Abstract
Oliveira and colleagues recently developed a powerful model to reason about mixin-based composition of effectful components and their interference, exploiting a wide variety of techniques such as equational reasoning, parametricity, and algebraic laws about monadic effects. This work addresses the issue of reasoning about interference with effectful aspects in the presence of unrestricted quantification through pointcuts. While global reasoning is required, we show that it is possible to reason in a compositional manner, which is key for the scalability of the approach in the face of large and evolving systems. We establish a general equivalence theorem that is based on a few conditions that can be established, reused, and adapted separately as the system evolves. Interestingly, one of these conditions, local harmlessness, can be proven by a translation to the mixin setting, making it possible to directly exploit previously established results about certain kinds of harmlessness extensions.

Categories and Subject Descriptors F.3.3 [Studies of Program Constructs]; D.3.2 [Language Classifications]: Functional Languages; D.3.3 [Programming Languages]: Language Constructs and Features

Keywords aspect-oriented programming, monads, equational reasoning, compositional reasoning, interference

1. Introduction
Aspect-oriented programming promotes separation of concerns at the textual level, but semantic interactions between components of an aspect-oriented program are challenging to predict and control. Consequently, the general issue of interference has received a lot of attention in the AOP literature and related areas, such as feature-oriented programming. A wide range of techniques have been studied, such as program analyses, type-and-effect systems, model checking, and equational reasoning.

Oliveira et al. developed MRI, which stands for Modular Reasoning about Interference, a purely functional model of mental programming with effects. Effects are made explicit through the use of monads. MRI enables both modular reasoning and reasoning about non-interference of effects using a range of reasoning techniques like equational reasoning and parametricity. MRI has been used to express two theorems about harmless mixins. The central notion is that a mixin is harmless if the advised program is equivalent to the unadvised program, provided we ignore the effects introduced by the mixin. In MRI, harmlessness can be defined with respect to any computational effect, as long as an associated projection function exists to ignore the introduced effects. MRI therefore subsumes Dantas and Walker’s notion of harmless advice, which is specific to I/O effects.

While originally formulated as “EffectiveAdvice” with a suggested connection to aspect-oriented programming, MRI does not address quantification; advices are mixins which are applied explicitly. The lack of quantification greatly simplifies modular reasoning, because it is enough to study a single module/function and a mixin in isolation. In addition, MRI only focuses on step-wise applications of mixins, in which the composition of a base component with a mixin can then be treated as a new base component for a subsequent mixin application. In contrast, in the pointcut/advice model of AOP, several aspects live in an aspect environment and are all woven at each join point.

This work addresses the challenge of reasoning about aspect interference in the presence of quantification. It has been argued that unrestricted quantification hampers modular reasoning, thereby requiring a form of global reasoning. Recovering modular reasoning can be achieved by restricting quantification, for instance following the Open Modules approach. Yet, as we demonstrate in this paper, while unrestricted quantification hampers modular reasoning, it is amenable to compositional reasoning: global harmlessness results can be obtained through the composition of smaller proofs. This compositional nature makes it possible to evolve an aspect-oriented system and reuse previously-established results.

The contribution of this paper is to develop a framework for establishing harmlessness results about aspect-oriented systems in a compositional manner. Like MRI, we develop a purely functional model with monadic effects, using Haskell as a convenient source language for System Fω and elaborating the model as a Haskell library. We formulate a general behavioral equivalence theorem between a given aspect-oriented system run with respect to two different aspect environments, modulo projection of additional side-effects. This general theorem is proven assuming four sufficient conditions that have to be established separately. When an aspect-oriented system evolves, only some of these conditions may need to be re-established in order to preserve the general theorem.

In Section 2, we illustrate the challenges of reasoning about aspect interference with quantification. Section 3 briefly introduces

---

* Funded by a CONICYT-Chile Doctoral Scholarship
the necessary background on monads and reasoning, and describes a general model of monadic AOP. Section 3 exposes the main theorem of compositional reasoning, discussing and illustrating each of the sufficient conditions. Section 4 presents a concrete minimal implementation of monadic AOP, which is used in Section 5 to study the local harmlessness condition in details. This section shows how we can exploit the formal results of MRI [22] in our setting. Section 6 discusses related work and Section 8 concludes. Several proofs are included in the paper; others can be found in an extended version available online as a technical report [13].

2. Reasoning about Aspect Interference

To illustrate the challenges of reasoning about aspect interference, we introduce a simple base program (written in an imaginary ML-like language) defined in terms of some known functions \( f \) and \( g \):

\[
\text{prog} \ x \ y = \\
\quad \text{let} \ r_1 = f \ x \ \text{in} \\
\quad \text{let} \ r_2 = g \ y \ \text{in} \\
\quad r_1 + r_2
\]

In the following, we present different changes to a system composed of this program and some aspects, and consider questions related to semantic equivalence. We define aspects as a pointcut/advise pair, and use \text{run} to execute programs with certain aspects.

Adding aspects We first add an aspect to the existing system. For instance, to log all calls to \( f \) we define a new system:

\[
s_1 = \text{run} \ [(\text{call } f \ \text{log})] \ \text{prog}
\]

with a typical implementation of the logging advice:

\[
\text{log} \ \text{proceed} \ x = \\
\quad \text{print} \ "\text{Entering function} \ldots" \\
\quad \text{proceed} \ x
\]

Is the behavior of \( s_1 \) equivalent to the original program? Strictly speaking, they are not equivalent if we consider the output generated by \text{print}. However, we observe that the return value of the system is left unchanged, and that if we ignore the printed output, both systems are equivalent. This corresponds to the notion of harmlessness established in MRI [22]. In the general case, establishing that applying the logging aspect is harmless requires to reason globally about the aspect and the composed system.

Some questions arise when we see, intuitively, that the logging advice is harmless for every function on which it may be applied. This property of logging when seen as a mixin is formalized and proven in MRI, but can we use this knowledge when the advice is applied to a system via quantification?

Widening quantification We now widen the quantification of the logging aspect, modifying the pointcut to match additional join points. For instance, if we now want to log calls to \( g \), it suffices to define a combined pointcut:

\[
s_2 = \text{run} \ [(\text{call } f \ \lor \text{call } g \ \text{log})] \ \text{prog}
\]

Intuitively, this change is also harmless. But how to prove it formally? Do we need to reason globally about the system from scratch? or can we reuse some facts from the proof that logging \( f \) in the system is harmless?

Evolving the base program We now evolve the base program by replacing the use of \( f \) with that of another function \( h \):

\[
\text{prog'} \ x \ y = \\
\quad \text{let} \ r_1 = h \ x \ \text{in} \\
\quad \text{let} \ r_2 = g \ y \ \text{in} \\
\quad r_1 + r_2
\]

\[
s_3 = \text{run} \ [(\text{call } f \ \lor \text{call } g, \text{log})] \ \text{prog'}
\]

A first observation is that \text{call } f \ will never match. We must change references to \( f \) also in the aspect environment:

\[
s_4 = \text{run} \ [(\text{call } h \ \lor \text{call } g, \text{log})] \ \text{prog'}
\]

Changing \( f \) for \( h \) will most assuredly modify the semantics of the base program, and consequently of the system. This is expected when the base program is evolving. However, we may want to know if the logging aspect is still harmless in this new system. The question is: what amount of reasoning do we need to perform? Do we need to prove again that logging is harmless with respect to the whole system, or can we reason compositionally and only verify that the advice is harmless with respect to \( h \)?

