

Abstract

Aspect orientation is a useful and popular programming paradigm that intends to offer programmers the opportunity to separate concerns when developing their applications. However, the flexibility provided by these languages comes together with a high level of interference between aspects and original code, preventing a modular treatment. We believe that reasoning and formal verification of aspect oriented developments should also be conducted modularly. We extend standard Hoare-like verification with a modular and proof preserving technique to certify that a program augmented by the introduction of new aspects preserves its original specification.

We define then a compiler from a simplified abstract oriented language to a standard RTL language and prove that it is amenable to be extended with a certificate translator.

1. Introduction

Aspect-oriented programming (AOP) is a useful and popular paradigm that offers programmers the opportunity to separate concerns when developing their applications. From a programmer perspective, an aspect-oriented program typically comprises three parts: a baseline program that performs the expected task, a set of advices encapsulated in different aspects handling different concerns independently; and pointcuts descriptors that determine when aspects are triggered in the baseline program, or in advices. From an applicative perspective, aspect-orientation is transparent and AOP compilers target typical back-ends: indeed, it is the role of the compiler to integrate these concerns into a single executable, through a weaving mechanism that modifies the code of each function depending on the aspects that operate over it.

Despite recent efforts to pinpoint the semantics of aspects, the verification of aspect-oriented programs is in its infancy. The lack of verification methods for AOP is partly explained by the non-modular nature of aspects [4]. Nevertheless, Dantas and Walker [5] have recently argued that many useful aspects are harmless, in that “they may change the termination behaviour but do not influence the final result of the mainline code.”

A first contribution of this paper is to develop a sound and modular verification method for aspect-oriented programs. The method is based on verification condition generators, which are commonly used in program verification environments, and adopts the following principles:

- each function of the program is verified against its specification in isolation;
- each advice is verified against its corresponding specification;
- proof obligations ensuring preservation of original specification are extracted from specifications of functions and advices. Discharging this proof obligations automatically may be feasible, depending on the level of interference involved.

A second contribution is done in the context of a PCC framework. It consists of extending a compiler for a simple AOP language to a standard RTL language. It illustrates the modularity of the approach, by reusing previously generated certificates from original methods and introduced advices, which are merged to form certificates for the augmented methods.

2. Related work

Other verification approaches: Pipa [15] is presented as a simple extension to JML [11] for aspectJ [1]. It supports specification for aspects invariants, pre- and post-conditions for advices and variable introductions, specification inheritance and crosscutting. The main goal is to bring the benefits of verification tools for Java

programs to the AspectJ language, by extending an AspectJ to Java compiler with a simultaneous translation of a Pipa specification into a standard JML specification. The intention and key ideas are rigorously explained, some examples are given illustrating the transformation and the advantages of the approach, but actually the discussion remains informal and a definition for the transformation is missing.

Non-interference and modularity: Non-interference of advices with respect to the underlying program as well as with other advices and modularity of the verification is a main topic in several publications [4, 5, 6, 7, 8, 10, 13, 14].

Dantas and Walker define in [5] the notion of *harmless advice*. They say a harmless advice may interfere with the control flow (by preventing termination) and may also perform I/O, but it does not interfere with the final result of the original code. This weak interference property permits to reason about the original program independently. They propose an information-flow type system over a core AOP language ([14]) to check harmlessness with respect to the main program, and then show how a more amenable and oblivious high level language can be compiled to this core language. The property they check is much coarser-grained than what can be specified in our framework. However, it can certainly be combined to be part of our hybrid logic to certify and check that an advice does not interfere with the original global state.

In [4], Clifton and Leavens define a notion of modular reasoning and show why modularity is not a general property in AspectJ and how this can be improved. One additional benefit that is closely related to modular reasoning is separate compilation, a technique that allows to weave new aspects to an already compiled (or even already running) program. This work proposes a set of language features that obviates the need for whole program analysis. It mainly consists of a classification for aspects as *spectators* or *assistants*. The former includes aspects that do not modify the behavior of the original program, i.e. they only modify the state space they own and they do not alter the control flow. In the other hand, *Assistants* can interfere with the original behavior of the program but only if that is *explicitly accepted* by the original program. In our work, we also rely on a declaration for the level of interference by specifying the frame conditions of each method and advice body. However, in contrast, control-flow can be altered as long as the original functionality is preserved. That implies, for instance, that even if termination is compromised, we can still certify that an augmented program preserves its original specification.

In [8, 7], Shmuel Katz et al. proposes a classification of aspects as *spectative*, *regulative* or *invasive*, depending on the level of interference with respect to the *underlying* program. The main motivation for this work is to simplify program verification by focusing on the properties that may be affected by the introduction of an aspect. Program properties are specified with temporal logic formulae, and each aspect category is described, analyzing how already valid properties are influenced. Following the result of this analysis, a concrete static procedure to classify advices is proposed. This work is similar to our VCGen in the sense that favors modularity of the verification process and makes emphasis on the preservation of original properties. It considers a more expressive language and proposes a more automatic interference checking. More concretely, it consists of model-checking a state machine representation of the aspect merged with a representation for the underlying program. In contrast, our verification of functionality preservation is based on interactive Hoare logics. Consequently, judgments that include only *spectative* and *regulative* advices are straightforward to derive.

PPO and certificate translation. Certificate translation for a simple AOP language is based mainly on previous work on Preservation of Proof Obligations (PPO). In [3], Barthe et al. show that,

given a specific VCGen, a sufficiently simple compiler generates, from an imperative source program, a stack based low-level piece of code, whose proof obligations are syntactically equal. Similar results on a wider verification framework are detailed in [12], for a significant subset of Java Bytecode.

However, when the compiler performs program transformations, proof obligations are not necessarily preserved and thus original certificates must be translated in consequence. In [2], a first certificate translator is defined addressing standard program optimizations for a simple RTL language.

3. Setting

We start with the definition of the base program, and extend later the syntax to introduce new aspects.

A base program is defined as a set of methods, together with global variable declarations and a special main statement. The base syntax can be found in Fig. 1. In the figure, v stands for any element in the domain of program variables \mathcal{V} , and f ranges in the set of predefined method names \mathcal{F} . The domain \mathcal{C} of statements is standard and include loops and conditional statements, together with method calls with secondary effects. Each method is composed of an identifier, its formal parameters and the command representing the function body.

Programs	$Prog$	$::=$	$vdecl^* meth^* c$
Variable declarations	$vdecl$	$::=$	$v:=e$
Methods	$meth$	$::=$	$f args^* vdecl^* c$
Commands	c	$::=$	$v:=f(e)$
			$return e$
			$while b do c$
			$if b then c else c$
			$v:=e \mid skip \mid abort$
comparison	rop	$::=$	$< \mid \leq \mid = \mid \geq \mid >$
integer expressions	e	$::=$	$n \mid v \mid e op e \mid \dots$
boolean expressions	b	$::=$	$true \mid false$
			$e rop e \mid \neg b \mid b bop b$

Figure 1. Syntax of Base Programs

3.0.1 Advices

The syntax for extending base programs by the introduction of a sequence of aspects is presented in Fig. 2. In the figure, a will stand for any advice identifier in the set \mathcal{A} .

Augmented Program	$AProg$	$::=$	$Prog aspect^*$
Aspects	$aspect$	$::=$	$vdecl^* advice^*$
Advices	$advice$	$::=$	$ptd^+ a args^* vdecl^* c_a$
Advice commands	c_a	$::=$	$v:= proceed(e)$
			$return e$
			$while b do c_a$
			$if b then c_a else c_a$
			$v:=e \mid skip \mid abort$
pointcuts descriptors	ptd	$::=$	$before(f) \wedge ptd'$
			$after(f) \wedge ptd'$
			$around(f) \wedge ptd'$
	ptd'	$::=$	$_ \mid if(b) \mid cflow(f)$
			$ptd' \wedge ptd' \mid ptd' \vee ptd'$
			$\neg ptd'$

Figure 2. Syntax for Aspect Extension

An extended program will be composed of a sequence of aspects attached to a base program defined as above. Each aspect will

be formed by a set of advices, with its corresponding point-cut descriptors, and a set of variables that are globally visible by any advice in the aspect. For simplicity, we suppose that every variable declared within an aspect is different to any global variable of the original program. We suppose that this syntax implies an order relation between aspects and between advices. This is desirable to statically solve the precedence of advices whose associated methods coincide.

Declarations for advices are similar to functions, in that they are composed of an identifier, a formal parameter and a command representing its body.

For clarity, we will initially focus on point-cut descriptors that allow us to infer statically the exact sequence of advices that will execute around a given join-point. That is, we will at first avoid considering cflow and conditional point-cut descriptors. Hence, point-cut descriptors are of limited expressiveness. We will explain later how we can extend the verification process to deal with this dynamic conditions.

An advice body, is a command similar to a function body, extended with a new statement `proceed`. The argument passing and returning is explained in following sections, together with the behavior of the `proceed` command. In addition, as can be seen in the syntax for advice commands, the expressiveness of an advice body is reduced by disallowing calls to base code functions.

