Modular Verification and Certificate Translation for Advice Weaving

César Kunz
INRIA Sophia-Antipolis

Abstract

Aspect oriented programming (AOP) is a paradigm that offers a significant degree of modularity, allowing developers to separate cross-cutting aspects of a system from its main functionality.

While this kind of programming modularity is appropriate to encapsulate concerns into single modules, namely aspects, program development may be highly error-prone due to the level of interference between aspects and the original code. Indeed, in order to take advantages of AOP modularity avoiding the harm of uncontrolled interference, verification techniques need to be developed. In this paper, we present a modular verification technique to certify that a program augmented by the introduction of aspects preserves its original specification.

Furthermore we define a mechanism to transform certificates for correctness of AOP programs into certificates for compiled weaved code, in the spirit of proof carrying code architectures. This mechanism inherits the modularity of the verification technique and allows to build a certificate for an augmented code from the certificates of its components.

1. Introduction

Aspect-oriented programming (AOP) is an emerging paradigm that offers programmers a new potential for the modularization of programs, by allowing developers to isolate cross-cutting aspects of the software from its main functionality.

This paradigm is commonly implemented as an extension to an already existing language. Consequently, an aspect-oriented program typically comprises three parts: a baseline program that performs the main functionality, a set of advices, i.e., computational units handling different aspects; and point-cuts descriptors that determine how advices are combined with the underlying base program.

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 object, through a weaving mechanism that modifies the code of each function depending on the advices that operate over it. This transparency allows to develop the main functionality being unaware of the rest of the components, a key advantage of AOP. However, while *obliviousness* is a very desirable property, it may come at the cost of an unclear interference between each component.

Despite recent efforts to pinpoint the semantics of aspects, the verification of aspect-oriented programs is not well developed. 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 advices are harmless, in that "they may change the termination behavior but do not influence the final result of the mainline code."

A first contribution of this work is the development 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 specifications are extracted from specifications of functions and advices. This is done following a rely-guarantee principle, in that the pre and postconditions of the original function are proved to be preserved (or refined) under the hypothesis that each advice is guaranteed to follow its specification. Discharging these proof obligations automatically may be feasible, depending on the level of interference between advices and original program.

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 with a certificate translator. This mechanism generates, from the proof of correctness for a source AOP program, a certificate that the code resulting from the compilation satisfies the intended specification. 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.

Related work

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 [5] define the notion of *harmless advice*. 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 underlying 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. This type system can be combined to form part

of our hybrid logic to certify and check that an advice does not interfere with the original global state. The conditions they check are coarser-grained that the properties that may be specified in our framework, but it can be certainly be combined to our hybrid logic to certify 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. It mainly consists of a classification for aspects as *spectators* or *assistants*; the former include aspects that only modify the state space they own and do not alter the control flow. On the other hand, *assistants* can interfere with the original behavior of the program but only if *explicitly accepted* by the original program. In our work, we also rely on a declaration for the level of interference between each method and advice body, and verification for behavior preservation may be more flexible but consequently not decidable.

Shmuel Katz et al. [8, 7] propose a classification of aspects as *spectative*, *regulative* or *invasive*, depending on the level of interference with 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. More concretely it model-checks a state machine representation of the aspect merged with the representation of the underlying program. Following the result of this analysis, a concrete static procedure to classify advices is proposed. This work resembles our VCGen in the sense that favors modularity of the verification process and makes emphasis on the preservation of original properties. However, the main difference when comparing it to our work come from the weaknesses and strengths of model-checking with respect to interactive verification.

JML-based verification: Pipa [15] is proposed as an extension to JML [11] for aspectJ [1], to support specification for aspects invariants, pre- and post-conditions for advices and variable introductions. The main motivation is to transfer the application of current 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 convenience and key ideas of the approach are rigorously explained, some examples are given illustrating the transformation, but the discussion remains informal.

PPO and certificate translation. Certificate translation for a simple AOP language is based mainly on previous work on Preservation of Proof Obligations (PPO). Barthe et al. [3] 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 to that of the source program. Similar results on a wider verification framework are detailed by Pavlova [12], for a significant subset of Java Bytecode.

2. Setting

We start with the definition of the base program, and later extend the syntax to introduce 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, where v stands for any element in the domain of program variables $\mathcal{V}$, and g ranges over the set of predefined method names $\mathcal{F}$. The domain $\mathcal{C}$ of statements is standard and includes 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	$Prog$	::=	$meth^*\ c$
Methods	$meth$	::=	$g\ arg\ c$
Commands	c	::=	$v{:=}g(e) \mid v{:=}e \mid c; c$
		|	$\texttt{return}\ e \mid \texttt{while}\ b\ \texttt{do}\ c \mid \texttt{skip}$
		|	$\texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \mid \texttt{abort}$
integer expressions	e	::=	$n \mid v \mid e\ \mathsf{op}\ e \mid \ldots$
boolean expressions	b	::=	$\mathsf{true} \mid \mathsf{false}$
		|	$e\ \mathsf{rop}\ e \mid \neg b \mid b\ \mathsf{bop}\ b$

Figure 1. Syntax of Base Programs

Advices

The syntax for base programs extended with aspects is presented in Fig. 2. In the figure, a stands for any advice identifier in the set $\mathcal{A}$. A program extended with aspects $AProg$ is composed of

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

Figure 2. Syntax for Aspect Extension

a sequence of modular components ($aspect^*$ in the figure) handling different concerns attached to a standard base program $Prog$, as defined above. Each aspect is implemented by combining of a set of advices, with its corresponding point-cut descriptors to specify when an advice should be executed. Advices are computational units declared similarly to functions, in that they are composed of an identifier a, a formal parameter arg and a command c_a representing its body. They are intended to be executed, as a result of a process called *code weaving* at specific execution points (*joint-points*), that can be specified with point-cut descriptors. Commonly, point-cut descriptors are composed of properties that can be statically checked together with dynamically decidable conditions. The most common characterization of a point-cut is a function call, i.e., the fact that a particular function is invoked. In the syntax, it can be seen that point-cut descriptors may specify that an advice is executed *before*, *after* or *around* the invocation of a particular method g, respectively with the descriptors $\mathsf{before}(g)$, $\mathsf{after}(g)$ and $\mathsf{around}(g)$. In addition, it is also possible to require extra conditions (checkable at run-time) such as whether an arbitrary boolean expression is valid b or whether the current execution occurs under the dynamic control flow of a call to a particular method g. The latter is specified with a $\mathsf{cflow}(g)$ descriptor and can be checked by call stack inspection.

For clarity, we initially focus on point-cut descriptors that allow us to infer statically the exact sequence of advices that execute for a given join-point. That is, advices may be specified to be executed when a particular method is invoked, and we will explain later how we can extend our specification and verification techniques to deal with dynamic conditions.

$$\frac{\lessdot[\![e]\!]^{\sigma}_{\eta}, \eta\gtrdot \overset{\theta_g}{\Uparrow} \lessdot n, \eta'\gtrdot}{\langle x{:=}g(e), (\!|\sigma, \eta|\!)\rangle \rightsquigarrow (\!|\sigma \oplus [x \mapsto n], \eta'|\!)}$$

where θ_g is the static weaving for g

Figure 3. Weaved Function Call Semantics

An advice body c_a, is a command similar to a function body, extended with a new statement `proceed`. The argument passing and returning is explained in following sections, as well as the behavior of the `proceed` command. In addition, as can be seen in Fig. 2, the expressiveness of an advice body is reduced by disallowing calls to base code functions.

2.1 Semantics

In this section we progressively define the semantics for an AOP program as defined in the previous section, starting from a standard semantics for base programs.

