Compositional Reasoning About Aspect Interference

(Full version with proofs)

Ismael Figueroa[1,3] Tom Schrijvers[2] Nicolas Tabareau[3] Éric Tanter[1]

[1]PLEIAD Lab, Computer Science Dept (DCC)—University of Chile, Chile
[2]Dept. of Applied Mathematics and Computer Science—Ghent University, Belgium
[3]ASCOLA Group—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.

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 incremental 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 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 [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_ω and elaborating the model as a Haskell library[1]. 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

Published as Technical Report TR/DCC-2013-8, October 2013, University of Chile.

[1] http://pleiad.cl/research/cri

setting. Section 7 discusses related work and Section 8 concludes. Several proofs are included in the paper; others are presented in Appendix **??**.

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.

$$
\begin{array}{l}
prog\ x\ y = \\
\quad \textbf{let}\ r_1 = f\ x\ \textbf{in} \\
\quad \textbf{let}\ r_2 = g\ y\ \textbf{in} \\
\quad r_1 + r_2
\end{array}
$$

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 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:

$$s_1 = run\ [(call\ f, log)]\ prog$$

with a typical implementation of the logging advice:

$$
\begin{array}{l}
log\ proceed\ x = \\
\quad print\ \texttt{"Entering function ..."} \\
\quad proceed\ x
\end{array}
$$

Is the behavior of s_1 equivalent to the original program? Strictly speaking, they are not equivalent if we consider the output generated by $print$. 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:

$$s_2 = run\ [(call\ f \vee call\ g, log)]\ prog$$

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:

$$
\begin{array}{l}
prog'\ x\ y = \\
\quad \textbf{let}\ r_1 = h\ x\ \textbf{in} \\
\quad \textbf{let}\ r_2 = g\ y\ \textbf{in} \\
\quad r_1 + r_2
\end{array}
$$

$$s_3 = run\ [(call\ f \vee call\ g, log)]\ prog'$$

A first observation is that $call\ f$ will never match. We must change references to f also in the aspect environment:

$$s_4 = run\ [(call\ h \vee call\ g, log)]\ prog'$$

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:

$$
\begin{array}{l}
memo\ proceed\ x = \\
\quad \textbf{if}\ (member\ x\ table)\ \textbf{then}\ table\ [x] \\
\qquad\qquad\qquad \textbf{else let}\ r = proceed\ x\ \textbf{in} \\
\qquad\qquad\qquad\quad insert\ (table, x, r) \\
\qquad\qquad\qquad\quad r
\end{array}
$$

The advice maintains a reference to a lookup $table$ 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 $memo$ 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?

$$s_5 = run\ [(call\ f \vee call\ g, memo)]\ prog$$

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 accomodates 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 $bind$ ($\gg=$ in Haskell) and $return$. At the

```
class Monad where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
class MonadTrans t where
  lift :: m a → t m a
```

```
-- State
S_T      :: S_T s m a
runS_T :: S_T s m a → s → m (a, s)
π_S      :: s → S_T s m a → m a
class Monad m ⇒ S_M m | m → s where
  get :: m s
  put :: s → m ()
```

```
-- Writer
W_T      :: W_T w m a
runW_T :: W_T w m a → m (a, w)
π_W      :: W_T w m a → m a
class (Monoid w, Monad m) ⇒
      W_M w m | m → w where
  tell :: w → m ()
```

Figure 1. Monads and monad transformer types used in this paper.

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 $return$ 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 $return$ and $\gg=$ depend on the computational effect being implemented.

Monad transformers A monad transformer is a type constructor that allows 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 ($\mathbb{S}_\mathrm{T}$) and writer ($\mathbb{W}_\mathrm{T}$) monad transformers. In Figure 1 we summarize the types of their constructors ($\mathbb{S}_\mathrm{T}$,$\mathbb{W}_\mathrm{T}$), evaluation functions ($run\mathbb{S}_\mathrm{T}$, $run\mathbb{W}_\mathrm{T}$), and projection functions (π_S, π_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 mtl defines a set of type classes associated to particular effects. This allows 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 $\mathbb{S}_\mathrm{M}$ and $\mathbb{W}_\mathrm{M}$ 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 $\mathbb{S}_\mathrm{T}$ and $\mathbb{W}_\mathrm{T}$ 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 are *observationally equivalent*. 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 η-reduction rule from the λ-calculus, which states that $\lambda x \to f\ x \equiv f$. Also, Haskell provides the **do** notation as syntactic sugar for $\gg=$, hence $\mathbf{do}\ \{x \leftarrow f; g\ x\} \equiv f \gg= \lambda x \to g\ x$.

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:

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

The first two laws, left and right identity, state that $return$ 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:

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

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 considerd 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 $\mathbb{A}_\mathrm{T}$ 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], JPIs [4]), etc.

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

```
data Jp m a b
type Pc m a b = Jp m a b → Bool
```

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 $a \to \mathbb{A}_\mathrm{T}\ m\ b$ and returns a function

of the same type whose application produces a join point (this effect is encapsulated in the $\mathbb{A}_\mathrm{T}$ monad transformer):

$$(\#) :: (a \to \mathbb{A}_\mathrm{T}\ m\ b) \to (a \to \mathbb{A}_\mathrm{T}\ m\ b)$$

An open application is realized explicitly using $\#$: $f\ \#\ 2$ 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.

$$\mathbf{type}\ Advice\ m\ a\ b = (a \to m\ b) \to (a \to m\ b)$$
$$\mathbf{type}\ Aspect\ m\ a\ b = (Pc\ m\ a\ b, Advice\ m\ a\ b)$$

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

$$\mathbf{type}\ AEnv\ m$$

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 $\mathbb{A}_\mathrm{T}$ transformer, we require a function that evaluates an $\mathbb{A}_\mathrm{T}$ computation given an aspect environment:

$$run\mathbb{A}_\mathrm{T} :: Monad\ m \Rightarrow AEnv\ (\mathbb{A}_\mathrm{T}\ m) \to \mathbb{A}_\mathrm{T}\ m\ a \to m\ a$$

