Monads as a theoretical foundation for AOP
Paper Review

Alexandru Stoica

Babeș-Bolyai University
Computer Science Department 248

Jurnal: Monads as a theoretical foundation for AOP

Author: Wolfgang De Meuter **Year**: 1997 **Pages**: 25 (6) **Publisher**: Springer-Verlag

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.8262

February 15, 2019

1 Title

The title of Wolfgang's paper "Monads as a theoretical foundation for AOP" is descriptive because it suggests precisely the primary idea behind the work; it describes directly what the content of the paper is about, without leaving space for interpretation or ambiguity.

2 Abstract

In contrast to the mainstream format of a research article, Meuter does not provide an abstract directly on his paper. The publication/conference provides a short indicative abstract for his paper, which presents the scope of the given problem, and the conclusions drawn from his research.

3 Introduction

Definition 3.1 *In functional programming, a monad is a design pattern that allows programmers to encapsulate computations/behaviours and treat them as wrappers for any other computational types within our monad. [Wik18b]*

In general, a monad has three significant operations which define his structure: [Wik18b]

- a constructor which depends on a value of type `X` and creates a monad of type `M<X>` `constructor(value: X) -> M<X>`
- a bind operator (*) which takes a function from type `X` to type `M<Y>`, and combines monadic operations `bind(monad: M<X>, function: X -> M<Y>) -> M<Y>`
- a value gateway which returns the value stored inside our monad (after we apply a specific

monad computation on our value, otherwise the concept would be useless) `value(monad: M<X>) -> X`

Also, a monad may have a specific operation which encapsulates his particular computation. [Wik18b]

Definition 3.2 *In aspect-oriented programming (AOP), an aspect represents a cross-cutting concern, a computation which affects multiple modules such as logging, security and others. [Wik18a]*

Enabling functional programming languages to take advantage of the aspect-oriented features such as cross-cutting concern encapsulation has become a primary research topic in the functional community in the past few years [WO09]. AOP is an extension of the object-oriented programming paradigm; many of its features had a direct impact on the way we compose and encapsulate behaviours in complex object-oriented systems [Wik18a].

Introducing such a notion in functional programming is challenging, since the structure of an aspect is similar to the structure of a regular class from object-oriented programming, and many aspects create side effects throughout our program [WO09]. Side effects usually originate from mutations of the global state of our program, an idea poorly viewed by the functional research community; yet a needed concept when we have to create useful programs.

Meuter's paper was one of the firsts to tackle this problem, introducing features from aspect-oriented programming into functional languages. His article was published in 1997 at the "International Workshop on Aspect-Oriented Programming at ECOOP" and influenced over 26 critical contributions in the past 20 years.

The author proposes a theoretical aspect-oriented foundation based on the monads from functional programming. In this model, a monad transformation represents an aspect in a monadic style program which acts as an aspect weaver for cross-cutting concerns. The `bind` operation on monads combines other monadic operations and represents the join points of our program.

4 Body

Meuter's paper describes a case study in which he combines the two programming paradigms using a simple Fibonacci computation as an experiment to test his observations. The author begins his article with an examination of the properties of a monad as and central entity for his case study, and elaborates on the concept's importance using experimental methodologies.

Wolfgang acts as an investigator in an exploratory mission to test the hypothesis of using monads as a method of encapsulation for cross-cutting concerns. The evaluation takes place in a natural functional environment, Meuter uses natural notations such a lambda calculus and Scheme to describe his examples and the monadic structures of his program.

The collected data presented in this paper originates from multiple sources such as PhD thesis, reports, conference records and personal observations.

The paper begins with a short introduction (section 1) into aspect-oriented programming describing its importance and benefits. Meuter defines the model of a standard monad (section 2) and explains how we can use monads in a complex environment (section 3), to compose monadic style programs.

The primary section of Meuter's work (section 4) describes a few resemblances between monadic programming and aspect-oriented programming, followed by a couple of valuable code examples. As expected the last section presents the results, conclusions and future research of his paper (section 5).

5 Results

The primary results of Wolfgang's article indicate a connection between aspects from aspect-oriented programming and monads; both concepts prove to be effective methods for computation encapsulation, allowing programmers to glue together different components as layers, one on top of the other.

Monadic programs encapsulate the computation in a special function on the monad's particular type and allow us to combine different [Wik18b] [De 01] computations using monad's "bind" function or monad transformations. At the same time, in aspect-oriented programming, an aspect represents our computation, and a join point glues together multiple aspects.

Both monads and aspects have the property of waving the effects of their computations throughout our system, a crucial property for aspect-oriented programming. Wolfgang's paper presents this property using simple code examples, in which he transforms a simple Fibonacci computation into an optimised version of Fibonacci (calculated using dynamic programming with parallel execution).

The optimisation outputs the same result as the original Fibonacci function, yet since the optimised version represents a monad, the new version acts as a decorator/wrapper for the old one, which does not affect his client code, allowing him to extend his code with multiple computations/behaviours without changing the initial implementation.

Meuter's research paper does not provide an answer to the following essential questions:

- How can monads identify cross-cutting concerns in a given monadic program?
- Are monads able to support advanced aspect-oriented features such as pointcuts and advices?

It is important to point out the influence of Wolfgang's paper in the functional research community; over 26 research contributions cite his work and represents a base of a few aspect-oriented programming frameworks for functional programming languages such as F# [MC17], Haskell [Lin06] and others.

References

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

[Lin06] Chuan-Kai Lin. *Programming monads operationally with Unimo*. Sept. 2006. DOI: 10.1145/1159803.1159840.

[WO09] Meng Wang and Bruno Oliveira. "What does aspect-oriented programming mean for functional programmers?" In: (Aug. 2009), pp. 37–48. DOI: 10.1145/1596614.1596621.

[MC17] Keith Mannock and N Chacowry. *An aspect-oriented framework for F#*. Dec. 2017.

[Wik18a] Wikipedia contributors. *Aspect-oriented programming — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-December-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Aspect-oriented_programming&oldid=868173898.

[Wik18b] Wikipedia contributors. *Monad (functional programming) — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-December-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Monad_(functional_programming)&oldid=871435511.