Taming aspects with monads and membranes

Ismael Figueroa, Nicolas Tabareau, Éric Tanter

Taming Aspects with Monads and Membranes

Ismael Figueroa
PLEIAD Lab & ASCOLA Group
DCC University of Chile & INRIA
ifiguero@dcc.uchile.cl

Nicolas Tabareau
ASCOLA Group
INRIA, France
nicolas.tabareau@inria.fr

Éric Tanter *
PLEIAD Lab
DCC University of Chile
etanter@dcc.uchile.cl

Abstract

When a software system is developed using several aspects, special care must be taken to ensure that the resulting behavior is correct. This is known as the *aspect interference problem*, and existing approaches essentially aim to detect whether a system exhibits problematic interferences of aspects.

In this paper we describe how to control aspect interference by construction by relying on the type system. More precisely, we combine a monadic embedding of the pointcut/advice model in Haskell with the notion of membranes for aspect-oriented programming. Aspects must explicitly declare the side effects and the context they can act upon. Allowed patterns of control flow interference are declared at the membrane level and statically enforced. Finally, computational interference between aspects is controlled by the membrane topology. To combine independent and reusable aspects and monadic components into a program specification we use *monad views*, a recent technique for conveniently handling the monadic stack.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Design

Keywords aspect-oriented programming, monads, monad views, membranes, non-interference.

1. Introduction

Aspect-oriented programming (AOP) is a programming paradigm to modularize *crosscutting concerns*. We focus on the pointcut/advice model of AOP [6]. Pointcuts are predicates over *join points*, points of interest raised during program execution, to decide application of advice. Advice is an action to be applied when its corresponding pointcut matches. Aspects are modular units that encompass a number of pointcuts and advice.

Although the benefits of aspects are well understood in isolation, there is a substantial body of research devoted to the *aspect interference problem* [2–4, 8, 10, 15, 16] (also known as the *shared join point problem*). Loosely speaking, two aspects interfere if they modify a shared context, or if there is a control flow dependency between them.

Controlling the interactions between aspects is important, since the correctness of the system may be affected by a seemingly harmless change to the system. Existing approaches aim to *detect* situations of aspect interference in AspectJ-like languages. One technique is to translate a program into a state machine which is model-checked for correctness [3]. Another approach relies on a special-purpose control- and data-flow analysis to determine interference [8].

In this work we provide a mechanism[1] where programmers must use types to specify the allowed interactions between aspects and components – in contrast to post-hoc detection as in other approaches. The main feature of this mechanism is that the type-checker only accepts programs that satisfy the specified interaction constraints between aspects and components.

More specifically, our approach combines (i) a monadic embedding of the pointcut/advice model in Haskell [11]; (ii) a type-based mechanism for reasoning about effects using monads [7]; and (iii) the model of membranes for AOP [14]. The monadic setting requires aspects to explicitly declare in their type the effects that they are able to perform. Allowed patterns of control flow interference are declared at the membrane level and statically enforced. Finally, computational interference between aspects is controlled by the membrane topology. In addition, to enable the development of independent and reusable aspects and monadic components, we use *monad views* [9], a recently developed technique for handling the monadic stack. In summary, the contributions of this paper are:

- Extending the implementation of the Haskell AOP library to use monad views.
- Developing a modular language extension [11] to provide membrane-based AOP semantics to the Haskell AOP library.
- An example of how to use the extended language to control aspect interference relying only on the Haskell type system[2].

We consider this work as a proof of concept of how to combine monads with membranes to control aspect interference. Future work will focus on a more rigorous formalization of these techniques with relation to aspect interference.

The paper starts with an overview of the Haskell AOP library (Sections 2 and 4), monadic programming (Section 3), and membranes for AOP (Section 5). Then we illustrate our approach with a simple example (Section 6). Familiarity with Haskell and monadic programming is not strictly required to grasp the essence of this paper, but it would surely help to understand all the details. For a brief overview of monadic programming see [9].

2. Aspects *à la* Haskell

The embedding of aspects in Haskell [11] consists of a small standard Haskell library with full support for the pointcut/advice model. Aspects, pointcuts, and advice are statically typed, and the (unmodified) Haskell type system proves that all pointcut/advice bindings are safe. In addition to being typed, the embedding is also

* É. Tanter is partially funded by FONDECYT project 1110051.
I. Figueroa is funded by a CONICYT-Chile Doctoral Scholarship.