We define the semantics for simple imperative statements operationally, i.e, as a relation $\rightsquigarrow$ involving the statement and two execution states. To represent the execution state, we distinguish local stores $\sigma : \Sigma$ from global environments $\eta : H$, which are both represented as mappings from program variables in $\mathcal{V}$ to values ($\mathbb{Z}$). As an extra condition, we require formal parameters to be considered different from common program variables, as they may not appear in the right hand side of an assignment and thus they cannot be modified. Execution states can be classified as intermediate (Δ^I) or final (Δ^F) states, where intermediate states $(\!|\sigma, \eta|\!)$ are composed of a local and a global environment ($\Delta^I = \Sigma \times H$) and final states $\lessdot n, \eta\gtrdot$ consist of a final environment and the return value ($\Delta^F = \mathbb{Z} \times H$). The presence of a final state in the right hand side of the relation indicates that a return command has already been executed.

A final state expresses the fact that a `return` statement has been executed, and we require execution states to reduce always to final states before returning from a function call.

Advice Weaving. If we do not consider boolean conditions or call stack inspection as point-cut descriptors, we are able to infer statically which advices are triggered to assist the execution of g, and in which order. To represent the result of this inference we use the following notation: we denote the result of augmenting any method g with θ_g, which is composed of a single method g, or of an advice a appended *before*, *after* or *around* a smaller augmented method θ'_g respectively denoted by $a \triangleright \theta'_g$, $\theta'_g \triangleleft a$ or $a \bowtie \theta'_g$ (we denote Θ the set of augmented methods.) When introducing a dynamic condition (such as cflow) as a point-cut descriptor, weaving is resolved by inserting at specific points a piece of code that checks at run-time whether the condition is satisfied or not. For simplicity, we 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 standard one to represent the fact that a sequence of advices may be weaved around the original function. The new rule for function calls (defined in Fig. 3) relies on a new relation $\dot{\Uparrow}: \Theta \to \Delta^F \to \Delta^F$ which represents the sequential execution of the components of an augmented method, and is defined in Fig. 4. The relation $\dot{\Uparrow}$ 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 g, while the cases for *before* and *after* augmentation rely on a subset of the rules for $\rightsquigarrow$. The case for *around* advices is a bit more complicated since the execution of the advised function must be explicitly allowed to continue. When an advice a is weaved *before* a method g, execution is yield to g immediately after a returns. In the case of *around* advices, a `proceed` statement signals the permission to continue the execution of g. To define the semantics of commands that may contain `proceed` statements, we introduce a new set of rules for relation $\overset{\theta}{\Rightarrow}: \Theta \to (\mathcal{C} \times \Delta^I) \to \Delta$ (defined in Fig. 5.) It is similar to the small-step semantics for standard commands ($\rightsquigarrow$), but it is more general since takes as parameter the sequence of advices (i.e. θ) to be triggered when a `proceed` statement is executed.

$$\frac{\langle c, (\!|[\mathtt{in} \mapsto n], \eta|\!)\rangle \rightsquigarrow \lessdot n', \eta'\gtrdot}{\lessdot n, \eta\gtrdot \overset{g}{\dot{\Uparrow}} \lessdot n', \eta'\gtrdot}$$

$$\frac{\langle c_a, (\!|[\mathtt{in}_a \mapsto n], \eta|\!)\rangle \rightsquigarrow \lessdot n', \eta'\gtrdot \quad \lessdot n', \eta'\gtrdot \overset{\theta}{\dot{\Uparrow}} \lessdot n'', \eta''\gtrdot}{\lessdot n, \eta\gtrdot \overset{a \triangleright \theta}{\dot{\Uparrow}} \lessdot n'', \eta''\gtrdot}$$

$$\frac{\lessdot n, \eta\gtrdot \overset{\theta}{\dot{\Uparrow}} \lessdot n', \eta'\gtrdot \quad \langle c_a, (\!|[\mathtt{in} \mapsto n'], \eta'|\!)\rangle \rightsquigarrow \lessdot n'', \eta''\gtrdot}{\lessdot n, \eta\gtrdot \overset{\theta \triangleleft a}{\dot{\Uparrow}} \lessdot n'', \eta''\gtrdot}$$

$$\frac{\langle c_a, (\!|[\mathtt{in} \mapsto n], \eta|\!)\rangle \overset{\theta}{\Rightarrow} \lessdot n', \eta'\gtrdot}{\lessdot n, \eta\gtrdot \overset{a \bowtie \theta}{\dot{\Uparrow}} \lessdot n', \eta'\gtrdot}$$

where c is $\mathtt{body}(g)$, c_a is $\mathtt{body}(a)$, and $\dot{\Rightarrow}$ is defined in Fig. 5.

Figure 4. Weaving Semantics

$$\frac{\langle c_1, (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} (\!|\sigma', \eta'|\!) \quad \langle c_2, (\!|\sigma', \eta'|\!)\rangle \overset{\theta}{\Rightarrow} S}{\langle c_1; c_2, (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} S}$$

$$\frac{\langle c_1, (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} \lessdot n, \eta'\gtrdot}{\langle c_1; c_2, (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} \lessdot n, \eta'\gtrdot}$$

$$\frac{\lessdot n, \eta\gtrdot \overset{\theta}{\Uparrow} (n', \eta')}{\langle x{:=}\,\mathtt{proceed}(e), (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} \lessdot \sigma \oplus [x \mapsto n'], \eta'\gtrdot}$$

$$\frac{}{\langle \mathtt{return}\, e, (\!|\sigma, \eta|\!)\rangle \overset{\theta}{\Rightarrow} \lessdot[\![e]\!]^{\sigma}_{\eta}, \eta\gtrdot}$$

Figure 5. Around Advice Statements Semantics (Excerpt)

3. 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 verification process by analyzing the validity of the specification for each method or advice in isolation.

Both methods and advices 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 is represented as an assertion in a first order logic. In a general setting an assertion refers to variables (either global or local), as well as some special-purpose variables. These later variables may include *starred variables* ($v^\star$) which appear in the post-condition and in intermediate assertions and represent the initial value of some global variable v. We do not need to refer to starred version of formal parameters, since we are assuming that they are different from program variables and thus they may not be modified. A postcondition refers to global variables, a special purpose res represented the return value and the formal parameter (denoted $\mathtt{in}_g$ for method g). Global variables may appear on the postcondition with a ($.^\star$) modifier if its value may change during the execution of the function. Preconditions refers to the corresponding formal parameter and any global variable.

$$
\begin{array}{l}
\text{let } \Gamma(f) = (P_f, Q_f, y) \text{ in} \\
\quad \mathsf{wp}_g(\texttt{skip}, \varphi) = (\varphi, \emptyset) \\
\quad \mathsf{wp}_g(x{:=}e, \varphi) = (\varphi[^e\!/_x], \emptyset) \\
\quad \mathsf{wp}_g(c_1;c_2, \varphi) = \\
\qquad \text{let } (\varphi_2, S_2){=}\mathsf{wp}(c_2, \varphi) \text{ in} \\
\qquad \text{let } (\varphi_1, S_1){=}\mathsf{wp}(c_1, \varphi_2) \text{ in} \\
\qquad (\varphi_1, S_1 \cup S_2) \\
\quad \mathsf{wp}_g(\texttt{return}\ e, \varphi) = (\varphi[^e\!/_{\mathsf{res}}], \emptyset) \\
\quad \mathsf{wp}_g(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \varphi) = \\
\qquad \text{let } (\varphi_1, S_1){=}\mathsf{wp}(c_1, \varphi) \text{ in} \\
\qquad \text{let } (\varphi_2, S_2){=}\mathsf{wp}(c_2, \varphi) \text{ in} \\
\qquad (b \Rightarrow \varphi_1 \wedge \neg b \Rightarrow \varphi_2, S_1 \cup S_2) \\
\quad \mathsf{wp}_g(\texttt{while}\ b\ \{Inv\}\ \texttt{do}\ c, \varphi) = \\
\qquad \text{let } (\varphi', S){=}\mathsf{wp}(c, Inv) \text{ in} \\
\qquad (Inv, \{Inv \Rightarrow (b \Rightarrow \varphi' \wedge \neg b \Rightarrow \varphi)\} \cup S) \\
\quad \mathsf{wp}_g(x{:=}f(e), \varphi) = \\
\qquad (P_f[^e\!/_{\mathtt{in}_f}] \wedge \\
\qquad \forall_{y', \mathsf{res}}. Q_f[^e\!/_{\mathtt{in}_f}][^{y'}\!/_y][^y\!/_{y^\star}] \Rightarrow \varphi[^{\mathsf{res}}\!/_x][^{y'}\!/_y], \emptyset)
\end{array}
$$

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

Figure 6. Weakest Precondition Function

The frame conditions specify the set of global variables that may be modified by the body of the method or advice. W_g stands for the set of variables modifiable by method g, and we denote $\models_{\mathsf{w}} g$ writes W_g the fact:

$$\forall_v. \exists_{\eta, \eta'}. \langle \mathsf{body}(g), (\sigma, \eta) \rangle \rightsquigarrow \ll n, \eta' \gg \wedge\, \eta v \neq \eta' v \Rightarrow v \in W_g$$

3.1 Verifying the 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 [5] or even Hoare-logics. We say that this certification is hybrid since proof validation can be performed by independent and possibly different analyses.

