarrow \overline{id \times id} \\
Advice:\ id \rightarrow e \\
Pointcut:\ id \rightarrow P
\end{array}
$$

returning for each aspect identifier its corresponding fields, advice and pointcut respectively. The function $(\circ\Psi)$ takes the current instruction but also the top of the evaluation stack (for target) and the control flow stack (for *cflow*).

$$
\begin{array}{rcl}
(\circ\Psi)(i,t,F) & = & \psi_{id_1}(i,t,F) \circ \ldots \circ \psi_{id_n}(i,t,F) \\
 & & \text{for } \mathsf{dom}(\Psi) = \{id_1,\ldots,id_n\} \\
 & = & (\psi_{id_1}(i,t,F),\ldots,\psi_{id_n}(i,t,F)) \\
 & = & ((\sigma(\phi_{id_{j_1}}),around),\ldots,(\sigma(\phi_{id_{j_n}}),around)) \\
 & & with\ 1 \leq j_i \leq n \\
\text{Because for each } id \in dom(\Psi) & & \\
\psi_{id}(i,t,F) & = & if\ match_G(Pointcut(id),i,F,t) = \sigma \\
 & & then\ (\sigma(\phi_{id}),around)\ else\ \varepsilon \\
 & & \\
 & & with\ \phi_{id} = Advice(id)
\end{array}
$$

The function $(\circ\Psi)$ evaluates the function ψ corresponding to each elementary aspect and returns a n-uplet. Each ψ_{id} function returns the pair $(\sigma(\phi(id)),around)$ if the pointcut in the current aspect identifier *id* match the current instruction else it returns ε. In this pair, σ is a mapping (substitution) that maps each variable in a pointcut to an object. Here it binds the

variable in target to the correspondent object which could be the top of the evaluation stack or a receiver in the control flow stack. Because we do not have a dynamic information in a pointcut, ϕ_{id} does not take Σ as parameter like in Section 3 and returns the advice of the aspect id. Therefore $\sigma(\phi_{id})$ returns the advice in which the variables in the pointcut used in this advice body are substituted by the correspondent values in σ. The substitution σ is returned by the function $match_G$ which represents the process of matching a pointcut P to an instruction (but also according to the control stack and the object on the top of the evaluation stack). The matching process has the following signature

$$match_G : P \times Instruction \times F \times Object \rightarrow \sigma \cup \{Fail\}$$

where $Object$ is the set of objects, $Instruction$ the set of instructions and F is the control flow stack. The function $match_G$ returns a substitution σ if the pointcut matches and returns $Fail$ otherwise. $match_G$ is defined as follows:

$$\begin{array}{lcl}
match_G(target(x),\ i,\ F,\ t) & = & \{x \rightarrow t\} \\
match_G((P_T)P_I,\ i,\ F,\ t) & = & if\ i = (T) \wedge (P_T = T \vee P_T = *) \wedge (P_I = I \vee P_I = *) \\
 & & then\ \emptyset \\
 & & else\ Fail \\
match_G(P_1 \wedge P_2,\ i,\ F,\ t) & = & if\ match_G(P_1,\ i,\ F,\ t)\ =\ \sigma_1 \\
 & & \wedge\ match_G(P_2,\ i,\ F,\ t)\ =\ \sigma_2 \\
 & & then\ \sigma_1 \cup \sigma_2 \\
 & & else\ Fail \\
match_G(\neg P,\ i,\ F,\ t) & = & if\ match_G(P,\ i,\ F,\ t)\ =\ \emptyset\ then\ \emptyset \\
 & & else\ Fail \\
match_G(P_1 \vee P_2,\ i,\ F,\ t) & = & if\ match_G(P_1,\ i,\ F,\ t)\ =\ \sigma\ then\ \sigma \\
 & & else\ \ match_G(P_2,\ i,\ F,\ t) \\
match_G(cflow(P),\ i,\ (i',r) : F',\ t) & = & if\ match_G(P,\ i,F,\ t)\ =\ \sigma\ then\ \sigma \\
 & & else\ if F = (i',r) : F'\ then\ \ match_G(cflow(P),\ i',\ F',\ r) \\
 & & else\ Fail
\end{array}$$

Matching the pointcut $target(x)$ against an instruction consists in binding x with the object passed to $match_G$ (*i.e.* the receiver linked to this instruction). Matching $(P_T)P_I$ returns the empty substitution ($\emptyset$) in case of success (the pointcut does not contain variables) or $Fail$. For the pointcut $cflow(P)$, we first try to match P against the current instruction, if this process failed then we try to match against the first method call in the call stack F and its corresponding receiver (r).

