Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice

Ismael Figueroa, Nicolas Tabareau, Éric Tanter

Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice

Ismael Figueroa[1,2⋆], Nicolas Tabareau[2], and Éric Tanter[1⋆⋆]

[1] PLEIAD Laboratory
University of Chile, Santiago, Chile.
{ifiguero,etanter}@dcc.uchile.cl
http://pleiad.cl
[2] ASCOLA group
INRIA – Nantes, France.
nicolas.tabareau@inria.fr

Abstract. Aspect-oriented programming (AOP) aims to enhance modularity and reusability in software systems by offering an abstraction mechanism to deal with crosscutting concerns. However, in most general-purpose aspect languages aspects have almost unrestricted power, eventually conflicting with these goals. In this work we present Effective Aspects: a novel approach to embed the pointcut/advice model of AOP in a statically-typed functional programming language like Haskell. Our work extends EffectiveAdvice, by Oliveira, Schrijvers and Cook; which lacks quantification, and explores how to exploit the monadic setting in the full pointcut/advice model. Type soundness is guaranteed by exploiting the underlying type system, in particular phantom types and a new anti-unification type class. Aspects are first-class, can be deployed dynamically, and the pointcut language is extensible, therefore combining the flexibility of dynamically-typed aspect languages with the guarantees of a static type system. Monads enables us to directly reason about computational effects both in aspects and base programs using traditional monadic techniques. Using this we extend Aldrich's notion of Open Modules with effects, and also with protected pointcut interfaces to external advising. These restrictions are enforced statically using the type system. Also, we adapt the techniques of EffectiveAdvice to reason about and enforce control flow properties. Moreover, we show how to control effect interference using the parametricity-based approach of EffectiveAdvice. However this approach falls short when dealing with interference between multiple aspects. We propose a different approach using *monad views*, a recently developed technique for handling the monad stack. Finally, we exploit the properties of our monadic weaver to enable the modular construction of new semantics for aspect scoping and weaving. These semantics also benefit fully from the monadic reasoning mechanisms present in the language. This work brings type-based reasoning about effects for the first time in the pointcut/advice model, in a framework that is both expressive and extensible; thus allowing development of robust aspect-oriented systems as well as being a useful research tool for experimenting with new aspect semantics.

Keywords: aspect-oriented programming, monads, pointcut/advice model, type-based reasoning, modular reasoning

⋆ Funded by a CONICYT-Chile Doctoral Scholarship
⋆⋆ Partially funded by FONDECYT project 1110051.

1 Introduction

Aspect-oriented programming languages support the modular definition of crosscutting concerns through a join point model [19]. In the pointcut/advice mechanism, crosscutting is supported by means of pointcuts, which quantify over join points, in order to implicitly trigger advice [48]. Such a mechanism is typically integrated in an existing programming language by modifying the language processor, may it be the compiler (either directly or through macros), or the virtual machine. In a typed language, introducing pointcuts and advices also means extending the type system, if type soundness is to be preserved. For instance, AspectML [7] is based on a specific type system in order to safely apply advice. AspectJ [18] does not substantially extend the type system of Java and suffers from soundness issues. StrongAspectJ [8] addresses these issues with an extended type system. In both cases, proving type soundness is rather involved because a whole new type system has to be dealt with.

In functional programming, the traditional way to tackle language extensions, mostly for embedded languages, is to use monads [27]. Early work on AOP suggests a strong connection to monads. De Meuter proposed to use them to lay down the foundations of AOP [26], and Wand *et al.* used monads in their denotational semantics of pointcuts and advice [48]. Recently, Tabareau proposed a weaving algorithm that supports monads in the pointcut and advice model, which yields benefits in terms of extensibility of the aspect weaver [38], although in this work the weaver itself was not monadic but integrated internally in the system. This connection was exploited in recent preliminary work by the authors to construct an extensible monadic aspect weaver, in the context of Typed Racket [14], but the proposed monadic weaver was not fully typed because of limitations in the type system of Typed Racket.

This work proposes Effective Aspects: a lightweight, full-fledged embedding of aspects in Haskell, that is typed and monadic.[3] By *lightweight*, we mean that aspects are provided as a small standard Haskell library. The embedding is *full-fledged* because it supports dynamic deployment of first-class aspects with an extensible pointcut language—as is usually found only in dynamically-typed aspect languages like AspectScheme [11] and AspectScript [45] (Sect. 3).

By *typed*, we mean that in the embedding, pointcuts, advices, and aspects are all statically typed (Sect. 4), and pointcut/advice bindings are proven to be safe (Sect. 5). Type soundness is directly derived by relying on the existing type system of Haskell (type classes [47], phantom types [21], and some recent extensions of the Glasgow Haskell Compiler). Specifically, we define a novel type class for anti-unification [32,33], which is key to define safe aspects.

Finally, because the embedding is *monadic*, we derive two notable advantages over ad-hoc approaches to introducing aspects in an existing language. First, we can directly

[3] This work is an extension of our paper in the 12th International Conference on Aspect-Oriented Software Development [39]. We mainly expand on using types to reason about aspect interference (Section 7). In addition, we provide a technical background about monadic programming in Haskell (Section 2). The implementation is available, with examples, at http://pleiad.cl/EffectiveAspects

reason about aspects and effects using traditional monadic techniques. In short, we can generalize the interference combinators of EffectiveAdvice [28] in the context of pointcuts and advice (Sect. 6). And also we can use non-interference analysis techniques such as those from EffectiveAdvice, and from other advanced mechanisms, in particular *monad views* [35] (Sect. 7). Second, because we embed a monadic weaver, we can modularly extend the aspect language semantics. We illustrate this with several extensions and show how type-based reasoning can be applied to language extensions (Sect. 8). Sect. 9 discusses several issues related to our approach, Sect. 10 reviews related work, and Sect. 11 concludes.

2 Prelude: Overview of Monadic Programming

To make this work self-contained and to cater to readers not familiar with monads, we present a brief overview of the key concepts of monadic programming in Haskell used throughout this paper. More precisely, we introduce the state and error monad transformers, and the mechanisms of explicit and implicit lifting in the monad stack.

The reader is only expected to know basic Haskell programming and to understand the concept of type classes. As a good tutorial we suggest [20]. Readers already familiar with monadic programming can safely skip this section.

2.1 Monads Basics

Monads [27,46] are a mechanism to embed and reason about computational effects such as state, I/O, or exception-handling in purely functional languages like Haskell. Monad transformers [22] allow the modular construction of monads that combine several effects. A monad transformer is a type constructor used to create a *monad stack* where each layer represents an effect. Monadic programming in Haskell is provided by the Monad Transformers Library (known as *MTL*), which defines a set of monad transformers that can be flexibly composed together.

A monad is defined by a type constructor m and functions $\gg\!=$ (called $bind$) and $return$. At the type level a monad is a regular type constructor, although conceptually we distinguish a value of type a from a *computation in monad* m of type $m\ a$. Monads provide a uniform interface for computational effects, as specified in the $Monad$ type class:

$$\begin{array}{l}\textbf{class}\ Monad\ m\ \textbf{where}\\ \quad return :: a \rightarrow m\ a\\ \quad (\gg\!=)\ \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\end{array}$$

Here $return$ promotes a value of type a into a computation of type $m\ a$, and $\gg\!=$ is a pipeline operator that takes a computation, extracts its value, and applies an action to produce a new computation. The precise meanings for $return$ and $\gg\!=$ are specific to each monad. The computational effect of a monad is “hidden” in the definition of $\gg\!=$, which imposes a sequential evaluation where the effect is performed at each step. To avoid cluttering caused by using $\gg\!=$ Haskell provides the **do**-notation, which directly translates to chained applications of $\gg=$. The $x \leftarrow k$ expression binds identifier x with the value extracted from performing computation k for the rest of a **do** block.[4]

A monad transformer is defined by a type constructor t and the *lift* operation, as specified in the *MonadTrans* type class:

$$\begin{array}{l}\textbf{class}\ \mathit{MonadTrans}\ t\ \textbf{where}\\ \quad \mathit{lift} :: m\ a \rightarrow t\ m\ a\end{array}$$

The purpose of *lift* 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 transformer t must declare in an effect-specific way how to make $t\ m$ an instance of the *Monad* class.

2.2 Plain Monadic Programming

To illustrate monadic programming we first describe the use of the state monad transformer *StateT*, denoted as $\mathbb{S}_\mathrm{T}$, whose computational effect is to thread a value with read-write access.

$$\begin{array}{l}\textbf{newtype}\ \mathbb{S}_\mathrm{T}\ s\ m\ a = \mathbb{S}_\mathrm{T}\ (s \rightarrow m\ (a, s))\\ \mathit{eval}\mathbb{S}_\mathrm{T} :: \mathbb{S}_\mathrm{T}\ s\ m\ a \rightarrow s \rightarrow m\ a\end{array}$$

A $\mathbb{S}_\mathrm{T}\ s\ m\ a$ computation is a function that takes an initial state of type s and returns a computation in the underlying monad m with a pair containing the resulting value of type a, and a potentially modified state of type s. The $\mathit{eval}\mathbb{S}_\mathrm{T}$ function evaluates a $\mathit{State}\ s\ m\ a$ computation using an initial state s and yields only the returning computation $m\ a$. In addition, functions $\mathit{get}\mathbb{S}_\mathrm{T}$ and $\mathit{put}\mathbb{S}_\mathrm{T}$ allow to retrieve and update the state inside a computation, respectively[5].

$$\begin{array}{l}\mathit{get}\mathbb{S}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow \mathbb{S}_\mathrm{T}\ s\ m\ s\\ \mathit{get}\mathbb{S}_\mathrm{T} \quad = \mathbb{S}_\mathrm{T}\ \$\ \lambda s \rightarrow \mathit{return}\ (s, s)\\ \mathit{put}\mathbb{S}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow s \rightarrow \mathbb{S}_\mathrm{T}\ s\ m\ ()\\ \mathit{put}\mathbb{S}_\mathrm{T}\ s' = \mathbb{S}_\mathrm{T}\ \$\ \lambda\_ \rightarrow \mathit{return}\ ((), s')\end{array}$$

Note that both functions get the current state from some previous operation ($\gg=$ or $\mathit{eval}\mathbb{S}_\mathrm{T}$). The difference is that $\mathit{get}\mathbb{S}_\mathrm{T}$ returns this value and keeps the previous state unchanged, whereas $\mathit{put}\mathbb{S}_\mathrm{T}$ replaces the previous state with its argument.

Example Application Consider a mutable queue of integers with operations to enqueue and dequeue its elements. To implement it we will define a monad stack M_1, which threads a list of integers using the $\mathbb{S}_\mathrm{T}$ transformer on top of the identity monad $\mathbb{I}$ (which has no computational effect). We also define runM_1, which initializes the queue with an empty list, and returns only the resulting value of a computation in M_1.

[4] $x \leftarrow k$ performs the effect in k, while $\textbf{let}\ x = k$ does not.

[5] Note the use of \$, here and throughout the rest of the paper, to avoid extra parentheses.

$$\textbf{type}\ M_1 = \mathbb{S}_\mathrm{T}\ [\mathit{Int}]\ \mathbb{I}$$
$$\mathit{runM}_1 :: M_1\ a \to a$$
$$\mathit{runM}_1\ c = \mathit{run}\mathbb{I}\ \$\ \mathit{eval}\mathbb{S}_\mathrm{T}\ c\ [\,]$$

The implementation of the queue operations using M_1 is simple, we just enqueue elements at the end of the list and dequeue elements from the beginning.

$$\mathit{enqueue}_1 :: \mathit{Int} \to M_1\ ()$$
$$\begin{aligned}\mathit{enqueue}_1\ n = \textbf{do}\ & \mathit{queue} \leftarrow \mathit{get}\mathbb{S}_\mathrm{T} \\ & \mathit{put}\mathbb{S}_\mathrm{T}\ \$\ \mathit{queue} +\!\!+ [n]\end{aligned}$$
$$\mathit{dequeue}_1 :: M_1\ \mathit{Int}$$
$$\begin{aligned}\mathit{dequeue}_1 = \textbf{do}\ & \mathit{queue} \leftarrow \mathit{get}\mathbb{S}_\mathrm{T} \\ & \mathit{put}\mathbb{S}_\mathrm{T}\ \$\ \mathit{tail}\ \mathit{queue} \\ & \mathit{return}\ \$\ \mathit{head}\ \mathit{queue}\end{aligned}$$

Handling Error Scenarios The above implementation of $\mathit{dequeue}_1$ terminates with a runtime error if it is performed on an empty queue, because *tail* fails when applied on an empty list. To provide an error-handling mechanism we use the error monad transformer ErrorT, denoted as $\mathbb{E}_\mathrm{T}$.

$$\textbf{newtype}\ \mathbb{E}_\mathrm{T}\ e\ m\ a = \mathbb{E}_\mathrm{T}\ m\ (\mathit{Either}\ e\ a)$$
$$\mathit{run}\mathbb{E}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow \mathbb{E}_\mathrm{T}\ e\ m\ a \to m\ (\mathit{Either}\ e\ a)$$

The type $\mathit{Either}\ e\ a$ represents two possible values: a $\mathit{Left}\ e$ value or a $\mathit{Right}\ a$ value. In this case the convention is that a $\mathit{Left}\ e$ value is treated as an error, while a $\mathit{Right}\ a$ value is considered a successful operation. Then, the $\mathit{throw}\mathbb{E}_\mathrm{T}$ and $\mathit{catch}\mathbb{E}_\mathrm{T}$ operations can be defined to raise and handle exceptions.

$$\mathit{throw}\mathbb{E}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow e \to \mathbb{E}_\mathrm{T}\ e\ m\ a$$
$$\mathit{throw}\mathbb{E}_\mathrm{T}\ e = \mathbb{E}_\mathrm{T}\ \$\ \mathit{return}\ (\mathit{Left}\ e)$$
$$\mathit{catch}\mathbb{E}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow \mathbb{E}_\mathrm{T}\ e\ m\ a \to (e \to \mathbb{E}_\mathrm{T}\ e\ m\ a) \to \mathbb{E}_\mathrm{T}\ e\ m\ a$$
$$\begin{aligned}m\ `\mathit{catch}\mathbb{E}_\mathrm{T}`\ h = \mathbb{E}_\mathrm{T}\ \$\ \textbf{do}\ & a \leftarrow \mathit{run}\mathbb{E}_\mathrm{T}\ m \\ & \textbf{case}\ a\ \textbf{of} \\ & \quad \mathit{Left}\ \mathit{err}\ \ \to \mathit{run}\mathbb{E}_\mathrm{T}\ (h\ \mathit{err}) \\ & \quad \mathit{Right}\ \mathit{val} \to \mathit{return}\ (\mathit{Right}\ \mathit{val})\end{aligned}$$

Observe that $\mathit{catch}\mathbb{E}_\mathrm{T}$ is intended to be used as an infix operator, where the first argument is the protected expression that would be inside a *try* block in Java, while the second argument is the exception handler.

Combining State and Error-Handling Effects To implement a queue with support for exceptions we first define a new monad stack M_2 that combines both effects (using *String*s as error messages):

$$\textbf{type}\ M_2 = \mathbb{S}_\mathrm{T}\ [\mathit{Int}]\ (\mathbb{E}_\mathrm{T}\ \mathit{String}\ \mathbb{I})$$
$$\mathit{runM}_2\ c = \mathit{run}\mathbb{I}\ \$\ \mathit{run}\mathbb{E}_\mathrm{T}\ \$\ \mathit{eval}\mathbb{S}_\mathrm{T}\ c\ [\,]$$

Then we define the $enqueue_2$ operation as before, but using M_2:

$$
\begin{array}{l}
enqueue_2 :: Int \to M_2\ () \\
enqueue_2\ n = \textbf{do}\ queue \leftarrow get\mathbb{S}_{\mathrm{T}} \\
\qquad\qquad\qquad\quad put\mathbb{S}_{\mathrm{T}}\ \$\ queue +\!\!+ [n]
\end{array}
$$

However, the straightforward definition of $dequeue_2$ fails with a typing error:

$$
\begin{array}{l}
dequeue_2 :: M_2\ Int \\
dequeue_2 = \textbf{do}\ queue \leftarrow get\mathbb{S}_{\mathrm{T}} \\
\qquad\qquad\qquad \textbf{if}\ null\ queue \\
\qquad\qquad\qquad\quad \textbf{then}\ throw\mathbb{E}_{\mathrm{T}}\ \texttt{"Queue is empty"}\ \ \text{-- typing error} \\
\qquad\qquad\qquad\quad \textbf{else}\ \ \textbf{do}\ put\mathbb{S}_{\mathrm{T}}\ \$\ tail\ queue \\
\qquad\qquad\qquad\qquad\qquad\quad return\ \$\ head\ queue
\end{array}
$$

The problem is that $throw\mathbb{E}_{\mathrm{T}}$ returns a computation whose type is $(\mathbb{E}_{\mathrm{T}}\ String\ \mathbb{I})\ Int$, but the return type of $dequeue_2$ is $(\mathbb{S}_{\mathrm{T}}\ [Int]\ (\mathbb{E}_{\mathrm{T}}\ String\ \mathbb{I}))\ Int$.