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. When verifying a command c, since it may include function invocations, a context Γ is used to specify a pre-condition, a post-condition and the frame condition for any method that may be called by c. For notational convenience, Γ is represented as a map from function identifiers to tuples of the form (P, Q, W) where P and Q stand for the pre and post-condition, and W for the set of global variables that may be modified. In addition, a single variable may stand for the set of modified variables. Therefore, we may write instead $\Gamma(f) = (P, Q, y)$ and the single substitution $[^{y'}\!/_y]$ denotes the simultaneous substitution of every modified variable by a fresh variable.

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 derivation for the judgment $\Gamma \vdash_{\mathsf{w}} g$ writes W_g, that certifies that the execution of g may only modify the variables in W_g. Under this assumption, we focus on the derivation of the judgment $\Gamma \vdash_{\mathsf{IMP}} \{P\} g \{Q\}$.

We start by defining a predicate transformer wp in Fig. 6, that takes a statement c, a post-condition ϕ and a context Γ specifying the behavior of any function that may be called by c. The function wp returns the weakest precondition and a set of verification conditions that ensure the validity of the post-condition taken as parameter. To give an intuition, we say that a base code function g is certified to follow its specification if the wp function returns a set S of valid proof obligations and a proposition that is implied by the pre-condition. The rules for the derivation of judgements over standard methods can be found in Fig. 7.

$$
\frac{(P \Rightarrow \phi[^y\!/_{y^\star}]) \wedge \bigwedge_{\mathsf{PO} \in S} \mathsf{PO} \qquad (\phi, S) = \mathsf{wp}_g(\mathsf{body}(g), Q)}{\Gamma \vdash_{\mathsf{IMP}} \{P\} g \{Q\}}
$$

$$
\frac{\Gamma \vdash_{\mathsf{IMP}} \{P'\} g \{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\Gamma \vdash_{\mathsf{IMP}} \{P\} g \{Q\}}
$$

$$
\frac{\Gamma \vdash_{\mathsf{w}} g \text{ writes } Y \quad Y \cap FV(\varphi) = \emptyset}{\Gamma_\varphi \vdash_{\mathsf{IMP}} \{\varphi\} g \{\varphi\}}
$$

$$
\frac{\Gamma^1 \vdash_{\mathsf{IMP}} \{P\} g \{Q\} \quad \Gamma^2 \vdash_{\mathsf{IMP}} \{P'\} g \{Q'\}}{\Gamma^3 \vdash_{\mathsf{IMP}} \{P \wedge P'\} g \{Q \wedge Q'\}}
$$

where $\forall_f. \Gamma_\varphi(f) = (\varphi, \varphi)$ and
for any function f, $\Gamma^3(f) = (P_1 \wedge P_2, Q_1 \wedge Q_2, W_1 \cup W_2)$
where $(P_1, Q_1, W_1) = \Gamma^1(f)$ and $(P_2, Q_2, W_2) = \Gamma^2(f)$.

Figure 7. Inference rules for $\vdash_{\mathsf{IMP}}$

When verifying mutually recursive functions, special care must be taken when defining a rule that removes elements from the context. To this end, we introduce a notation for expressing the quantification of a judgment over a context. That is, we denote $\Gamma \Vdash \Gamma'$ the fact that for every method f such that $\Gamma'(f) = (P, Q, y)$ we can derive $\Gamma \vdash \{P\} f \{Q\}$ and $\Gamma \vdash_{\mathsf{w}} f$ writes y.

$$\frac{\Gamma \cup \Gamma' \Vdash_{\mathsf{IMP}} \Gamma}{\Gamma' \Vdash_{\mathsf{IMP}} \Gamma}$$

The interpretation of the statement $\models_{\mathsf{IMP}} \{P_g\} g \{Q_g\}$ is that for any values n, n' and environments η, η' such that $[\![P_g[^n\!/_{\mathtt{in}}]]\!] \eta$ and $\ll n, \eta \gg \overset{g}{\Uparrow} \ll n', \eta' \gg$, we have that $[\![Q_g[^{n'}\!/_{\mathsf{res}}]]\!] \eta' [y^\star \mapsto \eta\ y]$. To give an intuition, we will later require that the result of weaving advices to the execution of the function g, simulates the original (simple imperative) behavior of g.

We generalize the previous statement by adding a context as hypothesis, such that $\Gamma \models_{\mathsf{IMP}} \{P_g\} g \{Q_g\}$ generalizes the previous interpretation by taking as assumption that for every specification $\Gamma(f) = (P_f, Q_f, W_f)$ we have $\Gamma \models_{\mathsf{IMP}} \{P_f\} f \{Q_f\}$ and $\Gamma \models_{\mathsf{w}} f$ writes W_f. Soundness of the VCGen implies that $\Gamma \models_{\mathsf{IMP}} \{P_g\} g \{Q_g\}$ whenever $\Gamma \vdash_{\mathsf{IMP}} \{P_g\} g \{Q_g\}$.

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 $m =$ `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 Pr contains a method main that represents any method that may invoke function m.

We extend the original program with the introduction of the standard functions $f_1 =$ `initializeCache`, $f_2 =$ `updateCache` and $f_3 =$ `isAvailable`. We add also two global array variables `available` and `cache`, and suppose they are accessible only by these functions. At the moment, we have not introduced any advice, we are simply providing some basic functionality that will prove useful when introducing new advices. We continue by specifying the method m defined in the previous section with pre and post-conditions

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

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 $\vdash_{\mathsf{IMP}} \{P_m\} m \{Q_m\}$ and $\{P_m\} m \{Q_m\} \vdash_{\mathsf{IMP}} \{P\} \mathsf{main} \{Q\}$, 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:

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

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

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

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

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

3.2 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 VC-Gen 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 $\texttt{in}'_a$ and res'_a that correspond respectively to the input and output value for the execution triggered by `proceed`. The assertion Q'_a refers to the variables $\texttt{in}'_a$ and res'_a, as well as any global variables (possible starred) and $\texttt{in}_a$. Similarly, P'_a include conditions over $\texttt{in}'_a$ as well as $\texttt{in}_a$ and any global variable. In addition, a normal postcondition Q_a for an around advice may refer also to the variable res'_a. We 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:

$$\begin{array}{l} \mathsf{wp}_a(x{:=}\ \texttt{proceed}(e), \phi) = \\ \quad (P'_a[^{e}/_{\texttt{in}'_a}] \\ \qquad \wedge \forall_{y', \mathsf{res}'}.Q'_a[^{e}/_{\texttt{in}'_a}][^{y'}/_{y}][^{y}/_{y^{\star\prime}}] \Rightarrow \phi[^{res'}/_{x}][^{y'}/_{y}][^{e}/_{\texttt{in}'_a}], S) \end{array}$$

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 $\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}} \{P_a\} a \{Q_a\}$ for *before* and *after* advices, and $\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}} \{(P_a, P'_a)\} a \{(Q'_a, Q_a)\}$ for *around* advices.