[1] Full implementation and example available at http://pleiad.cl/haskellaop

[2] We use several GHC extensions, which are summarized in the project website.

```
data Jp m a b = Jp (a -> m b) a
data PC m a b = Monad m => PC (∀ a' b'. m (Jp m a' b' -> m Bool))
type Advice m a b = (a -> m b) -> a -> m b
data Aspect m a b c d = (LessGen (a -> b) (c -> m d)) =>
  Aspect (PC m a (m b)) (Advice m c d)

 the Aspect data constructor is hidden, we use a function instead
aspect pc adv = Aspect pc adv newAspectHandle

 advice:
ensurePos proceed n = proceed (abs n)

 monadic version of sqrt:
sqrtM n = return (sqrt n)

 using an aspect:
program :: Int -> (𝔸𝕆T 𝕀) Int
program n = do deploy (aspect (pcCall sqrtM) ensurePos)
               sqrtM # n
```

Figure 1: Haskell AOP Join Point Model and a small example of open function application (adapted from [11]).

monadic. This means that aspects must be explicit in their types about the side effects they perform, and that we can use standard monadic reasoning techniques.

The main premise to introduce aspects into functional programming is that functions must be subject to aspect weaving. We use the term *open function application* to refer to a function application that emits a join point, opening it to weaving.

2.1 Join Point Model

The pointcut/advice model is comprised of join points, pointcuts, advice, and aspects. In Figure 1 we summarize these elements and present a small example. A join point represents a function application and contains a function of type `a -> m b` and the argument of type `a`. A pointcut is a predicate on join points, which is parameterized by a monad `m`, denoting the underlying computational effects. The predicate is parametric with respect to join points, and has access to the effects in `m`. The types `a` and `b` are not used in the right-hand side, but are required for type safety [11].

An advice is a function that executes in place of a join point matched by a pointcut. It receives a function, known as `proceed`, and returns a new function of the same type, which not necessarily applies `proceed` internally. An aspect is a first-class value which comprises a pointcut and an advice, and ensures that the binding is safe using the `LessGen` constraint, which requires the type of the pointcut to be *less general* than the type of the advice.

The example shows how the `ensurePos` advice guarantees that all calls to its advised function are made with a positive argument. `pcCall` is a predefined pointcut that matches all open applications of its argument. In addition, we provide a similar pointcut `pcType` based on type signatures, and logical pointcut combinators `pcAnd`, `pcOr` and `pcNot`; user-defined pointcuts are also allowed.

As it can be seen, aspects are deployed using `deploy`, and open function application is done using `#`, that is `sqrtM # n` is woven while `sqrtM n` is not. The $\mathbb{AO}_\mathrm{T}$ monad transformer (see next section) adds the *aspect-weaving* effect, which allows aspect deployment and open function applications.

3. An Overview of Monadic Programming

Monads are a popular mechanism to embed and reason about computational effects in purely functional languages like Haskell. Monad transformers [5] are type constructors that are used as building blocks to modularly create a monad that combines several effects. Each transformer of the composed monad builds a new effect, and has access to the effects of the underlying monad. A monad constructed with monads transformers is called *monad stack*.

In addition to the $\mathbb{AO}_\mathrm{T}$ transformer[3], some standard monad transformers used in this paper are the *state monad transformer* $\mathbb{S}_\mathrm{T}$, which threads a value with read-write access into a computation; the *reader monad transformer* $\mathbb{R}_\mathrm{T}$, which provides read access to an environment to enable context-dependent computations; and the *writer monad transformer* $\mathbb{W}_\mathrm{T}$ which provides write-only access to an auxiliar output value that is accumulated over time (e.g. a log). We also use the identity monad $\mathbb{I}$, which has no computational effect and serves as the base of a monadic stack.

3.1 Polymorphism on the Monadic Stack

Monadic components can impose constraints on the monadic stack, using type classes. For instance, if we need state-monad-like access to an integer value we declare the $\mathbb{S}_\mathtt{M}$ `Int m` constraint on `m`. These class constraints can be seen as *families of monads* and imply that the monadic component is polymorphic with respect to the concrete monadic stack, as long as the requirements are satisfied. In this paper we use the $\mathbb{S}_\mathrm{M}$, $\mathbb{R}_\mathrm{M}$, and $\mathbb{W}_\mathrm{M}$ type class constraints, which abstract the behavior of the corresponding monad transformers. The ability to deploy aspects and perform open function applications is abstracted in the $\mathbb{AO}_\mathrm{M}$ constraint.

