Compositional Reasoning About Aspect Interference

Ismael Figueroa, Tom Schrijvers, Nicolas Tabareau, Éric Tanter

1PLEIADE Lab, Computer Science Dept (DCC)—University of Chile, Chile
2Dept. of Applied Mathematics and Computer Science—Ghent University, Belgium
3ASCOLA Team—INRIA, France

ifiguero@dcc.uchile.cl, tom.schrijvers@ugent.be, nicolas.tabareau@inria.fr, etanter@dcc.uchile.cl

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 harmless 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 [23]. A wide range of techniques have been studied, such as program analyses [25], type-and-effect systems [6, 7], model checking [15, 18] and equational reasoning [21, 22].

Oliveira et al. developed MRI, which stands for Modular Reasoning about Interference, a purely functional model of mental programming with effects [22]. 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 [7].

While originally formulated as “EffectiveAdvice” [21] 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 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 [17]. Recovering modular reasoning can be achieved by restricting quantification, for instance following the Open Modules approach [1]. 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 compositionality 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 F. 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...
the necessary background on monads and reasoning, and describes a general model of monadic AOP. Section 4 exposes the main theorem of compositional reasoning, discussing and illustrating each of the sufficient conditions. Section 5 presents a concrete minimal implementation of monadic AOP, which is used in Section 6 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 7 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 \( g \):

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

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/advice 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:

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

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 \( h \):

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

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:

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

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 \( f \), 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, 28], and are widely used in Haskell. A monad is defined by a type constructor $m$ and functions $\text{return}$ ($\gg=$ 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 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 $\gg=$ operator. The concrete definitions of $\text{return}$ and $\gg=$ 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 MonadTrans type class (Figure 1) defines an interface for monad transformers. The purpose of the 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 Monad class.

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

**Polymorphism in the monad stack** In addition to the particular monad transformers, the 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 $S_M$ and $W_M$ classes that abstract the state and writer effects. The get operation retrieves the current value, which can be updated using put. Similarly, the tell operation appends a value $w$ to its output. Note that the $S_T$ and $W_T$ 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 $\lambda x \rightarrow f \ x \equiv f$ (when $x$ is no free in $f$). Also, Haskell provides the do notation as syntactic sugar for $\gg=$, hence do $\{ x \leftarrow f ; g \ x \} \equiv f \gg= \lambda x \rightarrow g \ x$.

**Monad laws** Monad laws are crucial for equational reasoning in a monadic setting [28]. A proper monad is one that obeys the following three laws:

- $\text{return} x \gg= f \equiv f x$ -- left identity
- $p \gg= \text{return} \equiv p$ -- right identity
- $(p \gg= f) \gg= h \equiv p \gg= \lambda x \rightarrow (f \ x \gg= h)$ -- associativity

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 $\gg=$ expression. In the same way, monad transformers need to satisfy the following laws:

- $\text{lift} \circ \text{return} \equiv \text{return}$ -- identity preservation
- $\text{lift} (m \gg= f) \equiv \text{lift} m \gg= (\text{lift} \circ f)$ -- comp. preservation

Note that Haskell does not enforce that declared instances of the Monad or 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 MRI [22].

In this section, we define an aspectual computation monad transformer denoted $\text{AHT}$ in an abstract manner, by prescribing its interface and properties. The theorem of compositional reasoning in Section 4 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 [27]), by their type (e.g. AspectJ, AOHaskell [26]), using a mechanism to explicitly attach tags or types to join points (e.g. Ptolemy [24], JPs [4]), etc.

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

```
class Monad m where  -- State
  return :: a -> m a
  (>$>) :: m a -> (a -> m b) -> m b

class MonadTrans t where  -- Writer
  lift :: Monad m => m a -> t m a

  class Monad m => S m | m -> s where
    get :: m s
    put :: s -> m ()

  class (Monoid w, Monad m) => W m | m -> w where
    tell :: w -> m ()
```

Figure 1. Monads and monad transformer classes used in this paper.
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 \( a \rightarrow \mathcal{A}_T m b \) and returns a function of the same type whose application produces a join point (this effect is encapsulated in the \( \mathcal{A}_T \) monad transformer):

\[
# :: (a \rightarrow \mathcal{A}_T m b) \rightarrow (a \rightarrow \mathcal{A}_T m b)
\]