Running example: To illustrate the approach with a running example we consider a extended program syntax. Suppose we have a program Pr , from which we isolate a method $m = \text{slowRetrieve}$ that returns the value stored in a slow access memory. This behavior is represented by taking as parameter the integer $Address$ i and by accessing a global array variable `mem` with this index. We also suppose that P contains a method `main` that represents any method that may invoke function m .

We proceed by extending the original program with new base functions $f_1 = \text{initializeCache}$, $f_2 = \text{updateCache}$ and $f_3 = \text{isAvailable}$. We add two global array variables `available` and `cache`, accessible only by this new functions. These new functions and variables are intended to be accessed later by the introduction of advices.

3.1 Semantics

Local stores $\sigma : \Sigma$ and global environments $\eta : H$ are both represented as mappings from variables in \mathcal{V} to values (\mathbb{Z}) . Execution states can be classified as intermediate (Δ^I) or final (Δ^F) states, where intermediate states $(\langle \sigma, \eta \rangle)$ are composed of a local and a global environment ($\Delta^I = \Sigma \times H$) and final states $\langle n, \eta \rangle$ consist of a final environment and the return value ($\Delta^F = \mathbb{Z} \times H$).

In the rest of the paper we may implicitly rely on a standard denotational semantics (denoted $\llbracket \cdot \rrbracket_{\eta}^{\sigma}$). This function may be possibly overloaded to be applied to integer or boolean expressions, and propositions.

Semantics over base statements is also standard and is defined as a relation \rightsquigarrow between execution states ($\rightsquigarrow : \mathcal{C} \times \Delta^I \rightarrow \Delta$). Clearly, final states will appear only on the right hand side of the relation. The definition of this semantic can be found in Fig. 3. As can be seen in the figure, in order to facilitate soundness proofs for the predicate transformer wp , we favor a *small-step* semantics. We will see later that computability of the weakest precondition requires called functions to be previously specified. In consequence, according to the same criteria, the rule for function calls is defined as a *big-step*.

A final state expresses the fact that a `return` statement has been executed, and some rules are defined accordingly. In addition, we require execution states to reduce always to final states before returning from a function call. However, the definition of a rule for function calls deserves special consideration. Since it represents

$$\begin{array}{c}
\frac{}{\langle \text{return } e, (\sigma, \eta) \rangle \rightsquigarrow \ll [e]_{\eta}^{\sigma}, \eta \gg} \\
\frac{}{\langle \text{while } b \text{ do } c, (\sigma, \eta) \rangle \rightsquigarrow (\sigma, \eta)} \neg \ll [b]_{\eta}^{\sigma} \\
\frac{\langle c, (\sigma, \eta) \rangle \rightsquigarrow \ll n, \eta' \gg}{\langle \text{while } b \text{ do } c, (\sigma, \eta) \rangle \rightsquigarrow \ll n, \eta' \gg} \ll [b]_{\eta}^{\sigma} \\
\frac{\langle c, (\sigma, \eta) \rangle \rightsquigarrow (\sigma', \eta') \quad \langle \text{while } b \text{ do } c, (\sigma', \eta') \rangle \rightsquigarrow S}{\langle \text{while } b \text{ do } c, (\sigma, \eta) \rangle \rightsquigarrow S} \ll [b]_{\eta}^{\sigma} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \rightsquigarrow S}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, (\sigma, \eta) \rangle \rightsquigarrow S} \ll [b]_{\eta}^{\sigma} \\
\frac{\langle c_2, (\sigma, \eta) \rangle \rightsquigarrow S}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, (\sigma, \eta) \rangle \rightsquigarrow S} \neg \ll [b]_{\eta}^{\sigma} \\
\frac{}{\langle x := e, (\sigma, \eta) \rangle \rightsquigarrow (\sigma \oplus [x \mapsto [e]_{\eta}^{\sigma}], \eta)} \\
\frac{}{\langle \text{skip}, (\sigma, \eta) \rangle \rightsquigarrow (\sigma, \eta)} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \rightsquigarrow \ll n, \eta' \gg}{\langle c_1; c_2, (\sigma, \eta) \rangle \rightsquigarrow \ll n, \eta' \gg} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \rightsquigarrow (\sigma', \eta') \quad \langle c_2, (\sigma', \eta') \rangle \rightsquigarrow S}{\langle c_1; c_2, (\sigma, \eta) \rangle \rightsquigarrow S} \\
\frac{\langle \text{body}(g), ([\text{in} \mapsto [e]_{\eta}^{\sigma}], \eta) \rangle \rightsquigarrow \ll n, \eta' \gg}{\langle x := g(e), (\sigma, \eta) \rangle \rightsquigarrow (\sigma \oplus [\text{in} \mapsto n], \eta')}
\end{array}$$

Figure 3. Standard Code Semantics

the point where advices may be weaved around, its definition is postponed.

The rule for function call deserves special attention since it will be later modified to formalize program execution under an aspect oriented environment.

Advice Weaving. Advice weaving is triggered when a join-point matches a corresponding point-cut descriptor. Commonly, point-cut designators are composed of properties that can be statically checked together with dynamic conditions. The most common characterization of a point-cut is a function call, i.e., the fact that a particular function (or a function with a particular prototype or from a particular module) is called can be used to specify the necessity of executing an advice. But in addition, other dynamic properties can be checked on run-time to decide whether to execute an advice or not. These include conditions such as cflow or the validity of an arbitrary boolean expression.

We proceed by simplifying the weaving procedure by taking as hypothesis that for every method f , we are able to infer statically which advices will be triggered to assist the execution of f , and in which order. To represent the result of this inference we will use the following notation: we denote the result of augmenting any method m with θ_m , which will be composed of a single method m , or of an advice a appended *before*, *after* or *around* a smaller augmented method θ'_m respectively denoted by $a \triangleright \theta'$, $\theta' \triangleleft a$ or $a \bowtie \theta'$ (this construction defines the domain Θ). When introducing a dynamic condition (such as cflow) as a point-cut descriptor, we will be forced to overestimate this sequence as a set of approximations. These non-statically resolvable conditions are usually implemented

$$\frac{\ll [e]_{\eta}^{\sigma}, \eta \gg \uparrow^{\theta_g} \ll n, \eta' \gg}{\langle x := g(e), (\sigma, \eta) \rangle \rightsquigarrow (\sigma \oplus [\text{in} \mapsto n], \eta')}$$

where θ_g is the static weaving for g

Figure 4. Function Call Semantics

$$\begin{array}{c}
\frac{\langle c, ([\text{in} \mapsto n], \eta) \rangle \rightsquigarrow \ll n', \eta' \gg}{\langle n, \eta \rangle \uparrow^m \ll n', \eta' \gg} \\
\frac{\langle c_a, ([\text{ina} \mapsto n], \eta) \rangle \rightsquigarrow \ll n', \eta' \gg \quad \ll n', \eta' \gg \uparrow^{\theta} \ll n'', \eta'' \gg}{\langle n, \eta \rangle \uparrow^{a \triangleright \theta} \ll n'', \eta'' \gg} \\
\frac{\ll n, \eta \gg \uparrow^{\theta} \ll n', \eta' \gg \quad \langle c_a, ([\text{in} \mapsto n'], \eta') \rangle \rightsquigarrow \ll n'', \eta'' \gg}{\langle n, \eta \rangle \uparrow^{\theta \triangleleft a} \ll n'', \eta'' \gg} \\
\frac{\langle c_a, ([\text{in} \mapsto n], \eta) \rangle \xrightarrow{x} \ll n', \eta' \gg}{\langle n, \eta \rangle \uparrow^{a \bowtie \theta} \ll n', \eta' \gg} \quad x \in \{\top, \perp\}
\end{array}$$

where c is $\text{body}(m)$, c_a is $\text{body}(a)$, and \xrightarrow{x} is defined in Fig. 6.

Figure 5. Weaving Semantics

by inserting at specific points a piece of code that checks on runtime whether the condition is satisfied or not. That involves the introduction of new structures to store extra information, such as a representation of the call stack. In our formalism we represent this variant by attaching to every advice augmentation ($a \triangleright \theta'$, $\theta' \triangleleft a$ or $a \bowtie \theta'$) a *condition residue* to resolve *dynamically* whether the concerned advice must be executed. Nevertheless, we will initially restrict our weaving semantics to be statically decidable and later show how the VCGen may be extended to deal with this dynamic conditions.

A new rule for the function call replaces the original one and is defined in Fig. 4. In the rest of the paper, we denote standard semantics as $\rightsquigarrow_{\text{IMP}}$ and weaved semantics as $\rightsquigarrow_{\text{AOP}}$. This new rule for function calls relies on a new relation $\uparrow: \Theta \rightarrow \Delta^F \rightarrow \Delta^F$ which is in turn defined in Fig 5.

As can be seen in Fig. 5, the relation \uparrow involves two final states and a sequence of advices θ , reducing the latter by one in each reduction step. The case for the trivial empty sequence simply executes the original method m , while the cases for *around* and *after* augmentation rely on a strict subset of the rules for \rightsquigarrow (strict since the advice body it is not allowed to call a function nor to proceed). The case for *around* advices is a bit more complicated since we have to deal with *proceed* statements. This is managed by introducing a new set of rules to define the semantics of commands that may contain a *proceed* statement.

