On the relation of aspects and monads

Christian Hofer and Klaus Ostermann
Computer Science Department
Darmstadt University of Technology, Germany

ABSTRACT

The relation between aspects and monads is a recurring topic in discussions in the programming language community, although it has never been elaborated whether their resemblences are only superficial, and if not, where they are rooted. The aim of this paper is to contrast both mechanisms w.r.t. their capabilities and their effects on modularity, first by looking at monads as a way to express tangling concerns in functional programming and by discussing whether they can be regarded as a form of AOP, then by taking the view that monads express concerns of computations and by analyzing the extent to which aspects are able to handle those concerns.

Our results are mostly negative: monads are not capable of quantifying over points in the program execution in a declarative way, whereas aspects are not very useful in abstracting over computational capabilities.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logic and Meanings of Programs**]: Studies of Program Constructs

General Terms

Design, Languages

Keywords

aspect-oriented programming, aspects, monads, monad transformers

1. INTRODUCTION

Since De Meuter [9] has first discovered resemblences between aspects and monads on the descriptive level – layering of code, system wide repercussions, easy integration – those are a recurring topic in discussions in the programming language community, although it has never been elaborated whether they are only superficial, and if not, where they are rooted. This paper tries to shed some light on this topic by contrasting the two mechanisms w.r.t. to their capabilities and their effects on modularity.

We first specify what we regard as the essence of each of the concepts. Then, in the next section, we will look at monads as a way to express tangling concerns in functional programming, and discuss the extent to which monadic programming can be regarded as a form of AOP. We therefore analyze a monads-based implementation of the display update example as used by Kiczales and Mezini [5] (and many others).

Finally, we take the view that monads express concerns of computations and we analyze the extent to which aspects are able to handle those concerns. This discussion is mainly theoretical, however a monadic interpreter (see Liang et al. [6]) for a small functional language with dynamic variable binding will be simulated in AspectJ to demonstrate the use of dynamic quantification.

Although this work started as a project to identify the commonalities of aspects and monads, it turned out that the differences prevail. Based on ones point of view, this may or may not be very surprising, but since the topic is repeatedly brought up in blogs or discussions, we believe there is some value in substantiating the debate.

In the following, we assume basic familiarity with monads and monad transformers as available in Haskell [11], and familiarity with AspectJ [3].

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ...$5.00.

1.1 Aspect-oriented programming

The aim of AOP is generally uncontested: the separation of cross-cutting concerns, i.e. a better source code organization that prevents that concerns are *scattered* around the source code, and complementarily, that several concerns are *tangled* at single places.

It is less clear, how to characterize the actual mechanisms for achieving this aim. Filman/Friedman [1] have given the famous characterization of AOP as "quantification and obliviousness", meaning that it allows the triggering of actions whenever a specified condition arises in a program (quantification), without the knowledge of the programmer (obliviousness). But as obliviousness is in conflict with the principle of explicit interfaces, and there is a case for non-oblivious quantification as well, this definition has been seen as too restrictive. On the other hand, the definition of Masuhara/Kiczales, requiring "a common frame of reference that two (or more) programs can use to connect with each other and each provide their semantic contribution" [8] seems too general to delimit AOP from other kinds of module systems.

Another characteristic of current AOP mechanisms, the "introduction of declarative policy languages" [12], is left implicit in both definitions, but will turn out to be helpful in drawing a line to "classical" module systems.

For the further analysis, we will pragmatically define a mechanism as aspect-oriented, if it aims at the separation of concerns in the organization of source code, by making use of declarative quantification and by finding a balance between the contradictory principles of obliviousness and of explicit interfaces.

1.2 Monads

Monads, besides being a notion of category theory and a means of defining a categorical semantics of computations [10], are a mechanism in functional programming that (1) allows the introduction of imperative statements, like stateful computations or exceptions, into purely functional languages, and (2) provides a way to abstract over different kinds of computations. We want to recapitulate briefly, how this abstraction works.

A computation can be regarded as a kind of function that will typically produce a value. Each kind of computation is characterized by a specific structure of parameters and return values. It can be defined by a type constructor that – together with the value typically produced by a computation – defines the type of this computation.

