Reasoning about Aspects With Common Sense

Klaus Ostermann
University of Aarhus, Denmark

Abstract

There has been a lot of debate about the modularity of aspect-oriented programs, and in particular the ability to reason about such programs in a modular way, although it has never been defined precisely what modular reasoning means. This work analyzes what it means to reason about a program, and separates "modular reasoning" into several well-defined properties of a reasoning model.

A comparison of an OO language semantics with an AO language semantics with respect to these properties reveals that explanations of AOP that are based on weaving are a major obstacle to reasoning about AO programs in a modular way. We argue that a more modular semantics that is easier to reason about can be given to AO programs if we renounce the monotonicity of the corresponding reasoning system - a sacrifice that is well-known in artificial intelligence to model "common sense" reasoning. More generally, we claim that AOP should be understood as a form of nonmonotonic knowledge representation.

1. Introduction

Many software engineers have noted that programs should be a direct image of our thought process about the problem domain, with as little representational overhead as possible. For example, Wegner talks about a direct correspondence between logical and physical hierarchies [35], Winkler demands a close relationship between the 'concept-oriented' and the 'program-oriented' view [36], Meyer postulates the 'direct mapping principle' [25], and Larman argues in favor of a 'low representational gap' [19]. This idea hence connects modularity and knowledge representation: The organization of a program should be similar to how humans structure their knowledge about the domain.

On the other hand, software developers want to reason about a program part without knowledge of the complete system. They want to "study the system, one module at a time" [29]. It usually goes without saying that the reasoning model is monotonic, which means that conclusions are never invalidated when we learn more about the system.

However, these two goals do not necessarily go hand in hand. The most "natural" decomposition and organization of a software system (from the perspective of a human) may not be one where many interesting properties can be derived in a modular way using, say, classical first-order logic. One of the fundamental insights in the AI community is that humans organize knowledge and reason in a nonmonotonic way, meaning that humans adapt their thought process when they learn something new; in particular, new knowledge may invalidate previous conclusions. A typical example is that we know birds usually fly, and that Tweety is a bird, and hence conclude that Tweety flies - until we learn that Tweety is actually a penguin. In classical (monotonic) logic, adding a piece of information to a knowledge base never reduces the set of its consequences. Intuitively, monotonicity indicates that learning a new piece of knowledge cannot contradict what was previously known, hence the "default" rule about birds cannot be expressed adequately in classical logic. Nonmonotonic logics (the formal incarnations of nonmonotonic reasoning) have been developed to deal with incomplete and changing information. Nonmonotonic logic allows to revise conclusions if new knowledge arrives, and provides rigorous mechanisms for taking back conclusions that no longer fit to newly learned knowledge, and deriving new, alternative conclusions instead [3, 20, 17].

We make three claims that permeate the whole paper: First, the logic with which we reason about a program must reflect and fit to the way we think about the program when designing its modular structure. Second, there is a deep conflict between direct mapping and monotonicity, and both the power and the problems of AOP can be explained as a new trade-off between these two goals that favors direct mapping at the expense of monotonicity. Third, nonmonotonicity is unavoidable and invariably also present in classical modularity mechanisms, hence we should strive for modularity mechanisms that embrace (rather than ignore) nonmonotonicity.

More specifically, this work makes the following contributions:

- We dissect what it means to reason about a program in a modular way, and propose a set of precise criteria to characterize modular reasoning.
- We analyse an object-oriented and an aspect-oriented language and compare and characterize the classes of properties that can be established in a modular way using classical reasoning in terms of their axiomatic semantics. In particular, we show that "weaving" basically makes all non-trivial equations depend on the global program, and conclude that weaving is an inappropriate model to explain aspects.
- We argue that the structure of a knowledge base in nonmonotonic logic is similar to the modularity allowed by (pointcut+advice) aspect-oriented programming, and claim that such logics are more appropriate to reason about AOP programs than classical predicate logics. We propose a new explanation of aspects based on a variant of nonmonotonic logic called *default logic*. We show that this semantics restores the ability to do local (but nonmonotonic) reasoning, and furthermore allows new classes of properties to be established modularly that could only be established using global, nonmonotonic reasoning in the object-oriented semantics. However, this reasoning approach should not be understood as the "solution" to the modular rea-

soning question in AOP, but rather as a new point of view on the problem.

- We expand on the relation between aspects and nonmonotonic logic and show that the fundamental problem of aspect interaction and, more specifically, of aspect precedence is quite related to priority mechanisms in nonmonotonic logics. To validate this claim, we study how precedence mechanisms for interacting pointcuts can be modeled and improved by using priority mechanisms from the field of nonmonotonic reasoning.
- We sketch how a form of rely-guarantee reasoning [37] can easily be added to the reasoning approach based on default logic.

The rest of the paper is structured as follows. In the next section, we analyze what modular reasoning means and compare an object-oriented and a weaving-based aspect-oriented axiomatic semantics with respect to these definitions. In Sec. 3 we give a short introduction to default logic. Sec. 4 describes how AOP languages can be explained in terms of default logic, and analyzes the ramifications for modular reasoning. Sec. 5 illustrates how aspect precedence can be modeled using prioritized default logic. Sec. 6 revisits the question of modular reasoning in the light of the new explanation of aspects based on default logic. Sec. 7 discusses what has been achieved and elaborates on related work. Sec. 8 concludes.

2. Reasoning about programs

This section investigates the question of what it means to reason about a program in a modular way from the perspective of formal logic. We assume that we have a program in a programming language with modules. To make things simple, we assume a flat name and module space. To make the role of modules and their dependencies explicit, we model a program as a finite mapping from a set of (module) names to modules: $P = N \rightarrow M$. The restriction of a program to a subset of its modules $N' \subseteq dom(P)$ is denoted by $P|_{N'}$, and the union of two programs is denoted by $P \cup P'$; both are defined in the obvious manner (the union is only defined if the defined names do not overlap). We also use the notation $P \subseteq P'$ to signify that P' contains P, i.e., $P(l) = m \Rightarrow P'(l) = m$.

From the perspective of logic, reasoning is the activity of applying rules of inference to a knowledge base, such as applying the *modus ponens* inference rule $\frac{P \Rightarrow Q \quad P}{Q}$ to the knowledge base $\{\texttt{sunny}, \texttt{sunny} \Rightarrow \texttt{happy}\}$ to conclude happy. We do not make any assumptions on the kind of logic; for our purposes it is sufficient to assume a logic L that is given by a set of formulae (also denoted by L) and an inference relation $\vdash_L \subseteq \mathcal{P}(L) \times L$. In the following, we call the set of formulae to the left of the turnstile the *knowledge base*.

The connection between a program and a logic is through a *representation function*. A *representation* of a programming language P in a logic L is a function $rep_L : P \rightarrow \mathcal{P}(L)$ that maps programs in the language to sets of formulae in the logic. Of course, the representation should be sound in some sense with respect to the actual meaning of the program (or it can be the *definition* of the language semantics), but this question is irrelevant for the discussion. A *program logic* for a programming language P is a logic L together with a representation of P in L.

Another option would be to make the logic depend on the program, i.e., make the inference relation depend on the program. For example, a Hoare logic [12] or a structural operational semantics [30] are usually defined via a program-dependent inference relation. However, it is always possible to "lift" such a program-dependent logic into a program-independent logic such as predicate logic by basically replacing the inference bar with an implication sign and universally quantifying over all meta-variables of the inference rule. We will later see examples for that.

Now we are ready to define criteria that are relevant to modular reasoning. We believe it would be arbitrary to select some subset of these criteria as *the* definition of 'modular reasoning', and we leave an assessment of their relative importance as future work.

One of the most basic requirements for modular reasoning is the compositionality of the representation rep_L. A representation is *compositional*, if $rep_L(P \cup P') = rep_L(P) \cup rep_L(P')$. Without a compositional representation it does not make sense to reason about a subpart of the program without knowing the full program, because the representation of the full program could be completely different from that of the subprogram.

A standard notion in logic is the monotonicity of the inference relation, which basically says that enlarging the knowledge base must not invalidate previous conclusions, that is, if $A \vdash F$ and $A \subseteq A'$, then $A' \vdash F$. Most standard logics such as propositional or predicate logic are monotonic. We will later discuss a nonmonotonic logic called default logic.