Abstracting open applications Similarly to the $\mathbb{S}_\mathrm{M}$ and $\mathbb{W}_\mathrm{M}$ type classes, we introduce a type class to define an abstract interface for performing open applications:

$$\mathbf{class}\ Monad\ m \Rightarrow \mathbb{A}_\mathrm{M}\ m\ \mathbf{where}$$
$$\quad \#^i :: (Int \to m\ Int) \to (Int \to m\ Int)$$
$$\mathbf{instance}\ Monad\ m \Rightarrow \mathbb{A}_\mathrm{M}\ (\mathbb{A}_\mathrm{T}\ m)\ \mathbf{where}\ ...$$

The only operation of this class is $\#^i$, and we require that any monad $\mathbb{A}_\mathrm{T}\ m$ be an instance of this class. Note that $\mathbb{A}_\mathrm{M}$ 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 $\mathbb{A}_\mathrm{M}$, cannot perform any open applications (and hence cannot emit join points).

3.3 Necessary Properties of $\mathbb{A}_\mathrm{T}$

To be a correct model, the $\mathbb{A}_\mathrm{T}$ 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 $run\mathbb{A}_\mathrm{T}\ aenv$ 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:

$$h \circ return \quad \equiv \quad return$$
$$h\ (m \gg\!= f) \quad \equiv \quad h\ m \gg\!= h \circ f \qquad (\forall m, f)$$

$$log :: \mathbb{W}_\mathrm{M}\ String\ m \Rightarrow Advice\ m\ a\ b$$
$$log\ proceed\ x = \mathbf{do}$$
$$\quad tell\ \texttt{"Entering function ..."}$$
$$\quad proceed\ x$$
$$memo :: (Ord\ a, \mathbb{S}_\mathrm{M}\ (Map\ a\ b)\ m) \Rightarrow Advice\ m\ a\ b$$
$$memo\ proceed\ x = \mathbf{do}$$
$$\quad table \leftarrow get$$
$$\quad \mathbf{if}\ member\ x\ table\ \mathbf{then}\ return\ (table\,!\,x)$$
$$\quad\quad \mathbf{else\ do}\ y \leftarrow proceed\ x$$
$$\quad\quad\quad\quad table' \leftarrow get$$
$$\quad\quad\quad\quad put\ (insert\ x\ y\ table')$$
$$\quad\quad\quad\quad return\ y$$

Figure 2. Logging and memoization advice in monadic style.

For $run\mathbb{A}_\mathrm{T}$, the first monad is $\mathbb{A}_\mathrm{T}\ m$ 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.[2]

In the same spirit, we also require that a third law holds for $run\mathbb{A}_\mathrm{T}\ aenv$:[3]

$$run\mathbb{A}_\mathrm{T}\ aenv \circ lift \quad \equiv \quad id$$

This law expresses that $run\mathbb{A}_\mathrm{T}\ aenv$ is a left inverse of *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 $\mathbb{A}_\mathrm{T}$ transformer is implemented. We will come back to this when presenting a simple $\mathbb{A}_\mathrm{T}$ transformer in Section 5.

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:

$$prog\ x\ y = \mathbf{do}\ r_1 \leftarrow f\ \#^i\ x$$
$$\quad\quad\quad\quad\quad\quad r_2 \leftarrow g\ \#^j\ y$$
$$\quad\quad\quad\quad\quad\quad return\ (r_1 + r_2)$$

The program can be run as an aspectual computation in the $\mathbb{A}_\mathrm{T}$ transformer with a logging aspect on open applications of f as follows:

$$run\mathbb{A}_\mathrm{T}\ [(fPc, log)]\ (prog\ 5\ 12)$$

The pointcut fPc 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 *log* and *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

[2] If $\mathbb{A}_\mathrm{T}$ 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 $return \equiv lift \circ return$.

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 can easily be decomposed into all the individual open functions (rather than just two parts) by repeated application of the theorem. In fact, the third condition below, which relates to the context, is an instance of the theorem and thus explicitly invites this systematic decomposition.

Theorem 1. *Given an expression:*

$$system :: \forall m.C\ m \Rightarrow A \to \mathbb{A}_T\ m\ B$$

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:

$$system \equiv c\ (f\ \#^i)$$

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):

$$c :: \forall m.C\ m \Rightarrow (A' \to \mathbb{A}_T\ m\ B') \to A \to \mathbb{A}_T\ m\ B$$
$$f :: \forall m.C_f\ m \Rightarrow A' \to \mathbb{A}_T\ m\ B'$$

Also, we are given two aspect environments $aenv$ and $aenv'$ of types:

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

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

The given projection function:

$$\pi :: \forall m\ a.Monad\ m \Rightarrow T\ m\ a \to m\ a$$

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:

$$run\mathbb{A}_T\ aenv\ system$$
$$\equiv$$
$$\pi\ (run\mathbb{A}_T\ aenv'\ system)$$

The four conditions on c and f are:

1. *Compositional weaving*

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

2. *Compositional projection*

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

3. *Contextual harmlessness*

$$run\mathbb{A}_T\ aenv \circ c \circ (\lambda g \to lift \circ g)$$
$$\equiv$$
$$\pi \circ run\mathbb{A}_T\ aenv' \circ c \circ (\lambda g \to lift \circ lift \circ g)$$

4. *Local harmlessness*

$$run\mathbb{A}_T\ aenv \circ (f\ \#^i)$$
$$\equiv$$
$$\pi \circ run\mathbb{A}_T\ aenv' \circ (f\ \#^i)$$