Widening quantification, revisited Let us now consider a memoization aspect, with the following advice definition:

\[
\text{memo} \ \text{proceed} \ x = \\
\quad \text{if} \ (\text{member } x \ \text{table}) \ \text{then} \ \text{table} [x] \\
\quad \text{else} \ \text{let} \ r = \text{proceed} \ x \ \text{in} \\
\quad \quad \text{insert} \ (\text{table}, x, r) \\
\quad \text{return} \ r
\]

The advice maintains a reference to a lookup \text{table} of precomputed values, indexed by argument \( x \). If the result bound to \( x \) is already in the table, it is immediately returned. Otherwise the value is computed, stored in the table for future references, and returned.

It is intuitively clear that adding memoization on calls to \( f \) is harmless. In fact, if we manually apply \text{memo} as a mixin on top of \( f \), then we even know formally that it is harmless [22].

Now, if we follow the quantification widening scenario from above—which was harmless with the logging advice—is the harmlessness of memoization preserved?

\[
s_5 = \text{run} \ [(\text{call } f \ \lor \text{call } g, \text{memo})] \ \text{prog'}
\]

The answer to the question actually depends on the context in which the advice is applied. In a context where \( f \) and \( g \) actually are the same function, or one of both is never applied, then harmlessness is preserved. But if \( f \) and \( g \) are different functions that are both applied, the behavior of the composed system is drastically affected because the same lookup table is used to store results from both functions!

Compositional reasoning The examples presented above illustrate that, in presence of quantification, it is generally not enough to establish local properties for aspects, but it is also required to reason about the context in which those aspects are applied. Therefore, the modular reasoning techniques developed in the case of MRI are not directly applicable in a setting with quantification, because some form of global reasoning is generally required.

But global reasoning need not be monolithic. The contribution of this work is to provide a formal framework to establish global equivalence properties in a compositional manner. Compositional reasoning facilitates the task of formally establishing properties about aspect-oriented programs. In practice, while it is possible to apply monolithic global reasoning to tiny systems like the ones considered in this section, this approach hardly scales to larger systems. Furthermore, compositional reasoning accommodates software evolution: it makes it possible to reuse previously-established results that are stable under the considered change scenarios.

3. Monads, Reasoning, and Monadic AOP

The compositional reasoning framework proposed in this work is formulated in a monadic setting. We first briefly review monads and monadic reasoning, and then describe a monadic formulation...
of AOP, which serves as the foundation for the formal development in the following sections.

3.1 Monads and Monadic Reasoning in a Nutshell

Monads are a denotational approach to modeling and reasoning about computational effects in pure functional languages [20, 23], and are widely used in Haskell. A monad is defined by a type constructor \( m \) and functions \( \text{bind} \) (\( \Rightarrow \) in Haskell) and \( \text{return} \). At the type level a monad is a regular type constructor, although conceptually we distinguish a value of type \( a \) from a computation in monad \( m \) of type \( m \ a \). Those computations produce values of the given type and may perform side effects, such as a mutable state and error handling. Additionally, monads provide a uniform interface for computational effects, as specified in the \text{Monad} type class (Figure 1). The \text{return} function promotes a value of type \( a \) into a computation of type \( m \ a \). Computations are composed sequentially using the \text{bind} operator. The concrete definitions of \text{return} and \( \Rightarrow \) depend on the computational effect being implemented.

**Monad transformers** A monad transformer is a type constructor that allows one to construct a monad stack that combines several effects [19]. The \text{MonadTrans} type class (Figure 1) defines an interface for monad transformers. The purpose of the \text{lift} operation is to promote a computation from an inner layer of the monad stack, of type \( m \ a \), into a computation in the monad defined by the complete stack, with type \( t \ m \ a \). Each specific transformer \( t \) must declare how to make \( t m \) an instance of the \text{Monad} class.

**Monadic programming in Haskell** Monadic programming in Haskell is provided by the standard Monad Transformers Library (known as \text{mtl}) which defines a set of monad transformers that can be flexibly composed together. Throughout this paper we will use the state (\text{S} \_\_\_\_ \text{S}) and writer (\text{W} \_\_\_\_ \text{W}) monad transformers. In Figure 1 we summarize the types of their constructors (\text{S} \_\_\_\_ \text{S}, \text{W} \_\_\_\_ \text{W}), evaluation functions (\text{runS} \_\_\_\_ \text{S}, \text{runW} \_\_\_\_ \text{W}), and projection functions (\text{S} \_\_\_\_ \text{S}, \text{W} \_\_\_\_ \text{W}). The projection functions remove the corresponding effect from the monad stack (here, by discarding the threaded state or writer).

**Polymorphism in the monad stack** In addition to the particular monad transformers, the \text{mtl} defines a set of type classes associated to particular effects. This allows us to constrain a monad stack such that it presents a particular effect, while being polymorphic in the actual shape of the stack. Figure 1 shows the \text{S} \_\_\_\_ \text{S} and \text{W} \_\_\_\_ \text{W} classes that abstract the state and writer effects. The \text{get} operation retrieves the current value, which can be updated using \text{put}. Similarly, the \text{tell} operation appends a value \( w \) to its output. Note that the \text{S} \_\_\_\_ \text{S} and \text{W} \_\_\_\_ \text{W} transformers are the canonical instances of these type classes, and that the evaluation functions provide the initial values for these computations.

**Equational reasoning and observational equivalence** Equational reasoning is the process of transforming a program by replacing expressions in a manner similar to high-school algebra. Expression \( e_1 \) can be replaced by \( e_2 \) only if the two are equivalent. Observational equivalence, denoted as \( \equiv \) in the paper, is an equivalence relation between expressions that holds whenever two expressions have the same observable behavior. That is, \( e_1 \equiv e_2 \iff \) for every program context \( C \), both \( C[e_1] \) and \( C[e_2] \) yield the same value, or both diverge. For example, consider the \( \eta \)-reduction rule from the \( \lambda \)-calculus, which states that \( \lambda x. f x \equiv f \) (when \( x \) is free in \( f \)). Also, Haskell provides the \text{do} notation as syntactic sugar for \( \equiv \), hence \( \text{do} \{ x \leftarrow f; g x \} \equiv f \Rightarrow \lambda x \rightarrow g x \).

**Monad laws** Monad laws are crucial for equational reasoning in a monadic setting [23]. A proper monad is one that obeys the following three laws:

\[
\begin{align*}
\text{return} x & \equiv f x \quad & \text{left identity} \\
(p \Rightarrow q) & \equiv p \quad & \text{right identity} \\
(p \Rightarrow f) & \equiv h \quad & \text{associativity}
\end{align*}
\]

The first two laws, left and right identity, state that \text{return} neither changes the value nor performs any computational effect. The associativity law states that only the order of computations is relevant in a \( \Rightarrow \) expression. In the same way, monad transformers need to satisfy the following laws:

\[
\begin{align*}
\text{lift} \circ \text{return} & \equiv \text{return} \quad & \text{identity preservation} \\
\text{lift} (m \Rightarrow f) & \equiv \text{lift} m \Rightarrow (\text{lift} \circ f) \quad & \text{comp. preservation}
\end{align*}
\]

Note that Haskell does not enforce that declared instances of the \text{Monad} or \text{MonadTrans} classes actually respect these laws. This has to be proven separately for each considered instance.

3.2 Monadic AOP

Our approach to compositional reasoning relies on a monadic formulation of AOP, but is independent from the concrete implementation of an aspectual computation monad transformer. Previous work by Tabareau et al. [26] develops a full-fledged polymorphic transformer. In Section 5 we describe a simple monomorphic implementation of the model, which we use to develop local reasoning about interference using the techniques of MRT [22].