We say that a formula F in a program logic is *monotonic*, if whenever $rep_L(P) \vdash_L F$ and $P \subseteq P'$, then $rep_L(P') \vdash_L F$. It is easy to see that all formulae in a program logic are monotonic, if the logic is monotonic and if the representation is compositional. Without compositionality, however, a formula could be nonmonotonic although the logic is monotonic. Monotonicity represents the idea of establishing a property "once and for all". A slightly weaker notion is monotonicity w.r.t. certain kinds of program extensions described by some subset R of all possible program extensions. We say that a formula F in a program logic is *monotonic w.r.t. R*, if whenever $rep_L(P) \vdash_L F$ and $P \subseteq P'$ and $P' \setminus P \subseteq R$, then $rep_L(P') \vdash_L F$. For example, a property could be monotonic w.r.t. addition of new classes, but not monotonic w.r.t. addition of new advice.

A formula F in a program logic is *local* in a program part $P' \subseteq P$, if $rep_L(P) \vdash_L F$ if and only if $rep_L(P') \vdash_L F$. This definition captures the idea that if one wants to prove some property of a single module (or set of modules), then one only needs to look at this module. Usually, all properties that can be deduced from the interface of modules are local in this module. For example, in a Java-like OO language it is sufficient to look at a method signature to see whether a call to this method will return a boolean value.

A formula F in a program logic is *traceable* in $P' \subseteq P$, if $rep_L(P) \vdash_L F$ if and only if $rep_L(P|_{names^*(P')}) \vdash_L F$, where $names^*(P')$ is the set of transitively occuring module names in P', that is, the names of the modules in P', all module names that occur in modules referenced by P' and so forth. Properties that can be established by an investigation of the implementation of transitively referenced modules are traceable. For instance, a proof that a procedure call will terminate in a procedural language usually requires an investigation of the procedure body and transitively all procedures called within the body.

Our last criterion is the program indepence of a formula. We say a formula F in a program logic is *program-independent*, if F does not contain any global information about the program. That is, if the program is extended, the formula must not be changed: It is still a formula about the full program. For example, if F would contain a copy of the full program, or some abstraction thereof, such as a global list of advice in an AOP language, it would not be program-independent. This criterion is important because it would be trivial to design a program logic that fulfills all criteria mentioned above by encoding an interpreter of the language in the logic, where one of the parameters of the interpreter is the program, and all questions about the program can only be answered by "passing" the full program in the formula. This criterion could be made more formal by formalizing a "is a formula about" relation between formulae and programs, but for the sake of brevity we leave it at that. In the remainder of the paper, we will always implicitly assume that the representation function generates program-independent formulae (i.e., no cheating).

Significance of a compositional representation

From this list of criteria, we believe that the existence of a compositional representation of programs in a logic is in some sense the most fundamental one. Without a compositional representation, *any* kind of non-global reasoning is more or less futile, because there is no manageable relation between the knowledge base of a program part and the knowledge base of the full program.

A compositional representation can also be seen as a kind of separate compilation, and as such it is an indication that the underlying module system defines useful, self-contained abstractions, and not just random parts of the program. This point of view is also emphasized by Luca Cardelli in his landmark paper on modularization [7]. He asks: "When does a module system *really* support modularization (meant as separate compilation)?" and later concludes: "We consider modularization as inseparable from separate compilation: Not merely as a program structuring mechanism."

A compositional representation is a *morphism* that preserves the structure of the program, and hence we believe that a definition of a language in terms of a compositional representation reveals deep symmetries between the language and the logic framework. We come back to this point in Sec. 7.

2.1 An OO program logic

In this section we show how a program logic for a simple OO language, based on its axiomatic semantics, looks like in terms of our formal framework. We will consider a fragment of Featherweight Java (FJ) [13]. FJ is defined via a standard small-step reduction semantics. To emphasize its role as an equational reasoning framework, we define the language in axiomatic style as a theory of *equations* (using the $\approx$ sign) between expressions, rather than directed reductions, but the equations can always also be read as reduction rules by replacing $\approx$ with a reduction operator $\hookrightarrow$.

We will only consider the rule for method calls; all other rules do not add any new aspects to the discussion. We implicitly assume the existence of the standard congruence, transitivity, reflexivity etc. rules for $\approx$. The inference rule for method calls is:

$$\frac{\mathit{MethodLookup}(P, C, m) = (\vec{x}, e)}{P \vdash \texttt{new}\ C(\vec{v'}).m(\vec{v}) \approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/_{\textbf{this}}, {}^{\vec{v}}/_{\vec{x}}\right]}$$

The notation $\vec{v}$ stands for a list of values $v_1, ..., v_n$. Every value in FJ is of the form $\texttt{new}\ C(\vec{v'})$, hence the receiver object is assumed to be in that form. The semantics is in small-step style, hence congruence rules such as $\frac{e \approx e'}{e.m(\vec{e}) \approx e'.m(\vec{e})}$ take care of nested expressions. This inference rule uses an auxiliary lookup function *MethodLookup* that is defined as a simple function on the program. The inference rule above can easily be lifted into predicate logic:

$$\forall P, C, m, \vec{x}, e, \vec{v'}, \vec{v}. \mathit{MethodLookup}(P, C, m) = (\vec{x}, e) \Rightarrow \\ P \vdash \texttt{new}\ C(\vec{v'}).m(\vec{v}) \approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/_{\textbf{this}}, {}^{\vec{v}}/_{\vec{x}}\right]$$

However, this rule would violate the program independence constraint defined above, since the complete program is a parameter of the formula. We can get the full program out of the rules by specializing the lookup function with the program. That is, the (now program-independent) rule in the logic is:

$$\forall C, m, \vec{x}, e, \vec{v'}, \vec{v}. \mathit{MethodLookup}(C, m) = (\vec{x}, e) \Rightarrow \\ \texttt{new}\ C(\vec{v'}).m(\vec{v}) \approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/_{\textbf{this}}, {}^{\vec{v}}/_{\vec{x}}\right] \tag{1}$$

and every class definition is represented by an extension of the axiomatization of the lookup function. For example, the class definition:

```
class Foo {
  void bar(a) { a.m(this) }
}
```

is represented by the formula:

$$\mathit{MethodLookup}(\texttt{Foo}, \texttt{bar}) = (\texttt{a}, \texttt{a.m(this)})$$

So, the rep_L function for this program logic represents every program by a set of program-independent general formulae such as (1) above, and an axiomatization of the lookup functions as illustrated in the example. With regard to the definitions above, it is obvious that this representation is compositional, and that formulae in this logic are program-independent. Since our logic (in this case: first-order predicate logic) is monotonic, and since the representation is compositional, all formulae in the program logic are monotonous, too.

To derive equations about the program, the general formulae can always be "instantiated" with the current program, such that each module is represented by the set of "instances" of the general rules for that program part. For example, the instantiation of the general rules for the class above could is:

$$\forall \vec{v'}, v.\texttt{new Foo}(\vec{v'}).\texttt{bar}(v) \approx \texttt{a.m(this)}\left[{}^{\texttt{new Foo}(\vec{v'})}/_{\textbf{this}}, {}^{v}/_{a}\right]$$

or, equivalently:

$$\forall \vec{v'}, v.\texttt{new Foo}(\vec{v'}).\texttt{bar}(v) \approx v.\texttt{m}(\texttt{new Foo}(\vec{v'}))$$

It is obvious that these instantiations of the general rules can also always be derived in a compositional manner - there is a set of rule instances for each program part.

Modular reasoning assessment

The described representation of OO programs is compositional. All formulae are hence monotonic, as long as we use only the standard monotonic inference rules of the underlying logic (and not rules such as *negation as failure*).

The "sex appeal" of the OO rules is that as soon as we fix C and m in (1), the equation is both monotonic and local. For example, if we take $C = \texttt{Foo}$ and $m = \texttt{bar}$, then we can derive $\texttt{new Foo}(\vec{v'}).\texttt{bar}(v) \approx v.\texttt{m}(\texttt{new Foo}(\vec{v'}))$ in the program above using only local and monotonic reasoning.

If, besides C and m, also the classes of all other variables of the method call ($\vec{v'}$ and $\vec{v}$) that are subsequently used as receivers of method calls are known, then the *complete equational theory* of the method call is monotonic and traceable, because then the equations for all method calls in the body of the method (and in bodies of transitively called methods) can also be deduced.