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 \( f \neq i \) 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 \( \text{proceed} \), is a function which represents the original computation given an aspect. The theorem considers an AOP following general theorem, which provides a framework for the 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. Compositional reasoning is achieved because the theorem splits the system into two parts, and open function \( f \neq i \) and a context \( c \), whose conditions are independent, can be proven separately, and can be reused in different compositions. Moreover, the system

### 3.3 Necessary Properties of \( \mathcal{A}_T \)

To be a correct model, the \( \mathcal{A}_T \) 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 \( \text{run}_{\mathcal{A}_T} aenv \) must be a monad morphism.

### 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. Compositional reasoning is achieved because the theorem splits the system into two parts, and open function \( f \neq i \) and a context \( c \), whose conditions are independent, can be proven separately, and can be reused in different compositions. Moreover, the system

---

\(^2\) If \( \mathcal{A}_T \) supports dynamic deployment of aspects (as in [26]), 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 \( \text{return} \equiv \text{lift} \circ \text{return} \).
The monad stack is a left-inverse of \( C \) where composition:

\[
\text{instance of } C
\]

Here is an instance of the theorem and thus explicitly invites this systematic (rather than just two parts) by repeated application of the theorem. The four conditions on \( C \) are some instance of the theorem and thus explicitly invites this systematic decomposition.

Theorem 1. Given an expression:

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

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:

\[
\text{system} \equiv c \ (f \ #)^1
\]

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):

\[
c :: \forall m. C \ m \Rightarrow (A' \rightarrow A_T \ m \ B') \rightarrow A \rightarrow A_T \ m \ B
\]

\[
f :: \forall m. C_f \ m \Rightarrow A' \rightarrow A_T \ m \ B'
\]

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

\[
aenv :: \forall m. D \ m \Rightarrow AEnv (A_T \ m)
\]

\[
aenv' :: \forall m. D \ m \Rightarrow AEnv (A_T (T \ m))
\]

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:

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

is a left-inverse of 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:

\[
\text{run} A_T \ aenv \ \text{system} \equiv \pi \ (\text{run} A_T \ aenv' \ \text{system})
\]

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

1. Compositional projection

\[
\forall env. \text{run} A_T \ env \ (c \ (f \ #)) \equiv \text{run} A_T \ env \ c \ (\text{lift} \circ \text{run} A_T \ env \circ (f \ #))
\]

2. Compositional projection

\[
\pi \circ \text{run} A_T \ aenv \circ c \ (\text{lift} \circ \text{lift} \circ \pi \circ \text{run} A_T \ aenv' \circ (f \ #)) \equiv \pi \circ \text{run} A_T \ aenv' \circ c \ (\text{lift} \circ \text{run} A_T \ aenv' \circ (f \ #))
\]

3. Contextual harmlessness

\[
\text{run} A_T \ aenv \circ c \circ (\lambda g \rightarrow \text{lift} \circ g) \equiv \pi \circ \text{run} A_T \ aenv' \circ c \circ (\lambda g \rightarrow \text{lift} \circ \text{lift} \circ g)
\]

4. Local harmlessness

\[
\pi \circ \text{run} A_T \ aenv' \circ (f \ #) \equiv \pi \circ \text{run} A_T \ aenv' \circ (f \ #)
\]

Proof. The proof proceeds by straightforward equation:

\[
\text{run} A_T \ aenv \ \text{system} \equiv \{ \text{-system decomposition -} \}
\]

\[
\text{run} A_T \ aenv \ (c \ (f \ #)) \equiv \{ \text{-compositional projection -} \}
\]

\[
\pi \ (\text{run} A_T \ aenv' \circ c \ (\text{lift} \circ \text{run} A_T \ aenv' \circ (f \ #)) \equiv \{ \text{-compositional projection -} \}
\]

\[
\pi \ (\text{run} A_T \ aenv' \circ c \ (\text{lift} \circ \text{run} A_T \ aenv' \circ (f \ #)) \equiv \{ \text{-compositional projection -} \}
\]

\[
\pi \ (\text{run} A_T \ aenv' \ (c \ (f \ #))) \equiv \{ \text{-system decomposition -} \}
\]

