A Practical Monadic Aspect Weaver

Ismael Figueroa * Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
{ifiguero,etanter}@dcc.uchile.cl

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

Abstract

We present Monascheme, an extensible aspect-oriented programming language based on monadic aspect weaving. Extensions to the aspect language are defined as monads, enabling easy, simple and modular prototyping. The language is implemented as an embedded language in Racket. We illustrate the approach with an execution level monad and a level-aware exception transformer. Semantic variations can be obtained through monad combinations. This work is also a first step towards a framework for controlling aspects with monads in the pointcut and advice model of AOP.

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

General Terms: Languages, Design

Keywords Aspect-oriented programming, monads, execution levels, weaving.

1. Introduction

In the pointcut-advice model of AOP, weaving is the fundamental mechanism by which aspects inject their crosscutting behavior in programs. Typically, prototype aspect languages reuse the facilities of an existing aspect language, like AspectJ [6] or AspectScheme [3], as a solid base on which to provide new extensions. However, in some cases it is necessary to maintain (potentially very similar) branches of the base language and the new features under focus of study. This situation is usually tedious, but can be managed with standard version control systems. However, this is but a symptom of a deeper problem: there is a lack of an abstraction mechanism for experimenting with aspect semantics without having to be concerned with the mechanics of a full blown aspect language.

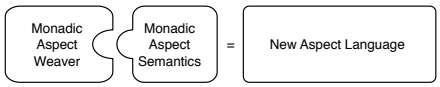

Figure 1. Combining a monadic weaver with a monadic aspect semantics to obtain a new aspect language.

We faced this maintenance problem during the development of an exception handling mechanism for execution levels. Tanter proposed *execution levels* for AOP [9] as a means to structure computation and avoid infinite regression by default, while providing flexibility to the programmer when required. In subsequent work, Figueroa and Tanter [4] address the *exception conflation* problem, that is, the problematic interactions between aspect and base code in presence of an exception handling mechanism. To solve this problem, they propose a *level-aware* exception handling mechanism in which exceptions are bound to the level at which they are thrown, and are only caught by handlers at those levels. During development of the level-aware exception mechanism, it was necessary to maintain two branches of LAScheme [9] (the original execution levels prototype language). The reason is that the new version introduced additional *noise* that was unused and unnecessary for the original version to work, so in order to clearly present both implementations we kept them separate. Unfortunately, both code bases diverged, requiring extra work to keep them in sync.

Independently, Tabareau recently proposed a monadic interpretation for execution levels [8], in which the exception semantics can be plugged by using a monad transformer. It was observed by Tabareau that in the standard execution levels calculus, the execution level is a property of the control flow and, as such, is propagated between expression evaluation in a sequential order. This matches the semantics of the State monad, in the case where the state is a natural number. He then defined the execution level monad, and two exception transformers to provide *flat* (standard exceptions that do not have access to the level of execution), and *level-aware* [4] exceptions to execution levels. More importantly,

* Funded by a CONICYT-Chile Doctoral Scholarship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FOAL'12, March 26, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1099-4/12/03. . . $10.00

Tabareau developed an *abstract weaving algorithm* that relies on monads to define concrete aspect semantics. Such a monadic aspect weaver allows for modular construction of languages (Figure 1). Using these concepts, he describes a monadic weaving algortihm on top of MinAML, a statically typed functional aspect language.

As a first practical instantiation of the formal development of Tabareau, we developed a monadic version of AspectScheme [3], called Monascheme. Monascheme is a higher-order functional aspect language with monadic aspect weaving, which supports dynamic aspect deployment with lexically, dynamically, and globally scoped aspects. Monascheme is implemented as an embedded language in Racket. At the heart of Monascheme lies the monadic aspect weaver, which provides management of the join point stack and is parametric with respect to the monadic aspect semantics of a given language. Our implementation is available at http://pleiad.cl/research/monascheme.

The contributions of our work are: first, we validate the work of Tabareau by providing a concrete implementation of a monadic aspect weaver for the Racket language, called Monascheme (Section 2). Monadic aspect semantics are defined as Racket modules, encapsulating the corresponding monad definition as well as supplementary procedures and syntactic extensions as needed. Second, we improve on MinAML by providing a full-fledged aspect language. Third, we exemplify the usefulness of our system by implementing the execution level monad, and combining it with a variant of the exception monad transformer to obtain level-aware exceptions (Section 3). We describe aspect deployment and weaving in Section 4. Finally, we see this development as the first step for a modular framework for controlling aspects with monads (Section 5). Section 6 discusses related work.