For example, the type constructor `Maybe` defines computations that typically produce values, but may fail. The types of this kind of computation can be defined polymorphically as:

```
data Maybe a = Just a | Nothing
```

There are two important operations that go with every kind of computation: (1) several computations of the same kind can be put together into a sequence; (2) a value can be injected into a computation with the effect that this value will be returned by the computation when it is run. Haskell provides a specific monadic (`do`) notation that is useful for putting together computations. Let us consider a simple example of a comparison of two values associated with two keys in a database:

```
eqVal key1 key2 db =
   do val1 <- lookup key1 db
      val2 <- lookup key2 db
      return (val1 == val2)
```

This is a sequence of three computation steps, the first will lookup `key1` and will typically produce a value `val1`, but may fail. The second step works analagously. The third computation compares both values: the boolean value that results from the comparison is injected into the kind of computation. For the `Maybe` monad, the `return` operator is the `Just` constructor, so if the computation arrives at the third computation step, the result is either `Just True` or `Just False`. The sequencing of computations is left implicit in the `do` notation. A `bind` operator binds the value returned by the first computation to a variable that can be accessed by the latter computations. If a computation step fails in the `Maybe` monad, the complete sequence is aborted, and `Nothing` is returned.

The monad abstracts over the possible failure of the computation. The programmer is oblivious to what happens behind the scenes in two regards: (1) she does not have to worry about the specific structure of parameters and return values, but can simply inject values into the computation where needed; (2) she does not have to worry about how the effects of one step are passed to the next computation steps (in this case, this concerns only the abortion of the complete computation; but e.g. in the `State` monad, the passing of the state through all computations is hidden behind the scenes.)

Different monads can be combined via monad transformers in order to express more complex kinds of computations. The different concerns of these computations are separated into the individual monads.

Modularity effects.

The addition of computational capabilities to some operations is the cause of a severe modularity problem in functional programming: often the parameter structure of a whole set of functions within a program has to be adapted to reflect this additional capability. Using monads, the parameter structure can be encapsulated together with the operations that make use of it (in the case of the `Maybe` monad, the only such operation is the `fail` operation that is associated with the `Nothing` constructor). In that way, monadic programming allows the careful inclusion of specific computational powers into a program, while keeping the different kinds of computations modularized.

The price to pay for this is referential transparency, if one takes the `do` notation of Haskell literally. For example, the result of the `get` operation in the state monad is dependent on the context, although it appears not to take any input parameters. Of course, if one takes the hidden `bind` operator into accout, it is still valid to reason with value substitution.

2. TANGLING CONCERNS IN FP

The tangling of concerns is not restricted to imperative programming, but prevalent in functional programming as well. Its simplest form is the *side effect*. But tangling concerns can go together in different ways. A transaction concern e.g. could rewind a computation. Monads are a natural starting point, if one implements those tangled concerns in a purely functional programming language. They allow for the production of side effects as well as for some control of execution, hiding all those computational details from the base functionality.

In the following section, we want to present an implementation of the display update example in Haskell using monadic programming. We want to discuss the capabilities of monads regarding the modularization of the cross-cutting concerns inherent in this example. The core of the display update example – as it is discussed e.g. in [4] – is a module that defines two simple shapes, points and lines, on a two-dimensional cartesian coordinate system. All shapes are updateable structures. In particular, they all have a `move` operator, that moves the shape along both coordinates. This module is complemented by an aspect that is responsible for performing a display refresh, whenever a shape on the display is updated.

The whole example rests on an imperative programming foundation, regarding the shapes as stateful objects. It shall not be discussed here, whether this implementation is the most natural for a functional programming language. As the programming style is monadic, anyway, the imperativeness of the task is no hindrance to the implementation and can still show the intricacies involved in handling tangling concerns.

Display module.

We do not want to discuss the display module in detail, but we cannot omit it completely: in contrast to the Java solution we have to be much more specific on which computational powers it shall have. The display functions are implemented via an IOable monad (i.e. a monad that implements a `liftIO` function – belonging to the `MonadIO` type class – that lifts an IO operation to the monad). Using the style of [2], we define the interface of the display functions via a type class, while giving an implementation using a state monad transformer.[1]

```
class MonadIO m => MonadDisplay m where
   setDisplay :: DisplayObject -> m ()
   refreshDisplay :: m ()

type DisplayT = StateT DisplayObject

instance MonadIO m => MonadDisplay (DisplayT m)
   ...
```

The `setDisplay` function can be used for specifying an object to be displayed, the `refreshDisplay` function has to be called, whenever some information on the display changed so that the information has to be refreshed.