However, not all formulae have these nice properties - in particular those, where C or m in (1) are not instantiated. For example, we might be interested in whether any method call to any method named `foo` starts with a `print("Hello")` statement. Formally, this could be expressed as:

$$\forall C \forall \vec{x} \forall e \forall \vec{v'} \forall \vec{v} \exists e'.\ \mathit{MethodLookup}(C, \texttt{foo}) = (\vec{x}, e) \Rightarrow \\ \texttt{new}\ C(\vec{v'}).\texttt{foo}(\vec{v}) \approx \texttt{print}(\texttt{"Hello"}); e'$$

Even if this property holds in the program, it could not be derived using standard inference rules of predicate logic. Only if we use negation-as-failure reasoning [8] (we assume the property holds if we cannot find a counterexample), we can derive the property by inspecting the whole program. Hence this property is neither local/traceable, nor monotonic (due to the use of negation-as-failure). The nonmonotonicity of this property can also be seen

$$\frac{\mathit{ApplicableAdvice}(P,C,m)=\vec{a}}{P\vdash \texttt{new}\ C(\vec{v'}).m(\vec{v})\approx \texttt{new}\ C(\vec{v'}).m[\vec{a}](\vec{v})} \qquad (\textsc{Weave})$$

$$\frac{\mathit{AdviceLookup}(P,a)=(\vec{x},e)}{P\vdash \texttt{new}\ C(\vec{v'}).m[a,\vec{a}](\vec{v})\approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/_{\textbf{this}},{}^{\vec{v}}/_{\vec{x}},{}^{\texttt{new}\ C(\vec{v'}).m[\vec{a}](\vec{v})}/_{\textsf{proceed}}\right]} \qquad (\textsc{AdvExec})$$

$$\frac{\mathit{MethodLookup}(P,C,m)=(\vec{x},e)}{P\vdash \texttt{new}\ C(\vec{v'}).m[\emptyset](\vec{v})\approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/_{\textbf{this}},{}^{\vec{v}}/_{\vec{x}}\right]} \qquad (\textsc{MethExec})$$

Figure 1. Weaving-based AO language semantics

by considering an extension of the program with an additional implementation of a `foo` method that does not have the desired property.

This is not just an esoteric example but an inherent problem of describing programs in a monotonic logic. The representation of a program in a monotonic logic is always *incomplete*, meaning that there are always properties F for which neither F nor its negation can be proved. It cannot be complete, because there are fundamental theoretical limitations (Gödel's first incompleteness theorem), and because a complete theory would not be extensible. The problem is that the program is described by a set of constraints in the logic (which has many solutions or "models"), whereas the actual running program is mathematically only a single distinguished model - usually some kind of "minimal" model. To reason about this minimal model, we hence have to apply a closed-world assumption, negation-as-failure, or other similar techniques that make reasoning nonmonotonic. These properties that can only be deduced by nonmonotonic reasoning are not only numerous but also quite relevant and include properties such as conformance to temporal protocols or concurrency properties. Traditional modularity and reasoning mechanisms focus on the monotonic properties only, but this author sees no reason to consider the monotonic properties more important than the nonmonotonic ones.

2.2 Weaving-based AO program logic

We will now study why a weaving-based semantics for AO programs is inappropriate for representing and reasoning about AO programs. As an illustration, let us consider an extension of our little object-oriented language with "around" advice [15] that can advise method calls. We will only consider very simple pointcuts that can advise method calls, but it would be straightforward to add more sophisticated pointcuts. Fig. 1 shows an excerpt of how a typical weaving-based semantics for such a language looks like; we have choosen a presentation in the style of Jagadesaan et al's calculus of aspect-oriented programs [14]. Again, we show only the rules for method calls and advice application; all other rules don't add anything interesting to the discussion.

A method call `new` $C(...).m(\vec{v})$ is executed (read rules from left to right) by first looking up all advice that applies to a method call and sorting the advice in some order (inside the *ApplicableAdvice* lookup function, whose definition is not shown here), and weaving the sorted list of advice $\vec{a}$ into the method call (WEAVE). This weaved method call is then executed by taking the first advice from the list, looking up the formal arguments and body of the first advice, substituting **this** and the formal parameters $\vec{x}$ by the receiver object and the actual parameter values, respectively, and substituting `proceed` by a method call that removes the first advice from the list of pending advice (ADVEXEC). If no advice is left, the original method body is executed (METHEXEC). In both cases, the lookup functions $\mathit{AdviceLookup}$ and $\mathit{MethodLookup}$ return a list with the names of the formal parameters and the advice/method body.

Modular reasoning assessment

The point which prevents a usage of this semantics as a compositional representation function is the *ApplicableAdvice* function: Since we need a sorted list of all advice that apply, the result of this function cannot be computed before the whole program is known, because every program part may contain an advice declaration or advice precedence hint that influences the result of the function. Hence it is not possible to axiomatize this lookup function in a compositional way, as above with the $\mathit{MethodLookup}$ function. Rather, the representation (in particular of the advice lookup function) can only be determined once the full program is known.

Another way to look at the situation is that *no* equation that says something about the right-hand side of the equation (which is basically every non-trivial equation) can be derived without knowing the full program. *No* non-trivial equation involving method calls is monotonic - adding an advice could invalidate the derived equation. All the typical equations that programmers implicitly use to reason about programs (such as the substitution of a name by its definition, the commutativity of certain operations, algebraic laws of data structures etc.) cannot be derived in a modular way.

2.3 Summary

As this discussion shows, it does not make sense to ask whether language X "allows modular reasoning", since this depends on the representation (is it compositional?), on the properties of the inference system (is it monotonic?), and in particular the kind of property one is interested in (is it local/traceable/monotonic?). It also depends on how much abstraction is involved in the representation (for example, with the representations we saw it is not possible to conclude anything about the energy consumption of the CPU during a computation), but we will ignore this issue for now.

In OO languages (or procedural/functional languages), many interesting properties are monotonic and local or traceable if we instantiate all variables that refer to program elements, but properties that involve quantification over program elements can only be answered by nonmonotonic closed-world reasoning.

The weaving-based semantics of an AO language, however, gives rise to a highly anti-modular equational theory of AOP programs, where every non-trivial equation depends on the full program. In this light, the complaints about the modularity (or lack thereof) of AOP are not surprising. In the opinion of this author, the complaints will not stop until a better model than weaving is found to explain aspects.

3. Nonmonotonic logic

Our attempt to give a better explanation of aspects is based on the observation that the kind of modularity that AOP languages strive

```
$E := Th(W); A := \emptyset;$
while there is a default $\delta \notin A$ that is applicable to E {
    $E := Th(E \cup \{consequent(\delta)\}); A := A \cup \{\delta\};$
}
if $\forall \delta \in A. E$ is consistent with all justifications of $\delta$
    then return $E$ else failure
```

Figure 2. Non-deterministic algorithm to compute extensions

for is in many respect similar to the "modularity" of knowledge bases and corresponding inference systems that have been developed in the AI community to model "common-sense" reasoning.

Nonmonotonic logics capture the kind of inference of everyday life in which human reasoners draw conclusions tentatively, reserving the right to retract them in the light of further information. They also allow to "modularize" knowledge in a way that more closely resembles how humans organize knowledge. For example, they allow to formulate general rules and separately, in a different context, exceptions to the general rule. This already sounds a bit like AOP, but before we go into the details of this relationship, we will make things concrete and study one particular variant of nonmonotonic logic: *Default logic*.

Default logic [31] is the most widely used variant of nonmonotonic logic. With default logic, one can state general "rules of thumb", that may or may not be extended by exceptional cases. For example, the knowledge that birds usually fly can be formalized as follows:

$$\frac{bird(X) \;:\; \mathit{flies}(X)}{\mathit{flies}(X)}$$

This rule is a so-called *default*, and can be read as "If X is a bird, and if it is consistent to assume that X flies (that is, it cannot be concluded that X does *not* fly), then conclude that X flies". In general, a default δ has the form $\frac{\varphi :\, \psi_1,\ldots,\psi_n}{\chi}$, where $\varphi, \psi_1, ..., \psi_n, \chi$ are predicate logic formulae, and $n > 0$. The formula φ is called *prerequisite*, the part to the right of the colon, $\psi_1, ..., \psi_n$ *justifications*, and the part below the bar, χ, is the *consequent*. A default is *applicable* to a deductively closed set of formulae E, if $\varphi \in E$ and $\neg\psi_1 \notin E, ..., \neg\psi_n \notin E$.

