Monads vs. aspects analysis for computation encapsulation

Alexandru Stoica
Babeș-Bolyai University
Computer Science Department 248

February 15, 2019

Abstract

Both monads and aspects had become a popular encapsulation method of effects into a legacy system. Programmers are increasingly using monads in their functional environments, and aspects in object-oriented settings to reduce their code's complexity and add features to their products. In this paper, we create a comparative analysis between monads and aspects, presenting their advantages and disadvantages in different contexts.

keywords aop, functional programming, monad, aspect, computation encapsulation

Subject Classification Theory of computation

1 Introduction

A monad represents a mathematical design pattern which encapsulates a given value and an additional computation or effect [MTY05]. Every monadic structure respects the following three laws

$$unit :: x \to Mx$$
$$value :: Mx \to x$$
$$bind :: Mx \to (x \to My) \to My$$

which helps us create a monad Mx from a given value x, transform our monad Mx to another monad of type My and return the value encapsulated in our structure. [MTY05]

In functional programming, monads are used to encapsulate effects, computations which at some point in time t return a value. Due to their elegant properties, monads are a good way to abstract actions which may produce side-effects such as IO operations.

In different circumstances, a new programming paradigm named aspect-oriented programming evolved around the idea of encapsulating cross-cutting concerns (computa-

tions which affect multiple modules) into aspects.

Thus, our community had split into two groups, to try to popularise each method and promote it in each other's programming paradigm.

In this paper, we do a comparative analysis over those two approaches, providing a simplified context in which we are using them to encapsulate behaviours:

- We describe the generic environment within monads and advices may help us to extend our code (section 2), and we simplify our generic model to provide a runnable implementation of our solutions.

- We present a monadic data structure which solves our initial problem, and we describe a couple of advantages and disadvantages when using a monad as a way to compose behaviours (section 3).

- We provide an aspect, containing a pointcut and an advice which resolves our challenging situation (section 4), and we evaluate the solution in contrast to a monadic structure.

- We discuss related work in section 5 and provide our evaluation's conclusions in section 6.

2 The Challenge

Integrating additional computations into an existing system can shift into a difficult assignment because of the following conventional restrictions:

- we are not permitted to modify a function's implementation

- we are not authorised to alter a function's parameters' types

- we are not allowed to transform the means our system composes subcomponents

Acknowledging all those limitations, one may question why such practices are necessary when mixing new services. In our examples, we limit ourselves by respecting those rules to separate our new behaviours from our legacy code. Extending our system matures into an exercise of writing new code, rather than changing existant instructions.

Furthermore, we evaluate the general case using a mathematical model which represents our legacy context:

$$\begin{cases} f_1 :: A_1 \to A_2 \to \cdots \to A_n \to X_1 \\ f_2 :: B_1 \to B_2 \to \cdots \to B_n \to X_2 \\ \vdots \\ f_n :: Z_1 \to Z_2 \to \cdots \to Z_n \to X_n \\ h :: X_1 \to X_2 \to \cdots \to X_n \to Y \end{cases} \tag{1}$$

Our model composes n functions $f_i, i = \overline{1, n}$ using h, each function represents an independent computation; to simplify our model, we can consider those a set of serial processes.

Our model is composed of n various functions $f_i, i = \overline{1, n}$ and one transformation h which combines all f_i, each mapping represents an independent computation; to simplify, we acknowledge this set of processes to be serial and synchronised.

The granted practices permit us to adjust the manner in which we apply our function $(h(f_1 a_1 \dots a_n) \dots (f_n z_1 \dots z_n))$, without altering the conclusive output. The circumstances enable us to consider two options when blending a new operation:

- using dynamic detection and automatic evaluation; with an aspect which encapsulates a new process, and pointcuts which detects the target action's location in our implementation scope
- using static detection and automatic evaluation; with a monad to encapsulate our computation, and its bind function to evaluate it

Those two proposals tackle unique situations; by using aspects, we enlarge our implementation space from a two-dimensional setting, defined by our operation's position (where we formulate our instructions), and its time (when performing our behaviour) to a multidimensional environment, in which each distinct computation defines a dissimilar dimension of implementation.