Proof. The proof proceeds by straightforward equational reasoning:

$$\begin{aligned}
& run\mathbb{A}_{\mathrm{T}}\ aenv\ system \\
\equiv\ & \{\text{-system decomposition -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv\ (c\ (f\ \#^i)) \\
\equiv\ & \{\text{-compositional weaving -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv \circ f\ \#^i) \\
\equiv\ & \{\text{-local harmlessness -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ c\ (lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^i) \\
\equiv\ & \{\text{-contextual harmlessness -}\} \\
& \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ lift \circ \pi \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^i)) \\
\equiv\ & \{\text{-compositional projection -}\} \\
& \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv' \circ c\ (lift \circ run\mathbb{A}_{\mathrm{T}}\ aenv' \circ f\ \#^i) \\
\equiv\ & \{\text{-compositional weaving -}\} \\
& \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv'\ (c\ (f\ \#^i))) \\
\equiv\ & \{\text{-system decomposition -}\} \\
& \pi\ (run\mathbb{A}_{\mathrm{T}}\ aenv'\ system)
\end{aligned}$$

□

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

$$\begin{aligned}
& f_1 = f \\
& c_1 = \lambda f\ x\ y \to \mathbf{do}\ r_1 \leftarrow f\ x \\
& \qquad\qquad\qquad\quad\ r_2 \leftarrow g\ \#^j\ y \\
& \qquad\qquad\qquad\quad\ return\ (r_1 + r_2)
\end{aligned}$$

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 $run\mathbb{A}_{\mathrm{T}}\ env$ 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.
2. c invokes f (once or more), which means embedding it in its larger computation (once or more) with $\gg\!=$, which is where the second law comes in. Note that the second law can be used repeatedly to tackle a larger computation sequence $m \gg\!= f_1 \gg\!= \ldots \gg\!= f_n$.

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:

$$c = \lambda f \to \mathit{lift} \circ \mathit{run}\mathbb{A}_{\mathrm{T}}\ [\,] \circ f$$

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 $\mathbb{A}_{\mathrm{T}}$ 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 $\mathbb{A}_{\mathrm{M}}$ constraint on the monad stack instead of applying the $\mathbb{A}_{\mathrm{T}}$ transformer. It prevents c from invoking the weaving function $\mathit{run}\mathbb{A}_{\mathrm{T}}$ locally on f because $\mathit{run}\mathbb{A}_{\mathrm{T}}$ only works for $\mathbb{A}_{\mathrm{T}}\ m'$ and not for all possible m that instantiate $\mathbb{A}_{\mathrm{M}}$.

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

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

$$c :: \forall m.(C\ \ m, \mathbb{A}_M\ m) \Rightarrow (A' \to m\ B') \to A \to m\ B$$
$$f :: \forall m.(C_f\ m, \mathbb{A}_M\ m) \Rightarrow\ \ A' \to m\ B'$$

the condition of compositional weaving holds.

We believe that this conjecture can be proven with logical relations, which is a rather technically challenging task that is out of 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 π is a monad morphism, then the same solution as for compositional weaving applies. For instance, the projection π_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 $\mathbb{W}_{\mathrm{M}}$, 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 π_S of the state effect is *not* a monad morphism:

$$\pi_S\ 0\ (\mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1) \gg \mathit{get}) \quad \equiv \quad \mathit{return}\ 1$$
$$\pi_S\ 0\ (\mathit{get} \gg\!= \lambda x \to \mathit{put}\ (x+1)) \gg \pi_S\ 0\ \mathit{get} \quad \equiv \quad \mathit{return}\ 0$$

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 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 $\mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}' \circ (f\ \#^i)$ 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 captures this as:

  $$\exists h, \mathit{lift} \circ h \equiv \mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}' \circ (f\ \#^i)$$

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

  $$\begin{aligned} & \mathit{lift} \circ \mathit{lift} \circ \pi \circ \mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}' \circ (f\ \#^i) \\ \equiv\ & \{\text{-assumption -}\} \\ & \mathit{lift} \circ \mathit{lift} \circ \pi \circ \mathit{lift} \circ h \\ \equiv\ & \{\text{-}\pi \text{ is left inverse of } \mathit{lift} \text{ -}\} \\ & \mathit{lift} \circ \mathit{lift} \circ h \\ \equiv\ & \{\text{-assumption -}\} \\ & \mathit{lift} \circ \mathit{run}\mathbb{A}_{\mathrm{T}}\ \mathit{aenv}' \circ (f\ \#^i) \end{aligned}$$

  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 $\mathbb{S}_{\mathrm{T}}$ 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 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 advices 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 $c_1\ (\mathit{lift} \circ h)$, where $h :: C\ m \Rightarrow A' \to m\ \ B'$ 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:

$$\begin{aligned}\mathit{system}' = \lambda x\ y \to \mathbf{do}\ & r_1 \leftarrow \mathit{lift}\ (h\ x)\\ & r_2 \leftarrow g\ \#^j\ y\\ & \mathit{return}\ (r_1 + r_2)\end{aligned}$$

which can be decomposed as $\mathit{system}' = c_2\ (f_2\ \#^j)$:

$$\begin{aligned}& f_2 = g\\ & c_2 = \lambda g\ x\ y \to \mathbf{do}\ r_1 \leftarrow \mathit{lift}\ (h\ x)\\ & \qquad\qquad\qquad\quad r_2 \leftarrow g\ y\\ & \qquad\qquad\qquad\quad \mathit{return}\ (r_1 + r_2)\end{aligned}$$

Here we can consider the harmlessness of the extended environment aenv' separately for g and c_2. Note that since c_2 does not contain any more open applications, contextual harmlessness is trivally 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 $\mathit{Int} \to m\ \mathit{Int}$, 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; $\mathbb{A}_\mathrm{T}$ 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.

5.1 An Embedding of Open Applications

We implement $\mathbb{A}_\mathrm{T}$ as a monad transformer that captures open function applications in a syntatic form.[4] The interpreter function

$$\begin{aligned}&\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathbb{A}_\mathrm{T}\ m)\ \mathbf{where}\\ &\quad \mathit{return}\ \ = \mathbb{A}_\mathrm{T} \circ \mathit{return} \circ \mathit{Return}\\ &\quad m \gg\!= f = \mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ m \gg\!= \lambda r \to \mathbf{case}\ r\ \mathbf{of}\\ &\qquad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x)\\ &\qquad \mathit{OpenApp}\ t\ g\ x\ k \to\\ &\qquad\quad \mathit{return}\ (\mathit{OpenApp}\ t\ g\ x\ (\lambda y \to k\ y \gg\!= f)))\\ &\mathbf{instance}\ \mathit{MonadTrans}\ \mathbb{A}_\mathrm{T}\ \mathbf{where}\\ &\quad \mathit{lift}\ ma = \mathbb{A}_\mathrm{T}\ (ma \gg\!= \lambda a \to (\mathit{return} \circ \mathit{Return})\ a)\end{aligned}$$

Figure 3. $\mathbb{A}_\mathrm{T}$ instances for the Monad and $\mathit{MonadTrans}$ type classes.