In general, the set of conclusions that can be drawn from a knowledge base with defaults is not unique. For example, if it is known that members of the green party typically do not like cars, and members of an automobile club usually like cars, and John is member of both green party and automobile club, then it can be concluded both that John likes cars and that he does not like cars.

This seeming chaos is ordered by so-called *extensions* - possible world views based on the given defaults. Technically, an extension is a superset of the knowledge base that is consistent and closed under deduction and application of defaults [31]. In the example, we would have two distinct extensions, in which John likes and does not like, respectively, cars. A large part of the theory of default logic is concerned with the existence and construction of extensions and the relations between different extensions.

Reiter's original definition of extensions is a non constructive fixed-point equation based on the above properties, but we give an equivalent operational definition based on [21, 3, 4]. For this purpose, we define a *default theory* to be a pair $T = (W, D)$ consisting of a set W of predicate logic formulae (sometimes called *background theory*) and a countable set of defaults D. The extensions of T are all sets E that can be generated by the algorithm in Fig. 2. The notation $Th(E)$ denotes the set of all formulae that follow from E using classical deduction (the *deductive closure* of E). The algorithm first uses applicable (recall the definition of *applicable* above) defaults in an arbitrary order to build a candidate for an extension. The consistency check in the last two lines then checks whether E is really an extension. In general, extensions are neither unique (due to the non-deterministic choice of the next default) nor need to exist at all (due to the consistency check).

It may look strange that every default is applied at most once in the algorithm. This is sufficient, because the rule about birds above is technically not a default but a *default schema* since it contains a free variable (namely X). Default schemata are implicitly interpreted to stand for the set of defaults $\frac{\varphi\sigma :\, \psi_1\sigma,\ldots,\psi_n\sigma}{\chi\sigma}$ for all ground substitutions σ that assign ground terms (terms with no free variables) to all free variables in the schema. For example, if we have two birds Tweety and Trixy, then our default schema implicitly stands for two separate defaults $\frac{bird(Tweety) \,:\, \mathit{flies}(Tweety)}{\mathit{flies}(Tweety)}$ and $\frac{bird(Trixy) \,:\, \mathit{flies}(Trixy)}{\mathit{flies}(Trixy)}$.

An important class of defaults are *normal* defaults. A default is normal if its consequent is its only justification, i.e., defaults have the form $\frac{\phi \,:\, \psi}{\psi}$, such as the default rule about flying birds above. A default theory $T = (W, D)$ is *normal* if all defaults in D are normal. Normal default theories are particularly well-behaved. In particular, *all* normal default theories possess extensions; defaults like $\frac{true \,:\, x}{\neg x}$ which can destroy all extensions are not allowed. In fact, it is not hard to see that the consistency check (last two lines) in the algorithm above will *never* fail in a normal default theory and can hence be omitted [31, 3]. All defaults used in this paper will be normal defaults.

4. Compositional representation for aspects

In this section, we study how AOP languages can be represented compositionally in default logic. We propose rules as presented in Fig. 3. The meaning of a method call `new` $C(\vec{v'}).m\langle\vec{a}\rangle(\vec{v})$ is that the advice $\vec{a}$ have already been executed. In contrast, in Fig. 1 an expression `new` $C(\vec{v'}).m[\vec{a}](\vec{v})$ denotes a method call where the execution of all advice in $\vec{a}$ is *pending*. The idea is that a "normal" method call is started with `new` $C(\vec{v'}).m\langle\emptyset\rangle(\vec{v})$, and the brackets are subsequently filled with all advice that have been executed until all applicable advice are in the list, and then the method body gets executed.

There are only two computation rules, (METH) and (ADV). Due to the different meaning of the advice list in method calls, (ADV) *adds* rather than removes the name of the executed advice to the method call that replaces `proceed`. There is no weaving rule anymore. Rather, the behavior of (METH) and (ADV) is controlled by the auxiliary predicates *NextAdvice* and *unadvised*, which are defined using defaults. If there is no information to the contrary, we assume that a method call is unadvised (UNADV). If however, there is some applicable advice a that has not yet been executed, and if it is consistent to assume that it is the next advice to execute[1], then we conclude that a will be the next advice (NEXTADV). Furthermore, a call with applicable advice is not unadvised (SOMEADV).

Since it is the standard notation, Fig. 3 shows the rules in inference rule format, but (METH), (ADV), and (SOMEADV) should again be interpreted as first-order formulae (i.e., replace bar by implication, universally quantify over all meta-variables), such that they fit into the background theory of a default theory. The program P has already been removed from the formulae, so they are program-independent.

Rather than having a global lookup function that returns a sorted list of all applicable advice as in Fig. 1, we model the association

[1] To avoid that two different advice are both simultaneously the next one, we implicitly assume the existence of the usual inference rules of equality, e.g., $\mathit{NextAdvice}(C, m, \vec{a}) = a \wedge \mathit{NextAdvice}(C, m, \vec{a}) = a'$ implies $a = a'$.

$$\frac{\begin{array}{c}\mathit{MethodLookup}(C,m)=(\vec{x},e)\\ \mathit{unadvised}(C,m,\vec{a})\end{array}}{\texttt{new}\ C(\vec{v'}).m\langle\vec{a}\rangle(\vec{v})\approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/\mathbf{this},{}^{\vec{v}}/_{\vec{x}}\right]}\qquad(\textsc{Meth})$$

$$\frac{\begin{array}{c}\mathit{NextAdvice}(C,m,\vec{a})=a\\ \mathit{AdviceLookup}(a)=(\vec{x},e)\end{array}}{\texttt{new}\ C(\vec{v'}).m\langle\vec{a}\rangle(\vec{v})\approx e\left[{}^{\texttt{new}\ C(\vec{v'})}/\mathbf{this},{}^{\vec{v}}/_{\vec{x}},{}^{\texttt{new}\ C(\vec{v'}).m\langle a,\vec{a}\rangle(\vec{v})}/\mathsf{proceed}\right]}\qquad(\textsc{Adv})$$

$$\frac{\mathit{true}\ \mathbf{:}\ \mathit{unadvised}(C,m,\vec{a})}{\mathit{unadvised}(C,m,\vec{a})}\qquad(\textsc{Unadv})$$

$$\frac{\mathit{ApplicableAdvice}(C,m,a)\wedge a\notin\vec{a}\ \mathbf{:}\ \mathit{NextAdvice}(C,m,\vec{a})=a}{\mathit{NextAdvice}(C,m,\vec{a})=a}\qquad(\textsc{NextAdv})$$

$$\frac{\mathit{ApplicableAdvice}(C,m,a)\wedge a\notin\vec{a}}{\neg\mathit{unadvised}(C,m,\vec{a})}\qquad(\textsc{SomeAdv})$$

Figure 3. AO language semantics using default logic

between joinpoint and advice as a predicate *ApplicableAdvice* - the full list of all advice is *never* needed. This is the decisive difference to Fig. 1 that allows a compositional representation. Each advice can be represented independently by a corresponding list of formulae about the advice lookup function/relation.

For example, this is how a simple sample advice is represented:

$$\mathit{rep}_{DL}\left(\begin{array}{ll}\texttt{advice a1(x) around} & \texttt{calls(A.m(int))}\\ & ||\texttt{calls(B.n(int))}\\ \{\texttt{bodya1}\} & \end{array}\right)=$$
$$\{\ \mathit{ApplicableAdvice}(\texttt{A},\texttt{m},\texttt{a1}),\ \mathit{ApplicableAdvice}(\texttt{B},\texttt{n},\texttt{a1}),\ \mathit{AdviceLookup}(\texttt{a1})=(\texttt{x},\texttt{bodya1})\}$$

Normal classes and methods are represented as described in Sec. 2.1. So the full representation of a program consists of the first-order default logic embedding of the rules in Fig. 3, and a set of axioms about *ApplicableAdvice*,*AdviceLookup*, and *MethodLookup* that can be computed for each module in a compositional way.

To appreciate the difference between default and classical logic, assume for a moment that the colons in Fig. 3 would be replaced by conjunction operators (i.e., we would use normal inference rules). In this case, we could never prove a goal of the form $\mathit{unadvised}(C,m,\vec{a})$ or $\mathit{NextAdvice}(C,m,\vec{a})=a$ because the same goal that we want to prove also appears in the premise of its rule. Hence the semantics would be useless. Similarly, if we would just remove the justifications, the semantics would be useless because we could prove $\mathit{unadvised}(C,m,\vec{a})$ for arbitrary C, m, and $\vec{a}$.