3.2 Handling the Monadic Stack

A benefit of using monads is that they present an abstract interface over arbitrary computational effects, which allows for (more) reusable software components. However, when using monad transformers and constraints on the monad stack it is necessary to establish a mechanism to access the effects of each layer. We now briefly describe current mechanisms; for a detailed description see [9].

Explicit Lifting The most primitive solution is to use explicit lifting operations (which are required to define a monad transformer). This however is not convenient for reusability, since each component is coupled (and limited) to a concrete monadic stack.

Implicit Lifting This mechanism relies on type classes and on the polymorphism of the monadic stack to automatically route the operations to the first layer of the monadic stack that satisfies a constraint. For instance, if `m` is an instance of $\mathbb{S}_\mathrm{M}$, then $\mathbb{W}_\mathrm{T}$ `m` is also an instance of $\mathbb{S}_\mathrm{M}$ which implicitly lifts the state-related operations to `m`. This is the mechanism currently used in Haskell and in [11].

Monad Views Although implicit lifting is flexible and convenient to use, it is impractical when combining multiple instances of the same monad transformer because the implicitly lifted operation is always performed on the first layer that satisfies the constraint. Thus, lower layers with the same effect cannot be accessed and must be explicitly lifted.

Monad views [9] are a novel mechanism to *virtualize* the monad stack. Using views, a programmer can selectively ignore layers of the monadic stack, such that implicit lifting is redirected to the desired remaining layers. In addition, a view can impose restrictions on particular layers – for instance a state layer can be *seen* as a reader layer, providing read-only access to an otherwise writeable component.

In this work we extended our implementation [11] to use monad views to control data-flow interference between aspects, by precisely defining which layers are accessible to aspects (note however that all the described mechanisms can co-exist in a system).

[3] Observe that the implementation of the $\mathbb{AO}_\mathrm{T}$ and the $\mathbb{MB}_\mathrm{T}$ (Section 5) monad transformers is identical to that of the standard state monad transformer – we just give them names that reflect their purpose.

4. A First Approach to Interference Control

Initial work regarding aspect interference is done in [11]. In particular, we adapted two techniques from EffectiveAdvice [7]: control flow combinators, and parametricity on the monadic stack.

Control Flow Combinators. The control flow combinators defined in [7] correspond to the categories described by Rinard *et al.* [8]. These combinators can be directly applied in the Haskell AOP library. The categories of control flow behavior are:

- **Augmentation advice** calls `proceed` exactly once and does not modify the arguments or the return value of `proceed`.
- **Narrowing advice** calls `proceed` at most once, and does not modify the arguments or the return value of `proceed`.
- **Replacement advice** completely replaces the join point, and never calls `proceed`.
- **Combination advice** can call `proceed` any number of times with modified arguments or return value.

It could be argued that this classification could be improved. For instance, the advice in the example of Figure 1 can only be implemented as a combination advice, which is too unrestricted. Indeed, in the Haskell AOP library it is possible to define arbitrary *advice combinators* to enforce new kinds of advice [11], for instance advice that is like augmentation advice but can modify the arguments of `proceed`.

Parametricity on the Monadic Stack. We used the technique of parametricity on the monadic stack [7] to define *non-interfering* advice, pointcuts, and programs. That is, we split the monadic stack into two parts: effects available to base computation, and effects available to aspect computation. For instance, a non-interfering advice is defined as

```
type NIAdvice t a b = ∀ m. Monad (t m)
                    ⇒ Advice (NIAOT t m) a b
```

where `t` is the upper part of the split monadic stack, available to aspects; and `m` is the lower part of the split stack, available to base computation. $\mathbb{NIAO}_\mathrm{T}$ is a variant of the $\mathbb{AO}_\mathrm{T}$ transformer that supports a split monadic stack. Since an `NIAdvice` has to be parametric with respect to `m`, it effectively cannot access any effects available to base programs.

The technique of parametricity on the monadic stack can be extended to hide arbitrary layers of the monad stack, but components are harder to write and they may become coupled to a particular stack shape. This is why we use monad views [9], which are more flexible and convenient for independent development of reusable components.

5. Membranes for AOP

Execution Levels and Topological Scoping. A particular instance of the aspect interference problem is that of *computational interference*: when the evaluation of an aspect (either pointcut or advice) produces join points that are visible to others. A specific case is self-computational interference—when an aspect advises its own computation—which results in infinite regression.