An object that shall be displayed must be made an instance of the `Displayable` class and implement the `display` function.

```
class Displayable a where
    display :: MonadIO m => a -> m ()

data DisplayObject =
   forall a. Displayable a => DisplayObject a
```

The use of the `forall` quantifier in the data type `DisplayObject` can be regarded as a trick to ensure polymorphism over all `Displayable` objects in the `setDisplay` function.

Shapes module.

The basic shapes functionality is implemented imperatively, in the example. `IORef`s are used as references to memory cells: a point is a reference to a pair of integers, a line is a reference to a pair of points. Due to the use of `IORef`s, all computations have to take place in an IOable monad. The constructor `newPoint` and the accessor `getPointX` are examples of functions that do not perform a state change and solely depend on the `MonadIO` class.

```
newtype Point = P (IORef (Int, Int))

newPoint :: MonadIO m => Int -> Int -> m Point
newPoint x y =
    do p <- liftIO $ newIORef (x, y)
       return (P p)
```

[1]The full code is available at `http://www.st.informatik.tu-darmstadt.de:8080/~ostermann/foal07/`

```
getPointX :: MonadIO m => Point -> m Int
getPointX (P p) =
    do (x,_) <- liftIO $ readIORef p
       return x
```

In contrast, the `movePointBy` function has to trigger a display refresh. The straightforward way to integrate the shapes functionality with the display functionality is to import the `Display` module and add a call to `refreshDisplay` at the end of all operations that perform a state change. The `movePointBy` function then looks like this:

```
movePointBy (P p) dx dy =
    do liftIO $ modifyIORef p
                 (\(x, y) -> (x+dx, y+dy))
       refreshDisplay
```

The type signature for this function is inferred as:

```
movePointBy :: (MonadDisplay m) =>
               Point -> Int -> Int -> m ()
```

Analogously, all the other state modifying functions have to run in a monad encompassing the display state. For making points and lines displayable, they have to be declared instances of the class `Displayable`.

2.1 Obliviousness

This solution is very similar to a typical object-oriented implementation (see e.g. the “GOFP” solution in [5]). The base functionality is tangled with calls to the display module. The monadic style allows us to abstract from the current display state that would otherwise have to be passed around through the base code. But it does not allow us to separate out the call triggering the display refresh.

In AspectJ we can externally define a pointcut that triggers the display refresh and that the programmer of the base functionality is oblivious of. We cannot achieve obliviousness with monads. The module boundaries are clearly respected, and there is no way to reflect over the names of the computations. The situation would be different, if monads were used to implement an interpreter for an AO language, because inside an interpreter reflective access is possible. Indeed, Wand et al. [13] have defined a monadic semantics of a pointcut mechanism: each procedure call takes place within a join-point environment that is extended by the name of the currently called procedure. This access to the procedure name is not available in a direct implementation. Adding reflection to a language with monads, on the other hand, runs the risk of breaking not only modularity, but as well the monad laws.

2.2 Non-oblivious AOP?

As it has been argued that obliviousness is not essential for AOP, we can try to achieve some form of non-oblivious separation of concerns. We first have to specify, which concerns are involved in the example. We will follow Kiczales/Mezini who have analyzed three further concerns besides the refresh implementation and the base functionality:

“**Context-to-Refresh** – What context from the actual display state change points should be available to the refresh implementation?

“**When-to-Refresh** – When should the display be refreshed?

“**What-Constitutes-Change** – What operations change the state that affects how shapes look on the display, i.e. their position?” [5]

The separation of the “context to refresh” concern makes use of obliviousness in the implementations discussed in [5]. A static pointcut like `this`, `target` or `args` has to be used within the aspect module in order to query the relevant information from a single place. The alternative would be to pass along the information with the `refreshDisplay` call. That would leave the concern scattered within the base functionality. While a reader monad could avoid the explicit parameter passing, the environment to be supplied to the display module has still to be defined somewhere in the state change operations of the base code. Thus, the scattering cannot be avoided in that way.