In this section, we define an aspectual computation monad transformer denoted \( \text{A} \) in an abstract manner, by prescribing its interface and properties. The theorem of compositional reasoning in Section 5 is established based on this abstract specification only.

**Join point model** We consider a join point model in which join points are function applications. In existing AOP languages, there are many ways by which pointcuts select advised entities: for instance, by name (e.g. method names in AspectJ [16], function names in AspectML [8], by reference equality (e.g. AspectScheme [12], AspectScript [21]), by their type (e.g. AspectJ, AOPHaskell [26]), using a mechanism to explicitly attach tags or types to join points (e.g. Ptolemy [23], JPLs [4]), etc.

Here, we abstract over these concrete design choices by introducing an abstract join point type, on which pointcuts predicate:

<table>
<thead>
<tr>
<th>data</th>
<th>( Jp \ m \ a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>( Pc \ m \ a \ b = Jp \ m \ a \ b \rightarrow \text{Bool} )</td>
</tr>
</tbody>
</table>
The type variables respectively denote the underlying monad stack, and the argument and return types of the applied function. The concrete representation of \( Jp \) can hold more information (e.g., contextual information, tags) or less, if some information is not meant to be used in pointcuts.

A denotational model cannot assume implicit generation of join points, so we require the presence of an open application operator \( # \) that takes a function of type \( a \to A_T m b \) and returns a function of the same type whose application produces a join point (this effect is encapsulated in the \( A_T \) monad transformer):

\[
(#) :: (a \to A_T m b) \to (a \to A_T m b)
\]

An open application is realized explicitly using \( #: f \neq 2 \) is the same as \( f 2 \) except that the application generates a join point that is subject to aspect weaving. Note that, in general, there is no reason to assume a single manner to generate join points, so there can indeed be a family of operators \( #^i \), which are interpreted by the aspect weaver as needed. Finally, one can view a partial open application \( f #^i \) as an open function, whose application produces join points.

An advice is a function that executes in place of a join point matched by a pointcut. The first argument of the advice, typically called \( proceed \), is a function which represents the original computation at the matched join point. An aspect simply pairs a pointcut with an advice.

\[
\text{type} \quad \text{Advice} m a b = (a \to m b) \to (a \to m b)
\]

\[
\text{type} \quad \text{Aspect} m a b = (Pc m a b, \text{Advice} m a b)
\]

**Aspect environment**

The aspects to be deployed in a given aspectual computation are specified in a list of aspects called an *aspect environment*:

\[
\text{type} \quad \text{AEnv} m = ... \quad \text{-- an ADT to be specified}
\]

Supporting polymorphic aspects implies that the aspect environment should be an heterogeneous list. Preserving type safety of aspect weaving then requires some care, as discussed elsewhere [26]. In order to avoid accidental complexity, we do not consider this issue in this paper.

**Aspectual computation**

Given a concrete \( A_T \) transformer, we require a function that evaluates an \( A_T \) computation given an aspect environment:

\[
\text{run}_{A_T} :: \text{Monad} m \Rightarrow \text{AEnv} (A_T m) \to A_T m a \to m a
\]

**Abstracting open applications**

Similarly to the \( S_M \) and \( W_M \) type classes, we introduce a type class to define an abstract interface for performing open applications:

\[
\text{class} \quad \text{Monad} m \Rightarrow k_M m \quad \text{where}
\]

\[
#^i :: (\text{Int} \to m \text{Int}) \to (\text{Int} \to m \text{Int})
\]

\[
\text{instance} \quad \text{Monad} m \Rightarrow k_M (A_T m) \quad \text{where}...
\]

The only operation of this class is \( #^i \), and we require that any monad \( A_T m \) be an instance of this class. Note that \( k_M \) allows a form of type-based reasoning about open applications: any function of type \( \forall m.D m \Rightarrow a \to m b \), where \( D \) is a class constraint that does not entail \( k_M \), cannot perform any open applications (and hence cannot emit join points).

### 3.3 Necessary Properties of \( A_T \)

To be a correct model, the \( A_T \) transformer needs to satisfy a number of properties. First, it has to satisfy the monad transformer laws, and when applied to any monad \( m \), the monad laws must be satisfied as well.

Moreover, for all aspect environments \( aenv \), the function \( \text{run}_{A_T} \ aenv \) must be a *monad morphism*.

**Definition 1.** A monad morphism \( h \) is a function of type

\[
h :: \forall a.M_1 a \to M_2 a
\]

that transforms computations in one monad \( M_1 \) into computations in another monad \( M_2 \). The function satisfies two laws:

\[
\begin{align*}
& h \circ \text{return} \equiv \text{return} \\
& h \ (m \bowtie f) \equiv h \ m \bowtie h \circ f \quad (\forall m, f)
\end{align*}
\]

For \( \text{run}_{A_T} \), the first monad is \( A_T m \) and the second monad is just \( m \). Moreover, the two monad morphism laws have an intuitive meaning in this setting: the first law expresses that weaving has no impact on pure computations, and the second law expresses that weaving is compositional [1].

In the same spirit, we also require that a third law holds for \( \text{run}_{A_T} aenv \)

\[
\text{run}_{A_T} aenv \circ \text{lift} \equiv \text{id}
\]

This law expresses that \( \text{run}_{A_T} aenv \) is a left inverse of \( \text{lift} \). In words, weaving an effectful computation that does not involve open applications has no impact.

These laws have to be established whenever a concrete \( A_T \) transformer is implemented. We will come back to this when presenting a simple \( A_T \) transformer in Section 3.

### 3.4 Running Example in Monadic Style

Section 2 used pseudo-code to describe a base program and aspects. In Haskell, the base program is defined in monadic style using the do notation as follows:

\[
\text{prog} \ x \ y = \text{do} \ r_1 \leftarrow f \ #^i \ x \\
& r_2 \leftarrow g \ #^i \ y \\
& \text{return} \ (r_1 + r_2)
\]

The program can be run as an aspectual computation in the \( A_T \) transformer with a logging aspect on open applications of \( f \) as follows:

\[
\text{run}_{A_T} [(f\text{Pc}, \log)] \quad \text{(prog 5 12)}
\]

The pointcut \( f\text{Pc} \) is left undefined at this stage, since in this abstract model we do not prescribe a specific way to denote functions. The definitions of the \text{log} and \text{memo} advices in monadic style is given in Figure 2. Their types reflect their side effects.

### 4. Compositional Reasoning, Formally

This section formalizes our approach to compositional reasoning about aspect interference. This approach revolves around the following general theorem, which provides a framework for the reasoning. The theorem considers an AOP system that is run with respect to a particular aspect environment \( aenv \). The theorem states that, under four sufficient conditions, the system preserves its observable behavior under an alternative aspect environment \( aenv' \) that may introduce additional effects. With the four conditions it provides a step-by-step guide to proving non-interference.

A key property of the theorem is that it supports compositional reasoning. Compositionality is achieved because the theorem splits the system into two parts, an open function \( f \ #^i \) and a context \( c \), whose conditions are independent, can be proven separately, and can be reused in different compositions. Moreover, the system

---

2 If \( A_T \) supports dynamic deployment of aspects (as in [25]), weaving cannot be compositional. We can nevertheless prove the monad morphism laws for the static fragment, and deal with dynamic deployment on a case-by-case basis.