Explicit Lifting in the Monad Stack Using *lift* we can reuse a function intended for an inner layer on the stack, like $throw\mathbb{E}_{\mathrm{T}}$. The number of *lift* calls corresponds to the distance between the top of the stack and the inner layer of the stack. Hence for $dequeue_2$ we need only one call to *lift*:

$$
\begin{array}{l}
dequeue_2 :: M_2\ Int \\
dequeue_2 = \textbf{do}\ queue \leftarrow get\mathbb{S}_{\mathrm{T}} \\
\qquad\qquad\qquad \textbf{if}\ null\ queue \\
\qquad\qquad\qquad\quad \textbf{then}\ (lift \circ throw\mathbb{E}_{\mathrm{T}})\ \texttt{"Queue is empty"} \\
\qquad\qquad\qquad\quad \textbf{else do}\ put\mathbb{S}_{\mathrm{T}}\ \$\ tail\ queue \\
\qquad\qquad\qquad\qquad\qquad\ \ return\ \$\ head\ queue
\end{array}
$$

Although we managed to implement a queue with support for both effects, this is not satisfactory from a software engineering point of view. The reason is that plain monadic programming and explicit liftings produce a strong coupling between functions and particular monad stacks, hampering reusability and maintainability of the software.

2.3 Polymorphism on the Monad Stack

To address the coupling of functions with particular monad stacks and to expand the notion of monads as a uniform interface for computational effects, the MTL defines a set of type classes associated to particular effects. This way, monadic functions can impose constraints in the monad stack using these type classes instead of relying on a specific stack. These class constraints can be seen as *families of monads*, making a function polymorphic with respect to the concrete monadic stack used to evaluate it.

State Operations The $MonadState$ type class, denoted as $\mathbb{S}_{\mathrm{M}}$, defines the interface for state-related operations, and $\mathbb{S}_{\mathrm{T}}$ is the canonical instance of this class.[6]

[6] Expression $m \to s$ denotes a *functional dependency* [17], which means that the type of m determines the type of s, allowing a more precise control of type inference.

$$\begin{array}{l}\textbf{class } \mathit{Monad}\ m \Rightarrow \mathbb{S}_{\mathrm{M}}\ s\ m \mid m \rightarrow s \textbf{ where}\\ \quad \mathit{get} :: m\ s\\ \quad \mathit{put} :: s \rightarrow m\ ()\end{array}$$

Error-Handling Operations The $\mathit{MonadError}$ type class, denoted as $\mathbb{E}_{\mathrm{M}}$, defines the standard interface for error-handling operations, with $\mathbb{E}_{\mathrm{T}}$ as its canonical instance.

$$\begin{array}{l}\textbf{class } \mathit{Monad}\ m \Rightarrow \mathbb{E}_{\mathrm{M}}\ e\ m \mid m \rightarrow e \textbf{ where}\\ \quad \mathit{throwError} :: e \rightarrow m\ a\\ \quad \mathit{catchError} :: m\ a \rightarrow (e \rightarrow m\ a) \rightarrow m\ a\end{array}$$

Implicit Lifting in the Monad Stack Going back to our example of the integer queue, the implementation using class contraints now is as follows:

$$\begin{array}{l}\mathit{enqueue} :: (\mathit{Monad}\ m, \mathbb{S}_{\mathrm{M}}\ [\mathit{Int}]\ m) \Rightarrow \mathit{Int} \rightarrow m\ ()\\ \mathit{enqueue}\ n = \textbf{do}\ \mathit{queue} \leftarrow \mathit{get}\\ \qquad\qquad\qquad\quad \mathit{put}\ \$\ \mathit{queue} +\!\!+ [n]\\ \mathit{dequeue} :: (\mathit{Monad}\ m, \mathbb{S}_{\mathrm{M}}\ [\mathit{Int}]\ m, \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ m) \Rightarrow m\ \mathit{Int}\\ \mathit{dequeue} = \textbf{do}\ \mathit{queue} \leftarrow \mathit{get}\\ \qquad\qquad\qquad \textbf{if}\ \mathit{null}\ \mathit{queue}\\ \qquad\qquad\qquad\quad \textbf{then}\ \mathit{throwError}\ \texttt{"Queue is empty"}\\ \qquad\qquad\qquad\quad \textbf{else do}\ \mathit{put}\ \$\ \mathit{tail}\ \mathit{queue}\\ \qquad\qquad\qquad\qquad\qquad \mathit{return}\ \$\ \mathit{head}\ \mathit{queue}\end{array}$$

Observe that the functions are defined in terms of an abstract monad m, which is required to be an instance of $\mathbb{S}_{\mathrm{M}}$, for insertions; and both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$ for retrieving values. Also note that *lift* is not required to use $\mathit{throwError}$ in $\mathit{dequeue}$. The reason is that using type classes, like $\mathbb{S}_{\mathrm{M}}$ or $\mathbb{E}_{\mathrm{M}}$, an operation is automatically routed to *the first layer of the monad stack that is instance of the respective class*. The MTL defines implicit liftings between its transformers, by defining several class instances for each of them. Because of this, M_2 is instance of both $\mathbb{S}_{\mathrm{M}}$ and $\mathbb{E}_{\mathrm{M}}$.

The major limitation of implicit liftings is that it *only* chooses the first layer of a given effect. Consequently, when more than one instance of the same effect are used, *e.g.* two state transformers to hold the state of a queue and a stack, the parts of the program that access inner layers must use explicit lifting.

Explicit and implicit lifting are the standard mechanism in Haskell to handle the monad stack. The mechanism used to handle the monad stack directly determines the expressiveness of the type-based reasoning techniques, and other properties like modularity and reusability of components. This is discussed in detail in Sect. 6 and 7; in particular we show that the standard mechanism falls short to deal with interference of multiple aspects. Then we use monad views, a recent mechanism for managing the monad stack developed by Schrijvers and Oliveira [35], to propose another approach to address this situation.

3 Introducing Aspects

The fundamental premise for aspect-oriented programming in functional languages is that function applications need to be subject to aspect weaving. We introduce the term *open application* to refer to a function application that generates a join point, and consequently, can be woven.

Open Function Applications Opening all function applications in a program or only a few selected ones is both a language design question and an implementation question. At the design level, this is the grand debate about *obliviousness* in aspect-oriented programming. Opening all applications is more flexible, but can lead to fragile aspects and unwanted encapsulation breaches. At the implementation level, opening all function applications requires either a preprocessor or runtime support.

For now, we focus on *quantification*—through pointcuts—and opt for a conservative design in which open applications are realized *explicitly* using the $\#$ operator: $f \# 2$ is the same as $f\ 2$, except that the application generates a join point that is subject to aspect weaving. We will come back to obliviousness in Sect. 9.3, showing how different answers can be provided within the context of our proposal.

Monadic Setting Our approach to introduce aspects in a pure functional programming language like Haskell can be realized without considering effects. Nevertheless, most interesting applications of aspects rely on computational effects (*e.g.* tracing, memoization, exception handling, etc.). We therefore adopt a monadic setting from the start. Also, as we show in Sect. 6 and 7, this allows us to exploit the approach of EffectiveAdvice [28] and other monadic reasoning mechanisms in order to perform type-based reasoning about effects in presence of aspects.

Illustration As a basic example, recall the $enqueue$ function (Sect. 2.3) and consider the $uniqueAdv$ advice, which enforces that the argument is only passed to $proceed$ if it is not already present in the underlying list l (*e.g.* to avoid repeated elements when representing a set using a list);

```
uniqueAdv proceed arg = do l ← get
                           if elem arg l
                              then return ()
                              else proceed arg
```

Then, in $program$ we $deploy$ an $aspect$ that reacts to applications of $enqueue$. This is specified using the pointcut $pcCall\ enqueue$.

```
program n m = do deploy (aspect (pcCall enqueue) uniqueAdv)
                 enqueue # n
                 enqueue # m
                 showQueue
```

Evaluating $program\ 1\ 2$ returns a string representation `"[1,2]"` with both elements, whereas $program\ 1\ 1$ returns `"[1]"` with only one element. Indeed, both results are as expected. As shown in this example, aspects consist of a pointcut/advice pair and are created with $aspect$, and deployed with $deploy$.

Our development of AOP simply relies on defining aspects (pointcuts, advices), the underlying aspect environment together with the operations to deploy and undeploy aspects, and open function application. The remainder of this section briefly presents these elements, and the following section concentrates on the main challenge: properly typing pointcuts and ensuring type soundness of pointcut/advice bindings.

3.1 Join Point Model

The support for crosscutting provided by an aspect-oriented programming language lies in its *join point model* [24]. A join point model is composed by three elements: *join points* that represents the (dynamic) steps in the execution of a program that aspects can affect, a *means of identifying* join points—here, pointcuts—and a *means of effecting* at join points—here, advices.

Join Points Join points are function applications. A join point JP contains a function of type $a \to m\ b$, and an argument of type a. The monad m denotes the underlying computational effect stack. Note that this means that only functions that are properly lifted to a monadic context can be advised. In addition, in order for pointcuts to be able to reason about the type of advised functions, we require the functions to be $PolyTypeable$[7].

$$\mathbf{data}\ JP\ m\ a\ b = (Monad\ m, PolyTypeable\ (a \to m\ b)) \Rightarrow JP\ (a \to m\ b)\ a$$

From now on, we omit the type constraints related to $PolyTypeable$ (the $PolyTypeable$ constraint on a type is required each time the type has to be inspected dynamically; exact occurrences of this constraint can be found in the implementation).

Pointcuts A pointcut is a predicate on the current join point. It is used to identify join points of interest. A pointcut simply returns a boolean to indicate whether it matches the given join point.

$$\mathbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool))$$

A pointcut is represented as a value of type $PC\ m\ a\ b$. Types a and b are used to ensure type safety, as discussed in Sect. 4.1. The predicate itself is a function with type $\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool)$, meaning it has access to the monad stack. The $\forall$ declaration quantifies on type variables a' and b' (using rank-2 types) because a pointcut should be able to match against any join point, regardless of the specific types involved (we come back to this in Sect. 4.1).

[7] Haskell provides the $Typeable$ class to introspect monomorphic types. $PolyTypeable$ is an extension that supports both monomorphic and polymorphic types.

Pointcut Language We provide two basic pointcut designators, $pcCall$ and $pcType$, as well as logical pointcut combinators, $pcOr$, $pcAnd$, and $pcNot$. A pointcut $pcType\ f$ matches all open applications to functions that have a type compatible with f (see Sect. 4.1 for a precise definition), and a pointcut $pcCall\ f$ matches all open applications to f.

$$\begin{aligned}
&pcType\ f = PC\ (typePred\ (polyTypeOf\ f)) \\
&\quad \mathbf{where}\ typePred\ t\ \ = return\ \$\ \lambda jp \rightarrow return\ (compareType\ t\ jp) \\
&pcCall\ f = PC\ (callPred\ f\ (polyTypeOf\ f)) \\
&\quad \mathbf{where}\ callPred\ f\ t = return\ \$\ \lambda jp \rightarrow return\ (compareFun\ \ f\ jp \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad compareType\ t\ jp)
\end{aligned}$$

In both cases we use the $polyTypeOf$ function (provided by $PolyTypeable$) to obtain the type representation of function f, and compare it to the type of the function in the join point using $compareType$. Additionally, to implement $pcCall$ we require a notion of function equality[8]. This is used in $compareFun$ to compare the function in the join point with the given function f. Note that in $pcCall$ we also need to perform a type comparison, using $compareType$. This is because a polymorphic function whose type variables are instantiated in one way is equal to the same function but with type variables instantiated in some other way (*e.g.* $id :: Int \rightarrow Int$ is equal to $id :: Float \rightarrow Float$).

Users can define their own pointcut designators. For instance, we can define control-flow pointcuts like AspectJ's $cflow$ (described in Sect. 8.1), data flow pointcuts [23], pointcuts that rely on the trace of execution [9] (Sect. 7.1), etc.

Advice An advice is a function that executes in place of a join point matched by a pointcut. This replacement is similar to open recursion in EffectiveAdvice [28]. An advice receives a function (known as the *proceed* function) and returns a new function of the same type (which may or may not apply the original *proceed* function internally). We introduce a type alias for advice:

$$\mathbf{type}\ Advice\ m\ a\ b = (a \rightarrow m\ b) \rightarrow a \rightarrow m\ b$$

For instance, the type $Monad\ m \Rightarrow Advice\ m\ Int\ Int$ is a synonym for the type $Monad\ m \Rightarrow (Int \rightarrow m\ Int) \rightarrow Int \rightarrow m\ Int$. For a given advice of type $Advice\ m\ a\ b$, we call $a \rightarrow m\ b$ the *advised type* of the advice.

Aspects An aspect is a first-class value binding together a pointcut and an advice. Supporting first-class aspects is important: it makes it possible to support aspect factories, separate creation and deployment/undeployment of aspects, exporting opaque, self-contained aspects as single units, etc. We introduce a data definition for aspects, parameterized by a monad m (which has to be the same in the pointcut and advice):

$$\mathbf{data}\ Aspect\ m\ a\ b\ c\ d = Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ c\ d)$$

We defer the detailed definition of $Aspect$ with its type class constraints to Sect. 4.2, when we address the issue of safe pointcut/advice binding.

[8] For this notion of function equality, we use the $StableNames$ API, which relies on pointer comparison. See Sect. 9.1 for discussion on the issues of this approach.

3.2 Aspect Deployment

Aspect Environment The list of aspects that are deployed at a given point in time is known as the *aspect environment*. To be able to define the type $AspectEnv$ as an heterogenous list of aspects, we use an existentially-quantified[9], data $EAspect$ that hides the type parameters of $Aspect$:[10]

$$\mathbf{data}\ EAspect\ m = \forall a\ b\ c\ d.EAspect\ (Aspect\ m\ a\ b\ c\ d)$$
$$\mathbf{type}\ AspectEnv\ m = [EAspect\ m]$$

This environment can be either fixed initially and used globally [24], as in AspectJ, or it can be handled dynamically, as in AspectScheme [11]. Different scoping strategies are possible when dealing with dynamic deployment [40]. Because we are in a monadic setting, we can pass the aspect environment implicitly using a monad. An open function application can then trigger the set of currently-deployed aspects by retrieving these aspects from the underlying monad.

There are a number of design options for the aspect environment, depending on the kind of aspect deployment that is desired. Following the $Reader$ monad, we can provide a fixed aspect environment, and add the ability to deploy an aspect for the dynamic extent of an expression, similarly to the $local$ method of the $Reader$ monad. We can also adopt a state-like monad, in order to support dynamic aspect deployment and undeployment with global scope. In this paper, without loss of generality, we go for the latter.

The $\mathbb{A}_\mathrm{T}$ Monad Transformer Because we are interested in using arbitrary computational effects in programs, we define the aspect environment through a *monad transformer* (Sect. 2.1), which allows the programmer to construct a monad stack of effects. The $\mathbb{A}_\mathrm{T}$ monad transformer is defined as follows:

$$\mathbf{newtype}\ \mathbb{A}_\mathrm{T}\ m\ a = \mathbb{A}_\mathrm{T}\ (\mathbb{S}_\mathrm{T}\ (AspectEnv\ (\mathbb{A}_\mathrm{T}\ m))\ m\ a)\ \mathbf{deriving}\ (Monad)$$

To define the $\mathbb{A}_\mathrm{T}$ transformer we reuse the $\mathbb{S}_\mathrm{T}$ data constructor, because the $\mathbb{A}_\mathrm{T}$ transformer is essentially a state transformer (Sect. 2.2) that threads the aspect environment. Using the $GeneralizedNewtypeDeriving$ extension of GHC, we can automatically derive $\mathbb{A}_\mathrm{T}$ as an instance of $Monad$. We also define a proper instance of $MonadTrans$ (not shown here), and implicit liftings for the standard monad transformers of the MTL.[11] Observe that the aspect environment is bound to the same monad $\mathbb{A}_\mathrm{T}\ m$, in order to provide aspects with access to open applications.

[9] In Haskell an existentially-quantified data type is declared using $\forall$ before the data constructor

[10] Because we cannot anticipate a fixed set of class constraints for deployed aspects, we left the type parameters unconstrained. Aspects with ad-hoc polymorphism have to be instantiated before deployment to statically solve each remaining type class constraint (see Sect. 9.2 for more details).