The distinction between the concerns “when to refresh” and “what constitutes change” can be made in the monadic program, and the former concern can thereby be modularized. For this purpose, the call of `refreshDisplay` is omitted and the state changing code is embedded into the call of a function `withStateChangeSignal`, instead.

```
movePointBy (P p) dx dy = withStateChangeSignal $
    liftIO $ modifyIORef p (\(x, y) -> (x+dx, y+dy))
```

The `withStateChangeSignal` takes a computation as parameter, and embeds the computation into the sending of a signal.[2] This allows the receiver of the signal to attach other computations before, after or around (including: instead of) the original computation. This is in contrast to firing an event, which would be the classical OO solution.[3]

The signal is declared via a type class `MonadStateChange`, which in effect globalizes the declaration and permits to define an implementation in some module that is not imported by the shapes module.

```
class MonadIO m => MonadStateChange m where
    withStateChangeSignal :: m a -> m a
```

The type of the `movePointBy` function is inferred as:

```
movePointBy :: (MonadStateChange m) =>
               Point -> Int -> Int -> m ()
```

We leave the concern of “what constitutes change” tangled with the base functionality. We could separate the latter by factoring out the former into a module working as a proxy, that simply passes on the operations while accompanying those operations that perform a state change with the corresponding signal. The viability of this approach depends on the power to redirect to the proxy the calls to the shapes module by its clients, that – if it is to be done in a non-oblivious way – requires a powerful module system. To which extent the Haskell type class system is apt for this task, cannot be discussed here.

But even then, the tangling of concerns would only be shifted to the proxy. Tangling is inevitable, because in monadic programming there is no mechanism for what Filman/Friedman [1] call “static quantification”: we cannot make quantified statements over the program text, in the sense of making statements that have an effect on more than one place in the elaborated program (see [1]).

[2]The `with` prefix is adapted from a Lisp macro convention.
[3]However a similar effect could be achieved in Java by encapsulating the state modifying code into an anonymous class.

2.3 Declarativeness

A separate module is responsible for the integration of the shapes and the display modules. It defines the `Displayable` instances for the shapes, and it implements the “when to refresh?” concern by defining an instance of the `MonadStateChange` class:

```
instance MonadDisplay m =>
         MonadStateChange m where
    withStateChangeSignal c =
        do result <- c
           refreshDisplay
           return result
```

The implementation evaluates the computation that has been provided and refreshs the display thereafter. However, in contrast to an AspectJ implementation (that depends on a `displayStateChange()` pointcut; adapted from [5]):

```
after() returning: displayStateChange() {
   Display.refresh();
}
```

it is obvious that the definition of when the display shall be refreshed is not done in a declarative manner.

2.4 Dynamic quantification

However, it is possible to implement what Filman/Friedman call “dynamic quantification”, the tying of “aspect behavior to something that happens at run-time” [1]. In the current implementation, the `moveLineBy` code looks like this:

```
moveLineBy :: MonadStateChange m =>
              Line -> Int -> Int -> m ()
moveLineBy (L l) dx dy = withStateChangeSignal $
    do (p1, p2) <- liftIO $ readIORef l
       movePointBy p1 dx dy
       movePointBy p2 dx dy
```

As `movePointBy` signals a state change as well, the signal is sent three times. We can, however, adapt our implementation of the `withStateChangeSignal` function, in order to prevent a repeated display refresh:

```
type StateChangeT = ReaderT Bool

instance MonadDisplay m =>
 MonadStateChange (StateChangeT m) where
    withStateChangeSignal c =
        do result <- local (\_ -> True) c
           p <- ask
           unless p (lift refreshDisplay)
           return result
```

The monad is adapted by transforming the display monad through a reader monad over a boolean flag. The flag signals whether a need to refresh the display has already been registered by a surrounding computation. The `withStateChange` computation first executes the computation that has been provided as its parameter and keeps the result. This execution is embedded into an environment where the flag set, because the function itself takes responsibility for the display refresh. Afterwards, the computation will check its own environment, whether the flag is set, and trigger a display refresh otherwise. In any case, the kept result is returned.