This operational semantics is represented by the relation $\xrightarrow{x}: \mathbb{B} \rightarrow \Theta \rightarrow (\mathcal{C} \times \Delta^I) \rightarrow \Delta$ and is defined in Fig. 6. It is similar to the small-step semantics for standard commands (\rightsquigarrow), but the difference relies mainly on the fact that a *proceed* statement must take into account a sequence of remaining advices to execute (θ). In addition, the parameter x represents the fact that a *proceed* statement has been executed. This way, advice execution will be restricted so that an advice cannot proceed twice.

$$\begin{array}{c}
\frac{\langle c_1, (\sigma, \eta) \rangle \xrightarrow[\perp]{\theta} \langle \sigma', \eta' \rangle \quad \langle c_2, (\sigma', \eta') \rangle \xrightarrow[x]{\theta} S}{\langle c_1; c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} \langle n, \eta' \rangle}{\langle c_1; c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} \langle n, \eta' \rangle} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \xrightarrow[\top]{\theta} \langle \sigma', \eta' \rangle \quad \langle c_2, (\sigma', \eta') \rangle \xrightarrow[\perp]{\theta} S}{\langle c_1; c_2, (\sigma, \eta) \rangle \xrightarrow[\top]{\theta} S} \\
\frac{\langle n, \eta \rangle \xrightarrow{\theta} \uparrow (n', \eta')}{\langle x := \text{proceed}(e), (\sigma, \eta) \rangle \xrightarrow[\top]{\theta} \langle \sigma \oplus [x \mapsto n'], \eta' \rangle} \\
\frac{\langle \text{return } e, (\sigma, \eta) \rangle \xrightarrow[\perp]{\theta} \langle [e]_{\eta}^{\sigma}, \eta \rangle}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S} \llbracket b \rrbracket_{\eta}^{\sigma} \\
\frac{\langle c_1, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S} \llbracket b \rrbracket_{\eta}^{\sigma} \\
\frac{\langle c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, (\sigma, \eta) \rangle \xrightarrow[x]{\theta} S} \neg \llbracket b \rrbracket_{\eta}^{\sigma} \\
\frac{\langle x := e, (\sigma, \eta) \rangle \xrightarrow[\perp]{\theta} \langle \sigma \oplus [x \mapsto [e]_{\eta}^{\sigma}], \eta \rangle}{\langle \text{skip}, (\sigma, \eta) \rangle \xrightarrow[\perp]{\theta} \langle \sigma, \eta \rangle}
\end{array}$$

where $x \in \{\top, \perp\}$.

Figure 6. Around Advice Statements Semantics

4. VCGen

When verifying a base program extended with aspects several approaches may be taken. In our case we prefer to keep the modularity of the process by separating concerns and analyzing the validity of the specification for each method or advice in isolation.

Both methods and advices will have an associated specification composed of a pre and post-condition, together with a *frame condition*. This specification states the expected functional behavior of the corresponding command and will be represented as an assertion in a first order logic. In a general setting an assertion will refer to variables (either global or local), as well as some special-purpose variables. These later variables may include *star variables* (v^*) which appear in the post-condition and in intermediate assertions and represent the initial value of some global variable v . A post-condition refers to global variables, res and the formal parameter (usually denoted in_f for method f). Global variables may appear on the postcondition with a $(.^*)$ modifier if its value may change during the execution of the function. Preconditions will refer to the corresponding formal parameter and any global variable.

The frame conditions are specified with the set of global variables that may be modified by the body of the method or advice. W_m will stand for the set of variables modifiable by method m , and we will denote $\vdash_w m$ writes W_m the fact:

$$\exists \eta, \eta'. \langle \text{body}(m), (\sigma, \eta) \rangle \rightsquigarrow \langle n, \eta' \rangle \wedge \eta v \neq \eta' v \Rightarrow v \in W_m$$

4.1 Verifying Base Program

Verifying a particular method involves proving that its body satisfies its pre- and post-condition (possibly with a set of auxiliary invariants) and independently proving the frame conditions. The latter may be certified by several means, for instance by dataflow analyses, non-interference typing systems [Harmless Advices, Walker] or even Hoare-logics. We say that this certification is hybrid since proof validation can be performed by independent and possibly different analyses. We do not need to specify which approach is taken to certify the frame conditions as long as our VCGen remains hybrid along the whole compilation process. Instead, we suppose we have a method to generate a certificate for the judgment $\Gamma \vdash_w m$ writes W_m .

To verify that a given statement c satisfies a specification, we define the predicate transformer wp , which takes a function g with body statement c , a predicate φ , and a context Γ that maps function identifiers to their specifications. When referring to a specification for a method f in context Γ , we will focus on its pre and post-condition, leaving implicit the set of variables that may be modified. When needed, we will denote this part of the specification as $W_f = \Gamma^w(f)$, but its important to always recall that Γ^w is a projection of Γ . Usually, and for simplicity, a single variable y will represent the set of possible modified variables (stated $y = \Gamma^w(f)$), and under this assumption the substitution $[y'/y]$ will represent the simultaneous substitution of the modified variables by fresh variables.

let $y = \Gamma^w(f)$ and $(P_f, Q_f) = \Gamma(f)$ in

$$\begin{aligned}
\text{wp}_g(\text{skip}, \varphi) &= (\varphi, \emptyset) \\
\text{wp}_g(x := e, \varphi) &= (\varphi[x], \emptyset) \\
\text{wp}_g(c_1; c_2, \varphi) &= \\
&\quad \text{let } (\varphi_2, S_2) = \text{wp}(c_2, \varphi) \text{ in} \\
&\quad \text{let } (\varphi_1, S_1) = \text{wp}(c_1, \varphi_2) \text{ in} \\
&\quad (\varphi_1, S_1 \cup S_2) \\
\text{wp}_g(\text{return } e, \varphi) &= (\varphi[\text{res}], \emptyset) \\
\text{wp}_g(\text{if } b \text{ then } c_1 \text{ else } c_2, \varphi) &= \\
&\quad \text{let } (\varphi_1, S_1) = \text{wp}(c_1, \varphi) \text{ in} \\
&\quad \text{let } (\varphi_2, S_2) = \text{wp}(c_2, \varphi) \text{ in} \\
&\quad (b \Rightarrow \varphi_1 \wedge \neg b \Rightarrow \varphi_2, S_1 \cup S_2) \\
\text{wp}_g(\text{while } b \text{ do } c, \varphi) &= \\
&\quad \text{let } (\varphi', S') = \text{wp}(c, \text{Inv}) \text{ in} \\
&\quad (\text{Inv}, \{\text{Inv} \Rightarrow (b \Rightarrow \varphi' \wedge \neg b \Rightarrow \varphi)\} \cup S') \\
\text{wp}_g(x := f(e), \varphi) &= \\
&\quad (P_f[\text{in}_f] \wedge \\
&\quad \forall y', \text{res}. Q_f[\text{in}_f][y'/y][y/y^*] \Rightarrow \varphi[\text{res}/x][y'/y], \emptyset) \\
&\quad \text{where } y \text{ represents any variable in } W_f
\end{aligned}$$

Notice that the definition of wp is implicitly parametric on Γ .

We say that a base code function g is certified to follow specification $\Gamma(g) = (P_g, Q_g)$ if we have a proof for $P_g \Rightarrow \varphi[y'/y^*]$ and every predicate in the set S of proof obligations is valid. More

formally:

$$\begin{array}{c}
\frac{(P_g \Rightarrow \phi[y/y^*]) \wedge \bigwedge_{PO \in S} PO}{\Gamma \vdash_{\text{IMP}} \{P_g\}g\{Q_g\}} \\
\frac{\Gamma \vdash_{\text{IMP}} \{P'\}g\{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\Gamma \vdash_{\text{IMP}} \{P\}g\{Q\}} \\
\frac{\Gamma^w \vdash_w f \text{ writes } Y \quad Y \cap FV(\varphi) = \emptyset}{\Gamma \vdash_{\text{IMP}} \{\varphi\}g\{\varphi\}} \\
\frac{\Gamma^1 \vdash_{\text{IMP}} \{P\}g\{Q\} \quad \Gamma^2 \vdash_{\text{IMP}} \{P'\}g\{Q'\}}{\Gamma^3 \vdash_{\text{IMP}} \{P \wedge P'\}g\{Q \wedge Q'\}} \\
\frac{\Gamma \cup \Gamma' \vdash_{\text{IMP}} \Gamma \quad \Gamma^w \vdash_w \Gamma^w}{\Gamma' \vdash_{\text{IMP}} \Gamma}
\end{array}$$

where $\Gamma \Vdash \Gamma'$ is a notation for $\forall x. \Gamma \vdash \Gamma'(x)$
 $\forall f. \Gamma^i(f) = (P_i, Q_i) \Rightarrow \Gamma^3(f) = (P_1 \wedge P_2, Q_1 \wedge Q_2)$
 $\forall f. \Gamma \varphi(f) = (\varphi, \varphi) \wedge \Gamma \varphi^w = \Gamma^w$

The interpretation of the statement $\vdash_{\text{IMP}} \{P_g\}g\{Q_g\}$ is standard: for any value n and environment η such that $\llbracket P_g \llbracket \eta \rrbracket \rrbracket \eta$, we have that

$$\langle \text{body}(c), (\sigma, \eta) \rangle \rightsquigarrow_{\text{IMP}} \llcorner n', \eta' \rceil \Rightarrow \llbracket Q_g \llbracket \eta' \rrbracket \rrbracket \eta' [y^* \mapsto \eta y].$$

To give an intuition, we will later require θ_g to behave exactly as g , to simulate the original (simple imperative) behaviour of the function.