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 an around advice $a_1 =$ `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

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

Notice that, since a_1 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 $\vdash_{\mathsf{ADV}} \{(P_{a_1}, P'_{a_1})\} a_1 \{(Q'_{a_1}, Q_{a_1})\}$. To this end, it is sufficient to show, as a premise, that the proposition

$$\begin{array}{l} P_{f_3} \wedge \forall_b.(Q_{f_3}[^{b}/_{\mathsf{res}}] \Rightarrow \\ \quad b \Rightarrow Q_m[^{\texttt{cache[i]}}/_{\mathsf{res}}] \wedge \phi \\ \quad \wedge \\ \quad \neg b \Rightarrow P_m \wedge \forall_{\mathsf{res}}.(Q_m \Rightarrow P_{f_2} \wedge \\ \qquad \forall_{\texttt{cache}'}.(Q_{f_2}[^{\texttt{cache}'}/_{\texttt{cache}}][^{\texttt{cache}}/_{\texttt{cache}^{\star}}] \Rightarrow \\ \qquad\quad (Q_m \wedge \phi)[^{\texttt{cache}'}/_{\texttt{cache}}]))) \end{array}$$

is implied by P_{a_1}.

3.3 Verifying the weaved code

To verify a method g included in an augmented program, we need to compute previously the sequence of advices that is executed around it (θ_g). We proceed by deriving an appropriate judgment $\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}} \{P\} \theta_g \{Q\}$. To understand the meaning of this judgment we first define as $\models^{\theta}_{\mathsf{AOP}} \{P\}\theta\{Q\}$ the fact that for any n, η, n' and η' if $\ll n, \eta \gg \Uparrow \ll n', \eta' \gg$ and $[\![P[^{n}/_{\texttt{in}}]]\!]\eta$ then $[\![Q[^{n}/_{\texttt{in}}][^{n'}/_{\mathsf{res}}]]\!]\eta[y^{\star} \mapsto \eta\, y]$. By the judgment $\Gamma, \Gamma_{\mathsf{a}} \models_{\mathsf{AOP}} \{P\}\theta\{Q\}$ we mean $\models_{\mathsf{AOP}} \{P\}\theta\{Q\}$ under the hypothesis that for any f and a such that $\Gamma(f) = (P_f, Q_f)$ and $\Gamma_{\mathsf{a}} = (P_a, Q_a)$, we have respectively that $\models_{\mathsf{IMP}} \{P_f\} f \{Q_f\}$ and $\models_{\mathsf{ADV}} \{P_a\} a \{Q_a\}$.

The derivation of this judgment is defined inductively on the construction of θ and relies 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 ($\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{w}} \theta_g$ writes W) by taking W as the union of the variables modifiable by all the components of θ_g.

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 say that an specification (P_g, Q_g, W) is a refinement of (P'_g, Q'_g, W') if $P'_g \Rightarrow P_g$ and $Q_g \Rightarrow Q'_g$ and the set W' contains W (modulo introduced variables). This latter condition reflects the fact that we only care about the variables that belonged to the original program. Furthermore, a context Γ refines a context Γ', if for any g in the domain of Γ, $\Gamma(g)$ is a refinement for $\Gamma'(g)$. 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.

$$\frac{\Gamma \vdash_{\mathsf{IMP}} \{P\} g \{Q\} \quad \Gamma_{\Theta} \text{ refines } \Gamma}{\Gamma_{\Theta}, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}} \{P\} g \{Q\}}$$

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:

$$\frac{\Vdash_{\mathsf{ADV}}\Gamma_{\mathsf{a}} \quad \Gamma_\Theta \cup \Gamma'_\Theta, \Gamma_{\mathsf{a}} \Vdash_{\mathsf{AOP}}\Gamma_\Theta}{\Gamma'_\Theta \Vdash_{\mathsf{AOP}}\Gamma_\Theta}$$

3.3.1 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 implies also that no subsequent advice is executed, and consequently, any specification refinement until this point is 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:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}}\{(P_a, P'_a)\}a\{Q'_a, Q_a\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P_\theta\}\theta\{Q_\theta\} \\ P \Rightarrow P_a \wedge \forall x'.(P'_a[{}^{x'}/_{x}] \Rightarrow P_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{x'}/_{x}]) \qquad W_\theta \subseteq W'_a \\ Q_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathtt{res}'}/_{\mathtt{res}}][{}^{y^\star{}'}/_{y^\star}] \Rightarrow Q'_a \wedge \forall x'.(Q_a[{}^{\mathtt{in}'_a}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}])\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P\}a \bowtie \theta\{Q\}}$$

where x' represents the global variables potentially modified by a, and W'_a specifies the variables that are allowed to be modified by the execution triggered by the `proceed` statement.
In case that it cannot be checked whether the control flow is preserved, we use this rule instead:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}}\{(P_a, P'_a)\}a\{(Q'_a, Q_a)\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P_\theta\}\theta\{Q_\theta\} \\ P'_a \Rightarrow P_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}] \qquad Q_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathtt{res}'}/_{\mathtt{res}}] \Rightarrow Q'_a \qquad W_\theta \subseteq W'_a\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P_a\}a \bowtie \theta\{Q_a\}}$$

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 functions with different behavior.

Example: To derive $\vdash_{\mathsf{AOP}}\{P_{a_1}\}a_1 \bowtie m\{Q_{a_1}\}$ we proceed by applying the rule for non *control-flow preserving* around advices. This requires the previous derivation of the judgments $\vdash_{\mathsf{AOP}}\{P_m\}m\{Q_m\}$ and $\vdash_{\mathsf{ADV}}\{(P_{a_1}, P'_{a_1})\}a_1\{(Q'_{a_1}, Q_{a_1})\}$ 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}.

3.3.2 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 θ that appears on the new postcondition Q.

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}}\{P_a\}a\{Q_a\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P_\theta\}\theta\{Q_\theta\} \\ (Q_\theta[{}^{y'}/_{y}][{}^{y}/_{y^\star}] \wedge Q_a[{}^{\mathtt{in}_\theta}/_{\mathtt{res}}] \Rightarrow Q[{}^{y'}/_{y}])[{}^{w'}/_{w}][{}^{z}/_{z^\star}] \\ P[{}^{\mathtt{in}_a}/_{\mathtt{in}}] \Rightarrow P_a \wedge (Q_a[{}^{w'}/_{w}][{}^{w}/_{w^\star}] \Rightarrow P_\theta[{}^{\mathtt{res}}/_{\mathtt{in}_\theta}][{}^{w'}/_{w}])\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P\}a \triangleright \theta\{Q\}}$$

e	::=	$n \mid x$	constant | variable
$instr$	::=	$\mathsf{nop} \mid \mathsf{assert}\ \phi$	
		$\mid x := e \mid x := e\ op\ e$	assignment
		$\mid \mathsf{jmp}\ l \mid \mathsf{jmpif}\ e\ cmp\ e,\ l$	jump
		$\mid x{:=}\mathsf{invoke}\ g\ e \mid \mathsf{return}\ e$	invocation and return

Figure 8. INSTRUCTION SET

(it is important to mention that if v is a variable, then v' must always be a fresh variable. That means that even when $w = y$ we require that $w' \neq y'$).