3 This law actually subsumes the first monad morphism law, as \( \text{return} \equiv \text{lift} \circ \text{return} \).
Given an expression:

\[ \text{system :: } \forall m. C \Rightarrow A \rightarrow \mathbb{A}_T \Rightarrow m B \]

Here \( A \) and \( B \) are some types, and \( m \) is a type variable constrained by some type class constraints \( C \) that at least require \( m \) to be an instance of Monad.

We assume that system is given in terms of the following decomposition:

\[ \text{system} \equiv c \ (f \ #^i) \]

where \( c, f \) and \( i \) are arbitrary values of the following types (with \( C_f \) entailed by \( C \); again \( A' \) and \( B' \) are some types):

\[ c :: \forall m. C \Rightarrow (A' \rightarrow \mathbb{A}_T \Rightarrow m B') \rightarrow A \rightarrow \mathbb{A}_T \Rightarrow m B \]
\[ f :: \forall m. C_f \Rightarrow A' \rightarrow \mathbb{A}_T \Rightarrow m B' \]

Also, we are given two aspect environments \( aenv \) and \( aenv' \) of types:

\[ aenv :: \forall m. D \Rightarrow AEnv (\mathbb{A}_T m) \]
\[ aenv' :: \forall m. D \Rightarrow AEnv (\mathbb{A}_T (T m)) \]

where \( T \) is some instance of MonadTrans and \( D \) is a type class constraint that requires \( m \) to be at least an instance of Monad.

The given projection function:

\[ \pi :: \forall m. a. \text{Monad } m \Rightarrow T m a \rightarrow m a \]

is a left-inverse of \( \text{lift} \) that removes the additional \( T \) effect from the monad stack \( T m \).

If the four conditions on \( c \) and \( f \) given below hold, then we have that:

\[ \text{run}_{\mathbb{A}_T} \ aenv \ \text{system} \equiv \pi \ (\text{run}_{\mathbb{A}_T} \ aenv' \ \text{system}) \]

The four conditions on \( c \) and \( f \) are:

1. Compositional weaving

\[ \forall env. \text{run}_{\mathbb{A}_T} \ env \ (c \ (f \ #^i)) \equiv \text{run}_{\mathbb{A}_T} \ env \ c \ (\text{lift} \circ \text{run}_{\mathbb{A}_T} \ env \circ (f \ #^i)) \]

2. Compositional projection

\[ \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (c \ (\text{lift} \circ \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i))) \equiv \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (c \ (\text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i))) \]

3. Contextual harmlessness

\[ \text{run}_{\mathbb{A}_T} \ aenv \circ c \circ (\lambda g \rightarrow \text{lift} \circ g) \equiv \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{lift} \circ g) \]

4. Local harmlessness

\[ \text{run}_{\mathbb{A}_T} \ aenv \circ (f \ #^i) \equiv \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i) \]

**Proof.** The proof proceeds by straightforward equational reasoning:

\[ \text{run}_{\mathbb{A}_T} \ aenv \ \text{system} \equiv \{	ext{-system decomposition -}\} \]
\[ \text{run}_{\mathbb{A}_T} \ aenv \ (c \ (f \ #^i)) \equiv \{	ext{-compositional weaving -}\} \]
\[ \text{run}_{\mathbb{A}_T} \ aenv \circ c \circ (\text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i)) \equiv \{	ext{-compositional projection -}\} \]
\[ \text{run}_{\mathbb{A}_T} \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i)) \equiv \{	ext{-compositional weaving -}\} \]
\[ \text{run}_{\mathbb{A}_T} \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i)) \equiv \{	ext{-contextual harmlessness -}\} \]
\[ \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i)) \equiv \{	ext{-local harmlessness -}\} \]
\[ \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{run}_{\mathbb{A}_T} \ aenv' \circ (f \ #^i)) \equiv \{	ext{-system decomposition -}\} \]
\[ \pi \circ \text{run}_{\mathbb{A}_T} \ aenv' \ \text{system} \]

\[ \square \]

We now explain and illustrate how the theorem can be used.

### 4.1 System Decomposition

The starting point is to view the system as the composition of a particular function \( f \) and a context \( c \). For instance, we can write our running example as \( c_1 \ (f_1 \ #^i) \) where

\[ f_1 = f \]
\[ c_1 = \lambda f \ x \ y \rightarrow \text{do} \ r_1 \leftarrow f \ x \]
\[ r_2 \leftarrow g \ #^j \ y \]
\[ \text{return} \ (r_1 + r_2) \]

Here the context \( c_1 \) is just system abstracted over \( f \ #^i \). Note that the same system can be decomposed in many different ways, in order to focus on different open functions.

### 4.2 Compositional Weaving

The first condition states that weaving the composite system is equivalent to weaving the context \( c \) and the function \( f \) separately and then composing them.

While the compositional weaving condition is formulated in terms of the specific \( c \) and \( f \), it comes almost for free from the three laws that run\(_{\mathbb{A}_T} \ aenv \) satisfies (recall Section 3).

To see why, let us consider the essential ways in which \( c \) can use \( f \). There are two permitted ways:

1. \( c \) does nothing with \( f \), and thus whether \( f \) is woven or not is inconsequential.
2. $c$ invokes $f$ (once or more), which means embedding it in its larger computation (once or more) with $\exists x$, which is where the second law comes in. Note that the second law can be used repeatedly to tackle a larger computation sequence $m \geq f_1 \geq \ldots \geq f_n$. However, there is also one way in which the condition can be violated:

3. The context $c$ is itself weaving the open function with a custom aspect environment. One such example is:

$$c = \lambda f \rightarrow \text{lift} \circ \text{run}_{\lambda T} [] \circ f$$

where $c$ weaves the function with an empty aspect environment, irrespective of the aspect environment used to weave $c$ itself.

This illegal use of $f$ can be avoided by introducing a measure of parametricity. Instead of using the fixed monad transformer $\lambda T$ and its fixed function $\#_1$ in $c$ and $f$, we make $c$ and $f$ parametric in the particular type and function definition. This parameterization is conveniently achieved by imposing the $\lambda M$ constraint on the monad stack instead of applying the $\lambda T$ transformer. It prevents $c$ from invoking the weaving function $\text{run}_{\lambda T}$ locally on $f$ because $\text{run}_{\lambda T}$ only works for $\lambda T m'$ and not for all possible $m$ that instantiate $\lambda M$.

We summarize our technique for establishing compositional weaving in the following conjecture.

**Conjecture 1.** Provided that $f$ and $c$ have the following polymorphic types:

$$c :: \forall m. (C \cdot m, \lambda M m) \Rightarrow (A' \rightarrow m B') \rightarrow A \rightarrow m B$$

$$f :: \forall m. (C f m, \lambda M m) \Rightarrow A' \rightarrow m B'$$

the condition of compositional weaving holds.

We believe that this conjecture can be proven with logical relations, but that is out of the scope of this paper.

### 4.3 Compositional Projection

The second condition expresses that composing the projected context $c$ and projected function $f$ is equivalent to projecting the composition.

This condition has a similar shape as that for compositional weaving. Hence, in the case that the projection function $\pi$ is a monad morphism, then the same solution as for compositional weaving applies. For instance, the projection $\pi_W$ of the writer effect (used in the logging advice) is well-known (and easily verified) to be a monad morphism. This means that, if the system abstracts over the implementation of the writer effect with the type class constraint $\forall M$, then its projection is indeed compositional.