To address this situation, Tanter proposed the notion of *execution levels* [12]. In execution levels, execution is stratified in a tower in which the flow of control navigates. Given an initial level, join points are always emitted one level above the current level. Aspects are deployed at a specific level and can only affect join points emitted one level below. This way, computation performed by aspects is not observable by aspects that are deployed at the same level, but only by aspects deployed one level above. In addition, the original computation, which corresponds to the last call to `proceed` in the advice chain, is always executed at the level at which the join point was emitted.

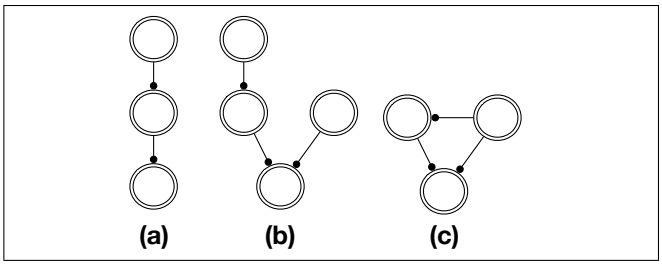


Figure 2: Topological scoping. (a) tower—corresponds to execution levels; (b) tree; (c) DAG. The lollypop arrow denotes the advising relation between membranes (adapted from [14]).

Execution levels can be used to avoid computational interference between different aspects, for instance to reuse off-the-shelf dynamic analyses aspects [13]. The form of *topological scoping* that execution levels provide is very interesting, however, the imposed topology—a tower—is too restrictive.

Flexible Topological Scoping of Aspects. The idea of membranes for AOP was born from the need to generalize execution levels to different topologies. A membrane encapsulates some computation. An operator is provided to spawn computation inside a membrane. The computation that happens inside a membrane produces join points that are only visible to the aspects registered in the *advising membranes*, *i.e.* membranes that have been explicitly bound to the current membrane. By binding membranes, it is possible to express flexible topologies of computation, because they directly define a directed graph, called the *membrane topology*, with membranes as vertices, and the advising relation as edges. Figure 2 shows three interesting topologies: a tower, which corresponds to execution levels; a tree, and a DAG. We will exploit the tree topology in the example in Section 6.

Because an aspect is registered in a given membrane, its computation (pointcuts, advice) happens inside that membrane, and is only visible to the advising membranes. Infinite regression can therefore happen only if the membrane topology is cyclic.

Beyond Topological Scoping. So far our description of membranes has focused on the topological scoping of aspects, because it is all we need for this work. Topological scoping does not make use of the programmable nature of membranes. Programmability of membranes is exploited in [14] to enforce certain properties on join point emission and reception. For instance, it is possible to encode the restrictions of the control flow combinators described in Section 4, albeit dynamically instead of statically. Here, we are interested in *static*, type-based reasoning about interference.

Bringing Membranes to Haskell AOP. In this work we implemented the $\mathbb{MB}_\mathrm{T}$ monad transformer as a modular language extension [11] to the Haskell AOP library to provide membrane-based AOP. Any monadic stack with the form $\mathbb{AO}_\mathrm{T}$ $\mathbb{MB}_\mathrm{T}$ `m` provides the membrane-related operations `newMembrane` (creates a new membrane), `register` (registers an aspect in a membrane), `advise` (binds two membranes), and `evalIn` (evaluates a computation in a given membrane).

6. An Illustration of Interference Control

We illustrate how to control aspect interference extending an example from [14]. Consider a web browser application, which processes URLs into HTML documents. To enhance the performance of the browser, an independent cache aspect is used. In addition, to control the memory consumption of both the browser and the

```
1  type CacheState a b = Map a b
2  type Cache = CacheState URL HtmlDoc
3  type Log = String
4  type HtmlDoc = String
5  type History = [(URL, UTCTime)]
6
7   Browser
8  browser :: 𝕊_M History m ⇒ URL → m HtmlDoc
9  browser url = do
10  history ← get
11  currentTime ← getCurrentTime
12  put ((url, currentTime):history)
13  return (getDocumentByUrl url)
14
15  Quota Advice
16 quotaAdv :: (ℝ_M s m, 𝕎_M Log m)
17          ⇒ String → (s → Int) → Int → Augmentation m a b ()
18 quotaAdv name sizeOf quota = Augmentation (augBef, augAft) where
19  augBef _     = return ()
20  augAft _ _ _ = do s ← ask;
21    if sizeOf s > quota
22       then tell ("[Q: " ++ name ++ "] Quota Exceeded...")
23       else return ()
24
25  Cache Advice
26 cacheAdv :: (Ord a, 𝕊_M (CacheState a b) m, 𝔸𝕆_M m)
27          ⇒ Narrowing m a b ()
28 cacheAdv = Narrowing (p, (augBef, augAft), repl) where
29  p url       = do {cache ← get; return (not (member url cache))}
30  augBef _          = return ()
31  augAft url page _ = cachePut # (url, page)
32  repl url          = do {cache ← get; return (cache ! url)}
33
34  cachePut allows to expose caching operations to advice
35 cachePut :: (Ord a, 𝕊_M (CacheState a b) m) ⇒ (a,b) → m ()
36 cachePut (a,b) = do {cache ← get; put (insert a b cache)}
```