Example: The judgment $\vdash_{\mathsf{AOP}}\{P_{a_1}\}a_1 \bowtie m\{Q_{a_1}\}$, derived previously, requires a pre-condition that is stronger than the original one (P_m) (recall that P_{a_1} is defined as $P_m \wedge \phi$.) For this reason, we initially proposed to extend the specification for $main$ by deriving $\{P_m \wedge \phi\}m\{Q_m \wedge \phi\} \vdash_{\mathsf{IMP}}\{P \wedge \phi\}\mathsf{main}\{Q \wedge \phi\}$. To prove preservation of the specification for main we need to weaken the precondition by introducing a *before* advice to ensure the validity of ϕ. We do not need to fully specify the implementation of this new advice a_2 that will simply call the function initializeCache. We will instead suppose that it is verified in isolation to satisfy the judgment $\vdash_{\mathsf{ADV}}\{\mathsf{true}\}a_2\{\phi\}$ and that it does not modify any variable that may occur in P. Under this assumptions, we can apply the derivation rule for *before* advices with the premises $P \Rightarrow \mathsf{true} \wedge (\phi \Rightarrow P \wedge \phi)$ and $Q \wedge \phi \wedge \phi \Rightarrow Q$ to get the judgment $\{P_m \wedge \phi\}m\{Q_m \wedge \phi\} \vdash_{\mathsf{AOP}}\{P\}a_2 \triangleright \mathsf{main}\{Q\}$.

3.3.3 After advices.

The case for weaving an after advice is symmetric to the previous one. Under the same hypothesis over w, y and z we have the following rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}}\{P_a\}a\{Q_a\} \qquad \Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P_\theta\}\theta\{Q_\theta\} \\ (Q_a[{}^{w'}/_{w}][{}^{w}/_{w^\star}] \wedge Q_\theta[{}^{\mathtt{in}_a}/_{\mathtt{res}}] \Rightarrow Q[{}^{w'}/_{w}])[{}^{y'}/_{y}][{}^{z}/_{z^\star}] \\ P[{}^{\mathtt{in}}/_{\mathtt{in}_\theta}] \Rightarrow P_\theta \wedge (Q_\theta[{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow P_a[{}^{\mathtt{res}}/_{\mathtt{in}_a}][{}^{y'}/_{y}])\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}}\{P\}\theta \triangleleft a\{Q\}}$$

4. Target Setting

In this section our target PCC architecture is defined. That is, we present an appropriate lower level language, without aspects, together with a logical framework to verify input/output specifications for procedures. We also define a compiler from our simple AOP language to the lower level language. In the followings sections we study the relation of the verification conditions on the source side with the verification conditions on the target side.

The target language is a non-structured RTL language, with function calls. The syntax is defined in Fig. 8. For simplicity, variables are defined in the same set $\mathcal{V}$ as before, and functions have a single formal parameter.

4.1 Compiler

A definition of the compiler for standard commands can be found in Fig. 9. It relies on a compiler for expressions and conditional jumps, respectively denoted $\mathsf{C_e}$ and $\mathsf{C_b}$ in the figure. The compiler is standard with the exception of the function call statement. In this case, instead of invoking a procedure g, the invocation refers to a procedure f_{θ_g} representing the result of augmenting g with its corresponding advices.

Compiler for advice commands: The compiler function for advice commands is similar to $\mathsf{C_c}$ but it takes an extra parameter that represents the code that has to be executed when a `proceed` statement is executed.

$$\mathsf{C_a}(l, x{:=}\ \mathtt{proceed}(e), f) = \begin{array}{l}\mathsf{let}\ (l', \mathsf{ins}){=}\mathsf{C^v_e}(l, e)\ \mathsf{in} \\ (l'+1, \mathsf{ins} :: [x{:=}\mathsf{invoke}\ f\ y])\end{array}$$

$$
\begin{array}{l}
\mathsf{C_c}(l, \texttt{skip}) = (l+1, [l : \mathsf{nop}]) \\
\mathsf{C_c}(l, x{:=}e) = \mathsf{C}^x_{\mathsf{e}}(l, e) \\
\mathsf{C_c}(l, c_1{;}c_2) = \quad \text{let } (l_1, \mathsf{ins}_1){=}\mathsf{C_c}(l, c_1) \text{ in} \\
\qquad\qquad\qquad\quad \text{let } (l_2, \mathsf{ins}_2){=}\mathsf{C_c}(l_1, c_2) \text{ in} \\
\qquad\qquad\qquad\quad (l_2, \mathsf{ins}_1 :: \mathsf{ins}_2) \\
\mathsf{C_c}(l, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2) = \\
\qquad \text{let } (l_1, \mathsf{ins}_1){=}\mathsf{C_c}(l+1, c_1) \text{ in} \\
\qquad \text{let } (l_2, \mathsf{ins}_2){=}\mathsf{C_c}(l_1+1, c_2) \text{ in} \\
\qquad \text{let } (l_b, \mathsf{ins}_b){=}\mathsf{C_b}(l_2+1, l+1, l_1+1, b) \text{ in} \\
\qquad (l_b, [l : \mathsf{jmp}\ l_2+1] :: \mathsf{ins}_1 :: [l_1 : \mathsf{jmp}\ l_b] :: \mathsf{ins}_2 \\
\qquad\qquad\qquad\qquad :: [l_2 : \mathsf{jmp}\ l_b] :: \mathsf{ins}_b) \\
\mathsf{C_c}(l, \texttt{while } b \texttt{ do } c) = \\
\quad \text{let } (l_c, \mathsf{ins}_c){=}\mathsf{C_c}(l+2, c) \text{ in} \\
\quad \text{let } (l_b, \mathsf{ins}_b){=}\mathsf{C_b}(l_c+1, l+2, l+1, b) \text{ in} \\
\quad (l_b, [l : \mathsf{jmp}\ l_c] :: [l+1 : \mathsf{jmp}\ l_b] :: \mathsf{ins}_c \\
\qquad\qquad :: [l_c : \mathsf{assert}] :: \mathsf{ins}_b) \\
\mathsf{C_c}(l, x{:=}h(e)) = \quad \text{let } (l', \mathsf{ins}){=}\mathsf{C}^y_{\mathsf{e}}(l, e) \text{ in} \\
\qquad\qquad\qquad\quad (l'+1, \mathsf{ins} :: [l' : x{:=}\mathsf{invoke}\ f_{\theta_h}\ y]) \\
\mathsf{C_c}(l, \texttt{return } e) = \quad \text{let } (l', \mathsf{ins}){=}\mathsf{C}^y_{\mathsf{e}}(l, e) \text{ in} \\
\qquad\qquad\qquad\quad (l'+1, \mathsf{ins} :: [l' : \mathsf{return}\ y])
\end{array}
$$

Figure 9. Compiler for standard commands

$$
\begin{array}{l}
\mathsf{C_w}(g) = \quad \text{let } (\_, \mathsf{ins}) = \mathsf{C_c}(1, body(g)) \text{ in} \\
\qquad\qquad\quad [g \mapsto \mathsf{ins}] \\
\mathsf{C_w}(a \triangleright \theta) = \quad \text{let } (\_, \mathsf{ins}_a){=}\mathsf{C_a}(1, body(a), f_\theta) \text{ in} \\
\qquad\qquad\qquad\quad \text{let } c = \{x{:=}a(\texttt{in}); x{:=}f_\theta(x); \texttt{return } x\} \\
\qquad\qquad\qquad\quad \text{let } (\_, \mathsf{ins}){=}\mathsf{C_c}(1, c) \text{ in} \\
\qquad\qquad\qquad\quad [f_{a \triangleright \theta} \mapsto \mathsf{ins}] \\
\mathsf{C_w}(\theta \triangleleft a) = \quad \text{let } (\_, \mathsf{ins}_a){=}\mathsf{C_a}(1, body(a), f_\theta) \text{ in} \\
\qquad\qquad\qquad\quad \text{let } c = \{x{:=}f_\theta(\texttt{in}); x{:=}a(x); \texttt{return } x\} \\
\qquad\qquad\qquad\quad \text{let } (\_, \mathsf{ins}){=}\mathsf{C_c}(1, c) \text{ in} \\
\qquad\qquad\qquad\quad [f_{\theta \triangleleft a} \mapsto \mathsf{ins}] \\
\mathsf{C_w}(a \bowtie \theta) = \quad \text{let } (\_, \mathsf{ins}_a){=}\mathsf{C_a}(1, body(a), f_\theta) \text{ in} \\
\qquad\qquad\qquad\quad [f_{a \bowtie \theta} \mapsto \mathsf{ins}_a]
\end{array}
$$

Figure 10. Compiler for methods

Compiler for methods: When compiling the whole program, since RTL function calls contains only literal function identifiers, we must infer statically how advices are weaved. Therefore, for any method g we suppose we have already computed the representation of the weaving θ_g, to assign a static function identifier for each component of θ_g.
For instance, if $\theta = a \triangleright \theta'$ we let f_θ points to the compilation of $\theta = a \triangleright \theta'$, that calls first a and then the function $f_{\theta'}$ that points to the compilation of the remaining components. The definition of a compiler fot methods is given in Fig. 10. In addition, when compiling standard commands, the result of compiling a call to method h is a call to function f_{θ_h}, which is in charge of calling the resulting weaved code.