However, it is a very strong requirement for the projection function to be a monad morphism. For instance, the projection $\pi_S$ of the state effect is not a monad morphism:

$$\pi_S 0 (get \geq \lambda x \rightarrow \text{put} \ (x + 1) \geq \text{get}) \equiv \text{return} 1$$

$$\pi_S 0 (get \geq \lambda x \rightarrow \text{put} \ (x + 1)) \geq \pi_S 0 \geq \text{get} \equiv \text{return} 0$$

This explains why we must be careful when adding the memo advice of Figure 3, which has a memo table as its state, to our running example. If the pointcut of this advice matches both the function $f$ on the one hand and the function $g$ in the context $c$ on the other hand, then the two uses of the advice may interfere through the shared state. For instance, the result for $f 3$ may be stored in the table and later wrongly used as if it were the result for $g 3$. This problem is not discovered when we consider the impact of memo on $c$ and $f$ separately. On the contrary, memo is contextually and locally harmless, but globally harmful. We only discover this problem because compositional projection does not hold. This illustrates why compositional projection is a crucial condition.

In some cases, the use of memo in a larger system is nevertheless harmless. As we cannot take the monad morphism route to establishing this, we need to resort to alternative techniques.

- If the woven function $\text{run}_{\lambda T} aenv' \circ f \#_1$ does not use the projected effect, then projection is indeed compositional. This is for instance the case when memo does not advise $f$. We can formally capture this as:

$$\exists h, \text{lift} \circ h \equiv \text{run}_{\lambda T} aenv' \circ f \#_1$$

Let us now reason about the relevant part of the left-hand side of the condition:

$$\text{lift} \circ \text{lift} \circ \pi \circ \text{run}_{\lambda T} aenv' \circ f \#_1$$

$$\equiv \{ \text{assumption} \} \text{lift} \circ \text{lift} \circ \pi \circ \text{lift} \circ h$$

$$\equiv \{ \pi \text{ is left inverse of lift} \} \text{lift} \circ \text{lift} \circ h$$

$$\equiv \{ \text{assumption} \} \text{lift} \circ \text{run}_{\lambda T} aenv' \circ f \\#_1$$

If we plug this conclusion into the left-hand side of the compositional projection condition, we obtain its right-hand side. In other words, the condition follows from the assumption.

- The dual assumption from the above is that the context $c$ does not use the projected effect. This is for instance the case when memo advises $f$ but not $c$. Unfortunately, this case is not as straightforward. While $c$ does not directly interfere with the effect, it may indirectly create interference by invoking $f$ repeatedly and those invocations may interfere with one another through their shared effect. This requires reasoning about the compatibility of an advised $f$ with itself. For instance, in the case of memo it is perfectly fine for multiple invocations of $f$ to share the memo table; in fact, that is exactly the point of memoization. A counterexample is an advice that monitors whether a function is invoked at most $n$ number of times, where $n$ is the first input its called with, and raises an error when that limit is exceeded. This advice is perfectly fine for a function in isolation that takes $n$ (recursive) calls, but when there are multiple separate invocations, then the error may be triggered inadvertently.

Note that we can safely memoize both $f$ and $g$ in our example, if separate tables are used. This amounts to using two instances of memo that each act on a different store layer in the monad stack. In this setup, the state of the components is isolated from each other. Hence, this scenario involves the two classes of compositional projection discussed above.

### 4.4 Contextual Harmlessness

The third condition expresses that as far as the context $c$ is concerned, the aspect environments $aenv$ and $aenv'$ are indistinguishable. There are various ways in which $aenv$ and $aenv'$ can be related to this to be true, for example:

- Unused aspects $(pc, a)$, where the pointcut $pc$ does not match any join point in $c$, can be freely added or removed.

- Two aspects $(pc_1, a_1)$ and $(pc_2, a_2)$ can be reordered if they either do not match on the same applications in $c$ or their advice commute $(a_1 \circ a_2 \equiv a_2 \circ a_1)$.

- The pointcut of an aspect can be replaced by one that matches the same join points in $c$.

- The advice of an aspect can be replaced by one that behaves in the same way with respect to $c$. 
Multiple aspects can be replaced simultaneously by another set of aspects that together behave in the same way on \( c \), redistributing the work among themselves, e.g. splitting a predicate into two disjoint ones.

Note that the contextual harmlessness condition is a variant of the general theorem itself, but on a smaller system that only consists of the context \( c \). Hence, it can be proven by recursively decomposing the context and invoking the general theorem on the two parts. This insight is essential to scale up our approach from a two-function system to arbitrarily complex systems.

For instance, in the running example we can build a simpler system from \( c_1 \), namely \( c_1 \) (\( \lambda \mathit{h} \). \( \mathit{h} : \mathbb{C} \Rightarrow \mathbb{A} \Rightarrow \mathbb{B} \)), which can be decomposed as \( \mathbb{A} \Rightarrow \mathbb{B} \). Hence, it can be proven by recursively decomposing the context and invoking the general theorem on the two parts. This insight is essential to scale up our approach from a two-function system to arbitrarily complex systems.

### 4.5 Local Harmlessness

The fourth condition requires the harmlessness of the extended aspect environment \( \mathit{env} \) with respect to a single function seen in isolation. In our recursively decomposed example, this means we can study the impact of \( \mathit{env} \) on \( f \) and \( g \) individually.

We do not go into detail here, but devote Section 5 to adapting the techniques of MRI for proving this condition in our setting. These techniques involve both regular proofs based on equational reasoning over the actual implementations of function and advice, as well as the more lightweight parametricity-based techniques that only need to consider the types.

### 5. A Simple Monadic AOP Model

In order to illustrate concrete applications of compositional reasoning about aspect interference, we now describe a simple monomorphic monadic model of pointcut/advice AOP in Haskell. The model is a simplification of the monadic embedding of aspects of Tabareau et al. [26]. The main differences are that this model:

1. does not support polymorphic aspects; only functions of type \( \mathbb{C} \Rightarrow \mathbb{A} \Rightarrow \mathbb{B} \), for some monad \( \mathbb{M} \), are open to advice.
2. only has pure pointcuts, i.e. pointcuts that cannot use monadic effects.
3. uses an abstract syntax tree of computations that expose function applications as join points and turn it into a monad transformer.
4. does not support dynamic aspect deployment; \( \mathit{A} \) computations are evaluated under a fixed aspect environment.
5. uses a more general model of tagged open weaving to specify quantification.

Section 5 discusses potential extensions to the model.

---

**Instance** Monad \( m \Rightarrow \) Monad \( (\mathit{A} \Rightarrow m) \) where

\[
\begin{align*}
\mathit{return} & = \mathit{A} \Rightarrow \mathit{return} \circ \mathit{Return} \\
\mathit{m} \gg f & = \mathit{A} \Rightarrow (\mathit{unA} \Rightarrow m) \gg \lambda x \Rightarrow \mathit{case r of} \\
& \qquad \mathit{Return} x \Rightarrow \mathit{unA} \Rightarrow \mathit{f} x \\
\mathit{OpenApp} t q x k & \Rightarrow \mathit{return} (\mathit{OpenApp} t q x (\lambda y \Rightarrow k y \gg f))
\end{align*}
\]

**Instance** MonadTrans \( \mathit{A} \Rightarrow m \) where

\[
\mathit{lift} ma = \mathit{A} \Rightarrow (\mathit{ma} \gg \lambda \Rightarrow (\mathit{return} \circ \mathit{Return}) a)
\]

**Figure 3.** \( \mathit{A} \Rightarrow m \) and \( \mathit{MonadTrans} \) type classes.

### 5.1 An Embedding of Open Applications

We implement \( \mathit{A} \Rightarrow m \) as a monad transformer that captures open function applications in a syntactic form. The interpreter function \( \mathit{runA} \Rightarrow m \) interprets the open applications by weaving them with the aspect environment.

