Specialized Aspect Languages Preserving Classes of Properties

Simplice Djoko Djoko
INRIA, EMN, LINA
simplice.djokodjoko@inria.fr

Rémi Douence
EMN, INRIA, LINA
douence@emn.fr

Pascal Fradet
INRIA
Pascal.Fradet@inria.fr

Abstract

Aspect oriented programming can arbitrarily distort the semantics of programs. In particular, weaving can invalidate crucial safety and liveness properties of the base program. In previous work, we have identified categories of aspects that preserve classes of temporal properties. We have formally proved that, for any program, the weaving of any aspect in a category preserves all properties in the related class. In this article, after a summary of our previous work, we present, for each aspect category, a specialized aspect language which ensures that any aspect written in that language belongs to the corresponding category. It can be proved that these languages preserve the corresponding classes of properties by construction. The aspect languages share the same expressive pointcut language and are designed w.r.t. *a common imperative base language. Each language is illustrated by simple examples. We also prove that all aspects written in one of the languages belong to the corresponding category.*

1 Introduction

Aspect oriented programming (AOP) proposes to modularize concerns that crosscut the base program [11]. However, aspects can in general distort the semantics of the base program. The programmer may have to inspect the woven program (or debug its execution) to understand its semantics. In a previous article [6], we have considered several categories of aspects that alter the semantics of the base program in a tightly controlled manner. For each category of aspects $\mathcal{A}_x$, we have identified a corresponding class of properties φ^x that is preserved by weaving these aspects. In other words, let P be a program that satisfies a property $\varphi \in \varphi^x$, then weaving any aspect $A \in \mathcal{A}_x$ on P will produce a program satisfying φ. Our categories of aspects, inspired by Katz's [10], comprise observers, aborters and confiners[1].

- ○ *Observers* do not modify the base program's state and control flow. Advice may only modify the aspect's local variables. Persistence, debugging, tracing, logging and profiling aspects are typical observers.

- ○ *Aborters* are observers which may also abort executions. The program's state is not modified but its control flow may be terminated. Aspects ensuring safety properties such as security aspects are usually aborters.

- ○ *Confiners* may modify the state and control flow but ensure that states remain in the reachable states of the base program. Some optimization aspects (which may use shortcuts to reach future states) or fault-tolerance aspects (which roll-back to past states) are confiners.

Categories of aspects are related by inclusion: observer aspects are included into aborters which are included into confiners. The classes of preserved properties are related by the opposite inclusion chain. Observers preserve a significant part of LTL properties whereas confiners only preserve invariant state properties.

In this article, we present for each aspect category a restricted aspect language which ensures that any aspect written in that language belongs to the corresponding category. Therefore, these languages ensure that the corresponding properties are preserved by construction. The aspect languages are designed for a simple imperative base language and use an expressive pointcut language. Each aspect language is illustrated using simple examples of aspects. We also prove that all aspects written in a language belong to the corresponding category.

In order to be self-contained, we recall our formal framework in Section 2 and the categories of aspects and classes of properties considered in Section 3. Section 4 introduces the imperative (base and advice) language, its associated pointcut language and three aspect languages corresponding to observers, aborters and confiners. Section 5 reviews some related work and Section 6 discusses possible future research directions and concludes.

[1]Several other categories are studied in [6] but, due to lack of space, we focus here on these three categories only

2 Semantic Framework

In order to formally study aspect categories, we have introduced a Common Aspect Semantics Base (CASB) for AOP [7]. The languages of Section 4 are defined within that abstract framework.

2.1 The Common Aspect Semantics Base

The CASB relies on the small step semantics of the base language. That semantics is described through a binary relation $\rightarrow_b$ on configurations (C, Σ^b) made of a program C (a sequence of basic instructions i terminated by $\bullet$) and a state Σ^b. A single reduction step of the base language semantics is of the form

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

Intuitively, i represents the current instruction and C the continuation.

In the following, woven configurations (C, Σ) are supposed to be made of the following components:

- C is the sequence of instructions of woven program. We write i_b for a base program instruction and i_a for an advice instruction. The instruction ϵ, which represents the final instruction of a program, is considered as an i_b instruction;
- The state Σ is made of three subsets $\Sigma^b \cup \Sigma^a \cup \Sigma^\psi$
  - Σ^b represents the state of the base program (*i.e.,* variables, environment, heap, manipulated by i_b and possibly i_a instructions);
  - Σ^a represents the local state of aspects (manipulated by i_a instructions only);
  - Σ^ψ represents aspects. It is a function that checks whether the current instruction should be woven and transforms the configuration accordingly.

When a new instance of an aspect is created, both Σ^a and Σ^ψ are modified. The semantics of woven reduction is represented by the binary relation $\rightarrow$ defined by:

$$\textsc{Reduce}\quad \frac{(C, \Sigma) \rightarrow_b (C', \Sigma') \qquad w(C', \Sigma') = (C'', \Sigma'')}{(C, \Sigma) \rightarrow (C'', \Sigma'')}$$

A reduction step $\rightarrow$ of the woven program first reduces the first instruction of the current configuration using $\rightarrow_b$, then it weaves the reduced configuration using the function w. The weaving function w is defined by two rules: either, the current instruction is not matched by the aspects (Σ^ψ returns nil) and w returns the configuration unchanged

$$\textsc{Weave}_0\quad \frac{\Sigma^\psi(C, \Sigma) = nil}{w(C, \Sigma) = (C, \Sigma)}$$

or the current instruction is matched by aspects and Σ^ψ returns a new configuration (C', Σ'):

$$\textsc{Weave}_1\quad \frac{\Sigma^\psi(C, \Sigma) = (C', \Sigma') \quad w(C', \Sigma') = (C'', \Sigma'')}{w(C, \Sigma) = (C'', \Sigma'')}$$

Note that weaving can be recursively applied on the code of a newly introduced advice. In some cases, we should prevent some instructions to be matched. For example, an aspect matching an instruction i and inserting a before advice a should not match i again just after executing a. We used tagged instructions such as $\bar{i}$ which have exactly the same semantics as i except that it is not subject to weaving. Formally

$$\textsc{Tagged}\quad \frac{(i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(\bar{i} : C, \Sigma) \rightarrow (C', \Sigma')}$$

Since weaving is always performed after a $\rightarrow_b$ reduction, it is not possible to weave the first instruction of the program. However, in some cases, it is useful to start the program by an advice. To permit such weaving, we assume that initial configurations are of the form $(start : C, \Sigma)$ where $start$ is a dummy first instruction.