The instance declaration ensures that the display monad transformer and the reader monad transformer of the integrating module are combined. The order of combination is irrelevant when combining a state and a reader monad transformer, but the implementation could potentially break if we were using another implementation of the display monad that would not just encapsulate a state monad.

When looking at the examples that Filman/Friedman [1] give for dynamic quantification, it is apparent, that some of them correspond directly to monads: raising of exceptions (error monad), calling a subprogram in temporal scope of another operation (reader monad), the history of the program execution (state monad). Furthermore, the authors note that AOP variants of other programming languages may include other ways of dynamic quantification, due to their native language features, and name the capturing of the current continuation in Scheme as an example (continuation monad).

On the other hand, monadic programming only allows non-oblivious dynamic quantification, i.e. quantification over properties that are captured explicitly by a monadic operation (that works in that regard as a semantic marker), while the typical use of e.g. the `cflowbelow` pointcut descriptor is quantification over the control flow based on syntactic names.

In addition, the monadic solution to the redundant display refresh problem is not declarative. It uses a sequential style for implementing the concern.

2.5 Advice confinement

Monads allow for a controlled extension of computational capabilities. Therefore we expect the handling of tangling concerns to be more controlled than in the AspectJ solution. Indeed, the Haskell type system gives us some guarantees on what part of the program the advice may affect. We know e.g. that the display refresh code cannot trigger a state change signal by some operation that it calls, because it does not run in an instance of the `MonadStateChange` monad. While this confinement of the powers of advice can simplify reasoning about the program, it may on the other hand be regarded as an unwanted restriction on the programmer's flexibility.

2.6 Conclusion

Monads are a common way to handle tangling concerns in purely functional programming. They are a traditional way to modularize computations in that they respect the module interfaces. They can therefore not achieve *oblivious* quantification. Furthermore, they differ from AOP by not allowing for *declarative* quantification. However they are similar to (a certain type of) AOP and more powerful than traditional module systems in one regard: they provide the abstractions that characterize *dynamic* quantification.

They appear to be similar to annotation-based AOP in that they are more powerful than a traditional modular solution, while remaining non-oblivious. Two differences to the annotation-based AspectJ solution are apparent, however: (1) the monadic solution does not allow for declarative quantification; (2) the AspectJ solution still encompasses a reflection mechanism via the `target`, `this`, and `args` pointcuts that break into the module implementation and allow for separation of the "context to refresh" concern.

3. ASPECTS OF COMPUTATIONS

While above monads were used to encapsulate specific tangling concerns, it can be argued that every monad can be regarded as expressing a concern: the kind of a computation can be regarded as an aspect of the computation. Based on this assumption, we want to analyze, to what extent aspects might be able to fulfill the role of monads in abstracting over kinds of computations. But we also want to shortly discuss, if the power of AOP to separate concerns were useful in monadic programming.

3.1 Abstracting over kinds of computations

At the heart of monads lie the two fundamental operators: the *return* operator that injects values into computations and allows the programmer to abstract over the parameter structure of the actual monad, and the *bind* operator that organizes the sequencing of computations. There are no equivalents for those operators in AOP. AOP is not about redefining the way that the sequencing of operations is interpreted. Instead it is about introducing additional action at specific points in the course of execution. This imposes severe limitations in the way that AOP mechanisms can manipulate the flow of control and enrich the parameter structure.

3.1.1 Manipulating the control flow

In AspectJ, after advice has been executed at some join-point, the execution is resumed after the join-point. There is no way to jump forward to some join-point matching another pointcut, or up to some position in the call stack. The only way to achieve the latter is by throwing an exception within the advice code, and by adding exception catching advice at the position where execution shall continue. But throwing of exceptions is not a mechanism introduced by AspectJ, but belongs to the mechanisms of the base language.