Now, the question arises whether the default theory in Fig. 3 has any extensions, and if any, what their relation to the semantics in Fig. 1 is. Luckily, all default rules in Fig. 3 are normal defaults (see Sec. 3). Normal default theories *always* possess extensions [31], which answers the first question.

Is there only a unique extension? No - in case more than one advice is applicable at some point, there is more than one extension, namely one for every possible advice execution order. This reflects the fact that there is no a-priori order among different overlapping advice. The difference to previous approaches is that we can now deal with this situation *within our logical framework*, and study the ambiguity in terms of extensions – we come back to this point in the next section.

For now, let us now analyze to which degree the two language semantics in Fig. 1 and 3 agree with each other. If at most one pointcut applies at any joinpoint, the two semantics agree because there is only one unique extension in the default theory, which is the same theory (in this case: set of equations) that is generated by the conventional semantics. The semantics differ in how they treat shared joinpoints, i.e., situations in which more than one advice applies. In Fig. 1, the *ApplicableAdvice* lookup function orders all applicable advice in some specific order (determined by the definition of *ApplicableAdvice*), whereas in Fig. 3 every potential execution order is represented by a different extension. In the next section, we discuss how *prioritized default logic* [5] can be used to model AspectJ-like global orders and ordering hints (such as *declare precedence* in AspectJ) on advice.

The main point to remember from this section is that the semantics does not require any global operations or lookups. In particular, a global list of all advice that apply at some point is never needed. In contrast to the semantics based on weaving, this semantics gives rise to a compositional representation: Every program part translates into a set of formulae for that program part, and adding a new module does not invalidate existing formulae. Every aspect has a logical meaning of its own: it is represented by a set of axioms about *ApplicableAdvice* and *AdviceLookup*, and these axioms are independent of other parts of the program. Hence, adding classes or aspects to an existing program does not invalidate the representation of the old program. In contrast, in Fig. 1 there is no logical representation of aspects as independent entities; they are only implicitly represented and tangled with the representation of methods to which they apply.

We will discuss the implications of this compositional representation for the other properties of modular reasoning in Sec. 6, but at first we will study how aspect precedence mechanisms can be modeled in this framework.

5. Priorities

This section can be skipped on first reading. In the previous section we have already hinted at the issue of advice ordering. If two advice apply at some joinpoint, the question arises in which order

```
$E := Th(W); A := \emptyset; Prio := \emptyset$
while there is a default $\delta \notin A$ that is applicable to E {
    $C := \{nameof(\delta') \mid \delta' \in D, \delta' \neq \delta, \delta'$ is applicable to $E\}$
    $Prio := Prio \cup \{nameof(\delta) \prec d \mid d \in C\}$
    $E := Th(E \cup \{consequent(\delta)\}); A := A \cup \{\delta\};$
}
if $E$ is consistent with $Prio$
    then return $E$ else failure
```

Figure 4. Refined algorithm to compute priority extensions

the advice are to be executed. Languages like AspectJ use an arbitrary order such as lexicographic order of aspect names by default, but enable the programmer to insert precedence hints into the program. We will now show that such mechanisms are very naturally supported in default logic. In the context of default logic, basically the same problem arises if multiple defaults are applicable at some point, and a multitude of different approaches has been developed to control the situation [5, 10, 38].

In this work, we will consider two simple but powerful variants of default logic with priorities: PDL and PRDL [5] [3, Chap. 8]. In PDL, the priority information is given in the form of a *strict partial order* $<$ on the set of defaults. The set of extensions of a default theory in PDL is restricted to those extensions that respect $<$, i.e., the order of default application in the algorithm in Sec. 2 is compatible with $<$. With such a global order, we could model the default order of aspects as in AspectJ, but not ordering hints (`declare precedence` specifications in AspectJ).

For this reason, we will consider the more powerful variant PRDL, because it allows to model the priority information *within* the logic, rather than as an external partial order as in PDL. In PRDL, every default δ_i has a name d_i. In our case, we will use the names given to the right of the corresponding default rule (such as NEXTADV), indexed by all free variables in the rule (recall that each default rule stands for a set of rule instances); for example, $\text{NEXTADV}_{C,m,\vec{a},a}$ is the name of a rule instance for some C, m, $\vec{a}$, and a.

These rule names can be used with a special symbol $\prec$. The formula $d_1 \prec d_2$ can be read as "give the default with name d_1 priority over the default with name d_2". A term $d_1 \prec d_2$ in PRDL is an ordinary formula that can be used both in the background theory W and in defaults D of a default theory $T = (W, D)$. In order to take these priorities into account, the algorithm to compute extensions has to be modified appropriately. Fig. 4 shows the adapted algorithm. Whenever a default δ is choosen, priority assumptions of the form $nameof(\delta) \prec d$ for all other applicable defaults d are added to the set $Prio$. For a set E to be an extension, E must be consistent with $Prio$, whereby consistent means that $\prec$ is a strict partial order in $E \cup Prio$. Since we are considering only normal defaults at this point, we omit the final consistency check from Fig. 2.

Fig. 5 shows how this mechanism can be used to specify AspectJ-like priorities. Rule (DEFAULT) says that, by default, we assume that two advice are ordered by their default order (such as lexicographic order of their names or compilation order). We assume that this default order is given in the form of an order relation $<_{default}$. The next rule (DECLDEFLT) injects the default order into the $\prec$ relation, provided that we have opted for the default order between the two aspects. If no explicit precedence declaration are given in the program, there will only be one unique extension that generates the same program equality relation as the the semantics in Fig. 1, provided that the $ApplicableAdvice$ function sorts advice in the order given by $<_{default}$.

The last two rules deal with explicit precedence declarations. A declaration `declare precedence` a_1, a_2 is represented by the formula $declareprecedence(a_1, a_2)$ - a trivially compositional representation. Rule (DECLPRECDEF) says that if the program contains a precedence declaration between a_1 and a_2, then these two advice are not in their default order. Finally, rule (DECLPREC) injects the precedence declarations into the $\prec$ relation. The only non-obvious part in this rule is that this rule is formulated as a default rule with $\text{NEXTADV}_{C,m,\vec{a},a_1} \prec \text{NEXTADV}_{C,m,\vec{a},a_2}$ as justification. An alternative would have been to make this a normal inference rule and omit the justification. The difference is relevant if the program contains contradicting precedence declarations, such as `declare precedence` a_1, a_2 and `declare precedence` a_2, a_1. If we would formulate (DECLPREC) as normal inference rule, we would inject this inconsistency into $\prec$ and would thereby destroy all extensions. With the additional justification, however, we get a separate extension for each possible resolution of the conflict. In the example, we would get one extension where a_1 has precedence over a_2 and a different extension where a_2 has precedence over a_1. The main advantage, however, is that we can now resolve the conflict using higher-order priority specifications – a mechanism which we will study next.

The important point of this section is the ease and compositionality with which the AspectJ priority mechanisms can be specified in default logic. There is again no need for any global operation in the specification. Rather, every `declare precedence` declaration just adds another axiom about *declareprecedence* to the knowledge base, without invalidating any existing rule instances.

Dynamic and higher-order priorities

The priority mechanisms in nonmonotonic logic go beyond what has been explored so far in the aspect-oriented community, and we believe that these mechanisms could be used to design new advanced priority mechanisms for AO languages. A typical example for higher-order and dynamic priorities in the AI community is the following example from legal reasoning [5, 6]: According to Uniform Commercial Code (UCC), a security interest in goods is perfected by taking possession of the collateral. According to Ship Mortgage Act (SMA), security interest in a ship may only be perfected by filing a financing statement. UCC is state law, SMA federal law. UCC is more recent than SMA The principle Lex Posterior gives precedence to newer laws. The principle Lex Superior gives federal law precedence to state law. Miller has possession of a certain ship but did not file a financing statement This situation is formalized in Fig. 6.

The first point to notice here is that (LEXPOSTERIOR) and (LEXSUPERIOR) are *dynamic* rules, as they depend on predicates like *moreRecent*, whose denotation may be determined during the computation of an extension. One could easily imagine to design an aspect-oriented priority mechanism along these lines, i.e., one that can take dynamic information into account. The second point to notice here is that we have a case where the two priority rules give contradicting answers. For illustration, assume a default theory $T = (W, D)$ with D as in Fig. 6 and

$$\begin{aligned} W \;= \{ \; & possession, ship, \neg financialstatement, \\ & moreRecent(UCC, SMA), \\ & higherAuthority(SMA, UCC)\} \end{aligned}$$

This theory has two separation extensions, and in only one of them is the security interest perfected.