[11] In the rest of the paper we use the same technique to define our custom monad transformers, hence we omit the **deriving** clauses and standard instance definitions, like $MonadTrans$.

Dynamic Aspect Deployment We now define the functions for dynamic deployment, which simply add and remove an aspect from the aspect environment:

$$
\begin{aligned}
&deploy, undeploy :: EAspect\ (\mathbb{A}_{\mathrm{T}}\ m) \to \mathbb{A}_{\mathrm{T}}\ m\ ()\\
&deploy\ asp \quad\ \ = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda aenv \to return\ ((), asp : aenv)\\
&undeploy\ asp = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda aenv \to return\ ((), deleteAsp\ asp\ aenv)
\end{aligned}
$$

Finally, in order to extract the computation of the underlying monad from an $\mathbb{A}_{\mathrm{T}}$ computation we define the $run\mathbb{A}_{\mathrm{T}}$ function, with type $Monad\ m \Rightarrow \mathbb{A}_{\mathrm{T}}\ m\ a \to m\ a$ (similar to $eval\mathbb{S}_{\mathrm{T}}$ in the state monad transformer), that runs a computation in an empty initial aspect environment. For instance, in the initial example of the $enqueue$ function, we can define a $client$ as follows:

$$client\ n\ m = run\mathbb{I}\ (run\mathbb{A}_{\mathrm{T}}\ (program\ n\ m))$$

3.3 Aspect Weaving

Aspect weaving is triggered through open applications, *i.e.* applications performed with the $\#$ operator, *e.g.* $f\ \#\ x$.

Open Applications We introduce a type class $OpenApp$ that declares the $\#$ operator. This makes it possible to overload $\#$ in certain contexts, and it can be used to declare constraints on monads to ensure that the operation is available in a given context.

$$
\begin{aligned}
&\mathbf{class}\ Monad\ m \Rightarrow OpenApp\ m\ \mathbf{where}\\
&\quad (\#) :: (a \to m\ b) \to a \to m\ b
\end{aligned}
$$

The $\#$ operator takes a function of type $a \to m\ b$ and returns a (woven) function with the same type. Any monad composed with the $\mathbb{A}_{\mathrm{T}}$ transformer has open application defined:

$$
\begin{aligned}
&\mathbf{instance}\ Monad\ m \Rightarrow OpenApp\ (\mathbb{A}_{\mathrm{T}}\ m)\ \mathbf{where}\\
&\quad f\ \#\ a = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda aenv \to \mathbf{do}\\
&\qquad\qquad (woven\_f, aenv') \leftarrow weave\ f\ aenv\ aenv\ (newjp\ f\ a)\\
&\qquad\qquad run\ (woven\_f\ a)\ aenv'
\end{aligned}
$$

An open application results in the creation of a join point, $newjp\ f\ a$, that represents the application of f to a. The join point is then used to determine which aspects in the environment match, produce a new function that combines all the applicable advices, and apply that function to the original argument.

Weaving The function to use at a given point is produced by the $weave$ function:

$$
\begin{aligned}
&weave :: Monad\ m \Rightarrow (a \to \mathbb{A}_{\mathrm{T}}\ m\ b) \to AspectEnv\ (\mathbb{A}_{\mathrm{T}}\ m) \to\\
&\quad AspectEnv\ (\mathbb{A}_{\mathrm{T}}\ m) \to JP\ (\mathbb{A}_{\mathrm{T}}\ m)\ a\ b \to m\ (a \to \mathbb{A}_{\mathrm{T}}\ m\ b, AspectEnv\ (\mathbb{A}_{\mathrm{T}}\ m))\\
&weave\ f\ [\,]\ fenv\ \_ \qquad\qquad\quad = return\ (f, fenv)\\
&weave\ f\ (asp : asps)\ fenv\ jp =
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{case}\ asp\ \textbf{of}\ EAspect\ (Aspect\ pc\ adv) \rightarrow \\
\quad \textbf{do}\ (match, fenv') \leftarrow apply\_pc\ pc\ jp\ fenv \\
\qquad weave\ (\textbf{if}\ match \\
\qquad\qquad\quad \textbf{then}\ apply\_adv\ adv\ f \\
\qquad\qquad\quad \textbf{else}\ f) \\
\qquad\qquad asps\ fenv'\ jp
\end{array}
$$

The $weave$ function is defined recursively on the aspect environment. For each aspect, it applies the pointcut to the join point. It then uses either the partial application of the advice to f if the pointcut matches, or f otherwise[12], to keep on weaving on the rest of the aspect list. This definition is a direct adaptation of AspectScheme's weaving function [11], and is also a *monadic weaver* [38] that supports modular language extensions (in Sect. 8 we show how to exploit this feature).

Applying Advice As we have seen, the aspect environment has type $AspectEnv\ m$, meaning that the type of the advice function is hidden. Therefore, advice application requires *coercing* the advice to the proper type in order to apply it to the function of the join point:

$$
\begin{array}{l}
apply\_adv :: Advice\ m\ a\ b \rightarrow t \rightarrow t \\
apply\_adv\ adv\ f = (unsafeCoerce\ adv)\ f
\end{array}
$$

The operation $unsafeCoerce$ of Haskell is (unsurprisingly) unsafe and can yield segmentation faults or arbitrary results. To recover safety, we could insert a runtime type check with $compareType$ just before the coercion. We instead make aspects type safe such that we can prove that the particular use of $unsafeCoerce$ in $apply\_adv$ is *always* safe. The following section describes how we achieve type soundness of aspects; Sect. 5 formally proves it.

4 Typing Aspects

Ensuring type soundness in the presence of aspects consists in ensuring that an advice is always applied at a join point of the proper type. Note that by "the type of the join point", we refer to the type of the function being applied at the considered join point.

4.1 Typing Pointcuts

The intermediary between a join point and an advice is the pointcut, whose proper typing is therefore crucial. The type of a pointcut as a predicate over join points does not convey any information about the types of join points it matches. To keep this information, we use *phantom type variables* a and b in the definition of PC:

$$\textbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \rightarrow m\ Bool))$$

[12] $apply\_pc$ checks whether the pointcut matches the join point and returns a boolean and a potentially modified aspect environment. Note that $apply\_pc$ is evaluated in the full aspect environment $fenv$, instead of the decreasing $(asp : asps)$ argument.

A phantom type variable is a type variable that is not used on the right hand-side of the data type definition. The use of phantom type variables to type embedded languages was first introduced by Leijen and Meijer to type an embedding of SQL in Haskell [21]; it makes it possible to "tag" extra type information on data. In our context, we use it to add the information about the type of the join points matched by a pointcut: $PC\ m\ a\ b$ means that a pointcut can match applications of functions of type $a \rightarrow m\ b$. We call this type the *matched type* of the pointcut. Pointcut designators are in charge of specifying the matched type of the pointcuts they produce.

Least General Types Because a pointcut potentially matches many join points of different types, the matched type must be a *more general type*. For instance, consider a pointcut that matches applications of functions of type $Int \rightarrow m\ Int$ and $Float \rightarrow m\ Int$. Its matched type is the parametric type $a \rightarrow m\ Int$. Note that this is in fact the *least general type* of both types.[13] Another more general candidate is $a \rightarrow m\ b$, but the least general type conveys more precise information. As a concrete example, below is the type signature of the $pcCall$ pointcut designator:

$$pcCall :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow PC\ m\ a\ b$$

Comparing Types The type signature of the $pcType$ pointcut designator is the same as that of $pcCall$:

$$pcType :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow PC\ m\ a\ b$$

However, suppose that f is a function of type $Int \rightarrow m\ a$. We want the pointcut $pcType\ f$ to match applications of functions of more specific types, such as $Int \rightarrow m\ Int$ and $Int \rightarrow m\ Char$. This means that $compareType$ actually checks that the matched type of the pointcut is *more general* than the type of the join point.

Logical Combinators We use type constraints in order to properly specify the matched type of logical combinations of pointcuts. The intersection of two pointcuts matches join points that are most precisely described by the *principal unifier* of both matched types. Since Haskell supports this unification when the same type variable is used, we can simply define $pcAnd$ as follows:

$$pcAnd :: Monad\ m \Rightarrow PC\ m\ a\ b \rightarrow PC\ m\ a\ b \rightarrow PC\ m\ a\ b$$

For instance, a control flow pointcut matches any type of join point, so its matched type is $a \rightarrow m\ b$. Consequently, if f is of type $Int \rightarrow m\ a$, the matched type of $pcAnd\ (pcCall\ f)\ (pcCflow\ g)$ is $Int \rightarrow m\ a$.

Dually, the union of two pointcuts relies on *anti-unification* [32,33], that is, the computation of the least general type of two types. Haskell does not natively support anti-unification. We exploit the fact that multi-parameter type classes can be used to

[13] The term *most specific generalization* is also valid, but we stick here to Plotkin's original terminology [32].

define relations over types, and develop a novel type class $LeastGen$ (for *least general*) that can be used as a constraint to compute the least general type t of two types t_1 and t_2 (defined in Sect. 5):

$$pcOr :: (Monad\ m, LeastGen\ (a \to b)\ (c \to d)\ (e \to f)) \Rightarrow \\ PC\ m\ a\ b \to PC\ m\ c\ d \to PC\ m\ e\ f$$

For instance, if f is of type $Int \to m\ a$ and g is of type $Int \to m\ Float$, the matched type of $pcOr\ (pcCall\ f)\ (pcCall\ g)$ is $Int \to m\ a$.

The negation of a pointcut can match join points of any type because no assumption can be made on the matched join points:

$$pcNot :: Monad\ m \Rightarrow PC\ m\ a\ b \to PC\ m\ a'\ b'$$

Observe that the type of $pcNot$ is quite restrictive. In fact, the advice of any aspect with a single $pcNot$ pointcut must be completely generic because the matched type corresponds to fresh type variables. The matched type of $pcNot$ can be made more specific using $pcAnd$ to combine it with other pointcuts with more specific types.

Open Pointcut Language The set of pointcut designators in our language is open. User-defined pointcut designators are however responsible for properly specifying their matched types. If the matched type is incorrect or too specific, soundness is lost.

Constraining Pointcuts to Specific Types A pointcut cannot make any type assumption about the type of the join point it receives as argument. The reason for this is again the homogeneity of the aspect environment: when deploying an aspect, the type of its pointcut is hidden. At runtime, then, a pointcut is expected to be applicable to any join point. The general approach to make a pointcut safe is therefore to perform a runtime type check, as was illustrated in the definition of $pcCall$ and $pcType$ in Sect. 3.1. However, certain pointcuts are meant to be conjoined with others pointcuts that will first apply a sufficient type condition.

In order to support the definition of pointcuts that *require* join points to be of a given type, we provide the $RequirePC$ type:

$$\mathbf{data}\ RequirePC\ m\ a\ b = Monad\ m \Rightarrow \\ RequirePC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool))$$

The definition of $RequirePC$ is similar to that of PC, with two important differences. First, the matched type of a $RequirePC$ is interpreted as a type *requirement*. Second, a $RequirePC$ is not a valid stand-alone pointcut: it has to be combined with a standard PC that enforces the proper type upfront. To safely achieve this, we overload $pcAnd$[14]:

$$pcAnd :: (Monad\ m, LessGen\ (a \to b)\ (c \to d)) \Rightarrow \\ PC\ m\ a\ b \to RequirePC\ m\ c\ d \to PC\ m\ a\ b$$

[14] The constraint is different from the previous constraint on $pcAnd$. This is possible thanks to the recent $ConstraintKinds$ extension of GHC.

In this case $pcAnd$ yields a standard PC pointcut and checks that the matched type of the PC pointcut is *less general* than the type expected by the $RequirePC$ pointcut. This is expressed using the constraint $LessGen$, which, as we will see in Sect. 5, is based on $LeastGen$.

To illustrate, let us define a pointcut designator $pcArgGT$ for specifying pointcuts that match when the argument at the join point is greater than a given n (of type a instance of the Ord type class):

$$pcArgGT :: (Monad\ m, Ord\ a) \Rightarrow a \rightarrow RequirePC\ m\ a\ b$$
$$pcArgGT\ n = RequirePC\ \$\ return\ (\lambda jp \rightarrow$$
$$\quad return\ (unsafeCoerce\ (getJpArg\ jp) \geqslant n))$$

The use of $unsafeCoerce$ to coerce the join point argument to the type a forces us to declare the Ord constraint on a when typing the returned pointcut as $RequirePC\ m\ a\ b$ (with a fresh type variable b). To get a proper pointcut, we use $pcAnd$, for instance to match all calls to $enqueue$ where the argument is greater than 10:

$$pcCall\ enqueue\ `pcAnd`\ pcArgGT\ 10$$

The $pcAnd$ combinator guarantees that a $pcArgGT$ pointcut is always applied to a join point with an argument that is indeed of a proper type: no runtime type check is necessary within $pcArgGT$, because the coercion is always safe.

4.2 Typing Aspects

The main typing issue we have to address consists in ensuring that a pointcut/advice binding is type safe, so that the advice application does not fail. A first idea to ensure that the pointcut/advice binding is type safe is to require the matched type of the pointcut and the advised type of the advice to be the same (or rather, unifiable):

$$\text{-- wrong!}$$
$$\mathbf{data}\ Aspect\ m\ a\ b = Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ a\ b)$$

This approach can however yield unexpected behavior. Consider this example:

$$idM\ x = return\ x$$
$$adv :: Monad\ m \Rightarrow Advice\ (Char \rightarrow m\ Char)$$
$$adv\ proceed\ c = proceed\ (toUpper\ c)$$
$$program = \mathbf{do}\ deploy\ (aspect\ (pcCall\ idM)\ adv)$$
$$x \leftarrow idM\ \#\ \texttt{'a'}$$
$$y \leftarrow idM\ \#\ [True, False, True]$$
$$return\ (x, y)$$

The matched type of the pointcut $pcCall\ idM$ is $Monad\ m \Rightarrow a \rightarrow m\ a$. With the above definition of $Aspect$, $program$ passes the typechecker because it is possible to unify a and $Char$ to $Char$. However, when evaluated, the behavior of $program$ is undefined because the advice is unsafely applied with an argument of type $[\mathit{Bool}]$, for which $\mathit{toUpper}$ is undefined.

The problem is that during typechecking, the matched type of the pointcut and the advised type of the advice can be unified. Because unification is symmetric, this succeeds even if the advised type is more specific than the matched type. In order to address this, we again use the type class $\mathit{LessGen}$ to ensure that the matched type is less general than the advice type:

$$\mathbf{data}\ \mathit{Aspect}\ m\ a\ b\ c\ d = (\mathit{Monad}\ m, \mathit{LessGen}\ (a \to m\ b)\ (c \to m\ d)) \Rightarrow \\ \mathit{Aspect}\ (\mathit{PC}\ m\ a\ b)\ (\mathit{Advice}\ m\ c\ d)$$

This constraint ensures that pointcut/advice bindings are type safe: the coercion performed in $\mathit{apply\_adv}$ always succeeds. We formally prove this in the following section.

5 Typing Aspects, Formally

We now formally prove the safety of our approach. We start briefly summarizing the notion of type substitutions and the *is less general* relation between types. Note that we do not consider type class constraints in the definition. Then we describe a novel anti-unification algorithm implemented with type classes, on which the type classes $\mathit{LessGen}$ and $\mathit{LeastGen}$ are based. We finally prove pointcut and aspect safety, and state our main safety theorem.

5.1 Type Substitutions

In this section we summarize the definition of type substitutions and introduce formally the notion of least general type in a Haskell-like type system (without ad-hoc polymorphism). Thus, we have types $t ::= \mathit{Int}, \mathit{Char}, \ldots, t_1 \to t_2, T\ t_1\ \ldots\ t_m$, which denote primitive types, functions, and m-ary type constructors, in addition to user-defined types. We consider a typing environment $\Gamma = (x_i : t_i)_{i \in \mathbb{N}}$ that binds variables to types.

Definition 1 (Type Substitution, from [31]). *A type substitution σ is a finite mapping from type variables to types. It is denoted $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, where $dom(\sigma)$ and $range(\sigma)$ are the sets of types appearing in the left-hand and right-hand sides of the mapping, respectively. It is possible for type variables to appear in $range(\sigma)$.*

Substitutions are always applied simultaneously on a type. If σ and γ are substitutions, and t is a type, then $\sigma \circ \gamma$ is the composed substitution, where $(\sigma \circ \gamma)t = \sigma(\gamma t)$. Application of substitution on a type is defined inductively on the structure of the type.

Substitution is extended pointwise for typing environments in the following way: $\sigma(x_i : t_i)_{i \in \mathbb{N}} = (x_i : \sigma t_i)_{i \in \mathbb{N}}$. Also, applying a substitution to a term t means to apply the substitution to all type annotations appearing in t.