$\mathit{run}\mathbb{A}_\mathrm{T}$ interpretes 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:

$$\begin{aligned}&\mathbf{type}\ \mathit{Tag} = \mathit{Int}\\ &\mathbf{data}\ \mathit{Jp}\ m\ a\ b = \mathit{Jp}\ \mathit{Tag}\\ &\mathit{pcTag}\ t\ (\mathit{Jp}\ t') = t \equiv t'\end{aligned}$$

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 $\mathbb{A}_\mathrm{T}$ transformer extends a given monad m with the ability to expose some open function applications. A computation $\mathbb{A}_\mathrm{T}\ \ m\ \ a$ is denoted by an alternating sequence of computations in the monad m and exposed open function applications starting with the former.

$$\begin{aligned}&\mathbf{data}\ \mathbb{A}_\mathrm{T}\ m\ a = \mathbb{A}_\mathrm{T}\ \{\mathit{un}\mathbb{A}_\mathrm{T} :: m\ (\mathit{Result}\mathbb{A}_\mathrm{T}\ m\ a)\}\\ &\mathbf{data}\ \mathit{Result}\mathbb{A}_\mathrm{T}\ m\ a\\ &\qquad = \mathit{Return}\ a\\ &\qquad \mid\ \mathit{OpenApp}\ \mathit{Tag} && \text{-- tag}\\ &\qquad\qquad\quad (\mathit{Int} \to \mathbb{A}_\mathrm{T}\ m\ \mathit{Int}) && \text{-- function}\\ &\qquad\qquad\quad \mathit{Int} && \text{-- argument}\\ &\qquad\qquad\quad (\mathit{Int} \to \mathbb{A}_\mathrm{T}\ m\ a) && \text{-- continuation}\end{aligned}$$

The $\mathit{Result}\mathbb{A}_\mathrm{T}$ 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 $\mathit{OpenApp}$ constructor. In particular, $\mathit{OpenApp}\ t\ g\ x\ k$ 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 $\mathit{MonadTrans}$ 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:

$$\mathit{openApp}\ t\ f\ x = \mathbb{A}_\mathrm{T}\ (\mathit{return}\ (\mathit{OpenApp}\ t\ f\ x\ \mathit{return}))$$

Because 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 $\mathit{openApp}$, $\mathbb{A}_\mathrm{T}$ can be declared as an instance of the $\mathbb{A}_\mathrm{M}$ type class:

$$\begin{aligned}&\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathbb{A}_\mathrm{M}\ (\mathbb{A}_\mathrm{T}\ m)\ \mathbf{where}\\ &\quad f\ \#^t\ x = \mathit{openApp}\ t\ f\ x\end{aligned}$$

[4] Our $\mathbb{A}_\mathrm{T}$ implementation is a close cousin of a free monad.

5.2 Running $\mathbb{A}_\mathrm{T}$ Computations

We define the $run\mathbb{A}_\mathrm{T}$ interpreter function which evaluates an $\mathbb{A}_\mathrm{T}$ computation:

$$
\begin{array}{l}
run\mathbb{A}_\mathrm{T}\ aenv\ m = un\mathbb{A}_\mathrm{T}\ m \gg\!= go\ \mathbf{where} \\
\quad go\ (Return\ r) = return\ r \\
\quad go\ (OpenApp\ t\ f\ x\ k) = \\
\quad\quad un\mathbb{A}_\mathrm{T}\ (weave\ f\ aenv\ (Jp\ t)) \gg\!= \\
\quad\quad\quad \lambda woven\_f \rightarrow woven\_f\ x \gg\!= k) \gg\!= go
\end{array}
$$

This function is defined in terms of the locally-defined go function. In case of $Return\ r$ values, it simply unwraps and $return$s value r. When it encounters an open application, it creates a join point $Jp\ t$ and uses the $weave$r 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 weaver is defined recursively on the aspect environment as follows.

$$
\begin{array}{l}
weave :: Monad\ m \Rightarrow (Int \rightarrow m\ Int) \rightarrow AEnv\ m \rightarrow \\
\qquad\qquad\qquad\qquad Jp\ m\ Int\ Int \rightarrow m\ (Int \rightarrow m\ Int) \\
weave\ f\ [\,]\ \_ \qquad\qquad\qquad\qquad = return\ f \\
weave\ f\ ((pc, adv) : asps)\ jp = \\
\quad weave\ (\mathbf{if}\ pc\ jp\ \mathbf{then}\ (adv\ f)\ \mathbf{else}\ f)\ asps\ jp
\end{array}
$$

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 $\mathbb{A}_\mathrm{T}$

To exploit the general result of the previous section, we need to establish that $\mathbb{A}_\mathrm{T}$ is a proper aspectual monad transformer that satisfies the necessary properties described in Section 3.

Lemma 1 (Monad laws for $\mathbb{A}_\mathrm{T}$)**.** *$\mathbb{A}_T$ fulfills the monad transformer laws. In addition, for any monad m, $\mathbb{A}_T\ m$ fulfills the monad laws.*

Lemma 2 ($run\mathbb{A}_\mathrm{T}$ monad morphism)**.** *For any aspect environment $aenv$, $run\mathbb{A}_T\ aenv$ 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 Appendix **??**. Crucially, the proofs rely on the monad and monad transformer laws for $\mathbb{A}_\mathrm{T}$.