Generally, aspect languages seem not to provide mechanisms that allow the programmer to explicitly manipulate the control flow. Thus, we cannot hope to express computations that are able to fail, like those represented by the `Maybe`, the `Error`, or the `List` monad, via AO mechanisms. We will have to use exception mechanisms to jump out of the current point in program execution. The exception mechanism makes equal the very different kinds of computations expressed via the different monads, and therefore can hardly count as a good abstraction mechanism over them. W.r.t. the `Continuation` monad, if one does not restrict oneself to escape continuations, the situation is even worse, as the exception mechanism will probably not be powerful enough.

3.1.2 Enriching the parameter structure

An important part of abstracting over kinds of computation is associated with the ability to abstract over the parameter structure of a computation. As we do not have the power to inject values into computations, or to pass the hidden parameters along in AOP, we cannot expect to have a general mechanism for this kind of abstraction. The sequencing of computations can only be translated into a sequence of programming statements in our base language.

On the other hand, some of the powers offered by monads might already be included as part of our base language. For example, in Java there is no need to separate out the passing around of a state through the execution sequence, in the way it is done by the `State` monad. It might be argued that AOP mechanisms allow the programmer to pass along state by introducing it within a separate module, e.g. by the inter-type declarations of AspectJ. But this is a redundant mechanism for abstracting over state, and is only useful for separating out a concern that is related to the state.

However there is a way to implement some hidden form of parameter passing in some AOP mechanisms, by using a combination of *dynamic* quantification and reflection.

3.1.3 Using dynamic quantification

The most prominent mechanism for dynamic quantification is the `cflow` pointcut in AspectJ. It allows for quantification over the control context and may be able to simulate powers of the `Reader` monad. We want to focus the discussion on this mechanism.

The reader monad is typically associated with a function to transform the environment of a control context (`local`) and a function to read the environment within a control context (`ask`). In order to simulate this in AspectJ, we have to define a pointcut that matches the point where the environment is transformed, and a pointcut that matches the point where the environment is demanded. Calling them `local()` and `ask()`, the pointcut for reading the environment can be defined as:

```
pointcut readEnvironment(Environment env):
   ask() && cflow(local(env));
```

The remaining problem is how to define the `local()` pointcut such that it contains the environment as a variable. First of all, it must be noted that due to the workings of the `cflow` pointcut, we can only access the innermost join-point that matches the `local()` pointcut in the context. Therefore, the complete environment must be made available there. We are able to collect the environment information by using one of the state-based pointcuts `this()`, `target()`, and `args()`.

A typical application of the reader monad is its use for keeping a variable environment in a programming language interpreter. Although it can be used for statically scoped variables, it is more naturally used for implementing (the less wide-spread) dynamic variable binding.

Let us look at a simple interpreter implementing dynamic binding for illustration purposes. A monadic interpreter written in Haskell could look like this:

```
data Term = Const Int
          | Var String
          | Lambda String Term
          | App Term Term

data Value = Num Int
           | Fun (Value -> Reader Environment Value)

interpret :: Term -> Reader Environment Value
interpret (Const c) = return (Num c)
interpret (Var varId) =
   do env <- ask
      return (lookupVar varId env)
interpret (Lambda varId body) = return $
   Fun $ \val -> local (\env -> (varId, val) : env)
                       (interpret body)
interpret (App e1 e2) =
   do Fun f <- interpret e1
      v <- interpret e2
      f v
```

In a non-monadic program, the environment would have to be passed through as a second parameter to the `interpret` function. The reader monad allows for increasing the modularity of interpreters by hiding this parameter (see: Liang et al. [6]).

In AspectJ, we can define a default implementation for a static method `Environment.read()` that returns an empty environment. This implementation is shadowed by an advice that is triggered, if the method is called within the context of an environment extension. The advice then returns the extended environment. The default implementation together with the advice plays the role of the `ask` function in the reader monad.

The interpreter for a `Var` term is implemented as follows (`Environment.lookup()` is a method that returns the value of a variable this is stored in the environment):[4]

```
public Value interpret() {
   return Environment.read().lookup(id);
}
```

The interpretation of a `Lambda` term is trivial: it creates a value of class `Function`, simply passing along its parameter name and its body:

```
public Value interpret() {
   return new Function(variable, body);
}
```

The `Function` class implements an `apply(Value)` method that will be called during the interpretation of an `App` term:

```
public Value apply(Value val) {
   return new InEnv(body,
                    Environment.read()
                        .extend(variable, val))
                .interpret();
}
```

The call of `local` in the Haskell code is replaced by the creation of a new term of a class `InEnv`, that is only meant to be used internally. The environment to be used for the function application is created by extending the surrounding environment via an `extend()` method on the `Environment` class. In order to get the environment that is to be extended, an `Environment.read()` message is sent here as well.