We generalize the previous statement by adding an hypothesis, such that

$$\Gamma \vdash_{\text{IMP}} \{P_g\}g\{Q_g\}$$

stands now for

$$\left(\bigwedge_{\Gamma(f) = (P_f, Q_f)} \vdash_{\text{IMP}} \{P_f\}f\{Q_f\} \right) \Rightarrow \vdash_{\text{IMP}} \{P_g\}g\{Q_g\}$$

Soundness of the VCGen implies that $\Gamma \vdash_{\text{IMP}} \{P_g\}g\{Q_g\}$ whenever $\Gamma \vdash_{\text{IMP}} \{P_g\}g\{Q_g\}$.

Running example: We continue by specifying the method m defined in the previous section with pre and post-conditions

$$\begin{array}{l}
P_m = 0 \leq i < N \\
Q_m = \text{res} = \text{mem}[i]
\end{array}$$

and by declaring that m does not modify any global variable ($W_m = \emptyset$). To emphasize the modularity of the approach we will deliberately leave its implementation and verification implicit. Instead, simply suppose that we can derive the judgments $\vdash_{\text{IMP}} \{P_m\}m\{Q_m\}$ and $\{P_m\}m\{Q_m\} \vdash_{\text{IMP}} \{P\} \text{main}\{Q\}$, where (P, Q) is an arbitrary specification. The intention is to prove that main preserves its original specification (P, Q) after extending the program Pr with the introduction of new aspects.

To specify methods f_1, f_2 and f_3 , we let ϕ stand for the consistency of the cache variable with respect to the availability array:

$$\phi = \forall i. (\text{available}[i] \Rightarrow \text{cache}[i] = \text{mem}[i]) .$$

To specify these new methods we define their respective pre/post-conditions:

$$\begin{array}{l}
P_{f_1} = \text{true} \\
Q_{f_1} = \phi \\
P_{f_2} = 0 \leq i < N \wedge \phi \\
Q_{f_2} = \text{cache} = \text{cache}^*[i \mapsto v] \wedge \phi \\
P_{f_3} = 0 \leq i < N \\
Q_{f_3} = \text{res} \equiv \text{available}[i]
\end{array}$$

Since these functions and the verification of their specification are standard we omit the actual implementation and the verification steps.

In addition, since we suppose that main does not modify neither mem nor the new variables available and cache, we can derive in two steps

$$\{P_m \wedge \phi\}m\{Q_m \wedge \phi\} \vdash_{\text{IMP}} \{P \wedge \phi\} \text{main}\{Q \wedge \phi\}$$

4.2 Verifying Advices in isolation

Verifying the specification of an advice body is similar to base program statements, since they share most of their commands and both are specified similarly.

For the sake of modularity, we extend the specification and VCGen for around advices. This extension consists of a new specification for the behavior expected when calling a proceed statement. We generalize the specification by introducing a new (proceed) specification P'_a and Q'_a for each around advice a , and new variables in'_a and res'_a that correspond respectively to the input and output value for the execution triggered by proceed. The assertion Q'_a will refer to the variables in'_a and res'_a , as well as any global variables (possible starred) and in_a . Similarly, P'_a will include conditions over in'_a as well as in_a and any global variable. In addition, a normal postcondition Q_a for an around advice may now refer also to the variable res'_a . We will require that a specification for proceed declares the set of variables that are allowed to change (W'_a).

The predicate transformer wp is extended for proceed statements:

$$\begin{array}{l}
\text{wp}_a(x := \text{proceed}(e), \phi) = \\
(P'_a \llbracket \eta \rrbracket \llbracket \text{in}'_a \rrbracket) \\
\wedge \forall y'. \text{res}'_a. Q'_a \llbracket \eta \rrbracket \llbracket \text{in}'_a \rrbracket [y'/y][y/y^*] \Rightarrow \phi \llbracket \eta \rrbracket \llbracket \text{res}'_a \rrbracket [y'/y][y/\text{in}'_a], S)
\end{array}$$

where y represents any variable that may be modified by the nested methods invoked by proceed, and P'_a and Q'_a is the augmented specification.

By using this wp function we can prove that an advice satisfies its specification by deriving a judgment $\Gamma_a \vdash_{\text{ADV}} \{P_a\}a\{Q_a\}$ for before and after advices, and $\Gamma_a \vdash_{\text{ADV}} \{(P_a, P'_a)\}a\{(Q'_a, Q_a)\}$ for around advices. **In the rest of the paper, since advices do not perform method calls, the context Γ_a is considered empty. We keep the rules this ways because we may want to extend advices commands with invocations to other advices or methods.**

Running example: We extend then the base program with a set of advices that improves the store access time by profiting from the introduced variables and functions. We start by introducing a non control-flow preserving around advice $a_1 = \text{fastRetrieve}$. This new advice will replace the functionality of method m by receiving as parameter the store address i and returning the cached value if available or, otherwise, by permitting the original function m to continue:

```

around slowRetrieve(Address i) fastRetrieve {
  b := isAvailable(i);
  if b
    return cache[i]
  else {

```

```

    Value v:=proceed(i);
    updateCache(i, v);
    return v
  }
}

```

The specification for this advice a_1 is

$$\begin{aligned}
P_{a_1} &= 0 \leq i < N \wedge \phi \\
P'_{a_1} &= P_m \\
Q'_{a_1} &= Q_m \\
Q_{a_1} &= Q_m \wedge \phi
\end{aligned}$$

Notice that, since a_1 is a non *control-flow preserving* advice, its specification is similar to the specification for m . In addition, since it is intended to be executed only around m , we can safely define the proceed specification (P'_{a_1}, Q'_{a_1}) equal to the specification for m .

We can now prove the correctness of a_1 in isolation, by deriving $\vdash_{\text{ADV}}\{(P_{a_1}, P'_{a_1})\}a_1\{(Q'_{a_1}, Q_{a_1})\}$. To this end, it is sufficient to show, as a premise, that the proposition

$$\begin{aligned}
&P_{f_3} \wedge \forall b. (Q_{f_3} \stackrel{[b]/\text{res}}{\Rightarrow} \\
&b \Rightarrow Q_m \stackrel{[\text{cache}[i]/\text{res}]}{\Rightarrow} \wedge \phi \\
&\wedge \\
&\neg b \Rightarrow P_m \wedge \forall \text{res}. (Q_m \Rightarrow P_{f_2} \wedge \\
&\quad \forall \text{cache}' . (Q_{f_2} \stackrel{[\text{cache}'/\text{cache}]}{\Rightarrow} \stackrel{[\text{cache}'/\text{cache}^*]}{\Rightarrow} \\
&\quad (Q_m \wedge \phi) \stackrel{[\text{cache}'/\text{cache}]}{\Rightarrow}))
\end{aligned}$$

is implied by P_{a_1} .

4.3 Verifying the weaved code

To verify a method m included in an augmented program, we need to compute previously the sequence of advices that will be executed around it (θ_m). We proceed by deriving an appropriate judgment $\Gamma, \Gamma_a \vdash_{\text{AOP}}\{P\}\theta_m\{Q\}$. To understand the meaning of this judgment we first define as $\vdash_{\theta} \{P\}\theta\{Q\}$ the

fact that for any n, η, n' and η' if $\langle n, \eta \rangle \uparrow \langle n', \eta' \rangle$ and $\llbracket P \stackrel{[n'/\text{in}]}{\Rightarrow} \rrbracket \eta$ then $\llbracket Q \stackrel{[n'/\text{res}]}{\Rightarrow} \rrbracket \eta[y^* \mapsto \eta y]$. By the judgment $\Gamma, \Gamma_a \vdash_{\text{AOP}}\{P\}\theta\{Q\}$ we mean $\vdash_{\text{AOP}}\{P\}\theta\{Q\}$ under the hypothesis that for any f and a such that $\Gamma(f) = (P_f, Q_f)$ and $\Gamma_a = (P_a, Q_a)$, we have respectively that $\vdash_{\text{IMP}}\{P_f\}f\{Q_f\}$ and $\vdash_{\text{ADV}}\{P_a\}a\{Q_a\}$.

The derivation of this judgment will be defined inductively on the construction of θ and will rely strongly on the interference conditions. Since this frame conditions are defined only for methods and advices, we should extend this definition to nested term θ . This definition does not represent any difficulty, its is straightforward and safe to extend the judgment for frame conditions to augmented methods ($\Gamma \vdash_w \theta_m$ writes W) by taking W as the union of the modifiable variables by all the components of θ_m .

