Modular Reasoning in Aspect-Oriented Languages from a Substitution Perspective

Tim Molderez[*] and Dirk Janssens

Ansymo (Antwerp Systems and Software Modelling)
University of Antwerp, Belgium
{tim.molderez,dirk.janssens}@ua.ac.be

Abstract In object-oriented languages, a notion of behavioural subtyping is needed to enable modular reasoning. This is no longer sufficient when such languages are extended with aspects. In general, all aspects need to be inspected in order to understand the behaviour of a single method or proceed call, which complicates reasoning about aspect-oriented programs. In this paper, we present an approach to modular reasoning that consists of two parts. First, the advice substitution principle, based on behavioural subtyping, identifies when it is possible to remain unaware of an advice while preserving modular reasoning. Second, in cases where it is undesired or impossible to be unaware of an advice, a simple specification clause can be used to restore modular reasoning and to become aware of this advice. We show that our approach effectively enables modular reasoning about pre- and postconditions in a minimal aspect-oriented language called ContractAJ. To ensure the approach is used correctly, we also provide a runtime contract enforcement algorithm that is specified in ContractAJ, and implemented in AspectJ.

Keywords: aspect-oriented programming languages, modular reasoning, behavioural subtyping, contract enforcement, design by contract

1 Introduction

Aspect-oriented programming (AOP) languages have introduced powerful mechanisms to modularize crosscutting concerns, as aspects allow for the modification of a program's behaviour in a quantifiable manner. However, aspects represent a double-edged sword: while powerful, the presence of aspects also complicates modular reasoning. Modular reasoning is mainly concerned with the ability to reason about a method call, based on the specifications of the receiver's static type. In an AOP language like AspectJ, modular reasoning is obstructed by the fact that aspects can implicitly alter the behaviour of method calls. In general, all aspects need to be inspected to determine whether or not a method call is affected by an aspect, which goes against modular reasoning.

In this paper, we present an approach to modular reasoning for aspect-oriented languages, without modifying the programming language itself. There

[*] Funded by a doctoral scholarship of the Research Foundation - Flanders (FWO)

are several languages that restrict AOP [2,33,39] in the sense that advice can only apply to join points that have been explicitly exposed by the developer, making it easy to distinguish which method calls may or may not be affected by advice. While this greatly helps in restoring modular reasoning, the advantages of the widely debated quantification and obliviousness [17] properties of AOP are lost to a certain extent.

If all usable join points need to be mentioned explicitly, it becomes less appealing to use aspects for crosscutting concerns where being unaware/oblivious of such concerns is unlikely to cause harm. This includes examples such as logging/tracing, caching, profiling, contract enforcement and various other concerns that provide additional functionality without interfering with the rest of the system. The extra effort needed to make join points explicit only grows as the aspects that implement such concerns rely more extensively on quantification, i.e. as they need to affect more and more locations in the source code. Given this observation, our goal is to enable modular reasoning in a manner that preserves obliviousness for those aspects where it is an advantage, and to become aware of those aspects where obliviousness is a disadvantage.

We will do this from a design by contract [30] perspective, based on a notion of substitution for advice. In object-oriented programming, the developer may not know the receiver's dynamic type when making method calls, so he/she can only take into account the specifications of the static type. Such specifications are defined in terms of preconditions, postconditions and invariants, also commonly referred to as contracts. To prevent any surprising behaviour when making method calls, subtypes should respect the contracts of their ancestors. In other words, they should adhere to a notion of behavioural subtyping [3,16,27]. This allows an instance of a subtype to substitute for any instance of an ancestor type. This paper uses a similar notion of substitution for advice: the advice substitution principle (ASP). This principle essentially states that an advice should comply with the contracts of the join points it advises. The ASP was first introduced informally by Wampler [41, Sec. 3.1.3] as one of several aspect-oriented design principles. We will present the ASP on a more formal level, as one of the two parts that form our approach to modular reasoning in AOP languages.

The second part of our approach focuses on advice that cannot satisfy the ASP. Unlike behavioural subtyping in OOP, we do not use the ASP as a strict rule that should hold for all advice. Instead, the ASP is used to distinguish between the advice that already preserve modular reasoning (e.g. logging, caching, profiling, ...), and the advice where extra effort is needed to restore modular reasoning (e.g. authentication, authorization, transaction management, ...). This extra effort comes in the form of a specification clause called `@advisedBy`. This clause explicitly indicates that a method may be advised by a given sequence of advice, such that this method and its clients become aware of these advice. This approach of using both the ASP and the `@advisedBy` clause forms the paper's main contribution. The approach is presented in the context of a representative, minimal aspect-oriented language called ContractAJ, which is based on ContractJava [18], AspectJ and Eos-U [34]. Within this language, we will show that our approach is sound. That is, the approach effectively preserves modular reasoning of method and proceed calls in ContractAJ, even in the presence of shared join points, overriding advice, higher-order advice (advice that advises advice) and pointcuts depending on runtime information. In addition, we specify an algorithm that performs runtime contract enforcement in ContractAJ. It is able to assign the blame when a contract is broken, taking into account behavioural subtyping, the ASP and the `@advisedBy` clause. To demonstrate an instantiation of this algorithm in a full programming language, it is also implemented as a small design-by-contract library for AspectJ.

In summary, this paper makes the following contributions:

– We first present the syntax and operational semantics of the ContractAJ language. (Sec. 2 and 3)
– We define and discuss the ASP. (Sec. 4)
– For those advice where it is undesired or impossible to preserve obliviousness, we present the `@advisedBy` clause to restore modular reasoning. (Sec. 5)
– We show that the approach preserves modular reasoning about pre- and postconditions in method and proceed calls in ContractAJ. (Sec. 6)
– Finally, we specify a runtime contract enforcement algorithm in ContractAJ. We also discuss its implementation in AspectJ. (Sec. 7)

2 ContractAJ

Before delving into the specifics surrounding the ASP and `@advisedBy` clause, we first introduce the ContractAJ language, which is used throughout the paper to study modular reasoning in the context of aspects. This section presents the motivation behind ContractAJ, its syntax and its informal operational semantics.

2.1 Motivation

There are two main reasons for introducing the ContractAJ language. First, it is a minimal language, which makes it better suited to study modular reasoning at a more formal level. The ContractAJ language is based on the minimal ContractJava language introduced in Findler et al. [18], where it was used to specify a contract enforcement algorithm in an object-oriented setting. In its turn, ContractJava is an extension that adds contracts to the ClassicJava calculus [19].

The second reason to introduce ContractAJ is that we wish to explore AOP in a more flexible and unified form than is present in AspectJ. While AspectJ currently is the most established AOP language, some design decisions were made to achieve better performance or faster language adoption, resulting in a number of more specialised, less flexible language constructs. This includes the distinction between aspects and classes, limited control over instantiating aspects, anonymous advice and the restriction that an aspect cannot extend from a concrete (non-abstract) aspect. We prefer to keep ContractAJ a small and flexible language. Additionally, unifying aspect-oriented concepts with object-oriented ones also helps us to relate modular reasoning in AOP to modular reasoning in OOP. This type of unification is visible in several design choices:

- Like Eos-U [34] and CaesarJ [5], ContractAJ unifies aspects and classes. This means aspects are first-class, and aspects can freely extend other aspects. Pointcut-advice pairs are also named, such that they can be overridden.
- Before and after advice are treated as special cases of around advice, rather than viewing around advice as a combination of before and after advice. Around advice have the closest relation to overriding methods, in the sense that around advice also override the behaviour of methods, and that proceed calls behave in a similar fashion to super calls.
- The execution of advice is specified as an extension of the method lookup mechanism.
- When an advice is about to be executed, the same lookup mechanism is reused to allow for higher-order advice (advice that advises other advice).

2.2 Syntax

The syntax of ContractAJ is shown in Fig. 1. To illustrate most of the syntax's constructs, an example of a simple ContractAJ program is given in Fig. 2. It describes an aspect called `Security` with an around advice named `authenticate`, which is executed at each method call to `Account.withdraw`. What is immediately noticeable is the lack of an `aspect` keyword, indicating the unification of aspects and classes. Like the classpects in Eos-U or Caesar classes in CaesarJ, there is no separate module type dedicated to aspects. Instead, definitions of pointcuts and advice are allowed in regular classes, such that they can effectively serve as aspects. This allows for more flexibility, as the developer regains precise control over the instantiation of aspects by reusing the class instantiation mechanism. Once an aspect is instantiated with the `new` keyword, its pointcuts are active.

As pointcuts and advice are now regular class members, aspects can extend other aspects as well. Note that, for reasons of simplicity, pointcuts and advice are paired. Because these pointcut-advice pairs are named, this enables overriding. That is, if an aspect with an overriding pointcut-advice pair is instantiated, the overriding pointcut-advice pair is active, but the overridden one is not.

The pointcut language, shown in the *pcut* rule, provides most of the basic pointcut constructs: method and advice executions can be captured with `execution`. Method calls are captured with `call`. Like AspectJ, the receiver of method calls / advice executions can be bound to a variable using `this` or `target`. While there is no `args` construct to bind parameters as in AspectJ, method/advice arguments are bound directly in the `execution`/`call` pointcut. Note that our pointcut language also includes an `if` construct. Just like AspectJ, when an `if` construct is used, the pointcut can only match when the given `if`-condition is true at the current join point. We intentionally included this `if` construct to demonstrate that our approach to modular reasoning also takes into account pointcuts that can only be determined at runtime.

The *prec* rule contains the syntax of the advice precedence/composition mechanism; it determines in which order advice should be executed when multiple advice share the same join point. In the example of Fig. 2, the `declare precedence` statement specifies that the advice `Security.authenticate` has a

$$\begin{array}{rcl}
\mathit{program} & ::= & \mathit{prec}\ \mathit{def}^*\ \texttt{main\{}e\texttt{\}} \\
\mathit{prec} & ::= & \texttt{declare precedence}\ (c.a)^*\texttt{;} \\
\mathit{def} & ::= & \texttt{class}\ c\ \texttt{extends}\ c\ \texttt{\{}(\mathit{field} \mid \mathit{method} \mid \mathit{adv})^*\texttt{\}} \\
\mathit{field} & ::= & t\ f \\
\mathit{method} & ::= & \mathit{contract}\ [\texttt{@advisedBy}\ (c.a)^*]\ t\ m\ \texttt{(}\mathit{arg}^*\texttt{)\{}e\texttt{\}} \\
\mathit{adv} & ::= & \mathit{contract}\ (\texttt{before} \mid \texttt{after} \mid \texttt{around})\ a\texttt{:}\ \ \mathit{pcut}\texttt{\{}e\texttt{\}} \\
\mathit{contract} & ::= & \texttt{@requires}\ e\ \texttt{@ensures}\ e \\
\mathit{arg} & ::= & t\ \mathit{var} \\
e & ::= & \texttt{new}\ c \mid \mathit{var} \mid \mathit{bool} \mid \texttt{null} \\
 & & \mid e.f \mid e.f\texttt{=}e \\
 & & \mid e.m\texttt{(}e^*\texttt{)} \mid \texttt{super.}m\texttt{(}e^*\texttt{)} \mid \texttt{proceed(}e^*\texttt{)} \\
 & & \mid \texttt{(}t\texttt{)}\ e \mid e\ \texttt{instanceof}\ t \\
 & & \mid \texttt{let\{}\mathit{binding}^*\texttt{\} in \{}e\texttt{\}} \\
 & & \mid \texttt{if(}e\texttt{)\{}e\texttt{\}else\{}e\texttt{\}} \\
 & & \mid \texttt{error(}e\texttt{)} \\
 & & \mid \texttt{\{}e\ \texttt{;}\ e\texttt{\}} \\
 & & \mid \texttt{proc} \\
\mathit{bool} & ::= & \texttt{true} \mid \texttt{false} \\
\mathit{binding} & ::= & \mathit{var}\texttt{=}e \\
\mathit{pcut} & ::= & \texttt{execution(}t\ c.x\texttt{(}\mathit{arg}^*\texttt{))}\ \texttt{\&\&}\ \texttt{this (}\mathit{var}\texttt{)}[\texttt{\&\& if (}e\texttt{)}] \\
 & & \quad \mid \texttt{call(}t\ c.m\texttt{(}\mathit{arg}^*\texttt{))}\ \texttt{\&\&}\ \texttt{target(}\mathit{var}\texttt{)}[\texttt{\&\& if (}e\texttt{)}] \\
\mathit{var} & ::= & \text{a variable name or } \texttt{this} \\
c & ::= & \text{a class name (or } \texttt{Object}\text{)} \\
f & ::= & \text{a field name} \\
m & ::= & \text{a method name} \\
a & ::= & \text{an advice name} \\
x & ::= & m \mid a \\
t & ::= & c \mid \texttt{boolean}
\end{array}$$

Figure 1. ContractAJ syntax

```
declare precedence Security.authenticate, Logger.write;

class Account extends Object {
    @requires  this.getAmount() >= m && m>0
    @ensures   this.getAmount() == old(this.getAmount()) - m
    @advisedBy Security.authenticate
    int withdraw(int m) {...} ... }

class Security extends Object {
    @requires proc
    @ensures if(isLoggedIn(acc.getOwner)){proc}else{true}
    around authenticate: call(int Account.withdraw(int m)) && target(acc) {
        if (isLoggedIn(acc.getOwner())) {
            proceed(acc,m);
        }
    } ... }
...
main {
    Security sec = new Security; Account acc = new Account;
    acc.withdraw(10); // advised by sec.authenticate
}
```

Figure 2. An example ContractAJ program

higher precedence than `Logger.write` (not shown). ContractAJ's precedence mechanism is similar to that of AspectJ, apart from two small differences: ContractAJ's precedence statement is slightly more fine-grained, as it lists advice rather than aspects. ContractAJ programs also contain only one global declare precedence statement. While AspectJ does allow for multiple precedence declarations, note that they can always combined into one global statement. (Otherwise there would be a precedence conflict.)

The constructs needed to specify contracts are provided in the *contract* rule. Methods and advice can specify their preconditions and postconditions using the `@requires` and `@ensures` constructs. Note that our main focus is on pre- and postconditions, which is why there is no syntax for invariants, history constraints or frame properties. Additionally, the optional `@advisedBy` clause can be used by a method if it should become aware of one or more advice. We should also mention the `proc` keyword in the e rule, which serves as a placeholder for the contracts of any proceed calls. This keyword can only be used in the contracts of advice, to refer to the pre- or postcondition of the next advice we are aware of. The semantics of both `@advisedBy` and `proc` are detailed in Sec. 5.

Finally, note that we will use the symbols for a class (c), field (f), method (m), advice (a), method-or-advice (x) and type (t) as naming conventions throughout the entire paper.

2.3 Informal ContractAJ semantics

The semantics of ContractAJ is an extension of the object-oriented ContractJava [18] language. It is not a pure extension, in the sense that support for interfaces was removed in order to keep the language minimal. What is added semantics-wise can be found mainly in the language's join point model, pointcut language and the lookup procedure. This section describes ContractAJ's join point model, as well as informally explains ContractAJ's lookup procedure.

Join point model The call and execution pointcuts of ContractAJ closely correspond to those of AspectJ: a call pointcut matches on method calls where the receiver's *static* type is, or is a subtype of, whichever type is specified in the pointcut. Similarly, an execution pointcut matches if the receiver's *dynamic* type is (a subtype of) the type specified in the pointcut.

However, what is different in ContractAJ is the join point model. In AspectJ, call pointcuts describe a set of call join points, where a call join point refers to the moment before method lookup. Likewise, execution pointcuts describe a set of execution join points, which refer to the moment after lookup.

In ContractAJ, there only are call join points. Both ContractAJ's call and execution pointcuts make use of this one kind of join point. This is possible as both the receiver's static and dynamic type are available at the moment before method lookup. We made this choice to simplify ContractAJ's semantics, while it still is representative for what is possible when using AspectJ's pointcuts.

Another difference between call and execution join points in AspectJ is that they are associated with different locations in the source code (i.e. join point shadows). However, this difference is only relevant when combining call/execution pointcuts with other pointcut constructs that match on join point shadows (e.g. AspectJ's `within` and `withincode`), which are not present in ContractAJ. Because of this, we argue that it is sufficient to support call join points only.

Lookup semantics The execution of advice in ContractAJ is expressed as an extension of the method lookup mechanism, which implies that advice execution is late-bound. Executing a method call $c.m$ in ContractAJ is done as follows:

1. For all instances of classes with pointcuts, try to match these pointcuts on the $c.m$ call join point. (If there are multiple instances of the same class, the pointcut is checked for each instance.) Note that, when looking for matching pointcuts, we do not yet consider any `if` pointcut constructs.
2. Given all matching pointcuts, the precedence mechanism will produce a composition/sequence $\langle c_1.a_1, c_2.a_2, \ldots, c_n.a_n, c.m\rangle$, where each $c_i.a_i$ represents the advice body associated with a matching pointcut. This composition determines the precedence order of the advice that advise $c.m$. Push this composition on a global stack.
3. Find the first advice in the composition where its corresponding pointcut either does not contain an `if` construct, or the `if` construct's condition evaluates to `true`.
   (a) If no such advice is found, pop the entire composition from the stack, perform method lookup on $c.m$ and execute the body that is found.
   (b) If an advice was found, remove this advice and all preceding advice from the composition, then *call* the advice that was found (as if it were a method call).

Note the emphasis on "call" in step 3.(b). This enables the use of higher-order advice, which are advice that match on other advice executions. A call to an advice is handled just like a method call, meaning that it reuses the same lookup mechanism.

Finally, the semantics of a proceed call is a simpler version of the above steps: As the desired composition already is on the stack whenever a proceed call is made, it only performs step 3. As this step includes testing `if` pointcut constructs, this implies that these tests are delayed until an advice body is about to be executed, which corresponds to AspectJ's semantics.

3 Formal ContractAJ semantics

This section presents the operational semantics of the ContractAJ language in its entirety. The semantics follows a similar style as the ContractJava [18] language it is based on, making use of a contextual rewriting system [42].

$$\begin{array}{c} P, c \vdash a\texttt{: call(}t\ c'\texttt{.}m\texttt{(}t_1\ x_1\ \ldots\ t_m\ x_m\texttt{)) \&\& target(}var\texttt{)}\ldots\ \texttt{\{}e\texttt{\}} \\ \rightharpoonup_{\mathsf{sep}}\ a\texttt{: call(}t\ c'\texttt{.}m\texttt{(}t_1\ x_1\ \ldots\ t_m\ x_m\texttt{)) \&\& target(}var\texttt{)}\ \ldots \\ t\ a\texttt{(}c'\ var, t_1\ x_1\ \ldots\ t_m\ x_m\texttt{)\{}e\texttt{\}} \end{array}$$

Figure 3. Moving advice bodies to method bodies

$$\frac{\begin{array}{c} m \in c'' \text{ and } c' \leq c'' \\ \nexists c''' : m \in c''' \text{ and } c'' < c''' \end{array}}{\begin{array}{c} P, c \vdash a\texttt{: execution(}t\ c'\texttt{.}x\texttt{(}t_1\ x_1\ \ldots\ t_m\ x_m\texttt{)) \&\& this(}var\texttt{)} \\ \rightharpoonup_{\mathsf{exec}}\ a\texttt{: call(}t\ c''\texttt{.}x\texttt{(}t_1\ x_1\ \ldots\ t_m\ x_m\texttt{)) \&\& target(}var\texttt{) \&\& if(}var\ \texttt{instanceof}\ c'\texttt{)} \end{array}}$$

Figure 4. Converting execution into call pointcuts

3.1 Source modifications

Before describing the semantics of ContractAJ, we will first perform three small, harmless transformations at the source code level, which make it easier to describe the operational semantics.

The first transformation, defined by the $\rightharpoonup_{\mathsf{sep}}$ judgement[1] in Fig. 3, consists of removing the advice body from each pointcut-advice pair, and moving this body into a separate method declaration. This method declaration gets the same name and the same parameters as the corresponding pointcut-advice pair. If a pointcut now matches during program execution, ContractAJ's semantics can simply call the method that corresponds to the advice. After applying this transformation, ContractAJ essentially is a minimal version of the Eos-U [34] language, which unifies classes and aspects, and uses regular method bodies as advice bodies.

```
class Logger {
    before log: execution(void Duck.fly(int dist)) && this(duck) {...}}
```

After applying the $\rightharpoonup_{\mathsf{sep}}$ and $\rightharpoonup_{\mathsf{exec}}$ judgements:

```
class Logger {
    before log: call(void Bird.fly(int dist)) && target(duck)
        && if(duck instanceof Duck)
    void log(Duck duck, int dist) {...}}
```

Figure 5. Example application of the first two transformations

In the second transformation, we convert every[2] execution pointcut into an equivalent call pointcut. This transformation is defined by the $\rightharpoonup_{\mathsf{exec}}$ judgement[3]

[1] An analogous definition of $\rightharpoonup_{\mathsf{sep}}$ can be given for advice with an execution pointcut.

[2] Note that execution pointcuts matching on advice executions are also converted into call pointcuts, which is accepted by ContractAJ's operational semantics as it reuses the dynamic type as static type in advice executions.