Definition 2 (Less General Type). *We say type t_1* is less general *than type t_2, denoted $t_1 \preceq t_2$, if there exists a substitution σ such that $\sigma t_2 = t_1$. Observe that $\preceq$ defines a partial order on types (modulo α-renaming).*

Definition 3 (Least General Type). *Given types t_1 and t_2, we say type t is the* least general type *iff t is the supremum of t_1 and t_2 with respect to $\preceq$.*

```
1  class LeastGen' a b c σin σout | a b c σin → σout
2    -- Inductive case: The two type constructors match,
3    -- recursively compute the substitution for type arguments ai,bi.
4  instance (LeastGen' a1 b1 c1 σ0 σ1, ...,
5            LeastGen' an bn cn σn−1 σn,
6            T c1 ... cn∼c)
7    ⇒ LeastGen' (T a1 ... an) (T b1 ... bn) c σ0 σn
8    -- Default case: The two type constructors don't match, c has to be a variable,
9    -- either unify c with c' if c' ↦ (a, b), or extend the substitution with c ↦ (a, b)
10 instance (Analyze c (TVar c),
11           MapsTo σin c' (a, b),
12           VarCase c' (a, b) c σin σout)
13   ⇒ LeastGen' a b c σin σout
```

Fig. 1. The $LeastGen'$ type class. An instance holds if c is the least general type of a and b.

5.2 Statically Computing Least General Types

In an aspect declaration, we statically check the type of the pointcut and the type of the advice to ensure a safe binding. To do this we encode an anti-unification algorithm at the type level, exploiting the type class mechanism. A multi-parameter type class $R\ t_1 \ldots t_n$ can be seen as a *relation* R on types $t_1 \ldots t_n$, and **instance** declarations as ways to inductively define this relation, in a manner very similar to logic programming.

The type classes $LessGen$ and $LeastGen$ used in Sect. 4 are defined as particular cases of the more general type class $LeastGen'$, shown in Fig. 1. This class is defined in line 1 and is parameterized by types a, b, c, σ_{in} and σ_{out}. Note that σ_{out} is functionally dependent on a, b, c and σ_{in}; and that there is no **where** keyword because the class declares no operations. Both σ_{in} and σ_{out} denote substitutions encoded at the type level as a list of mappings from type variables to *pairs* of types. We use pairs of types in substitutions because we have to simultaneously compute substitutions from c to a and from c to b.

To be concise, lines 4-7 present a single definition parametrized by the type constructor arity but in practice, there needs to be a different instance declaration for each type constructor arity.

Proposition 1. *If $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then substitution σ_{out} extends σ_{in} and $\sigma_{out} c = (a, b)$.*

Proof. *By induction on the type representation of a and b.*

A type can either be a type variable, represented as $TVar\ a$, or an n-ary type constructor T applied to n type arguments[15]*. The rule to be applied depends on whether the type constructors of a and b are the same or not.*

[15] We use the $Analyze$ type class from $PolyTypeable$ to get a type representation at the type level. For simplicity we omit the rules for analyzing type representations.

*(*i*) If the constructors are the same, then the rule defined in lines 4-7 computes $(T\ c_1 \dots c_n)$ using the induction hypothesis that $\sigma_i c_i = (a_i, b_i)$, for $i = 1 \dots n$. The component-wise application of constraints is done from left to right, starting from substitution σ_0 and extending it to the resulting substitution σ_n. The type equality constraint $(T\ c_1 \dots c_n) \sim c$ checks that c is unifiable with $(T\ c_1 \dots c_n)$ and, if so, unifies them. Then, we can check that $\sigma_n c = (a, b)$.*

*(*ii*) If the type constructors are not the same the only possible generalization is a type variable. In the rule defined in lines 10-13 the goal is to extend σ_{in} with the mapping $c \mapsto (a, b)$ such that $\sigma_{out} c = (a, b)$, while preserving the injectivity of the substitution (see next proposition).* □

Proposition 2. *If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then σ_{out} is an injective function.*

Proof. *By construction $LeastGen'$ introduces a binding from a fresh type variable to (a, b), in the rule defined in lines 10-13, only if there is no type variable already mapping to (a, b)—in which case σ_{in} is not modified.*

To do this, we first check that c is actually a type variable $(TVar\ c)$ by checking its representation using $Analyze$. Then in relation $MapsTo$ we bind c' to the (possibly inexistent) type variable that maps to (a, b) in σ_{in}. In case there is no such mapping, then c' is $None$.

Finally, relation $VarCase$ binds σ_{out} to σ_{in} extended with $\{c \mapsto (a, b)\}$ in case c' is $None$, otherwise $\sigma_{out} = \sigma_{in}$. It then unifies c with c'. In all cases c is bound to the variable that maps to (a, b) in σ_{out}, because it was either unified in rule $MapsTo$ or in rule $VarCase$. The hypothesis that σ_{in} is injective ensures that any preexisting mapping is unique. □

Proposition 3. *If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then c is the least general type of a and b.*

Proof. *By induction on the type representation of a and b.*

*(*i*) If the type constructors are different the only generalization possible is a type variable c.*

*(*ii*) If the type constructors are the same, then $a = T\ a_1 \dots a_n$ and $b = T\ b_1 \dots b_n$. By Proposition 1, $c = T\ c_1 \dots c_n$ generalizes a and b with the substitution σ_{out}. By induction hypothesis c_i is the least general type of (a_i, b_i).*

Now consider a type d that also generalizes a and b, i.e. *$a \preceq d$ and $b \preceq d$, with associated substitution α. We prove c is less general than d by constructing a substitution τ such that $\tau d = c$.*

Again, there are two cases, either d is a type variable, in which case we set $\tau = \{d \mapsto c\}$, or it has the same outermost type constructor, i.e. *$d = T d_1 \dots d_n$. Thus $a_i \preceq d_i$ and $b_i \preceq d_i$; and because c_i is the least general type of a_i and b_i, there exists a substitution τ_i such that $\tau_i d_i = c_i$, for $i = 1 \dots n$.*

Now consider a type variable $x \in dom(\tau_i) \cap dom(\tau_j)$. By definition of α, we know that $\sigma_{out}(\tau_i(x)) = \alpha(x)$ and $\sigma_{out}(\tau_j(x)) = \alpha(x)$. Because σ_{out} is injective (by Proposition 2), we deduce that $\tau_i(x) = \tau_j(x)$ so there are no conflicting mappings between τ_i and τ_j, for any i and j. Consequently, we can define $\tau = \bigcup \tau_i$ and check that $\tau d = c$. □

Definition 4 (LeastGen type class). *To compute the least general type c for a and b, we define:*

$LeastGen\ a\ b\ c \triangleq\ LeastGen'\ a\ b\ c\ \sigma_{empty}\ \sigma_{out}$*, where* σ_{empty} *is the empty substitution and* σ_{out} *is the resulting substitution.*

Definition 5 (LessGen type class). *To establish that type a is less general than type b, we define:*

$LessGen\ a\ b \triangleq\ LeastGen\ a\ b\ b$

5.3 Pointcut Safety

We now establish the safety of pointcuts with relation to join points.

Definition 6 (Pointcut match). *We define the relation $matches(pc, jp)$, which holds iff applying pointcut pc to join point jp in the context of a monad m yields a computation $m\ True$.*

Definition 7 (Safe user-defined pointcut). *Given a join point term jp and type environment Γ, a user-defined pointcut is safe if:*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
implies that $a' \to m\ b' \preceq a \to m\ b$.

Now we prove that the matched type of a given pointcut is more general than the join points matched by that pointcut.

Proposition 4. *Given a join point term jp and a pointcut term pc, and type environment Γ; and that if pc is user-defined, then it is safe (according to Definition 7). Then, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
then $a' \to m\ b' \preceq a \to m\ b$.

Proof. By induction on the matched type of the pointcut.

- *Case $pcCall$: By construction the matched type of a $pcCall\ f$ pointcut is the type of f. Such a pointcut matches a join point with function g if and only if: f is equal to g, and the type of f is less general than the type of g. (On both $pcCall$ and $pcType$ this type comparison is performed by $compareType$ on the type representations of its arguments.)*
- *Case $pcType$: By construction the matched type of a $pcType\ f$ pointcut is the type of f. Such a pointcut only matches a join point with function g whose type is less general than the matched type.*
- *Case $pcAnd$ on $PC\ PC$: Consider pc_1 ‘$pcAnd$‘ pc_2. The matched type of the combined pointcut is the* principal unifier *of the matched types of the arguments—which represents the intersection of the two sets of join points. The property holds by the induction hypothesis applied to pc_1 and pc_2.*

- *Case $pcAnd$ on PC $RequirePC$: Consider pc_1 '$pcAnd$' pc_2. The matched type of the combined pointcut is the type of pc_1 and it is checked that the type required by pc_2 is* more general *so the application of pc_2 will not yield an error. The property holds by induction hypothesis on pc_1.*
- *Case $pcOr$: Consider pc_1 '$pcOr$' pc_2. The matched type of the combined pointcut is the* least general type *of the matched types of the argument, computed by the $LeastGen$ constraint—which represents the union of the two sets of join points. The property holds by induction hypothesis on pc_1 and pc_2.*
- *Case $pcNot$: The matched type of a pointcut constructed with $pcNot$ is a fresh type variable, which by definition is more general than the type of any join point.* □

5.4 Advice Type Safety

If an aspect is well-typed, then the advised type of the advice is more general than the matched type of the pointcut:

Proposition 5. *Given a pointcut term pc, an advice term adv, and a type environment Γ, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash adv : Advice\ m\ c\ d$
$\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$
then $a \to m\ b \preceq c \to m\ d$.

Proof. *Using the definition of $Aspect$ (Sect. 4.2) and because $\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$, we know that the constraint LessGen is satisfied, so by Definitions 4 and 5, and Proposition 1, $a \to m\ b \preceq c \to m\ d$.* □

5.5 Safe Aspects

We now show that if an aspect is well-typed, then the advised type of the advice is more general than the type of join points matched by the corresponding pointcut:

Theorem 1 (Safe Aspects). *Given the terms jp, pc and adv representing a join point, a pointcut and an advice respectively, given a type environment Γ; and assuming that if pc is a user-defined pointcut, then it is safe (according to Definition 7). Then, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash adv : Advice\ m\ c\ d$
$\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$
and
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
then $a' \to m\ b' \preceq c \to m\ d$.

Proof. *By Proposition 4 and 5 and the transitivity of $\preceq$.* □

Corollary 1 (Safe Weaving). *The coercion of the advice in apply_adv is safe.*

$$
\begin{array}{l}
\textbf{module}\ \mathit{Fib}\ (\mathit{fib}, \mathit{pcFib})\ \textbf{where} \\
\textbf{import}\ \mathit{AOP} \\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ `\mathit{pcAnd}`\ \mathit{pcArgGT}\ 2 \\
\mathit{fibBase}\ n = \mathit{return}\ 1 \\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ \mathit{f1} \leftarrow \mathit{fibBase} \# (n-1) \\
\qquad\qquad\qquad\qquad\quad\ \mathit{f2} \leftarrow \mathit{fibBase} \# (n-2) \\
\qquad\qquad\qquad\qquad\quad\ \mathit{return}\ (\mathit{f1} + \mathit{f2}) \\
\mathit{fib} :: \mathit{Monad}\ m \Rightarrow m\ (\mathit{Int} \rightarrow m\ \mathit{Int}) \\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv}) \\
\qquad\quad \mathit{return}\ \$\ \mathit{fibBase}\ \#
\end{array}
$$

Fig. 2. Fibonacci module.

Proof. Recall apply_adv (Sect. 3.3):

$$
\begin{array}{l}
\mathit{apply\_adv} :: \mathit{Advice}\ m\ a\ b \rightarrow t \rightarrow t \\
\mathit{apply\_adv}\ \mathit{adv}\ f = (\mathit{unsafeCoerce}\ \mathit{adv})\ f
\end{array}
$$

By construction, $\mathit{apply\_adv}$ *is used only with a function* f *that comes from a join point that is matched by a pointcut associated to* adv*. Using Theorem 1, we know that the join point has type* $JP\ m\ a'\ b'$ *and that* $a' \rightarrow m\ b' \preceq a \rightarrow m\ b$*. We note* σ *the associated substitution. Then, by compatibility of substitutions with the typing judgement [31], we deduce* $\sigma\Gamma \vdash \sigma adv : \mathit{Advice}\ m\ a'\ b'$*. Therefore* $(\mathit{unsafeCoerce}\ \mathit{adv})$ *corresponds exactly to* σadv*, and is safe.* □

6 Open and Protected Modules, with Effects

This section illustrates how we can exploit the monadic embedding of aspects to encode Open Modules [2] extended with effects. Additionally we present the notion of *protected pointcuts*, which are pointcuts whose type places restrictions on admissible advice. We illustrate the use of protected pointcuts to enforce control flow properties of external advice, reusing the approach of EffectiveAdvice [28].

6.1 A Simple Example

We first describe a simple example that serves as the starting point. Figure 2 describes a Fibonacci module, following the canonical example of Open Modules. The module uses an internal aspect to implement the recursive definition of Fibonacci: the base function, $\mathit{fibBase}$, simply implements the base case; and the fibAdv advice implements recursion when the pointcut pcFib matches. Note that pcFib uses the user-defined pointcut $\mathit{pcArgGT}$ (defined in Sect. 4.1) to check that the call to $\mathit{fibBase}$ is done with an argument greater than 2. The fib function is defined by first deploying the internal aspect, and then partially applying $\#$ to $\mathit{fibBase}$. This transparently ensures that an application

```
module MemoizedFib (fib) where
import qualified Fib
import AOP

memo proceed n =
  do table ← get
     if member n table
       then return (table ! n)
       else do y ← proceed n
               table′ ← get
               put (insert n y table′)
               return y

fib = do deploy (aspect Fib.pcFib memo)
         Fib.fib
```

Fig. 3. Memoized Fibonacci module.

of *fib* is open. The *fib* function is exported, together with the *pcFib* pointcut, which can be used by an external module to advise applications of the internal *fibBase* function. Figure 3 presents a Haskell module that provides a more efficient implementation of *fib* by using a memoization advice. To benefit from memoization, a client only has to import *fib* from the *MemoizedFib* module instead of directly from the *Fib* module.

Note that if we consider that the aspect language only supports the *pcCall* pointcut designator, this implementation actually represents an open module proper. Preserving the properties of open modules, in particular protecting from external advising of internal functions, in presence of arbitrary quantification (*e.g.* *pcType*, or an always-matching pointcut) is left for future work. Importantly, just like Open Modules, the approach described here does not ensure anything about the advice beyond type safety. In particular, it is possible to create an aspect that incorrectly calls *proceed* several times, or an aspect that has undesired computational effects. Fortunately, the type system can assist us in expressing and enforcing specific interference properties.

6.2 Protected Pointcuts

In order to extend Open Modules with effect-related enforcement, we introduce the notion of *protected pointcuts*, which are pointcuts enriched with restrictions on the effects that associated advice can exhibit. Simply put, a protected pointcut embeds a *combinator* that is applied to the advice in order to build an aspect. If the advice does not respect the (type) restrictions expressed by the combinator, the aspect creation expression simply does not typecheck and hence the aspect cannot be built. A combinator is any function that can produce an advice:

$$\textbf{type}\ Combinator\ t\ m\ a\ b = Monad\ m \Rightarrow t \rightarrow Advice\ m\ a\ b$$

The *protectPC* function packs together a pointcut and a combinator:

$$protectPC :: (Monad\ m, LessGen\ (a \rightarrow m\ b)\ (c \rightarrow m\ d)) \Rightarrow$$
$$PC\ m\ a\ b \rightarrow Combinator\ t\ m\ c\ d \rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d$$

A protected pointcut, of type $ProtectedPC$, cannot be used with the standard aspect creation function $aspect$. The following $pAspect$ function is the only way to get an aspect from a protected pointcut (the constructor PPC is not exposed):

$$pAspect :: Monad\ m \Rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d \rightarrow t \rightarrow Aspect\ m\ a\ b\ c\ d$$
$$pAspect\ (PPC\ pc\ comb)\ adv = aspect\ pc\ (comb\ adv)$$

The key point here is that when building an aspect using a protected pointcut, the combinator $comb$ is applied to the advice adv. We now show how to exploit this extension of Open Modules to restrict control flow properties, using the proper type combinators. The next section describes how to control computational effects.

6.3 Enforcing Control Flow Properties

Rinard *et al.* present a classification of advice in four categories depending on how they affect the control flow of programs [34]:

- **Combination:** The advice can call $proceed$ any number of times.
- **Replacement:** There are no calls to $proceed$ in the advice.
- **Augmentation:** The advice calls $proceed$ exactly once, and it does not modify the arguments to or the return value of $proceed$.
- **Narrowing:** The advice calls $proceed$ at most once, and does not modify the arguments to or the return value of $proceed$.