In the following, programs are represented by their execution traces. For simplicity and regularity, we only consider infinite traces. In order to do so, the final instruction ϵ is supposed to have the following reduction rule:

$$\forall \Sigma.(\epsilon : \bullet, \Sigma) \rightarrow_b (\epsilon : \bullet, \Sigma)$$

The base program execution trace, with (C_0, Σ_0) as initial configuration, will be denoted by $\mathcal{B}(C_0, \Sigma_0)$ (definition 1).

Definition **1.**

$$\mathcal{B}(C_0, \Sigma_0) = (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots$$
$$\textit{with}\ \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow_b (i_{j+1} : C_{j+1}, \Sigma_{j+1})$$

Since the properties we consider concern only states and current instructions, continuation (the control stack) does not appear in traces. We write $\mathcal{W}(C_0, \Sigma_0)$ for the infinite woven execution trace (definition 2).

Definition **2.**

$$\mathcal{W}(C_0, \Sigma_0) = (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots$$
$$\textit{with}\ \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow (i_{j+1} : C_{j+1}, \Sigma_{j+1})$$

2.2 Properties

Properties are defined as LTL formulae [14] *w.r.t.* our (base and woven) execution traces. In our context, an atomic proposition of LTL is either an atomic proposition sp on states Σ (*e.g.,* $x \geq 0$), or an atomic proposition ep

on instructions or events (*e.g.,* `foo` which is *true* when the method `foo` is called). We consider LTL formulae in positive normal form *i.e.,* where negation occurs only on atomic propositions (Grammar 3).

GRAMMAR **3.**

$$\begin{array}{lll}\varphi & ::= & sp \mid \neg sp \mid ep \mid \neg ep \mid \varphi_1 \vee \varphi_2 \mid \\ & & \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \cup \varphi_2 \mid \varphi_1 W \varphi_2\end{array}$$

The operator $\bigcirc$ is read "next", $\cup$ is read "until", and W is read "weak until".

Standard classes of temporal properties [18] comprise: liveness properties (*e.g.,* *true* $\cup$ `backup` *i.e.,* the function `backup` is eventually called) and safety properties (*e.g.,* $\neg$`reset`$W false$ *i.e.,* the function `reset` is never called). In general, they are not preserved by aspect weaving. For instance, an aspect replacing calls to `backup` by an empty advice would invalidate the previous liveness property and an aspect with a call to `reset` in its advice would invalidate the previous safety property. The next section identifies categories of aspects that preserve classes of temporal properties.

3 Categories of aspects

Our aspect categories are: observers ($\mathcal{A}_o$), aborters ($\mathcal{A}_a$) and confiners ($\mathcal{A}_c$). The weaving of any aspect of a category $\mathcal{A}_x$ preserves a class of properties φ^x (a subset of LTL). Aspect categories are related by inclusion:

$$\mathcal{A}_o \subset \mathcal{A}_a \subset \mathcal{A}_c$$

The observer category is the most restricted category; it is included in all the other. The corresponding classes of properties are also related by inclusion:

$$\varphi^o \supset \varphi^a \supset \varphi^c$$

Not surprisingly, the most restricted category of aspects ($\mathcal{A}_o$) preserves the largest class of properties (φ^o) and the inclusion chain is in the opposite direction.

An important point to keep in mind is that our preservation properties stand for any program, any aspect of the category and any property of the class. Of course, for a specific program and aspect many more properties might be preserved. In that case, as soon as the program is modified the preservation of properties should be proved again. The advantage of our approach is that when an aspect is shown to belong to a category, then the corresponding class of properties that will be preserved for *any* program. So, when a program is modified, as long as it continues to satisfy a property of the considered class, we know that weaving the aspect will preserve it.

3.1 Observers

An observer does not modify the control flow of the base program but only inserts advice instructions i_a. This is formalized by Definition 4 which states that the woven and the base execution traces can be projected onto the same sequence of base instructions and that advice instructions i_a preserve the base state Σ^b. The projection function $proj_b$ (that discards i_a and Σ from traces) and the predicate $preserve_b$ (that checks that no i_a modifies Σ^b) are defined in the appendix.

DEFINITION **4.**

$$\begin{array}{lll}\forall (C,\Sigma).\ \Sigma^\psi \in \mathcal{A}_o & \Leftrightarrow & proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ & & \wedge\, preserve_b(\tilde{\alpha}) \\ \textit{with } \alpha = \mathcal{B}(C,\Sigma^b) \textit{ and } \tilde{\alpha} = \mathcal{W}(C,\Sigma) & & \end{array}$$

In other words, Definition 4 states that observers may only modify execution traces by inserting new advice instructions (i_a) and a new local state (Σ^a). This definition entails that the advice terminates.

The class of properties φ^o preserved by observer aspects is defined by Grammar 5.

GRAMMAR **5.**

$$\begin{array}{lll}\varphi^o & ::= & sp \mid \neg sp \mid \varphi^o_1 \vee \varphi^o_2 \mid \varphi^o_1 \wedge \varphi^o_2 \mid \varphi^o_1 \cup \varphi^o_2 \mid \\ & & \varphi^o_1 W \varphi^o_2 \mid \textit{true} \cup \varphi'^o \\ & & \\ \varphi'^o & ::= & ep \mid \neg ep \mid sp \mid \neg sp \mid \varphi'^o_1 \vee \varphi'^o_2 \mid \\ & & \varphi'^o_1 \wedge \varphi'^o_2 \mid \varphi^o_1 \cup \varphi^o_2 \mid \varphi^o_1 W \varphi^o_2 \mid \textit{true} \cup \varphi'^o\end{array}$$

As in the previous section, the variables sp and ep denote atomic propositions on the base state and instructions respectively. The language φ^o is LTL without the $\bigcirc$ operator when atomic propositions are state propositions (sp). So, it can express all safety, liveness and invariant properties (without $\bigcirc$) on base states Σ^b. The class is more restricted when the property involves atomic propositions on events (ep). These properties can only occur as $true \cup \varphi'^o$. This makes it possible to define liveness properties on events. Indeed, a liveness property can be expressed as $true \cup \varphi'^o$ and a liveness fair property as $(true \cup \varphi'^o) W false$. On the other hand, this language forbids safety properties on events. A safety property such as $\neg ep W false$ does not belong to grammar 5. Intuitively, safety properties on events forbid some sequences of instructions. Since an observer inserts new instructions, it may introduce a forbidden sequence. Persistence, debugging, tracing, logging and profiling aspects typically belong to the class of observers.