[3] An analogous definition of $\rightharpoonup_{\mathsf{exec}}$ can be given in case an `if` construct is already present in the execution pointcut.

in Fig. 4. ($<$ is the strict subtyping relation, such that $c < c'$ relates c to an ancestor class c'. $\in$ relates a method to its class.) As discussed in Sec. 2.3, the only difference between ContractAJ's call and execution pointcuts is that the first matches on the receiver's static type whereas the latter matches on the dynamic type. When converting an execution pointcut into a call pointcut, $\rightharpoonup_{\mathsf{exec}}$ ensures that the dynamic type still is taken into account by adding an if pointcut construct with a simple `instanceof` test. Note that, even though the call pointcut construct also tests the receiver's static type, we made sure that this test has no effect: The call construct tests that the receiver is an instance of the class where the desired method is first declared, which always is the case when the dynamic type test passes. An example application of both $\rightharpoonup_{\mathsf{exec}}$ and $\rightharpoonup_{\mathsf{sep}}$ can be found in Fig. 5 (where `Bird` declares method `fly`, and `Duck` is a subtype of `Bird`).

$$\begin{array}{ll} e ::= & \ldots \mid e\!:\!c.f \mid e\!:\!c.f\texttt{=}e \\ & \mid e\!:\!c.m\texttt{(}e^*\texttt{)} \\ & \mid \texttt{super} \equiv \texttt{this:}c.m\texttt{(}e^*\texttt{)} \mid \ldots \end{array} .$$

Figure 6. Syntax modifications

In the third and final transformation, we will modify the syntax of method calls, super calls, field accesses and field assignments in such a way that the static type is always included explicitly, so we can easily refer to it when needed. For example, a method call $e.m\texttt{(}e*\texttt{)}$ now becomes $e\!:\!c.m\texttt{(}e*\texttt{)}$, where c is the static type of the receiver. We assume a type checker can be easily implemented which infers the static type for each of these statements. The altered syntax for these statements is shown in the e rule of Fig. 6.

3.2 Operational semantics

The operational semantics of ContractAJ, like the ContractJava language, is expressed as a contextual rewriting system [42]. Our rewriting system operates on triples consisting of an expression, a store and a stack. That is, each evaluation rule has the following shape:

$$P \vdash \langle e,\, \mathcal{S},\, \mathcal{J} \rangle \hookrightarrow \langle e,\, \mathcal{S},\, \mathcal{J} \rangle$$

Such a rule can be read as: within program P, the left-hand-side (a triple with expression e, store $\mathcal{S}$ and join point stack $\mathcal{J}$) evaluates to the right-hand-side if the rule is applied. Each of the different data structures used in these triples is defined as follows:

P A program, as defined by the ContractAJ syntax.

e Each e is an expression, as defined by the syntax.

$\mathcal{S}$ $\mathcal{S} ::= obj \mapsto \langle c, \mathcal{F} \rangle$
$\mathcal{F} ::= f \mapsto v$

The store $\mathcal{S}$ allows us to find the field values of each object: it is a mapping from objects to $\langle c, \mathcal{F} \rangle$ pairs, where each pair consists of a class c and a field record $\mathcal{F}$. A field record $\mathcal{F}$ contains the values for all fields in a particular object: it is a mapping from field names to field values.

$\mathcal{J}$ $\mathcal{J} ::= \mathcal{A}; \mathcal{J} \mid \bullet$
$\mathcal{A} ::= \mathcal{E} + \mathcal{A} \mid \bullet$
$\mathcal{E} ::= \langle c, x, obj, bool \rangle$

The join point stack $\mathcal{J}$ keeps track of the sequence of advice/methods that should be executed at each join point that is encountered. The join point stack is a stack of $\mathcal{A}$ records. In turn, each $\mathcal{A}$ record is a stack[4] of $\mathcal{E}$ tuples $\langle c, x, obj, bool \rangle$. Such a tuple respectively describes a method/advice body $c.x$, the `this` object to be used and a boolean value that indicates whether $c.x$ is ready to be executed (`true`), or further lookup is needed (`false`). The ordering of $\mathcal{E}$ tuples within an $\mathcal{A}$ record will be determined by ContractAJ's precedence mechanism.

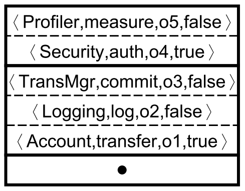
⟨ Profiler,measure,o5,false ⟩
⟨ Security,auth,o4,true ⟩
⟨ TransMgr,commit,o3,false ⟩
⟨ Logging,log,o2,false ⟩
⟨ Account,transfer,o1,true ⟩
•

Figure 7. Join point stack

The example in Fig. 7 gives a more visual idea of the join point stack's structure, in the context of a banking application: In this example, the stack contains three $\mathcal{A}$ records. The method `Account.transfer` was called at some point and `Security.auth`, `TransMgr.commit` and `Logging.log` respectively matched on this method call. The `Security.auth` advice was already moved into topmost $\mathcal{A}$ record to look for higher-order advice. One higher-order advice `Profiler.measure` was found. The boolean in its $\mathcal{E}$ tuple still is `false`, indicating that we are about to check whether any higher-order advice match on `Profiler.measure`.

In addition to the store and the join point stack, we also provide a number of predicates and functions in Fig. 8 that help define the operational semantics. Most of these are self-explanatory, but we will highlight the precedence and lookup mechanisms in more detail:

The $<^{\mathsf{prec}}$ predicate defines ContractAJ's precedence mechanism, which makes use of the global `declare precedence` statement to determine the ordering of advice when multiple pointcuts match at the same join point. The $<^{\mathsf{prec}}$ predicate is defined in terms of $precLook(c, a)$, which determines the element in the `declare precedence` statement that corresponds to a particular advice $c.a$, either directly

[4] Note that we use two different concatenation symbols to avoid ambiguity. ";" concatenates records in $\mathcal{J}$ and "+" concatenates tuples in an $\mathcal{A}$ record.

Subtyping relations

$$\begin{array}{ll} \prec & c \prec c' \Leftrightarrow \texttt{class } c \texttt{ extends } c'\texttt{\{}\ldots\texttt{\}} \text{ is in } P \\ \leq & \leq\equiv \text{transitive, reflexive closure of } \prec \\ < & <\equiv \text{transitive, irreflexive closure of } \prec \end{array}$$

Field f is a member of c

$$\begin{array}{ll} \in & \langle c, f, t\rangle \in c \Leftrightarrow \texttt{class } c\texttt{\{}\ldots t\, f \ldots\texttt{\}} \text{ is in } P \end{array}$$

Method/advice $x$is declared in class c

$$\begin{array}{ll} & \langle x, (t_1, \ldots, t_n \to t), (var_1, \ldots, var_n), e\rangle \in c \\ \in & \Leftrightarrow \texttt{class } c\texttt{\{}\ldots t\, x\texttt{(}t_1\, var_1 \ldots t_n\, var_n\texttt{)\{}e\texttt{\}}\ldots\texttt{\}} \text{ is in } P \end{array}$$

Advice $c\texttt{.}a$ is listed in the precedence statement as $c'\texttt{.}a$

$$\begin{array}{ll} precLook(c,a) & \dfrac{\begin{array}{c}\texttt{declare precedence}\ldots c'\texttt{.}a\ldots \text{ is in } P \\ c \leq c' \text{ and } \langle a, \_, \_, \_\rangle \in c' \\ \nexists c\text{"} :\ (c \leq c\text{"} < c' \text{ and } \texttt{declare precedence}\ldots c''\texttt{.}a\ldots \text{ is in } P)\end{array}}{preclook(c,a) = c'} \end{array}$$

Advice precedence relation

$$\begin{array}{ll} & \langle c_a, a_a, \_, \_\rangle <^{\mathsf{prec}} \langle c_b, a_b, \_, \_\rangle \\ <^{\mathsf{prec}} & \Leftrightarrow \text{either } (precLook(c_a, a_a) = c'_a \text{ and } precLook(c_b, a_b) = c'_b \\ & \quad \textit{and } \texttt{declare precedence}\ldots c'_b\texttt{.}a_b \ldots c'_a\texttt{.}a_a \ldots \text{ is in } P) \\ & \quad \text{or } (\nexists c'_a :\ precLook(c_a, a_a) = c'_a \text{ and } precLook(c_b, a_b) = c'_b) \end{array}$$

Test advice kind

$$\begin{array}{ll} isBefore(c,a) & \texttt{class } c\ldots\texttt{\{}\ldots\texttt{before } a\texttt{:}\ldots\texttt{\}} \text{ is in } P \\ isAfter(c,a) & \texttt{class } c\ldots\texttt{\{}\ldots\texttt{after } a\texttt{:}\ldots\texttt{\}} \text{ is in } P \\ isAround(c,a) & \texttt{class } c\ldots\texttt{\{}\ldots\texttt{around } a\texttt{:}\ldots\texttt{\}} \text{ is in } P \\ isMethod(c,m) & !isBefore(c,m) \text{ and } !isAfter(c,m) \text{ and } !isAround(c,m) \end{array}$$

Retrieve name of `target` binding

$$\begin{array}{ll} target(c,a) & \dfrac{\texttt{class } c\ldots\texttt{\{}\ldots\ a\texttt{:call(}\ldots\texttt{) \&\& target(}var\texttt{)\}} \text{ is in } P}{target(c,a) = var} \end{array}$$

Retrieve the condition of an `if` pointcut construct

$$\begin{array}{ll} ifPcut(c,a) & \dfrac{\texttt{class } c\ldots\texttt{\{}\ldots\ a\texttt{:}\ldots\ \texttt{\&\& if(}e\texttt{)}\ldots\texttt{\}} \text{ is in } P}{ifPcut(c,a) = e} \\ ifPcut(c,a) & \dfrac{\texttt{class } c\ldots\texttt{\{}\ldots\ a\texttt{:}\ldots\ \texttt{\&\& if(}e\texttt{)}\ldots\texttt{\}} \text{ is not in } P}{ifPcut(c,a) = \texttt{true}} \end{array}$$

Method lookup of m in the dynamic type c

$$\begin{array}{ll} mlook(c,m) & \dfrac{\begin{array}{c} c \leq c' \text{ and } \langle m, \_, \_, \_\rangle \in c' \\ \nexists c'' : \langle m, \_, \_, \_\rangle \in c'' \text{ and } c \leq c'' < c' \end{array}}{mlook(c,m) = c'} \end{array}$$

Body $c\texttt{.}x$ matches with `call` pointcut of advice $c'\texttt{.}a$

$$\begin{array}{ll} & \texttt{class } c'\ldots\texttt{\{}\ldots a\texttt{:call(}t\ c_{pcut}\texttt{.}x\texttt{(}\ldots\texttt{))}\ldots\texttt{\}}\text{is in } P \\ call(c, x, c', a) & \text{where } c \leq c_{pcut} \text{ and } \exists c'_{pcut} : c'_{pcut} = mlook(c_{pcut}, x) \end{array}$$

Find the sequence of advice matching on $c\texttt{.}x$

$$\begin{array}{ll} alook(c, x, \mathcal{S}) & \dfrac{\begin{array}{c} \mathcal{A} \text{ is a sequence of distinct } \mathcal{E}\text{-tuples such that:} \\ \forall \mathcal{E}_i \in \mathcal{A} :\ (\mathcal{E}_i = \langle c_i, a_i, o_i, \texttt{false}\rangle \text{ and } call(c, x, c_i, a_i)) \\ \text{and } \mathcal{S}(o_i) = \langle c'_i, \_\rangle \text{ and } c_i = mlook(c'_i, a) \\ \forall \mathcal{E}_i, \mathcal{E}_{i+1} \in \mathcal{A} :\ \mathcal{E}_i <^{\mathsf{prec}} \mathcal{E}_{i+1} \end{array}}{alook(c, x, \mathcal{S}) = \mathcal{A}} \end{array}$$

Figure 8. Helper predicates and functions

or as a subtype. (The well-formedness rules in Sec. 3.4 ensures this element can always be uniquely determined.) Note that $<^{\mathsf{prec}}$ is undefined when neither of the two advice being compared are mentioned in the `declare precedence` statement. In case only one advice is mentioned, it gets the higher precedence.

The core of ContractAJ's lookup mechanism is defined by *mlook*, *call* and *alook*. The $mlook(c, m)$ function performs regular method lookup. Predicate $call(c, x, c, a')$ tests whether a particular method/advice $c.x$ matches with the `call` pointcut construct in advice $c'.a$, either directly or as a subtype. Finally, the *alook* function determines the sequence of advice whose `call` pointcut construct matched on $c.x$. More specifically, the *alook* function produces a list of $\mathcal{E}$ tuples, ordered by the precedence mechanism, where each tuple describes which advice body matched, the corresponding aspect instance o_i and a `false` value to indicate that we have not checked for higher-order advice yet. Regarding the aspect instance o_i, we also specify that the matching advice $c_i.a_i$ does not always have to be part of the type of o_i directly, but may also be inherited.

After discussing the helper predicates and functions, we can now present the rules that compose ContractAJ's operational semantics, shown in Fig. 9. First, note that the syntax of expressions (e) is extended with two new statements: `return` and `jpop`. These two statements are not available to the developer writing ContractAJ programs, but are only used internally by the semantics. The `jpop` statement is used whenever an $\mathcal{A}$ record needs to be popped from the join point stack. The only purpose of the `return` statement is to serve as a marker in the theorems of Sec. 6, such that we can easily refer to any configuration where the execution of a method/advice body is about to finish. Next, the definition of evaluation context E specifies the order in which subexpressions should be evaluated for each type of compound expression, which ensures there can only be one possible sequence of rule applications to evaluate a ContractAJ program. Finally, we can discuss the evaluation rules themselves in Fig. 9. The more interesting rules are those that specify the behaviour of method and proceed calls: [**call**], [**before**], [**after**], [**around**], [$\mathbf{call_{around}}$], [**exec**] and [**jpop**]:

[**call**] - This rule matches on method calls obj:$c.x$. This method call is replaced with a proceed call (wrapped in a `jpop` expression). While it may seem peculiar to replace every method call with a proceed call, they both share the same intuition: Try to execute the next matching advice; otherwise do regular method lookup. The only difference is that a method call has the additional task of looking for the matching advice, which is exactly what this [**call**] rule does with the help of the *alook* function. This function produces a list of $\mathcal{E}$ tuples that each represent an advice whose `call` pointcut construct matched on $c.x$. The $\langle c, x, obj, \mathtt{true}\rangle$ tuple, representing the method call, is appended to the result of *alook* to complete the $\mathcal{A}$ record that is then pushed onto the join point stack. After an application of the [**call**] rule, the proceed call it produced must be evaluated next. The semantics of proceed calls is defined by the [**before**], [**after**], [**around**] and [**exec**] rules.

[**before**] - This rule matches if there is a before advice $c.a$ in the tuple at the top of the join point stack. This tuple is popped and the `proceed` expression is

$$\begin{array}{rl} & \ldots \mid obj \\ e = & \mid \texttt{jpop\{}e\texttt{\}} \\ & \mid \texttt{return:}c\texttt{\{}e\texttt{\}} \\ \\ v = & obj \mid \texttt{null} \\ & \mid \texttt{true} \mid \texttt{false} \end{array} \qquad \begin{array}{rl} \mathsf{E} = & [\,] \mid \mathsf{E} : c.f \mid \mathsf{E} : c.f\texttt{=}e \mid v\texttt{:}c.f = \mathsf{E} \\ \mid & \mathsf{E}\texttt{.}m\texttt{(}e\ldots\texttt{)} \mid v\texttt{.}m\texttt{(}v\ldots\mathsf{E}\,e\ldots\texttt{)} \\ \mid & \texttt{super} \equiv v\texttt{:}c\texttt{.}m\texttt{(}v\ldots\mathsf{E}\,e\ldots\texttt{)} \\ \mid & \texttt{(}t\texttt{)}\ \mathsf{E} \mid \mathsf{E}\ \texttt{instanceof}\ t \mid \texttt{if(}\mathsf{E}\texttt{)\{}e\texttt{\}else\{}e\texttt{\}} \\ \mid & \texttt{\{}\mathsf{E}\texttt{;}e\texttt{\}} \mid \texttt{let\{}var\texttt{=}v\ldots var\texttt{=}\mathsf{E}\ldots var\texttt{=}e\ldots\texttt{\}in\{}e\texttt{\}} \\ \mid & \texttt{return:}c\texttt{\{}\mathsf{E}\texttt{\}} \mid \texttt{jpop\{}\mathsf{E}\texttt{\}} \end{array}$$

$$\begin{array}{ll}
[\mathbf{call}] & P \vdash \langle \mathsf{E}[obj\texttt{:}c\texttt{.}x\texttt{(}v_1\ldots v_n\texttt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{jpop\{proceed(}obj\, v_1\ \ldots\ v_n\texttt{)\}}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \text{where } \mathcal{A} = alook(c, x, \mathcal{S}) + \langle c, x, obj, \texttt{true}\rangle \text{ and } !isAround(c, x) \\
[\mathbf{before}] & P \vdash \langle \mathsf{E}[\texttt{proceed(}obj\, v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{if(}e'\texttt{)\{}obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}obj\, v_1\ldots v_n\texttt{)\};proceed(}obj\, v_1\ldots v_n\texttt{)}], \mathcal{S}, \mathcal{A}'; \mathcal{J}\rangle \\
& \text{where } \mathcal{A} = \langle c, a, obj_{asp}, \texttt{false}\rangle + \mathcal{A}' \text{ and } \langle a, (t_1, \ldots, t_n \to t), (var_1, \ldots, var_n, e)\rangle \in c \\
& \text{and } isBefore(c, a) \text{ and } var_{tgt} = target(c, a) \text{ and } e = ifPcut(c, a) \\
& \text{and } e' = e[obj_{asp}/\texttt{this}, obj/var_{tgt}, v_1/var_1, \ldots, v_n/var_n] \\
[\mathbf{after}] & P \vdash \langle \mathsf{E}[\texttt{proceed(}obj\, v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{proceed(}obj\, v_1\ldots v_n\texttt{);if(}e'\texttt{)\{}obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}obj\, v_1\ldots v_n\texttt{)\}}], \mathcal{S}, \mathcal{A}'; \mathcal{J}\rangle \\
& \text{(same constraints as [}\mathbf{before}\text{], except that } isBefore(c, a) \text{ becomes } isAfter(c, a)) \\
[\mathbf{around}] & P \vdash \langle \mathsf{E}[\texttt{proceed(}obj\, v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{if(}e'\texttt{)\{}obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}obj\, v_1\ldots v_n\texttt{)\}else\{proceed(}obj\ \ v_1\ldots v_n\texttt{)\}}], \mathcal{S}, \mathcal{A}'; \mathcal{J}\rangle \\
& \text{(same constraints as [}\mathbf{before}\text{], except that } isBefore(c, a) \text{ becomes } isAround(c, a)) \\
[\mathbf{call_{around}}] & P \vdash \langle \mathsf{E}[obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{proceed(}obj_{asp}\, v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}' + \mathcal{A}; \mathcal{J}\rangle \\
& \text{where } \mathcal{A}' = alook(c, a, \mathcal{S}) + \langle c, a, obj_{asp}, \texttt{true}\rangle \text{ and } isAround(c, a) \\
[\mathbf{exec}] & P \vdash \langle \mathsf{E}[\texttt{proceed(}obj\, v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\texttt{return:}c\texttt{\{}e[obj_{this}/\texttt{this}, v_1/var_1, \ldots, v_n/var_n]\texttt{\}}, \mathcal{S}, \mathcal{A}'; \mathcal{J}\rangle \\
& \text{where } \mathcal{A} = \langle c, x, obj', \texttt{true}\rangle + \mathcal{A}' \\
& \text{and if } isMethod(c, x) \text{ then } (obj_{this} = obj) \text{ else } (obj_{this} = obj') \\
& \text{and } \mathcal{S}(obj_{this}) = \langle c', \ldots\rangle \text{ and } c'' = mlook(c', x) \\
& \text{and } \langle x, (t_1, \ldots, t_n \to t), (var_1, \ldots, var_n, e)\rangle \in c'' \\
[\mathbf{jpop}] & P \vdash \langle \mathsf{E}[\texttt{jpop\{}v\texttt{\}}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}, \mathcal{J}\rangle \\
[\mathbf{super}] & P \vdash \langle \mathsf{E}[\texttt{super} \equiv obj\texttt{:}c\texttt{.}m(v_1\ldots v_n)], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[e[obj/\texttt{this}, v_1/var_1, \ldots, v_n/var_n], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \langle m, (t_1, \ldots, t_n \to t), (var_1, \ldots, var_n, e)\rangle \in c \\
[\mathbf{return}] & P \vdash \langle \mathsf{E}[\texttt{return:}c\texttt{\{}e\texttt{\}}],\ \mathcal{S},\ \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}, \mathcal{J}\rangle
\end{array}$$

$$\begin{array}{ll}
[\mathbf{new}] & P \vdash \langle \mathsf{E}[\texttt{new}\ c], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[obj], \mathcal{S}[obj \mapsto \langle c, \mathcal{F}\rangle], \mathcal{J}\rangle \\
& \text{where } obj \notin dom(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.f \mapsto \texttt{null} \mid c \le c' \text{ and } \exists t : \langle c', f, t\rangle \in c'\} \\
[\mathbf{get}] & P \vdash \langle \mathsf{E}[obj\texttt{:}c'\texttt{.}f], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{S}(obj) = \left\langle c, \mathcal{F}\right\rangle \text{ and } \mathcal{F}(c'.f) = v \\
[\mathbf{set}] & P \vdash \langle \mathsf{E}[obj\texttt{:}c'\texttt{.}f\texttt{=}v], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[obj \mapsto \langle c, \mathcal{F}[c'.f \mapsto v]\rangle], \mathcal{J}\rangle \\
& \text{where } \mathcal{S}(obj) = \left\langle c, \mathcal{F}\right\rangle \\
[\mathbf{cast}] & P \vdash \langle \mathsf{E}[\texttt{(}t\texttt{)}\ \ obj], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[obj], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{S}(obj) = \left\langle c, \mathcal{F}\right\rangle \text{ and } c \le t \\
[\mathbf{inst_{true}}] & P \vdash \langle \mathsf{E}[obj\ \texttt{instanceof}\ \ t], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[\texttt{true}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{S}(obj) = \left\langle c, \mathcal{F}\right\rangle \text{ and } c \le t \\
[\mathbf{inst_{false}}] & P \vdash \langle \mathsf{E}[obj\ \texttt{instanceof}\ \ t], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[\texttt{false}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{S}(obj) = \left\langle c, \mathcal{F}\right\rangle \text{ and } c \not\le t \\
[\mathbf{let}] & P \vdash \langle \mathsf{E}[\texttt{let \{}var_1\texttt{=}v_1\ldots var_n\texttt{=}v_n\texttt{\} in \{}e\texttt{\}}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[e[v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}, \mathcal{J}\rangle \\
[\mathbf{if_{true}}] & P \vdash \langle \mathsf{E}[\texttt{if(true)\{}e_1\texttt{\}else\{}e_2\texttt{\}}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[e_1], \mathcal{S}, \mathcal{J}\rangle \\
[\mathbf{if_{false}}] & P \vdash \langle \mathsf{E}[\texttt{if(false)\{}e_1\texttt{\}else\{}e_2\texttt{\}}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[e_2], \mathcal{S}, \mathcal{J}\rangle \\
[\mathbf{seq}] & P \vdash \langle \mathsf{E}[\texttt{\{}v\texttt{;}e\texttt{\}}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}, \mathcal{J}\rangle \\
[\mathbf{error}] & P \vdash \langle \mathsf{E}[\texttt{error(}msg\texttt{)}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \texttt{error:}\ \ msg, \mathcal{S}, \mathcal{J}\rangle
\end{array}$$

Figure 9. Operational semantics of ContractAJ

replaced with an explicit method call to the advice body, followed by the implicit `proceed` call. Note that this explicit call to the advice body will only be executed if the advice's `if` pointcut construct succeeds. Because we are using an explicit method call, this will cause the [**call**] rule to match, which will then look for any higher-order advice that match on $c.a$. An infinite regression cannot occur when this before advice reappears at a later point in the evaluation; this is due to the boolean value in each $\mathcal{E}$ tuple. It indicates whether we have already processed this advice or not. More specifically, the [**before**] rule will only match if the boolean in the join point stack's top entry is `false`. Once the [**call**] rule has processed the explicit call to the before advice, that boolean will be set to `true`.