Figure 3: Independent Components: browser, quota advice, and cache advice. Each component imposes its own constraints on the monadic stack to perform its functionality.

cache, two quota aspects are used. We model each part of this system as an independently developed component (Figure 3), and compose them controlling their control- and data-flow interference (Figure 4).

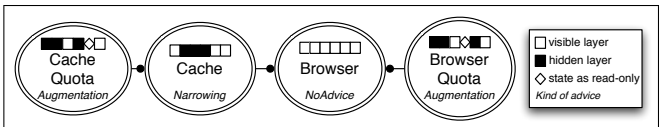
Browser. The `browser` (lines 8-13) retrieves an `HtmlDoc` associated to a given `url`, and updates an internal history with the time of access to the url[4]. In the composed program, `browserSession` simulates the use of the browser providing several urls. This component requires read-write access to a `History` state, as expressed by the $\mathbb{S}_M$ `History m` constraint in its signature.

Quota Advice. This is a generic quota advice (lines 16-23) which requires read-only access to an arbitrary state `s` (the $\mathbb{R}_M$ `s m` constraint), and write-access to a `Log` (the $\mathbb{W}_M$ `Log m` constraint). The advice receives a function to compute the size of `s`, and reports to the log using the given `name` when the quota is exceeded. Of course, a more realistic quota advice should raise an exception or abort the operation before it happens, but we use a simple approach for our example. This advice is defined as an augmentation advice, which is composed of a before (`augBef`) and an after advice (`augAft`). The single call to `proceed` between these two pieces of advice is mandatory and implicit[5].

Cache Advice. This advice (lines 27-36) is a general purpose cache, which uses a dictionary to store already computed results.

[4] `getCurrentTime` requires the $\mathbb{IO}_M$ constraint, but we omit it for simplicity.

[5] In general `augBef` can pass a value to `augAft`, but we just pass the unit value `()`. That is why the advice type is `Augmentation m a b ()`. The same happens in the augmentation part of `cacheAdv`.




```
1   The true monadic stack
2  type M = 𝔸𝕆_T (𝕄𝔹_T (𝕎_T Log (𝕊_T History (𝕊_T Cache 𝕀))))
3
4  program :: M ()
5  program = do
6   let cacheView   = □ ::: ■ ::: ■ ::: ■ ::: □ ::: □
7   let brQtView    = ■ ::: ■ ::: □ ::: ◇ ::: ■ ::: □
8   let cacheQtView = ■ ::: ■ ::: □ ::: ■ ::: ◇ ::: □
9   browserMb ← newMembrane NoAdvice         idv
10  cacheMb   ← newMembrane NarrowingAdv     cacheView
11  brQtMb    ← newMembrane AugmentationAdv brQtView
12  cacheQtMb ← newMembrane AugmentationAdv cacheQtView
13  register (pcCallGlobal cacheView browser) cacheAdv cacheMb
14  register (pcCallGlobal brQtView browser)
15           (quotaAdv "Browser Quota" length 2) brQtMb
16  register (pcCallView cacheView cacheQtView cachePut)
17           (quotaAdv "Cache Quota" length 2) cacheQtMb
18  advise cacheMb browserMb
19  advise browserQuotaMb browserMb
20  advise cacheQuotaMb cacheMb
21  evalIn browserMb (browserSession ["www.example.com", ...])
```

Figure 4: Membrane topology of the composed program. A membrane can only contain aspects that conform to its *view* and its control flow combinator (adapted from [14]).