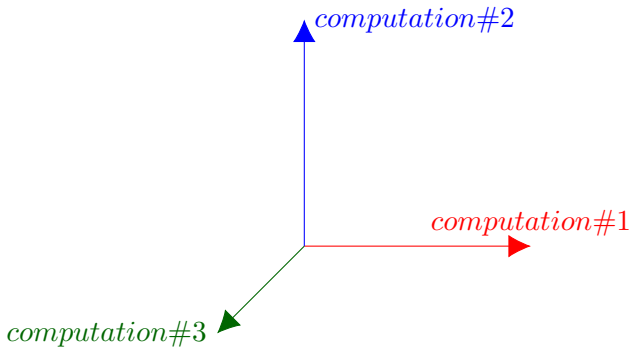

Figure 1: Aspect-Oriented Implementation Space

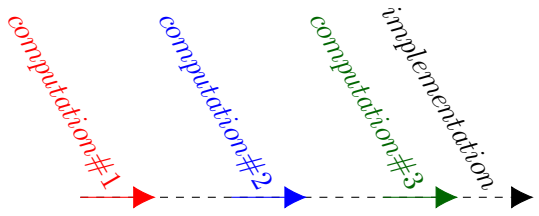

Figure 2: Aspect-Oriented Implementation Space in Object-Oriented Environments

In different circumstances, monads elevate our computation into a different scope, which distinguishable to an extension is not self-supporting from the previously mentioned framework. A monad represents a pattern which uses high-level functions, mathematical functors, and comes as a solution for effect encapsulation.

Let us examine a reduced variant of our overall intricacy, to explain each strategy using monads and aspects.

$$\begin{cases} f :: Int \to String \to Boolean \\ g :: String \to Float \to Double \\ h :: Boolean \to Double \to Int \end{cases} \tag{2}$$

Although all the thoughts exhibited in our paper are self-supporting from a programming language or paradigm, we use Kotlin to produce a *runnable* implementation of our clarifications for several reasons:

- Kotlin is a multi-paradigm language with supports functional and object-oriented features
- Kotlin holds a static and powerful type system
- Kotlin empowers us with extensions on generic types for supplementary operations, which is useful to clarify our monad's implementation

3 Monad's Approach

Our monadic data structure operates as a wrapper which carries our extra behaviour and shrouds our original transformation's application $h(f(4, "test"), g("test", 4.0))$; the aforementioned technique holds beneficial features because it lets us to chain further operations, mixes them and later determines which calculation we evaluate at runtime.

This approach provides us with adaptability at the expense of loquacity; we have

```
class Log<A> private constructor(
    private val instance: A) {

    companion object {
        fun <A> just(instance: A):
            Log<A> = Log(instance)
    }

    fun <B> bind(f: (A) -> Log<B>):
        Log<B> = instance
            .also { println(it) }
            .let(f)

     fun value(): A = instance
}

Log.just(f(4, "test")).bind { lhs ->
Log.just(g("test", 4.0f).bind { rhs ->
    h(lhs, rhs) }) }
```

Figure 3: Monad's generic implementation in Kotlin

to improve the method of applying our function h with a less convenient lambda nesting arrangement. Some functional languages permit us to manage this behaviour with syntactic sugar such as a do notation.

```
main = do
    resultF <- f(4, "test")
    resultG <- g("test", 4.0)
    return h(resultF, resultG)
```

Figure 4: Alternative calling method in with do notation in Haskell

We can also decorate our initial operation with another new behaviour by using a chain of bind function calls, or we can apply our initial computation multiple times.

```
f(4, "test").bind { it }.bind { lhs ->
g("test", 4.0f).bind { it }.bind { rhs ->
    h(lhs, rhs)}}
```

Figure 5: Composing computations with bind function

Composing monads of different types with a map function will allow us to convert between the bind calls the type of additional computation we are expecting.

$$map :: a \to b \to (MNa \to MNb) \tag{3}$$

$$map :: map_M \cdot map_N \tag{4}$$

Flexibility is the primary advantage of using monads over aspects; we can easily extend our monads to generate new computation or add additional computations on top of our bind function.

4 Aspect's Approach

Aspects provide an alternative solution for our problem, in which we can control when and where the execution of our additional computation takes place.

Depending on the programming language, aspects may provide us with a rich and

```
aspect Logging {
    pointcut exec() :
        execution(* f(* *) && * g(* *));
    after : exec() {
        print(@JointPoint.result)
    }
}
```

Figure 6: Aspect-oriented platform independent solution

powerful detection mechanism. In languages such as Java, C#, Kotlin or Groovy, we can detect annotations and compose aspects using them on our functions, classes or properties.

Other programming languages, especially functional ones do not provide us with this type of reflection detection; we are limited to detect the location where our additional computation will execute using the function's properties such as its name, parameters types or return value type.