[**after**] - This rule is analogous to [**before**], except that the implicit proceed call comes before explicitly calling the advice.

[**around**] - This rule is analogous to [**before**] as well. If the `if` pointcut construct evaluates to true, an explicit call is made to the around advice. Otherwise, the advice is skipped by only making a `proceed` call to the next advice.

[**call$_{\textbf{around}}$**] - This rule is a variant of the [**call**] rule; it only handles explicit calls to around advice. Whereas [**call**] will push a new record onto the join point stack, [**call$_{\textbf{around}}$**] will extend the existing record that is currently at the top of the stack. The reason for this difference is to support higher-order around advice that do *not* make a proceed call. If an around advice $c.a$ does not proceed, this means that any remaining advice in the current advice composition, and the method/advice body being advised will no longer be executed. Additionally, it is possible that the body being advised is another around advice: in this case all remaining advice in the composition of that around advice and the body it advises will not be executed either, and so on. To achieve this behaviour, [**call$_{\textbf{around}}$**] extends the record at the top of the stack: all of the bodies that should no longer be executed are now grouped into one record, such that they will be removed from the stack once the execution of $c.a$ is finished.

[**exec**] - This rule matches once we are ready to execute the $\mathcal{E}$ tuple at the top of the join point stack, as indicated by the `true` value in this tuple. The *mlook* function is first used to perform regular method lookup using dynamic type c and method/advice x, resulting in lookup result c'. The receiver object obj_{this} is determined in one of two ways: If x represents an advice, it is retrieved from the top tuple in the stack. Otherwise, we use the first argument of the proceed call (obj), which represents the binding of the `target` pointcut construct. This adds support for receiver substitution, as the value of obj can be chosen by the developer (if the proceed call is part of an around advice).

We can then replace the `proceed` expression with e, the body of $c'.x$. Formal parameters and the *this* object are also bound to their values. A `return` wrapper is also added to the e expression, which ensures the [**return**] rule will match once the evaluation of e is finished. Finally, the $\mathcal{E}$ tuple at the top of the stack is removed, as it is no longer needed[5].

[5] The tuple might still be needed in case an around advice makes multiple proceed calls. This is however not supported, as it is an uncommon scenario and would unnecessarily complicate the semantics.

[**jpop**] - This rule pops the top $\mathcal{A}$ record from the join point stack. This top record typically already is empty at this point, unless there were around advice that did not make a proceed call. Note that `jpop` expressions are only created by the [**call**] rule, not by [$\mathbf{call_{around}}$]. This is because there should only be one `jpop` expression per record, and [$\mathbf{call_{around}}$] only extends an existing record instead of adding a new one.

[**super**] - This rule handles super calls. In order to keep the semantics simple, we chose not to support advice on super calls. As such, the rule can immediately replace the call with the corresponding method body.

[**return**] - As mentioned earlier, return expressions only serve as markers that indicate the end of a body's execution. These expressions are created by the [**exec**] rule at the start of a body's execution.

3.3 Lookup sequences

To provide a more high-level view on ContractAJ's lookup mechanism, Fig. 10 represents all possible sequences of rule applications that can be taken starting from a configuration with a method/proceed call, and ending with the configuration where we have determined which body will be executed. A few example sequences are also given in Fig. 11.

$$\begin{aligned}
\textit{mcall} &::= [\mathbf{call}]\ \textit{lookup}{*}|\textit{skip}{*}\ [\mathbf{exec}] \\
\textit{pcall} &::= \textit{lookup}{*} \mid \textit{skip}{*}\ [\mathbf{exec}] \\
\textit{lookup} &::= \textit{skip}{*}\ \textit{match} \\
\textit{match} &::= ([\mathbf{before}]\ \ldots\ [\mathbf{if_{true}}]\ [\mathbf{call}]) \mid ([\mathbf{around}]\ \ldots\ [\mathbf{if_{true}}]\ [\mathbf{call_{around}}]) \\
\textit{skip} &::= ([\mathbf{after}]\ \ldots\ [\mathbf{if_{false}}] \mid [\mathbf{if_{true}}]) \mid ([\mathbf{before}] \mid [\mathbf{around}]\ \ldots\ [\mathbf{if_{false}}])
\end{aligned}$$

Figure 10. All possible lookup sequences

mcall/*pcall* - *mcall* represents all possible sequences of rule applications for method calls, whereas *pcall* represents proceed calls. Note that the only difference between the two is that *mcall* initially applies the [**call**] rule. A method call starts in a configuration $\langle e, \mathcal{S}, \mathcal{J} \rangle$, where e decomposes into the method call to be executed. The [**call**] rule then replaces this method call in e with a proceed call, such that the lookup mechanism for proceed calls can be reused.

lookup - The *lookup* sequence may be applied multiple times in *mcall* and *pcall*. It searches for the first before/around advice that must be executed. (After advice will be discussed later as a separate case, due to the preceding implicit proceed call.) If *lookup* does not match in *mcall*/*pcall*, the subsequent application of [**exec**] must initiate the execution of a method body, as there is a method body at the top of $\mathcal{J}$. If *lookup* matches exactly once, [**exec**] will initiate a non-higher-order before/around advice. If *lookup* matches more than once, a higher-order before/around advice will be initiated.

skip - The *skip* sequence represents an advice that initially matches, but will not be executed. This can happen for one of two reasons: The `call` pointcut

Method call leading to the execution of a method body (no matching advice):
[**call**] [**exec**]

Method call where the first advice to be executed is an around advice:
(The `call` pointcut of an after advice did match first, but its `if` construct failed.)
[**call**] [**after**] ... [$\mathbf{if}_{\mathbf{false}}$] [**around**] ... [$\mathbf{if}_{\mathbf{true}}$] [$\mathbf{call}_{\mathbf{around}}$] [**exec**]

Method call advised by a before advice, which is advised by an around advice:
[**call**] [**before**] ... [$\mathbf{if}_{\mathbf{true}}$] [**call**] [**around**] ... [$\mathbf{if}_{\mathbf{true}}$] [$\mathbf{call}_{\mathbf{around}}$] [**exec**]

Figure 11. A few example lookup sequences

construct of an advice matches, but its `if` construct does not. In this case, either [**before**],[**after**] or [**around**] is applied first, which will insert a runtime test for the `if` pointcut construct. The subsequent rule applications (indicated with an ellipsis) represent the evaluation of this `if` condition. If it fails, [$\mathbf{if}_{\mathbf{false}}$] matches and this advice will not be executed. The second reason for not (immediately) executing an advice is because it is an after advice. Because there is an preceding implicit proceed call which must be evaluated first, a method/proceed call cannot directly result in the execution of an after advice body.

match - The *match* sequence represents a matching before/around advice. In this case, the advice's `if` pointcut construct does succeed, as indicated by the application of [$\mathbf{if}_{\mathbf{true}}$]. We will now explicitly call the matching advice (to look for any higher-order advice), as indicated by the application of [**call**]/[$\mathbf{call}_{\mathbf{around}}$].

Finally, we should still discuss the execution of after advice bodies: Due to the presence of the implicit proceed call, an after advice body can only be initiated once this implicit proceed call finishes, resulting in the "[**return**] *mcall*" rule sequence. The application of [**return**] represents the end of the after advice's implicit proceed call, which is then followed by an explicit call to the after advice body.

3.4 Well-formedness rules

To wrap up the definition of ContractAJ's semantics, Fig. 12 presents the constraints that must be satisfied by every ContractAJ program in order to be well-formed. Most constraints were carried over from the object-oriented ClassicJava [19] language. The constraints specific to ContractAJ are mostly self-explanatory. Only the $AdvByOK$ constraint should be discussed in some more detail: This constraint is defined in terms of the $advBy(c, m)$ helper function, which retrieves the complete `@advisedBy` clause of $c.m$, including the part inherited from its super class. An `@advisedBy` clause specifies a list of advice that a method is expecting to be advised by, in the given order. At runtime, this expectation may also be fulfilled by an overriding advice. To prevent ambiguities when determining which element in an `@advisedBy` clause corresponds to a particular advice, the $AdvByOK$ constraint requires that these elements may not override each other. Additionally, the constraint requires that the elements of the `@advisedBy` clause are ordered such that they respect the precedence declaration.

$UniqClasses$	Each class is defined only once. $\forall\, c, c'$ $\texttt{class}\, c \ldots \texttt{class}\, c' \ldots$ is in $P \implies c \neq c'$
$UniqFields$	Each member is defined only once per class. $\forall\, f, f'$ $\texttt{class} \ldots \texttt{\{} \ldots f \ldots f' \ldots \texttt{\}}$ is in $P \implies f \neq f'$
$UniqMethods$	$\forall\, m, m'$ $\texttt{class} \ldots \texttt{\{} \ldots m(\ldots)\texttt{\{} \ldots \texttt{\}} \ldots m'(\ldots)\texttt{\{} \ldots \texttt{\}} \ldots \texttt{\}}$ is in $P \implies m \neq m'$
$UniqAdvice$	$\forall\, a, a'$ $\texttt{class} \ldots \texttt{\{} \ldots a \texttt{ : } \ldots \texttt{\{} \ldots \texttt{\}} \ldots a' \texttt{ : } \ldots \texttt{\{} \ldots \texttt{\}} \ldots \texttt{\}}$ is in $P \implies a \neq a'$
$CompleteClasses$	The superclass of each class is defined. $rng(\prec) \subseteq dom(\prec) \cup \{\texttt{Object}\}$
$WellFoundedClasses$	Class hierarchy is an order. $\leq$ is antisymmetric
$ClassMethodsOK$	Method overriding preserves the type. $\forall c, c', e, e', m, T, T', V, V'$ $(\langle m, T, V, e\rangle \in c$ and $\langle m, T', V', e'\rangle \in c')$ $\implies (T = T'$ or $c \leq c')$
$PrecedenceOK$	No duplicate entries in the precedence declaration. $\forall\, c, a, c', a'$ $\texttt{declare precedence}$ $\ldots c\texttt{.}a \ldots c'\texttt{.}a' \ldots$ is in $P \implies \langle c, a\rangle \neq \langle c', a'\rangle$
$ProceedInAdvice$	Proceed calls may not be used in methods. $\nexists m :$ $\texttt{class} \ldots \texttt{\{} \ldots m(\ldots)\texttt{\{} \ldots \texttt{proceed} \ldots \texttt{\}} \ldots \texttt{\}}$ is in P
$ProcInSpecs$	The `proc` keyword may only be used in specifications. $\nexists x$: $\texttt{class} \ldots \texttt{\{} \ldots x(\ldots)\texttt{\{} \ldots \texttt{proc} \ldots \texttt{\}} \ldots \texttt{\}}$ is in P
$advBy(c, m)$	Retrieve the complete `@advisedBy` clause of a method $\dfrac{\begin{array}{c}\texttt{class}\ c \ldots \texttt{\{} \ldots \texttt{@advisedBy}\ c_1\texttt{.}a_1\texttt{,} \ldots \texttt{,}c_n\texttt{.}a_n\texttt{;} \ldots m \ldots \texttt{\}} \text{ is in } P \\ A = advBy(c', m) \text{ if } (\exists c' : c' = mlook(c, m) \text{ and } c \neq c') \\ A = \emptyset \text{ otherwise}\end{array}}{advBy(c, m) = (c_1, a_1, \ldots, c_n, a_n) \circ A}$
$AdvByOK$	Advice in an `@advisedBy` clause may not override each other and should respect the precedence order. $\forall c, m, A\ \langle m, \_, \_, \_\rangle \in c$ and $A = advBy(c, m) = (c_1, a_1, \ldots, c_n, a_n)$ $\implies ((c_i \leq c_j \implies a_i \neq a_j)$ where $i \neq j)$ and $(c_i <^{\mathsf{prec}} c_j$ where $i < j)$

Figure 12. Static constraints on ContractAJ programs

4 The advice substitution principle

After defining the ContractAJ language, we can use it to present our approach to modular reasoning in AOP languages. This approach can be divided into two parts: the advice substitution principle (ASP) and the `@advisedBy` clause. This section will focus on the first part, the ASP. If an advice complies with this principle, modular reasoning is possible even while remaining oblivious of this advice. That is, the advice will not cause any surprising behaviour whenever a method call (or proceed call) is made. The purpose of the ASP is similar to the notion of observers, spectators, spectative and harmless advice [11,15,21,37]. However, the key difference between these notions and the ASP is that the ASP is a property of an advice's specification rather than its implementation. This allows for two advantages: First, our approach to modular reasoning should be familiar to OOP developers, as it is a natural extension of modular reasoning in OOP, which is typically also defined in terms of a program's specifications. Second, because a program's specifications describe the expected behaviour of each module, it also is clear what constitutes unexpected/surprising behaviour. This is what allows the ASP to be weaker/less conservative than observers, spectators, spectative and harmless advice. These notions only rely on the program implementation, where it is far from trivial to deduce what constitutes unexpected behaviour.

The ASP is based on the notion of behavioural subtyping in object-oriented languages [3,16,24,27]. The ASP presented in this paper is however slightly different than the ASP first introduced by Wampler [41], which is an adaptation of Liskov and Wing's constraint-based behavioural subtyping [27, Fig. 4]. Our version of the ASP is based on Dhara and Leavens' strong behavioural subtyping (SBS) [16, Def. 4.1 and 4.2], as it has a postcondition rule that is more flexible than Liskov and Wing's. We paraphrase the rules on preconditions, postconditions and invariants of the SBS definition as follows:

Strong behavioural subtyping (SBS). *Type t is a strong behavioural subtype of type u, if and only if $t < u$ and:*

- *For all objects of type t, and for all common methods m in t and u:*
  - *The precondition of $t.m$ must be equal to or weaker than the precondition of $u.m$.*
  - *The postcondition of $t.m$ must be equal to or stronger than the postcondition of $u.m$, if the precondition of $u.m$ held in the pre-state.*
- *For all objects of type t:*
  - *The invariant of u should be preserved in t.*

To adapt the SBS rules to an aspect-oriented setting, the basic idea is to view the execution of an advice as a form of substitution. This is quite easy to understand when all advice are viewed as around advice. This is not a simplification, as a before advice can be seen as an around advice where the proceed call at the end is implicit. Likewise, an after advice corresponds to an around advice where the implicit proceed call is at the beginning. If the pointcut associated with an around advice matches on a certain join point, then that join point essentially is replaced with the execution of that advice. In other words, the join point representing a method call is *substituted* with the execution of an advice. In order to perform a method call, while remaining unaware of the advice that substitutes for it, an advice's contracts should comply with the contracts of those join points it advises.

4.1 Around advice

From the point of view that executing advice can be seen as a form of substitution, the SBS rules can be adapted as follows to an advice substitution principle for around advice:

ASP for around advice. *Consider an around advice a in type t that is applied to join point $u.x$, representing a method call or an advice execution. If x is a method, u is the static type of the receiver. If x is an advice, u is the class containing x. The around advice satisfies the ASP if and only if, for all objects of type t:*

- *The precondition of* $t.a$ *must be equal to or weaker than the precondition of* $u.x$.
- *The postcondition of* $t.a$ *must be equal to or stronger than the postcondition of* $u.x$*, if the precondition of* $u.x$ *held in the pre-state.*
- *The invariant of* u *should be preserved in* t.

What is important to note is that we defined the ASP in terms of a single advice applying at a particular join point. Stating that "aspect t complies with the ASP" means that each advice in t must take into account the contracts of *all* the join points it advises. Each of these join points can have its own contracts, which means that an advice may need to take into account several different contracts, depending on the advice's pointcut. While the exact set of join points in a pointcut can only be determined at runtime, the developer only needs to take into account all join point shadows, i.e. the mapping of each join point to its location in the source code. These join point shadows can be determined statically by examining the advice's pointcut.

4.2 Before and after advice

As mentioned earlier, before/after advice can be interpreted as special cases of around advice. It is important to note however that it would be unintuitive to include the effects of the implicit proceed call in the contracts of a before/after advice, such that these contracts would effectively be the same as an equivalent around advice. It is unintuitive for the simple reason that the developer does not need to be aware of any implicit proceed calls. Moreover, even if the developer knows there is an implicit proceed call, he/she may not consider it to be part of the advice body. Consequently, the ASP needs to be adjusted for before/advice to take this into account.

In a before advice, its postcondition refers to the moment *before* executing the implicit proceed call at the end of the advice body. In order for the composition of the before advice body ($t.a$) and the implicit proceed call to be substitutable for the advised join point ($u.x$), the **ASP for before advice** becomes:

- *The precondition of* $t.a$ *must be equal to or weaker than the precondition of* $u.x$.
- *If the precondition of* $u.x$ *held before executing the advice, it should still hold after the advice (at the implicit proceed call). This implies the postcondition of* $t.a$ *may not invalidate* $u.x$*'s precondition.*
- *The invariant of* u *should be preserved in* t.

Similarly, an after advice's precondition refers to the moment *after* executing the implicit proceed call in the beginning of the advice body. The **ASP for after advice** is as follows, for an after advice body $t.a$ advising join point $u.x$:

- *The precondition of* $t.a$ *must be equal to or weaker than the* ***post****condition of* $u.x$.

– *If the postcondition of* $u.x$ *held before executing the advice, it should still hold after the advice. This implies the postcondition of* $t.a$ *may not invalidate* $u.x$*'s postcondition.*
– *The invariant of* u *should be preserved in* t*.*

4.3 Relating the principle to quantification

As pointcuts are a quantification mechanism, a pointcut may potentially describe a large set of join point shadows. For example, a call/execution pointcut not only matches with the given type, but also its subtypes. Likewise, a pointcut (in AspectJ) could make use of wildcards to match with a large amount of shadows. If an advice now wants to comply with the ASP, it is important to consider that the number of reasoning tasks grows with the number of join point shadows it advises. While this paper does not aim to tackle this scaling problem, as it is separate from modular reasoning about method calls, the problem can be mitigated in a number of ways. First, examining only the advice body itself can sometimes already reveal whether it is ASP-compliant or not. For example, an advice that only modifies its own state, often classified as observer, spectator or spectative advice [11,21,37], most likely is ASP-compliant. Second, the developer can rely on tool support like the runtime contract enforcement algorithm of Sec. 7 to test whether an advice complies with the ASP. Finally, this scaling problem is also closely related to the fragile pointcut problem [22], which is about pointcuts (typically in AspectJ) relying on the names of types and methods to determine the set of matching join points, which is quite sensitive to changes. The various methods to tackle this fragility problem may also mitigate the scaling problem, as pointcuts can only get more fragile when they intend to match with a larger set of join point shadows.

4.4 Call and execution pointcuts

The ASP essentially states that advice should take into account the contracts of the join points they advise. There are however two subtleties to `call` and `execution` pointcuts when trying to determine this set of join points.