Property 6 formally states that the weaving of an observer preserves all properties in φ^o which were satisfied by the base program. Its proof can be found in [6].

PROPERTY **6.**

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_o \ \Rightarrow\ \forall(p \in \varphi^o).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

3.2 Aborters

An aborter does not modify the state of the base program. As in the previous definition of observers, the predicate $\mathit{preserve}_b$ holds for the woven trace. However, an aborter can modify the control flow by terminating the execution of the woven program. This is modeled by an i_a instruction `abort` which reduces any configuration into the final one:

$$\forall(C,\Sigma).\ (\texttt{abort} : C, \Sigma) \rightarrow (\epsilon : \bullet, \Sigma)$$

If `abort` is never executed, the projections of the base and woven traces are equal; the aborter behaves like an observer. Otherwise, the projection of an aborted woven trace on base instructions is a prefix of the projection of the base program trace. After this point, all instructions are equal to ϵ. The formal definition of aborters can be found in [6].

The class of properties preserved by aborters is defined by Grammar 7.

GRAMMAR **7.**

$$\varphi^a ::= \ \ sp \mid \neg sp \mid \varphi_1^a \vee \varphi_2^a \mid \varphi_1^a \wedge \varphi_2^a \mid \varphi_1^a W \varphi_2^a \mid \mathit{true} \cup \varphi'^a$$

$$\varphi'^a ::= \ \ \neg ep \ \mid \varphi'^a \vee \varphi^a \ \mid \varphi_1'^a \wedge \varphi_2'^a \ \mid \mathit{true} \cup \varphi'^a$$

The language φ^a is included in the set of properties preserved by observers. It is LTL for atomic propositions on states (sp) without $\cup$ and $\bigcirc$ operators. This includes invariant and safety properties on states. Atomic propositions on events (ep) occur only under a negation and only as an "eventually" formula (*i.e.,* in $\mathit{true} \cup \varphi'^a$). This language makes it possible to define liveness properties on $\neg ep$. Examples of aborters are security aspects that detect forbidden states or sequences of instructions or aspects that guarantee that a computation stops after a time-out. The preservation of properties of Grammar 7 by aborters is formalized in the same way as Property 6.

3.3 Confiners

An aspect is a confiner if the state of any configuration of the woven program is a reachable state. In general, confiners can modify the control flow and the state of the base program.

The set of reachable states from the configuration made of the program C and the state Σ^b is denoted by $\mathit{Reach}_b(C,\Sigma^b)$ with:

$$\mathit{Reach}_b(C,\Sigma^b) = \{\Sigma^{b'} \mid (C,\Sigma^b) \xrightarrow{*}_b (C',\Sigma^{b'})\}$$

Confiners are defined by the fact that the base states of the configurations of the woven trace remain in $\mathit{Reach}_b(\cdots)$. This is formalized by Definition 8.

DEFINITION **8.**

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_c \quad \Leftrightarrow \quad \forall(j \geq 1).\ \tilde{\alpha}_j = (i,\Sigma_j) \wedge \Sigma_j^b \in \mathit{Reach}_b(C,\Sigma^b)$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

The class of properties preserved by confiners is defined by Grammar 7.

GRAMMAR **9.**

$$\varphi^c ::= sp \ \mid \neg sp \ \mid \varphi_1^c \vee \varphi_2^c \ \mid \varphi_1^c \wedge \varphi_2^c \ \mid \varphi_1^c W \mathit{false}$$

The language φ^c is restricted to invariant properties (*i.e.,* $\varphi W \mathit{false}$) on states. Since confiner aspects can modify the control flow of events without restriction no properties involving atomic propositions on events in φ^c are preserved. For the same reason, safety properties such as $\varphi_1^c W \varphi_2^c$ are not preserved by confiners.

Examples of confiners are reset aspects that restore the initial state of the base program, fault-tolerance aspects that restore a safe execution state from a previous checkpoint, or memo aspects that shortcut a computation (or a already performed request) and returns its cached result. The preservation of the properties of Grammar 9 by confiners is formalized in the same way as Property 6.

4 Specialized Aspect Languages

In this section, we present the imperative base language (Section 4.1) and a generic pointcut language (Section 4.2) used by our aspect languages. We introduce in Sections 4.3, 4.4 and 4.5 three aspect languages corresponding to the three categories of the previous section. All aspects defined in a language belong to the corresponding category. Therefore each language ensures the preservation of the corresponding class of properties by construction.

4.1 Base language

A base program $Prog$ is a sequence D of declarations of global variables (g) and procedures followed by a main statement S. Besides usual commands (assignment, procedure call, sequencing, conditional, while loop), the instruction `abort` ends a program execution, `skip` does nothing and `loop`(A) S repeats A times the statement S. Arithmetic and boolean expressions are described by nonterminals A and B respectively. There are two distinguished kinds of variables:

- global variables (g) which are declared in D;
- local variables (l) declared as parameters of procedures.

Both kinds of variables can be used in assignments and expressions.