We start by defining the rule that relates executions on the simple imperative language to executions augmented with advices. This rule serves as a basis for the construction of the term θ_f , but in addition requires that the original specification of the invoked functions are preserved. We will say that a specification (P_g, Q_g) is a refinement of (P'_g, Q'_g) if $P'_g \Rightarrow P_g$ and $Q_g \Rightarrow Q'_g$. Furthermore, a context Γ refines a context Γ' , if for any g in the domain of Γ , $\Gamma(g)$ is a refinement for $\Gamma'(g)$ and if for any f , the set $\Gamma^{w'}(f)$ contains $\Gamma^w(f)$ (modulo introduced variables). This latter condition reflects the fact that we only care about the variables that belonged to the original program. The intention of the rule is to propagate a derivation on the simple imperative side relying on functionality preservation after weaving the advices as declared.

$$\frac{\Gamma \vdash_{\text{IMP}}\{P\}m\{Q\} \quad \Gamma_{\Theta} \text{ refines } \Gamma}{\Gamma_{\Theta}, \Gamma_a \vdash_{\text{AOP}}\{P\}m\{Q\}}$$

This rule resembles a standard rule for weakening the judgment by strengthening the hypothesis but it is explicitly stated here to emphasize the fact that we are requiring functional preservation when moving to an aspect oriented context.

The next rule helps remove hypothesis from the context, and is intended to deal with mutually recursive functions:

$$\frac{\vdash_{\text{ADV}}\Gamma_a \quad \Gamma_{\Theta} \cup \Gamma'_{\Theta}, \Gamma_a \vdash_{\text{AOP}}\Gamma_{\Theta} \quad \Gamma^w_{\Theta} \Vdash \Gamma^w_{\Theta}}{\Gamma'_{\Theta} \Vdash_{\text{AOP}}\Gamma_{\Theta}}$$

4.3.1 Around advices.

In this case we need to consider two alternatives, depending on how much the around advice interferes with the control flow of the rest of the system. If `proceed` is never executed by a , that will imply also that no subsequent advice will be executed, and consequently, any specification refinement until this point will be lost. That means, that even if we can ensure a stronger postcondition for θ thanks to the already weaved advices, $a \bowtie \theta$ may not propagate this augmented specification if `proceed` is not always executed. A similar reason explains why we should not call `proceed` more than once. We define *control flow preserving* advices as those whose every path in its control flow contains exactly one `proceed` statement. It may be argued that this condition is too strong, since we are relying on syntactic properties. An alternative approach may be designing a special purpose VCGen that ensures that a `proceed` statement is executed exactly once. However, since this technique implies defining a new set of loop invariants, and generating and merging a new certificate, we prefer instead a static checking approach. Under the hypothesis that a is a *control flow preserving* advice we can apply the following rule:

$$\frac{\Gamma_a \vdash_{\text{ADV}}\{P_a\}a\{Q_a\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}}\{P_{\theta}\}\theta\{Q_{\theta}\} \quad W_{\theta} \subseteq W'_a \quad P \Rightarrow P_a \wedge \forall x'. (P'_a \stackrel{[x'/x]}{\Rightarrow} P_{\theta} \stackrel{[\text{in}'_a/\text{in}_{\theta}]}{\Rightarrow} Q[x'/x]) \quad Q_{\theta} \stackrel{[\text{in}'_a/\text{in}_{\theta}]}{\Rightarrow} \stackrel{[\text{res}'/\text{res}]}{\Rightarrow} Q'_a \wedge \forall x'. (Q_a \stackrel{[\text{in}'_a/\text{in}_a]}{\Rightarrow} Q[x'/x])}{\Gamma, \Gamma_a \vdash_{\text{AOP}}\{P\}a \bowtie \theta\{Q\}}$$

where x' represents the global variables potentially modified by a . In case that it cannot be checked whether the control flow will be preserved, we use this rule instead:

$$\frac{\Gamma_a \vdash_{\text{ADV}}\{P_a\}a\{Q_a\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}}\{P_{\theta}\}\theta\{Q_{\theta}\} \quad W_{\theta} \subseteq W'_a \quad P'_a \Rightarrow P_{\theta} \stackrel{[\text{in}'_a/\text{in}_{\theta}]}{\Rightarrow} Q_{\theta} \stackrel{[\text{in}'_a/\text{in}_{\theta}]}{\Rightarrow} \stackrel{[\text{res}'/\text{res}]}{\Rightarrow} Q'_a}{\Gamma, \Gamma_a \vdash_{\text{AOP}}\{P_a\}a \bowtie \theta\{Q_a\}}$$

It can be argued that the rule above restricts the completeness and modularity of the approach. However we believe that is inherent to advices with such level of interference. If an around advice replaces the original functionality of a base function (re-implementation), then it would be expected to be specified at least as the original function (and in addition the functionality of subsequent advices is lost). This former fact may prevent to reuse the advice around different functions.

Running example: To derive $\vdash_{\text{AOP}}\{P_{a_1}\}a_1 \bowtie m\{Q_{a_1}\}$ we proceed by applying the rule for non *control-flow preserving* around advices. This requires the previous derivation of the judgments $\vdash_{\text{AOP}}\{P_m\}m\{Q_m\}$ and $\vdash_{\text{ADV}}\{(P_{a_1}, P'_{a_1})\}a_1\{(Q'_{a_1}, Q_{a_1})\}$ as premises. The first judgment is clearly derivable since the context is empty. It remains to discharge two proof obligations, but according to the rule applied they are trivial by definition of P'_{a_1}, Q'_{a_1} .

4.3.2 Before advices.

In the following rule, w stand for global variables modified by the body of the advice, y stands for the variables modified by the nested construction θ and z for any global variable modified either by a or

until the compilation of the original method m .

$$\begin{aligned}
C_w(m) &= \text{let } (\cdot, \text{ins}) = C_c(1, \text{body}(m)) \text{ in} \\
&\quad [m \mapsto \text{ins}] \\
C_w(a \triangleright \theta) &= \text{let } (\cdot, \text{ins}_a) = C_a(f_\theta, \text{body}(a)) \text{ in} \\
&\quad \text{let } c = x := a(\text{in}); \\
&\quad \quad x := f_\theta(x); \\
&\quad \quad \text{return } x \\
&\quad \text{let } (\cdot, \text{ins}) = C_c(1, c) \text{ in} \\
&\quad [f_{a \triangleright \theta} \mapsto \text{ins}] \\
C_w(\theta \triangleleft a) &= \text{let } (\cdot, \text{ins}_a) = C_a(f_\theta, \text{body}(a)) \text{ in} \\
&\quad \text{let } c = x := f_\theta(\text{in}); \\
&\quad \quad x := a(x); \\
&\quad \quad \text{return } x \\
&\quad \text{let } (\cdot, \text{ins}) = C_c(1, c) \text{ in} \\
&\quad [f_{\theta \triangleleft a} \mapsto \text{ins}] \\
C_w(a \bowtie \theta) &= \text{let } (\cdot, \text{ins}_a) = C_a(f_\theta, \text{body}(a)) \text{ in} \\
&\quad [f_{a \bowtie \theta} \mapsto \text{ins}_a]
\end{aligned}$$

In addition, when compiling standard commands, the result of compiling a call to method h will be a call to function f_{θ_h} , which will be in charge of calling the resulting weaved code.

5.2 Verification over the target language

VCGen Suppose $(\text{ins}, l') = C_c(l, \text{body}(g))$

$$\begin{aligned}
\text{Let } (P_f, Q_f) &= \Gamma(f) \text{ and} \\
(\varphi', S') &= \text{wp}_{\text{ins}}(\text{succ}(l), \varphi) \text{ in} \\
\text{wp}_{\text{ins}}(l : \perp, \varphi) &= (\varphi, \emptyset) \quad (\text{ins}[l] \text{ is undefined}) \\
\text{wp}_{\text{ins}}(l : \text{nop}, \varphi) &= (\varphi', S') \\
\text{wp}_{\text{ins}}(l : \text{assert } \psi, \varphi) &= (\psi, \{\psi \Rightarrow \varphi'\} \cup S') \\
\text{wp}_{\text{ins}}(l : x := e, \varphi) &= (\varphi'[\frac{e}{x}], S') \\
\text{wp}_{\text{ins}}(l : x := e_1 \text{ op } e_2, \varphi) &= (\varphi'[\frac{e_1 \text{ op } e_2}{x}], S') \\
\text{wp}_{\text{ins}}(l : \text{jmp } l', \varphi) &= \text{wp}_{\text{ins}}(l', \varphi) \\
\text{wp}_{\text{ins}}(l : \text{jmpif } e_1 \text{ cmp } e_2, l', \varphi) &= \\
&\quad \text{let } (\varphi'', S'') = \text{wp}_{\text{ins}}(l', \varphi) \text{ in} \\
&\quad (e_1 \text{ cmp } e_2 \Rightarrow \varphi'' \wedge \neg e_1 \text{ cmp } e_2 \Rightarrow \varphi', S' \cup S'') \\
\text{wp}_{\text{ins}}(l : x := \text{invoke } f \ e, \varphi) &= \\
&\quad (P_f[\frac{e}{\text{in}}] \wedge \forall \text{res}, y'. Q_f[\frac{e}{\text{in}}][\frac{y'}{y}][\frac{y}{y^*}] \Rightarrow \varphi'[\frac{\text{res}}{x}][\frac{y}{y'}], S') \\
\text{wp}_{\text{ins}}(l : \text{return } e, \varphi) &= (\varphi[\frac{e}{\text{res}}], \emptyset)
\end{aligned}$$