First, there is the fact that the specified type in `execution` pointcuts refers to the *dynamic* type of method calls, whereas the ASP is defined in terms of the contracts in the *static* type. It would be much easier if developers who write advice with an `execution` pointcut could ensure the ASP by only looking at the type specified directly in the pointcut. Fortunately, this is possible, as long as those specified types satisfy the SBS rules. For example, consider a method `User.toString()`. If `User` is a strong behavioural subtype, it may substitute for any of its ancestor classes. By extension, if an advice with pointcut `execution(* User.toString())` only takes into account the contracts of `User.toString`, the advice may substitute for any call to `toString` where the static type is `User`, *or an ancestor*. In other words, the ASP also is satisfied if an advice takes into account the dynamic type of its join points, assuming those types are behavioural subtypes.

The second subtlety involving `call` and `execution` pointcuts is that these pointcuts not only match if the static/dynamic type equals the pointcut's specified type, but they also match on subtypes. Complying with the ASP then means that the developer should be aware of *all subtypes* of the pointcut's specified type, which goes against the grain of modular reasoning. Unfortunately, in this case the ASP is not automatically guaranteed if we only take into account the specified types, even if all of their subtypes comply with the SBS rules. This can be demonstrated with the counterexample shown in Fig. 13.

```
class A {
    @requires x > 0
    void foo(int x) {...}}

class B extends A {
    @requires x > -10 // Weaker than A's precondition
    void foo(int x) {...}}

class C {
    @requires x > -5  // Weaker than A's precondition
    around anAdvice: execution(void A.foo(int x)) {...}}

main {
    A inst = new B;
    inst.foo(5);      // No contract violations
    B inst2 = new B;
    inst2.foo(-8);}   // Contract violation in C
```

Figure 13. Contract violation caused by `C` only taking into account `A`

The advice in `C` is written such that it takes the contracts of `A.foo` into account. However, the developer of `C` may not take into account subclass `B`, which overrides `A.foo`. Note that `B.foo` complies with the SBS rules, but its preconditions happen to be stronger than the advice in `C`. The advice could now inadvertently cause a contract violation whenever `B` is the static type in a method call. It is possible that the problem illustrated in Fig. 13 hardly ever occurs in practice, as it seems quite unlikely to accidentally create a situation where an advice does not comply with a subtype, but does comply with the specified type. Nonetheless, a practical approach to solve the problem is that the developers of aspects initially only take into account the types specified directly in a pointcut, but then also use tool support (like the contract enforcement algorithm of Sec. 7) to monitor whether any subtypes are causing ASP violations.

4.5 The `proc` keyword

Because an advice may need to comply with the contracts of several different join points, a mechanism is needed to keep advice contracts reasonably compact, and to prevent any unnecessary coupling with the contracts of each join point. After all, aspects are meant to implement *crosscutting* concerns, which indicates that they are typically loosely coupled to the functionality implemented by the advised join point.

ContractAJ provides a “specification inheritance” mechanism in the form of the `proc` keyword. When `proc` is used in the pre/postcondition of an advice that complies with the ASP, it refers to the pre/postcondition of the join point being advised. For example, consider the caching aspect in Fig. 14. If `Cache.store` is advising a call to `List.set`, its precondition is `i>=0 && i<this1.getLength()`. The postcondition is `this1.get(i)==val && this2.isCached(i,val)`. Note that we numbered each `this` keyword to avoid naming conflicts, as one refers to the instance of `List` and the other to the `Cache` instance.

```
class List{
    @requires i>=0 && i<this.getLength()
    @ensures this.get(i)==val
    void set(int i, Object val) {...} ...}

class Cache {
    @requires proc
    @ensures proc && this.isCached(i,val)
    around store: call(void List.set(int i, Object val)) {...} ...}
```

Figure 14. Example of using the `proc` keyword

The `proc` keyword is somewhat similar to e.g. the `also` keyword used in JML [23, Sec. 2.3] to inherit the specifications of an overridden method. The main difference is that the `proc` keyword can be used anywhere in the pre- or postcondition, whereas the use of `also` is constrained such that behavioural subtyping is always enforced by construction. We do not impose such constraints on `proc`, and hence make it possible for preconditions to be too strong, and postconditions too weak. This is needed to allow for aspects that cannot comply with the ASP, which is explained in Sec. 5.

4.6 Effective specifications

To make our notion of modular reasoning precise, we need to define which pre- and postconditions need to be ensured whenever a method or proceed call is made. We refer to these specifications as the “effective pre/postcondition” of a particular method call or proceed call. These effective pre/postconditions should allow for modular reasoning, in the sense that a developer who wants to make a call within a certain class, only needs to consider the specifications of that class itself, or anything explicitly referenced by that class.

The definition[6] of effective preconditions can be found in Fig. 15. The $pre(c, x)$ function simply retrieves the precondition of a body $c.x$ from the program’s

[6] This definition only applies if all advice are ASP-compliant; it will be extended later in Sec. 5.5 to take into account non-ASP-compliant advice as well.

$$\mathit{pre}(c,x) \qquad \frac{\texttt{class}\ c\ldots\texttt{\{@requires}\ e\ldots\ t\ x\ldots\texttt{\}}\text{ is in }P}{\mathit{pre}(c,x)=e}$$

$$\mathit{eff}_{pre}(c,m) = \mathit{pre}(c,m)$$

$$\mathit{effProc}_{pre}(c,m) = \mathit{pre}(c,m)$$

$$\mathit{effProc}_{pre}(c,a) = \mathit{pre}(c,a)[\texttt{proc} \mapsto \mathit{effProc}_{pre}(c',x)]$$
$$\text{where } c.a \text{ advises } c'.x \text{ and } !\mathit{isMethod}(c,a)$$

Figure 15. Defining the effective precondition of method/proceed calls

code. The $\mathit{eff}_{pre}(c,m)$ function defines the effective precondition of a method call $\mathit{obj} : c.m$. Note that we use the method call notation (defined in Fig. 6 of Sec. 3.1) of ContractAJ's semantics to emphasize that c stands for the static type of the receiver (obj). Because the developer does not need to be aware of any ASP-compliant advice, the effective precondition simply is $\mathit{pre}(c,m)$, which is no different from modular reasoning in object-oriented languages.

The $\mathit{effProc}_{pre}(c,m)$ function defines the effective precondition of a proceed call, where the proceed call is located in an advice that advises a method call $\mathit{obj} : c.m$. Because an ASP-compliant advice does not have to be aware of any other advice, the effective precondition of a proceed call also is $\mathit{pre}(c,m)$. However, keep in mind that $\mathit{effProc}_{pre}$ is defined in terms of a single method body $c.m$, and that an advice can apply to multiple different method bodies. As the ASP requires that an advice takes into account all of its advised join points, the developer should also take into account the multiple applicable versions of $\mathit{effProc}_{pre}$ when making a proceed call.

Finally, $\mathit{effProc}_{pre}$ also is defined for proceed calls that occur in higher-order advice. That is, $\mathit{effProc}_{pre}(c,a)$ defines the effective precondition of a proceed call, if this proceed call occurs within an advice that advises $c.a$. In this case, the effective precondition is $\mathit{pre}(c,a)$, but because $c.a$ might make use of the `proc` keyword, we should also evaluate the keyword to its concrete value. The `proc` keyword is replaced with the effective precondition of any proceed calls inside $c.a$, which is $\mathit{effProc}_{pre}(c',x)$, considering that $c.a$ advises $c'.x$.

Next to the definitions of effective preconditions, one can also give definitions of effective postconditions in eff_{post} and $\mathit{effProc}_{post}$, which are identical to eff_{pre} and $\mathit{effProc}_{pre}$ apart from replacing every occurrence of “pre” with “post”.

4.7 Frame conditions

To allow for formal verification, methods and advice should also have a frame condition [8], which specifies what does *not* change after executing a method/advice. While the paper does not focus in particular on formal verification with program specifications, we should briefly discuss the relation between the ASP and frame conditions, as they also play a role in modular reasoning.

In a specification language such as JML, a frame condition is specified by an `@assignable` clause, which lists what *might* change after executing a method.

This implies that everything that is not listed will not change. Because frame conditions are considered part of postconditions, this means they may not be weakened according to the ASP. That is, the frame condition of an advice may not modify *more* variables/fields than the frame conditions of the advised join points. With a strict interpretation of frame conditions, this means an aspect is not allowed to modify its own fields. This would exclude several aspects that are otherwise ASP-compliant, such as aspects that implement logging, caching or contract enforcement. However, the base system can safely remain unaware of such aspects, despite the fact that they modify their own (private) fields. Given this observation, it seems reasonable that the frame properties of a method may ignore any modifications to private fields of aspects, as these modifications are irrelevant to the behaviour of the base system. This should cause no harm, as long as the values of these fields cannot be accessed outside the corresponding aspect's control flow. Perhaps a more precise idea can be formed of what can safely be considered irrelevant to the frame condition of a method, in order to give ASP-compliant advice more freedom to modify locations. However, we should then consider systems like data groups [25] or ownership types [9,14], which goes beyond the scope of this paper.

5 The `@advisedBy` clause

The ASP ensures that, if an advice complies with this principle, that advice can safely substitute for the join points it advises without causing any surprising behaviour. While several kinds of crosscutting concerns (e.g. logging, profiling, caching, monitoring, ...) can be implemented in an ASP-compliant manner, there also are several others that inherently cannot comply with the ASP. That is, they must alter the specifications of their advised join points in some way. For example, consider the `Security.authenticate` advice in Fig. 16. If the user is currently logged in, the advice will ensure the same postcondition as its advised join points, which is okay with the ASP in this instance. However, if the user is not logged in, the advice will block the execution of the advised method and ensures nothing at all (i.e. `true`). In this case, the postcondition clearly is weaker than the advised join point, which violates the ASP. It is also clear that this advice cannot be rewritten in an ASP-compliant way, as its very purpose is to ensure that the requested operation is blocked when the user is not authenticated.

Even if only a single advice in the system would violate the ASP, it seems that we should revert back to global reasoning, which would defeat the purpose of the ASP. To deal with such non-ASP-compliant advice (or "non-ASP advice" for short), one option is to simply avoid non-ASP advice altogether and implement their functionality using plain method calls instead. While this certainly is a valid option, it also sacrifices AOP's benefits. In particular, there no longer is a notion of quantification: Rather than using one pointcut to indicate all the join points where a block of code should be executed, several method calls have to be added instead, possibly including additional residual logic if these calls should only be executed on certain (run-time) conditions.

Rather than going back to square one, we propose a simple clause called "`@advisedBy`" that preserves modular reasoning, quantification, allows pointcuts that can only be determined at runtime, and allows ASP-advice to share join points with non-ASP advice. The starting observation is that, if an advice cannot comply with the ASP, it must be doing something surprising that was not expected by the caller of the advised method. To prevent such surprises, the caller should be made aware of any non-ASP advice that apply to this method, which is done by adding an `@advisedBy` clause to the method's specifications.

An example usage of the `@advisedBy` clause is shown in Fig. 16. The clause is used in `Account.withdraw`, which specifies that this method expects to be advised by `Security.authenticate` and `Security.authorize`, in that order. Note that the `deposit` method is also advised by `Logger.log`, which does not need to be mentioned in the `@advisedBy` clause as it is ASP-compliant.

```
class Account {
    @requires  this.getAmount() >= m && m>0
    @ensures   this.getAmount() == old(this.getAmount()) - m
    @advisedBy Security.authenticate, Security.authorize
    int withdraw(int m) {...} ...}

class Security {
    @requires proc
    @ensures if(isLoggedIn()){proc}else{true}
    around authenticate: call(void Account.withdraw(int m)) {...}

    @requires proc
    @ensures if(isAuthorised()){proc}else{true}
    around authorize: call(void Account.withdraw(int m))
        && if(isEnabled()) {...} ...}

class Logger {
    @requires true
    @ensures old(getLogEntries())+1==getLogEntries()
    before log: execution(void Account.withdraw(int m)) {...} ...}
```

Figure 16. Using the `@advisedBy` clause

In general, the `@advisedBy` clause indicates that a method is expecting to be advised by the listed advice. Consequently, any client that wants to call this method will notice its `@advisedBy` clause and should take into account the listed advice. From the perspective of an advice, if it is mentioned in an `@advisedBy` clause and it makes a proceed call, that advice should now be aware of the next element executed in the clause. In the example of Fig. 16, when `authenticate` makes a proceed call, it should be aware that this will execute the `authorize` advice. Likewise, the `proc` keyword mentioned in `authenticate`'s specifications will refer to the pre/postcondition of `authorize`. Because the `log` advice is not mentioned explicitly in the `@advisedBy` clause, the `proc` keyword in `authorize` refers directly to the specifications in `Account.deposit`, as discussed in Sec. 4.5.

Finally, note that the `@advisedBy` clause can only be added to methods, which implies that all higher-order advice should be ASP-compliant. Conceptually it is possible to add an `@advisedBy` clause to an advice, to indicate that it is aware of the listed higher-order advice. However, constructing effective pre/postconditions is complicated by a combination of two factors: First, when advising a before/after advice, only the advice body is advised, not the implicit proceed call. Second, all before/after advice should take into account the effects of their implicit proceed call (even if they are non-ASP-compliant). These two factors complicate the definition of effective pre/postconditions in method/proceed calls, which is why we decided to leave support for non-ASP-compliant higher-order advice as future work.

5.1 Relating the `@advisedBy` clause to quantification

At this point, the reader may wonder how the `@advisedBy` clause still preserves AOP's notion of quantification. After all, for every method call $obj{:}c.m$ that may be advised by a non-ASP advice, there should be an `@advisedBy` clause in $c.m$ that mentions this advice. At first glance, all the extra effort required to add all of these `@advisedBy` clauses seems to cancel out the benefits of having pointcuts as a quantification mechanism. However, it is possible to provide tool support that automatically inserts all `@advisedBy` clauses in the right places of the source code, and removes the need to write each clause manually. Given such tool support, we consider that the `@advisedBy` clause does not inhibit the quantification property of AOP.

Generating `@advisedBy` clauses in AspectJ is quite straightforward, where it is sufficient to inspect the pointcut of every advice. In a nutshell: If a pointcut makes use of a `call` construct, an `@advisedBy` clause should only be added to the method bodies specified directly in the `call` construct. In case a pointcut makes use of an `execution` construct, which only matches if the dynamic type is a certain (sub)type, the `@advisedBy` clause should be added into the types that declare the specified method bodies.

To take into account the fact that `call`/`execution` constructs also include subtypes, `@advisedBy` clauses are implicitly inherited by subtypes. In this manner, an `@advisedBy` clause will mention the desired advice when examining the static type of any method call that may be advised by that advice. Tool support that could generate `@advisedBy` clauses is actually already largely present in the AspectJ Development Tools [10], as it is quite similar to the markers that indicate each join point shadow.

In case of ContractAJ, some additional information is required to generate `@advisedBy` clauses in the right places. This is because it is possible to override advice, and an advice may be mentioned either directly or as a subtype in an `@advisedBy` clause, which allows for additional expressivity. Nonetheless, we assume ContractAJ can be easily extended with a simple construct to describe in which locations an `@advisedBy` clause should be added. For example, a class might contain the following "`declare @advisedBy`" statement:

```
declare @advisedBy Authentication.authenticate: Bank.set*(*);
```

Tool support can then make use of this information to insert `@advisedBy` clauses that list `Authentication.authenticate` in every method body that matches `Bank.set*(*)`. Finally, if multiple such statements want to add an `@advisedBy` clause to the same method, the program's precedence declaration is used to ensure that the advice listed in the `@advisedBy` clause are ordered correctly.

5.2 Interaction with ASP-compliant advice

Using the `@advisedBy` clause allows advice to alter the contracts of the join points they advise. However, what does this mean when such advice shares join points with an ASP-compliant advice (that is not mentioned in the `@advisedBy` clause)? An ASP-compliant advice only needs to take into account the advised join points' pre- and postconditions, but it can ignore any `@advisedBy` clauses. As a consequence, if both an ASP- and a non-ASP advice advise the same join point, the ASP-advice should get a lower precedence. That is, it should come after the non-ASP advice in an advice composition. While it is possible that the ASP-advice may never cause surprising behaviour if it had a higher precedence, this is not automatically ensured by the ASP. Aside from this constraint, the benefits of ASP-advice remain: An ASP-advice can be positioned anywhere after the non-ASP advice in advice compositions, while the remainder of the system does not have to be aware of this advice. In case it is required for an ASP-advice to be executed at a higher precedence, it always is possible to explicitly mention it in `@advisedBy` clauses.

5.3 Overriding advice

Because each advice in ContractAJ has a name, it becomes possible to override advice. Analogous to method overriding, when an advice in one aspect has the same name as an advice in one of its ancestor aspects, that advice is said to be overriding. However, overriding only has a purpose when the developer is *expecting* one advice body to be executed, but at runtime an overridden version might substitute for it. There are no such expectations if advice execution is implicit, which is the case for ASP-compliant advice.

The introduction of the `@advisedBy` clause gives purpose to overriding advice. When a method uses an `@advisedBy` clause, it is *expecting* those listed advice. These expectations, which can be determined statically, do not have to be fulfilled per se by exactly those listed advice. They might as well be implemented by a subaspect with an overriding advice. As can be seen in the example of Fig. 17, `Account.withdraw` is expecting to be advised by `Security.authenticate`. At runtime, this expectation could be filled in by a subaspect `RemoteSecurity`, which overrides the `authenticate` advice to e.g. implement a different authentication system. A similarly useful scenario could be that `Security` only is an abstract aspect, and the overriding advice in `RemoteSecurity` provides a concrete implementation. This notion of overriding advice implies that, similar to overriding methods in classes, overriding advice leave aspects "open for extension, but closed for modification" (the open/closed principle [29]). That is, an aspect's

```
class Security {
    around authenticate: call(int Account.withdraw(int i)) {...}}

class RemoteSecurity extends Security {
    around authenticate: call(int Account.withdraw(int i)) {...}}

class Account {
    @advisedBy Security.authenticate
    void withdraw(int i) {...}}

main {
    Security sec = new RemoteSecurity;
    Account acc = new Account(30);
    account.withdraw(10);}
// The @advisedBy clause in Account.withdraw lists Security
// , but is actually advised at runtime by RemoteSecurity.
```

Figure 17. Example of overriding advice

behaviour can be modified by defining a subaspect, without the need to modify any `@advisedBy` clauses.

As the use of overriding advice is similar to overriding methods, the SBS rules can be reused to ensure that aspects with overriding advice can substitute for any ancestor aspect without causing surprising behaviour. We only need to make a minor extension to the SBS such that the pre- and postcondition rules apply to both methods and advice:

Strong behavioural subtyping, extended (SBS'). *Type t is a strong behavioural subtype of type u, if and only if $t < u$ and:*

- *For all objects of type t, and for all common methods* ***and advice*** *x in t and u:*
  - *The precondition of $t.x$ must be equal to or weaker than the precondition of $u.x$.*
  - *The postcondition of $t.x$ must only be equal to or stronger than the postcondition of $u.m$, if the precondition of $u.m$ held in the pre-state.*
- *The invariant of u should be preserved in t.*

Note that ensuring the SBS' rules for an advice is independent from the join points it advises, even if the `proc` keyword is used. While the actual value of the `proc` keyword does depend on which join point is advised, we know that this value will be the same for the pre/postcondition of both $t.x$ and $u.x$. Consequently, it is not necessary to know `proc`'s value to determine whether one advice's pre/-postcondition is stronger or weaker than another. However, while the SBS' rules can be satisfied without considering the value of the `proc` keyword, an advice's implementation should of course correspond to its specifications. By simply using `proc` in the precondition of an advice (without knowing which contracts it refers to), we can ascertain that the right precondition is satisfied to make a proceed call at the very beginning of the advice body. However, if a proceed call is made at a later point in time, we need to make sure that this precondition is preserved. Because of this, we may still need to determine which contracts `proc` refers to. Likewise, we may need these contracts to ensure that the postcondition of a proceed call is preserved at the end of the advice.

5.4 Constraints on the `@advisedBy` clause

To avoid ambiguities when executing a call with an `@advisedBy` clause, there may not be multiple aspect instances that correspond to the same element in the `@advisedBy` clause. Moreover, if an advice is mentioned in the `@advisedBy` clause of a method, it is expected that this advice, or an overriding version, *will* be executed when that method is called (unless the advice's `if` pointcut construct failed). If the developer is free to instantiate aspects at any time, which is the case in ContractAJ, these two constraints cannot be enforced statically. The easiest solution would be to throw a runtime exception whenever the constraints are broken, which is also done by ContractAJ's contract enforcement algorithm in Sec. 7. Another approach would be to use AspectJ's more constrained instantiation mechanism, which ensures by construction there cannot be more than one instance of the same aspect at a given join point. While this approach avoids runtime exceptions, we still opted to give the developer full control over aspect instantiation in ContractAJ, as it would be difficult to combine AspectJ's instantiation mechanism with the notion of overriding advice in an intuitive/useful manner. We give preference to supporting overriding advice, because it leaves aspects open for extension in the presence of non-ASP advice.