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 \( c_1 \ (f_1 \ #) \) where:

\[
c_1 = f
\]

\[
c_1 = \lambda f \ x \ y \rightarrow \text{do} \ r_1 \leftarrow f \ x \ r_2 \leftarrow g \ #^3 \ y \ \text{return} \ (r_1 + r_2)
\]

Here the context \( c_1 \) is just system abstracted over \( f \ # \). 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 \( \text{run} A_T \ env \) satisfies (recall Section 3.3). 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.

\[
\begin{align*}
\text{log} &:: \forall M \text{ String } m \Rightarrow \text{Advice} \ m \ a \ b \\
\text{log} \ \text{proceed} \ x &\ = \ \text{do} \\
&\ \	ext{tell} \ "\text{Entering function ...}"
\end{align*}
\]

\[
\begin{align*}
\text{memo} &:: (\text{Ord } a, \text{SMap } (\text{Map } a \ b) \ m) \Rightarrow \text{Advice} \ m \ a \ b \\
\text{memo} \ \text{proceed} \ x &\ = \ \text{do} \\
&\ \text{table} \ \leftarrow \ \text{get} \\
&\ \text{if} \ \text{member} \ x \ \text{table} \ \text{then} \ \text{return} \ (\text{table} \ ! \ x) \\
&\ \text{else} \ \text{do} \ y \ \leftarrow \ \text{proceed} \ x \\
&\ \text{table}' \ \leftarrow \ \text{get} \\
&\ \text{put} \ (\text{insert} \ x \ y \ \text{table}') \\
&\ \text{return} \ y
\end{align*}
\]
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 \( m \Rightarrow f_1 \Rightarrow \ldots \Rightarrow f_n \).

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:

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

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 \( \#^i \) 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:

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

\[ f :: \forall m. (C_j m, \lambda M m) \Rightarrow A' \rightarrow m \cdot B' \]

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 \( \pi \cdot \pi \) 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:

\[ \pi_S 0 \ (\text{get} \Rightarrow \lambda x \rightarrow \text{put} \ (x+1) \Rightarrow \text{get}) \equiv \text{return} \ 1 \]

\[ \pi_S 0 \ (\text{get} \Rightarrow \lambda x \rightarrow \text{put} \ (x+1)) \Rightarrow \pi_S 0 \ \text{get} \equiv \text{return} \ 0 \]

This explains why we must be careful when adding the memo advice of Figure 2, which has a memo table as its state, to our running example. If the pointcut of the 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 \( \text{run}_{\lambda T} \cdot \text{aenv} \circ (f \ #^i) \) 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:

\[ \exists h, \text{lift} \circ h \equiv \text{run}_{\lambda T} \cdot \text{aenv} \circ (f \ #^i) \]

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

\[ \text{lift} \circ \text{lift} \circ \pi \circ \text{run}_{\lambda T} \cdot \text{aenv}' \circ (f \ #^i) \]

\[ \equiv \{-\text{assumption} \} \]

\[ \text{lift} \circ \text{lift} \circ \pi \circ \text{lift} \circ h \]

\[ \equiv \{-\pi \text{ is left inverse of lift} \} \]

\[ \text{lift} \circ \text{lift} \circ h \]

\[ \equiv \{-\text{assumption} \} \]

\[ \text{lift} \circ \text{run}_{\lambda T} \cdot \text{aenv} \circ (f \ #^i) \]

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 \( \exists y \) 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 \( \text{aenv} \) and \( \text{aenv}' \) are indistinguishable. There are various ways in which \( \text{aenv} \) and \( \text{aenv}' \) can be related for 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 c₁, namely c₂ (lift o h), where h :: C m ⇒ A' → m. B' is universally quantified. This form is smaller than the original system because it features fewer open applications; h's type is constrained to not feature any. The resulting system has the form:

\[ system' = \lambda x \rightarrow do \ r₁ \leftarrow lift (h x) \]
\[ \quad r₂ \leftarrow g \#^{c₂} y \]
\[ \quad return (r₁ + r₂) \]

which can be decomposed as \( system' = c₂ (f₂ \#^{c₂}) \):

\[ f₂ = g \]
\[ c₂ = \lambda g \rightarrow do \ r₁ \leftarrow lift (h x) \]
\[ \quad r₂ \leftarrow g \]
\[ \quad return (r₁ + r₂) \]

Here we can consider the harmlessness of the extended environment \( aenv' \) separately for \( g \) and \( c₂ \). Note that since \( c₂ \) does not contain any more open applications, contextual harmlessness is trivially established for it.