In the following subsections and sections, we will consider only the aspects which respect the above grammar.

7.3 Around aspects

Like in AspectJ, in featherweight AspectJ around aspects are applied only at method calls. Like before, the rule NOADVICE below executes the current instruction i if no advice is to be executed. Note that, the component Σ is refined and an evaluation stack S appears. The proceed stack P is used only by the woven program ($\rightarrow$). Additionally, we call the function update to check if new aspects should be created at run-time(see subsection 7.5 for the definition of update).

$$\textsc{NoAdvice}\ \frac{\begin{array}{c}\mathsf{update}(\Psi,i,t:S,\Sigma)=(\Psi',\Sigma') \\ (\circ\Psi')(i,t,F)=\varepsilon \qquad (i:C,t:S,\Sigma',F)\rightarrow_b (C',S',\Sigma'',F')\end{array}}{(i:C,t:S,\Sigma,F,P,\Psi)\rightarrow(C',S',\Sigma'',F',P,\Psi')}$$

Because S is a stack shared by the evaluation of expressions, the values in the top of S could not be correspond to the arguments of the current method call at the execution of proceed. To avoid this problem, we modify the AROUND rule as follows

$$\textsc{Around}\ \frac{\begin{array}{c}\Sigma(v_0)=(X,Fd) \qquad mbody(m,X)=(x_1,\ldots,x_n).e \\ \mathsf{update}(\Psi,call\ m^n,S,\Sigma)=(\Psi',\Sigma') \qquad (\circ\Psi')(call\ m^n,v_0,F)=(\sigma(\phi),around)\end{array}}{\begin{array}{c}(call\ m^n:C,v_0:v_1:\ldots:v_n:S,\Sigma,F,P,\Psi) \\ \rightarrow(\sigma(\phi):pop_p:C,S,\Sigma',F,(e[x_1/v_1,\ \ldots,\ x_n/v_n,\ this/v_0],((X)m,v_0)):P,\Psi')\end{array}}$$

$$\textsc{Proceed}\ \frac{}{\begin{array}{c}(\mathsf{proceed}:C,S,\Sigma,F,(e,((X)m,v_0)):P,\Psi) \\ \rightarrow(e:\{push_p\ (e,((X)m,v_0)):C\},S,\Sigma,((X)m,v_0):F,P,\Psi)\end{array}}$$

$$\textsc{Pop}\ \frac{}{(pop_p:C,S,\Sigma,F,Z:P,\Psi)\rightarrow(C,S,\Sigma,F,P,\Psi)}$$

In those rules, it is not the current instruction ($call\ m^n$) which is pushed in the proceed stack but the body of this method call in which its arguments are substituted by the corresponding values and its signature. That permits to resolve the problem of values which do not correspond to the arguments of the current method call at the execution of proceed. Like in the general rules of around aspects, the rule PROCEED executes the expression on top of the proceed stack. Because this expression is the body of the method of which the signature is on top of P, PROCEED pushes also this signature in F and inserts a block representing the return address of this method. Also, we do not have to tag $e[x_1/v_1,\ \ldots,\ x_n/v_n,\ this/v_0]$ because it could be match by an aspect. Another modification is the replacement of $test\ \phi$ by ϕ because in featherweight AspectJ, ϕ does not depend on Σ.

Let us now consider a simple around Aspectj aspect `A1` that assigns the field `o` with `this` before proceeding with `bar`.

```
aspect A1 {

  Moo around(Moo r): call(Moo Moo.bar()) && target(r) {
    r.o = r;
    proceed();
  }
}
```

One way to express this aspect as a featherweight aspectj aspect is:

- to add the following class in the base program

```
class A1 {

  Moo advice(Moo r) {
    return this.advice2(r.o = r,proceed());
  }
  Moo advice2(Object o, Moo e) {
    return e;
  }
}
```

  the methods `advice` and `advice2` permits to transform the statements in the advice into an expression similarly to the AFJ transformation of a java program.
- after that, we transform the aspect `A1` into a featherweight aspectj aspect `A1'` as follows

```
aspect A1' {

   ((Moo)bar) /\ target(r)
   Moo around(Moo r)  {
     (new A1()).advice(r)
   }
}
```