GRAMMAR **10.**

$$\begin{array}{lll}
Prog & ::= & D\ S \\
D & ::= & \texttt{var}\ g{:=}A \mid \texttt{proc}\ I(l_1, \ldots l_n)\ S \mid D_1;D_2 \\
S & ::= & V{:=}A \mid I(A_1, \ldots A_n) \mid S_1;S_2 \mid \\
 & & \texttt{if}(B)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(B)\ S \mid \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop}(A)\ S \\
A & ::= & n \mid V \mid A_1 + A_2 \\
B & ::= & \mathit{true} \mid A_1{=}A_2 \mid A_1{<}A_2 \mid B_1\&B_2 \mid !B \\
V & ::= & g \mid l \\
I & ::= & p
\end{array}$$

Note that since all variables are integers, we avoid typing issues. However, the language could be easily extended and equipped with a type system. As required by our framework (Section 2.1), its semantics is defined by a relation $\rightarrow_b$ on (C, Σ^b) where C is a sequence of statements S and Σ^b is made of environments associating global variables and parameters to their values and of a return stack for procedure calls. The operational semantics of this language is very similar to the *While* language of [16]. We omit it here. Example 11 illustrates the base language with a simple program which will be used throughout.

EXAMPLE **11.** *The fourth fibonacci number is specified as follows:*

```
var result := 0;
proc fib(x)
   if(x = 0) then result := result + 1 else
   if(x = 1) then result := result + 1
   else fib(x − 1); fib(x − 2)
fib(4)
```

4.2 Generic pointcut language

Our aspect languages share the same pointcut language which is defined by grammar 12.

GRAMMAR **12.**

$$\begin{array}{lll}
P & ::= & S^p \mid \mathit{if}(B^p) \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \\
S^p & ::= & V^p{:=}A^p \mid I^p(A^p_1, \ldots, A^p_n) \mid S^p_1;S^p_2 \mid \\
 & & \texttt{if}(B^p)\ \texttt{then}\ S^p_1\ \texttt{else}\ S^p_2 \mid \texttt{while}(B^p)\ S^p \mid \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop}(A^p)\ S^p \mid \beta_S \mid \neg S^p \\
A^p & ::= & n \mid V^p \mid A^p_1 + A^p_2 \mid \beta_A \mid \neg A^p \\
B^p & ::= & \mathit{true} \mid A^p_1{=}A^p_2 \mid A^p_1{<}A^p_2 \mid B^p_1\&B^p_2 \mid !B \mid \\
 & & \beta_B \mid \neg B^p \\
V^p & ::= & g \mid l \mid \beta_V \mid \neg V^p \\
I^p & ::= & p \mid \beta_I \mid \neg I^p
\end{array}$$

A pointcut is either a statement with pattern variables S^p (a static pointcut), or a predicate $\mathit{if}(B^p)$ (a dynamic pointcut), or a logical composition of pointcuts. A statement pattern S^p is a statement which enables, for each syntactic category (expressions, variables, . . .), pattern variables as well as negative patterns (*e.g.,* $\neg S$). For example, A^p defines patterns on arithmetic expressions with pattern variables (β_A) (able to match any arithmetic expression) and negations. I^p defines patterns of procedure identifiers. Matching of a pattern S^p *w.r.t.* a current configuration $(i : C, \Sigma)$ assigns values to pattern variables β_S, β_A, . . . These values will be substituted for the occurrences of pattern variables occurring in dynamic pointcuts $\mathit{if}(b)$ as well as in advice. The semantics of patterns with negation (called anti-patterns) is described in details in [12].

Dynamic pointcuts $\mathit{if}(b)$ should represent valid boolean expressions after substitution. To ensure this property, negation of patterns (*e.g.,* $\neg B^p$) are not allowed to occur within dynamic pointcuts. Also, variables occurring in dynamic pointcuts (and advice) should also occur outside the scope of a negation in the static pointcut (to have a unique substitution).

EXAMPLE **13.** *To provide some intuition, here are a few examples of patterns*

- $\mathtt{x} := \beta_A$ *matches all assignments to* x*;*
- $(\neg \mathtt{x}) := \beta_A$ *matches all assignments but those to x;*
- $\neg(\mathtt{x} := \mathtt{y})$ *matches all statements but* x := y*;*
- $\texttt{while}(\beta_B)\ \beta_S$ *matches all while statements.*
- $\mathtt{p}(3, \beta_A) \wedge \mathtt{if}(\beta_A = 0)$ *matches all calls to* p *with* 3 *and an arithmetic expression whose value is* 0*;*

Our implementation of pointcuts relies on a preliminary transformation described in [7]. A pointcut p is transformed into an equivalent pointcut of the form

$$(p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n))$$

where the static patterns p_i are *mutually exclusive*. Each static pattern is matched to the current instruction using the anti-pattern algorithm [12] written match^s until a match is found. The function match^s returns a substitution which is applied to the corresponding dynamic pointcut and advice that will be evaluated relatively to the state. If no match exists, the function match^s returns *Fail*. For instance, $\mathit{match}^s(\mathtt{p}(3, \beta_A), \mathtt{p}(3, 0))$ returns $[\beta_A \mapsto 0]$ and $\mathit{match}^s(\neg\beta_A, 0)$ returns *Fail*.

4.3 Observer language

In this section, we define a restricted aspect language that ensures that any aspect defined in this language is an observer. As seen in Section 3.1, an observer does not modify the control flow of the base program but only inserts advice instructions (i_a). In order to remain consistent with AspectJ and most aspect-oriented languages, we consider around aspects composed of an arbitrarily complex statement of i_a instructions, followed by the command `proceed` to execute the matched statement, followed by another arbitrarily complex statement of i_a. When the advice execution is over, the base program execution is resumed after the matched statement.

Note that our `proceed` instruction does not have parameters. Otherwise, observers would be able to modify the parameters of procedures and arbitrarily change the state or the control-flow of the base program. Furthermore, the advice should terminate, otherwise the base program execution is never resumed and its control flow is not preserved. We ensure termination by disallowing while statements in advice, checking that there is no loop in the call graph of advice and ensuring that the pointcut cannot match any statement of its own advice. Another option would be to permit while-loops and recursion in advice and make the programmer responsible for ensuring termination.

The second condition an observer should obey is not to modify the state of the base program (*i.e.,* i_a instructions do not change the state Σ^b). We distinguish the base program variables (that can be read by an advice) from the aspect variables (that can be read *and* written by a i_a).