#### Join point model

Join points represent open function application.

In order not to deal with function equality or type comparisons as in [26], we rely on tagged applications: pointcuts match join points based on tag equality (\( \mathit{pcTag} \)). Here, tags are just integers:

- **Tag** = \( \mathbb{Int} \)
- **data** \( \mathit{Jp} \) = \( \mathbb{Monad} \) \( m \) \( a \) \( b \) = \( \mathit{Jp} \) \( \mathit{Tag} \)

- **data** \( \mathit{Tag} \) = \( \mathbb{Int} \)

- **pcTag** \( t \) (\( \mathit{Jp} \) \( t' \)) = \( t \equiv t' \)

Note that in this simple instantiation of monadic AOP, join points only embed the tag of an open application, and neither the applied function nor the argument.

#### Defining the monad transformer

The \( \mathit{A} \Rightarrow m \) transformer extends a given monad \( m \) with the ability to expose some open function applications. A computation \( \mathit{A} \Rightarrow m \) \( a \) is denoted by an alternating sequence of computations in the monad \( m \) and exposed open function applications starting with the former.

- **data** \( \mathit{A} \Rightarrow m \) \( a \) = \( \mathit{A} \Rightarrow m \) \( \{ \mathit{unA} \Rightarrow m \) (\( \mathit{ResultA} \Rightarrow m \) \( a \) )\)

- **data** \( \mathit{ResultA} \Rightarrow m \) \( a \)
  
  - \( = \mathit{Return} a \)
  - \| \( \mathit{OpenApp} \) \( \mathit{Tag} \) -- tag
    - \( \mathit{Int} \Rightarrow \mathit{A} \Rightarrow m \) \( \mathit{Int} \) -- function
    - \( \mathit{Int} \) -- argument
    - \( \mathit{Int} \Rightarrow \mathit{A} \Rightarrow m \) \( a \) -- continuation

The \( \mathit{ResultA} \Rightarrow m \) value indicates what comes next after an \( m \) computation. Either the computation is done, which is denoted by the \( \mathit{Return} \) constructor, or an open function application comes next, denoted by the \( \mathit{OpenApp} \) constructor. In particular, \( \mathit{OpenApp} \) \( t \) \( q \) \( x \) \( k \) denotes the open application of \( g \) to \( x \) with tag \( t \), followed by the continuation \( k \) that proceeds the computation with the result of the open application. Figure 4 shows the instances for the \( \mathit{Monad} \) and \( \mathit{MonadTrans} \) type classes. Observe that for open applications, \( \gg \) extends the corresponding continuation \( k \) with operation \( f \).

#### Open Applications

Function \( \mathit{openApp} \) creates the denotation of tagged open applications:

\[
\mathit{openApp} t f x = \mathit{A} \Rightarrow (\mathit{return} (\mathit{OpenApp} t f x \mathit{return}))
\]

Because \( \mathit{return} \) is the left and right identity of \( \gg \), we use it as the continuation that proceeds with the result of the open application. Hence, in isolation, open applications provide a semantics-preserving connection point for composition through \( \gg \). Using \( \mathit{openApp} \), \( \mathit{A} \Rightarrow m \) can be declared as an instance of the \( \mathit{A} \Rightarrow m \) type class:

---

4 Our \( \mathit{A} \Rightarrow m \) implementation is a close cousin of a free monad.
instance Monad m ⇒ kM (aT m) where
f #! x = openApp t f x

5.2 Running aT Computations

We define the runaT interpreter function which evaluates an aT computation:
runaT aenv m = unkT m ≃ go where
go (Return r) = return r
go (OpenApp t x f x k) =
    unkT (weave f aenv (Jp t)) ≃
    λwoven_f → woven_f x ≃ k) ≃ go

This function is defined in terms of the locally-defined go function. In case of Return r values, it simply unwraps and returns value r. When it encounters an open application, it creates a join point Jp t and uses the weave to apply the matching aspects deployed in aenv. This yields the woven_f function which is applied to argument x. The result of the application is given to continuation k, whose resulting computation is evaluated recursively using go.

5.3 Aspect Weaving

The weave is defined recursively on the aspect environment as follows.
weave :: Monad m ⇒ (Int → m Int) → AEnv m → 
Jp m Int Int → m (Int → m Int)
weave f [] = return f
weave f ((pc, adv) : asps) jp =
weave (if pc jp then (adv f) else f) asps jps

For each aspect it applies the pointcut to the join point. Then it continues weaving on the rest of the aspect environment using either (adv f) if the pointcut matches, or f otherwise.

5.4 Properties of aT

To exploit the general result of the previous section, we need to establish that aT is a proper aspectual monad transformer that satisfies the necessary properties described in Section 3.

Lemma 1 (Monad laws for aT). aT fulfills the monad transformer laws. In addition, for any monad m, aT m fulfills the monad laws.

Lemma 2 (runaT monad morphism). For any aspect environment aenv, runaT aenv is a monad morphism. Furthermore, it is also a left inverse of lift.

The proofs proceed by straightforward co-induction and equational reasoning on the shape of the monadic composition, and are available in the companion technical report [13]. Crucially, the proofs rely on the monad and monad transformer laws for aT.

Given the importance of the compositionality of weaving (which corresponds to the second law of monad morphisms), we show its proof in Figure 4. This law is fundamental for the formalization of Section 3 as well as for the theorem of the following section. The proof consists of folding and unfolding the definitions of runaT, its internally-defined function go, and the ≃ operation of aT; it also uses the monad laws on m, and the identity unkT ◦ aT ≃ aT ◦ unkT ≃ id. A crucial step is the application of the co-induction hypothesis, which allows us to start folding the definitions until its final form.

6 Local Harmlessness

In Section 3, we have shown how the first three conditions of Theorem 1 can be met. This section develops local harmlessness results using the monadic AOP model of Section 5. We now discuss how local harmlessness of the updated aspect environment aenv' with respect to the initial environment aenv can be established in this setting. Concretely, we must prove that:

Figure 4. Proof of the second monad morphism law for runaT.
We observe that the problem of reasoning about aspect interference for an isolated function woven by aspects is directly analogous to the work of MRI in the model of mixins. Therefore, we can benefit from the established results of MRI in at least two ways:

- Translate AOP programs into the setting of MRI; establish the required program equivalence in this setting, and interpret this result back into the AOP model. This approach allows us to reuse directly all the theorems proven in the MRI model.
- Lift the reasoning techniques developed in MRI to the AOP setting, to establish similar harmlessness results. This path is potentially more general and avoids a translation to MRI, but it does entail the need to re-establish all theorems proven in the MRI model in the AOP model.

Here, we adopt the first approach, leaving the second one as a possible line of future work.

**AOP-MRI translation** We present a commutative correspondence diagram that gives a high-level overview of the chosen technique. In this diagram, the local harmlessness condition of Theorem 1 is represented by path (d). Instead of proving this directly, the goal is to obtain (d) by the composition of paths (a), (b) and (c).

\[
\begin{align*}
\text{run}_\beta \cdot \text{env} \circ f (\#^*) & \equiv \pi \circ \text{run}_\beta \cdot \text{env}' \circ f (\#^*) \\
\text{We adopt the model of inheritance in Figure 1 because it is sim}
\end{align*}
\]

Finally, the inheritance operator \( \text{mixin} \) defines component composition. Composition is associative and commutative, and we have the following properties:

\[
\begin{align*}
\text{run}_\beta \cdot \text{env} \circ f (\#^*) & \equiv \pi \circ \text{run}_\beta \cdot \text{env}' \circ f (\#^*)
\end{align*}
\]

To create a component \( \oplus \) instantiates \( \text{super} \) references for every extended component, and \( \text{new} \) instantiates the self-reference \( \text{this} \).