PO's

$$\begin{aligned}
&\frac{(P_g \Rightarrow \phi[\frac{y}{y^*}]) \wedge \bigwedge_{\text{PO} \in S} \text{PO}}{(\text{ins}, l') = C_c(l, \text{body}(g)) \quad (\phi, S) = \text{wp}_{\text{ins}}(l, Q_g)} \\
&\quad \Gamma \vdash_{\text{RTL}} \{P_g\} g \{Q_g\} \\
&\quad \frac{\Gamma \cup \Gamma' \Vdash_{\text{RTL}} \Gamma \quad \Gamma^w \Vdash_w \Gamma^w}{\Gamma' \Vdash_{\text{RTL}} \Gamma}
\end{aligned}$$

6. Certificate Translation

Lemma 1 (expressions compiler). Let $C_e^x(l, e) = (l', \text{ins})$ and let ins' be a sequence of instructions containing ins , i.e. ins is equal to the infix $\text{ins}'[l, \dots, l']$. If $(\phi', S') = \text{wp}_{\text{ins}'}(l', \phi)$, then $\text{wp}_{\text{ins}'}(l, \phi)$ is equal to $(\phi'[\frac{e}{x}], S')$.

Lemma 2 (booleans compiler). Let $C_b(l, l_t, l_f, b) = (l', \text{ins})$ and let ins' be s.t. $\text{ins} = \text{ins}'[l, \dots, l']$. If $(\phi_t, S_t) = \text{wp}_{\text{ins}'}(l_t, \phi)$ and $(\phi_f, S_f) = \text{wp}_{\text{ins}'}(l_f, \phi)$ then $\text{wp}_{\text{ins}'}(l, \phi)$ is equivalent to $(b \Rightarrow \phi_t \wedge \neg b \Rightarrow \phi_f, S_t \cup S_f)$.

We say proof obligations are equivalent instead of syntactically equal since when compiling a composed boolean condition, for instance $b_1 \wedge b_2$, the wp function returns the proposition $b_1 \Rightarrow b_2 \Rightarrow \phi$ but, in the other hand, syntactic equality would require the equivalent proposition $(b_1 \wedge b_2) \Rightarrow \phi$.

Lemma 3 (statement compiler). Suppose f_θ points to the compilation of method f (i.e., no advices have been weaved to f). Let $C_c(l, c) = (l', \text{ins})$ and let ins' be s.t. $\text{ins} = \text{ins}'[l, \dots, l']$. If $(\phi', S') = \text{wp}_{\text{ins}'}(l', \phi)$ and $(\phi'', S'') = \text{wp}(c, \phi')$ then $\text{wp}_{\text{ins}'}(l, \phi) \equiv (\phi'', S'' \cup S')$.

We are abusing notation, when we say $(\phi, S) \equiv (\phi', S')$ we mean $\phi \equiv \phi'$ and that for any proposition P in S there is an equivalent proposition P' in S' and vice-versa.

Lemma 4 (merging judgments). (Consequently, one of the derivation rules for \vdash_{IMP} is redundant.)

Proof. The proof is by simple structural induction on a command c . It can be proved that for any assertion φ if we compute $(\phi_1, S_1) = \text{wp}(c, \varphi, \Gamma^1)$, $(\phi_2, S_2) = \text{wp}(c, \varphi, \Gamma^2)$ and $(\phi_3, S_3) = \text{wp}(c, \varphi, \Gamma^3)$ then $\phi_1 \wedge \phi_2$ implies ϕ_3 , and S_3 can be proved from S_1 and S_2 . Furthermore, it can be proved that if c does not contain function invocations then proof obligations are indeed equivalent. \square

Lemma 5 (global variable interference). Suppose we have an assertion ϕ and c is the function body for function f . Suppose also that ϕ does not contain res nor a variable that may be modified by c . Under the hypothesis that the frame condition is verified, we can generate a certificate for $\Gamma_\phi \vdash_{\text{IMP}} \{\phi\} f \{\phi\}$ where for any function literal g , $\Gamma_\phi(g) = (\phi, \phi)$. (Consequently, one of the derivation rules for \vdash_{IMP} is redundant.)

Proof. When we say that a variable is not modified by a statement we mean, generally speaking, that it can be proved that at the end of the execution it will contain the same value. However, since the possible approaches to ensure this condition may differ on simplicity and completeness, we consider two cases:

- A simple to verify approach is to require that only variables declared as modifiable appear in the right hand side of an assignment (or a function call). Under this strong hypothesis, is straightforward to generate a certificate for the validity of ϕ along the whole program c . To this end, it is sufficient to show that for every statement c , if we replace every assertion on c (i.e invariants and postcondition) with ϕ (the resulting command named c'), and compute $(\phi', S) = \text{wp}(c', \phi)$, then $\phi \equiv \phi'$ and S contains proof obligations of the form $\phi \Rightarrow \chi$ with $\chi \equiv \phi$.
- However, if the condition of non-interference for a variable, x for instance, is guaranteed by proving that the statement $x = Z_x$ is valid at both the pre and postcondition, then it may be the case that x is modified and restored later. Formally that means that for any variable y not modifiable by a statement c , $(\phi, S) = \text{wp}(c, y = Z_y, \Gamma_{y=Z_y}(g))$ (where $\Gamma_{y=Z_y}(g) = (y = Z_y, y = Z_y)$) will be such as ϕ is implied by $y = Z_y$ and S contains only valid propositions. However, we cannot assume anything about verification conditions in S , since it contains proofs for intermediate loop invariants that not necessarily imply $y = Z_y$. We proceed instead by renaming ϕ to remove every modifiable variable: $\phi[\frac{Z}{x}]$. We know from the previous case that we can prove that this assertion is preserved along c , deriving the judgment $\Gamma_{\phi[\frac{Z}{x}]} \vdash_{\text{IMP}} \{\phi[\frac{Z}{x}]\} c \{\phi[\frac{Z}{x}]\}$. We can then merge this result with the derivation of $\Gamma_{x=Z} \vdash_{\text{IMP}} \{x = Z\} c \{x = Z\}$ to get $\Gamma \vdash_{\text{IMP}} \{\phi\} c \{\phi\}$.

\square

Corollary 1 (base code judgment preservation). Suppose f is a function in an advice-free program and we have a certificate for $\Gamma \vdash_{\text{IMP}} \{P\} f \{Q\}$, then we can generate a certificate for $\Gamma \vdash_{\text{RTL}} \{P\} f \{Q\}$.

Proof. Since we have proved that compilation from IMP to RTL preserves wp modulo equivalence, and that some of the rules to derive \vdash_{IMP} are redundant, we can conclude that \vdash_{IMP} implies \vdash_{RTL} . \square

Corollary 2 (before and after advice code preservation of proof obligations). *Suppose a is a before or after advice (it does not contain proceed statements) and we have a certificate for $\Gamma_a \vdash_{\text{ADV}} \{P_a\} a \{Q_a\}$, then we can generate a certificate for $\Gamma_a \vdash_{\text{RTL}} \{P_a\} a \{Q_a\}$.*

Suppose that for a given method m , we have computed the weaving representation θ_m . We can show that for every auxiliary function representing a sub-term of θ_m , we can generate a certificate that it satisfies the specification inferred by using the rules for the weaving representations. Since we have defined the body of each auxiliary function by compiling a particular statement we can concentrate on a high level version and rely on the PPO for standard statements.

Lemma 6 (embedding a simple imperative program in an aspect oriented context). *Suppose we have inferred the judgment $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{P\} m \{Q\}$, then we can generate a certificate for $\Gamma_\Theta \cup \Gamma_a \vdash_{\text{RTL}} \{P\} m \{Q\}$.*