5.5 Extending the effective specifications

Due to the introduction of the `@advisedBy` clause, applications may now contain both ASP- and non-ASP-compliant advice. However, this also requires some modifications to our definitions of effective specifications in Sec. 4.6. These new definitions can be found in Fig. 18. The eff_{pre} and $\mathit{effProc}_{pre}$ functions still have the same intuition behind them: eff_{pre} defines the effective precondition of method calls and $\mathit{effProc}_{pre}$ defines the effective precondition of proceed calls. However, note that $\mathit{effProc}_{pre}(c, m, i)$ has now gained a third parameter i, which indicates a certain position within the `@advisedBy` clause of $c.m$. More specifically, $\mathit{effProc}_{pre}(c, m, i)$ describes the effective precondition of any proceed calls within an advice mentioned at position i of $c.m$'s `@advisedBy` clause. This effective precondition essentially states that the advice at position i should become aware of all subsequent advice in the `@advisedBy` clause. Let us now take a closer look at each of the functions in Fig. 18:

[eff_{pre}] - The effective precondition of method calls $\mathit{eff}_{pre}(c, m)$, corresponds to $\mathit{effProc}_{pre}(c, m, 0)$. Because the first advice of the `@advisedBy` clause only has index 1, setting i to 0 intuitively means that we should become aware of all advice in the `@advisedBy` clause when making a method call, which indeed is the desired definition. An example of the effective precondition of a method call to `Bank.createAccount` is also given in Fig. 19.

$$\mathit{eff}_{pre}(c,m) = \mathit{effProc}_{pre}(c,m,0)$$

$$\mathit{effProc}_{pre}(c,m,i) = \begin{cases} \texttt{if(}\mathit{ifPcut}(c_{i+1}, a_{i+1})\texttt{)\{}\mathit{adv}_{pre}(c_{i+1}, a_{i+1}, c, m, i+1)\texttt{\}} \\ \texttt{else if(}\mathit{ifPcut}(c_{i+2}, a_{i+2})\texttt{)\{}\mathit{adv}_{pre}(c_{i+2}, a_{i+2}, c, m, i+2)\texttt{\}} \\ \ldots \\ \texttt{else if(}\mathit{ifPcut}(c_n, a_n)\texttt{)\{}\mathit{adv}_{pre}(c_n, a_n, c, m, n)\texttt{\}} \\ \texttt{else\{}\mathit{pre}(c,m)\texttt{\}} \end{cases}$$
$$\text{where } \mathit{isMethod}(c,m) \text{ and } \mathit{advBy}(c,m) = \langle c_1, a_1, \ldots, c_n, a_n \rangle \text{ and } 0 \le i < n$$

$$\mathit{effProc}_{pre}(c,m,i) = \mathit{pre}(c,m)$$
$$\text{where } \mathit{isMethod}(c,m) \text{ and } |\mathit{advBy}(c,m)| = 0$$

$$\mathit{effProc}_{pre}(c,a,i) = \mathit{pre}(c,a)[\texttt{proc} \mapsto \mathit{effProc}(c',x,j)]$$
$$\text{where } c.a \text{ advises } c'.x \text{ and } !\mathit{isMethod}(c,a)$$
$$\text{and } c.a \text{ is mentioned at position } j \text{ of the } \texttt{@advisedBy} \text{ clause of } c'.x$$

$$\mathit{adv}_{pre}(c,a,c',x,i) = \mathit{iPre}(c,a)[\texttt{proc} \mapsto \mathit{effProc}_{pre}(c',x,i+1)]$$
$$\text{where } c.a \text{ advises } c'.x$$

$$\mathit{iPre}(c,a) = \begin{matrix} \mathit{pre}(c,a) & \text{if } \mathit{isAround}(c,a) \\ \mathit{pre}(c,a)\ \texttt{\&\& proc} & \text{if } \mathit{isBefore}(c,a) \text{ or } \mathit{isAfter}(c,a) \end{matrix}$$

Figure 18. Definition of a method's effective preconditions

```
class Bank {
    @requires u.getBank()==this
    @ensures result!=null && result.getOwner()==u
    @advisedBy Security.authenticate, Security.authorize
    Account createAccount(User u) {...}}

class Security {
    @requires proc
    @ensures if(isLoggedIn(u)){proc}else{true}
    around authenticate: call(Account Bank.createAccount(User u))
        && target(b) {...}

    @requires isLoggedIn(u) && proc
    @ensures if(isAuthorized(u)){proc}else{true}
    around authorize: call(Account Bank.createAccount(User u))
        && target(b) && if(isEnabled()) {...}}

main {
    Bank b = new Bank;
    b.createAccount(new User);}
// Effective precondition of Bank.createAccount:
//  if(isEnabled()) {
//      isLoggedIn(u) && u.getBank()==this
//  } else {
//      u.getBank()==this
//  }
```

Figure 19. Example of an effective precondition

[$\mathit{effProc}_{pre}$] - The effective precondition of proceed calls is split up into three different cases. The second and third case are straightforward, as they correspond to the old definition of $\mathit{effProc}_{pre}$ in Sec. 4.6. The first case however is new: it describes the effective precondition of proceed calls where the advised method has a non-empty `@advisedBy` clause. Given i, the effective precondition that is produced consists of an `if` statement that iterates over all succeeding advice (from $i + 1$ to n) in the `@advisedBy` clause of $c.m$. What may be surprising is that we test the `if` pointcut construct for each of these advice (using the *ifPcut* function defined in Fig. 8). This is necessary because, if we allow the advice mentioned in the `@advisedBy` clause to modify the expected behaviour of a method/proceed call, we must know which advice will be executed next. Because the `if` constructs are the only[7] part of a ContractAJ pointcut that (in general) cannot be determined statically, they are included in the effective precondition itself.

[adv_{pre}] - Once we have determined which advice will be executed next, the adv_{pre} function is used to retrieve that advice's precondition. Additionally, adv_{pre} replaces any occurrences of the `proc` keyword with the effective precondition of any proceed calls in its advice body using $\mathit{effProc}_{pre}$, meaning that we recursively iterate through the remainder of the `@advisedBy` clause. This continues until we eventually reach the `else` branch in $\mathit{effProc}_{pre}$, which ends with the precondition of the advised method.

[*iPre*] - What is important to note about adv_{pre} is that we do not directly retrieve the precondition of an advice with the *pre* function, but we make use of $\mathit{iPre}(c, a)$ instead. In case $c.a$ is an around advice body, $\mathit{iPre}(c, a) = \mathit{pre}(c, a)$. However, if $c.a$ is a before or after advice, the `proc` keyword will also be conjoined. This is necessary because otherwise the effects of the implicit proceed call would not be included into the effective precondition. In other words, *iPre* makes the implicit proceed call of a before/after advice explicit, such that we obtain that advice's precondition as if it were an around advice. This also holds for the analogous *iPost* function, which conjoins `proc` to a before/after advice's postcondition. In general, we are not allowed to simply conjoin the `proc` keyword, e.g. because a before advice body might not preserve the precondition of its proceed call. However, this is remedied by the rules that will be defined in Sec. 5.6 and ensure that all before/after advice take into account their implicit proceed call.

This covers each function used to define the effective preconditions of method and proceed calls. The definitions for effective postconditions are analogous, as each occurrence of "pre" is simply replaced with "post".

We should note that, in general, the entire effective specification of a method/proceed call can become quite large and complex. However, the functions of Fig. 18 do take into account the worst case where every pointcut effectively makes use of an `if` construct (that can only be determined at runtime). How-

[7] An `execution` pointcut can only be determined dynamically, but we assume that it has already been converted into a `call` pointcut conjoined with an `if` construct, as described in Fig. 4 of Sec. 3.1.

ever, as indicated in Apel et al. [4], typically only a small fraction of aspect code makes use of such advanced features. In the best (and more common) case, each pointcut in an `@advisedBy` clause can be determined statically, drastically simplifying the entire effective precondition of a method call to the following:

$$\mathit{eff}_{pre}(c,m) = \begin{array}{l} \mathit{iPre}(c_1,a_1)[\texttt{proc} \mapsto \\ \quad \mathit{iPre}(c_2,a_2)[\texttt{proc} \mapsto \\ \qquad \ldots \\ \qquad\quad \mathit{iPre}(c_n,a_n)[\texttt{proc} \mapsto \mathit{pre}(c,m)] \ldots]] \\ \text{where } \mathit{isMethod}(c,m) \text{ and } \mathit{advBy}(c,m) = \langle c_1,a_1,\ldots,c_n,a_n\rangle \end{array}$$

That is, $c.m$ would use the precondition of the first advice in the `@advisedBy` clause, and each advice's `proc` keyword would be replaced with the precondition of the next advice in the clause.

5.6 Before and after advice

The *iPre* function in Fig. 18 includes the effects of the implicit proceed call in before/after advice by simply conjoining the `proc` keyword to its pre- and post-condition. However, in order for this to be correct, there are some rules that must be satisfied. These rules, which are a weaker version of the ASP for before/after advice, are referred to as the "implicit proceed rules" (IPR):

IPR for before advice. *Consider a before advice* `t.a` *that is applied to join point* `u.x`*. If* `t.a` *is mentioned in the* `@advisedBy` *clause of* `u.x`*, it is mentioned at position* i*. If* `t.a` *is not mentioned in the* `@advisedBy` *clause (or such a clause is not present) then* $i = 0$*. The before advice satisfies the IPR if and only if, for all objects of type* t*:*

– *If* $\mathit{effProc}_{pre}(u,x,i)$ *held before executing the advice, it should still hold after the advice.*
– *If* $\mathit{post}(t,a)$ *holds after the advice, it should still hold after the implicit proceed call.*
– *The invariant of* u *should be preserved in* t*.*

Likewise, the **IPR for after advice** are defined as:

– *If* $\mathit{effProc}_{post}(u,x,i)$ *held before executing the advice, it should still hold after the advice.*
– *If* $\mathit{pre}(t,a)$ *held before the implicit proceed call, it should still hold after the implicit proceed call.*
– *The invariant of* u *should be preserved in* t*.*

Note that the second rule in the IPR of both before- and after advice requires that the implicit proceed call should preserve a certain condition. As the advice body has no control over what the proceed call may or may not preserve, we can only rely on what we can determine to be preserved. If we are conservative, we can only ascertain that the proceed call will preserve anything that is equal to or weaker than its postcondition. However, a weaker notion of preservation is possible when frame conditions are available: In addition to preserving the proceed call's own postcondition, we know that it will preserve conditions that depend only on variables/fields that are *not* modified by the proceed call.

5.7 Modular reasoning in ContractAJ

To summarize Sec. 4 and 5, modular reasoning about method/proceed calls in ContractAJ can be achieved as follows:

– All classes should take into account the SBS' rules (Sec. 5.3). That is, a subtype should take into account the contracts of its supertype. This also holds for overriding advice.
– To remain oblivious of an advice, it should comply with the ASP (Sec. 4.1 and 4.2). That is, the advice should comply with the contracts of all join points it advises. If an ASP-compliant advice shares join points with advice mentioned in an `@advisedBy` clause, the ASP-compliant advice should have a lower precedence. (Sec. 5.2)
– If it is not possible or undesired to be oblivious of an advice $c.a$, it should be mentioned by an `@advisedBy` clause in all method bodies $c'.m$, where c' is the static type in any method calls to m that are advised by $c.a$. Advice $c.a$ may either be mentioned directly in an `@advisedBy` clause, or indirectly as an overridden advice. If $c.a$ is a before/after advice, it should comply with the IPR (Sec. 5.6) in order to take into account the effects of its implicit proceed call. Finally, the constraints of Sec. 5.4 should be taken into account as well.

In using this approach, there is little that changes from the perspective of an OOP developer who writes classes (without advice). This is especially important when considering that the bulk of a typical aspect-oriented application consists of regular classes and only a fraction of the code is made up of aspects [4]. When making a method call that only has ASP-compliant advice applied to it, nothing changes. When calling a method that has an `@advisedBy` clause, the listed advice need to be taken into account. This arguably does not add much complexity compared to reasoning about OOP applications: In OOP applications, the contracts of a method not only reflect the concern that it implements, but the contracts of any crosscutting concerns would have to be included as well. In AOP, these two are now separated, though the `@advisedBy` clause provides an explicit link needed to construct the effective specification. Moreover, the developer does not even have to construct this specification by him/herself, as it should be straightforward to build a tool that statically determines the effective pre- and postcondition of all method and proceed calls.

As for the developers writing advice, ensuring that the rules for either ASP or non-ASP advice hold can be done quite easily for advice that advise only a small number of join point shadows. As discussed in Sec. 4.3, the number of reasoning tasks grows with the number of advised join point shadows. In this case, the contracts of an advice can be written without rigorously inspecting each join point shadow, but tools like the contract enforcement algorithm of Sec. 7 should be used to help verify whether the advice satisfies the approach.

Finally, in case an advice is non-ASP-compliant, the right methods should mention the advice in an `@advisedBy` clause. As discussed in Sec. 5.1, tool support can be provided to generate these clauses. Because the use of an `@advisedBy` affects the effective specification of method calls, deciding on which `@advisedBy` clauses go where should be done as early as possible. Keep in mind that crosscutting concerns also form part of a system's design and aspects should therefore not be added as an afterthought. Nonetheless, using `@advisedBy` clauses still leaves the system open for extension, as the listed advice may also be implemented by subaspects.

6 Soundness of the approach

In Sec. 4 and 5 we presented our approach to modular reasoning in aspect-oriented languages. We now want to show that this approach is sound. That is, if a ContractAJ program is written such that it takes into account the SBS' rules, the ASP, IPR and the constraints on `@advisedBy` clauses, then all pre- and postconditions encountered in any program execution are satisfied. Note that we do not consider invariants: In OOP, the precondition and postcondition rules of SBS are known to be sound, but the invariant rule is not [24]. For example, because a subclass may strengthen its invariant, and because an invariant can be considered a part of each method's pre- and postcondition, it is possible to create conflicts with the precondition rule. Such problems with the invariant rule are carried over to the ASP as well. While achieving soundness in invariants could be done by only allowing an invariant to depend on certain parts of the program state, or by placing restrictions on ownership [26,31,32], we consider this to go beyond the paper's scope.

Aside from focusing only on pre- and postconditions, we do consider the various complex ways in which advice can be composed: Multiple advice can apply to the same join point; advice can depend on pointcuts that can only be determined at runtime; advice can be overridden, and higher-order advice (advice that advise other advice executions) are allowed. We will first define what constitutes a valid advice composition. Once this is done, we can show that our approach preserves modular reasoning in a number of theorems: Initially programs with only ASP-compliant advice are considered. We then add support for advice mentioned in an `@advisedBy` clause and finally allow for higher-order advice.

6.1 Valid advice compositions

In Sec. 5.4 we discussed two dynamic constraints that should hold for each advice composition, in order to avoid ambiguities in `@advisedBy` clauses. To show that our approach to modular reasoning is sound, we should first make these constraints precise by defining what constitutes a valid advice composition. An advice composition is a sequence $\langle c_1 . a_1, \ldots, c_n . a_n, c . x\rangle$, where each $c_i . a_i$ represents an advice body that will be executed, in the given precedence order, and $c . x$ represents the body advised by each $c_i . a_i$. We can now define a *valid* advice composition (for method call join points) as follows:

Valid advice composition *(for method calls). During the evaluation of program P, consider a join point that represents a method call obj:$c . m$. The method call corresponds to an advice composition $\langle c_1 . a_1, \ldots, c_n . a_n, c . m\rangle$. Method body $c . m$ has the following valid* `@advisedBy` *clause:* `@advisedBy` *$d_1 . b_1, \ldots, d_k . b_k$,where $k \geq 0$. The advice composition is valid if and only if there exists an index j, $0 \leq j \leq k$ such that:*

- *There is an order-preserving, injective function f from $(c_1 . a_1, \ldots, c_j . a_j)$ to $(d_1 . b_1, \ldots, d_k . b_k)$ such that, for each i, either $c_i . a_i = f(c_i . a_i)$ or $c_i . a_i$ overrides $f(c_i . a_i)$.*
- *For each $d_i . b_i$ in $(d_1 . b_1, \ldots, d_1 . b_k)$ the pointcut of $d_i . b_i$ matches on obj:$c . m$ if and only if $d_i . b_i \in \{f(c_1 . a_1), \ldots, f(c_j . a_j)\}$.*

This definition states that a valid advice composition can be divided into two parts: $(c_1 . a_1, \ldots, c_j . a_j)$ represents the advice mentioned in the `@advisedBy` clause and $(c_{j+1} . a_{j+1}, \ldots, c_n . a_n)$ represents the ASP-compliant advice, which have a lower precedence. The function f is order-preserving to ensure that each advice mentioned in the `@advisedBy` clause respects the precedence order specified by the `@advisedBy` clause. (This is already ensured statically by the constraints in Sec. 3.4) The function is injective to prevent that two advice in the composition could be mentioned by the same element in the `@advisedBy` clause. Finally, we require that the pointcut of each advice in the `@advisedBy` clause is respected, such that it matches the expectations created by the effective pre- and post-conditions of $c . m$. This constraint only is needed because an overriding advice might use a different pointcut than the advice it overrides.

The definition of valid advice compositions for advice execution join points is trivial: Because advice cannot have an `@advisedBy` clause, all advice compositions on advice executions are valid.

6.2 Shared join points

We will now show that our approach is sound when multiple advice can share the same join points. That is, the developer can assume that all effective pre/postconditions of any method/proceed calls and all pre/postconditions of method and advice bodies during program execution will hold, on the condition that our approach is taken into account and that he/she ensures that the implementation of each advice/method body correctly implements its own specifications. In doing so the developer is allowed to assume modular reasoning at any method call. To make these conditions more precise, we first define what it means when "modular reasoning about a method call is possible":

Definition 1. *Modular reasoning about a method call* $obj\texttt{:}c.m$ *during an execution of a program* P *is possible if the following holds: Let* $\langle e, \mathcal{S}, \mathcal{J} \rangle$ *be a configuration in* P*, where* e *decomposes to a context with method call* $obj\texttt{:}c.m$*. If* $\mathit{eff}_{pre}(c,m)$ *is satisfied in this configuration, then* $\mathit{eff}_{post}(c,m)$ *is satisfied in the configuration that represents the end of the method call.*

An analogous definition can be given for proceed calls. Intuitively, the developer should ensure the preconditions of the next advice that this proceed call will execute. However this developer can, and should, only determine which advice comes next based on the `@advisedBy` clauses where the current advice is mentioned, which can be statically determined using the *effProc* functions of Sec. 5.5.

Definition 2. *Modular reasoning about a proceed call during an execution of a program* P *is possible if the following holds: Let* $\langle e, \mathcal{S}, \mathcal{J} \rangle$ *be a configuration in* P*, where* e *decomposes to a context with a proceed call. Let this proceed call be part of the execution of an advice body* $c'.a$ *that advises method* $c.m$ *and assume that either* $c'.a$ *satisfies the ASP or it is mentioned at position* i *of* $c.m$*'s* `@advisedBy` *clause. If* $\mathit{effProc}_{pre}(c,m,i)$ *is satisfied in this configuration,* $\mathit{effProc}_{post}(c,m,i)$ *is satisfied in the configuration that represents the end of the proceed call.*

Next, we can define what it means for an advice/method body to correctly implement its own specifications; we refer to this property as "local correctness":

Definition 3. *An advice/method body* x *in class* c *is locally correct in a program* P *if the following holds:*

- *If, at a method/proceed call in any execution of body* $c.x$ *in* P*: (1)* $pre(c,x)$ *was satisfied at the beginning of executing* $c.x$*. (2) Modular reasoning is possible for all prior method/proceed calls in the execution of* $c.x$*.*
  Then: the effective precondition of that method/proceed call is satisfied.
- *If, in any execution of body* $c.x$ *in* P*: (1)* $pre(c,x)$ *was satisfied at the beginning of executing* $c.x$*. (2) Modular reasoning is possible for all method/proceed calls in the execution of* $c.x$*.*
  Then: $post(c,x)$ *is satisfied when the execution of* $c.x$ *finishes.*

In Theorem 1, we start by considering the execution of ContractAJ programs that only contain ASP-compliant advice, no `@advisedBy` clauses and no higher-order advice. Theorem 2 then extends the first theorem with the use of the `@advisedBy` clause. We assume that all pre- and postconditions are free from side-effects and always terminate. Likewise, we also assume that the conditions in any `if` pointcut constructs are free from side-effects and always terminate, as they should only be used to determine whether an advice matches, rather than altering the program's state.

Theorem 1. *Let* P *be a program without any* `@advisedBy` *clauses or higher-order advice. The effective pre/postcondition of each method/proceed call (including implicit proceed calls) and the pre/postcondition of each method/advice* *body during the evaluation of P will be satisfied if: (1) All advice in P satisfy the ASP rules. (2) All classes in P satisfy the SBS' rules. (3) All method/advice bodies in P are locally correct. (4) The initial precondition of P is satisfied at the start of the program.*

We only need to focus on those points during the execution where a method/advice body is about to start or about to end. It is only at those points that we need to check whether the pre/postcondition of a method/advice body is satisfied, as well as the effective pre/postcondition of the method/proceed call that initiated the body's execution. Any other point in the execution of P is irrelevant, as there are no method/advice contracts to be considered.

The proof proceeds by induction on the length of the reduction sequence leading to such relevant points, with a case analysis on the last steps. In the base case, 0 steps, P's initial precondition is satisfied by (4). In the inductive step, consider the start of a method/advice body. The "start of a body" does not refer to just one particular configuration in the evaluation of a program, but rather to the entire sequence of configurations that decomposes a method/proceed call into the body that will be executed. These sequences are precisely described by Fig. 10 in Sec. 3.3, which specifies all possible sequences of rule applications during the lookup process. During the entirety of such a sequence, both the effective precondition of the method/proceed call and the precondition of the body to be executed must hold. Note that it is sufficient to show this for any one configuration in this sequence, because none[8] of the rules in Fig. 10 can modify the program's store.