MRI formally captures the notion of harmlessness that has been used in this paper. Given a mixin \( \text{mix} \) and base component \( \text{bse} \), then \( \text{mix} \) is harmless if:

\[
\pi (\text{new} (\text{mix} \oplus \text{bse})) \equiv \text{run}_\beta \circ \text{new} \circ \text{bse}
\]

for some projection \( \pi \). Here \( \text{run}_\beta \) is the projection of the identity monad, which has no computational effect.

MRI provides two harmless mixin theorems [22]. Using these theorems it is formally proven that logging is harmless for any arbitrary function. It is also proven that memoization is harmless when applied to the Fibonacci function. In the examples of next section we detail the specific techniques used in these proofs.

### 6.2 Connecting MRI to AOP

There is a direct connection between an advice and a mixin, as witnessed by the types of these entities: both type synonyms \( \text{Advice} \) and \( \text{Open} \) denote the same type \( (a \rightarrow m b) \rightarrow (a \rightarrow m b) \). This reveals that any MRI mixin can be used as an advice. However the converse is not generally true if an advice performs open applications. For instance an aspect could trigger infinite recursion by matching join points emitted on its own advice. However, if an advice uses a type class constraint that does not entail \( \text{Monad} \) (which means that it cannot perform any open application), this cannot happen.

To connect a base function \( \text{f}_{\text{AOP}} \) with an open recursive equivalent function, we need a stronger constraint. Namely, we ask that \( \text{f}_{\text{AOP}} \) is equivalent to the fixpoint of an open recursion \( \text{f}_{\text{MRI}} \) (that does not make use of open application) in the following way:

\[
\text{f}_{\text{AOP}} \equiv \text{fix} (\lambda f \rightarrow \text{f}_{\text{MRI}} (f \#^*))
\]

Putting these together, we can state a general theorem that relates MRI to AOP and eases considerably the establishment of steps (a) and (c) of the correspondence diagram.

**Definition 2** (AOP-agnostic function). A function

\[
f :: \forall m.C \; m \Rightarrow (A \rightarrow m B) \rightarrow (A \rightarrow m B)
\]

is AOP-agnostic if \( C \) is a type class constraint that entails \( \text{Monad} \) but not \( \text{Monad} \). This means that the function does not emit join points.

**Theorem 2.** Given AOP-agnostic functions

\[
\text{f}_{\text{MRI}} :: \forall m.C \; m \Rightarrow (A \rightarrow m B) \rightarrow (A \rightarrow m B)
\]

and given an aspect environment

\[
\text{env} = [(pcTag \; t_1, \text{adv}_1), \ldots, (pcTag \; t_n, \text{adv}_n)]
\]

we have that

\[
\text{run}_\beta \cdot \text{env} \circ \text{f}_{\text{AOP}} \#^* \equiv \text{new} (\text{adv}_1 \oplus \ldots \oplus \text{adv}_n \oplus \text{f}_{\text{MRI}})
\]

where \( \text{f}_{\text{AOP}} \) is such that

\[
\text{f}_{\text{AOP}} \equiv \text{fix} (\lambda f \rightarrow \text{f}_{\text{MRI}} (f \#^*))
\]
and \([adv_1',...,adv_v']\) = \([adv_1, | pcTag t_i (Jp t) \equiv True]\) is the list of all advices in \(aenv\) whose \(pcTag t_i\) pointcut matches \#^i\) applications.

Proof. The proof proceeds by equational reasoning and coinduction:
\[
\begin{align*}
& run_{AOT} aenv \circ f_{AOP} \#^i \\
& \equiv \text{[-definition of } f_{AOP} - ] \\
& run_{AOT} aenv \circ fix (\lambda f \rightarrow f_{MRI} (f \#^i)) \#^i \\
& \equiv \text{[-unfolding of the fixpoint -] } \\
& run_{AOT} aenv \circ f_{MRI} (f_{AOP} \#^i) \#^i \\
& \equiv \text{[-compositionality of weaving -] } \\
& run_{AOT} aenv \circ f_{MRI} (run_{AOT} aenv \circ f_{AOP} \#^i) \#^i \\
& \equiv \text{[-co-induction hypothesis -] } \\
& run_{AOT} aenv \circ f_{MRI} (new (adv_k \oplus ... \oplus adv_1 \oplus f_{MRI})) \#^i \\
& \equiv \text{[-weaving -] } \\
& adv_k \circ ... \circ adv_1 \\
& \circ f_{MRI} (new (adv_k \oplus ... \oplus adv_1 \oplus f_{MRI})) \\
& \equiv \text{[-definition of } \oplus - ] \\
& adv_k \oplus ... \oplus adv_1 \\
& \oplus f_{MRI} (new (adv_k \oplus ... \oplus adv_1 \oplus f_{MRI})) \\
& \equiv \text{[-folding the new fixpoint -] } \\
& new (adv_k \oplus ... \oplus adv_1 \oplus f_{MRI})
\end{align*}
\]

The same proof can be made for any model of AOP as described in Section\(5\), one just has to accommodate the proof according to the concrete way (in particular the ordering) in which aspects are woven.

Example that cannot use Theorem\(2\). We now present an aspect-oriented implementation of the Fibonacci function that cannot be translated into MRI by Theorem\(2\). In this example, taken from \([1]\), the function is split into a base case that simply returns 1, and an advice that handles the recursive calls. The composed function \(plainFib\) combines the base program and advice to provide the regular unoptimized version of Fibonacci.

\[
\begin{align*}
plainFib\ n = \ & run_{AOT} [pcTag t, fibAdv] (fibBase \ #^i\ n) \\
\fibBase\_run = \ & return\ 1 \\
\fibAdv\_run = \ & \text{if } (n \leq 2) \text{ then } \text{return } n \\
\ & \text{else do } f_1 \leftarrow \fibBase\ #^i\ (n - 1) \\
\ & \ f_2 \leftarrow \fibBase\ #^i\ (n - 2) \\
\ & \text{return } (f_1 + f_2)
\end{align*}
\]

We cannot apply Theorem\(2\) because of the type of \(fibAdv\). Since \(fibAdv\) performs open applications of \(fibBase\), its type necessarily contains a type class constraint that entails \(\text{has}\); thus violating the initial condition of Theorem\(2\). In fact, it does not seem possible to define \(fibAdv\) using mixins, because the full aspect environment is woven upon each open application, whereas mixins can only execute the next component using \(\text{super}\).

Applying the theorem. We now present an example, using the Fibonacci function as a concrete value for \(f\), on how to follow the steps of the correspondence diagram to prove the harmlessness of the logging and memoization advices of Figure\(2\). We consider the starting environment \(aenv\) to be empty; and illustrate the case of adding each aspect individually. Figure\(3\) presents the Fibonacci function in the AOP and MRI models, along with their plain, logged and memoized versions.

6.3 Harmlessness of Logging

The local harmlessness of \(log\) applied to \(fib_{AOP}\) corresponds to the following lemma:

**Lemma 3.** \(plainFib_{AOP} \equiv \pi_W \circ logFib_{AOP}\)

Proof. Following the commutative correspondence diagram, by composition of Lemmas\(4,5\), and \(6\).