In this transformation, we create an object of `A1` to access to the method `advice` of this class. We apply the AROUND rules above to `A1'` in the following example

EXAMPLE **9** *In this example, we weave the aspect `A1'` and execute the result woven program of the base program executed at the EXAMPLE 8 page 24. After the build of program and aspect environments, we have:*

$$
\begin{array}{lcl}
mbody & = & [(foo, Moo) \mapsto this.foo1(this.o = m, m), (foo1, Moo) \mapsto this.bar().bar(), \\
 & & (bar, Moo) \mapsto (Moo)(this.o), (advice, A1) \mapsto this.advice2(r.o = r, proceed()), \\
 & & (advice2, A1) \mapsto e] \\
FieldName & = & [Moo \mapsto (Object, o)] \\
\Psi & = & \{A1' \rightarrow \psi_{A1'}\} \\
Pointcut & = & [A1' \mapsto ((Moo)bar) \wedge target(r)] \\
Advice & = & [A1' \mapsto (new\ A1()).advice(r)] \\
Field & = & \perp
\end{array}
$$

The execution of the woven program without matching an instruction by an aspect is like the execution of the base program by applying the rule NoAdvice. *Then,*

$$
\begin{array}{l}
(new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object())))).call\ bar() : \varepsilon,\ \varepsilon,\ \perp,\ \varepsilon,\ \varepsilon,\ \Psi) \\
\rightarrow^*\ (call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi) \\
wiht\ \Sigma = [v_0 \mapsto (Object, \perp), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), \\
v_3 \mapsto (Object, \perp), v_4 \mapsto (Moo, o \mapsto v_2)] \\
\quad F = ((Moo)foo1, v_4) : ((Moo)foo, v_4) \\
\quad \Psi = \{A1' \rightarrow \psi_{A1'}\} \\
Because \\
\qquad update(\Psi, call\ bar^0, v_4 : \varepsilon, \Sigma) = (\Psi, \Sigma) \\
\qquad\quad (\circ\Psi)(call\ bar^0, v_4, F) = \psi_{A1'}(call\ bar^0, v_4, F) \\
\qquad\qquad\qquad = (\sigma(\phi_{A1'}), around)\ \ with\ \sigma = \{r \rightarrow v_4\} \\
\qquad Because \\
\qquad match_G(Pointcut(A1'), call\ bar^0, F, v_4) = match_G((Moo)bar \wedge target(r), call\ bar^0, F, v_4) \\
\qquad = match_G((Moo)bar, call\ bar^0, F, v_4)\ \wedge\ match_G(target(r), call\ bar^0, F, v_4) \\
\qquad = \emptyset \cup \{r \rightarrow v_4\} = \{r \rightarrow v_4\} \\
\qquad then\ \sigma(\phi_{A1'}) = \sigma(Advice(A1')) \\
\qquad\qquad = \sigma((new\ A1()).advice(r)) \\
\qquad\qquad = (new\ A1()).advice(v_4) \\
then \\
(call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi) \\
\rightarrow\ (\sigma(\phi_{A1'}) : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ F,\ P,\ \Psi) \\
with\ P = ((Moo)v_4.o, ((Moo)bar, v_4)) : \varepsilon \\
\rightarrow\ ((new\ A1()).advice(v_4) : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ F,\ P,\ \Psi) \\
\rightarrow\ (v_4 : new\ A1() : call\ advice^1 : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ \varepsilon,\ \Sigma,\ F,\ P,\ \Psi) \\
\rightarrow\ (new\ A1() : call\ advice^1 : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ P,\ \Psi) \\
\rightarrow\ (New^0_{A1} : call\ advice^1 : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ P,\ \Psi) \\
\rightarrow\ (v_5 : call\ advice^1 : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma[v_5 \mapsto (A1, \perp)],\ F,\ P,\ \Psi) \\
\rightarrow\ (call\ advice^1 : pop_p : call\ bar^0 : \{\{call\ bar^0\}\},\ v_5 : v_4,\ \Sigma[v_5 \mapsto (A1, \perp)],\ F,\ P,\ \Psi)
\end{array}
$$