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 $run\mathbb{A}_\mathrm{T}$, its internally-defined function go, and the $\gg\!=$ operation of $\mathbb{A}_\mathrm{T}$; it also uses the monad laws on m, and the identity $un\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathbb{A}_\mathrm{T} \circ un\mathbb{A}_\mathrm{T} \equiv id$. A crucial step is the application of the co-induction hypothesis, which allows 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 $aenv'$ with respect to the initial environment $aenv$ can be established in this setting. Concretely, we must prove that:

$$run\mathbb{A}_\mathrm{T}\ aenv \circ (f\ \#^i) \equiv \pi \circ run\mathbb{A}_\mathrm{T}\ aenv' \circ (f\ \#^i)$$

$$
\begin{array}{ll}
 & run\mathbb{A}_\mathrm{T}\ aenv\ (m\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f) \\
\equiv & \{\text{-unfold } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{-}\} \\
 & run\mathbb{A}_\mathrm{T}\ aenv\ (\mathbb{A}_\mathrm{T}\ (un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow un\mathbb{A}_\mathrm{T}\ (f\ x) \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow return_m\ (\\
 & \quad\quad OpenApp\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)))) \\
\equiv & \{\text{-unfold } run\mathbb{A}_\mathrm{T} \text{ and } un\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv id \text{ -}\} \\
 & (un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow un\mathbb{A}_\mathrm{T}\ (f\ x) \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow return_m \\
 & \quad\quad (OpenApp\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)))\ \gg\!=_m\ go \\
\equiv & \{\text{-associativity of } \gg\!=_m \text{ + distributing } go \text{ over } \mathbf{case} \text{ -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow un\mathbb{A}_\mathrm{T}\ (f\ x)\ \gg\!=_m\ go \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow return_m \\
 & \quad\quad (OpenApp\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))\ \gg\!=_m\ go \\
\equiv & \{\text{-folding } run\mathbb{A}_\mathrm{T} \text{ + } un\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv id \text{ + left identity + } go \text{ -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow run\mathbb{A}_\mathrm{T}\ aenv\ (f\ x) \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow un\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (return_m \\
 & \quad\quad (OpenApp\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))))\ \gg\!=_m\ go \\
\equiv & \{\text{-left identity of } m \text{ + folding } run\mathbb{A}_\mathrm{T} \text{ -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow run\mathbb{A}_\mathrm{T}\ aenv \\
 & \quad\quad (\mathbb{A}_\mathrm{T}\ (return_m\ (OpenApp\ t\ x\ g\ (\lambda y \rightarrow k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)))) \\
\equiv & \{\text{-folding } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{ -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow run\mathbb{A}_\mathrm{T}\ aenv \\
 & \quad\quad ((\mathbb{A}_\mathrm{T}\ (return_m\ (OpenApp\ t\ x\ g\ k))\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))) \\
\equiv & \{\text{-co-induction hypothesis -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow run\mathbb{A}_\mathrm{T}\ aenv \\
 & \quad\quad (\mathbb{A}_\mathrm{T}\ (return_m\ (OpenApp\ t\ x\ g\ k))) \\
 & \quad \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
\equiv & \{\text{-factoring } run\mathbb{A}_\mathrm{T}\ aenv \circ f \text{ from the } \mathbf{case} \text{ branches -}\} \\
 & un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow (\mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow run\mathbb{A}_\mathrm{T}\ aenv \\
 & \quad\quad (\mathbb{A}_\mathrm{T}\ (return_m\ (OpenApp\ t\ x\ g\ k)))) \\
 & \quad \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
\equiv & \{\text{-associativity of } m \text{ -}\} \\
 & (un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow run\mathbb{A}_\mathrm{T}\ aenv \\
 & \quad\quad (\mathbb{A}_\mathrm{T}\ (return_m\ (OpenApp\ t\ x\ g\ k)))) \\
 & \quad \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
\equiv & \{\text{-unfolding of } run\mathbb{A}_\mathrm{T} \text{ + } un\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv id \text{ -}\} \\
 & (un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m \lambda r \rightarrow \mathbf{case}\ r\ \mathbf{of} \\
 & \quad Return\ x \rightarrow return\ x \\
 & \quad OpenApp\ t\ x\ g\ k \rightarrow return_m \\
 & \quad\quad (OpenApp\ t\ x\ g\ k\ \gg\!=_m\ go))\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
\equiv & \{\text{-folding } go \text{ -}\} \\
 & (un\mathbb{A}_\mathrm{T}\ m\ \gg\!=_m\ go)\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f \\
\equiv & \{\text{-folding } run\mathbb{A}_\mathrm{T} \text{ -}\} \\
 & run\mathbb{A}_\mathrm{T}\ aenv\ m\ \gg\!=_m\ run\mathbb{A}_\mathrm{T}\ aenv \circ f
\end{array}
$$

Figure 4. Proof of the second monad morphism law for $run\mathbb{A}_\mathrm{T}$.

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).

$$\begin{array}{ccc} f_{MRI} + mix & \xrightarrow[\equiv]{\text{(b)}} & \pi(f_{MRI} + mix') \\ \text{(a)} \Big\downarrow & & \Big\downarrow \text{(c)} \\ f_{AOP} + aenv & \xrightarrow[\text{(d)}]{\equiv} & \pi(f_{AOP} + aenv') \end{array}$$

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 mix, such that their composition is equivalent to this initial system. In the same way, step (c) requires to find a mixin 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 mix and the projection of f_{MRI} composed with 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).

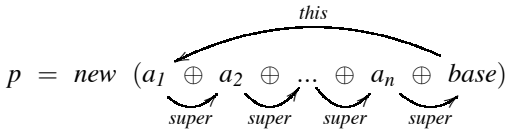
We now briefly summarize the MRI model and prove a theorem connecting MRI to AOP. Then, using the Fibonacci function as a concrete example, we show how to prove that the logging and memoization aspects from Figure 2 are locally harmless.