However, the ambigous choice between (LEXPOSTERIOR) and (LEXSUPERIOR) can be removed by a *higher-order priority rule* which removes the ambiguity and leaves only one unique extension, such as:

$$\text{LEXSUPERIOR}_{X,Y} \prec \text{LEXPOSTERIOR}_{U,V}$$

$$\frac{\mathit{true} : \mathit{defaultOrder}(\{a_1, a_2\})}{\mathit{defaultOrder}(\{a_1, a_2\})} \qquad (\text{DEFAULT})$$

$$\frac{\mathit{defaultOrder}(\{a_1, a_2\}) \wedge (a_1 <_{\mathit{default}} a_2)}{\text{NEXTADV}_{C,m,\vec{a},a_1} \prec \text{NEXTADV}_{C,m,\vec{a},a_2}} \qquad (\text{DECLDEFLT})$$

$$\frac{\mathit{declareprecedence}(a_1, a_2)}{\neg \mathit{defaultOrder}(\{a_1, a_2\})} \qquad (\text{DECLPRECDEF})$$

$$\frac{\mathit{declareprecedence}(a_1, a_2) : (\text{NEXTADV}_{C,m,\vec{a},a_1} \prec \text{NEXTADV}_{Co,m,\vec{a},a_2})}{\text{NEXTADV}_{C,m,\vec{a},a_1} \prec \text{NEXTADV}_{C,m,\vec{a},a_2}} \qquad (\text{DECLPREC})$$

Figure 5. Modeling AspectJ-like priorities in PRDL

$$\frac{\mathit{possession} : \mathit{perfected}}{\mathit{perfected}} \qquad (\text{UCC})$$

$$\frac{\mathit{ship} \wedge \neg \mathit{financialstatement} : \neg \mathit{perfected}}{\neg \mathit{perfected}} \qquad (\text{SMA})$$

$$\frac{\mathit{moreRecent}(X, Y) : X \prec Y}{X \prec Y} \ (\text{LEXPOSTERIOR})$$

$$\frac{\mathit{higherAuthority}(X, Y) : X \prec Y}{X \prec Y} \ (\text{LEXSUPERIOR})$$

Figure 6. Example from [5, 6] illustrating dynamic and higher-order priorities

We believe that such higher-order priority rules are useful in AOP as well. For example, conflicting aspect precedence rules between aspects a1 and a2 could be resolved by rules of the form:

$$\forall C \forall m \forall \vec{a}. \text{DECLPREC}_{C,m,\vec{a},\mathtt{a1},\mathtt{a2}} \prec \text{DECLPREC}_{C,m,\vec{a},\mathtt{a2},\mathtt{a1}}$$

Of course, the priority specifications could be easily made even more sophisticated by including dynamic information from the execution, or by making the priority specifications conditional, or by having third-order (or higher) priority specifications. In each case, the compositionality of the representation is not at risk.

6. Reasoning abouts aspects in default logic

We will now revisit the question of modular reasoning about aspects in the light of the semantics based on defaults. In the second part of this section, we will elaborate on issue of nonmonotonicity.

6.1 Modularity assessment

By having a compositional representation for an AO program in a program logic, the most basic prerequisite for any useful kind of non-global reasoning is fulfilled, although a price had to be paid: The sacrifice of monotonicity. This means that conclusions that are made about a part of the program may have to be withdrawn when learning about a larger part of the program.

With regard to the other modularity reasoning criteria, however, the semantics based on defaults performs much better than the one based on weaving. If we consider our little example from Sec. 2.1 again, then the equation

$\mathtt{new}\ \mathtt{Foo}(\vec{v'}).\mathtt{bar}\langle\emptyset\rangle(v) \approx v.\mathtt{m}(\mathtt{new}\ \mathtt{Foo}(\vec{v'}))$

can be derived using only local reasoning (provided that no aspect advising the call is in our knowledge base).

Also, similarly to the situation in Sec. 2.1, the full equational theory of a method call can be derived using traceable reasoning, if the types of all arguments that influence the method dispatch in subsequent calls are known.

Most importantly, a new class of properties can now be established using local reasoning, namely properties derived from the representations of aspects in the system. In Sec. 2.1, we discussed that a property such as

$$\forall C \forall \vec{x} \forall e \forall \vec{v'} \forall \vec{v} \exists e'.\ \mathit{MethodLookup}(C, \mathtt{foo}) = (\vec{x}, e) \Rightarrow \\ \mathtt{new}\ C(\vec{v'}).\mathtt{foo}(\vec{v}) \approx \mathtt{print}(\text{"}\mathtt{Hello}\text{"}); e'$$

can only be derived using global, nonmonotonic reasoning in the OO language. Let us assume that our AO program contains an advice

```
advice a2() around calls(*.foo(..)) {
  print("Hello"); proceed
}
```

and corresponding representation

$\forall C. \mathit{ApplicableAdvice}(C, \mathtt{foo}, \mathtt{a2})$
$\mathit{AdviceLookup}(\mathtt{a2}) = (\{\mathtt{print}(\text{"}\mathtt{Hello}\text{"}); \mathtt{proceed}\}$

then this property could indeed be derived by local reasoning using only the rules in Fig. 3 and the representation of the aspect! In a sense, it is of course trivial that we can conclude what the advice says, but we believe it is an important observation that, while AO languages make modular reasoning harder in those cases that work particularly well for non-AO languages, they also allow new classes of properties to be established locally - properties that could only be established by global, nonmonotonic reasoning in a non-AO language.

6.2 Dealing with nonmonotonicity

Nonmonotonicity means that some work has to be done in order to check whether a property that was established by looking at a small part of the program still holds in the full program (or still holds after extending the program). However, if possible, it should not be necessary to redo the whole proof of the property. Of course, sometimes it is unavoidable to revisit the program. As an extreme example consider the property that the text of a program has a certain MD4 checksum - the whole program must be revisited after every change to check whether the property still holds.

Our goal in this section is to minimize the number of cases where the program needs to be revisited due to changes in the applicability of defaults – that is, in the case when we learn about new aspects in the system. To this end, we will first examine what a proof of a property in default logic looks like [2]. A *default*

```
bool f(int n) {
  if n<=0 then return g(n)
          else return isPrime(n);
}
bool g(int n) { return isPrime(-n); }

bool isPrime(int n) {
  if n<=1 then return false;
  for (int i=2; i<n; i++) {
    if n modulo i = 0 then return false;
  }
  return true;
}
```

Figure 7. Example program for verification

proof of a formula ϕ_n in a default theory $T = (W, D)$ is a sequence $s = \phi_1, ..., \phi_n$ such that $\phi_1, ..., \phi_n$ is a classical proof in $W \cup \{consequent(\delta) | \delta \in D\}$, and there is a *proof justification set* $J(s)$ satisfying the following conditions:

- Either ϕ_i derives from $W \cup \{\phi_1, ..., \phi_{i-1}\}$ or there is some default δ such that $s = \phi_1, ..., \phi_j, ..., \phi_i, ..., \phi_n$ with $\phi_j = prerequisite(\delta)$, $\phi_i = consequent(\delta)$, and $justification(\delta) \subseteq J(s)$.
- If $\phi \in J(s)$, then for some $\delta \in D$, $\phi \in justification(\delta) \subseteq J(s)$ and $s = \phi_1, ..., \phi_j, ..., \phi_i, ..., \phi_n$ such that $\phi_j = prerequisite(\delta)$ and $\phi_i = consequent(\delta)$.

These two conditions require $J(s)$ to contain exactly the justifications of those defaults that are used in the proof. There are some additional conditions on proofs to make sure that they are indeed valid proofs in default logic [2], but for our purposes it is enough to see that a default proof consists of a number of steps (proven subgoals of the proof), where each step depends on other previous steps, and some of these steps are applications of defaults. The set $J(s)$ represents the justifications of those defaults that were applied during the proof. In our case, it would contain justifications such as "the method call `new` $\mathtt{C}().\mathtt{m}\langle\emptyset\rangle()$ is unadvised", or "the next advice for `new` $\mathtt{C}().\mathtt{m}\langle\emptyset\rangle()$ is `a1`".