4.5 Local Harmlessness

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

We do not go into detail here, but devote Section 6 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 \( Int \rightarrow m \ Int \), for some monad \( m \), 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; \( kT \) computations are evaluated under a fixed aspect environment.
5. uses a more general model of tagged open weaving to specify quantification.

Section 8 discusses potential extensions to the model.

\begin{align*}
\text{instance } Monad \Rightarrow Monad (kT \Rightarrow m) \text{ where } \\
\quad \text{return } = kT \circ \text{return } \circ \text{Return} \\
\quad m \gg f = kT (\text{un}kT \Rightarrow m \gg \lambda r \rightarrow \text{case } r \text{ of} \\
\quad \quad \begin{cases} 
\quad \quad \text{Return } x \rightarrow \text{un}kT (f x) \\
\quad \quad \text{OpenApp } t \rightarrow \text{kg } x \rightarrow \text{return } (\text{OpenApp } t \rightarrow q x \rightarrow (\lambda y \rightarrow k y \gg f)) 
\end{cases}
\end{align*}

\begin{align*}
\text{instance } MonadTrans kT \text{ where } \\
\quad \text{lift } ma = kT (\text{ma } \gg \lambda a \rightarrow (\text{return } \circ \text{Return} ) a)
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{A simple monadic AOP model.}
\end{figure}

5.1 An Embedding of Open Applications

We implement \( kT \) as a monad transformer that captures open function applications in a syntactic form.\(^4\) The interpreter function \( \text{run}kT \) 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 \( (pcTag) \). Here, tags are just integers:

\begin{align*}
\text{type } \text{Tag} = \text{Int} \\
\text{data} \ Jp \ m a b = \text{Jp } \text{Tag} \\
\text{pcTag } t \ (\text{Jp } t') = t \equiv t'
\end{align*}

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

\begin{align*}
\text{data} \ kT \Rightarrow m a = \text{ks}x (\text{un}kT : m (\text{Result}kT \Rightarrow m a)) \\
\text{data} \ \text{Result}kT \Rightarrow m a \\
\quad = \text{Return } a \\
\quad \mid \text{OpenApp } \text{Tag} \quad \text{-- tag} \\
\quad \quad (\text{Int } \rightarrow kT \Rightarrow m \text{ Int}) \quad \text{-- function} \\
\quad \quad \text{Int} \quad \text{-- argument} \\
\quad \quad (\text{Int } \rightarrow kT \Rightarrow m a) \quad \text{-- continuation}
\end{align*}

The \( \text{Result}kT \Rightarrow m \) value indicates what comes next after an \( m \) computation. Either the computation is done, which is denoted by the return constructor, or an open function application comes next, denoted by the OpenApp constructor. In particular, OpenApp \( t g x k \) 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 3 shows the instances for the \( Monad \) and \( MonadTrans \) type classes. Observe that for open applications, \( \gg \) extends the corresponding continuation \( k \) with operation \( f \).

Open Applications Function OpenApp creates the denotation of tagged open applications:

\[ \text{openApp } t f x = kT (\text{return } (\text{OpenApp } t f x \text{ return})) \]

Because \text{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 \text{openApp}, \( kT \) can be declared as an instance of the \( kM \) type class:

\(^4\) Our \( kT \) implementation is a close cousin of a free monad.
instance Monad m ⇒ κₜ (Aₜ m) where
f # t x = openApp t f x

5.2 Running κₜ Computations
We define the run κₜ interpreter function which evaluates an Aₜ computation:
run κₜ aenv m = run κₜ m ≈ go where
go (Return r) = return r
  go (OpenApp t x f k) =
un κₜ (woven f aenv (Jp t) ≈ woken 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 f 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 pe jp then (adv f) else f) asps jp

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

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

Lemma 2 (run κₜ monad morphism). For any aspect environment aenv, run κₜ 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 κₜ.