Proof. To derive $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{P\} m \{Q\}$ we need the premise $\Gamma \vdash_{\text{IMP}} \{P\} m \{Q\}$ and that Γ_Θ is a *refinement* for Γ (that implies that $W_{\theta_m} \cap \text{Orig} \subseteq W_m$ where Orig is the set of original program variables). By Corollary 1 we know that we can certify $\Gamma \vdash_{\text{RTL}} \{P\} m \{Q\}$. To transform this certificate in a certificate for $\Gamma_\Theta \vdash_{\text{RTL}} \{P\} m \{Q\}$, we can show that wp_{ins} is a monotone function on the context. That means that if ϕ_1 is stronger than ϕ_2 and Γ_2 is a refinement of Γ_1 then if we compute $(\phi'_1, S_1) = \text{wp}_{\text{ins}}(l, \phi_1, \Gamma_1)$ and $(\phi'_2, S_2) = \text{wp}_{\text{ins}}(l, \phi_2, \Gamma_2)$, then ϕ'_1 will be stronger than ϕ_2 and S_2 will contain weaker proof obligations than S_1 . This property is standard and can be proved by induction on l (the induction principle comes from ins being the result of compiling a command c .) We will pay attention on the function call case. If $\text{ins}[l] = x := \text{invoke } g \ w$ then $\text{wp}_{\text{ins}}(l, \phi_1, \Gamma_1) = (\phi'_1, S_1)$, with $\phi'_1 = P_g^2[w/\text{in}] \wedge \forall \text{res}, y'. Q_g^2[w/\text{in}][y'/y][y/y^*] \Rightarrow \phi_1''[y'/y][\text{res}/x]$ and $(\phi'_1, S_1) = \text{wp}(\text{succ}(l), \phi_1)$. In the other hand if we compute $\text{wp}_{\text{ins}}(l, \phi_2, \Gamma_2)$ we get (ϕ'_2, S_2) where ϕ'_2 is equal to the proposition $P_g^2[w/\text{in}] \wedge \forall \text{res}, z'. Q_g^2[w/\text{in}][z'/z][z/z^*] \Rightarrow \phi_2''[z'/z][\text{res}/x]$ and $(\phi'_2, S_2) = \text{wp}(\text{succ}(l), \phi_2)$. Then, by inductive hypothesis, we have that S_2 is provable from S_1 and that ϕ'_1 is stronger than ϕ'_2 . This latter condition, together with P_g^1 and Q_g^2 being respectively stronger than P_g^2 and Q_g^1 , and that modified variables represented by y includes the ones represented by z , we have that ϕ'_2 is weaker than ϕ'_1 . \square

Lemma 7 (translation for before weaving certificates). *Suppose we have inferred $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \triangleright \theta \{Q\}$, then we can generate a certificate for $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P\} f_{a \triangleright \theta} \{Q\}$.*

Proof. If we have derived the judgment $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \triangleright \theta \{Q\}$, then we have as hypothesis:

1. $\Gamma_a \vdash_{\text{ADV}} \{P_a\} a \{Q_a\}$,
2. $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P_\theta\} \theta \{Q_\theta\}$,
3. $(Q_\theta[y'/y][y/y^*] \wedge Q_a[\text{in}\theta/\text{res}] \Rightarrow Q[y'/y][w'/w][z/z^*])$,
4. $P[\text{in}\theta/\text{in}] \Rightarrow (P_a \wedge Q_a[w'/w][w/w^*] \Rightarrow P_\theta[\text{res}/\text{in}\theta][w'/w])$.

From $\Gamma_a \vdash_{\text{ADV}} \{P_a\} a \{Q_a\}$ and Corollary 2 we can certify the judgment $\Gamma_a \vdash_{\text{RTL}} \{P_a\} a \{Q_a\}$ and by inductive hypothesis $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P_\theta\} \theta \{Q_\theta\}$.

To complete the proof we need to specify that f_θ preserves Q_a modulo modified variables. To this end, we rely on the introduction of logical variables in the specification of a program. Thus now a pre- and post-condition may refer to a logic variable Z , and $\Gamma' \vdash_{\text{RTL}} \{P(Z)\} c \{Q(Z)\}$ will be interpreted as for

any constant value v , the judgment $\Gamma' \vdash_{\text{RTL}} \{P(v)\} c \{Q(v)\}$ is valid. Now, let Q'_a stand for $Q_a[\text{in}\theta/\text{res}][Z_{y^*}/y^*][Z_{y/y}][Z_{\text{in}\theta/\text{in}\theta}]$. Since Q'_a contains only global variables not modified by F_θ (and in_θ), we can easily generate a certificate for the extended specification $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P_\theta \wedge Q'_a\} \theta \{Q_\theta \wedge Q'_a\}$.

Now we modify the predicate transformer wp for the case of function invocation, so we can profit from the existence of logical variables:

$$\begin{aligned} \text{wp}(x := \text{invoke } g \ v, \phi) = & \\ & (P_g[v/\text{in}_g][e'/z] \wedge \\ & \forall y', \text{res}. P_g[v/\text{in}_g][y'/y][y/y^*][e'/z] \Rightarrow \phi[\text{res}/x][y'/y], S) \end{aligned}$$

where e' is any appropriate expression (with the same type as Z). This makes the VCGen not decidable, unless we insert some annotations around the function call. (Notice that, for soundness, substitutions must be performed in the given order).

The weaving function $f_{a \triangleright \theta}$ is compiled exactly from the code

```
x := a(in);
x := f_theta(x);
return x
```

When computing the weakest precondition for the compilation of the first statement above we get:

$$P_a[\text{in}_a] \wedge \forall \text{res}, w'. (Q_a[\text{in}_a][w'/w][w/w^*] \Rightarrow (\varphi)[\text{res}/x][w'/w]) ,$$

where φ is the weakest precondition for the rest of the body of $f_{a \triangleright \theta}$ (after simplifications):

$$\begin{aligned} \varphi = & P_\theta[y'/\text{in}_\theta] \wedge Q_a[z/\text{res}] \wedge \\ & \forall \text{res}'. (Q_\theta[\text{res}'/\text{res}][z'/\text{in}_\theta][y'/y][y/y^*] \wedge Q_a[z/\text{res}] \Rightarrow Q[\text{res}'/\text{res}][y'/y]) \end{aligned}$$

If we compose it with the rest of the proof obligation, that is, the proposition

$$P \Rightarrow (P_a[\text{in}_a] \wedge \forall \text{res}, w'. (Q_a[\text{in}_a][w'/w][w/w^*] \Rightarrow (\varphi)[\text{res}/x][w'/w]))[z/z^*]$$

where z are the global variables possibly modified both by a and θ , we can see that it is sufficient to require the premises

- $(Q_\theta[y'/y][y/y^*] \wedge Q_a[\text{in}\theta/\text{res}] \Rightarrow Q[y'/y][w'/w][z/z^*])$
- $P[\text{in}\theta/\text{in}] \Rightarrow P_a \wedge (Q_a[w'/w][w/w^*] \Rightarrow P_\theta[\text{res}/\text{in}\theta][w'/w])$

\square

Lemma 8 (translation for after weaving certificates).

$\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \triangleleft a \{Q\}$ implies $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P\} f_{\theta \triangleleft a} \{Q\}$

Proof. The proof for *after* advices is symmetrical to the one for *before* advices \square

Lemma 9 (translation for around weaving certificates).

$\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \bowtie \theta \{Q\}$ implies $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P\} f_{a \bowtie \theta} \{Q\}$

Proof. Suppose we have derived the judgment $\{P\} a \bowtie \theta \{Q\}$ by using the rule

$$\frac{\Gamma_a \vdash_{\text{ADV}} \{P_a\} a \{Q_a\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P_\theta\} \theta \{Q_\theta\} \quad P \Rightarrow P_a \wedge \forall x'. (P_a[x'/x] \Rightarrow P_\theta[\text{in}\theta/\text{in}\theta][x'/x]) \quad Q_\theta[\text{in}\theta/\text{in}\theta][\text{res}'/\text{res}][y^*/y^*] \Rightarrow Q'_a \wedge \forall x'. (Q_a[\text{in}\theta/\text{in}\theta][x'/x] \Rightarrow Q[x'/x]) \quad W_\theta \subseteq W'_a}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \bowtie \theta \{Q\}}$$

then a is a *control-flow preserving* advice. By definition a is such that for any program point, either it is located before or after a proceed statement. That will imply, on the RTL side, that the result of compiling a will be a sequence of instructions such as every instruction occurs either before or after (in the control flow graph) an invocation to f_θ , and no invocation to f_θ can reach an invocation to f_θ .

From $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P_\theta\} \theta \{Q_\theta\}$ and by inductive hypothesis we have a certificate for $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P_\theta\} f_\theta \{Q_\theta\}$.

We proceed by extending the original compiler:

- replacing any intermediate assertion ϕ that occurs before a proceed statement with $\phi \wedge \forall x'. (P'_a[x'/x] \Rightarrow P_\theta[\text{in}'_a/x][x'/x])$,
- replacing any intermediate assertion ϕ that occurs after a proceed statement with $\phi \wedge \forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x])$ and

we can prove inductively that for any subcommand c_a of a , if $(l', \text{ins}) = C_a(l, c_a, f_\theta)$ and ins' is the modified (as explained) variant for ins , and we let (ϕ, S) and (ϕ', S') stand respectively for $\text{wp}(c_a, \psi)$ and $\text{wp}_{\text{ins}'}(l, \psi')$ then S' can be discharged from S , and ϕ' is implied by $\phi \wedge \forall x'. (P'_a[x'/x] \Rightarrow P_\theta[\text{in}'_a/x][x'/x])$ (if l is located before an invocation to f_θ) or $\phi \wedge \forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x])$, otherwise.

Proof. To prove this sub-lemma we can rely on the equivalence of verification conditions between simple commands (that does not contain a proceed) and its RTL compilation result. In addition, since we know that neither $\forall x'. (P'_a[x'/x] \Rightarrow P_\theta[\text{in}'_a/x][x'/x])$ nor $\forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x])$ contain modifiable variables, it is clear that $\text{wp}_{\text{ins}} \equiv \text{wp}_{\text{ins}'}$ and then the conclusion holds for this cases. Therefore, the only cases remaining are the return (since the post-condition is changed) and proceed statements.