The component requires read-write access to a `CacheState` value (the $\mathbb{S}_M$ `CacheState m` constraint[6]), and it is defined as a narrowing advice. This means that its behavior is specified as a predicate `p`, an augmentation advice (`augBef, augAft`), and a replacement advice `repl`. When the advice is evaluated, if the predicate is true the augmentation advice is applied, otherwise the implicit call to `proceed` is replaced by `repl`.

Observe that we need to expose the operation of the cache advice to aspect weaving. To do it we define the `cachePut` function and perform an open application of it in the body of `augAft`. This requires us to add the $\mathbb{AO}_M$ constraint to `cacheAdv`.

6.1 Control Flow Interference

Control flow interference between the browser, the cache aspect and the quota aspects can occur if (i) each aspect can *see* join points emitted by the other, and (ii) the application of one aspect depends on how many times the other aspect calls `proceed`. To control join point visibility we establish the membrane topology shown in Figure 4, registering each aspect in its own membrane (lines 9-20 in the code). In this topology the cache aspect can only observe the browser, and each quota aspect can only observe its intended target. Observe that in a language like AspectJ it is not clear how to model our example without making aspects aware of each other, because aspects can see all join points.

The evaluation of every aspect (and the browser) is dependent on how many times some other component calls `proceed`. In a non-interfering scenario, each quota aspect has to call `proceed` exactly once, keeping the original arguments and return value. Also, the cache advice has to either call `proceed` with the same restrictions, or retrieve a value from the dictionary.

To enforce these restrictions, each membrane is created by specifying a tag that represents a control flow behavior, according to Rinard *et al.*'s categories described in Section 4. Thus the cache membrane only accepts aspects with narrowing advice (line 10); and the quota membranes only accept aspects with augmentation advice

[6] The `Ord a` constraint is because the keys of a dictionary must be ordered.

(lines 11 and 12). In addition, we use the special `NoAdvice` tag to specific that the browser membrane does not accept aspects (line 9). Concretely, the aspect registration function `register` is overloaded, and the tag is used to choose the appropriate implementation that statically enforces the corresponding control flow interference pattern.

6.2 Data Flow Interference

The components of the system are defined on a constrained but abstract monadic stack. To actually use these components we need to define a concrete monadic stack with the proper requirements. In our example (Figure 4) we use the stack `M` (line 2), which provides effects for aspect weaving with membrane semantics ($\mathbb{AO}_\mathtt{T}$ $\mathbb{MB}_\mathtt{T}$), write access to a `Log` value ($\mathbb{W}_\mathtt{T}$ `Log`, as required by the quota advice), and read-write access to a `History` and a `Cache` value ($\mathbb{S}_\mathtt{T}$ `History` and $\mathbb{S}_\mathtt{T}$ `Cache`, as required by the browser and the cache advice).

Interference Problems If all the components share the full monadic stack the cache component cannot work because, due to implicit lifting, the $\mathbb{S}_\mathtt{T}$ `Cache` layer is hidden. Thus the `get` and `put` operations on the advice will be redirected to the $\mathbb{S}_\mathtt{T}$ `History` layer, producing a typechecking error. This problem can be solved by using explicit lifting in the cache advice, but this solution breaks the premise of independently developed components, because the cache advice must be aware of other aspects.

A more serious problem is that state layers are accessible to every component. For instance, if the developer adds a $\mathbb{S}_\mathtt{M}$ `History m` constraint to the quota advice, then the history of the browser can be modified by any quota aspect. Eventually any component could write to the log, or even deploy aspects. It could be argued that this is not the case in our example since the constraints on each component are minimal – however the problem is that it is *possible* to use *some* component that interferes with the behavior of the others.

Solution Using Monad Views Our solution is based on the premise that a component must only *see* what it actually requires from the monadic stack – and nothing else. To this end, a membrane receives as a second argument a *monad view* (Section 3.2), which maps between `M` and the virtual stack given to aspects inside the membrane.

A view is constructed using a *structural mask*[7], where it is specified from left to right whether the respective layer is visible or not. The mask is created by concatenating unary masks for each layer: □ indicates a visible layer, ■ a hidden layer, and ◊ a read-only layer (must originally be a state layer)[8]. For the browser membrane we use the identity view `idv`, which yields `M`.

Extending the Pointcut Language Observe that the cache advice is triggered by calls to `browser` detected by a pointcut, and that for type safety reasons the monad stack associated to the pointcut and the advice must be the same [11].