The `InEnv` class stores the function body as well as the environment in which the body shall be executed. Its `interpret()` method just calls the `interpret()` method of its body. Its sole aim is to store the environment such that it can be accessed by a pointcut. Whenever `Environment.read()` is called, the aspect code can access the environment in the following advice:

```
Environment around(InEnv inEnv):
   execution(* Environment.read())
   && cflow(execution(* InEnv.interpret())
           && target(inEnv)) {

   return inEnv.getEnvironment();
}
```

In this way, the `cflow` and `target` pointcuts can achieve the same effect as the `Reader` monad. The `local` function cannot be perfectly imitated for two reasons: firstly,

[4]The full code is available at `http://www.st.informatik.tu-darmstadt.de:8080/~ostermann/foal07/`

its first argument is an environment transformer, while due to the nature of the `cflow` pointcut, the complete environment must be available in the innermost join-point matching the `cflow` pointcut. And secondly, its second argument can be any function, while the `cflow` pointcut has to define the join-point, at which the environment is extended, by name. In the solution given above, these issues could be solved by making use of a new class `InEnv` that stores the environment, and by its method `InEnv.interpret()` that serves as a marker for the `cflow` pointcut.

3.2 Separating kinds of computations

In the following, it shall shortly be discussed whether the lack of separation of concerns when using monads has to be regarded as a shortcoming in relation to AOP. The question is, whether it is useful to separate the operations that access and manipulate the extended parameter structure from the normal execution of the sequence of operations.

In the example of the interpreter given above, the separation of the environment retrieval from the interpreter concern would be artificial: it is relevant for the understanding of how the interpretation of e.g. the `Var` term takes place to know that the environment is accessed at that point.

In contrast, there are situations in which the separation of concerns might seem appropriate: one could think of some logging function that is enabled or disabled by setting a dynamic boolean variable. It could be useful to put the logging code into a separate module.[5]

Something similar holds for the error monad. While Lippert/Lopez [7] give examples of the usefulness of separating out exception detection and handling into separate modules, this is not generally so. The basic reason for throwing an exception is that a computation cannot continue with a reasonable result. This breakdown in the execution of the current concern is normally a relevant part of this concern and it therefore is not appropriate for singling it out into a separate module. Looking at these two examples, it has to be expected, that there is no general answer to this question.

3.3 Conclusion

There is no general way to introduce computational powers in a controlled fashion in AOP. State and exceptions are part of the native mechansims of most languages. Their use cannot be restricted. Nevertheless, a certain extension of computational powers can be achieved by the mechanisms of dynamic quantification.

The limitation of referential transparency that has been discussed as a problem of the use of monads is omnipresent in those languages, anyway. On the other hand, the absent power of monads to separate out concerns, in the way that AOP does it, can in some cases be regarded as a limitation of their expressiveness.

4. CONCLUSION

To sum it up, monads and aspects have to be regarded as quite different mechanisms, not able to express each other. On the one hand, monads are not capable of oblivious quantification. The only kind of quantification they allow is dynamic quantification in a non-declarative way. It would therefore stretch the meaning of AOP to still consider monads as an aspect-oriented mechanism. On the other hand, aspects are not very useful in abstracting over computational capabilities.

[5]Of course, logging in Haskell requires the combination with some writer or similar monad.

5. REFERENCES

[1] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[2] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, number Lecture Notes in Computer Science 925, 1995.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[4] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

[5] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*, pages 195–213, 2005.

[6] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[7] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[8] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP2003*, July 2003.

[9] W. D. Meuter. Monads as a theoretical foundation for aop. position paper in ecoop '97 workshop on aspect-oriented programming.

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

[11] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report.* Cambridge University Press, 2003.

[12] M. Wand. Understanding aspects: extended abstract. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 299–300, New York, NY, USA, 2003. ACM Press.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.