The semantics of `proceed` is expressed using a proceed stack (written Σ^P) in the global state [7]. When an around advice applies, the matched instruction is pushed onto that stack. The `proceed` instruction pops and executes the instruction on top on the proceed stack:

$$\textsc{Proceed}\quad \frac{\Sigma^P = i : \Sigma'^P}{(\texttt{proceed} : C, X \cup \Sigma^P) \to (i : C, X \cup \Sigma'^P)}$$

The syntax of observers is defined by the Grammar 14.

GRAMMAR **14.**

$$\begin{array}{lll}
Asp^o & ::= & D^o\ \texttt{around}\ P\ \{S_1^o; \texttt{proceed};\ S_2^o\} \\
D^o & ::= & \texttt{var}\ g^o := A^o \mid \texttt{proc}\ I^o(l_1^o, \ldots, l_n^o)\ S^o \mid \\
 & & D_1^o; D_2^o \\
S^o & ::= & V^o{:=}A^o \mid I^o(A_1^o, \ldots, A_n^o) \mid S_1^o; S_2^o \mid \texttt{skip} \mid \\
 & & \texttt{if}(B^o)\ \texttt{then}\ S_1^o\ \texttt{else}\ S_2^o \mid \texttt{loop}(A^o)\ S^o \\
A^o & ::= & n \mid V' \mid A_1^o + A_2^o \mid \beta_A \\
B^o & ::= & \mathit{true} \mid A_1^o{=}A_2^o \mid A_1^o{<}A_2^o \mid B_1^o \& B_2^o \mid \\
 & & !B^o \mid \beta_B \\
V^o & ::= & g^o \mid l^o \\
V' & ::= & V^o \mid g \mid \beta_V \\
I^o & ::= & p^o
\end{array}$$

An observer Asp^o defines variables g^o and procedures p^o to form the local state of the aspect. Then, `around` associates a pointcut with an advice which contains exactly one `proceed`. We have considered that an aspect has one pointcut and one advice to simplify the presentation but this could be easily generalized to several pointcuts and advices. The declarations D^o must not contain any occurrence of pattern variables. Other statements S^o are similar to statement patterns S^p but without negation $\neg$. Indeed, an advice must be a valid executable code after substitution of its pattern variables (β_A, β_B, β_V). Note that, the statement `abort` is not allowed in advice since it would change the control flow of the base program. Similarly, pattern variables β_S for statements are forbidden since they could match (and execute) assignments to base program variables. Note that, assignment statements in advice can only modify variables of the aspect (V^o). Of course, aspect and base variables (V') can both be read. Finally, an advice can only call procedure defined in the aspect (I^o) since calling a base program procedure could modify the base program state.

An aspect that counts calls to `fib` (Example 11) is defined in Example 15. This profiling aspect respects the grammar Asp^o and is therefore an observer.

EXAMPLE **15.** *Profiling calls to* `fib`

$$\texttt{var n} := \texttt{0 around (fib}(\beta_A)\texttt{)}\ \texttt{n} := \texttt{n} + \texttt{1}$$

The semantics of weaving (Section 2.1) represents an aspect as a function Σ^ψ that takes the current configuration (C, Σ) as parameter and returns either a new woven configuration (C', Σ'), or nil when the pointcut does not match. We define the semantics of our aspect language in order to generate Σ^ψ from an aspect definition as follows. The resulting function takes the current configuration as parameter and matches the first instruction i. First, as mentioned in the previous section, the pointcut p of the aspect is transformed into an exclusive disjunction of the form $(p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n))$. The function tests if the current instruction i is matched one of the static pointcuts p_i. If i is not matched, the function returns nil. Otherwise, the current instruction i is replaced by a code a and i is pushed on the proceed stack Σ^P. When it is executed, the conditional a tests the dynamic part b_i of the matched pointcut. If b_i is satisfied the advice s is executed, otherwise the execution `proceed`s with the original instruction i (the advice is not executed). The pattern variables in b and s are substituted by their matched values using the substitution σ returned by $match^s$.

$$\begin{array}{l}
[\![\texttt{around}\ (p)\ s]\!] = \\
\mathit{let}\ (p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n)) = \mathit{Transf}(p)\ \mathit{in} \\
\lambda(i : C, X \cup \Sigma^P). \\
\quad \begin{array}{llll}
\mathit{case} & match^s(p_1, i) = \sigma_1 & \mapsto & (\bar{a_1} : C, X \cup \bar{i} : \Sigma^P) \\
 & & \ldots & \\
 & match^s(p_n, i) = \sigma_n & \mapsto & (\bar{a_n} : C, X \cup \bar{i} : \Sigma^P) \\
 & \mathit{otherwise} & \mapsto & nil
\end{array} \\
\quad \mathit{where}\ a_i = \sigma_i(\texttt{if}(b_i)\ \texttt{then}\ s\ \texttt{else}\ \texttt{proceed})
\end{array}$$

The instruction $\bar{i}$ and the conditional $\bar{a_i}$ are tagged (see

Section 2.1) to prevent infinite weaving by matching them again and again.

That semantics distinguishes evaluation of the static part of a pointcut from the evaluation of its dynamic part. This is mandatory in order to faithfully model AspectJ-like languages where several aspects can interact together (*i.e.,* the dynamic part of a pointcut can depend on a previous advice execution). Property 16 formalizes the fact that any aspect in Asp^o is an observer.

PROPERTY **16.** $\forall a \in Asp^o . [\![a]\!] \in \mathcal{A}_o$

A sketch of the proof can be found in the appendix.

4.4 Aborter language

An aborter is an observer which may abort the execution. The aborter language is therefore very similar to the observer language. Its grammar Asp^a is expressed exactly as Asp^o except that the statement `abort` is allowed in S^a. The `abort` instruction reduces any configuration in a final configuration (see Section 3.2).

Example 17 specifies an aspect counting the number of calls to the procedure `fib` (of the Example 11). If the number of calls reaches 100.000 the program is aborted. This aspect can be used to enforce some computation quota. It is defined in Asp^a so it is an aborter.