$\rightarrow\ (v_5.advice2(v_4.o = v_4, proceed()) : \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\},\ \varepsilon,\ \Sigma,\ F,\ P,\ \Psi)$
$with\ F = ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\quad \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2), v_5 \mapsto (A1, \bot)]$
$\rightarrow\ (v_4.o = v_4 : proceed() : v_5 : call\ advice2^2 : \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\},\ \varepsilon, \Sigma, F,\ P, \Psi)$
$\rightarrow^*\ (proceed() : v_5 : call\ advice2^2 : \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ P,\ \Psi)$
$with\ \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot)]$

$\rightarrow\ ((Moo)v_4.o : \{push\ ((Moo)v_4.o, ((Moo)bar, v_4)) : v_5 : call\ advice2^2$
$: \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi)$
$with\ F = ((Moo)bar, v_4) : ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow^*\ (call\ advice2^2 : \{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\},\ v_5 : v_4 : v_4,\ \Sigma,\ F,\ P,\ \Psi)$
$with\ P = ((Moo)v_4.o, ((Moo)bar, v_4)) : \varepsilon$
$\quad F = ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (v_4 : \{\{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\},\ \varepsilon,\ \Sigma,\ F,\ P,\ \Psi)$
$with\ F = ((Moo)advice2, v_5) : ((Moo)advice1, v_5) : ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (\{\{pop_p : call\ bar^0 : \{\{call\ bar^0\}\}\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ P,\ \Psi)$
$\rightarrow^*\ (pop_p : call\ bar^0 : \{\{call\ bar^0\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ P,\ \Psi)$
$with\ F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\rightarrow\ (call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi)$
at this state also
$(\circ\Psi)(call\ bar^0, v_4, F) = (\sigma(phi_{A1'}), around)\ \ with\ \sigma = \{r \rightarrow v_4\}\ \ then$

$(call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi)$
$\rightarrow^*\ (\{\{call\ bar^0 : \varepsilon\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi)$
$with\ F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\quad \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), v_3 \mapsto (Object, \bot),$
$v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot), v_6 \mapsto (A1, \bot)]$
$\rightarrow^*\ (call\ bar^0 : \varepsilon,\ v_4 : \varepsilon,\ \Sigma,\ \varepsilon,\ \varepsilon,\ \Psi)$
this state is also match by the aspect

$\rightarrow^*\ (\varepsilon,\ v_4 : \varepsilon,\ \Sigma,\ \varepsilon,\ \varepsilon,\ \Psi)$
$with\ \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), v_3 \mapsto (Object, \bot),$
$v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot), v_6 \mapsto (A1, \bot), v_7 \mapsto (A1, \bot)]$
$\quad \Psi = \{A1' \rightarrow \psi_{A1'}\}$

7.4 Control flow pointcuts

Here, we want to illustrate the weaving of an aspect with the *cflow* pointcut. For that we consider the following aspect `A2'`. This aspect crosscuts calls to `bar` which are in the control flow of `foo1`.

It assigns the field `o` of the receiver of `foo1` with the value of this receiver before proceeding with `bar`.

```
aspect A2′{

  ((Moo)bar) /\ cflow((Moo)foo1 /\ target(r))
  Moo around(Moo r){
   (new A1()).advice(r)
  }
 }
```

The weaving of this aspect and the execution of the woven program is as follows

EXAMPLE **10** *The base program environments mbody and FieldName is as in the EXAMPLE 9. The aspects environments are:*

$$\begin{array}{lcl} \Psi & = & \{A2' \rightarrow \psi_{A2'}\} \\ P & = & [A2' \mapsto ((Moo)bar) \wedge cflow((Moo)foo1 \wedge target(r))] \\ Advice & = & [A2' \mapsto (newA1()).advice(r)] \\ Field & = & \bot \end{array}$$

then,
$(new\ Moo(new\ Object()).foo(new\ Moo(new\ Moo(new\ Object()))).call\ bar() : \varepsilon,\ \varepsilon,\ \bot,\ \varepsilon,\ \varepsilon,\ \Psi)$
$\rightarrow^*\ (call\ bar^0 : call\ bar^0 : \{\{call\ bar^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi)$
wiht $\Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1),$
$v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_2)]$
$F = ((Moo)foo1, v_4) : ((Moo)foo, v_4)$
$\Psi = \{A2' \rightarrow \psi_{A2'}\}$
Because
$(\circ\Psi)(call\ bar^0, v_4, F) = \psi_{A2'}(call\ bar^0, v_4, F)$
$= (\sigma(\phi_{A2'}), around)\ \ with\ \sigma = \{r \rightarrow v_4\}$