4.2 Verification over the target language

Verification of RTL programs can be defined in terms of a weakest precondition function wp, as long as the sequence of instructions under consideration is *well-annotated* [2, 3, 12]. Intuitively, a sequence of instructions is *well-annotated* if there are not non-annotated cycles in the control-flow (for computability of wp function.) The definition of a VCGen for programs at the RTL level can be found in Fig. 11.

5. Certificate Translation

In this section we study the relation of the verification conditions in the high-level side with its RTL counterpart, showing that the simple compiler defined in previous section is amenable to be extended with a certificate translator. The first result is taken from

$$
\begin{array}{l}
\text{Let} \quad (\mathsf{ins}, l') = \mathsf{C_c}(l, \mathsf{body}(g)), (P_f, Q_f) = \Gamma(f) \text{ and} \\
\qquad (\varphi', S') = \mathsf{wp}_{\mathsf{ins}}(\mathsf{succ}(l), \varphi) \text{ in} \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \bot, \varphi) = (\varphi, \emptyset) \qquad (\mathsf{ins}[l] \text{ is undefined}) \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \mathsf{nop}, \varphi) = (\varphi', S') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \mathsf{assert}\ \psi, \varphi) = (\psi, \{\psi \Rightarrow \varphi'\} \cup S') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : x := e, \varphi) = (\varphi'[{}^{e}/_{x}], S') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : x := e_1\ op\ e_2, \varphi) = (\varphi'[{}^{e_1\ op\ e_2}/_{x}], S') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \mathsf{jmp}\ l', \varphi) = \mathsf{wp}_{\mathsf{ins}}(l', \varphi) \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \mathsf{jmpif}\ e_1\ cmp\ e_2,\ l', \varphi) = \\
\qquad \text{let } (\varphi'', S''){=}\mathsf{wp}_{\mathsf{ins}}(l', \varphi) \text{ in} \\
\qquad (e_1\ cmp\ e_2 \Rightarrow \varphi'' \wedge \neg e_1\ cmp\ e_2 \Rightarrow \varphi', S' \cup S'') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : x{:=}\mathsf{invoke}\ f\ e, \varphi) = \\
\qquad (P_f[{}^{e}/_{\mathtt{in}}] \wedge \forall \mathsf{res}, y'.Q_f[{}^{e}/_{\mathtt{in}}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow \varphi'[{}^{\mathsf{res}}/_{x}][{}^{y'}/_{y}], S') \\
\quad \mathsf{wp}_{\mathsf{ins}}(l : \mathsf{return}\ e, \varphi) = (\varphi[{}^{e}/_{\mathsf{res}}], \emptyset)
\end{array}
$$

$$
\frac{(P_g \Rightarrow \phi[{}^{y}/_{y^\star}]) \wedge \bigwedge_{\mathsf{PO} \in S} \mathsf{PO} \qquad (\phi, S) = \mathsf{wp}_{\mathsf{ins}}(l, Q_g)}{\Gamma \vdash_{\mathsf{RTL}} \{P_g\} g \{Q_g\}}
$$

$$
\frac{\Gamma \cup \Gamma' \Vdash_{\mathsf{RTL}} \Gamma}{\Gamma' \Vdash_{\mathsf{RTL}} \Gamma}
$$

Figure 11. Target VCGen

similar work on preservation of proof obligations [3, 12], and then, auxiliary lemmas about not modifiable variables are stated. Finally a certificate translation is defined for *before*, *after* and *around* weaving of advices.

Lemma 1. *Suppose f_{θ_g} points to the compilation of method g (i.e., no advices have been weaved to g). Let $\mathsf{C_c}(l, c) = (l', \mathsf{ins})$ and let ins' be a sequence of instructions containing ins, i.e. ins is equal to the infix $\mathsf{ins}'[l, \ldots, l')$. If $(\phi', S') = \mathsf{wp}_{\mathsf{ins}'}(l', \phi)$ and $(\phi'', S'') = \mathsf{wp}(c, \phi')$ then $\mathsf{wp}_{\mathsf{ins}'}(l, \phi) \equiv (\phi'', S'' \cup S')$.*

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 2. *The rule*

$$
\frac{\Gamma^1 \vdash_{\mathsf{IMP}} \{P\} g \{Q\} \quad \Gamma^2 \vdash_{\mathsf{IMP}} \{P'\} g \{Q'\}}{\Gamma^3 \vdash_{\mathsf{IMP}} \{P \wedge P'\} g \{Q \wedge Q'\}}
$$

where *for any function f, $\Gamma^3(f) = (P_1 \wedge P_2, Q_1 \wedge Q_2, W_1 \cup W_2)$ where $(P_1, Q_1, W_1) = \Gamma^1(f)$ and $(P_2, Q_2, W_2) = \Gamma^2(f)$.*

is redundant, in the sense that the same result can be obtained by applying the wp *function with a suitable context and a different specification.*

Proof. The proof is by simple structural induction on a command c. It can be proved that for any assertion φ if we compute $(\phi_1, S_1) = \mathsf{wp}(c, \varphi, \Gamma^1)$, $(\phi_2, S_2) = \mathsf{wp}(c, \varphi, \Gamma^2)$ and $(\phi_1, S_3) = \mathsf{wp}(c, \varphi, \Gamma^3)$ 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. □

Lemma 3. *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 $\Gamma_\phi \vdash_{\mathsf{IMP}} \{\phi\} f \{\phi\}$ where for any function literal g, $\Gamma_\phi(g) = (\phi, \phi)$. (Consequently, one of the derivation rules for $\vdash_{\mathsf{IMP}}$ is redundant.)*

Proof. Since the possible approaches to verify which variables are modified by a statement may differ on simplicity and completeness, we consider two cases. Let c be the body of function f:

- 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 $(\phi', S) = \mathsf{wp}(c', \phi)$, 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 the statement c, $(\phi, S) = \mathsf{wp}(c, y = Z_y, \Gamma_{y=Z_y})$ (where the context $\Gamma_{y=Z_y}$ is such that for any g, $\Gamma_{y=Z_y}(g) = (y = Z_y, y = Z_y)$) is 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: $\phi[{}^{Z}/_{x}]$. We know from the previous case that we can prove that this assertion is preserved along c, deriving the judgment $\Gamma_{\phi[{}^{Z}/_{x}]} \vdash_{\mathsf{IMP}} \{\phi[{}^{Z}/_{x}]\} f \{\phi[{}^{Z}/_{x}]\}$. We can then merge this result with the derivation of $\Gamma_{x=Z} \vdash_{\mathsf{IMP}} \{x = Z\} f \{x = Z\}$ to get $\Gamma \vdash_{\mathsf{IMP}} \{\phi\} f \{\phi\}$. □

The following result follows from Lemmas 1, 2 and 3.