Such imposed limitations make our aspects challenging to combine and extend; we have to modify our existing code, the aspect's pointcut to change the location of the execution of our new behaviour.

Furthermore, some programming languages do not support aspects as an extension framework or standard library, so their usage is limited, unlike monad's universal usages.

The dynamic control of our aspect's execution, defined by its pointcut provides a powerful mechanism which allows us to add additional computations without changing our existent code (not even the application of our h

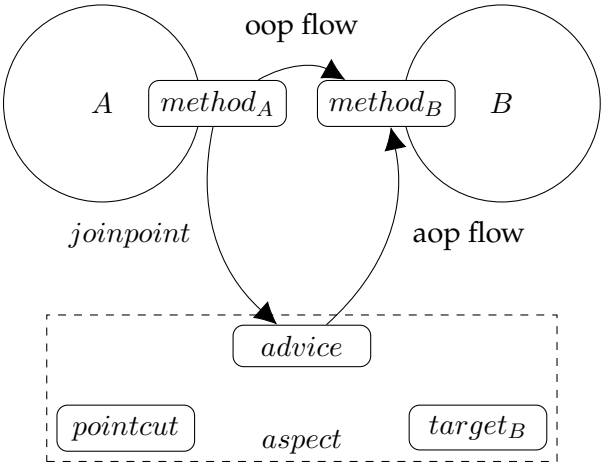

Figure 7: Aspect-Oriented As Proxy

function).

Unlike monads, aspects do not patch our code with additional wrappers (at least not explicitly), our compiler or interpreter takes care of the job of their coupling with our existent system.

5 Related Work

Over the past two decades, a couple of impressive papers highlighted the connection between our two approaches. Prof. Wolfgang De Meuter's paper "Monads as a theoretical foundation for AOP" [De 01] presents a set of similarities between monads and aspects, and how we can use monads in functional programming to simulate or replace the need for aspects from the aspect-oriented paradigm.

1. Aspect-oriented extensions for functional programming languages

An intriguing branch of the functional community tries to combine the two approaches presented in this paper into a single environment, by implementing an aspect-oriented extension for functional languages.

A refreshing example of such a language is Aspectual Caml based on Objective Caml [MTY05]. An aspect is viewed as an extension of a given data structure and a modification of the evaluation's behaviour.

In a functional programming language, currying functions define the structure of a given program; Aspectual Caml offers unique features such as curried pointcuts, which detect those types of functions in our code base.

2. Anticipation of effects

A challenging requirement in aspect-oriented code represents the anticipation of effects. In large systems controlled and manipulated by multiple aspects, it gets difficult to manually evaluate the effects produced by our functions and data structures [WO09].

Usually, in functional programming languages, all effects are encapsulated into monadic structures (or applicative functors or comonads) [WO09], adding advices into a functional program will generate a new approach in encapsulating

side-effects. This addition may generate a difficulty when evaluating the time complexity of a given algorithm, and a challenging situation in lazily evaluated computations.

The predictability factor of our programs will suffer if we use both methods to encapsulate computations [WO09].

6 Conclusions

When dealing with cross-cutting concerns which wave their effects in multiple modules, we have two methods to tackle the encapsulation of our decorative behaviour, (1) using monads, mathematical data structures from functional programming or (2) aspects from aspect-oriented programming. Both methods allow us to decorate existing systems for functions with new operations such as logging or security, although depending on our system's complexity, one method may challenge the other.

Aspects may unpredictably influence an existent system, due to their target detection power provided by their pointcuts. Although monads are more predictable and more comfortable to compose, the difficulty of their application varies based on a couple of critical features of a given programming language such as do notations.

One may use aspects to detect monads in functional programming languages since all program's effects are encapsulated inside them, or implement aspect-oriented features such as aspect waving using monadic structures.

References

[De 01] Wolfgang De Meuter. "Monads as a theoretical foundation for AOP". In: (Jan. 2001).

[MTY05] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. "Aspectual Caml: An aspect-oriented functional language". In: vol. 40. Jan. 2005, pp. 320–330. DOI: 10.1145/1086365.1086405.

[WO09] Meng Wang and Bruno C. d. S. Oliveira. "What Does Aspect-oriented Programming Mean for Functional Programmers?" In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*. WGP '09. Edinburgh, Scotland: ACM, 2009, pp. 37–48. ISBN: 978-1-60558-510-9. DOI: 10.1145/1596614.1596621. URL: http://doi.acm.org/10.1145/1596614.1596621.