Now, if a property ϕ of program P has been established using proof s, and P is extended with aspects (or the developer learns about new aspects) P', we can distinguish three cases:

1. As a first quick check, the representation $rep_L(P')$ can be checked whether it is consistent with the justification set $J(s)$. This check could be performed by a machine. If $J(s)$ is consistent with $rep_L(P')$, then the property is not affected. The set $J(s)$ could even be regarded as a kind of *interface* [18] towards extensions of the program, i.e., only those extensions are admitted that are consistent with $J(s)$ and the proof of any other property of interest in the program. In this sense, $J(s)$ is like the *rely* part in rely-guarantee reasoning [37].
2. If the justification set is not consistent with the extension, then the property potentially no longer holds. However, in many cases it is possible to "repair" the proof by re-establishing the violated subgoal or a "cut" through the set of invalided subgoals (explained in detail below).
3. In the last case, it is necessary to revisit the program. For example, if a new aspect intercepts calls to method `f` and calls `g` instead, where `g` is already part of the old program, and if the implementation of `f` was important for the proof, then it is unavoidable to revisit `g` to re-establish the property.

To illustrate the issue, consider the small program in Fig. 7 (for simplicity, we concentrate on the procedural part of the language in this example and omit classes and objects) and a proof of the property

$\forall n. f(n) \approx true \Leftrightarrow (n > 0 \wedge prime(n)) \vee (n \leq 0 \wedge prime(-n))$

The structure of a proof of this property is depicted in Fig. 8. The picture shows subgoals of the proof, and the dependencies among subgoals. It also shows the justifications of applied defaults (the $J(s)$ set). Those parts of the proof that are irrelevant for our purposes (such as the proof that the body of `isPrime` is correct, which is performed by normal classical reasoning) have been omitted.

Now consider an extension of the program with additional advice. The question is whether the proof (and hence property) is still valid. The quick check is to compare whether the justification set $J(s)$ is consistent with the expansion. In our example this means that if neither `f`, nor `g` or `isPrime` have been advised, then we already know that the property still holds.

If one of the assumptions in $J(s)$ has been violated by the extension, however, the property may no longer hold. For example, consider an expansion of the program with the following "optimization" advice:

```
advice a(n)
        around call(isPrime(n)) {
  if n modulo 2 = 0 then false
                    else proceed;
}
```

This advice violates the justification about `isPrime` not being advised. In terms of the proof structure, it invalidates all subgoals that depend on the corresponding default application. The set of invalidated goals together with their dependencies form an acyclic directed graph with the invalidated assumption at the top and the property the proof is about at the bottom. Now, the basic idea is that it is sufficient to re-establish all goals in some *cut* through this graph of invalidated subgoals, i.e., a minimal set of nodes whose deletion would break the graph of invalidated subgoals apart. In the example, there are three sets of nodes that establish a cut, all of which happen to consist of only a single node (see right-hand side of Fig. 8). Since the purpose of the advice above is to optimize the `isPrime` function while preserving its semantics, we choose the top-most cut and can repair the proof as indicated in Fig. 9.

Of course, it is also possible that a property still holds after an extension, but the proof of the property cannot be maintained in an incremental manner. This could be because the correctness of the property depends on some internal implementation detail of the program that was not relevant for the "old" proof. However, if one wants to hide the internals of the program from the advice writer, one could expose well-defined parts of the proof tree as an interface of the program towards expansions, such that a programmer who wants to extend the program with advice has to make sure that the proof of the property can be repaired as explained above. That is, if the property cannot be re-established incrementally, the advice is rejected.

The main point which makes this whole idea of checking the preservation of properties possible is the compositionality of the program representation. If the program representation is not compositional, the knowledge base of the extended program may look completely different than the knowledge base of the smaller program, hence doing *any* kind of reasoning on program parts seems to be nonsensical.

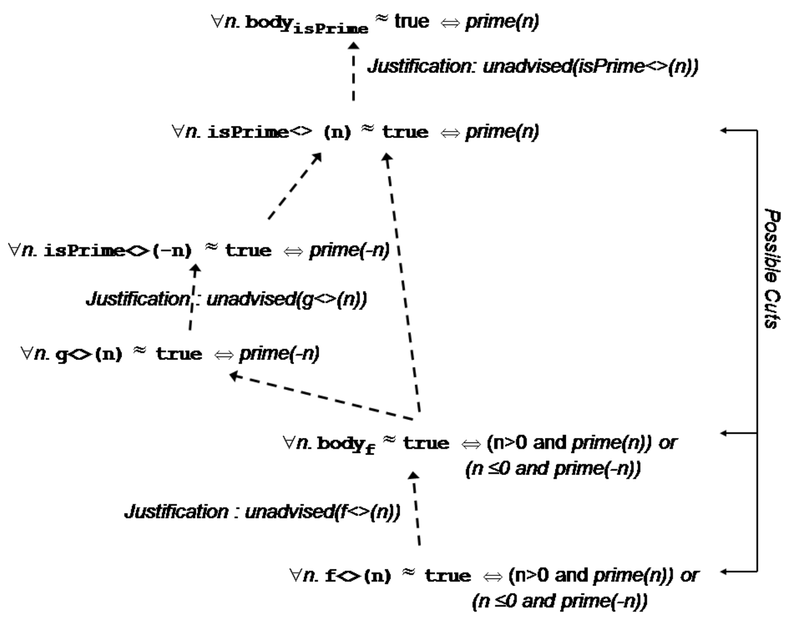
∀n. body_isPrime ≈ true ⇔ prime(n)
Justification: unadvised(isPrime<>(n))
∀n. isPrime<> (n) ≈ true ⇔ prime(n)
∀n. isPrime<>(-n) ≈ true ⇔ prime(-n)
Justification: unadvised(g<>(n))
∀n. g<>(n) ≈ true ⇔ prime(-n)
∀n. body_f ≈ true ⇔ (n>0 and prime(n)) or (n ≤0 and prime(-n))
Justification : unadvised(f<>(n))
∀n. f<>(n) ≈ true ⇔ (n>0 and prime(n)) or (n ≤0 and prime(-n))
Possible Cuts

Figure 8. Proof before extension by optimization aspect

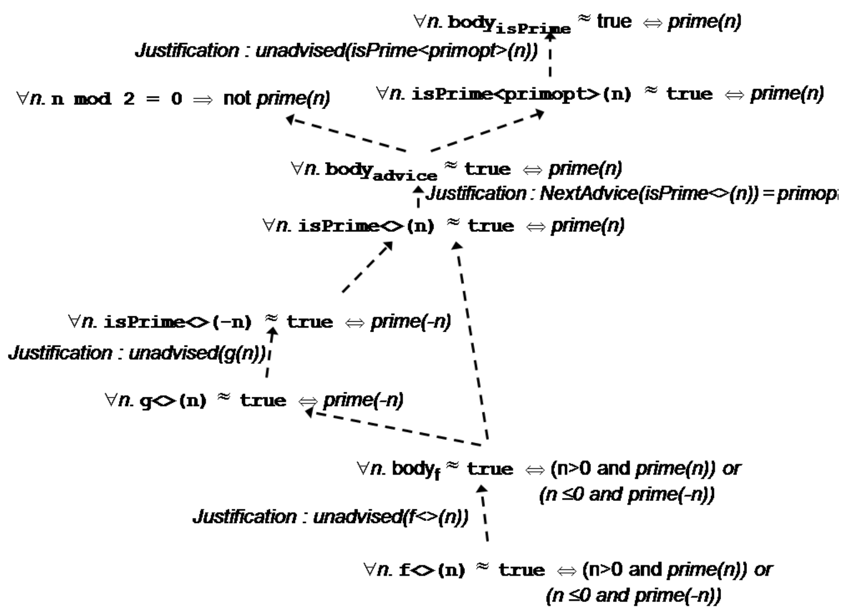
∀n. body_isPrime ≈ true ⇔ prime(n)
Justification : unadvised(isPrime<primopt>(n))
∀n. n mod 2 = 0 ⇒ not prime(n)
∀n. isPrime<primopt>(n) ≈ true ⇔ prime(n)
∀n. body_advice ≈ true ⇔ prime(n)
Justification : NextAdvice(isPrime<>(n)) = primop
∀n. isPrime<>(n) ≈ true ⇔ prime(n)
∀n. isPrime<>(-n) ≈ true ⇔ prime(-n)
Justification : unadvised(g(n))
∀n. g<>(n) ≈ true ⇔ prime(-n)
∀n. body_f ≈ true ⇔ (n>0 and prime(n)) or (n ≤0 and prime(-n))
Justification : unadvised(f<>(n))
∀n. f<>(n) ≈ true ⇔ (n>0 and prime(n)) or (n ≤0 and prime(-n))

Figure 9. Repaired proof after extension by optimization aspect

7. Discussion and related work

In this section we revisit the discussion from Sec. 1 about the connection between modularity and knowledge representation, and elaborate on some related work.