- case $c_a = \text{return } e$. Statement c occurs obviously after a proceed statement. Since $\text{wp}(c, \psi) = (Q_a[\text{res}], \emptyset)$ and $\text{wp}(c, \psi) = (Q[\text{res}], \emptyset)$ we can see that $\text{wp}(c, \psi)[\text{in}'_a] \wedge \forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x])$ implies $\text{wp}(c, \psi)$
- case $c_a = w := \text{proceed}(e)$. We focus on the assertion returned by wp , since this statement does not generate proof obligations. In one side we have

$$P'_a[\text{in}'_a] \wedge \forall \text{res}', y'. (Q'_a[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi[y'/y][\text{res}'/w][\text{in}'_a])$$

and we have to prove that this together with the proposition $\forall x'. (P'_a[x'/x] \Rightarrow P_\theta[\text{in}'_a/x][x'/x])$ implies the corresponding assertion $P_\theta[\text{in}'_a] \wedge \forall \text{res}', y'. (Q_\theta[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi'[y'/y][\text{res}'/w])$ for the modified statement. It is clear by definition that $P_\theta[\text{in}'_a]$ is implied by $P'_a[\text{in}'_a] \wedge \forall x'. (P'_a[x'/x] \Rightarrow P_\theta[\text{in}'_a/x][x'/x])$, thus we can focus in proving $Q_\theta[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi'[y'/y][\text{res}'/w]$ taking $Q'_a[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi[y'/y][\text{res}'/w][\text{in}'_a]$ as hypothesis. The former assertion can be proved by taking first the stronger (by Inductive Hypothesis) formula

$$Q_\theta[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow (\phi \wedge \forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x]))[y'/y][\text{res}'/w],$$

which in turn may be split to:

$$Q_\theta[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi[y'/y][\text{res}'/w]$$

and

$$Q_\theta[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow (\forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x]))[y'/y][\text{res}'/w].$$

The former can be discharged since we have a proof for $Q_\theta[\text{in}'_a][\text{res}'/w][y^*/y^*] \Rightarrow Q'_a$ and we already have as hypothesis that $Q'_a[\text{in}'_a][y'/y][y^*/y^*] \Rightarrow \phi[y'/y][\text{res}'/w][\text{in}'_a]$ (notice that in'_a may not occur in ϕ .)

The latter is a rewriting of one of the premises of the applied rule: $Q_\theta[\text{in}'_a][\text{res}'/w] \Rightarrow \forall x'. (Q_a[\text{in}'_a][x'/x] \Rightarrow Q[x'/x])$.

□

To derive $\Gamma \cup \Gamma_a \vdash_{\text{RTL}} \{P\} a \bowtie \theta \{Q\}$ we must prove that every proof obligation in S is valid and that P implies $\phi[x^*/x^*]$, where $(\phi, S) = \text{wp}_{\text{ins}}(1, \text{false})$ and $(\text{ins}, -) = C_a(c_a, 1, f_\theta)$. By hypothesis, since $\Gamma_a \vdash_{\text{ADV}} \{P_a\} a \{Q_a\}$ has been derived, we have as premises that S' contains only valid proof obligations and P_a implies $\phi'[x^*/x^*]$, where $(\phi', S') = \text{wp}_a(c_a, \text{false})$. By previous sub-lemma we know that S is provable from S' and that ϕ is equivalent to $\phi' \wedge \forall x'. (P'_a[x^*/x^*] \Rightarrow P_\theta[\text{in}'_a/x^*][x^*/x^*])$. Therefore by using the premise $P \Rightarrow P_a \wedge \forall x'. (P'_a[x^*/x^*] \Rightarrow P_\theta[\text{in}'_a/x^*][x^*/x^*])$ of the rule for around advices applied, we get a proof for the remaining verification condition $P \Rightarrow \phi[x^*/x^*]$.

□

7. Extensions for dynamic point-cuts descriptors

7.0.1 Extension for Conditional point-cut descriptors

Suppose b is a boolean condition, thus it may be represented in our assertion language. We can extend the language for point-cut descriptors with a boolean condition that is checked on runtime to decide whether an advice is executed or not. Usually when compiling AOP, since this condition is not statically decidable, this residue is ignored at first, and a piece of code that checks for this is attached to the weaved code, and therefore θ is over-approximated. We simulate this approach by attaching a corresponding boolean condition to the constructor for θ : “ $\overset{b}{\triangleright}$ ”, “ $\overset{b}{\triangleleft}$ ” and “ $\overset{b}{\bowtie}$ ”.

$$\frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge b\} a \triangleright \theta \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge \neg b\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \overset{b}{\triangleright} \theta \{Q\}} \quad \frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge b\} \theta \triangleleft a \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge \neg b\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \overset{b}{\triangleleft} a \{Q\}} \quad \frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge b\} a \bowtie \theta \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P \wedge \neg b\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \overset{b}{\bowtie} \theta \{Q\}}$$

Running example: Recall the previous example, we have shown that $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{P_m\} a_2 \triangleright (a_1 \bowtie m) \{Q_{a_1}\}$ is a valid judgment. Since P_m represents the boolean condition $0 \leq i < N$ we can introduce an advice triggered *before* $a_2 \triangleright (a_1 \bowtie m)$ (but under the condition $b = \neg(0 \leq i < N)$) to enforce the precondition. The implementation of this advice will take any possible measure to deal with the initial states that does not satisfy the precondition P_m . For simplicity, we consider the body of this new advice a_3 to be an abort statement, and its specification $P_{a_3} = \text{true}$ and $Q_{a_3} = P_m$. It can be easily shown that the advice satisfies its specification. Using the rule for before advices we can derive the judgment $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{\neg P_m\} a_3 \triangleright a_2 \triangleright (a_1 \bowtie m) \{Q_{a_1}\}$, and together with the judgment $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{P_m\} a_2 \triangleright (a_1 \bowtie m) \{Q_{a_1}\}$ we can apply one of the conditional rules to derive the more refined judgment $\Gamma_\Theta, \Gamma_a \vdash_{\text{AOP}} \{\text{true}\} a_3 \overset{b}{\triangleright} a_2 \triangleright (a_1 \bowtie m) \{Q_{a_1}\}$, which states that original functionality is indeed preserved.

Extending the definition for the semantics relation \uparrow to consider this case is straightforward. Compiling the augmented construction $a \overset{b}{\triangleright} \theta$ is similar to $a \triangleright \theta$, but with the call to function a executed under a conditional statement that checks for the condition b . Therefore, under this simple definition, it is not difficult to see the the rules given above permits to translate the certificate of an augmented method to a certificate for its corresponding RTL representation.

7.0.2 Extension for cflow point-cut descriptors (cflow considered harmful)

We proceed by defining a set of rules to deal with cflow point-cut descriptors. However, in this case we omit the function name and

consider cflow as a random choice. This presents the fact that, since a priori we cannot associate each cflow declaration to a condition specifiable in our logic, we are not able to analyze them statically. When defining the semantics of this weaving with residue, we may (and certainly have to) extend execution states to include a call stack, so that we can decide whether a cflow condition is valid. However, specifying and reasoning about a call stack will generate huge and discouraging proof obligations.

$$\begin{array}{c}
\frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \triangleright \theta \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \overset{\text{cflow}}{\triangleright} \theta \{Q\}} \\
\frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \triangleleft a \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \overset{\text{cflow}}{\triangleleft} a \{Q\}} \\
\frac{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \bowtie \theta \{Q\} \quad \Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \overset{\text{cflow}}{\bowtie} \theta \{Q\}}
\end{array}$$

The simplicity of this rules comes with the cost of incompleteness, but that is not surprising considering the harmfulness of cflow declaration.

However, it can be easily and modularly with non-interfering advices. To illustrate this, if a is an around advice that does not modify any variable (but is *control-flow preserving*), with a trivial specification, we can derive $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \bowtie \theta \{Q\}$ from $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} \theta \{Q\}$. And then, by applying one of the rules above, we get $\Gamma, \Gamma_a \vdash_{\text{AOP}} \{P\} a \overset{\text{cflow}}{\bowtie} \theta \{Q\}$.

References

- [1] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from <http://eclipse.org/aspectj>, 2006.
- [2] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.
- [3] Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In Theodosios Dimitrakos, Fabio Martinelli, Peter Y. A. Ryan, and Steve A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 3866 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.
- [4] C. Clifton and G. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning, 2002.
- [5] Daniel S. Dantas and David Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM Press.
- [6] Rmi Douence, Pascal Fradet, and Mario Sdholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Aspect-Oriented Software Development (AOSD)*, pages 141–150. ACM, ACM Press, 2004.
- [7] M Goldman and Shmuel Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect Languages Workshop (FOAL06)*, 2006.
- [8] Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer, 2006.
- [9] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.
- [10] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press.
- [11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from <http://www.jmlspecs.org>, February 2007.
- [12] Mariela Pavlova. *Java bytecode verification and its applications*. Thèse de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.
- [13] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.
- [14] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.
- [15] Jianjun Zhao and Martin C. Rinard. Pipa: A behavioral interface specification language for aspectj. In Mauro Pezzè, editor, *FASE*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2003.