In EffectiveAdvice [28], Oliveira and colleagues show a type-based enforcement of these categories, through advice combinators (Fig. 4). These combinators fit the general $Combinator$ type we described in Sect. 6.2, and can therefore be embedded in protected pointcuts. Observe that no combinator is needed for combination advice, because no interference properties are enforced. Replacement advice is advice that has no access to $proceed$. Augmentation advice is represented by a pair of $before/after$ advice functions, such that $after$ has access to the argument, the return value, and an extra value optionally exposed by the $before$ function. A narrowing advice is in fact the combination of both a replacement advice and an augmentation advice, where the choice between both is driven by a runtime predicate.

As an illustration, observe that memoization is a typical example of a narrowing advice: the combination of a replacement advice ("return memoized value without proceeding") and an augmentation advice ("proceed and memoize return value"), where the choice between both is driven by a runtime predicate ("is there a memoized value for this argument?"). Therefore it is now straightforward for the Fib module to expose a protected pointcut that restricts valid advice to narrowing advice only:

$$\textbf{module}\ Fib\ (fib, ppcFib)\ \textbf{where}$$
$$ppcFib = protectPC\ pcFib\ narrow$$
$$\ldots$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{Replace}\ m\ a\ b = (a \to m\ b)\\
\mathit{replace} :: \mathit{Replace}\ m\ a\ b \to \mathit{Advice}\ m\ a\ b\\
\mathit{replace}\ \mathit{radv}\ \mathit{proceed} = \mathit{radv}\\
\textbf{type}\ \mathit{Augment}\ a\ b\ c\ m = (a \to m\ c, a \to b \to c \to m\ ())\\
\mathit{augment} :: \mathit{Monad}\ m \Rightarrow \mathit{Augment}\ a\ b\ c\ m \to \mathit{Advice}\ m\ a\ b\\
\mathit{augment}\ (\mathit{before}, \mathit{after})\ \mathit{proceed}\ \mathit{arg} =\\
\quad \textbf{do}\ c \leftarrow \mathit{before}\ \mathit{arg}\\
\qquad\ \ b \leftarrow \mathit{proceed}\ \mathit{arg}\\
\qquad\ \ \mathit{after}\ \mathit{arg}\ b\ c\\
\qquad\ \ \mathit{return}\ b\\
\textbf{type}\ \mathit{Narrow}\ m\ a\ b\ c = (a \to m\ \mathit{Bool}, \mathit{Augment}\ m\ a\ b\ c, \mathit{Replace}\ m\ a\ b)\\
\mathit{narrow} :: \mathit{Monad}\ m \Rightarrow \mathit{Narrow}\ m\ a\ b\ c \to \mathit{Advice}\ m\ a\ b\\
\mathit{narrow}\ (p, \mathit{aug}, \mathit{rep})\ \mathit{proceed}\ x =\\
\quad \textbf{do}\ b \leftarrow p\ x\\
\qquad\ \ \textbf{if}\ b\ \textbf{then}\ \mathit{replace}\ \mathit{rep}\ \mathit{proceed}\ x\\
\qquad\qquad\ \ \textbf{else}\ \ \mathit{augment}\ \mathit{aug}\ \mathit{proceed}\ x
\end{array}
$$

Fig. 4. Replacement, augmentation and narrowing advice combinators (adapted from [28]).

$$
\begin{array}{l}
\mathit{memo} :: (\mathbb{S}_{\mathrm{M}}\ (\mathit{Map}\ a\ b)\ m, \mathit{Ord}\ a) \Rightarrow \mathit{Narrow}\ a\ b\ ()\ m\\
\mathit{memo} = (\mathit{pred}, (\mathit{before}, \mathit{after}), \mathit{rep})\ \textbf{where}\\
\quad \mathit{pred}\ n \qquad\ \ = \textbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{return}\ (\mathit{member}\ n\ \mathit{table})\}\\
\quad \mathit{before}\ \_ \qquad = \mathit{return}\ ()\\
\quad \mathit{after}\ n\ r\ \_ = \textbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{put}\ (\mathit{insert}\ n\ r\ \mathit{table})\}\\
\quad \mathit{rep}\ x \qquad\ \ \ = \textbf{do}\ \{\mathit{table} \leftarrow \mathit{get}; \mathit{return}\ (\mathit{table}\ !\ n)\}
\end{array}
$$

Fig. 5. Memoization as a narrowing advice (adapted from [28]).

The protected pointcut $ppcFib$ embeds the $narrow$ type combinator. Hence, only advice that can be statically typed as narrowing advice can be bound to that pointcut. A valid definition of the $memo$ advice is given in Fig. 5. Note that the protected pointcut is only restrictive with respect to the control flow effect of the advice, but not with respect to its computational effect: any monad m is accepted.

Finally, note that this approach is not limited to the four categories of Rinard *et al.*; custom kinds of advice can be defined in a similar way. For instance, we consider *adaptation* advice as a weaker version of narrowing where the advice is allowed to modify the arguments to *proceed*. The implementation is straightforward:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Adaptation}\ a\ b\ c\ m = (a \to a, a \to m\ c, a \to b \to c \to m\ ())\\
\mathit{adapt} :: \mathit{Adaptation}\ a\ b\ c\ m \to \mathit{Advice}\ m\ a\ b\\
\mathit{adapt}\ (\mathit{adapter}, \mathit{before}, \mathit{after})\ \mathit{proceed}\ \mathit{arg} =\\
\quad \mathit{augment}\ (\mathit{before}, \mathit{after})\ \mathit{proceed}\ (\mathit{adapter}\ \mathit{arg})
\end{array}
$$

A relevant design choice is whether the $adapter$ function is pure or is allowed to perform effects. This choice affects which properties can be statically checked based on the type of the advice. Allowing effects is more expressive, but it is source of potential interferences, in addition to advices and pointcuts. The next section describes how to control effect interference between these components.

7 Controlling Effect Interference

The monadic embedding of aspects also enables reasoning about computational effects. We are particularly interested in reasoning about *effect interference* between components of a system: aspects, base programs, and combinations thereof. To do this, in Section 7.1 we first show how to adapt the non-interference types defined in EffectiveAdvice [28], which distinguish between aspect and base computation. The essence of this technique is to use parametricity to forbid components from making assumptions about some part of the monad stack. Then, because components must work uniformly over the restricted section of the stack, they can only utilize effects available in the non-restricted section.

However this approach falls short when considering several aspects in a system, because aspects (and base programs) can still interfere between them. In Section 7.2 we show how a refinement of the technique can be used to address this situation, but that unfortunately is impractical because it requires explicit liftings and strongly couples components to particular shapes of the monad stack—hampering modularity and reusability.

Finally, we show in Section 7.4 a different approach to enforce non-interference based on *monad views* [35], a recently developed mechanism for handling the monad stack, which is summarized in Section 7.3.

7.1 Distinguishing Aspect and Base Computation

To illustrate the usefulness of distinguishing between aspect and base computation, consider a Fibonacci module where the internal calls throw an exception when given a negative integer as argument. In that situation, it is interesting to ensure that the external advice bound to the exposed pointcut cannot throw or catch those exceptions.

Following EffectiveAdvice [28], we can enforce an advice to be parametric with respect to a monad used by base computation, effectively splitting the monad stack into two. To this end we define the $\mathbb{NIA}_{\mathrm{T}}$ ($\mathbb{NI}$ stands for non-interference) type:

$$\mathbf{newtype}\ \mathbb{NIA}_{\mathrm{T}}\ t\ m\ a = \mathbb{NIA}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{AspectEnv}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m))\ (t\ m)\ a)$$

Observe that $\mathbb{NIA}_{\mathrm{T}}$ splits the monad stack into an upper part t, with the effects available to aspects; and a lower part m, with the effects available to base computation. We extend other definitions (weave, deploy, etc.) accordingly.

Note that $\mathbb{NIA}_{\mathrm{T}}$ is a proper monad, but not a monad transformer. This is because the $\mathit{MonadTrans}$ class is designed for a type constructor t that is applied to some monad m, but $\mathbb{NIA}_{\mathrm{T}}$ takes two types as arguments. We could define the partial application $\mathbb{NIA}_{\mathrm{T}}\ t$ as a monad transformer, but this is inconvenient because explicit *lift* operations would skip the upper layer of the stack[16]. However, for allowing explicit lifting into

[16] Because we would lift from m to $(\mathbb{NIA}_{\mathrm{T}}\ t)\ m$

$\mathbb{NIA}_\mathrm{T}$ we need an operation to transform a computation from $t\ m$ into an $\mathbb{NIA}_\mathrm{T}\ t\ m$ computation. To this end we provide the *niLift* operation as follows:

$$
\begin{array}{l}
\mathit{niLift} :: \mathit{Monad}\ (t\ m) \Rightarrow t\ m\ a \rightarrow \mathbb{NIA}_\mathrm{T}\ t\ m\ a \\
\mathit{niLift}\ ma = \mathbb{NIA}_\mathrm{T}\ \$\ \mathbb{S}_\mathrm{T}\ \$\ \lambda aenv \rightarrow \mathbf{do} \\
\qquad\qquad a \leftarrow ma \\
\qquad\qquad return\ (a, aenv)
\end{array}
$$

Effect Interference and Pointcuts The novelty compared to EffectiveAdvice is that we also have to deal with interferences for pointcuts. But to allow effect-based reasoning on pointcuts, we need to distinguish between the monad used by the base computation and the monad used by pointcuts. Indeed, in the interpretation of the type $PC\ m\ a\ b$, m stands for both monads, which forbids to reason separately about them. To address this issue, we need to interpret $PC\ m\ a\ b$ differently, by saying that the matched type is $a \rightarrow b$ instead of $a \rightarrow m\ b$. In this way, the monad for the base computation (which is implicitly bound by b) does not have to be m at the time the pointcut is defined. To accommodate this new interpretation with the rest of the code, very little changes have to be made[17]. Mainly, the types of $pcCall$, $pcType$ and the definition of $Aspect$:

$$
\begin{array}{l}
\mathit{pcCall}, \mathit{pcType} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow PC\ m\ a\ b \\
\mathbf{data}\ \mathit{Aspect}\ m\ a\ b\ c\ d = (\mathit{Monad}\ m, \mathit{LessGen}\ (a \rightarrow b)\ (c \rightarrow m\ d)) \Rightarrow \\
\qquad\qquad\qquad\qquad \mathit{Aspect}\ (PC\ m\ a\ b)\ (\mathit{Advice}\ m\ c\ d)
\end{array}
$$

Note how the definition of $Aspect$ forces the monad of the pointcut computation to be unified with that of the advice, and with that of the base code. The results of Sect. 5 can straightforwardly be rephrased with these new definitions.

Typing Non-Interfering Pointcuts and Advices Using rank-2 types [30] we can restrict the type of pointcuts and advices. The following types synonyms guarantee that non-interfering pointcuts ($NIPC$) and advices ($NIAdvice$) only use effects available in t.

$$
\begin{array}{l}
\mathbf{type}\ \mathit{NIPC}\ t\ a\ b \quad\ \ = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow \\
\qquad\qquad\qquad\qquad PC\ (\mathbb{NIA}_\mathrm{T}\ t\ m)\ a\ b \\
\mathbf{type}\ \mathit{NIAdvice}\ t\ a\ b = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow \\
\qquad\qquad\qquad\qquad \mathit{Advice}\ (\mathbb{NIA}_\mathrm{T}\ t\ m)\ a\ b
\end{array}
$$

By universally quantifying over the type m of the effects used in the base computation, these types enforce, through the properties of parametricity, that pointcuts or advices cannot refer to specific effects in the base program. We can define aspect construction functions that enforce different (non-)interference patterns, such as non-interfering pointcut $NIPC$ with unrestricted advice $Advice$, unrestricted pointcut PC with non-interfering advice $NIAdvice$, etc.

[17] The implementation available online uses this interpretation of $PC\ m\ a\ b$.

$$
\begin{array}{l}
\textbf{module}\ \mathit{FibErr}\ (\mathit{fib}, \mathit{ppcFib})\ \textbf{where}\\
\textbf{import}\ \mathit{AOP}\\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ \text{`}\mathit{pcAnd}\text{`}\ \mathit{pcArgGT}\ 2\\
\mathit{ppcFib} = \mathit{protectPC}\ \mathit{pcFib}\ \mathit{niAdvice}\\
\mathit{fibBase}\ n = \mathit{return}\ 1\\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ \mathit{f1} \leftarrow \mathit{errorFib}\ \#\ (n-1)\\
\qquad\qquad\qquad\qquad\quad \mathit{f2} \leftarrow \mathit{errorFib}\ \#\ (n-2)\\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{f1} + \mathit{f2})\\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv})\\
\qquad\quad \mathit{return}\ \mathit{errorFib}\\
\mathit{errorFib} :: (\mathit{MonadTrans}\ t, \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ m) \Rightarrow \mathit{Int} \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ \mathit{Int}\\
\mathit{errorFib}\ n = \textbf{if}\ n < 0\\
\qquad\qquad\quad \textbf{then}\ \ (\mathit{niLift} \circ \mathit{lift} \circ \mathit{throwError})\text{``}\mathit{Error} : \mathit{negative\ argument}''\\
\qquad\qquad\quad \textbf{else}\ \ \ \mathit{fibBase}\ \#\ n
\end{array}
$$

Fig. 6. Fibonacci with error.

Enforcing Non-Interference Coming back to Open Modules and protected pointcuts, to enforce non-interfering advice we need to define a typed combinator that requires an advice of type $\mathit{NIAdvice}$:

$$
\begin{array}{l}
\mathit{niAdvice} :: (\mathit{Monad}\ (t\ m), \mathit{Monad}\ m) \Rightarrow\\
\qquad\qquad \mathit{NIAdvice}\ t\ a\ b \rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m)\ a\ b\\
\mathit{niAdvice}\ \mathit{adv} = \mathit{adv}
\end{array}
$$

Observe that the $\mathit{niAdvice}$ combinator is computationally the identity function, but it does impose a type requirement on its argument. Using this combinator, a module can expose a protected pointcut that enforces non-interference with base effects.

Fibonacci Module with Error Handling We now define a Fibonacci module (Fig. 6) where base functions $\mathit{fibBase}$ and fibAdv raise an exception when given a negative argument.[18] The exception is raised on monad m that corresponds to base computation, and which is required to be an instance of $\mathbb{E}_{\mathrm{M}}$. The definition of ppcFib enforces that external advice cannot manipulate exceptions in m, because it uses the $\mathit{niAdvice}$ advice combinator. The drawback is that because we are using an effect in an inner layer of the stack, we need to use explicit lifting to satisfy the expected type.

Non-Interfering Base Computation Symmetrically, we can check that a part of the base code cannot interfere with effects available to aspects by using the type synonym NIBase, which universally quantifies over the type t of effects available to the advice:

$$
\begin{array}{l}
\textbf{type}\ \mathit{NIBase}\ m\ a\ b = \forall t.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow\\
\qquad\qquad\qquad\qquad a \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ b
\end{array}
$$

[18] We do not use an error-checking argument on purpose, for the sake of illustration. We use such an aspect in Sect. 7.2 where we consider the issues of multiple effectful aspects.

Reasoning About Pointcut Interference Another use of effect reasoning can be done at the level of pointcuts. Indeed, in the monadic embedding of aspects, we allow for effectful pointcuts. For example, we can define a sequential pointcut combinator [10] $pcSeq\ pc_1\ pc_2$, that matches first pc_1 and then pc_2:

$$
\begin{array}{l}
pcSeq :: (\mathbb{S}_{\mathrm{M}}\ Bool\ m) \Rightarrow PC\ m\ a\ b \rightarrow PC\ m\ c\ d \rightarrow PC\ m\ c\ d \\
pcSeq\ (PC\ mpc_1)\ (PC\ mpc_2) = \\
\quad PC\ \$\ \mathbf{do}\ pc_1 \leftarrow mpc_1 \\
\qquad\qquad pc_2 \leftarrow mpc_2 \\
\qquad\qquad return\ \$\ \lambda jp \rightarrow \mathbf{do}\ b \leftarrow get \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{if}\ b\ \mathbf{then}\ pc_2\ jp \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathbf{do}\ b' \leftarrow pc_1\ jp \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad put\ b' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ False
\end{array}
$$

As expressed in the $\mathbb{S}_{\mathrm{M}}\ \ Bool\ \ m$ constraint, the pointcut requires a boolean state in which to store the current point of its matching behavior: *False* (resp. $True$) means pc_1 (resp. pc_2) is to be matched. Consequently, any base program that modifies this state will alter the behavior of the pointcut. This situation can be avoided by using the non-interfering base computation type $NIBase$, just described above.