Hence it is not possible to register an aspect that applies the cache advice on calls to the browser using only the predefined `pcCall` pointcut (Section 2), because its type is `(a → m b) → PC m a b` [11]. That is, the function must see the same monad as the resulting pointcut – but `browser` sees the full stack, while `cacheAdv` sees a restricted monad. Moreover, even if they see the same virtual stack the browser cannot access its required state layer, which is hidden. To solve this issue we provide a new pointcut, `pcCallGlobal`, with type

```
n ⋈ AO_T MB_T m → (a → AO_T MB_T m b) → PC n a (n b)
```

where `m` and `n` are monads, and $\mathbb{AO}_\mathtt{T}$ $\mathbb{MB}_\mathtt{T}$ `m` is any stack with membrane semantics for weaving. The type `n` $\bowtie$ $\mathbb{AO}_\mathtt{T}$ $\mathbb{MB}_\mathtt{T}$ `m` denotes any monad view that maps between a full stack $\mathbb{AO}_\mathtt{T}$ $\mathbb{MB}_\mathtt{T}$ `m` and a restricted view `n`. Therefore with `pcCallGlobal` it is possible to advise calls to functions that see the full monadic stack with advice on a restricted monad.

A `pcCallGlobal` pointcut is non-interfering because it is *pure*, *i.e.* there are no side-effecting operations on the full stack. This is important because the typing issue could be solved by lifting a `pcCall` pointcut to the proper type using views – but then we could not ensure non-interference in general if the pointcut is composed using pointcut combinators. For instance, one could lift a pointcut that combines `pcCall` with a pointcut that writes to a layer of the full stack that is unavailable in the restricted view, without type errors.

Finally, a more general instance of the same typing issues arises when registering the cache quota aspect, which we solve with the `pcCallView` pointcut, with type

```
l ⋈ AO_T MB_T m → n ⋈ AO_T MB_T m → (a → l b) → PC n a (n b)
```

that takes into account a function and two views, `l` and `n`, which correspond to the view seen by the function, and the view of the resulting pointcut, respectively; and yields a proper pointcut, in the same way as `pcCallGlobal`.

7. Conclusions and Future Work

We have shown a complete example of how to control aspect interference using monadic reasoning about effects, control flow combinators, and membranes. Non-interference is directly enforced by the type system, not by external analysis tools as in other approaches. We believe this system is a straightforward approach to specify and enforce by construction the allowed interactions not only between aspects, but also between aspects and components in a system.

A rigorous formalization of interference scenarios and how they are managed by our system is an immediate line of future work. It would also be interesting to perform some case studies to assess the capabilities of our framework in practice.

Acknowledgements

This work was supported by the INRIA Associated team RAPIDS. We also thank the anonymous reviewers for their valuable recommendations.

References

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

[2] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proc. of the nternational Workshop on Foundations of Aspect-Oriented Languages (FOAL 2002)*, 2002.

[3] C. Disenfeld and S. Katz. A closer look at aspect interference and cooperation. In *Proceedings of the 11th International Conference on Aspect-oriented Software Development (AOSD)*, 2012.

[4] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[5] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symposium on Prin-*

[7] There are also *nominal masks* where each layer is labeled with a name. For now we only consider structural masks. For more details see [9].

[8] This graphical representation of views is adapted from the original proposal [9], and is intended to ease the understanding of how masks work. The corresponding code can be found in the implementation.

ciples of Programming Languages (POPL 95), pages 333–343, San Francisco, California, USA, Jan. 1995. ACM Press.

[6] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[7] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [1], pages 109–120.

[8] 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.

[9] T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*, pages 32–44, Tokyo, Japan, Sept. 2011. ACM Press.

[10] M. Störzer, R. Sterr, and F. Forster. Detecting precedence-related advice interference. In *Proc. of the 21st International Conference on Automated Software Engineering (ASE 2006)*, 2006.

[11] 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)*, Fukuoka, Japan, Mar. 2013. ACM Press.

[12] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [1], pages 37–48.

[13] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[14] É. Tanter, N. Tabareau, and R. Douence. Taming aspects with membranes. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 3–8, Potsdam, Germany, Mar. 2012. ACM Press.

[15] K. Tian, K. Cooper, K. Zhang, and S. Liu. Towards a new understanding of advice interference. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2010.

[16] K. Tian, K. Cooper, K. Zhang, and H. Yu. A classification of aspect composition problems. In *Proceedings of the 3th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, 2009.