7.1 Modularity and the Frame Problem

A compositional representation is a morphism between a programming language and a logic that preserves the modularity structure. We believe that the existence of such a morphism reveals deep underlying symmetries. For "conventional" procedural, object-oriented, or functional languages, it is quite easy to find compositional representations in conventional predicate logic, whereas we have seen that it is not obvious whether a similar representation in predicate logic exists for AO programs.

We believe that the underlying reason is that conventional procedural, OO, or functional languages and predicate logic have similar notions of abstraction, name definition/binding, and substitution. Lambda abstraction or function/method abstraction is very similar to universal quantification, and the associated notion of "instantiating" an abstraction by substitution is the same. In both worlds, we have clear definitions of and distinctions between name binding and bound occurences of a name.

We believe that the modularity problems (scattering, tangling) that have been observed in these languages are quite related to the frame problem [24], which was one of the main driving forces behind non-monotonic logics. There are many different formulations of the frame problem; the one that makes the connection to modularity most obvious is maybe a variant of the frame problem known as *qualification problem*, which McCarthy defines as follows:

> "It seemed that in order to fully represent the conditions for the successful performance of an action, an impractical and implausible number of qualifications would have to be included in the sentences expressing them." [22]

The logic used in this paper, default logic, provides a new kind of abstraction, namely general rules whose applicability can be influenced by other rules, which can define exceptions to the general rule. Hence it is in general not sound to just "instantiate" the general rules with arbitrary substitutions - just as it is in general not sound to replace a method call with the method body in an AO language. The direct connection between priorities in default logic and precedence mechanisms in AO languages is further evidence of the close relationship. Hence we believe that there is a lot of potential in studying the relationship between nonmonotonic logic and aspects and, more generally, between knowledge representation and modularity mechanisms more closely.

7.2 Related Work

There are ample opportunities to do so. For example, it would probably also be possible to define our language semantics in *autoepistemic logic* [27]. Autoepistemic logic introduces an operator L, where $L\phi$ is interpreted as 'I believe in ϕ'. Using this operator, our (UNADV) rule, for example, could be encoded as

$$\neg L \neg unadvised(o, m, \vec{a}) \rightarrow unadvised(o, m, \vec{a})$$

Since autoepistemic logic is intuitively based on introspection (rather than default rules), autoepistemic logic might provide another interesting interpretation of AOP.

Another well-known approach in nonmonotonic logic is circumscription [22, 23]. We believe that circumscription could be useful to devise a model-theoretic interpretation of AOP. We believe it would be possible to define a variant of our semantics where the *unadvised* and *NextAdvice* predicates are circumscribed (i.e., their meaning is minimized), rather than defining them via defaults. However, this is clearly a topic for future work.

Kiczales and Mezini have informally compared modular reasoning in OO and AO languages in terms of an example [16]. Their distinction between modular and expanded modular reasoning is similar to our distinction between local and traceable reasoning. Their observation that certain properties of their example program can be established more modularly in the the AO version is an example of the general characterization of modular properties in AO languages that was discussed in Sec. 6.1. Their idea of *aspect-aware interfaces* would correspond to a kind of pre-processing step that adds annotations about potentially applicable advice to method signatures. However, even though this step could be performed by a machine, the underlying representation function would still not be compositional - if the program is extended, the attached annotations may look completely different.

Many other authors have proposed restrictions of AOP to ease modular reasoning [1, 33, 9]. In contrast, the goal of this work is to consider AOP in its full generality, but find a more modular explanation of AOP than weaving. The MAVEN tool [11] and the approach by Krishnamurthi et al [18] aim at modular verification of aspects, but these works focus on temporal properties expressed in LTL that are checked via model checkers, and not general equational reasoning.

Our notion of a compositional representation is related to compositionality in denotational semantics [32]. The additional requirement of program-independent formulae makes our compositionality criterion in a sense more strict than the compositionality requirement in denotational semantics, because a denotation can be parameterized with global information. For example, the denotation of a program part in the denotational semantics for an AO language by Wand et al [34] is parameterized by a global "aspect environment", hence a "refactoring" of their semantics into a representation function would be compositional but with program-dependent formulae.

The idea of modularity in language semantics is not new, but usually has a different goal than this work. For example, modular SOS [28] and monadic denotational semantics [26] both aim at improving the modularity of the semantics itself, i.e., it should be easy to extend the programming language in a modular way. In contrast, this work is concerned with the modularity of programs *in* the programming language.

8. Conclusions

"Weaving" as a metaphor to explain the meaning of aspects is a bad idea, because it is inherently non-compositional and is incompatible with any kind of modular reasoning. We claim that the logic with which we explain and reason about a program must reflect and fit to the way we think about the program when designing its modular structure. We have shown that the modularity problems that AOP aims to solve are similar to the frame problem and qualification problem in knowledge representation and logic. As a concrete argument for this connection, we have proposed an AO semantics based on default logic, and have shown that it restores the ability to do modular reasoning, at the price of nonmonotonicity.

We have also shown that there is a deep conflict between direct mapping and monotonicity, and both the power and the problems of AOP can be explained as a new trade-off between these two goals that favors direct mapping at the expense of monotonicity. Lastly, we have argued that nonmonotonicity is unavoidable and invariably also present in classical modularity mechanisms, where reasoning techniques such as a closed-world assumption have to be used to reason about many important properties. However, a closed-world assumption is in some sense a sledge-hammer method, because the whole program has to be investigated for every reasoning step (which is the flip-side of a scattered implementation of a property), hence reasoning becomes very fragile. Nonmonotonic logics can make the unavoidable nonmonotonicity more disciplined and robust by introducing explicit devices such as default rules to represent knowledge in a nonmonotonic way.

Acknowledgements

We thank Todd Milstein, James Noble, the program chair, and the anonymous reviewers for helpful comments.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*, Lecture Notes in Computer Science, pages 144–168. Springer, 2005.

[2] G. Amati, L. C. Aiello, and F. Pirri. Defaults as restrictions on classical Hilbert-style proofs. *Journal of Logic, Language and Information*, 3(4):303–326, 1994.

[3] G. Antoniou. *Non-monotonic reasoning*. MIT Press, 1996.

[4] G. Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999.

[5] G. Brewka. Reasoning about priorities in default logic. In *Proceedings of the 12th national conference on Artificial intelligence (AAAI)*, pages 940–945. American Association for Artificial Intelligence, 1994.

[6] G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4:19–36, 1996.

[7] L. Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[8] K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[9] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT!) at AOSD 2003.*, 2003.

[10] P. M. Dung and T. C. Son. An argument-based approach to reasoning with specificity. *Artif. Intell.*, 133(1-2):35–85, 2001.

[11] M. Goldman and S. Katz. Maven: Modular aspect verification. In *TACAS, Springer LNCS 4424*, pages 308–322, 2007.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[13] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.

[14] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, pages 54–73, 2003.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[17] S. Kraus, D. J. Lehmann, and M. Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artif. Intell.*, 44(1-2):167–207, 1990.

[18] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT FSE'04*, pages 137–146. ACM, 2004.

[19] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2001.

[20] D. Makinson. General patterns in nonmonotonic reasoning. In *Handbook of logic in artificial intelligence and logic programming (vol. 3): nonmonotonic reasoning and uncertain reasoning*, pages 35–110, New York, NY, USA, 1994. Oxford University Press, Inc.

[21] W. Marek and M. Trusczynski. *Nonmonotonic Logic*. Springer, 1993.

[22] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[23] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[24] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[25] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[26] E. Moggi. A modular approach to denotational semantics. In *Category Theory and Computer Science, Springer LNCS 530*, pages 138–139, 1991.

[27] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[28] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[29] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[30] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[31] R. Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[32] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[33] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/SIGSOFT FSE*, pages 166–175, 2005.

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

[35] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1:7–87, August 1990.

[36] J. Winkler. Objectivism: "class" considered harmful. *Communications of the ACM*, 35(8):128–130, 1992.

[37] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[38] X. Zhao. Complexity of argument-based default reasoning with specificity. *AI Commun.*, 16(2):107–119, 2003.