7.2 Interference Between Multiple Aspects

$\mathbb{NIA}_{\mathrm{T}}$ only distinguishes between base and aspect computation. Although useful, this implies that interference between aspects is still possible because all of them will share the same upper part of the monad stack. A similar situation happens with base programs and the lower part of the monad stack.

To illustrate this issue, consider a Fibonacci module program that uses the $memo$ advice to improve the performance, and also uses a $checkArg$ advice that throws an exception when given a negative argument (instead of a base code check as in Fig. 6). In this setting, $checkArg$ could update the cache with incorrect values, either accidentally or intentionally; or conversely, $memo$ could throw arbitrary exceptions, even with a non-negative argument.

Finer-Grained Splitting of the Monad Stack Following the idea used in $\mathbb{NIA}_{\mathrm{T}}$, to enforce non-interference between $memo$ and $checkArg$ we need to split the monad stack into the monad for base computation m, and two upper layers t_1 and t_2. The idea is to assign to each aspect a unique layer in the stack, and to use parametricity to ensure non-interference. To this end we define the $\mathbb{NIA}_{\mathrm{T}_2}$ monad, which splits the monad stack as described. We also consider $niLift_2$, which serves the same role as $niLift$.

$$
\begin{array}{l}
\mathbf{newtype}\ \mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m\ a = \\
\qquad\qquad \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{S}_{\mathrm{T}}\ (AspectEnv\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m))\ (t_1\ (t_2\ m))\ a)
\end{array}
$$

Again, we extend other definitions properly ($weave$, etc.). Using rank-2 types, the following type synonyms guarantee that non-interfering pointcuts and advices access can only access the effect available in the first layer L_1, which corresponds to t_1; or in the second layer L_2, which corresponds to t_2.

$$\begin{aligned}
&\textbf{type } \mathit{NIPC}_{L_1} \quad t_1\ a\ b = \forall t_2\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{PC}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\textbf{type } \mathit{NIPC}_{L_2} \quad t_2\ a\ b = \forall t_1\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{PC}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\textbf{type } \mathit{NIAdvice}_{L_1}\ t_1\ a\ b = \forall t_2\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\textbf{type } \mathit{NIAdvice}_{L_2}\ t_2\ a\ b = \forall t_1\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b
\end{aligned}$$

Non-Interference Combinators To enforce non-interference properties we need to define advice combinators, as we did with $\mathit{niAdvice}$. Again, we can enforce different non-interference patterns, by defining as many construction functions as required. We describe the advice combinators $\mathit{niAdvice}_{L_1}$ and $\mathit{niAdvice}_{L_2}$ that enforce that aspects work exclusively with the effect provided by the first and second layer, respectively.

$$\begin{aligned}
&\mathit{niAdvice}_{L_1} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1, \mathit{MonadTrans}\ t_2) \Rightarrow\\
&\qquad \mathit{NIAdvice}_{L_1}\ t_1\ a\ b \rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathit{niAdvice}_{L_1}\ \mathit{adv} = \mathit{adv}\\
&\mathit{niAdvice}_{L_2} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1, \mathit{MonadTrans}\ t_2) \Rightarrow\\
&\qquad \mathit{NIAdvice}_{L_2}\ t_2\ a\ b \rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathit{niAdvice}_{L_2}\ \mathit{adv} = \mathit{adv}
\end{aligned}$$

Now we define the monad stack S that provides the state and error-handling effects.

$$\textbf{type } S = \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{E}_{\mathrm{T}}\ \mathit{String})\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int}))\ \mathbb{I}$$

Then, we define the new fibonacci function using the $\mathit{checkArg}_{L_1}$ and memo_{L_2} advices, which operate on the first and second layer of the monad stack, respectively.

$$\begin{aligned}
&\mathit{fibMemoErr} :: \mathit{Int} \rightarrow S\ \mathit{Int}\\
&\mathit{fibMemoErr}\ n = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ (\mathit{niAdvice}_{L_2}\ \mathit{memo}_{L_2}))\\
&\qquad f \leftarrow \mathit{fib}\\
&\qquad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{niAdvice}_{L_1}\ \mathit{checkArg}_{L_1}))\\
&\qquad f \mathbin{\#} n
\end{aligned}$$

The implementation of $\mathit{checkArg}_{L_1}$ is as follows:

$$\begin{aligned}
&\mathit{checkArg}_{L_1}\ \mathit{proceed}\ \mathit{arg} =\\
&\quad \textbf{if}\ \mathit{arg} < 0\\
&\qquad \textbf{then}\ (\mathit{niLift}_2 \circ \mathit{throwError})\ \texttt{"Error: negative argument"}\\
&\qquad \textbf{else}\ \mathit{proceed}\ \mathit{arg}
\end{aligned}$$

And similarly, we define memo_{L_2}:

$$
\begin{array}{l}
\mathit{memo}_{L_2}\ \mathit{proceed}\ n = \\
\quad \textbf{do}\ \mathit{table} \leftarrow \mathit{niLift}_2\ \$\ \mathit{lift}\ \$\ \mathit{get} \\
\qquad \textbf{if}\ \mathit{member}\ n\ \mathit{table} \\
\qquad\quad \textbf{then}\ \mathit{return}\ (\mathit{table}\ !\ n) \\
\qquad\quad \textbf{else}\ \ \textbf{do}\ y \leftarrow \mathit{proceed}\ n \\
\qquad\qquad\qquad\quad \mathit{table}' \leftarrow \mathit{niLift}_2\ \$\ \mathit{lift}\ \$\ \mathit{get} \\
\qquad\qquad\qquad\quad (\mathit{niLift}_2 \circ \mathit{lift} \circ \mathit{put})\ (\mathit{insert}\ n\ y\ \mathit{table}') \\
\qquad\qquad\qquad\quad \mathit{return}\ y
\end{array}
$$

Note that $\mathit{checkArg}_{L_1}$ is applied on calls to the *external* fibonacci function f, while memo_{L_2} is applied to the *internal* calls of the Fibonacci module, exposed by pcFib.

While an improvement over the binary base/aspect approach of EffectiveAdvice, illustrated in Section 7.1, this approach has two major drawbacks. First, it is not scalable because we need a different $\mathbb{NIA}_{\mathrm{T}_n}$ monad to support a setting with n mutually exclusive effects for aspects. Second, it is necessary to use explicit lifting in the implementation of advice. The reason is that we are explicitly using an effect from a layer at an arbitrary position in monad stack. Because we need to preserve parametricity to enforce non-interference, an advice cannot make *any* assumptions on the monad transformers that compose the stack. In particular, it cannot assume that the transformers support implicit liftings from the inner layers of the stack. In fact, in the presence of implicit lifting the layer from which an effect comes depends on the concrete monad stack used. These issues hamper modularity and reusability of aspects. In general, there is a tension between implicit lifting—designed to make a layer provide several effects at once—and splitting the monad stack with one aspect/effect per layer. In Section 7.4 we address these issues by using monad views [35].

7.3 Interlude: Monad Views

Monad views, recently developed by Schrijvers and Oliveira [35], are a technique for handling the monad stack, which extends and complements the standard mechanisms of explicit and implicit liftings (Section 2). Monad views provide robust support for accessing the effects of the monad stack without being coupled to a particular stack layout. Views are denoted using $\rightsquigarrow$, and are an instance of the View type class that defines the *from* operation. Additionally, we use *bidirectional views*, denoted with the $\bowtie$ type operator. In addition to from, a bidirectional view supports the to operation.

$$
\begin{array}{lll}
\mathit{from} & :: (\mathit{Monad}\ m, \mathit{Monad}\ n, \mathit{View}\ (\rightsquigarrow)) & \Rightarrow n \rightsquigarrow m \rightarrow n\ a \rightarrow m\ a \\
\mathit{to} & :: (\mathit{Monad}\ m, \mathit{Monad}\ n) & \Rightarrow n \bowtie m \rightarrow m\ a \rightarrow n\ a
\end{array}
$$

In short, given two monads n and m, a view $n \rightsquigarrow m$ transforms computations from n to m, and a bidirectional view $n \bowtie m$ can also transform computations from m to n.

View-specific operations. Views are first-class values, hence they can be used as arguments. For instance, consider the functions getv and putv defined in [35]:

$$
\begin{array}{l}
\mathit{getv} :: (\mathit{Monad}\ m, \mathbb{S}_{\mathrm{M}}\ s\ n, \mathit{View}\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow m\ s \\
\mathit{getv}\ v = \mathit{from}\ v\ \$\ \mathit{get}
\end{array}
$$

$$putv :: (Monad\ m, \mathbb{S}_{\mathrm{M}}\ s\ n, View\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow s \rightarrow m\ ()$$
$$putv\ v = from\ v \circ put$$

Given an initial monad m and a view $n \rightsquigarrow m$, $getv$ returns a computation $m\ s$ from an arbitrary state layer n. Conversely, $putv$ puts a new value into state layer n.

Creating views Schrijvers and Oliveira propose the construction of views using *structural* and *nominal masks*, which are applied onto the layers of a monad stack [35].

- A structural mask is a bit-like mask applied to the monadic stack in order to hide the layers that conflict with implicit lifting. Such a mask is created by concatenating unary masks for each layer using the ::: type operator:[19] □ indicates a visible layer and ■ a hidden layer.
- A nominal mask refers to layers of the stack using *names* instead of relative positions. This is done with the *tag monad transformer* $\mathbb{T}$. Given an arbitrary type Tag, the layer $\mathbb{T}^{Tag}$ labels a particular position of the monad stack using type Tag. An example of a tagged monad stack (for some types Tag_1 and Tag_2) is:

  $$\mathbf{type}\ M = \mathbb{T}^{Tag_1}\ (\mathbb{S}_{\mathrm{T}}\ Int\ (\mathbb{T}^{Tag_2}\ \mathbb{E}_{\mathrm{T}}\ String\ \mathbb{I}))$$

  where the $\mathbb{S}_{\mathrm{T}}$ layer is labeled with Tag_1 and the $\mathbb{E}_{\mathrm{T}}$ layer is labeled with Tag_2. For inspecting tagged monad stacks, the type class $n \sqsubseteq_{Tag} m$ exposes a monad n representing the layer of the stack m tagged with type Tag. It also provides the $structure$ operation to obtain the view between n and m associated to t:

  $$\mathbf{class}\ (Monad\ m, Monad\ n) \Rightarrow n \sqsubseteq_{Tag} m\ \mathbf{where}$$
  $$structure :: View\ (\rightsquigarrow) \Rightarrow Tag \rightarrow (n \rightsquigarrow m)$$

7.4 Beyond the Aspect/Base Distinction

Monad views enable a different approach to enforce non-interference. The idea is that aspects will be generic with respect to the effects they require using type class constraints, assuming exclusive access to a monad stack with those effects. To avoid non-interference, client code uses a concrete monad stack and transforms each advice into a view-specific advice where the aspect only sees the sections of the monad stack that it is allowed to access.

For instance, the $memo$ advice described in Fig. 3 requires access to a dictionary to store the precomputed results. This is explicit in the (inferred) type of the advice:

$$memo :: (Monad\ m, Ord\ a, \mathbb{S}_{\mathrm{M}}\ (Map\ a\ b)\ m) \Rightarrow Advice\ m\ a\ b$$

In a similar way we define $checkArg$, which requires access to an error effect:

$$checkArg :: (Monad\ m, Num\ a, \mathbb{E}_{\mathrm{M}}\ String\ m) \Rightarrow Advice\ m\ a\ b$$
$$checkArg\ proceed\ arg =$$
$$\quad \mathbf{if}\ arg < 0$$
$$\qquad \mathbf{then}\ throwError\ \text{“}Error : negative\ argument''$$
$$\qquad \mathbf{else}\ \ proceed\ arg$$

[19] We follow the graphical notation used in [35]

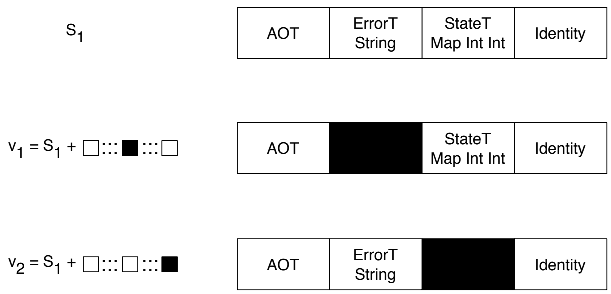

Fig. 7. Applying structural masks to the monad stack S_1.

Arbitrarily Splitting the Monad Stack with Views Observe now that the advice does not depend on the specific position of an effect in the monad stack. The novelty with respect to using implicit liftings is that we can assign to each aspect a *virtual view* of the monad stack that only contains the effect available to them. To assign a part of the monad stack to an advice we define the $withView$ function:

$$withView :: (Monad\ n, Monad\ m) \Rightarrow n \bowtie m \rightarrow Advice\ n\ a\ b \rightarrow Advice\ m\ a\ b$$
$$withView\ v\ adv\ proceed\ arg = from\ v\ \$\ adv\ (\lambda a \rightarrow to\ v\ (proceed\ a))\ arg$$

This function transforms an advice from a restricted monad n to an advice in the "complete" stack m, using a bidirectional view provided as argument. We require a bidirectional view because we need to lift the $proceed$ function, with type $a \rightarrow n\ b$ into an equivalent function with type $a \rightarrow m\ b$—which by construction performs effects only on n. Then, because evaluation of the restricted advice yields a computation $n\ b$, we use the $from$ operation to lift it into a computation $m\ b$.

Observe that partially applying $withView$ with a given view yields a function of type $Advice\ n\ a\ b \rightarrow Advice\ m\ a\ b$, which fits with the notion of advice combinators (Sect. 6.2). Therefore it is possible to export protected pointcuts that expose a particular section of the monad stack to external advice. Additionally we can define functions to transform join points and pointcuts, in a similar way to $withView$.

Using Structural Masks Consider a concrete monad stack S_1 which holds the required state and error effects.

$$\textbf{type}\ S_1 = \mathbb{A}_T\ (\mathbb{E}_T\ String\ (\mathbb{S}_T\ (Map\ Int\ Int)\ \mathbb{I}))$$

Then, we define the fibonacci function as follows:

$$
\begin{aligned}
&fibMemoErr'\ n = \textbf{do}\ deploy\ (aspect\ pcFib\ (withView\ v_1\ memo))\\
&\qquad\qquad\qquad\quad f \leftarrow fib\\
&\qquad\qquad\qquad\quad deploy\ (aspect\ (pcCall\ f)\ (withView\ v_2\ checkArg))\\
&\qquad\qquad\qquad\quad f\ \#\ n\\
&\quad \textbf{where}\ v_1 = \square ::: \blacksquare ::: \square\\
&\qquad\qquad\ v_2 = \square ::: \square ::: \blacksquare
\end{aligned}
$$

We define views v_1 and v_2 using structural masks. Both allow access to $\mathbb{A}_\mathrm{T}$, allowing AOP-specific operations into advice (*e.g.* deploying aspects). Besides that, v_1 exposes only the $\mathbb{S}_\mathrm{T}$ transformer, whereas v_2 only allows accessing to the $\mathbb{E}_\mathrm{T}$ transformer. Figure 7 depicts how views v_1 and v_2 define new *virtual monad stacks*, by applying structural masks to S_1. Note that structural masks can be applied only to monad transformers, but not the monad at the bottom of the stack.

It is clear that now aspects do not need to perform explicit liftings and are not coupled to a particular monad stack. However, these issues are present when constructing views using nominal masks. Changes to the monad stack that is used to run client code need to be reflected in (potentially many) client functions that use structural masks.

Using Nominal Masks A more flexible approach that is not coupled to any particular monad stack is to use nominal masks to tag each effect required by aspects. Then client code can use the tags to directly access the effects and properly transform the advices. Consider a monad stack S_2, where the state and error layers are tagged:

$$\begin{array}{l}\mathbf{data}\ \mathit{StateTag}\\ \mathbf{data}\ \mathit{ErrorTag}\\ \mathbf{type}\ S_2 = \mathbb{A}_\mathrm{T}\ (\mathbb{T}^{\mathit{ErrorTag}}\ (\mathbb{E}_\mathrm{T}\ \mathit{String}\ (\mathbb{T}^{\mathit{StateTag}}\ (\mathbb{S}_\mathrm{T}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ \mathbb{I})))\end{array}$$

The stack is tagged at the type level, therefore we define two singleton types (with no data constructors), namely $\mathit{StateTag}$ and $\mathit{ErrorTag}$, to use as arguments for the $\mathbb{T}$ monad transformer.