6.1 Background: the MRI Framework

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

$$\begin{array}{l} \textbf{type}\ \mathit{Open}\ s = s \to s \\ \mathit{new} :: \mathit{Open}\ s \to s \\ \mathit{new}\ a = \mathit{fix}\ (\lambda f \to a\ f) \\ (\oplus) :: \mathit{Open}\ s \to \mathit{Open}\ s \to \mathit{Open}\ s \\ a_1 \oplus a_2 = \lambda \mathit{super} \to a_1\ (a_2\ \mathit{super}) \end{array}$$

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



To create a component $\oplus$ instantiates $super$ references for every extended component, and new instantiates the self-reference $this$.

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

$$\pi\ (new\ (mix \oplus bse)) \equiv run\mathbb{I} \circ new\ bse$$

for some projection π. Here $run\mathbb{I}$ 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 $Advice\ m\ a\ b$ and $Open\ (a \to m\ b)$ denote the same type $(a \to m\ b) \to (a \to m\ b)$. 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 regression by matching join points emitted on its own advice. However, if an advice uses a type class constraint that does not entail $\mathbb{A}_{\mathrm{M}}$ (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:

$$f_{AOP} \equiv \mathit{fix}\ (\lambda f \to f_{MRI}\ (f\ \#^t))$$

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.

Theorem 2. *Given functions*

$$\begin{array}{l} f_{MRI} :: \forall m.C\ \ m \Rightarrow (A \to m\ B) \to (A \to m\ B) \\ adv_i\ :: \forall m.D_i\ m \Rightarrow (A \to m\ B) \to (A \to m\ B) \end{array}$$

where C and D_i are type class constraints that entail $Monad$ but not $\mathbb{A}_M$, and given an aspect environment

$$aenv = [(pc_1, adv_1), ..., (pc_n, adv_n)]$$

we have that

$$run\mathbb{A}_T\ aenv \circ f_{AOP}\ \#^t \equiv new\ (adv'_k \oplus ... \oplus adv'_1 \oplus f_{MRI})$$

where f_{AOP} is such that

$$f_{AOP} \equiv \mathit{fix}\ (\lambda f \to f_{MRI}\ (f\ \#^t))$$

*and $[adv'_1, ..., adv'_k]$ is the list of all advices in $aenv$ that are used with a pointcut that matches $\#^t$ applications (*e.g. *when the pointcut is $pcTag\ t$).*

Proof. The proof proceeds by equational reasoning:

[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.

$$
\begin{array}{ll}
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{AOP}\ \#^t \\
\equiv & \{\text{-definition of } f_{AOP}\text{ -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ fix\ (\lambda f \to f_{MRI}\ (f\ \#^t))\ \#^t \\
\equiv & \{\text{-unfolding of the fixpoint -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (f_{AOP}\ \#^t)\ \#^t \\
\equiv & \{\text{-compositionality of weaving -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{AOP}\ \#^t)\ \#^t \\
\equiv & \{\text{-co-induction hypothesis -}\} \\
& run\mathbb{A}_{\mathrm{T}}\ aenv \circ f_{MRI}\ (new\ (adv'_k \oplus ... \oplus adv'_1 \oplus f_{MRI}))\ \#^t \\
\equiv & \{\text{-weaving -}\} \\
& adv'_k \circ ... \circ adv'_1 \\
& \quad \circ f_{MRI}\ (new\ (adv'_k \oplus ... \oplus adv'_1 \oplus f_{MRI})) \\
\equiv & \{\text{-definition of } \oplus\text{ -}\} \\
& adv'_k \oplus ... \oplus adv'_1 \\
& \quad \oplus f_{MRI}\ (new\ (adv'_k \oplus ... \oplus adv'_1 \oplus f_{MRI})) \\
\equiv & \{\text{-folding the } new \text{ fixpoint -}\} \\
& new\ (adv'_k \oplus ... \oplus adv'_1 \oplus f_{MRI})
\end{array}
$$

□

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 $plainFib$ combines the base program and advice to provide the regular unoptimized version of Fibonacci.

$$
\begin{array}{l}
plainFib\ n = run\mathbb{A}_{\mathrm{T}}\ [(pcTag\ t, fibAdv)]\ (fibBase\ \#^t\ n) \\
fibBase\ \_ = return\ 1 \\
fibAdv\ proceed\ n = \\
\quad \textbf{if}\ (n \leqslant 2)\ \textbf{then}\ proceed\ n \\
\qquad\qquad \textbf{else do}\ f_1 \leftarrow fibBase\ \#^t\ (n-1) \\
\qquad\qquad\qquad\quad f_2 \leftarrow fibBase\ \#^t\ (n-2) \\
\qquad\qquad\qquad\quad return\ (f_1 + f_2)
\end{array}
$$

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 $\mathbb{A}_{\mathrm{M}}$; 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 $super$.

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 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 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

$$fib_{AOP} \equiv fix\ (\lambda f \to fib_{MRI}\ (f\ \#^t))$$

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 *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 $super$ 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 satisfy Theorem 2.

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 to the work on EffectiveAdvice [21] and its successor, MRI [22]. The present work 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.

```
fib_AOP :: 𝔸_M m ⇒ Int → m Int
fib_AOP n = case n of
  0 → return 1
  1 → return 1
  _ → do y ← fib_AOP #^t (n − 1)
         x ← fib_AOP #^t (n − 2)
         return (x + y)

plainFib_AOP  :: Monad m ⇒ Int → m Int
plainFib_AOP  = run𝔸_T [] ∘ fib_AOP #^t

logFib_AOP    :: Monad m ⇒ Int → 𝕎_T String m Int
logFib_AOP    = run𝔸_T [(pcTag t, log')] ∘ fib_AOP #^t

memoFib_AOP :: Monad m ⇒ Int → 𝕊_T (Map Int Int) m Int
memoFib_AOP = run𝔸_T [(pcTag t, memo)] ∘ fib_AOP #^t
```

```
fib_MRI :: Monad m ⇒ Open (Int → m Int)
fib_MRI this n = case n of
  0 → return 1
  1 → return 1
  _ → do y ← this (n − 1)
         x ← this (n − 2)
         return (x + y)

plainFib_MRI  :: Monad m ⇒ Int → m Int
plainFib_MRI  = new fib_MRI

logFib_MRI    :: Monad m ⇒ Int → 𝕎_T String m Int
logFib_MRI    = new ∘ (log ⊕ fib_MRI)

memoFib_MRI :: Monad m ⇒ Int → 𝕊_T (Map Int Int) m Int
memoFib_MRI = new (memo ⊕ fib_MRI)
```

Figure 5. Fibonacci function. Left: in the simple pointcut/advice model of Section 5. Right: in the MRI setting (taken from [22])

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 sealing 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 spectative superposition (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, Djoko Djoko *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 advices [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 combinators 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. Compositionality 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 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.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. 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, July 2005. Springer-Verlag.