Corollary 1 (standard PPO). *Suppose f is a function in an advice-free program and we have a certificate for $\Gamma \vdash_{\mathsf{IMP}} \{P\} f \{Q\}$, then we can generate a certificate for $\Gamma \vdash_{\mathsf{RTL}} \{P\} f \{Q\}$.*

Corollary 2 (*before* and *after* advice code PPO). *Suppose a is an* after *or* before *advice (it does not contain* `proceed` *statements) and we have a certificate for $\Gamma_{\mathsf{a}} \vdash_{\mathsf{ADV}} \{P_a\} a \{Q_a\}$, then we can generate a certificate for $\Gamma_{\mathsf{a}} \vdash_{\mathsf{RTL}} \{P_a\} a \{Q_a\}$.*

Suppose that for a given method g, we have computed the weaving representation θ_g. We can show that for every auxiliary function representing a sub-term of θ_g, 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 4 (embedding a simple imperative program in an aspect oriented context). *A certificate for $\Gamma_\Theta \cup \Gamma_{\mathsf{a}} \vdash_{\mathsf{RTL}} \{P\} f \{Q\}$ can be generated from the derivation of $\Gamma_\Theta, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}} \{P\} f \{Q\}$.*

Proof. To derive $\Gamma_\Theta, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}} \{P\} f \{Q\}$ we need the premise $\Gamma \vdash_{\mathsf{IMP}} \{P\} f \{Q\}$ and that Γ_Θ is a *refinement* for Γ (that implies that $W_{\theta_f} \cap \mathsf{Orig} \subseteq W_f$ where Orig is the set of original program variables). By Corollary 1 we know that we can certify $\Gamma \vdash_{\mathsf{RTL}} \{P\} f \{Q\}$. To transform this certificate in a certificate for $\Gamma_\Theta \vdash_{\mathsf{RTL}} \{P\} f \{Q\}$, we can show that $\mathsf{wp}_{\mathsf{ins}}$ 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 $(\phi_1', S_1) = \mathsf{wp}_{\mathsf{ins}}(l, \phi_1, \Gamma_1)$ and $(\phi_2', S_2) = \mathsf{wp}_{\mathsf{ins}}(l, \phi_2, \Gamma_2)$, then ϕ_1' is stronger than ϕ_2' and S_2 contains 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 show only the function call case. Suppose $\mathsf{ins}[l] = x{:=}\mathsf{invoke}\ g\ w$ and let (P_g^1, Q_g^1) and (P_g^2, Q_g^2) stand respectively for $\Gamma_1(f)$ and $\Gamma_2(f)$. Then $\mathsf{wp}_{\mathsf{ins}}(l, \phi_1, \Gamma_1) = (\phi_1', S_1)$, with $\phi_1' = P_g^2[{}^{w}/_{\mathtt{in}}] \wedge \forall_{\mathsf{res},y'}.Q_g^2[{}^{w}/_{\mathtt{in}}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow \phi_1''[{}^{y'}/_{y}][{}^{\mathsf{res}}/_{x}]$ and $(\phi_1'', S_1) = \mathsf{wp}(\mathsf{succ}(l), \phi_1)$. In the other hand if we compute $\mathsf{wp}_{\mathsf{ins}}(l, \phi_2, \Gamma_2)$ we get (ϕ_2', S_2) where ϕ_2' is equal to the proposition $P_g^2[{}^{w}/_{\mathtt{in}}] \wedge \forall \mathsf{res}, z'.Q_g^2[{}^{w}/_{\mathtt{in}}][{}^{z'}/_{z}][{}^{z}/_{z^\star}] \Rightarrow \phi_2''[{}^{z'}/_{z}][{}^{\mathsf{res}}/_{x}]$ and $(\phi_2'', S_2) = \mathsf{wp}(\mathsf{succ}(l), \phi_2)$. Then, by inductive hypothesis, we have that S_2 is provable from S_1 and that ϕ_1'' is stronger that 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'. □

Lemma 5 (translation for before weaving certificates). *It is possible to generate, from a derivation of $\Gamma, \Gamma_{\mathsf{a}} \vdash_{\mathsf{AOP}} \{P\} a \triangleright \theta \{Q\}$, a certificate for $\Gamma \cup \Gamma_{\mathsf{a}} \vdash_{\mathsf{RTL}} \{P\} f_{a \triangleright \theta} \{Q\}$.*

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

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

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

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

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

$$\begin{array}{l}\mathsf{wp}(x{:=}\mathsf{invoke}\ g\ v, \phi) = \\ \quad (P_g[{}^{v}/_{\mathtt{in}_g}][{}^{e'}/_{Z}] \wedge \\ \quad\quad \forall y', \mathsf{res}.P_g[{}^{v}/_{\mathtt{in}_g}][{}^{y'}/_{y}][{}^{y}/_{y^\star}][{}^{e'}/_{Z}] \Rightarrow \phi[{}^{\mathsf{res}}/_{x}][{}^{y'}/_{y}], S)\end{array}$$

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

$$\begin{array}{l} x{:=}a(\mathtt{in}); \\ x{:=}f_\theta(x); \\ \mathtt{return}\ x \end{array}$$

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

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

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

$$\begin{array}{l}\varphi = P_\theta[{}^{x}/_{\mathtt{in}_\theta}] \wedge Q_a[{}^{x}/_{\mathsf{res}}] \wedge \\ \quad \forall \mathsf{res}'.(Q_\theta[{}^{\mathsf{res}'}/_{\mathsf{res}}][{}^{x}/_{\mathtt{in}_\theta}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \wedge Q_a[{}^{x}/_{\mathsf{res}}] \Rightarrow Q[{}^{\mathsf{res}'}/_{\mathsf{res}}][{}^{y'}/_{y}])\end{array}$$

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

$$\begin{array}{l}P \Rightarrow (P_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}] \wedge \\ \quad \forall_{\mathsf{res}}, w'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{w'}/_{w}][{}^{w}/_{w^\star}] \Rightarrow (\varphi)[{}^{\mathsf{res}}/_{x}][{}^{w'}/_{w}]))[{}^{z}/_{z^\star}]\end{array}$$

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

- $(Q_\theta[{}^{y'}/_{y}][{}^{y}/_{y^\star}] \wedge Q_a[{}^{\mathtt{in}_\theta}/_{\mathsf{res}}] \Rightarrow Q[{}^{y'}/_{y}])[{}^{w'}/_{w}][{}^{z}/_{z^\star}]$
- $P[{}^{\mathtt{in}_a}/_{\mathtt{in}}] \Rightarrow P_a \wedge (Q_a[{}^{w'}/_{w}][{}^{w}/_{w^\star}] \Rightarrow P_\theta[{}^{\mathsf{res}}/_{\mathtt{in}_\theta}][{}^{w'}/_{w}])$ □

Lemma 6 (translation for after weaving certificates).
A certificate for $\Gamma \cup \Gamma_\mathsf{a} \vdash_{\mathsf{RTL}} \{P\} f_{\theta \triangleleft a} \{Q\}$ *can be derived from a certificate of* $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\}\theta \triangleleft a\{Q\}$.

Proof. The proof for the case of advices weaved *after* a function call is symmetrical to the previous case. □

Lemma 7 (translation for around weaving certificates).
A certificate for $\Gamma \cup \Gamma_\mathsf{a} \vdash_{\mathsf{RTL}} \{P\} f_{a \bowtie \theta} \{Q\}$ *can be derived from a certificate of* $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \bowtie \theta\{Q\}$

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

$$\frac{\begin{array}{c}\Gamma_\mathsf{a} \vdash_{\mathsf{ADV}} \{P_a\}a\{Q_a\} \qquad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P_\theta\}\theta\{Q_\theta\} \\ P \Rightarrow P_a \wedge \forall x'.(P'_a[{}^{x'}/_{x}] \Rightarrow P_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{x'}/_{x}]) \\ Q_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathsf{res}'}/_{\mathsf{res}}][{}^{y^{\star\prime}}/_{y^\star}] \Rightarrow Q'_a \wedge \forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}]) \\ W_\theta \subseteq W'_a\end{array}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \bowtie \theta\{Q\}}$$

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 implies, on the RTL side, that the result of compiling a is 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 $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P_\theta\}\theta\{Q_\theta\}$ and by inductive hypothesis we have a certificate for $\Gamma \cup \Gamma_\mathsf{a} \vdash_{\mathsf{RTL}} \{P_\theta\} f_\theta \{Q_\theta\}$.