The fibonacci function implemented using nominal masks is:

$$\begin{array}{l}\mathit{fibMemoErr}'' :: \forall m\ n_1\ n_2.(\mathit{Monad}\ m,\\ \quad n_1 \sqsubseteq_{\mathit{StateTag}} (\mathbb{A}_\mathrm{T}\ m), \mathbb{S}_\mathrm{M}\ (\mathit{Map}\ \mathit{Int}\ \mathit{Int})\ n_1,\\ \quad n_2 \sqsubseteq_{\mathit{ErrorTag}} (\mathbb{A}_\mathrm{T}\ m), \mathbb{E}_\mathrm{M}\ \mathit{String}\ n_2)\\ \quad \Rightarrow \mathit{Int} \rightarrow \mathbb{A}_\mathrm{T}\ m\ \mathit{Int}\\ \mathit{fibMemoErr}''\ n = \mathbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ (\mathit{withView}\ v_1\ \mathit{memo}))\\ \qquad\qquad\qquad\qquad\quad f \leftarrow \mathit{fib}\\ \qquad\qquad\qquad\qquad\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{withView}\ v_2\ \mathit{checkArg}))\\ \qquad\qquad\qquad\qquad\quad f\ \#\ n\\ \quad \mathbf{where}\ v_1 = \mathit{structure}\ \mathit{StateTag} :: n_1 \bowtie m\\ \qquad\qquad\ v_2 = \mathit{structure}\ \mathit{ErrorTag} :: n_2 \bowtie m\end{array}$$

In contrast to the previous definition, we need to use explicit type annotations because using nominal masks can lead to ambiguity in type inference[20]. Observe that we assume a monad m that is tagged with two singleton types $\mathit{StateTag}$ and $\mathit{ErrorTag}$. We use $\sqsubseteq$ to expose these layers as monads n_1 and n_2 respectively, and we constrain these monads to expose the corresponding effects. Therefore, by using nominal masks we can independently evolve the definition of S_2, as long as we keep the tagged layers expected by $\mathit{fibMemoErr}''$ (satisfying both the tag name and the required effect).

[20] The $\forall m\ n_1\ n_2$ annotation is required to use the type variables in the scope of a $\mathbf{do}$ expression.

Perspectives on Using Views The content of the **do** expression is the same using structural or nominal masks. In fact it is possible to define a more generic function that takes views v_1 and v_2 as argument. Because views are first-class values, there is a wide design space on how to use them to control aspect interference. For example, aspects can be defined directly using $\sqsubseteq$ constraints as required.

On the other hand, programmers must carefully define the views that are provided to each advice, because the typechecker cannot distinguish between intentional and accidental sharing of effects.

Controlling effect interference between aspects is a well-known and widely researched area in the AOP community. The two approaches presented in this section show that the concrete mechanism used to manage the monad stack determines the expressiveness of type-based reasoning techniques. We believe that the problem of assigning exclusive access to effects in the monadic stack originates from the fact that the monad stack is *public* and *transparent* to all components in a system. We conjecture that a mechanism that statically controls access to effects, while being flexible for developers ought to be devised, and indeed is a line of future work that transcends aspect-oriented programming. An additional line of future work is the connection between monad views and MRI [29] (a framework for monadic reasoning that extends EffectiveAdvice [35]), which is based on parametricity and considers only implicit and explicit lifting.

As a final remark, in a setting with an unrestricted $deploy$ operation the restrictions on advice must be applied at each particular aspect deployment. This makes it difficult to establish global properties about advice in a system (which may require external static analysis). This can be solved with a custom $\mathbb{A}_{\mathrm{T}}$-like monad transformer that provides a more restricted deployment mechanism.

8 Language Extensions

The typed monadic embedding of aspects supports modular extensions of the aspect language. The simplest extension is to introduce new user-defined pointcuts. More interestingly, because the language features a monadic weaver [38], we can modularly implement new semantics for aspect scoping and weaving. In addition, all language extensions benefit from the type-based reasoning techniques described in this paper—to the best of our knowledge, this is a novel contribution of this work. In this section we describe the following developments:

- A user-defined $pcCflow$ pointcut designator.
- Secure weaving, in which a set of join points can be hidden from advising.
- Privileged aspects that can see hidden join points from a secure computation.
- Aspect weaving with *execution levels* [41].
- An example of type-based reasoning in the semantics of execution levels.

8.1 Cflow Pointcut

An interesting illustration of extending the language with user-defined pointcuts is the case of control flow checks. Essentially, implementing the $pcCflow$ pointcut requires a way to track join points emitted during program execution. This tracking mechanism can be implemented modularly using a state monad transformer that holds a stack of join points, and an aspect that matches every join point, stores it in the stack, and then proceeds to obtain the result, which is returned after popping the stack. This corresponds to the stack-based implementation of *cflow* described in [24].

Join Point Stack To do this, we first define a join point stack as a list of existentially-quantified join points, EJP, just like we did to define the aspect environment as a list of homogeneous $EAspect$ values (Sect. 3.1).

$$\textbf{data}\ EJP = \forall a\ b\ m.Monad\ m \Rightarrow EJP\ (JP\ m\ a\ b)$$
$$\textbf{type}\ JPStack = [EJP]$$

Then, to collect the join points into a $JPStack$ we define the $\mathbb{JP}_\mathrm{T}$ monad transformer, reusing the implementation of the standard $\mathbb{S}_\mathrm{T}$ transformer:

$$\textbf{newtype}\ \mathbb{JP}_\mathrm{T}\ m\ a = \mathbb{JP}_\mathrm{T}\ (\mathbb{S}_\mathrm{T}\ JPStack\ m\ a)$$

In addition, to support a polymorphic monad stack we define the $\mathbb{JP}_\mathrm{M}$ type class as follows, and declare $\mathbb{JP}_\mathrm{T}$ as an instance.

```
class Monad m ⇒ JP_M m where
  getJPStack  :: m JPStack
  pushJPStack :: EJP → m ()
  popJPStack  :: m ()
instance Monad m ⇒ JP_M (JP_T m) where ...
```

Defining pcCflow Given the definitions above, the implementation of *pcCflow* is very similar to that of *pcCall* (Sect. 3.1).[21]

```
pcCflow :: JP_M m ⇒ (a → m b) → PC m c (m' d)
pcCflow f = return (λ_ → do
  jpStack ← getJPStack
  return $ any (λejp → compareFunEJP f ejp ∧ compareTypeEJP f ejp)
               jpStack
```

Here *compareFunEJP* checks the equality of the function bound to the join point and function f; and *compareTypeEJP* checks that the type of f is more general than the type of the join point. Function *any* returns whether any element of *jps* satisfies a given predicate. We can define the *pcCflowbelow* pointcut in a similar way.

[21] Note that, as discussed in Sect. 4.1, we specifically declare that the matched type of the pointcut is in a different monad m'.

Maintaining the Join Point Stack Now it remains to define the aspect that maintains the join point stack. We first define the $pcAny$ pointcut, which matches all functions applications and pushes the corresponding join point into the stack.

$$\begin{aligned}
&pcAny :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow PC\ m\ a\ b \\
&pcAny = PC\ \$\ return\ \$\ \lambda jp \rightarrow \mathbf{do}\ pushJPStack\ (EJP\ jp) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ True
\end{aligned}$$

Note that the definition of $pcAny$ preserves type soundness (Sect. 4.1) because its matched type is given by two fresh type variables a and b, and hence is the most general type possible. Next, we define $collectAdv$ as an advice that performs $proceed$, pops the stack and returns the result.

$$\begin{aligned}
collectAdv\ proceed\ arg = \mathbf{do}\ & result \leftarrow proceed\ arg \\
& popJPStack \\
& return\ result
\end{aligned}$$

Finally, we define the $maintainJpStack$ aspect as follows.

$$\begin{aligned}
&maintainJpStack :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow Aspect\ m\ a\ (m\ b)\ a\ b \\
&maintainJpStack = aspect\ pcAny\ collectAdv
\end{aligned}$$

This approach is inefficient because we are matching and storing all join points, instead of only those that can be queried in existing uses of $pcCflow$. Alternative optimizations can be defined, for example putting in the stack only relevant join points, or a per-flow deployment that allows using a boolean instead of a stack [24].

A consequence of not defining $pcCflow$ as a *primitive* pointcut is that we need to ensure that evaluation of $maintainJpStack$ occurs before than any other advice. Otherwise, control flow pointcuts from other aspects will have incorrect information to determine whether to execute the advice. This can be implemented directly in a custom $\mathbb{A}_{\mathrm{T}}$ transformer that takes a list of priority aspects and ensures they are always evaluated first during weaving.

8.2 Secure Weaving

For security reasons it can be interesting to protect certain join points from being advised. To support such a secure weaving, we define a new monad transformer $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, which embeds an (existentially quantified) pointcut that specifies the hidden join points, and we modify the weaving process accordingly (not shown here).

$$\begin{aligned}
&\mathbf{data}\ EPC\ m = \forall a\ b.EPC\ (PC\ m\ a\ b) \\
&\mathbf{data}\ \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m\ a = \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m) \\
&\qquad\qquad\qquad \rightarrow EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m) \\
&\qquad\qquad\qquad \rightarrow m\ (a, (AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m), EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m))))
\end{aligned}$$

This can be particularly useful when used with the $pcCflow$ pointcut to protect the computation that occurs in the control flow of critical function applications. For instance, we can ensure that the whole control flow of function f is protected from advising during the execution of program p , assuming a function $run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, similar to $run\mathbb{A}_{\mathrm{T}}$ (Sect. 3.2):

$$run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (EPC\ (pcCflow\ f))\ p$$

8.3 Privileged Aspects

Hiding some join points to *all* aspects may be too restrictive. For instance, certain "system" aspects like access control should be treated as privileged and view all join points. Another example is the aspect in charge of maintaining the join point stack for the sake of control flow reasoning (used by $pcCflow$). In such cases, it is important to be able to define a set of privileged aspects, which can advise all join points, even those that are normally hidden in a secure computation. The implementation of a privileged aspects list is a straightforward extension to the secure weaving mechanism described above.

8.4 Execution Levels

Execution levels avoid unwanted *computational interference* between aspects, *i.e.* when an aspect execution produces join points that are visible to others, including itself [41]. Execution levels give structure to execution by establishing a tower in which the flow of control navigates. Aspects are deployed at a given level and can only affect the execution of the underlying level. The execution of an aspect (both pointcuts and advices) is therefore not visible to itself and to other aspects deployed at the same level, only to aspects standing one level above. The original computation triggered by the last $proceed$ in the advice chain is always executed at the level at which the join point was emitted. If needed, the programmer can use level-shifting operators to move execution up and down in the tower.

The monadic semantics of execution levels are implemented in the $\mathbb{EL}_{\mathrm{T}}$ monad transformer (Fig. 8). The $Level$ type synonym represents the level of execution as an integer. $\mathbb{EL}_{\mathrm{T}}$ wraps a run function that takes an initial level and returns a computation in the underlying monad m, with a value of type a and a potentially-modified level. As in the $\mathbb{A}_{\mathrm{T}}$ transformer, the monadic $bind$ and $return$ functions are the same as in the state monad transformer. The private operations inc, dec, and at are used to define the user-visible operations $current$, up, $down$, and $lambda\_at$. In addition to level shifting with up and $down$, $current$ reifies the current level, and $lambda\_at$ creates a *level-capturing function* bound at level l. When such a function is applied, execution jumps to level l and then goes back to the level prior to the application [41].

The semantics of execution levels can be embedded in the definition of aspects themselves, by transforming the pointcut and advice of an aspect at deployment time, as shown in Fig. 9.[22] This is done by functions $pcEL$ and $advEL$. $pcEL$ first ensures that the current execution level $lapp$ matches $ldep$, the level at which the aspect is deployed. If so it then runs the pointcut one level above. Similarly, $advEL$ ensures that the advice is run one level above, with a $proceed$ function that captures the deployment level.

[22] For simplicity, in Sect. 3.2 we only described the default semantics of aspect deployment; aspect (un)deployment is actually defined using overloaded $(un)deployInEnv$ functions.

```
type Level = Int
newtype 𝔼𝕃T m a = 𝔼𝕃T (𝕊T Level m a)
  -- primitive operations
inc  = 𝔼𝕃T $ λl  → return ((), l + 1)
dec  = 𝔼𝕃T $ λl  → return ((), l − 1)
at l = 𝔼𝕃T $ λ_ → return ((), l)
  -- user-visible operations
current = 𝔼𝕃T $ λl → return (l, l)
up c     = do { inc; result ← c; dec; return result }
down c = do { dec; result ← c; inc; return result }
lambda_at f l = λarg → do n ← current
                          at l
                          result ← f arg
                          at n
                          return result
```

Fig. 8. Execution levels monad transformer and level-shifting operations

```
deployInEnv (Aspect (pc :: PC (𝔸T (𝔼𝕃T m)) tpc) adv) aenv =
  let
    pcEL ldep = (PC $ return $ λjp → do
      lapp ← current
      if lapp ≡ ldep then up $ runPC pc jp
                     else  return False) :: PC (𝔸T (𝔼𝕃T m)) tpc
    advEL ldep proceed arg = up $ adv (lambda_at proceed ldep) arg
  in do l ← current
        return EAspect (Aspect (pcEL l) (advEL l)) : aenv
```

Fig. 9. Redefining aspect deployment for execution levels semantics. An aspect is made level-aware by transforming its pointcut and advice.

Example Figure 10 defines a generic logging advice, $logAdv$, which appends the argument and result of advised functions to the log[23]. In $program$, we deploy an aspect that intercepts all calls to $showM$ (the monadic version of $show$) where the argument is of type Int (we require a type annotation for the pointcut because $showM$ is a bounded polymorphic function—see Sect. 9.2 for details).

The evaluation of the program depends on the instantiation of the monad stack M. In a setting without execution levels, advising $showM$ with $logAdv$ triggers an infinite loop because $logAdv$ internally performs open applications of $showM$, which are matched by the same aspect. Using the execution level semantics, evaluation terminates because the join point emitted by the advice is not visible to the aspect itself.

[23] Using the $tell$ function of the $MonadWriter$ class (denoted $\mathbb{W}_{\mathrm{M}}$), which is not described in Sect. 2, but which essentially is a state monad with append-only access.

$$
\begin{array}{l}
showM\ a = return\ (show\ a) \\
logAdv\ proceed\ a = \textbf{do}\ argStr \leftarrow showM \# a \\
\qquad\qquad\qquad\qquad tell\ (\texttt{"Arg: "} + argStr) \\
\qquad\qquad\qquad\qquad result \leftarrow proceed\ a \\
\qquad\qquad\qquad\qquad return\ result \\
program\ n = runM\ \$\ \textbf{do} \\
\quad deploy\ (aspect\ (pcCall\ (showM :: \to Int \to M\ String))\ logAdv) \\
\quad showM \# n
\end{array}
$$

Fig. 10. A program that loops unless execution levels are used.

Interestingly, explicit open applications limit the possibilities of unwanted advising. More obliviousness, *e.g.* through partial application of $\#$, makes it harder to track down these issues (we come back to obliviousness in Sect. 9.3). Nevertheless, identifying the source of the regression is not sufficient *per se*: in our example, if it is necessary for $logAdv$ to use open applications (so that other aspects can intervene), there is not much that can be done to avoid regression.

Beyond execution levels Execution levels adds a topological dimension to the composition of aspects into a system. However, their tower-like structure may be too restricted for certain scenarios, for instance for dynamic analyses aspects [43]. Recently, Tanter *et al.* proposed *programmable membranes* [44] as a generalization of execution levels. We have developed a prototype implementation of membrane semantics in Effective Aspects [12], using the same approach of converting pointcuts and advices at deployment time. However, instead of passing the current level of execution (an integer), we maintain the bindings between membranes (a graph) using a state monad.

8.5 Reasoning about Language Extensions

The above extensions can be implemented in an dynamically typed language such as LAScheme [41]. However, it is challenging to provide any kind of reasoning about effects due to the dynamic nature of the language.

Enforcing Non-Interference in Language Extensions We can combine the monadic interpretation of execution levels with the management of effect interference (Sect. 7) in order to reason about level-shifting operations performed by base and aspect computations. For instance, it becomes possible to prevent aspect and/or base computation to use effects provided by the $\mathbb{EL}_\mathrm{T}$ monad transformer, thus ensuring that the default semantics of execution levels is preserved (and therefore that the program is free of aspect loops [42]). For this we must consider a concrete monad stack that has the $\mathbb{A}_\mathrm{T}$ and $\mathbb{EL}_\mathrm{T}$ transformers on top:

$$\textbf{type}\ \mathbb{AEL}_\mathrm{T}\ m = \mathbb{A}_\mathrm{T}\ \mathbb{EL}_\mathrm{T}\ m$$

Observe that this monad stack is general with respect other effects it may contain. Then, we simply define an advice combinator that forbids access to the $\mathbb{EL}_T$ layer, which provides the level-shifting operations.

$$levelAgnosticAdv = withView\ (\square ::: \blacksquare ::: \square)$$

This mask hides the layer with the execution-level-related effects, but allows access to $\mathbb{A}_T$ at the top, and to the rest of the stack. Then to ensure level agnostic advice we just redefine *program* to use this combinator, in a suitable monad stack M:[24]

$$\begin{array}{l} \mathbf{type}\ M = \mathbb{AEL}_T\ (\mathbb{W}_T\ String\ \mathbb{I}) \\ runM\ c\ = run\mathbb{I}\ \$\ run\mathbb{W}_T\ \$\ run\mathbb{EL}_T\ (run\mathbb{A}_T\ c)\ 0 \\ program\ n = runM\ \$\ \mathbf{do} \\ \quad deploy\ (aspect\ (pcCall\ (showM :: \to Int \to M\ String)) \\ \qquad\qquad\qquad (levelAgnosticAdv\ logAdv)) \\ \quad showM\ \#\ n \end{array}$$

If more advanced use of execution levels is required, this contraint can be explicitly relaxed in the $\mathbb{A}_T$ or $\mathbb{EL}_T$ monad transformer, thus stressing in the type that it is the responsibility of the programmer to avoid infinite regression.