Because
$match_G(Pointcut(A2'), call\ bar^0, F, v_4)$
$= match_G(((Moo)bar) \wedge cflow((Moo)foo1 \wedge target(r)), call\ bar^0, F, v_4)$
$= match_G((Moo)bar, call\ bar^0, F, v_4) \wedge match_G(cflow((Moo)foo1 \wedge target(r)), call\ bar^0, F, v_4)$
$= match_G((Moo)bar, call\ bar^0, F, v_4) \wedge match_G(cflow((Moo)foo1 \wedge target(r)), call\ foo1^2, F', v_4)$
$with\ match_G((Moo)foo1 \wedge target(r), call\ bar^0, F, v_4) = Fail$
$F' = ((Moo)foo, v_4) : \varepsilon$
$= match_G((Moo)bar, call\ bar^0, F, v_4) \wedge match_G((Moo)foo1 \wedge target(r), call\ foo1^2, F', v_4)$
$= \emptyset \cup \{r \rightarrow v_4\}$
$= \{r \rightarrow v_4\}$
$then\ \sigma(\phi_{A2'}) = \sigma(Advice(A2'))$
$= \sigma((new\ A1()).advice(r))$
$= (new\ A1()).advice(v_4)$

$$
\begin{array}{l}
\textit{then} \\
(\textit{call bar}^0 : \textit{call bar}^0 : \{\{\textit{call bar}^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi) \\
\rightarrow^*\ (\textit{call bar}^0 : \{\{\textit{call bar}^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi) \\
\textit{with } F = ((Moo)foo1, v_4) : ((Moo)foo, v_4) \\
\qquad \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), \\
v_3 \mapsto (Object, \bot), v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot)] \\
\qquad \Psi = \{A2' \rightarrow \psi_{A2'}\} \\
\textit{as we have compute at the above state, at this state also} \\
(\circ\Psi)(\textit{call bar}^0, v_4, F) = \psi_{A2'}(\textit{call bar}^0, v_4, F) \\
\qquad\qquad\qquad\qquad = (\sigma(\phi_{A2'}), \textit{around})\ \ \textit{with } \sigma = \{r \rightarrow v_4\}
\end{array}
$$

$$
\begin{array}{l}
\textit{then} \\
(\textit{call bar}^0 : \{\{\textit{call bar}^0\}\},\ v_4 : \varepsilon,\ \Sigma,\ F,\ \varepsilon,\ \Psi) \\
\rightarrow^*\ (\textit{call bar}^0 : \varepsilon,\ v_4 : \varepsilon,\ \Sigma,\ \varepsilon,\ \varepsilon,\ \Psi) \\
\textit{with } \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), v_3 \mapsto (Object, \bot), \\
v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot), v_6 \mapsto (A1, \bot)] \\
\textit{at this state} \\
(\circ\Psi)(\textit{call bar}^0, v_4, \varepsilon) = \varepsilon \\
\qquad \textit{Because} \\
\qquad match_G(Pointcut(A2'), \textit{call bar}^0, \varepsilon, v_4) \\
\qquad\ = match_G(((Moo)bar) \wedge cflow((Moo)foo1 \wedge target(r)), \textit{call bar}^0, \varepsilon, v_4) \\
\qquad\ = \textit{Fail because } match_G(cflow((Moo)foo1 \wedge target(r)), \textit{call bar}^0, \varepsilon, v_4) = Fail
\end{array}
$$

$$
\begin{array}{l}
\textit{then} \\
(\textit{call bar}^0 : \varepsilon,\ v_4 : \varepsilon,\ \Sigma,\ \varepsilon,\ \varepsilon,\ \Psi) \\
\rightarrow^*\ (\varepsilon,\ v_4 : \varepsilon,\ \Sigma,\ \varepsilon,\ \varepsilon,\ \Psi) \\
\textit{with } \Sigma = [v_0 \mapsto (Object, \bot), v_1 \mapsto (Moo, o \mapsto v_0), v_2 \mapsto (Moo, o \mapsto v_1), v_3 \mapsto (Object, \bot), \\
v_4 \mapsto (Moo, o \mapsto v_4), v_5 \mapsto (A1, \bot), v_6 \mapsto (A1, \bot)] \\
\qquad \Psi = \{A2' \rightarrow \psi_{A2'}\}
\end{array}
$$