EXAMPLE **17.** *Regulating calls to* `fib`

```
var nbCalls := 0; around (fib(β_A))
nbCalls := nbCalls + 1;
if(nbCalls = 100000) then abort else skip;
proceed; skip
```

Property 18 states that any aspect in Asp^a is an aborter.

PROPERTY **18.** $\forall a \in Asp^a . [\![a]\!] \in \mathcal{A}_a$

4.5 A confiner language

Confiners can arbitrarily modify the control flow and the state of the base program as long as the base state remains in the set of originally reachable states. A general purpose language ensuring this property is very hard to design. However, two specialized confiner languages come to mind:

- optimization dedicated languages whose advice would jump directly to a future reachable state;
- fault-tolerance dedicated languages whose advice would roll-back to a previous reachable state.

We propose here a specialized language dedicated to defining *memo aspects*. A memo aspect is an optimizing aspect that caches computations. It introduces memoization in the woven program: when a computation is performed for the first time, it stores its arguments and results. When the same computation is performed again, it shortcuts it and directly returns its previously stored results. Grammar 19 presents the syntax of this language.

GRAMMAR **19.**

$$\begin{array}{ll} Asp^m ::= & \mathsf{memo}\ (I^m(A^p_1, \ldots, A^p_n) \wedge \mathit{if}(B^o)) \\ I^m ::= & p \mid \beta_I \end{array}$$

A memo aspect is a primitive memo applied to a pointcut whose static part denotes the procedure calls to be memoized, and dynamic part is an arbitrary predicate. In order to implement sophisticated strategies of memoization a memo aspect can be combined with an observer. Since observers (and aborters) are included in the confiner category, the composition of a confiner aspect with any observer (aborter, confiner) aspect is also a confiner. For example, the base program could be first woven with an observer that collects statistics regarding procedure calls (*e.g.,* number of calls, depth of recursion,. . . .) in its variables. It is then woven with a memoization aspect whose predicate accesses the variables holding statistics.

To give the semantics of a memoization aspect, we need to compute the lists of variables a procedure reads and writes. These two lists are computed by the functions $read$ and $write$. We can now define the semantics of a memo aspect as a program transformation taking the aspect and the declarations (D) of the base program:

```
$T[\![\mathsf{memo}\ (p(a_1, \ldots, a_n) \wedge \mathit{if}(B^o))]\!]D =$
var cache := empty
around $(p(a_1, \ldots, a_n) \wedge \mathit{if}(B^o))$
if contain$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$
then   $\mathit{write}[\![D]\!]p :=$ lookup$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$
else   proceed;
       store$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p, \mathit{write}[\![D]\!]p)$
```

A memo aspect defines an initially empty `cache` variable to store computation results. A cache entry associates a triplet $(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$ (a procedure identifier, the list of its arguments and the list of the variables read) to the list of values of its written variable $\mathit{write}[\![D]\!]p$.

When the pointcut is matched, the resulting substitution σ is applied to the advice and it fully instantiates the procedure, its arguments, as well as the lists of read ($\mathit{read}[\![D]\!]p$) and written ($\mathit{write}[\![D]\!]p$) variables. When the advice is executed, if the cache contains the result of the computation (`contain`$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$) then the written variables are assigned with the result stored in the cache (`lookup`$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$), else the computation is performed and the cache is updated (`store`$(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p, \mathit{write}[\![D]\!]p)$). Actually, such an aspect is a confiner only if the updating ($\mathit{write}[\![D]\!]p :=$ `lookup`(...)) is considered as atomic. Otherwise the updating of several variables produces temporary unreachable states. In a concurrent context, updating should also be atomic.

Note that, for the sake of conciseness, we have defined the advice in the base language extended with data structures (*i.e.,* `cache` implements a hash table, and lists to represent the values of read and written variables) and a return value for procedures (*e.g.,* `contain`, `lookup`).

Example 20 defines a memo aspect for the `fib` procedure defined in the Example 11. It is easy to check that the procedure `fib` reads no variable and writes the single variable `result`.

EXAMPLE **20.** *Memoizing* `fib`

$$\texttt{memo}\ (\texttt{fib}(\beta_A) \wedge \mathit{if}(\beta_A > 10))$$

This aspect, generated by the transformation T, memoizes calls to `fib` *only if its argument is greater than 10 (to amortize the cost of caching).*

```
var cache := empty
around (fib(βA) ∧ if(βA > 10))
if(contain(fib,[βA], []))
then result := lookup(fib,[βA], [])
else proceed; store(fib,[βA], [], [result])
```

Our version of `fib` *(Example 11) computes many times the same calls and has exponential complexity. The previous memo aspect suffices to improve its complexity to linear time.*

Property 21 formalizes the fact that any memo aspect is a confiner.

PROPERTY **21.** $\forall a \in Asp^m . [\![T[\![a]\!]]\!] \in \mathcal{A}_c$

4.6 Other languages

In the previous sections, we have presented restricted aspect languages that preserve classes for properties for sequential programs. Actually, non-deterministic (*i.e.,* concurrent) programs bring new interesting categories of aspects and classes of properties. We have identified in [6] the categories of selectors and regulators. Selector aspects select some executions among the set of possible executions. Regulators select or abort some executions among the set of possible executions.

We briefly discuss how to design specialized aspect languages for these categories. First, the base language must be extended with a non-deterministic statement. For instance, the statement s_1 `or` s_2 executes non-deterministically either s_1, or s_2. Second, the aspect languages must take into account that new statement. The advice language can be extended with proceedLeft and proceedRight in order to define selectors. For instance, the aspect `around(`s_1 `or` s_2`) proceedLeft` would make deterministic a non-deterministic program by selecting always the left part of `or` statements. This language could be used to specify scheduling aspects. Regulators are selectors which can abort the program.