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 4, as well as for the theorem of the following section. The proof consists of folding and unfolding the definitions of run κₜ, its internally-defined function go, and the ≈ operation of κₜ; it also uses the monad laws on m, and the identity un κₜ o κₜ ≡ κₜ o un κₜ ≡ 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 4, 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'

Figure 4. Proof of the second monad morphism law for run κₜ.

with respect to the initial environment aenv can be established in this setting. Concretely, we must prove that:


\[
\text{run} \lambda f \cdot aenv \circ (f \#') \equiv \pi \circ \text{run} \lambda f \cdot aenv' \circ (f \#')
\]

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.\(^5\)

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).

\[
f_{MRI} + \text{mix} \quad \overset{(b)}{\equiv} \quad \pi(f_{MRI} + \text{mix}')
\]

\[
f_{AOP} + aenv \quad \overset{(a)}{\equiv} \quad \pi(f_{AOP} + aenv')
\]

Starting from an AOP system composed by function \( f_{AOP} \) and aspect environment \( aenv \), step (a) involves finding a function \( f_{MRI} \) and a mixin \( \text{mix} \), such that their composition is equivalent to this initial system. In the same way, step (c) requires to find a mixin \( \text{mix}' \) equivalent to aspect environment \( aenv' \). Given this, we can reuse the reasoning techniques and established results of MRI to determine the equivalence of step (b) between \( f_{MRI} \) composed with \( \text{mix} \) and the projection of \( f_{MRI} \) composed with \( \text{mix}' \).

A drawback of this approach is that it is not known how to perform steps (a) and (c) in a general manner, because there are AOP programs that cannot be expressed using mixins, as illustrated later. Still, we can prove that a connection exists for a wide family of functions and aspect environments (Theorem 2 below).

We now briefly summarize the MRI model and prove a theorem connecting MRI to AOP. Then, using the Fibonacci function as a concrete example, we show how to prove that the logging and memoization aspects from Figure 2 are locally harmless.

### 6.1 Background: the MRI Framework

MRI models inheritance by the composition of mixins through open recursion. This inheritance model is defined as [22]:

**type** \( \text{Open} \; s = s \to s \)

\[
\text{new} : \; \text{Open} \; s \to s
\]

\[
\text{new} \; a = \text{fix} (\lambda f \to a \; f)
\]

\[
(\oplus) : \; \text{Open} \; s \to \text{Open} \; s \to \text{Open} \; s
\]

\[
\text{a}_1 \oplus \text{a}_2 = \lambda \text{super} \to \text{a}_1 (\text{a}_2 \; \text{super})
\]

\( ^5 \) Tabareau et al. [26] adapt the use of parametricity to enforce non-interference of pointcuts, advice and base programs to the AOP model, but formal results have not been established.

The type \( \text{Open} \; s \) represents an open component of type \( s \). \( \text{new} \) is a fixpoint combinator that closes, or instantiates, an open component that is potentially extended. Finally, the \( \circ \) operator defines component composition. The following diagram (taken from [22]) illustrates the inheritance model:

\[
p = \text{new} \; (a_1 \oplus a_2 \oplus \ldots \oplus a_n \oplus \text{base})
\]

\( \text{super} \) represents an open component of type \( \text{super} \). For an isolated function woven by aspects is directly analogous to the finding of \( \text{mix} \) and a mixin \( \text{mix}' \), does not entail \( \lambda \text{bse} \) (which means that it cannot perform any open application), this cannot happen.

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

\[
f_{AOP} \equiv \text{fix} (\lambda f \to f_{MRI} (f \#'))
\]

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 \( f : \exists m. C \; m \Rightarrow (A \to m \; B) \to (A \to m \; B) \) is AOP-agnostic iff \( C \) is a type class constraint that entails \( \text{Monad} \) but not \( \lambda \text{bse} \). This means that the function does not emit join points.

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

\[
f_{MRI} :: \exists m. C \; m \Rightarrow (A \to m \; B) \to (A \to m \; B)
\]

\[
ad_{\text{adv}} :: \exists m. D \; m \Rightarrow (A \to m \; B) \to (A \to m \; B)
\]

and given an aspect environment

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

we have that

\[
\text{run} \lambda f \cdot aenv \circ f_{AOP} \#' \equiv \text{new} \; (\text{adv}_1' \oplus \ldots \oplus \text{adv}_n' \oplus f_{MRI})
\]

where \( f_{AOP} \) is such that

\[
f_{AOP} \equiv \text{fix} (\lambda f \to f_{MRI} (f \#'))
\]
and \([\text{adv}^1, \ldots, \text{adv}^k] = [\text{adv}^1, \mid \text{pcTag } t_i (Jp \ t) \equiv \text{True}]\) is the list of all advices in \(\text{aenv}\) whose \(\text{pcTag}\) \(t_i\) pointcut matches \#\(i\) applications.