We proceed by extending the original compiler:

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

we can prove inductively that for any subcommand c_a of a, if $(l', \mathsf{ins}) = \mathsf{C_a}(l, c_a, f_\theta)$ and ins' is the modified (as explained) variant for ins, and we let (ϕ, S) and (ϕ', S') stand respectively for $\mathsf{wp}(c_a, \psi)$ and $\mathsf{wp}_{\mathsf{ins}'}(l, \psi')$ then S' can be discharged from S, and ϕ' is implied by $\phi \wedge \forall x'.(P'_a[{}^{x'}/_{x}] \Rightarrow P_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{x'}/_{x}])$ (if l is located before an invocation to f_θ) or $\phi \wedge \forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}])$, 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 $\forall x'.(P'_a[{}^{x'}/_{x}] \Rightarrow P_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{x'}/_{x}])$ nor $\forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}])$ contain modifiable variables, its is clear that $\mathsf{wp}_{\mathsf{ins}} \equiv \mathsf{wp}_{\mathsf{ins}'}$ 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 = `return` e.
  Statement c occurs obviously after a `proceed` statement. Since $\mathsf{wp}(c, \psi) = (Q_a[{}^{e}/_{\mathsf{res}}], \emptyset)$ and $\mathsf{wp}(c, \psi) = (Q[{}^{e}/_{\mathsf{res}}], \emptyset)$ we can see that $\mathsf{wp}(c, \psi)[{}^{\mathtt{in}}/_{\mathtt{in}_a}] \wedge \forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}])$ implies $\mathsf{wp}(c, \psi)$
- case $c_a = w{:=}\,\mathtt{proceed}(e)$. We focus on the assertion returned by wp, since this statement does not generate proof obligations. In one side we have

$$P'_a[{}^{e}/_{\mathtt{in}'_a}] \wedge \forall \mathsf{res}', y'.(Q'_a[{}^{e}/_{\mathtt{in}'_a}][{}^{y'}/_{y}][{}^{y}/_{y^{\star\prime}}] \Rightarrow \phi[{}^{y'}/_{y}][{}^{\mathsf{res}'}/_{w}][{}^{e}/_{\mathtt{in}'_a}])$$

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

$$\begin{array}{l} Q_\theta[{}^{e}/_{\mathtt{in}_\theta}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow \\ \quad (\phi \wedge \forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}]))[{}^{y'}/_{y}][{}^{\mathsf{res}}/_{w}] \ , \end{array}$$

which in turn may be split to:

$$Q_\theta[{}^{e}/_{\mathtt{in}_\theta}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow \phi[{}^{y'}/_{y}][{}^{\mathsf{res}}/_{w}]$$

and

$$\begin{array}{l} Q_\theta[{}^{e}/_{\mathtt{in}_\theta}][{}^{y'}/_{y}][{}^{y}/_{y^\star}] \Rightarrow \\ \quad (\forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}]))[{}^{y'}/_{y}][{}^{\mathsf{res}}/_{w}] \ . \end{array}$$

The former can be discharged since we have a proof for $Q_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathsf{res}'}/_{\mathsf{res}}][{}^{y^{\star\prime}}/_{y^\star}] \Rightarrow Q'_a$ and we already have as hypothesis that $Q'_a[{}^{e}/_{\mathtt{in}'_a}][{}^{y'}/_{y}][{}^{y}/_{y^{\star\prime}}] \Rightarrow \phi[{}^{y'}/_{y}][{}^{\mathsf{res}'}/_{w}][{}^{e}/_{\mathtt{in}'_a}]$ (notice that $\mathtt{in}'_a$ may not occur in ϕ.)
The latter is a rewriting for one of the premises of the applied rule: $Q_\theta[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathsf{res}'}/_{\mathsf{res}}] \Rightarrow \forall x'.(Q_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow Q[{}^{x'}/_{x}])$. □

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

6. Extensions for dynamic point-cuts descriptors

Extension for Conditional point-cut descriptors

We can extend the language for point-cut descriptors with a boolean condition b that is checked at 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 must be statically over-approximated. We simulate this approach by attaching a corresponding boolean condition to the constructors for θ: "$\overset{b}{\triangleright}$", "$\overset{b}{\triangleleft}$" and "$\overset{b}{\bowtie}$".

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge b\}a \triangleright \theta\{Q\} \qquad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge \neg b\}\theta\{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \overset{b}{\triangleright} \theta\{Q\}}$$

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge b\}\theta \triangleleft a\{Q\} \qquad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge \neg b\}\theta\{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \overset{b}{\triangleleft} a\{Q\}}$$

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge b\}a \bowtie \theta\{Q\} \qquad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P \wedge \neg b\}\theta\{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \overset{b}{\bowtie} \theta\{Q\}}$$

Example: Recall the previous example, we have shown that $\Gamma_\Theta, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P_m\}a_2 \triangleright (a_1 \bowtie m)\{Q_{a_1}\}$ is a valid judgment. Since

P_m represents the boolean condition $0 \leq \mathtt{i} < N$ we can introduce an advice triggered *before* $a_2 \triangleright (a_1 \bowtie m)$ (but under the condition $b = \neg(0 \leq \mathtt{i} < N)$) 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 considerer the body of this new advice a_3 to be an `abort` statement, and its specification $P_{a_3} = \mathsf{true}$ 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 $\Gamma_\Theta, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{\neg P_m\} a_3 \triangleright a_2 \triangleright (a_1 \bowtie m)\{Q_{a_1}\}$, and together with the judgment $\Gamma_\Theta, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P_m\} a_2 \triangleright (a_1 \bowtie m)\{Q_{a_1}\}$ we can apply one of the conditional rules to derive the more refined judgment $\Gamma_\Theta, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{\mathsf{true}\} a_3 \overset{b}{\triangleright} a_2 \triangleright (a_1 \bowtie m)\{Q_{a_1}\}$, which states that original functionality is indeed preserved.

Extending the definition for the semantics relation $\dot{\Uparrow}$ 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.

Extension for cflow **point-cut descriptors**

We may also define a set of rules to deal with cflow point-cut descriptors. As can be seen in the following set of rules, since a priory 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.

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \triangleright \theta \{Q\} \quad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \overset{\mathsf{cflow}}{\triangleright} \theta \{Q\}}$$

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \triangleleft a \{Q\} \quad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \overset{\mathsf{cflow}}{\triangleleft} a \{Q\}}$$

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \bowtie \theta \{Q\} \quad \Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \{Q\}}{\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \overset{\mathsf{cflow}}{\bowtie} \theta \{Q\}}$$

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

However, it can be dealt 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 $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \bowtie \theta \{Q\}$ from $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} \theta \{Q\}$. And then, by applying one of the rules above, we get $\Gamma, \Gamma_\mathsf{a} \vdash_{\mathsf{AOP}} \{P\} a \overset{\mathsf{cflow}}{\bowtie} \theta \{Q\}$.

7. Conclusion

We have shown that it is possible to extend a Hoare-like verification environment to verify that the result of weaving an advice to a standard method preserves the originally intended functionality. We have done this, by showing also that the whole process can be conducted modularly by relying on an earlier verification of the method and advice in isolation.

Assuming that the verification process outputs a representation of the proof (aka. certificate), we have complemented the previous result by extending a simple compiler with a certificate translator. More precisely, we showed that for a given simple compiler, we can define a mechanism that builds a certificate of correctness for the result of augmenting a standard method with advices from the certificates of its components. This modularity condition is desirable in any PCC environment, since it allows to reuse already generated code certificates.

Merging the original certificate with proofs of equivalence of verification conditions may imply a significant growth on the final proof representation. For this reason, one possible direction for further research is defining a more appropriate VCGen or supporting verification with deduction modulo equivalence.

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 Theodosis 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.* Thése 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.