Observers and aborters remains valid when S_1^p `or` S_2^p is added to the pointcut language. Regarding confiners, our memo aspects must be adapted: functions $read$ and $write$ must be extended in order to collect variables in both branches of non-deterministic `or` statements. As in the deterministic case, this static analysis of read and written variables always terminates. Other specialized aspect languages could be defined for confiners. For instance, an aspect language for fault-tolerance could be defined using two kinds of advice. An advice `commit` would save the current state of the system then proceeds. Another advice `rollback` would restore the previously saved state of the system. Another option is: when the pointcut is s_1 `or` s_2, the advice `commit` non-deterministically selects (*i.e.,* proceeds with) a branch and saves a state such that a rollback will always execute the other branch. This option makes it possible to write an aspect that systematically explores all possible executions of a non-deterministic program. Both options save and restore reachable states, so such specialized aspect languages would preserve confiner properties.

5 Related Work

The starting point of our study is seminal work by Katz [10] that introduces the categories of spectative aspects (corresponding to observers) and regulative aspects (close to our aborters). However, that study is largely informal. Furthermore, it suggests static analyses (*e.g.,* alias analysis) to ensure that an aspect belongs to a category (*a posteriori* approach), while our work proposes syntactic criteria for language definitions (*a priori* approach).

Our work is based on an abstract (*i.e.,* language independent) small step semantics of woven execution. Several other works have formalized aspect languages. For example, Wand *et al.* [20]) propose a denotational semantics for a subset of AspectJ, Bruns *et al.* [1] present a formal aspect calculus μ**ABC**, and Clifton and Leavens [3] define an operational semantics for an imperative OO language.

Concerning aspect categories, Clifton *et al.* in [4] propose annotations to formally specify that aspects have restricted effects on the base program and on other aspects. These annotations can limit the control flow impact of an advice and they introduce a notion of ownership to specify possible side effects. This makes it possible to define observers and aborters but not confiners. Dantas and Walker [5] formally describe an aspect category named harmless advice. This category corresponds to our aborters. Their formalization is based on a big step semantics and a type system to ensure that an aspect cannot modify the final values of the base program. However, none of these works study or define classes of preserved temporal properties.

Krishnamurthi *et al.* [13] propose a modular verification technique for verifying a property is preserved in the woven program but it requires analyzing each aspect definition. Clifton and Leavens [2] define a formal semantics of weaving with a Hoare-logic and give informal definitions of observer and assistant aspects (which look close to our observers and aborters). Rinard, Salcianu, and Bugara [17] also propose categories of aspects. Some of them seem closely related to ours (*e.g.,* their observation aspects to ours observers), but they are informally defined and their impact on properties is not studied (*e.g.,* several of their categories can completely modify the semantics of the base program).

There have been several proposals of domain specific aspect languages. For example, Lopes [19] proposes two specialized languages RIDL and COOL for remote data transfer and synchronization. Mendhekar *et al.* [15] present an aspect language which makes use of a memoization primitive to optimize image processing systems. Fradet and Hong Tuan Ha [8] define an aborter-like language to prevent the denials of service such as starvation caused by resource management. The preservation of properties is not studied.

6 Conclusion

In previous work, we have formally identified categories of aspects that preserves classes of properties [6]. It was proved that *any* aspect in a category preserves *any* property of the corresponding class (*i.e.,* if the base program satisfies the property then the woven program still satisfies the property). However, it remained to check whether an aspect belongs to a category. This article solves that question by defining restricted aspect languages that ensure aspects to belong to specific categories and therefore to preserve a class of property. In particular, we have proposed a general language for observers and aborters and a domain-specific language for memo aspects (which belongs to confiners). We also discussed how to design further specialized aspect languages for other categories. Using that language approach, the programmer does not have to prove *a posteriori* that an aspect belongs to a category. The programmer uses the specialized aspect language that ensures *a priori* that the aspect belongs to the category. We have shown how to prove that the observer language can only specify observer aspects.

Several research directions are worth following. Our languages of aspects should be shown to be maximal. For instance, we should prove that all observers (resp. aborters) can be defined in the observer (resp. aborter) language. Of course, our memo language is not maximal: it does not enable the definition of rollback aspects that are also confiners. However, other specialized languages belonging to the confiner family should be studied (*e.g.,* dynamic optimizations, fault-tolerance aspects). Finally, these languages should be implemented to build an aspect programming workbench allowing to reason about aspect composition and the preservation of properties.

References

[1] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *CONCUR 2004*, pages 209–224. Springer-Verlag, 2004.

[2] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL Workshop*, 2002.

[3] C. Clifton and G. T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:321–374, 2006.

[4] C. Clifton, G. T. Leavens, and J. Noble. MAO: Ownership and effects for more effective reasoning about aspects. In *ECOOP*, volume 4609 of *LNCS*, pages 451–475, 2007.

[5] D. S. Dantas and D. Walker. Harmless advice. *SIGPLAN Not.*, 41(1):383–396, 2006.

[6] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *PEPM'08*, pages 135–145. ACM, 2008.

[7] S. Djoko Djoko, R. Douence, and P. Fradet. A common aspect semantics base and some applications. Technical Report AOSD-Europe Deliverable D135, August 2008.

[8] P. Fradet and S. Hong Tuan Ha. Aspects of availability. In *GPCE'07*, pages 165–174. ACM, October 2007.

[9] J. Gibbons and G. Hutton. Proof Methods for Structured Corecursive Programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, Aug. 1999.

[10] S. Katz. Aspect categories and classes of temporal properties. *TAOSD*, 1, 2006.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, June 1997.

[12] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In *ESOP*, pages 110–124, 2007.

[13] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12*, pages 137–146. ACM Press, November 2004.

[14] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[15] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Palo Alto, CA, USA, February 1997.

[16] F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, 1992.

[17] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12*, pages 147–158. ACM Press, 2004.

[18] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85*, pages 39–48, 1985.

[19] C. Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1997.

[20] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Trans. on Prog. Lang. and Sys.*, 26(5):890–910, 2004.

Appendix

This appendix presents the proof of property 16. It relies on Property 22 which implies directly Property 16 by definition of $\mathcal{A}_o$. The proofs of others properties are similar.

PROPERTY **22.**

$$\begin{array}{l} \forall(a \in Asp^o).\forall(C,\Sigma).\ \Sigma^\psi = [\![a]\!] \\ \quad \Rightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha}) \ \wedge\ preserve_b(\tilde{\alpha}) \\ with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Property 22 is proved using Lemmas 23 and 26 which show respectively that aspects do not modify the base program state and its control flow.