**Proof:** The proof proceeds by equational reasoning and coinduction:

\[
\begin{align*}
\text{run}_{\text{AOP}} \text{aenv} \circ f_{\text{AOP}} & \equiv [-\text{definition of } f_{\text{AOP}} -] \\
\text{run}_{\text{AOP}} \text{aenv} \circ \text{fix} (\lambda f \to f_{\text{MRI}} (f \ #^i)) & \equiv [-\text{unfolding of the fixpoint -}] \\
\text{run}_{\text{AOP}} \text{aenv} \circ f_{\text{MRI}} (f_{\text{AOP}} \ #^i) & \equiv [-\text{compositionality of weaving -}] \\
\text{run}_{\text{AOP}} \text{aenv} \circ f_{\text{MRI}} (\text{run}_{\text{AOP}} \text{aenv} \circ f_{\text{AOP}} \ #^i) & \equiv [-\text{co-induction hypothesis -}] \\
\text{run}_{\text{AOP}} \text{aenv} \circ f_{\text{MRI}} (\text{new } (\text{adv}^k \oplus \ldots \oplus \text{adv}^1 \oplus f_{\text{MRI}})) & \equiv [-\text{-weakening -}] \\
\text{adv}^k \circ \ldots \circ \text{adv}^1 & \circ f_{\text{MRI}} (\text{new } (\text{adv}^k \oplus \ldots \oplus \text{adv}^1 \oplus f_{\text{MRI}})) \\
& \equiv [-\text{-definition of } \oplus -] \\
\text{adv}^k \circ \ldots \circ \text{adv}^1 & \oplus f_{\text{MRI}} (\text{new } (\text{adv}^k \oplus \ldots \oplus \text{adv}^1 \oplus f_{\text{MRI}})) \\
& \equiv [-\text{-co-induction hypothesis -}] \\
\text{new } (\text{adv}^k \oplus \ldots \oplus \text{adv}^1 \oplus f_{\text{MRI}}) & \equiv [-\text{-folding the new fixpoint -}] \\
\end{align*}
\]

\(\Box\)

The same proof can be made for any model of AOP as described in Section 3; 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 \(\text{plainFib}\) combines the base program and advice to provide the regular unoptimized version of Fibonacci.

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

We cannot apply Theorem 2 because of the type of \(\text{fibAdv}\). Since \(\text{fibAdv}\) performs open applications of \(\text{fibBase}\), its type necessarily contains a type class constraint that entails \(\text{adv}\); thus violating the initial condition of Theorem 2. In fact, it does not seem possible to define \(\text{fibAdv}\) 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 \(\text{aenv}\) to be empty; and illustrate the case of adding each aspect individually. Figure 5 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 \(\text{fib}_{\text{AOP}}\) corresponds to the following lemma:

**Lemma 3.** \(\text{plainFib}_{\text{AOP}} \equiv \pi_W \circ \text{logFib}_{\text{AOP}}\)

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

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

**Lemma 4.** \(\text{plainFib}_{\text{AOP}} \equiv \text{plainFib}_{\text{MRI}}\)

The proof is direct consequence of Theorem 2, using the equality

\[
\text{fib}_{\text{AOP}} \equiv \text{fix} (\lambda f \to \text{fib}_{\text{MRI}} (f \ #^i))
\]

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.** \(\text{plainFib}_{\text{MRI}} \equiv \pi_W \circ \text{logFib}_{\text{MRI}}\)

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

In fact the general harmlessness of logging is an application of the **harmless 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 2).