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

[3] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.

[4] E. Bodden, É. Tanter, and M. Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 2013. To appear.

[5] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *In FOAL Workshop*, 2002.

[6] C. Clifton, G. T. Leavens, and J. Noble. MAO: Ownership and effects for more effective reasoning about aspects. pages 451–475.

[7] D. S. Dantas and D. Walker. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 383–396, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

[8] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

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

[10] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. *Science of Computer Programming*, 77(3):393 – 422, 2012.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 141–150, Lancaster, UK, Mar. 2004. ACM Press.

[12] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[13] I. Figueroa, T. Schrijvers, N. Tabareau, and É. Tanter. Compositional reasoning about aspect interference – extended with supplementary material. Technical Report TR/DCC-2013-8, Computer Science Department, University of Chile, Oct. 2013.

[14] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, Apr. 1993.

[15] S. Katz. Aspect categories and classes of temporal properties. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 106–134. Springer Berlin Heidelberg, 2006.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. 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, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[17] G. Kiczales and M. Mezini. Aspect-oriented programming and modular 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 advice modularly. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pages 137–146, 2004.

[19] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL 95)*, pages 333–343, San Francisco, California, USA, Jan. 1995. ACM Press.

[20] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[21] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [2], pages 109–120.

[22] B. C. D. S. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular reasoning about interference in incremental programming. *Journal of Functional Programming*, 22:797–852, Nov. 2012.

[23] C. Prehofer. Semantic reasoning about feature composition via multiple aspect-weavings. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *Proceedings of the 5th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2006)*, pages 237–242, Portland, Oregon, Oct. 2006. ACM Press.

[24] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, 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, Cyprus, july 2008. Springer-Verlag.

[25] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*, pages 147–158. ACM Press, 2004.

[26] N. Tabareau, I. Figueroa, and É. Tanter. A typed monadic embedding of aspects. In J. Kinzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 171–184, Fukuoka, Japan, Mar. 2013. ACM Press.

[27] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [2], pages 13–24.

[28] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL 92)*, pages 1–14, Albuquerque, New Mexico, USA, Jan. 1992. ACM Press.

A. Monad and Transformer Laws

A.1 Monad Laws

A.1.1 Left Identity

$$
\begin{array}{l}
\mathit{return}_{\mathbb{A}_\mathrm{T}}\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f \\
\equiv\ \{\text{-unfolding } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{ and } \mathit{return}_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (\mathit{return}_m\ (\mathit{Return}\ x)))\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \{\text{-}\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{return}\ (\mathit{Return}\ a)\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \{\text{-left identity of } \gg\!=_m\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathbf{case}\ \mathit{Return}\ x\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \{\text{-applying } \mathbf{case} \text{ + } \mathbb{A}_\mathrm{T} \circ \mathit{un}\mathbb{A}_\mathrm{T} \equiv \mathit{id}\text{ -}\} \\
f\ x
\end{array}
$$

A.1.2 Right Identity

$$
\begin{array}{l}
p\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ \mathit{return}_{\mathbb{A}_\mathrm{T}} \\
\equiv\ \{\text{-unfolding } \gg\!=_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{return}\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ \mathit{return}_{\mathbb{A}_\mathrm{T}})) \\
\equiv\ \{\text{-}\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id} \text{ + unfolding } \mathit{return}_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (\mathit{return}_m\ (\mathit{Return}\ x))) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ \mathit{return}_{\mathbb{A}_\mathrm{T}})) \\
\equiv\ \{\text{-}\mathbb{A}_\mathrm{T} \circ \mathit{un}\mathbb{A}_\mathrm{T} \equiv \mathit{id} \text{ + co-induction hypothesis -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{return}_m\ (\mathit{Return}\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ (\mathit{OpenApp}\ t\ x\ g\ k)) \\
\equiv\ \{\text{-folding } \mathbf{case} \text{ branches -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ (\lambda r \to \mathit{return}_m\ r)) \\
\equiv\ \{\text{-}\eta\text{-reduction -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \mathit{return}_m) \\
\equiv\ \{\text{-right identity of } \gg\!=_m \text{ + } \mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id}\text{ -}\} \\
p
\end{array}
$$