In proofs, if α is a trace then its i^{th} element is denoted by α_i and its prefix $\alpha_1 : \ldots : \alpha_j$ by $\alpha_{\rightarrow j}$. The auxiliary functions $proj_b$ and $preserve_b$ are defined as follows:

$$\begin{array}{l} proj_b : Traces_{\mathcal{B}} \cup Traces_{\mathcal{W}} \rightarrow Sequence_{i_b} \\ proj_b((i_b,\Sigma) : T) \ = \ i_b : (proj_b\, T) \\ proj_b((i_a,\Sigma) : T) \ = \ proj_b\, T \end{array}$$

$$\begin{array}{lcl} preserve_b : Traces_{\mathcal{W}} \rightarrow bool && \\ preserve_b(\tilde{\alpha}) & = & \forall(j \geq 1).\ \tilde{\alpha}_j = (i_a, \Sigma_j) \\ && \Rightarrow\ \tilde{\alpha}_{j+1} = (i, \Sigma_{j+1})\ \wedge\ \Sigma^b_j = \Sigma^b_{j+1} \end{array}$$

where $Traces_{\mathcal{B}}$, $Traces_{\mathcal{W}}$ and $Sequence_{i_b}$ denote the sets of base program execution traces, woven execution traces and sequences of base instructions respectively.

LEMMA **23.**

$$\begin{array}{l} \forall(a \in Asp^o).\forall(C,\Sigma).\ \Sigma^\psi = [\![a]\!]\ \Rightarrow\ preserve_b(\tilde{\alpha}) \\ with\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Proof. It is easy to see (proof by cases) that all i_a instructions of $\{S^o; \mathsf{proceed}; S^o\}$ modify only Σ^a after reduction by $\rightarrow$. Indeed, instructions of S^o write only aspects variables and the proceed stack Σ^P (modified by proceed) is a subset of Σ^a ($\Sigma^P \subset \Sigma^a$). □

To prove Lemma 26, we first prove Lemma 24 which expresses that for any prefix of α, there exists a prefix of $\tilde{\alpha}$ equal after projection on base program instructions.

LEMMA **24.**

$$\begin{array}{l} \forall(a \in Asp^o).\forall(C,\Sigma).\ \Sigma^\psi = [\![a]\!] \\ \quad \Rightarrow\ \forall(l \geq 1).\exists(m \geq l).\ proj_b(\alpha_{\rightarrow l}) = proj_b(\tilde{\alpha}_{\rightarrow m}) \\ with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Proof. By induction on length of α and $\tilde{\alpha}$ and assuming that the advice terminates (Hypothesis 25).

HYPOTHESIS **25.**

$$\forall(D^o\ \texttt{around}\ P\ \{s\} \in Asp^o).\ s\ \ terminates$$

By Hypothesis 25
$(\forall(j \geq 1).\ \tilde{\alpha}_j = (i_a, \_)\ \Rightarrow\ \exists(k > j).\ \tilde{\alpha}_k = (i_b, \_))$

Base case $\quad l = 1$

$\alpha_{\rightarrow 1} = (i_1, \_)$

$\Sigma^\psi(i_1 : \_, \_) = nil\ \Rightarrow\ \tilde{\alpha}_{\rightarrow 1} = (i_1, \_)$
by definition of $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ proj_b(\alpha_{\rightarrow 1}) = proj_b(\tilde{\alpha}_{\rightarrow 1})$
by definition of $proj_b$

$\Sigma^\psi(i_1 : \_, \_) \neq nil\ \Rightarrow\ \tilde{\alpha}_{\rightarrow 1} = (i_a, \_)$
by definition of $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ \exists(m > 1).\ \tilde{\alpha}_m = (i_1, \_) \ \wedge$
$\forall(m' < m).\ \tilde{\alpha}_{m'} = (i_a, \_)$
by Hypothesis 25, and definition of $\mathcal{W}(C,\Sigma)$
$\Rightarrow \exists(m > 1).\ proj_b(\alpha_{\rightarrow 1}) = proj_b(\tilde{\alpha}_{\rightarrow m})$
by definition of $proj_b$

Induction $\quad l = n$

We assume that

$$\exists(m \geq n).\ proj_b(\alpha_{\rightarrow n}) = proj_b(\tilde{\alpha}_{\rightarrow m})$$

and show that this is the case for $l = n + 1$
$\alpha_{\rightarrow n+1} = \alpha_1 : \ldots : \alpha_n : \alpha_{n+1}\ \wedge\ \alpha_{n+1} = (i_{n+1}, \_)$

$\Sigma^\psi(i_{n+1} : \_, \_) = nil$
$\Rightarrow\ \exists(m' = m + 1 \geq n + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_)$
$\vee\ (\exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_)$
$\wedge\ \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_))$
by Hypothesis 25, and definition of $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ \ \exists(m' \geq n + 1).\ proj_b(\alpha_{\rightarrow n+1}) = proj_b(\tilde{\alpha}_{\rightarrow m'})$
by definition of $proj_b$

$\Sigma^\psi(i_{n+1} : \_, \_) \neq nil$
$\Rightarrow\ \exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_)$
$\wedge\ \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_))$
by Hypothesis 25, and definition of $\mathcal{W}(C,\Sigma)$
$\Rightarrow\ \exists(m' > n + 1).\ proj_b(\alpha_{\rightarrow n+1}) = proj_b(\tilde{\alpha}_{\rightarrow m'})$
by definition of $proj_b$ and $\mathcal{W}(C,\Sigma)$ □

LEMMA **26.**

$$\begin{array}{l} \forall(a \in Asp^o).\forall(C,\Sigma). \\ \Sigma^\psi = [\![a]\!]\ \Rightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Proof. Using Lemma 24 and the coinduction relation [9] below

$$\begin{array}{l} proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ \Leftrightarrow\ \forall(k \geq 1).\ approx\ k\ proj_b(\alpha)\ =\ approx\ k\ proj_b(\tilde{\alpha}) \end{array}$$

where $approx\ k\ \alpha$ is a function returning the k-first elements of the sequence α. □