**Step (a)** We must translate \(plainFib_{AOP}\) into MRI. We choose \(plainFib_{MRI}\) as its translation, hence we must prove the following:

**Lemma 4.** \(plainFib_{AOP} \equiv plainFib_{MRI}\)

The proof is direct consequence of Theorem\(2\) using the equality \(fib_{AOP} \equiv \fix (\lambda f \rightarrow fib_{MRI} (f \#^i))\) that can be proven by equational reasoning and induction on the integer argument.

**Step (b)** For the second step we need to prove:

**Lemma 5.** \(plainFib_{MRI} \equiv \pi_W \circ logFib_{MRI}\)

Here we benefit from the results of MRI. In MRI the local harmlessness of logging is proven for any arbitrary component, like \(fib_{MRI}\); hence it holds for this particular case \([22]\).

In fact the general harmlessness of logging is an application of the harmlessness mixin theorem of MRI \([22]\). This theorem is proven using: (i) parametricity to ensure that the base component cannot access the effects used by the mixin; (ii) a mixin combinator to guarantee that \(\text{super}\) is called exactly once, and that the arguments and return values are not modified; and (iii) the algebraic laws for monadic effects. Consequently, any mixin that satisfies this theorem is also harmless for functions that are AOP-agnostic (Definition\(3\)).

**Step (c)** Finally, we prove the equivalence between \(logFib_{AOP}\) and \(logFib_{MRI}\):

**Lemma 6.** \(\pi_W \circ logFib_{AOP} \equiv \pi_W \circ logFib_{MRI}\)

Again, this is a direct consequence of Theorem\(2\).

6.4 Harmlessness of Memoization

Proving the harmlessness of memoization involves the same steps as that of logging. In this case we greatly benefit from the established results of MRI, because proving step (b) is rather complex.

The issue is that, conversely to logging, memoization is not harmless in general; hence this property must be proven for each particular function. The main difficulty of such a proof is to show that the function maintains an invariant on the memoization table: namely, that the stored values actually correspond to the results of the original function. In \([22]\) this is proven for \(fib_{MRI}\), developing a long equational reasoning proof—the Coq proof assistant is used to manage the complexity of the proof.

It is in complex situations like this that the interest of following the steps of the AOP-MRI correspondence diagram is justified. In addition, we can benefit from new results about harmlessness of specific mixins.

7. Related Work

There is a large body of work on modular reasoning and interference. Here, we only discuss the most directly related work; an extensive and recent review of the area, which also covers reasoning techniques in functional, object-oriented, and feature-oriented programming can be found in \([22]\).

We have extensively related our work to the work on EffectiveAdvice \([21]\) and its successor, MRI \([22]\). The present work
Figure 5. Fibonacci function. Left: in the simple pointcut/advice model of Section 5. Right: in the MRI setting (taken from [22]).
reasoning under certain scenarios. A first approach is to use para-
metricity. For instance, Tabareau et al. [26] use parametricity to
define non-interfering pointcuts and advices, following the tech-
niques of MRI [22]. Additionally, Tabareau et al. provide protected
pointcuts as a mechanism to recover modular reasoning. However,
the approach is not yet formalized.

Because, ultimately, unrestricted quantification is incompatible
with modular reasoning, it is appealing to combine the coarse-
grained modular reasoning provided by Open Modules [1] with
our compositional reasoning techniques for reasoning about equa-

Finally, the model of AOP presented in this paper is very sim-
plified compared to that of [26]; hence an additional challenge is to
calce the expressiveness of the model while preserving the results
established in this paper.

Acknowledgments. This work was supported by the INRIA Asso-
ciated Team REAL and FONDECYT Project 1110051, and by the
Fund for Scientific Research - Flanders (FWO-Vlaanderen).

References

Black, editor, Proceedings of the 19th European Conference
on Object-Oriented Programming (ECOOP 2005), number 3586 in
Lecture Notes in Computer Science, pages 144–168, Glasgow, UK,

[2] Proceedings of the 9th ACM International Conference on Aspect-
Oriented Software Development (AOSD 2010), Rennes and Saint

cid contracts: Expressive specification and modular verification for
aspect-oriented interfaces. In Proceedings of the 10th ACM Interna-
tional Conference on Aspect-Oriented Software Development (AOSD

and flexible decoyling of aspects. ACM Transactions on Software


fects for more effective reasoning about aspects. In E. Ernst, editor,
Proceedings of the 21st European Conference on Object-oriented Pro-
gramming (ECOOP 2007), number 4609 in Lecture Notes in Com-
Springer-Verlag.

ACM SIGPLAN-SIGACT Symposium on Principles of Programming
Languages (POPL 2006), pages 383–396, Charleston, South Carolina,

polymorphic aspect-oriented functional programming language. ACM
Transactions on Programming Languages and Systems, 30(3):Article

[9] C. Eisenfeld and S. Katz. Specification and verification of event de-
tectors and responses. In Proceedings of the 12th annual interna-
tional conference on Aspect-oriented software development, AOSD
'13, pages 121–132, New York, NY, USA, 2013. ACM.


formation analysis of stateful aspects. In K. Liebeherr, editor, Pro-
ceedings of the 3rd ACM International Conference on Aspect-Oriented
Software Development (AOSD 2004), pages 141–150, Lancaster, UK,

scoping of aspects in higher-order languages. Science of Computer

reasoning about aspect interference – extended with supplementary
material. Technical Report TR/DCC-2013-8, Computer Science De-
partment, University of Chile, Oct. 2013.


Rashid and M. Aksit, editors, Transactions on Aspect-Oriented
Software Development I, volume 3880 of Lecture Notes in Computer

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Gris-
wohl. An overview of AspectJ. In J. L. Knudsen, editor, Proceedings of the 15th
European Conference on Object-Oriented Programming (ECOOP 2001), number 2072 in Lecture Notes in Computer Science,

[17] G. Kiczales and M. Mezini. Aspect-oriented programming and mod-
ular reasoning. In Proceedings of the 27th international conference on
Software engineering (ICSE 2005), pages 49–58, St. Louis, MO,
USA, 2005. ACM Press.

[18] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect ad-
dvice modularly. In Proceedings of the 12th ACM SIGSOFT Inter-
national Symposium on Foundations of Software Engineering (FSE-12),

interpreters. In Proceedings of the 22nd ACM Symposium on Prin-
ciples of Programming Languages (POPL 95), pages 333–343, San

[20] E. Moggi. Notions of computation and monads. Information and

disciplined advice with explicit effects. In AOSD 2010 [2], pages 109–120.

reasoning about interference in incremental programming. Journal of

[23] C. Prehofer. Semantic reasoning about feature composition via mul-
Veldhuizen, editors, Proceedings of the 5th ACM SIGPLAN/SIGSOFT
Conference on Generative Programming and Component Engineer-
Press.

typed events. In J. Vitek, editor, Proceedings of the 22nd European
Conference on Object-oriented Programming (ECOOP 2008), number 5142 in Lecture Notes in Computer Science, pages 155–179, Paphos,

analysis for aspect-oriented programs. In Proceedings of the 12th
ACM Symposium on Foundations of Software Engineering (FSE 12),

[26] N. Tabareau, I. Figueroa, and É. Tanter. A typed monadic embedding
of aspects. In J. Kinzle, editor, Proceedings of the 12th Internation-
al Conference on Aspect-Oriented Software Development (AOSD 2013),


the 19th ACM Symposium on Principles of Programming Languages
ACM Press.