Using Types to Enforce Weaving Semantics The type system makes it possible to specify functions that can be woven, but only within a specific aspect monad. For instance, suppose that we want to define a *critical* computation, which must only be run with secure weaving for access control. The computation must therefore be run within the $\mathbb{A}_T^S$ monad transformer with a given pointcut pc_ac (ac stands for access control).

To enforce the use of $\mathbb{A}_T^S$ with a specific pointcut value would require the use of a dependent type, which is not possible in Haskell. This said, we can use the **newtype** data constructor together with its ability to derive automatically type class instances, to define a new type $\mathbb{A}_T^{AC}$ that encapsulates the $\mathbb{A}_T^S$ monad transformer and forces it to be run with the pc_ac pointcut:

$$\begin{array}{l} \mathbf{newtype}\ \mathbb{A}_T^{AC}\ m\ a = \mathbb{A}_T^{AC}\ (\mathbb{A}_T^S\ m\ a)\ \mathbf{deriving}\ (Monad, OpenApp, \ldots) \\ runSafe\ (\mathbb{A}_T^{AC}\ c) = run\mathbb{A}_T^S\ (EPC\ pc\_ac)\ c \end{array}$$

Therefore, we can export the *critical* computation by typing it appropriately:

$$critical :: Monad\ m \Rightarrow \mathbb{A}_T^{AC}\ m\ a$$

Because the $\mathbb{A}_T^{AC}$ constructor is hidden in a module, the only way to run such a computation typed as $\mathbb{A}_T^{AC}$ is to use *runSafe*. The *critical* computation is then only advisable with secure weaving for access control.

9 Discussion

We now discuss a number of issues related to our approach: how to define a proper notion of function equality, how to deal with overloaded functions, and finally, we analyze the issue of obliviousness.

[24] We use the $WriterT$ transformer ($\mathbb{W}_T$), which is the canonical instance of $\{\mathbb{W}_M$.

9.1 Supporting Equality on Functions

Pointcuts quantify about join points, and a major element of the join point is the function being applied. The $pcType$ designator relies on type comparison, implemented using the $PolyTypeable$ type class in order to obtain representations for polymorphic types. The $pcCall$ is more problematic, as it relies on function equality, but Haskell does not provide an operator like eq? in Scheme.

A first workaround is to use the $StableNames$ API that allows comparing functions using pointer equality. Unfortunately, this notion of equality is fragile. $StableNames$ equality is safe in the sense that it does not equate two functions that are not the same, but two functions that are equal can be seen as different.

The problem becomes even more systematic when it comes to bounded polymorphism. Indeed, each time a function with constraints is used, a new closure is created by passing the current method dictionary of type class instances. Even with optimized compilation (*e.g.* ghc -O), this (duplicated) closure creation is unavoidable and so $StableNames$ will consider different any two constrained functions, even if the passed dictionary is the same.

To overcome this issue, we have overloaded our equality on functions with a special case for functions that have been explicitly tagged with a unique identifier at creation (using $Data.Unique$). This allows us to have a robust notion of function equality but it has to be used explicitly at each function definition site.

9.2 Advising Overloaded Functions

From a programmer's point of view, it can be interesting to advise an overloaded function (that is, the application of all the possible implementations) with a single aspect. However, deploying aspects in the general case of bounded polymorphism is problematic because of the resolution of class constraints. Recall that in order to be able to type the aspect environment, we existentially hide the matched and advised types of an aspect. This means that all type class constraints must be solved statically at the point an aspect is deployed. If the matched and advised types are both bounded polymorphic types, type inference cannot gather enough information to statically solve the constraints. So advising all possible implementations requires repeating deployment of the same aspect with different type annotations, one for each instance of the involved type classes.

To alleviate this problem, we developed a macro using TemplateHaskell [36]. The macro extracts all the constrained variables in the matched type of the pointcut, and generates an annotated deployment for every possible combination of instances that satisfy all constraints. In order to retain safety, the advised type of an aspect must be less constrained than its matched type. This is statically enforced by the Haskell type system after macro expansion.

9.3 Obliviousness

The embedding of aspects we have presented thus far supports quantification through pointcuts, but is not oblivious: open applications are explicit in the code. A first way to introduce more obliviousness without requiring non-local macros or, equivalently, a preprocessor or ad hoc runtime semantics, is to use *partial applications* of $\#$. For instance, the $enqueue$ function can be turned into an implicitly woven function by defining $enqueue' = enqueue\ \#$. This approach was used in Fig. 2 for the definition of fib. It can be sufficient in similar scenarios where quantification is under control. Otherwise, it can yield issues in the definition of pointcuts that rely on function identity, because $enqueue'$ and $enqueue$ are different functions. Also, this approach is not entirely satisfactory with respect to obliviousness because it has to be applied specifically for each function.

De Meuter proposes [26] to use the binder of a monad to redefine function application. His approach focuses on defining one monad per aspect, but can be generalized to a list of dynamically-deployed aspects as presented in Sect. 3.2. For this, we can redefine the monad transformer $\mathbb{A}_\mathrm{T}$ to make all monadic applications open transparently:

$$
\begin{array}{l}
\mathbf{instance}\ Monad\ m \Rightarrow Monad\ (\mathbb{A}_\mathrm{T}\ m)\ \mathbf{where} \\
\quad return\ a = \mathbb{A}_\mathrm{T}\ \$\ \lambda aenv \rightarrow return\ (a, aenv) \\
\quad k \gg\!= f \quad = \mathbf{do}\ x \leftarrow k \\
\qquad\qquad\qquad\quad f\ \#\ x
\end{array}
$$

This presentation improves obliviousness because any monadic application is now an open application, but it suffers from a major drawback: it breaks the *monadic laws*. Indeed, left identity and associativity

-- Left identity:
$return\ x \gg\!= f = f\ x$
-- Associativity:
$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$

can be invalidated, depending on the current list of deployed aspects. This is not surprising as AOP allows one to redefine the behavior of a function and even to redefine the behavior of a function depending on its context of execution. Breaking monadic laws is not prohibited by Haskell, but it is very dangerous and fragile; for instance, some compilers exploit the laws to perform optimizations, so breaking them can yield incorrect optimizations.

9.4 Technical Requirements of our Model

The current implementation of Effective Aspect uses several extensions of the GHC Haskell compiler (see the details at http://plead.cl/effectiveaspects). Nevertheless, we believe that the anti-unification algorithm at the type level (Section 4.1) is the essential feature that would be required to make our approach work on other languages. A potential line of work is to port Effective Aspects to Scala, which has some likeness to Haskell and also has monads, and investigate what kind of issues arise in the process.

10 Related Work

The earliest connection between aspects and monads was established by De Meuter in 1997 [26]. In that work, he proposes to describe the weaving of a given aspect directly in the binder of a monad. As we have just described above (Sect. 9.3), doing so breaks the monad laws, and is therefore undesirable.

Wand *et al.* [48] formalize pointcuts and advice and use monads to structure the denotational semantics; a monad is used to pass the join point stack and the store around evaluation steps. The specific flavor of AOP that is described is similar to AspectJ, but with only pure pointcuts. The calculus is untyped. The reader may have noticed that we do not model the join point stack in this paper. This is because it is not *required* for a given model of AOP to work. In fact, the join point stack is useful only to express control flow pointcuts. In our approach, this is achieved by specifying a user-defined pointcut designator for control flow, which uses a monad to thread the join point stack (or, depending on the desired level of dynamicity, a simple control flow state [24]). Support for the join point stack does not have to be included as a primitive in the core language. This is in fact how AspectJ is implemented [24,15].

Hofer and Osterman [16] shed some light on the modularity benefits of monads and aspects, clarifying that they are different mechanisms with quite different features: monads do not support declarative quantification, and aspects do not provide any support for encapsulating computational effects. In this regard, our work does not attempt at unifying monads and aspects, contrary to what De Meuter suggested. Instead, we exploit monads in Haskell to build a flexible embedding of aspects that can be modularly extended. In addition, the fully-typed setting provides the basis for reasoning about monadic effects.

The notion of *monadic weaving* was described by Tabareau [38], where he shows that writing the aspect weaver in a monadic style paves the way for modular language extensions. He illustrated the extensibility approach with execution levels [41] and level-aware exception handling [13]. The authors then worked on a practical monadic aspect weaver in Typed Racket [14]. However, the type system of Typed Racket turned out to be insufficiently expressive, and the top type Any had to be used to describe pointcuts and advices. This was the original motivation to study monadic weaving in Haskell. Also in contrast to this work, prior work on monadic aspect weaving does not consider a base language with monads. In this paper, both the base language and the aspect weaver are monadic, combining the benefits of type-based reasoning about effects (Sect. 6) and modular language extensions (Sect. 8)—including type-based reasoning about language extensions.

Haskell has already been the subject of AOP investigations using the type class system as a way to perform static weaving [37]. AOP idioms are translated to type class instances, and type class resolution is used to perform static weaving. This work only supports simple pointcuts, pure aspects and static weaving, and is furthermore very opaque to modular changes as the translation of AOP idioms is done internally at compile time.

The specific flavor of pointcut/advice AOP that we developed is directly inspired by AspectScheme [11] and AspectScript [45]: dynamic aspect deployment, first-class aspects, and extensible set of pointcut designators. While we have not yet developed the more advanced scoping mechanisms found in these languages [40], we believe there are no specific challenges in this regard. The key difference here is that these languages are both dynamically typed, while we have managed to reconcile this high level of flexibility with static typing.

In terms of statically-typed functional aspect languages, the closest proposal to ours is AspectML [7]. In AspectML, pointcuts are first-class, but advice is not. The set of pointcut designators is fixed, as in AspectJ. AspectML does not support: advising anonymous functions, aspects of aspects, separate aspect deployment, and undeployment. AspectML was the first language in which first-class pointcuts were statically typed. The typing rules rely on anti-unification, just like we do in this paper. The major difference, though, is that AspectML is defined as a completely new language, with a specific type system and a specific core calculus. Proving type soundness is therefore very involved [7]. In contrast, we do not need to define a new type system and a new core calculus. Type soundness in our approach is derived straightforwardly from the type class that establishes the anti-unification relation. Half of section 5 is dedicated to proving that this type class is correct. Once this is done (and it is a result that is independent from AOP), proving aspect safety is direct. Another way to see this work is as a new illustration of the expressive power of the type system of Haskell, in particular how phantom types and type classes can be used in concert to statically type embedded languages.

Aspectual Caml [25] is another polymorphic aspect language. Interestingly, Aspectual Caml uses type information to influence matching, rather than for reporting type errors. More precisely, the type of pointcuts is inferred from the associated advices, and pointcuts only match join points that are valid according to these inferred types. We believe this approach can be difficult for programmers to understand, because it combines the complexities of quantification with those of type inference. Aspectual Caml is implemented by modifying the Objective Caml compiler, including modifications to the type inference mechanism. There is no proof of type soundness.

The advantages of our typed embedding do not only lie within the simplicity of the soundness proof. They can also be observed at the level of the implementation. The AspectML implementation is over 15,000 lines of ML code [7], and the Aspectual Caml implementation is around 24,000 lines of Objective Caml code [25]. In contrast, our implementation, including the execution levels extension (Sect. 8), is only 1,600 lines of Haskell code. Also, embedding an AOP extension entirely inside a mainstream language has a number of practical advantages, especially when it comes to efficiency and maintainability of the extension.

Finally, reasoning about advice effects has been studied from different angles. For instance, harmless advice can change termination behavior and use I/O, but no more [6]. A type and effect system is used to ensure conformance. Translucid contracts use grey box specifications and structural refinement in verification to reason about control effects [5]. In this work, we rather follow the type-based approach of EffectiveAdvice (EA) [28], which also accounts for various control effects and arbitrary computational effects. A limitation of EA is its lack of support quantification. A contribution of this work is to show how to extend this approach to the pointcut/advice mechanism. The subtlety lies in properly typing pointcuts. An interesting difference between both approaches is that in EA, it is not possible to talk about "the effects of all applied advices". Once an advice is composed with a base function, the result is seen as a base function for the following advice. In contrast, our approach, thanks to the aspect environment and dynamic weaving, makes it possible to keep aspects separate and ensure base/aspect separation at the effect level even in presence of multiple aspects. We believe that this splitting of the monad stack is more consistent with programmers expectations.

11 Conclusion

We develop a novel approach to embed aspects in an existing language. We exploit monads and the Haskell type system to define a typed monadic embedding that supports both modular language extensions and reasoning about effects with pointcut/advice aspects. We show how to ensure type soundness by design, even in presence of user-extensible pointcut designators, relying on a novel type class for establishing anti-unification. Compared to other approaches to statically-typed polymorphic aspect languages, the proposed embedding is more lightweight, expressive, extensible, and amenable to interference analysis. The approach can combine Open Modules and EffectiveAdvice, and supports type-based reasoning about modular language extensions.

Acknowledgments. This work was supported by the INRIA Associated team REAL. We thank the anonymous reviewers from the AOSD'13 conference and from this journal, and we also thank Tom Schrijvers for all his useful feedback.

References

1. *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, Potsdam, Germany, Mar. 2012. ACM Press.
2. 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.
3. *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, Apr. 2008. ACM Press.
4. *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.
5. 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.
6. 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.
7. 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.
8. B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In AOSD 2008 [3], pages 60–71.

9. R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.
10. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In A. Yonezawa and S. Matsuoka, editors, *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, Sept. 2001. Springer-Verlag.
11. 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.
12. I. Figueroa, N. Tabareau, and É. Tanter. Taming aspects with monads and membranes. In *Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2013)*, pages 1–6, Fukuoka, Japan, Mar. 2013. ACM Press.
13. I. Figueroa and É. Tanter. A semantics for execution levels with exceptions. In *Proceedings of the 10th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2011)*, pages 7–11, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.
14. I. Figueroa, É. Tanter, and N. Tabareau. A practical monadic aspect weaver. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)* [1], pages 21–26.
15. E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, Mar. 2004. ACM Press.
16. C. Hofer and K. Ostermann. On the relation of aspects and monads. In *Proceedings of AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007)*, pages 27–33, 2007.
17. M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, number 1782 in Lecture Notes in Computer Science, pages 230–244. Springer-Verlag, 2000.
18. 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.
19. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.
20. Learn you a haskell website, 2013. http://learnyouahaskell.com/.
21. D. Leijen and E. Meijer. Domain specific embedded compilers. In T. Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.
22. 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.
23. H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, Nov. 2003.
24. 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.

25. H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005)*, pages 320–330, Tallin, Estonia, Sept. 2005. ACM Press.
26. W. D. Meuter. Monads as a theoretical foundation for aop. In *In International Workshop on Aspect-Oriented Programming at ECOOP*, page 25. Springer-Verlag, 1997.
27. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
28. B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [4], pages 109–120.
29. 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.
30. S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, Jan. 2007.
31. B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
32. G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.
33. J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.
34. 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.
35. 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.
36. T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.
37. M. Sulzmann and M. Wang. Aspect-oriented programming with type classes. In *Proceedings of the Sixth Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007)*, pages 65–74, Vancouver, British Columbia, Canada, Mar. 2007. ACM Press.
38. N. Tabareau. A monadic interpretation of execution levels and exceptions for AOP. In É. Tanter and K. J. Sullivan, editors, *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012)*, Potsdam, Germany, Mar. 2012. ACM Press.
39. 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.
40. É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD 2008 [3], pages 168–179.
41. É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [4], pages 37–48.
42. É. Tanter, I. Figueroa, and N. Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. *Science of Computer Programming*, 2013. Available online.
43. 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.
44. 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)* [1], pages 3–8.

45. R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [4], pages 13–24.
46. 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.
47. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 60–76, Austin, TX, USA, Jan. 1989. ACM Press.
48. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, Sept. 2004.