7.5 Association

As seen in section 6.3, we deal with association by defining the function update. Initially, update simply returns its arguments as such, that is, by default $\mathsf{update}(\Psi, i, S, \Sigma) = (\Psi, \Sigma)$, meaning that no aspect association is defined yet. As expected, each aspect that needs dynamic instantiation (pertarget and percflow) modifies update by creating a new instance when needed. Let us look at an example.

```
aspect A3 pertarget((Moo)bar) {
   ((Moo)bar) /\ cflow((Moo)foo1 /\ target(r))
```

```
    Moo around(Moo r){
     (new A1()).advice(r)
    }

}
```

For A3, the update function is modified as follows:

$$\mathsf{update}(\Psi, \mathsf{call\ bar}, v : S, \Sigma) \quad \stackrel{\Delta}{=} \quad (\Psi', \Sigma') \qquad \begin{array}{l} \text{if } \mathsf{A3}_v \notin \mathsf{dom}(\Psi) \\ \text{with } G_3(v,\Sigma) = (\psi_{id}, \Sigma') \\ \text{and } \Psi' = \Psi\{id \rightarrow \psi_{id}\} \end{array}$$

Here $G_3(v,\Sigma)$ is the generator for this aspect. Given the object v that triggered aspect instantiation and the current store Σ, it creates a new aspect instance, that is, a new function ψ_{id} and a new object a_v in Σ'.

$$G_3(v,\Sigma) = (\psi_{id}, \Sigma') \qquad\qquad \Sigma' = \Sigma[a_v \mapsto (A3, Fd)]]$$

where Fd is defined as in Rule $NEW2$. For ψ_{id}, we suppose to have the function

$$PertargetPc : id \rightarrow P$$

returning for each aspect identifier the pointcut $(P_T)P_I$ of the syntactic expression $pertarget((P_T)P_I)$ which is in the definition of a pertarget aspect. Then, ψ_{id} is defined as in page 27 in which $Pointcut(id)$ is replaced by $PertargetPc(id)$

Comparison with the single instance case In summary, the changes to the default case are the following:

- Interpreting a pertarget or a percflow aspect does not modify the global environment, but modifies the update function, as shown above.
- The function ψ associated to the aspect is similar to the default case, except that it refers to an instance a_v that depends on the current target v.
- The aspect is not instantiated initially (in function init), but rather dynamically, when needed by the function update.

8 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. These mechanisms are not fully orthogonal, for instance, both aspect association and stateful aspects update Ψ. We have illustrated the integration of several features into a core AspectJ language with around aspects, cflow and aspect association. In the future, we 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.

Appendix

As requested by an email of Alessandro Garcia on february 13th, this appendix list new terms for our ontology.

- stateful aspect: a stateful aspect has a control flow, hence a state. This state evolves when joinpoints are matched and their corresponding advice executed. This can be used to match *sequences* of pointcuts.
- aspect association/instantiation: when an aspect has a state (*e.g.* fields in AspectJ) it can have several instances. Each instance is usually associated with dynamic values of joinpoints (*e.g.* an instance per object).

References

[BJJR04] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc: A minimal aspect calculus. In *CONCUR 2004*, volume 3170. Springer LNCS, September 2004.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–66. ACM Press, 2000.

[CLW03] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. submitted for publication, October 2003.

[DB05] Rémi Douence and Didier Le Botlan. Towards a taxonomy of aop semantics: Milestone m8.1 for aosd noe. Technical Report M8.1, Formal Lab, Inria, 2005.

[DT04] Rémi Douence and Luc Teboul. A crosscut language for control-flow. In *GPCE 2004 - 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, Lecture Notes in Computer Science, Vancouver, Canada, October 2004. Springer-Verlag. to appear.

[DWWW] Daniel Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Analyzing polymorphic advice. unpublished.

[JJR03a] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, pages 54–73. Springer LNCS, 2003.

[JJR03b] Radha Jagadeesan, Alan Jeffrey, and James Riely. A typed calculus of aspect-oriented programs. unpublished, 2003.

[JJR05] Radha Jagadeesan, Alan Jeffrey, and James Riely. Featherweight aspect gj: Typed parametric polymorphism for aspects. unpublished, 2005.

[Läm02] Ralf Lämmel. A Semantical Approach to Method-Call Interception. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[MP05] Thomas Molhave and Lars H. Petersen. Assignment featherweight java: Bringing mutable state to featherweight java. 2005.

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, September 2004. Earlier versions of this paper were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *ICFP'03*, pages 127–139. ACM Press, 2003.