(A) Body initiated via method call - Consider the case where the execution of a body is initiated by a method call, as described by the *mcall* sequence in Fig. 10. Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration in P where e decomposes into a method call $obj\colon c.m$. (Keep in mind that c represents the receiver's static type.)

(A.1) We first show that $pre(c, m)$ is satisfied in this configuration, using (3) and induction on the length of the execution: Let $c'.x$ be the body that contains the method call $obj\colon c.m$. By (3), we know $c'.x$ is locally correct. By induction, the pre/postconditions in all configurations prior to $\langle e, \mathcal{S}, \mathcal{J} \rangle$ were satisfied. This implies that $pre(c', x)$ was satisfied at the start of $c'.x$, and moreover modular reasoning is possible for all method/proceed calls in the execution of $c'.x$, prior to the call $obj\colon c.m$. Now local correctness of $c'.x$ indeed ensures that $pre(c, m)$ holds at $\langle e, \mathcal{S}, \mathcal{J} \rangle$. (We will reuse this reasoning a few times; we will use "by (3)+induction" as a shorthand for it.) Because there are no `@advisedBy` clauses in P, $\mathit{eff}_{pre}(c, m) = pre(c, m)$, and hence the effective precondition of method call $obj\colon c.m$ holds.

(A.2) The *mcall* sequence of Fig. 10 always leads to the execution of either a method, before advice or around advice body. It remains to be shown that the precondition of this body holds when its execution starts. Consider configuration $\langle e', \mathcal{S}, \mathcal{J}' \rangle$, in which the [**exec**] rule at the end of *mcall* is applied:

[8] The evaluation of the condition in `if` pointcut constructs might involve store-altering rule applications. However, this is harmless as we assume that these conditions are free from side effects and always terminate.

- **Method body** - Let $c''.m$ be the method body that is executed by our method call $obj\!:\!c.m$. Because $pre(c, m)$ holds, and because SBS' ensures that $pre(c'', m)$ cannot be stronger than $pre(c, m)$ (2), the precondition of the method body $pre(c'', m)$ is satisfied.
- **Before/around advice body** - Let $c''.a$ be the before/around advice initiated by $obj\!:\!c.m$. Because $pre(c, m)$ holds, and because the ASP (1) ensures that $pre(c'', a)$ cannot be stronger than $pre(c, m)$, the precondition of the advice body $pre(c'', a)$ is satisfied.

(B) Body initiated via proceed call - Now consider the case where the execution of a body is initiated by a proceed call, as described in the *pcall* sequence of Fig. 10. Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes into a proceed call (i.e. a `proceed` not produced by the [**call**] rule, as this corresponds to the *mcall* sequence). Let $c.m$ be the method that is being advised by $c'.a$, the advice that initiated this proceed call. If $c'.a$ is an around advice or an after advice, the effective precondition of this proceed call $\mathit{effProc}_{pre}(c, m)$ holds by (3)+induction. If $c'.a$ is a before advice, we only know by (3)+induction that $\mathit{effProc}_{pre}(c, m)$ is satisfied when the execution of $c'.a$ starts. However, because the ASP (1) requires that before advice preserve this effective precondition, $\mathit{effProc}_{pre}(c, m)$ still holds at $\langle e, \mathcal{S}, \mathcal{J} \rangle$. We now know $\mathit{effProc}_{pre}(c, m)$ always holds at $\langle e, \mathcal{S}, \mathcal{J} \rangle$. As there are no `@advisedBy` clauses, $\mathit{effProc}_{pre}(c, m) = pre(c, m)$. Because $\mathit{effProc}_{pre}(c, m)$ holds, and because the *pcall* expression is identical to *mcall* (apart from the initial application of [**call**]), we can reuse the reasoning of (A.2) to conclude that the precondition of the body to be executed is satisfied in this case as well.

(C) Initiating after advice bodies - As the execution of an after advice body cannot be initiated directly by a method/proceed call, it is treated as a separate case. For the same reason, we only need to show that the after advice body's precondition holds; there is no effective precondition that needs to hold. As discussed in Sec. 3.3, the execution of an after advice body corresponds to the rule sequence "[**return**] *mcall*" where [**return**] indicates the end of the implicit proceed call. Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be the configuration where we are about to apply the [**call**] rule in *mcall*. In this configuration, e decomposes to $obj\!:\!c.a$, representing a "call" to the after advice to be executed. (Keep in mind that the operational semantics internally uses method calls to execute advice bodies.) Let $c'.m$ be the method advised by $c.a$. In case the implicit proceed call preceding $c.a$ ended with another after advice body, we know $\mathit{effProc}_{post}(c', m)$ held at the beginning of executing this body, by (3)+induction. By the ASP (1), we know that $\mathit{effProc}_{post}(c', m)$ still holds at $\langle e, \mathcal{S}, \mathcal{J} \rangle$, as after advice are required to preserve the effective postcondition. In any other case, the preceding implicit proceed call directly ensures $\mathit{effProc}_{post}(c', m)$ at $\langle e, \mathcal{S}, \mathcal{J} \rangle$ by (3)+induction. As there are no `@advisedBy` clauses, $\mathit{effProc}_{post}(c', m) = post(c', m)$. Because the ASP (1) requires that $pre(c.a)$ may not be stronger than $post(c', m)$, the precondition of the after advice body $pre(c, a)$ holds.

(D)Returning from a body - After covering every possible way to initiate the execution of a body, we still need to do the analogue for the end of executing a body. In short: We now only need to consider the [**return**] rule, as it always represents the end of a body's execution. Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes into a `return` expression. By (3)+induction, the body's postcondition holds. In case of method, after advice and around advice bodies, we still need to show that the effective postcondition holds of the method/proceed call that initiated the execution of this body. For method bodies, this is ensured by SBS' (2). For around/after advice bodies, this is ensured by the ASP (1). In case of a before advice body, there is no effective postcondition to be shown at $\langle e, \mathcal{S}, \mathcal{J} \rangle$, due to the presence of the implicit proceed call that follows.$\square$

Theorem 2. *Let P be a program without any higher-order advice. The effective pre/postcondition of each method/proceed call (including implicit proceed calls) and the pre/postcondition of each method/advice body during the evaluation of P will be satisfied if: (1) All advice in P that are not mentioned in any `@advisedBy` clauses satisfy the ASP rules. (2) All classes in P satisfy the SBS' rules. (3) All method/advice bodies in P are locally correct. (4) The initial precondition of P is satisfied at the start of the program. (5) Each advice composition is valid. (6) All before/after advice satisfy the IPR rules. (7) Advice mentioned in an `@advisedBy` clause have a higher precedence than advice not mentioned in any `@advisedBy` clauses.*

This proof is an extension of Theorem 1's proof; the main difference is that we now have a distinction between those advice that are mentioned in an `@advisedBy` clause, and those that are not.

Body initiated via method call - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes to a method call $obj : c.m$. By (3)+induction, the effective precondition of the method call, $\mathit{eff}_{pre}(c, m) = \mathit{effProc}_{pre}(c, m, 0)$, is ensured at $\langle e, \mathcal{S}, \mathcal{J} \rangle$. Let $c'.a$ be the body to be executed.

- In case $c'.a$ is an ASP-compliant around/before advice or a method body, then either $c.m$ does not have an `@advisedBy` clause, or the `if` pointcut constructs of each advice in the `@advisedBy` clause must have evaluated to `false` (because each advice composition must be valid (5)). In both cases, we know that $\mathit{effProc}_{pre}(c, m, 0) = \mathit{pre}(c, m)$, which corresponds to the situation after (A.1) in the proof of Theorem 1. Consequently, we may conclude that the precondition of the method/ASP-compliant body to be executed is satisfied.
- If $c'.a$ is an around/before advice mentioned at position i of the `@advisedBy` clause of $c.m$, then it follows from (5) that all preceding `if` pointcut constructs have evaluated to `false`. We now know that $\mathit{effProc}_{pre}(c, m, 0) = \mathit{adv}_{pre}(c_i.a_i, c, m, i)$, where $c_i.a_i$ is the i^{th} advice in the `@advisedBy` clause of $c.m$ and $c'.a \leq c_i.a_i$. Because $c'.a$ satisfies SBS' (2) and its precondition may not be stronger than $\mathit{adv}_{pre}(c_i.a_i, c, m, i)$, $\mathit{adv}_{pre}(c'.a, c, m, i)$ is satisfied as well, which is the precondition of $c'.a$ (with the `proc` keyword adjusted to the current join point).

Body initiated via proceed call - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes to a proceed call which executes body $c.a$. Consider that this proceed call is located in $c'.a'$, which advises $c''.m$. Method $c''.m$ has an `@advisedBy` clause with n elements, where $n \geqslant 0$.

- If $c'.a'$ is an ASP-compliant advice, $\mathit{pre}(c'', m)$ is satisfied. If it is mentioned as the last advice in the `@advisedBy` clause of $c''.m$, $\mathit{effProc}_{\mathit{pre}}(c'', m, n) = \mathit{pre}(c'', m)$ is satisfied. Both cases correspond to the situation after (A.1) in the proof of Theorem 1, and we know the body to be executed must be either ASP-compliant or a method body, due to (7). We conclude that the body's precondition holds as well.
- If $c'.a'$ is mentioned at position i of the `@advisedBy` clause, where $i \neq n$, then by (3)+induction (and the IPR if $c'.a'$ is a before advice), $\mathit{effProc}_{\mathit{pre}}(c'', m, i)$ is satisfied. Let the body to be executed at $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be located at position j, where $i < j \leq n$. Due to (5), all `if` constructs between i and j must have evaluated to `false`, so $\mathit{effProc}_{\mathit{pre}}(c'', m, i) = \mathit{adv}_{\mathit{pre}}(c_j, a_j, c'', m, j)$. Because $c.a \leq c_j.a_j$ and due to SBS', $\mathit{adv}_{\mathit{pre}}(c, a, c'', m, j)$ is satisfied as well.

Initiating after advice bodies - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes to $obj\,{:}\,c.a$, representing the "call" to an after advice body. By (3)+induction (and the IPR if the preceding proceed call ended with another after advice), $\mathit{effProc}_{\mathit{post}}(c', m, i)$ is ensured by the preceding proceed call. Due to the IPR rules, $\mathit{pre}(c, a)$ holds.

Returning from a body - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes to a `return` expression, where the execution of a body $c.x$ is about to finish. By (3)+induction, its postcondition holds. If $c.x$ is a method body, the SBS' ensures the effective postcondition of the method/proceed call that initiated $c.x$. If $c.x$ is an after/around advice not mentioned in the `@advisedBy` clause, this is ensured by the ASP. Finally, if $c.x$ is an after/around advice mentioned at position i of the `@advisedBy` clause of $c'.m$, then SBS' implies that $\mathit{adv}_{\mathit{post}}(c_i, a_i, c', m, i)$ holds. Due to (5), the `if` condition of $c.a$ must have been the first to evaluate to `true` at the method/proceed call initiating the execution of $c.a$. In case of a method call, we know that $\mathit{adv}_{\mathit{post}}(c_i, a_i, c', m, i) = \mathit{effProc}_{\mathit{post}}(c', m, \mathit{0}) = \mathit{eff}_{\mathit{post}}(c', m)$. In case of a proceed call, $\mathit{adv}_{\mathit{post}}(c_i, a_i, c', m, i) = \mathit{effProc}_{\mathit{post}}(c', m, j)$, where $0 < j < i$.$\square$

6.3 Higher-order advice

As ContractAJ's pointcuts can also match on the execution of advice, it is possible to create higher-order advice: advice with pointcuts that match on other advice executions. For example, it is possible that an advice `myAdvice` intercepts executions of `myMethod`, and there is another advice `myMetaAdvice` which intercepts executions of `myAdvice`. We will refer to `myAdvice` as a first-order advice and `myMetaAdvice` as a second-order advice. There also is no restriction on the number of orders, so there may be a third-order `myMetaMetaAdvice` which advises `myMetaAdvice`, and so on. Note that "the order of an advice" is not a fixed number; it can change per join point, as the same pointcut can match on advice executions of different orders.

Note that all higher-order advice must be ASP-compliant, which is a consequence of the fact that we only allow `@advisedBy` clauses to be specified for method bodies, not for advice bodies. This implies that a higher-order advice cannot be mentioned in an `@advisedBy` clause.

We can now show once more that, if our approach to modular reasoning is used, the effective pre/postconditions of each method/proceed call and the pre/postconditions of each method/advice body will be satisfied. While we allow the use of higher-order advice, we do make the assumption that the order of an advice must be a finite number. This is necessary to avoid situations where an advice (directly or indirectly) advises itself and creates an infinite recursion. In this case, we of course cannot show that the advice's postcondition would ever hold, as the program no longer terminates.

Theorem 3. *Let P be program where higher-order advice (of finite order) are allowed. The effective pre/postcondition of each method/proceed call (including implicit proceed calls) and the pre/postcondition of each method/advice body during the evaluation of P will be satisfied if: (1) All advice in P that are not mentioned in any* `@advisedBy` *clauses satisfy the ASP rules. (2) All classes in P satisfy the SBS' rules. (3) All method/advice bodies in P are locally correct. (4) The initial precondition of P is satisfied at the start of the program. (5) Each advice composition is valid. (6) All before/after advice satisfy the IPR rules. (7) Advice mentioned in an* `@advisedBy` *clause have a higher precedence than advice not mentioned in any* `@advisedBy` *clauses.*

The proof extends that of Theorem 2. In this previous theorem, *lookup* could only match at most once within the *mcall* and *pcall* rules of Fig. 10. Because higher-order advice are now allowed, *lookup* may now match multiple times.

Body initiated via method call - Consider a configuration $\langle e, \mathcal{S}, \mathcal{J} \rangle$ in the evaluation of P. Let e decompose into a context where we are about to execute a method call $obj\texttt{:}c.m$. By (3)+induction, $\mathit{eff}_{pre}(c, m)$ is satisfied. In case the body to be executed is either a method body or a first-order before/around advice body, this corresponds to cases covered by Theorem 2. If the body to be executed is an n^{th}-order before/around advice $c_n.a_n$, it can be shown that its precondition will be satisfied by transitively applying the ASP: In order for the lookup procedure to reach this n$^{\text{th}}$-order advice body, there must have been $n - 1$ lower-order advice ($c_1.a_1$, …, $c_{n-1}.a_{n-1}$) that matched first. That is, $c_n.a_n$ advises $c_{n-1}.a_{n-1}$, which in turn advises $c_{n-2}.a_{n-2}$, and so on, until we end up at $c_1.a_1$ which advises the $obj\texttt{:}c.m$ call. Due to the ASP (1), $pre(c_1, a_1)$ may not be stronger than $pre(c, m)$. In turn, $pre(c_2, a_2)$ may not be stronger than $pre(c_1, a_1)$, and so on. Because of this, we can conclude that $pre(c_n, a_n)$ is satisfied, as it cannot be stronger than $pre(c, m)$.

Body initiated via proceed call - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration, where e decomposes into a context with a proceed call. Let this proceed call be located in an n^{th}-order advice body $c_n\,.\,a_n$, $n \geq 1$. The effective precondition of this proceed call is ensured by (3)+induction (and the ASP if $c_n\,.\,a_n$ is a before advice). The execution of the proceed call at $\langle e, \mathcal{S}, \mathcal{J} \rangle$ can lead to either another n^{th}-order advice body (the next advice in the composition), an advice with an order greater than n (if the next n^{th}-order advice that would otherwise be executed is being advised) or an $n - 1^{th}$-order body (if we are executing the last element of the composition). Knowing that the order of the body to be executed is at least $n - 1$, showing that its precondition holds can be divided into three cases:

– If $c_n \,.\, a_n$ is a first-order advice and its proceed call leads to a method body or a first-order advice body, this situation corresponds to Theorem 2.
– If $c_n \,.\, a_n$ is a first-order advice and its proceed call leads to a higher-order advice body, we can apply the reasoning of Theorem 2 and conclude that the precondition holds of the first-order advice that would normally be executed, in the absence of any matching higher-order advice. Knowing that this precondition holds, the ASP can be applied transitively to conclude that the precondition holds of the higher-order advice that is actually executed.
– Finally, if $c_n \,.\, a_n$ is a higher-order advice, the effective precondition of its proceed call takes into account the precondition of $c_{n-1} \,.\, a_{n-1}$. By transitively applying the ASP, we know the precondition of the body to be executed holds, as its order must be $n - 1$ or greater.

Initiating after advice bodies - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration, where e decomposes into a context where an n^{th}-order after advice is about to be executed. If it is a first-order advice, this corresponds to Theorem 2. By (3)+induction (and the ASP if the preceding proceed call ended with another after advice), the effective postcondition of the preceding implicit proceed call holds. Consequently, $pre(c_n.m_n)$ holds by the ASP.

Returning from a body - Let $\langle e, \mathcal{S}, \mathcal{J} \rangle$ be a configuration where e decomposes to a `return` expression, where the execution of a body $c \,.\, x$ is about to finish. By (3)+induction, $post(c, x)$ holds. In case $c \,.\, x$ is a method body or a first-order advice body, which is initiated by a method call, or a proceed call in a first-order advice, these cases correspond to Theorem 2. In any other case, the ASP can be applied transitively to show that the effective postcondition holds of the call that initiated $c.x$.$\square$

7 Runtime contract enforcement

To enforce correct usage of our approach to modular reasoning, this section presents a runtime contract enforcement algorithm for the ContractAJ language. This algorithm produces an error whenever a contract is broken, and determines which type is to blame. Because the contract enforcement algorithm needs access to some additional join point information, we will first define a few extensions to expose this information. The contract enforcement algorithm is then specified by means of a transformation that adds contract enforcement aspects to a given ContractAJ program. Finally, we also discuss the algorithm's implementation in AspectJ, to demonstrate that it can also be applied to a full-fledged aspect-oriented programming language.

7.1 ContractAJ extensions

In this section, a few extensions are made the ContractAJ language so that we can retrieve additional information about each advised join point, similar to the notion of the `thisJoinPoint` variable in AspectJ. This runtime information is needed to be able determine which contracts need to be checked.

Additional run-time information is also needed in case a higher-order around advice makes use of the `proc` keyword. To evaluate the `proc` keyword within an n$^{\text{th}}$-order around advice, we should be able to determine which n-1$^{\text{th}}$-order advice it advises. In turn, if this n-1$^{\text{th}}$-order advice uses `proc`, we need to determine the n-2$^{\text{th}}$-order it advises, and so on.

To make this information available, we will first extend the $\mathcal{E}$ tuples used in the join point stack with a fifth element:

$$\mathcal{E} ::= \langle c, x, obj, bool, \mathcal{E}|\bullet\rangle$$

When considering an $\mathcal{E}$ tuple that represents a particular advice execution, this fifth element is used as a pointer to the $\mathcal{E}$ tuple of its advised join point. We will refer to this pointer as the AP (advised join point pointer). For example, consider the execution of an advice $c.a$ that advises method $c'.m$. This results in the following tuple for $c.a$:

$$\langle c, a, \_, \_, \mathcal{E}_{AP}\rangle \text{ where } \mathcal{E}_{AP} = \langle c', m, \_, \_, \bullet\rangle$$

The AP in the tuple of $c.a$ refers to the tuple of $c'.m$. In turn, the AP in the tuple of $c'.m$ is empty, as it represents a method. We can now use this AP to navigate from an n$^{\text{th}}$-order advice to all lower-order advice it advises (i.e. with an order smaller than n). To ensure that the AP is filled in correctly, the [**call**], [**around**] and [**call$_{\textbf{around}}$**] rules of the operational semantics are modified accordingly in Fig. 20. The [**epop**] rule is added as well, and ContractAJ's syntax is extended with the `epop` expression:

[**call**]
$$\begin{aligned}&P \vdash \langle \mathsf{E}[obj\texttt{:}c\texttt{.}x\texttt{(}v_1 \ldots v_n\texttt{)}], \mathcal{S}, \mathcal{J}\rangle\\ &\hookrightarrow \langle \mathsf{E}[\texttt{jpop\{proceed(}obj\ v_1\ \ldots\ v_n\texttt{)\}}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle\\ &\text{where } \mathcal{A} = alook(c, x, \mathcal{S}, \mathcal{E}) + \mathcal{E} \text{ and } \mathcal{E} = \langle c, x, obj, \texttt{true}, \bullet\rangle \text{ and } !isAround(c, x)\end{aligned}$$

[**around**]
$$\begin{aligned}&P \vdash \langle \mathsf{E}[\texttt{proceed(}obj\ v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle\\ &\hookrightarrow \langle \mathsf{E}[\texttt{if(}e'\texttt{)\{}obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}obj\ v_1 \ldots v_n\texttt{)\}else\{epop();proceed(}obj\ \ v_1 \ldots v_n\texttt{)\}}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle\\ &\text{where } \mathcal{A} = \langle c, a, obj_{asp}, \texttt{false}, \mathcal{E}\rangle + \ldots\\ &\text{and } isAround(c, a) \text{ and } var_{tgt} = target(c, a) \text{ and } e = ifPcut(c, a)\\ &\text{and } e' = e[obj_{asp}/\texttt{this}, obj/var_{tgt}, v_1/var_1, \ldots, v_n/var_n]\end{aligned}$$

[**call$_{\textbf{around}}$**]
$$\begin{aligned}&P \vdash \langle \mathsf{E}[obj_{asp}\texttt{:}c\texttt{.}a\texttt{(}v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{E} + \mathcal{A}; \mathcal{J}\rangle\\ &\hookrightarrow \langle \mathsf{E}[\texttt{proceed(}obj_{asp}\ v_1\ \ldots\ v_n\texttt{)}], \mathcal{S}, \mathcal{A}' + \mathcal{A}; \mathcal{J}\rangle\\ &\text{where } \mathcal{A}' = alook(c, a, \mathcal{S}, \mathcal{E}'') + \mathcal{E}'' \text{ and } isAround(c, a)\\ &\text{and } \mathcal{E} = \langle c, a, obj_{asp}, \texttt{false}, \mathcal{E}'\rangle \text{ and } \mathcal{E}'' = \langle c, a, obj_{asp}, \texttt{true}, \mathcal{E}'\rangle\end{aligned}$$

[**epop**]
$$P \vdash \langle \mathsf{E}[\texttt{epop()}], \mathcal{S}, \mathcal{E} + \mathcal{A}; \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[\texttt{true}], \mathcal{S}, \mathcal{A}; \mathcal{J}\rangle$$

Figure 20. Adding support for the AP

[**call**] - In case of the [**call**] rule, which handles method calls, the AP of the tuple that represents the method ($\mathcal{E}$) is empty. The *alook* function (not shown here) can be easily extended such that the AP of the tuples for all matching advice is set to $\mathcal{E}$. Note that the execution of a before/after advice body is also handled via the [**call**] rule, meaning that their AP is also set to $\bullet$. This is not a problem, as we only use the AP to resolve the value of the `proc` keyword, which can only be used in the pre/postcondition of around advice.