A.1.3 Associativity of $\gg\!=_{\mathbb{A}_\mathbf{T}}$

$$
\begin{array}{l}
(p\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h \\
\equiv\ \{\text{-unfold } \gg\!=_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
[\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\qquad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\qquad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad\quad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))]\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h \\
\equiv\ \{\text{-unfold } \gg\!=_{\mathbb{A}_\mathrm{T}} \text{ + simplifications -}\} \\
\mathbb{A}_\mathrm{T}\ ((\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots)) \\
\quad \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots)) \\
\equiv\ \{\text{-associativity of } \gg\!=_m\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda x \to ((\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots)\ x \\
\quad \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots))) \\
\equiv\ \{\text{-}\beta\text{-reduction -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \ \ \lambda x \to (\mathbf{case}\ x\ \mathbf{of} \\
\qquad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x) \\
\qquad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad\quad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\quad \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots)) \\
\equiv\ \{\text{-distributing } \gg\!=_m \text{ over the } \mathbf{case} \text{ branches -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda x \to (\mathbf{case}\ x\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x)\ \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f) \\
\qquad \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots))) \\
\equiv\ \{\text{-}\mathit{id} \equiv \mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \text{ + left unit of } m \text{ and } \mathbf{case}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda x \to (\mathbf{case}\ x\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \\
\qquad (\mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x)\ \gg\!=_m\ (\lambda r \to \mathbf{case}\ r\ \mathbf{of}\ldots)) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h))) \\
\equiv\ \{\text{-folding definition of } \gg\!=_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda x \to (\mathbf{case}\ x\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (f\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h))) \\
\equiv\ \{\text{-}\eta\text{-abstraction + }\alpha\text{-renaming -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ p\ \gg\!=_m\ \lambda r \to (\mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ ((\lambda x \to f\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h)\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}} \lambda x \to f\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h))) \\
\equiv\ \{\text{-folding definition of } \gg\!=_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
p\ \gg\!=_{\mathbb{A}_\mathrm{T}} \lambda x \to (f\ x\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ h)
\end{array}
$$

A.2 Monad Transformer Laws

A.2.1 Identity Preservation

$$
\begin{array}{l}
\mathit{lift}\ (\mathit{return}_m\ x) \\
\equiv\ \{\text{-unfold } \mathit{lift}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{return}_m\ x\ \gg\!=_m\ (\lambda a \to \mathit{return}_m \circ \mathit{Return}\ a)) \\
\equiv\ \{\text{-left identity -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{return}_m \circ \mathit{Return}\ x) \\
\equiv \\
\mathit{return}_{\mathbb{A}_\mathrm{T}}\ x
\end{array}
$$

A.2.2 Composition Preservation

$$
\begin{array}{l}
\mathit{lift}\ m\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ (\mathit{lift} \circ f) \\
\equiv\ \{\text{-unfold } \gg\!=_{\mathbb{A}_\mathrm{T}}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift}\ m)\ \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f)) \\
\equiv\ \{\text{-unfold } \mathit{lift}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (\mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (m\ \gg\!=_m\ \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a))) \\
\quad \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))) \\
\equiv\ \{\text{-}\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id} \text{ + associativity of } \gg\!=_m\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (m\ \gg\!=_m\ \lambda a \to (\mathit{return}_m\ (\mathit{Return}\ a) \\
\quad \gg\!=_m\ \lambda r \to \mathbf{case}\ r\ \mathbf{of} \\
\quad \mathit{Return}\ x \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ x) \\
\quad \mathit{OpenApp}\ t\ x\ g\ k \to \mathit{return}_m\ \$ \\
\qquad \mathit{OpenApp}\ t\ x\ g\ (\lambda y \to k\ y\ \gg\!=_{\mathbb{A}_\mathrm{T}}\ f))) \\
\equiv\ \{\text{-left identity + } \mathbf{case}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (m\ \gg\!=_m\ \lambda a \to \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathit{lift} \circ f\ a)) \\
\equiv\ \{\text{-unfold } \mathit{lift}\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ (m\ \gg\!=_m\ \lambda a \to \\
\quad \mathit{un}\mathbb{A}_\mathrm{T}\ (\mathbb{A}_\mathrm{T}\ (f\ a\ \gg\!=_m\ \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a)))) \\
\equiv\ \{\text{-}\mathit{un}\mathbb{A}_\mathrm{T} \circ \mathbb{A}_\mathrm{T} \equiv \mathit{id} \text{ + } \eta\text{-reduction + assoc. of } \gg\!=_m\text{ -}\} \\
\mathbb{A}_\mathrm{T}\ ((m\ \gg\!=_m\ f)\ \gg\!=_m\ \lambda a \to \mathit{return}_m\ (\mathit{Return}\ a)) \\
\equiv\ \{\text{-fold } \mathit{lift}\text{ -}\} \\
\mathit{lift}\ (m\ \gg\!=_m\ f)
\end{array}
$$

B. $run\mathbb{A}_{\mathbf{T}}$ is a Monad Morphism

B.1 Identity preservation

$run\mathbb{A}_{\mathrm{T}}\ aenv \circ return_{\mathbb{A}_{\mathrm{T}}}$
$\equiv$ {-unfolding $return_{\mathbb{A}_{\mathrm{T}}}$ -}
$run\mathbb{A}_{\mathrm{T}}\ aenv \circ \mathbb{A}_{\mathrm{T}} \circ return_m \circ Return$
$\equiv$ {-unfolding $run\mathbb{A}_{\mathrm{T}}$ -}
$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \circ return_m \circ Return\ \gg\!=_m\ go$
$\equiv$ {-$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id$ + left identity -}
$go \circ Return$
$\equiv$ {-evaluation -}
$return_m$

B.2 Compositionality

See Figure 4.

B.3 $run\mathbb{A}_{\mathbf{T}}$ is left inverse of $lift$

$run\mathbb{A}_{\mathrm{T}}\ aenv\ (lift\ m)$
$\equiv$ {-unfold *lift* -}
$run\mathbb{A}_{\mathrm{T}}\ aenv\ (\mathbb{A}_{\mathrm{T}}\ (m\ \gg\!=_m \lambda a \rightarrow return_m \circ Return\ a))$
$\equiv$ {-unfold $run\mathbb{A}_{\mathrm{T}}$ -}
$un\mathbb{A}_{\mathrm{T}}\ (\mathbb{A}_{\mathrm{T}}\ (m\ \gg\!=_m \lambda a \rightarrow return_m \circ Return\ a))\ \gg\!=_m\ go$
$\equiv$ {-$un\mathbb{A}_{\mathrm{T}} \circ \mathbb{A}_{\mathrm{T}} \equiv id$ + associativity of $\gg\!=_m$ -}
$m\ \gg\!=_m \lambda a \rightarrow (return_m \circ Return\ a\ \gg\!=_m\ go)$
$\equiv$ {-left identity + evaluating go -}
$m\ \gg\!=_m\ return_m$
$\equiv$ {-right identity -}
m
$\equiv$ {-fold id -}
$id\ m$