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

**Lemma 6.** \(\pi_W \circ \text{logFib}_{\text{AOP}} \equiv \pi_W \circ \text{logFib}_{\text{MRI}}\)

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 \(\text{fib}_{\text{MRI}}\), 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
AOP is future work. Scaling up this work to the more complete model of AOP was motivated by the desire to bring the reasoning power of MRI to aspect-oriented programming with quantification. The monadic embedding of aspects in Haskell developed recently [26] is a practical programming system that extends EffectiveAdvice with quantification, but it does not describe how to do formal reasoning. Compared to the simple monadic AOP system presented in this paper, it supports polymorphic aspects while preserving type soundness thanks to anti-unification, and supports dynamic deployment of aspects. Scaling up this work to that more complete model of AOP is future work.

Kiczales and Mezini argue that strictly modular reasoning about programs written in the presence of quantification is not feasible, and introduce a notion of aspect-aware interfaces that rely on a global reasoning step to infer precise dependencies [17]. Aspect-aware interfaces have not been used to perform formal reasoning.

Aldrich introduced the concept of Open Modules [1] to allow modular reasoning on aspects. Technically, modularity is obtained by using a special module scaling operator that hides internal join points from external advices. While formally establishing modular reasoning results, the approach has strong limitations when it comes to dealing with realistic aspects because the model does not support effects. Also, proving the equivalence of two modules relies on "global" reasoning with unrestricted quantification; our framework could be used to enhance that part of the reasoning.

There is a vast literature on interference analysis in the setting of AOP. Starting from his pioneering work on superposition for distributed systems [14], Katz has later refined it to give a classification of aspects [15]. He distinguishes between superpositional (that amounts to harmlessness), regulative superposition (that can modify which actions occur, but cannot change the computation performed by an individual action) and invasive superposition (that can change anything).

Inspired by these categories, Diko Diko et al. [10] have recently proposed to capture observer, aborter and confiner aspects directly in the language under consideration. Namely for each category, they define a specific aspect language with the property that any aspect written in that language belongs to the category.

Rinard et al. [25] present a classification for different kinds of interference, using program analyses for automatic classification. No proofs are given that the analyses are actually correct.

Dantas and Walker define an object calculus extended with harmless advice [7]. In their work, harmless advice is advice that can only change the termination of a program and perform I/O operations. Harmlessness is guaranteed using a type-and-effect system related to information flow type systems that prevent information flow from advice to base component using protection domains. Their notion of harmlessness is a particular instance of the more general notion studied in MRI and in this work.

Douence et al. [11] present a formal approach to establish that two stateful aspects commute, and in that sense do not interfere. Their work, specific to stateful effect, is also based on equational reasoning, but the language under consideration is only partially defined and no theorem is stated.

A well-known situation of non-interference has been captured by Clifton and Leavens as observers [5]. They have later proposed an extension of AspectJ with annotations to control two forms of interference on control and heap effects [6]. The correctness of annotations is also checked using a type-and-effect system.

Translucid contracts use grey box specifications and structural refinements in verification to enforce control-flow properties [3]. Using the interference combinator of MRI, similar properties can be enforced at the level of types [22].

Krishnamurthi et al. present a technique for modular model checking of aspects [18]. Given a set of properties to satisfy and a fixed set of pointcuts, they generate sufficient conditions on the pointcuts themselves to enable modular verification.

Recently, Disenfeld and Katz define a compositional model checking method for events and aspects specification using temporal logic on event detection [9]. The technique is used to detect interference in systems where aspects may be activated during the execution of other aspects.

8. Conclusions and Future Work

In the pointcut/advice model of aspect-oriented programming, unrestricted quantification through pointcuts forces global reasoning. We show that such global reasoning can be compositional. Compositional reasoning is crucial for formal reasoning to scale up to large systems; equivalence proofs are hard to develop, so they should be partially reused as much as possible when a system evolves. We develop a framework for compositional reasoning about interference, using monads to express and reason about effects in a pure functional setting.

We introduce a general equivalence theorem that relies on four sufficient conditions—namely compositional weaving, compositional projection of effects, contextual and local harmlessness—that can be proven and reused independently. We demonstrate how the framework can be used to reason about a variety of scenarios related to the evolution of aspect-oriented programs.

A promising line of future and ongoing research is to study means to strengthen compositional reasoning to achieve modular reasoning
reasoning under certain scenarios. A first approach is to use parametricity. For instance, Tabareau et al. [26] use parametricity to define non-interfering pointcuts and advices, following the techniques 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 equivalence of modules.

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

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

References