[**around**] - The main modification made to this rule, which processes around advice, is that it no longer pops the around advice's tuple from the stack. This is done because we still need its AP if e' evaluates to `true` and the obj_{asp}:c.a call is processed by [$\mathbf{call_{around}}$]. However, if e' evaluates to `false` the tuple is not needed and should be removed, which is done by the `epop()` statement.

[$\mathbf{call_{around}}$] - When looking for higher-order advice matching on an around advice, the tuples of these matching advice will get $\mathcal{E}$" as their AP, representing the around advice. In turn, the AP of $\mathcal{E}$", which is $\mathcal{E}$', can be found in the tuple that was left on the stack by the application of [**around**].

[**epop**] - This rule simply pops the top tuple from the join point stack.

After introducing the AP, we can use it to define the helper functions in Fig. 21: The $\mathit{jpElem}(\mathcal{J}, i, j)$ function is used to retrieve a particular $\mathcal{E}$ tuple from the join point stack, relative to the tuple at the top of the stack. Given that this top $\mathcal{E}$ tuple is an n$^{\text{th}}$-order advice, we make use of $\mathit{jpElem}(\mathcal{J}, \mathcal{E}, i, j)$ to navigate to the $\mathcal{E}$ tuple of the n-i$^{\text{th}}$-order advice. If this tuple is the k$^{\text{th}}$ element in the $\mathcal{A}$ record that contains it, the k+j$^{\text{th}}$ element is finally returned.

The two *advBy* functions relate to the `@advisedBy` clause of a method: One determines the position at which an advice is mentioned in an `@advisedBy` clause, whereas the other retrieves an advice's class at a certain position in the clause. Both make use of the $\mathit{advBy}(c, m)$ function (defined in Fig. 12 of Sec. 3.4) to retrieve the complete `@advisedBy` clause of a method, which includes the clause inherited from the super class.

Finally, using *jpElem* and *advBy*, we can extend the ContractAJ language with the following new keywords that will be used to define the contract enforcement algorithm: `advBy`, `jpStatic`, `jpThis`, `pre`, `iPre`, `effProcPre` and `advPre`. Their semantics is defined in Fig. 22:

[**advBy**] - The `advBy(`c, a, c', m`)` expression directly uses the *advBy* function to retrieve the position in which advice c.a is mentioned in the `@advisedBy` clause of c'.m.

[**jpStatic**] - If the $\mathcal{E}$ tuple at the top of $\mathcal{J}$ is an n$^{\text{th}}$-order advice, the `jpStatic(`i`)` expression uses *jpElem* to retrieve the type of the n-i$^{\text{th}}$-order body being advised.

[**jpThis**] - This expression is similar to `jpStatic`, but it retrieves the instance corresponding to the n-i$^{\text{th}}$-order body being advised.

[**pre**] - The `pre(`$c, x, obj, obj', proc, v_1 \ldots v_n$`)` expression retrieves the precondition e of body c.x, as well as binds all variables in e to their values. These variables respectively constitute of the `this` object, the `target` binding, the `proc` keyword and the arguments of c.x.

Retrieve the j^{th} element in an $(n-i)^{th}$-order composition, with an n^{th}-order element at the top of $\mathcal{J}$

$$jpElem(\mathcal{J}, i, j) \qquad \frac{\mathcal{J} = \mathcal{A}\,;\ \ldots \text{ and } \mathcal{A} = \mathcal{E} + \ldots \text{ and } \mathcal{E}' = jpElem(\mathcal{J}, \mathcal{E}, i, j)}{jpElem(\mathcal{J}, i, j) = \mathcal{E}'}$$

$$jpElem(\mathcal{J}, \mathcal{E}, i, j) \qquad \frac{i = 0 \text{ and } (\mathcal{J} = \langle \ldots, \langle \mathcal{E}_1, \ldots, \mathcal{E}_k, \ldots, \mathcal{E}_{k+j}, \ldots \rangle, \ldots \rangle \text{ and } \mathcal{E}_k = \mathcal{E} \text{ and } \mathcal{E}_j = \mathcal{E}'}{jpElem'(\mathcal{J}, \mathcal{E}, i, j) = \mathcal{E}'}$$

$$\frac{i > 0 \text{ and } \mathcal{E} = \langle \_, \_, \_, \_, \mathcal{E}'' \rangle \text{ and } \mathcal{E}' = jpElem'(\mathcal{J}, \mathcal{E}'', i-1, j)}{jpElem'(\mathcal{J}, \mathcal{E}, i, j) = \mathcal{E}'}$$

Determine at which position $c.a$ is mentioned in the `@advisedBy` clause of $c'.m$

$$advBy(c, a, c', m) \qquad \frac{\begin{array}{c} advBy(c', m) = (c_1, a_1, \ldots, c_n, a_n) \\ \text{where } a = a_i \text{ and } c \leq c_i \text{ and } \nexists c_j : j \neq i \text{ and } c \leq c_j < c_i) \\ \text{otherwise } i = -1 \end{array}}{advBy(c, a, c', m) = i}$$

Retrieve the class at position i in the `@advisedBy` clause of $c.m$

$$advBy(c, m, i) \qquad \frac{advBy(c, m) = (c_1, a_1, \ldots, c_i, a_i, \ldots, c_n, a_n)}{advBy(c, m, i) = c_i}$$

Figure 21. Helper functions

[**effProc$_{\text{pre}}$**] - The `effProcPre(`i`,`j`,`k`,`v_1 ... v_n`)` expression is the runtime equivalent of *effProc$_{pre}$* in Fig. 18 of Sec. 5.5, which defines the effective precondition of a proceed call. Parameters i and j are used to describe which advice contains the proceed call in question, as determined by $jpElem(\mathcal{J}, i, j)$. Parameter k indicates the position of the advice within the `@advisedBy` clause (if any) of its advised join point. The `effProcPre` expression is defined as three cases: The first describes the effective precondition if there is an `@advisedBy` clause, whereas the second describes the effective precondition in the absence of an `@advisedBy` clause. The third case detects invalid advice compositions, as defined in Sec. 6.1. If $jpElem(\mathcal{J}, i, j)$ is not mentioned as the k^{th} element in the `@advisedBy` clause, a runtime error is produced, as it is no longer possible to evaluate the effective precondition in an unambiguous manner.

[**adv$_{\text{pre}}$**] - Finally, the `advPre(`i`,`j`,`k`,`v_1 ... v_n`)` expression is the runtime equivalent of the static *adv$_{pre}$* function defined in Fig. 18 of Sec. 5.5. It retrieves the precondition of an advice body using `iPre` and determines the value of its `proc` keyword. Note that `iPre` of course is the runtime equivalent of the *iPre* function of Fig. 18. Its semantics are not shown here, as the rule is nearly identical to [**pre**].

7.2 Contract enforcement in ContractAJ

After defining the extensions to the ContractAJ language, we can make use of them to present our contract enforcement algorithm. This algorithm has been implemented as a number of judgements, based on the algorithm for the object-oriented ContractJava language of Findler et al. [18]. Applying these judgements will transform a ContractAJ program into a new version of the program where support for contract enforcement is added. What is different from Findler et al. [18] is that, because contract enforcement is a crosscutting concern, we can

$$\begin{array}{ll}
[\textbf{advBy}] & P \vdash \langle \mathsf{E}[\mathtt{advBy(}c, a, c', m\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[i], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } i = \mathit{advBy}(c, a, c', m)
\end{array}$$

$$\begin{array}{ll}
[\textbf{jpStatic}] & P \vdash \langle \mathsf{E}[\mathtt{jpStatic(}i\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[c], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{E} = \mathit{jpElem}(\mathcal{J}, i, 1) \text{ and } \mathcal{E} = \langle c, \_, \_, \_, \_\rangle
\end{array}$$

$$\begin{array}{ll}
[\textbf{jpThis}] & P \vdash \langle \mathsf{E}[\mathtt{jpThis(}i\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{obj}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathcal{E} = \mathit{jpElem}(\mathcal{J}, i, 1) \text{ and } \mathcal{E} = \langle \_, \_, \mathit{obj}, \_, \_\rangle
\end{array}$$

$$\begin{array}{ll}
[\textbf{pre}] & P \vdash \langle \mathsf{E}[\mathtt{pre(}c, x, \mathit{obj}, \mathit{obj}', \mathit{proc}, v_1 \ \ldots \ v_n\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[e[\mathit{obj}/\mathtt{this}, \mathit{obj}'/\mathit{tgt}, \mathit{proc}/\mathtt{proc}, v_1/\mathit{var}_1, \ldots, v_n/\mathit{var}_n], \mathcal{S}, \mathcal{J}\rangle \\
& \ \text{and } \mathtt{class}\ c\mathtt{\{}\ldots\mathtt{@requires}\ e\ \ldots\ t\ x\mathtt{(}\mathit{var}_1, \ldots, \mathit{var}_n\mathtt{)}\ldots\mathtt{\}} \text{ is in } P \\
& \ \text{and } \mathit{tgt} = \mathit{target}(c, x)
\end{array}$$

$$\begin{array}{ll}
[\textbf{effProc}_{\textbf{pre}}] & P \vdash \langle \mathsf{E}[\mathtt{effProcPre(}i\mathtt{,}j\mathtt{,}k\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\mathtt{if(}\mathit{ifCond}[\mathit{obj}/\mathtt{this}, \mathit{obj}'/\mathit{tgt}, v_1/\mathit{var}_1, \ldots, v_n/\mathit{var}_n]\mathtt{)\{} \\
& \qquad \mathtt{advPre(}i\mathtt{,}j\mathtt{,}k\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)} \\
& \quad \mathtt{\}\ else\ \{effProcPre(}i\mathtt{,}j\mathtt{,}k+1\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)\}}, \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathit{jpElem}(\mathcal{J}, i, j) = \langle c, a, \mathit{obj}, \_, \mathcal{E}\rangle \text{ and } \mathcal{E} = \langle c', m, \mathit{obj}', \_, \_\rangle \\
& \text{and } \mathit{advBy}(c, a, c', m) \neq -1 \text{ and } c \leqslant \mathit{advBy}(c', m, k) \\
& \text{and } \langle a, (t_1, \ldots, t_n \to t), (\mathit{var}_1, \ldots, \mathit{var}_n, e)\rangle \in c \\
& \text{and } \mathit{ifCond} = \mathit{ifPcut}(c, a) \text{ and } \mathit{tgt} = \mathit{target}(c, a)
\end{array}$$

$$\begin{array}{ll}
[\textbf{effProc}_{\textbf{pre}}] & P \vdash \langle \mathsf{E}[\mathtt{effProcPre(}i\mathtt{,}j\mathtt{,}k\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\mathtt{pre(}c'\mathtt{,}m'\mathtt{,}\mathit{obj}'\mathtt{,}\mathit{obj}''\mathtt{,}\mathit{proc}\mathtt{,}v_1 \ \ldots \ v_n\mathtt{))}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathit{jpElem}(\mathcal{J}, i, j) = \langle c, m, \mathit{obj}, \_, \mathcal{E}\rangle \text{ and } \mathcal{E} = \langle c', m', \mathit{obj}', \_, \mathcal{E}''\rangle \\
& \text{and } \mathit{advBy}(c, m, c', m') = -1 \\
& \text{and if } \mathit{isMethod}(c', m') \text{ then } (\mathit{proc} = \mathtt{true} \text{ and } \mathit{obj}'' = \mathtt{null}) \\
& \text{else } (\mathit{proc} = \mathtt{effProcPre(}i+1\mathtt{,}1\mathtt{,}\mathit{advBy}(c', m', c'', m'')\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)} \\
& \text{and } \mathcal{E}'' = \big\langle c'', m'', \mathit{obj}'', \_, \_\big\rangle)
\end{array}$$

$$\begin{array}{ll}
[\textbf{effProc}_{\textbf{pre}}] & P \vdash \langle \mathsf{E}[\mathtt{effProcPre(}i\mathtt{,}j\mathtt{,}k\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\mathtt{error("@advisedBy\ mismatch,\ expected\ instance\ of\ }c'\mathtt{")}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathit{jpElem}(\mathcal{J}, i, j) = \langle c, \_, \_, \_, \mathcal{E}\rangle \text{ and } \mathcal{E} = \langle c', m', \_, \_, \_\rangle \\
& \text{and } c'' = \mathit{advBy}(c', m', k) \text{ and } c \nleqslant c''
\end{array}$$

$$\begin{array}{ll}
[\textbf{adv}_{\textbf{pre}}] & P \vdash \langle \mathsf{E}[\mathtt{advPre(}i\mathtt{,}j\mathtt{,}k\mathtt{,}v_1 \ \ldots \ v_n\mathtt{)}], \mathcal{S}, \mathcal{J}\rangle \\
& \hookrightarrow \langle \mathsf{E}[\mathtt{iPre(}c\mathtt{,}a\mathtt{,}\mathit{obj}\mathtt{,}\mathit{obj}'\mathtt{,effProcPre(}i\mathtt{,}j+1\mathtt{,}k+1\mathtt{,}v_1 \ \ldots \ v_n\mathtt{),}v_1 \ \ldots \ v_n\mathtt{))}], \mathcal{S}, \mathcal{J}\rangle \\
& \text{where } \mathit{jpElem}(\mathcal{J}, i, j) = \langle c, a, \mathit{obj}, \_, \mathcal{E}\rangle \text{ and } \mathcal{E} = \langle \_, \_, \mathit{obj}', \_, \_\rangle
\end{array}$$

Figure 22. Semantics of contract enforcement expressions

implement it by adding a number of aspects. The judgements that form the contract enforcement transformation have the following shape:

$$\mathit{context} \vdash \mathit{lhs} \rightharpoonup \mathit{rhs}$$

This can be read as: "Within *context*, an occurrence of *lhs* in the source code produces the code in *rhs*". The most important judgements describing the contract enforcement algorithm are shown in Fig. 23:

[**def**] - This rule specifies the $\rightharpoonup_{\mathsf{def}}$ judgement, which describes the entire transformation at a high level: the transformation is initiated by applying this judgement to every class in the program. As a result, a contract-checking class (`Contract_`c) will be generated for each existing class (c). An example is given in Fig. 24 where the contract checking class `Contract_Security` is generated for class `Security`.

```
          $x_j$ is a method or an advice                    $for\ j \in [1,m]$
          $P,c \vdash x_j \rightharpoonup_{\mathsf{adv}} xcheck_j$   if $\nexists c'' : x_j \in c''$ and $c < c''$
          $P,c \vdash x_j \rightharpoonup_{\mathsf{pre}} xpre_j$   $P,c \vdash x_j \rightharpoonup_{\mathsf{post}} xpost_j$
[def]   ------------------------------------------------------------------------------
        $P \vdash$ declare precedence ...; ... class $c$ extends $c'$ {$x_1 \dots x_m$}...main{...} $\rightharpoonup_{\mathsf{def}}$
        declare precedence ...,Contract_$c$.$xcheck_1$,...,Contract_$c$.$xcheck_m$; ...
        class Contract_$c$ extends Object {$xcheck_1 \dots xcheck_m$}
        class $c$ extends $c'$ {$x_1 \dots x_m$   $xpre_1 \dots xpre_m$   $xpost_1 \dots xpost_m$}
        ...main{Contract_$c$ c_$c$ = new Contract_$c$;...}

        $P,c \vdash t\ m(t_1\ x_1, \dots, t_n\ x_n)) \rightharpoonup_{\mathsf{adv}}$
        around $m$: call($t\ c.m$($t_1\ x_1$,...,$t_n\ x_n$)) && target($dyn$) {
          if(pre(jpStatic(0),$m$,$dyn$,null,true,$x_1 \dots x_n$)) {
            $dyn.m$_SbsPreCheck($x_1 \dots x_n$)
          } else {error("Precondition violation: getStackTrace[1]")}
[adv^m]   let {$returnVal$=proceed($dyn$,$x_1 \dots x_n$)} in {
            if(post(jpStatic(0),$m$,$returnVal$,$dyn$,null,true,$x_1 \dots x_n$)) {
              $dyn.m$_SbsPostCheck($returnVal$,$x_1 \dots x_n$)
            } else {error("Postcondition violation:" $dyn$)};
            $returnVal$}}

        $P,c \vdash$ around $a$: call|execution($t\ c'.x$($t_1\ x_1 \dots t_n\ x_n$) $\rightharpoonup_{\mathsf{adv}}$
        around $a$: execution($t\ c.a$($t_1\ x_1 \dots t_n\ x_n$)) && this($dyn$) {
          let {$i$ = advBy(jpStatic(0),$a$,jpStatic(1),$x$)} in {
          if($i$ ==-1) { // If the user-advice is not mentioned in @advisedBy
            let {$jpPre$ = effProcPre(0,1,-1,$x_1 \dots x_n$)
              $advPre$ = pre(jpStatic(0),$a$,$dyn$,jpThis(1),$jpPre$ $x_1 \dots x_n$)} in {
              if($jpPre$) {
                if(!$advPre$) {error("ASP violation, precondition too strong:" $dyn$)}
              } else {error ("Precondition violation: getStackTrace[1]")}}
            let {$returnVal$=proceed($dyn$,$x_1 \dots x_n$)
              $jpPost$ = effProcPost(0,1,-1,$x_1 \dots x_n$)
              $advPost$ = post(jpStatic(0),$a$,$returnVal$,$dyn$,jpThis(1),$jpPost$,$x_1 \dots x_n$)} in {
              if($jpPost$) {
[adv^ar]        if(!$advPost$) {error("ASP violation, postcondition too weak:" $dyn$)}
              } else {error("Postcondition violation:" $dyn$)};
              $returnVal$}
          } else { // If the user-advice is mentioned in @advisedBy
            let {$proc$ = effProcPre(0,2,$i+1$,$x_1 \dots x_n$)
              $pre$ = pre(jpStatic(0),$a$,$dyn$,jpThis(1),$proc$,$x_1 \dots x_n$)} in {
              if($pre$) {$dyn.a$_SbsPreCheck(jpThis(1),$proc$,$x_1 \dots x_n$)
              } else {error("Precondition violation: getStackTrace[1]")}}
            let {$returnVal$=proceed($dyn$,$x_1 \dots x_n$)
              $proc$ = effProcPost(0,2,$i+1$,$x_1 \dots x_n$)
              $post$ = post(jpStatic(0),$a$ $dyn$,jpThis(1),$proc$,$x_1 \dots x_n$)} in {
              if($post$) {$dyn.a$_SbsPostCheck(jpThis(1),$proc$,$returnVal$,$x_1 \dots x_n$)
              } else {error("Postcondition violation:" $dyn$)};
              $returnVal$}}}

        $P,c \vdash t\ m(t_1\ x_1, \dots, t_n\ x_n)\ \{e\} \rightharpoonup_{\mathsf{pre}}$
        boolean $m$_SbsPreCheck($t_1\ x_1$, ..., $t_n\ x_n$) {
        let{$next$=($\exists^?$super.$m$_SbsPreCheck($x_1 \dots x_n$)
[pre^m]   $res$=pre($c$,$m$,this,null,true,$x_1 \dots x_n$)} in {
          if (!$next$ || $res$) {
            $res$
          } else {error("SBS' violation, precondition too strong:" this)}}
```

Figure 23. Contract enforcement judgements

```
declare precedence Security.authenticate, contract_Security.authenticate;

class Bank {
  @requires u.getBank()==this
  @ensures result!=null && result.getOwner()==u
  @advisedBy Security.authenticate
  Account createAccount(User u) {...}}

class Security {
  @requires proc
  @ensures if(isLoggedIn(u)){proc}else{true}
  around authenticate: call(Account Bank.createAccount(User u)) && target(b) {...}
  boolean authenticate_SbsPreCheck(Bank b,boolean proc,User u) { ... }
  boolean authenticate_SbsPostCheck(Bank b,boolean proc,Account returnVal,User u) {...}}

class Contract_Security {
  around authenticate: execution(Account Security.authenticate(User u)) && this(dyn) {
    let{i=advBy(Security,authenticate,Bank,createAccount)} in {
    if(i==-1) { ...
    } else {
      let{proc="u.getBank()==this"
        pre="u.getBank()==this"} in {
        if(pre) {dyn.authenticate_SbsPreCheck(jpThis(1),proc,u)
        } else {error("Precondition violation: getStackTrace[1]")}}
      let{returnVal=proceed(dyn,u)
        proc="result!=null && result.getOwner()==u"
        post="if(isLoggedIn(u)){result!=null && result.getOwner()==u}else{true}"} in {
        if(post) {dyn.authenticate_SbsPostCheck(jpThis(1),proc,returnVal,u)
        } else {error("Postcondition violation:" dyn)};
        returnVal}}}

main {
  Contract_Security c_security = new Contract_Security; ...}
```

Figure 24. Example of a contract enforcement aspect

First, the `declare precedence` statement is extended to ensure that every advice in `Contract_`c has a lower precedence than all other advice. This is necessary to make sure that the existing advice will not interfere with contract enforcement advice at shared join points. Next, we can examine the definition of `Contract_`c itself: For each method/advice x_j in c, an advice $xcheck_j$ is added to `Contract_`c. This around advice is executed whenever x_j is called and will then perform contract enforcement. To avoid confusion, henceforth we will call the contract enforcement advice "contract-advice". Normal advice defined in the original program will be called "user-advice". Note that the $xcheck_j$ contract-advice are only created for non-overriding methods/advice. It would be redundant to add them to overriding members due to the use of a call pointcut, which also matches on subtypes. The code of each contract-advice is produced by the $\rightharpoonup_{\mathsf{adv}}$ judgement, defined in the [**adv**] rules (only [$\mathbf{adv}^{\mathrm{m}}$] and [$\mathbf{adv}^{\mathrm{ar}}$] are shown). Next, $xpre_j$ and $xpost_j$ are two helper methods that are added to the existing c class; they check whether the SBS' pre- and postcondition rules hold. These two methods are specified by the [**pre**] and [**post**] rules (only [$\mathbf{pre}^{\mathbf{m}}$] is shown). Finally, the program's main expression is extended such that `Contract_`c is instantiated.

[$\mathbf{adv}^{\mathbf{m}}$] - This rule specifies the contract-advice that enforces contracts of methods. An around contract-advice is used for this purpose, associated with a `call` pointcut that matches whenever $c.m$ is called, where c is (a subtype of) the static type of the call.

In the contract-advice's body, the precondition of m is first checked, in class `jpStatic(0)`, which corresponds to the receiver's static type in the $c.m$ call. If this precondition check fails, we state that "`getStackTrace[1]`" is to be blamed. In most cases, this simply means that the caller of the method is to be blamed. However, if one or more user-advice are present at this method, the previous user-advice in the composition is to be blamed. We can only blame this previous advice; otherwise an error would have been generated earlier by the contract-advice that check each user-advice. After `pre(jpStatic(0),`m`,...)` is checked, we invoke $dyn.m$`_SbsPreCheck`, the SBS'-checking helper method, which is specified in rule [$\mathbf{pre^m}$]. After this SBS'-check has passed for the preconditions, we can make a proceed call to execute the method that we are checking. Once this is done, the postconditions are checked, which is analogous to checking the preconditions.

[$\mathbf{adv^{ar}}$] - This lengthy rule specifies the contract-advice that checks all around user-advice. The reason for its length is the fact that it handles both ASP-compliant advice and advice mentioned in an `@advisedBy` clause. First, we try to determine the position (i) of the user-advice within the `@advisedBy` clause of its advised join point using `advBy`. If it returns `-1`, this indicates the user-advice is not mentioned in the `@advisedBy` clause (if there was one), which implies the user-advice is ASP-compliant:

ASP-compliant advice - We first determine the precondition of the advised join point in *jpPre*. This precondition corresponds to the effective precondition of the user-advice's proceed call, which is why `effProcPre(0,1,...)` is used. The precondition of the user-advice itself is stored in *advPre* (with *jpPre* as the value of the `proc` keyword). Next, *jpPre* is checked: Similar to [$\mathbf{adv^m}$], "`getStackTrace[1]`" is to be blamed if the test fails. If the test passes, *advPre* is tested next: If checking the user-advice's precondition fails, we know it is too strong and the ASP is broken. If this check passes, we can proceed with executing the user-advice, do the analogous checks for postconditions and finally return the user-advice's return value.

Advice mentioned in an `@advisedBy` clause - In case the user-advice is mentioned in an `@advisedBy` clause, we will first determine the value of the `proc` keyword using `effProcPre(0,2,`$i+1$`,...)`. The *pre* variable then contains the user advice's precondition. The contract enforcement procedure itself is quite similar to [$\mathbf{adv^m}$]: The precondition is first tested. If this test passes, $dyn.a$`_SbsPreCheck` should be tested, as the elements in an `@advisedBy` clause can also match with subtypes. Once this test passes, the proceed call is made and the analogous postcondition tests are done. Fig. 24 presents an example of a contract-advice, such that the contracts that will be checked are visible. These concrete contracts are only shown to improve the example's clarity. The actual source code of `Contract_Security.authenticate` would show `pre`, `effProcPre`, `post` and `effProcPost` expressions instead, as this contract-advice should also be able to enforce the contracts of any user-advice that override `Security.authenticate`.

[$\mathbf{pre^m}$] - This rule checks that the precondition of a particular method is not stronger than any of its ancestors. This is done by recursively traversing up the subtype hierarchy tree with a `super.m_SbsPreCheck` call. (The $\exists^?$ symbol before `super.`m`_SbsPreCheck` means: if m`_SbsPreCheck` does not exist in `super`, the call is left out.) There is also a similar [**pre**] variant of this rule (not shown) for advice: The only difference is that this helper method contains two extra parameters: the value of the `target` binding and the value of the `proc` keyword.

To finish up the contract enforcement algorithm, we should still discuss the [**adv**] rules that are not shown in Fig. 23. The [$\mathbf{adv}^{ar}$] rule that was discussed here is focused on around user-advice. We should also specify separate variants for before and after user-advice, which are mostly similar to [$\mathbf{adv}^{ar}$]. In case of before advice, the postcondition checks are replaced by only checking the postcondition of the advice itself, i.e. without testing the ASP or SBS'. If this check fails, the advice itself is to blame. If the before advice interferes with the next element in the composition, this will be detected by that element's precondition check, which will correctly blame `getStackTrace[1]`. The after advice is treated similarly, precondition checks are replaced by only checking the precondition of the advice itself.

7.3 AspectJ implementation

To demonstrate an instantiation of our contract enforcement algorithm in a full programming language, we have implemented it as a lightweight design by contract library for AspectJ, which is available for download[9]. The library itself is written completely using the constructs provided by AspectJ itself. As the library consists of aspects, and aspects are implicitly instantiated in AspectJ, contract enforcement is automatically enabled as soon as the library is included on a project's build path. Contracts are specified as strings in Java annotations (`@ensures`, `@requires`, `@invariant`); they are evaluated at runtime by a scripting engine. To keep the library small and simple, we make the assumption that the developer will not produce any side effects in the program's contracts, rather than build a fleshed-out behavioural interface specification language like JML. Contracts have access to the necessary information to implement basic design by contract support in AspectJ: the `this` object, parameters, the return value (in case of postconditions) and the `old()` function to evaluate expressions in the pre-state of a method/advice execution and retrieve the result in the post-state.

Regarding the implementation of the enforcement algorithm, most of the algorithm can be translated fairly directly from ContractAJ to AspectJ. Rather than creating a contract enforcement aspect per class/aspect, the implementation only has two contract enforcement aspects that handle all method/advice executions. The extensions made in Sec. 7.1 can be reproduced in AspectJ using reflection and the `thisJoinPoint` variable. The AspectJ counterpart of the AP (the fifth element in each $\mathcal{E}$ tuple) is also available in higher-order advice as the

[9] The library and source code are available at: https://github.com/timmolderez/adbc

last argument of `thisJoinPoint.getArgs()`, which points to the `thisJoinPoint` variable[10] of the advised join point.

One subtle difference between the algorithm and the implementation is concerned with making sure that a contract-advice always is the very last advice to be executed in an advice composition, which requires using an execution pointcut. This is due to the fact that, in AspectJ, an advice matching at a call join point is always executed before an advice that matches at the corresponding execution join point. This is not the case for ContractAJ, due to its simpler join point model discussed in Sec. 2.3. Using an execution pointcut creates a new problem, as AspectJ only provides access to the dynamic type of the advised method call. We also need to know the static type to be able to determine which contracts need to be enforced. Luckily, we can work around this problem by introducing a helper advice that matches on any call join point, and temporarily stores the static type in a stack, until contract enforcement is performed at the corresponding execution join point.

8 Related work

Observers and assistants - There are several related papers in the field of modular reasoning for aspects. Most closely related is the work on observers and assistants by Clifton and Leavens [11,13,12]. Their work focuses on modular reasoning in AspectJ, using JML as the specification language. On the surface, our work looks quite similar to theirs, as our distinction between ASP-advice and non-ASP advice is akin to the distinction between observers and assistants. As mentioned in Sec. 4, the main difference between ASP-compliant advice and observers is that the ASP is defined in terms of the advice's specification rather than its implementation. This allows the ASP to be more permissive than the notion of observers: An observer can only modify the state it owns, as well as global state. In Clifton's thesis [13], observers (also called spectators) are further restricted since around advice must make exactly one proceed call. ASP-advice is not subjected to these restrictions: In addition to the state it owns and global state, ASP-advice may also modify any state that is being modified by the join points it advises, as long as the specifications of its advised join points are taken into account. Likewise, ASP-compliant advice may make as many proceed calls (including zero) as desired. To give a few examples that distinguish ASP-compliant advice from observers: Consider a caching aspect for e.g. a database. Whenever a query is made, an advice would test the cache. This advice would be ASP-compliant, as it does not alter the database's behaviour. It is however not an observer, since the advice does not make a proceed call on a cache hit. Another kind of example, closely related to context-oriented programming [20], would be an advice that acts like an overriding method (i.e. a body of code that extends the behaviour of an existing method), but one that is only active within

[10] One caveat is that an advice body should make use of its `thisJoinPoint` variable in order for it to be available for higher-order advice.

a certain context, e.g. a particular control flow. Such an advice should be ASP-compliant, just like overriding methods should be SBS-compliant. However, if this advice modifies a field in the receiver of the advised method calls, it cannot be an observer, as this field is not owned by the corresponding aspect.

Moreover, ASP-compliant advice are guaranteed to preserve modular reasoning, on the assumption that each module correctly implements its own specifications. Ensuring that an advice is ASP-compliant can be done statically by the developer, given the advice's specifications and those of its join point shadows. On the other hand, it is undecidable in general to statically ensure that an observer will only modify its own/global state and will make one proceed call in any control flow path. Granted, in case of ASP-compliant advice, the undecidability problems have been delegated to the assumption that each module correctly implements its own specifications. Nonetheless, this problem is well-known and the vast amount of work that strives to approximate a solution can be readily leveraged.

Clifton and Leavens' notion of assistants is closely related to the use of `@advisedBy` clauses. Restoring modular reasoning for assistants is done by explicitly mentioning these aspects in an `accepts` statement in a module, to indicate that any join points in this module may be advised by the mentioned advice. Using an `@advisedBy` clause has a similar purpose, but it is more fine-grained: An `@advisedBy` clause is associated with a method and mentions a number of advice, which makes it straightforward to tell which advice are expected at a particular method call. Constructing the effective specification of an advice is also done very differently: In case of assistants, a graph is constructed that contains the possible advice compositions per join point shadow. The specification of each path in the graph is then constructed by composing the specification of each node on the path and eliminating all intermediate states, which can be a demanding process. It also does not prevent interference between the advice, as before/after advice are not constrained such that they take into account their implicit proceed call. In case of non-ASP advice, the effective specification is arguably much simpler to construct, the IPR of Sec. 5.6 prevents interference and the specification leaves room for extension using overriding advice or higher-order advice. Finally, Clifton and Leavens' work also approaches around advice quite differently: There is no equivalent of a `proc` keyword. Instead, the specification of an around advice is split up into a before and an after part. The before part refers to all code up until a proceed call is made, and the after part refers to the code after a proceed call is made. This becomes more complex/verbose once proceed calls appear in control statements (e.g. in an authentication or authorization advice). Multiple specification cases are needed, as there now are multiple possible before-and after parts. While the `proc` keyword does not expose when/whether a proceed call will be made, it also is simpler to use.

Translucid contracts - The work on translucid contracts for the Ptolemy language [6] is closely related as well. The Ptolemy language requires all advisable join points to be announced explicitly as events, including which information is exposed to event handlers (similar to advice). Each event is associated with its own specifications called translucid contracts. The modules that announce events are aware of these contracts, and event handlers have to comply with the contracts in order not to cause any surprising effects. While this approach is flexible in the sense that the event announcer can be oblivious of which, how many or in what order event handlers are present, this flexibility arguably also makes it more difficult to design the specifications of events such that they are sufficiently restrictive for the event announcer, but sufficiently permissive for any event handlers that might register to the event. An `@advisedBy` clause is more restrictive, as it announces which advice are expected, in what order, but it leaves open which (sub)types should implement the listed advice. Another difference between translucid contracts and our approach is that translucid contracts are grey-box specifications, as they also expose how event handlers should alter control flow, i.e. when a proceed call is made or not. Our contracts in ContractAJ are black-box, yet the use of the `proc` keyword often gives away on what conditions a proceed call is made. Nonetheless, the `proc` keyword is nothing but a placeholder for another module's pre- or postcondition, and is not a guarantee that an advice body will (or will not) make a proceed call on certain conditions.

Pipa and CONA - In Zhao et al. [43], the design of Pipa is presented, a language to specify contracts for AspectJ programs, as an extension to the JML specification language. The interaction of an advice with the base system is viewed from a weaving perspective, at a syntactical level: the contracts of an advice are woven into the contracts of the corresponding advised methods. There is no notion of an advice substitution principle however, as no constraints are placed on how an advice's contracts should relate to those of advised methods. In Lorenz et al. [28], aspects are classified as agnostic, obedient or rebellious. Each of the three types correspond to aspects with only ASP-compliant advice. To enforce that an aspect is of a particular type, blame assignment tables are presented for each type. Developers can indicate which of the three types an aspect belongs to, and a prototype implementation called CONA will then perform contract enforcement using these blame assignment tables. The CONA tool uses aspects to enforce contracts on objects, but uses objects to enforce contracts on aspects. The work of Agostinho et al. [1] as well is based on the advice substitution principle in Wampler [41] and informally discusses its application to various concrete aspect-oriented languages (AspectJ, CaesarJ and FuseJ).

Modular reasoning in AOP-like languages - Related work can also be found for other types of languages similar to AOP: the work of Thüm [40] discusses design by contract for feature-oriented programming. Several different approaches to integrate design by contract are discussed and compared in a number of use cases. The "explicit contract refinement" approach is closest to the ASP, as it is based on method refinement and has the `original` keyword, which is similar to `proc`. The different approaches are compared on a number of case studies, and some cases were identified where none of the approaches were sufficiently expressive, which could indicate the need for a construct similar to the `@advisedBy` clause.

In Hähnle et. al [36], design by contract is discussed for delta-oriented programming, which is as well closely related to AOP. Liskov substitution [27] is adapted to this type of languages. It also demonstrates that making a language's constructs more dynamic also makes it more difficult to reason about: a dynamic element is present, in the sense that the feature configuration of delta-oriented programs can take on various different forms. As a consequence, method contracts will need to take into account more than just the superclass. This is similar to the fact that our `proc` keyword depends on the advised join point.

Finally, there are also several related papers that focus on restoring modular reasoning by establishing new kinds of interfaces between aspects and the modules they advise, such as open modules [2], crosscutting program interfaces (XPIs) [39,35], join point types [38] or join point interfaces [7]. While some of these interfaces restrict what each aspect is allowed to advise, this work is mostly complementary to ours, in the sense that such intermediate interfaces can alleviate the fragile pointcut problem and reduce the number of reasoning tasks for developers of aspects.

9 Conclusion

In this paper an approach has been presented to achieve modular reasoning in aspect-oriented languages, using the ContractAJ language. This approach is centred around an advice substitution principle and the `@advisedBy` clause. For those advice that satisfy the ASP, obliviousness can be preserved without affecting modular reasoning. For all other advice, the `@advisedBy` clause should be used to become aware of such advice. Apart from the fact that each advice should take into account the contracts of any (implicit) proceed calls, there are no restrictions on what an advice is allowed to do. The approach is shown to preserve modular reasoning when making method/proceed calls and an algorithm is provided to perform runtime contract enforcement.

In terms of future work, case studies can be performed to study our approach on existing AspectJ applications and to answer questions such as: What proportion of all advice is ASP-compliant versus non-ASP-compliant? If an advice advises a large amount of join point shadows, can there be a lot of coupling between the advice and its shadows? In other words, does the developer always need to pay close attention to the value of the `proc` keyword or is it unlikely to cause conflicts with our approach?

Another interesting path of future work is to provide support for static contract enforcement. That is, the ability to determine statically whether an advice satisfies the ASP or not. If not, we should examine whether the advice is mentioned in the relevant `@advisedBy` clauses. Finally, another path worth exploring is to study invariants and frame conditions in more detail, and to extend the approach with e.g. an ownership type system such that it can be used for formal verification.

References

1. Sérgio Agostinho, Ana Moreira, and Pedro Guerreiro. Contracts for aspect-oriented design. In *Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies (SPLAT)*, page 1:1–1:6, New York, NY, USA, 2008. ACM.
2. Jonathan Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2005.
3. Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in Lecture Notes in Computer Science, pages 60–90. Springer Berlin Heidelberg, January 1991.
4. S. Apel and D. Batory. How AspectJ is used: an analysis of eleven AspectJ programs. *Journal of Object Technology (JOT)*, 9(1):117–142, 2010.
5. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, page 135–173. 2006.
6. Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, page 141–152, New York, NY, USA, 2011. ACM.
7. Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 23(1):7:1–7:41, February 2014.
8. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.
9. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, October 1998.
10. Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-oriented programming with AJDT. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.
11. Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proceedings of the 1st workshop on Foundations of aspect-oriented languages*, FOAL '02, page 33, 2002.
12. Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2003.
13. Curtis Clifton and Gary T. Leavens. *A design discipline and language features for modular reasoning in aspect-oriented programs.* PhD thesis, Iowa State University, 2005.
14. Curtis Clifton, Gary T. Leavens, and James Noble. MAO: ownership and effects for more effective reasoning about aspects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, number 4609 in Lecture Notes in Computer Science, pages 451–475. Springer Berlin Heidelberg, January 2007.
15. Daniel S. Dantas and David Walker. Harmless advice. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 383–396, New York, NY, USA, 2006. ACM.
16. Krishna K. Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, 1996*, pages 258–267, 1996.

17. Robert E Filman and Daniel P Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced separation of Concerns, OOPSLA*, 2000.
18. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, page 1–15, New York, NY, USA, 2001. ACM.
19. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, page 171–183, New York, New York, USA, January 1998. ACM Press.
20. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.
21. Shmuel Katz. Aspect categories and classes of temporal properties. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, number 3880 in Lecture Notes in Computer Science, pages 106–134. Springer Berlin Heidelberg, January 2006.
22. Christian Koppen and Maximilian Störzer. PCDiff: attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, 2004.
23. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.
24. Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. 2006.
25. K. Rustan M. Leino. Data groups: specifying the modification of extended state. *SIGPLAN Not.*, 33(10):144–153, October 1998.
26. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 491–515. Springer Berlin Heidelberg, January 2004.
27. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.
28. David H Lorenz and Therapon Skotiniotis. Extending design by contract for aspect-oriented programming. *http://arxiv.org/abs/cs/0501070*, January 2005.
29. Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.
30. Bertrand Meyer. Applying "Design by contract". *Computer*, 25(10):40 –51, October 1992.
31. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006.
32. David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Computer Science*, 365(1–2):143–168, November 2006.
33. Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179. Springer Berlin Heidelberg, 2008.
34. Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *27th International Conference on Software Engineering (ICSE)*, pages 59 – 68, 2005.

35. Henrique Rebelo, Gary T. Leavens, Ricardo Massa Ferreira Lima, Paulo Borba, and Márcio Ribeiro. Modular aspect-oriented design rule enforcement with XPIDRs. In *Proceedings of the 12th workshop on Foundations of aspect-oriented languages*, FOAL '13, page 13–18, New York, NY, USA, 2013. ACM.
36. Reiner Hähnle and Ina Schaefer. A liskov principle for delta-oriented programming. *Formal Verification of Object-oriented Software*, 2011.
37. Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, page 147–158, New York, NY, USA, 2004. ACM.
38. Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1):1:1–1:43, July 2010.
39. Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with XPIs. *ACM Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42, September 2010.
40. Thomas Thüm, Ina Schaefer, Martin Kuhlemann, Sven Apel, and Gunter Saake. Applying design by contract to feature-oriented programming. In *15th International Conference on Fundamental Approaches to Software Engineering*, 2012.
41. Dean Wampler. Aspect-oriented design principles: Lessons from object-oriented design. In *Sixth International Conference on Aspect-Oriented Software Development*, http://aosd.net/2007/program/industry/I6-AspectDesignPrinciples.pdf, 2007.
42. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.
43. Jianjun Zhao and Martin Rinard. Pipa: A behavioral interface specification language for AspectJ. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.