2. Monascheme in Action

We start by describing shortly how to write aspect-oriented programs in Monascheme, assuming the semantics given by the execution level (`EL`) monad. Monascheme is implemented in Typed Racket [10], a typed version of Racket designed to ease the transition between untyped and typed programs. Consider the program shown in Listing 1. The `(: ... )` notation is for type annotations in Typed Racket. The `run` form (line 6) evaluates a given program providing the initial level of execution (internally defined as 0). A tracing aspect is dynamically deployed using `deploy-fluid` (line 6). It matches all calls to `write` and executes `trace` around those calls. Then, when `(write 'hello)` (line 7) is called, the trace advice executes writing `'tracing` to the output, and then proceeds (line 3-4). The end result is that the symbol `'tracinghello` is written to the standard output. Observe that the advice does not loop when calling `write`, due to the semantics of execution levels (which we explain in Section 3.3).

```
1 (: trace ((Any -> (EL Any)) -> (Any -> (EL Any))))
2 (define (trace proceed)
3  (lambda: ([a : Any]) (do (write 'tracing)
4  (proceed a))))
5
6 (run (deploy-fluid (call write) (return trace)
7  (write 'hello)))
```

Listing 1. A simple aspect-oriented monadic program in Monascheme (with the execution level monad).

Besides type annotations, the only difference between the program of Listing 1 and its LAScheme equivalent is the use of `return` and `do` in Monascheme[1]. This is because both base and aspect code are written in monadic style, and all functions must return a computation inside the corresponding monad. In this example, all functions must `return` an `EL` computation. Computations are also needed when deploying aspects, as reflected by the use of `(return trace)` at line 6. Additionally, we borrow the `do` notation from Haskell as a shorthand for using the monadic `bind` function. It chains operations sequentially and allows the programmer to extract the value of a computation. Also, additional bindings can be provided by a specific semantics. In our example, the `run` form is specific to semantics of the `EL` monad. Finally, the forms `deploy-fluid`, `deploy-static` and `deploy-top` allow the deployment of aspects with dynamic, static, and global scope respectively. All these forms are implemented using the Racket macro system and are valid in all aspect semantics.

3. Instantiating Monascheme

This section describes how to define monadic aspect semantics as modules that plug into Monascheme. First, we describe how to plug the modular semantics into Monascheme. Then, we present the interface that a module has to adhere to. Finally, we illustrate how to implement such an interface for both the execution levels and level-aware exceptions semantics, as described in [8].

3.1 Pluggable Aspect Semantics

In our setting, the aspect semantics of a language is specified by a monad, with the usual `return` and `bind` functions, plus a function responsible to create monadic aspects.

The core Monascheme module is responsible for configuring the monadic aspect semantics. It must require (*i.e.* import) an aspect semantics module in order to typecheck and compile correctly. When a client program requires the core module, the monadic aspect semantics are stored in two Racket parameters (*i.e.* thread- and continuation-safe dynamic bindings). The current configuration can be queried using functions like `get-current-monad`. For instance, the `do` form we used above queries the current monad in order to

[1] We provide a `fmap` form to reuse Racket primitives in the monadic setting. It is possible to define a `require/lift` form to automatically require and lift the bindings of a Racket module. However, this feature is currently not implemented so specific bindings must be manually lifted using fmap.

```
1 (struct: (A) ValueLevel([value : A] [level : Integer]))
2 (struct: (A) EL ([value : (Integer -> (ValueLevel A))]))
3
4 (: EL-return (All (A) (A -> (EL A))))
5 (define (EL-return a)
6   (EL (lambda: ([s0 : Integer])
7       (ValueLevel a s0)))))
8
9 (: EL-bind (All (A B)
10  ((EL A) (A -> (EL B)) -> (EL B))))
11 (define (EL-bind ma f)
12   (EL (lambda: ([s0 : Integer])
13       (let ([result ((EL-value ma) s0)])
14        ((EL-value (f (ValueLevel-value result)))
15         (ValueLevel-level result)))))))
```

Listing 2. Definition of the execution level monad.

obtain the associated binder. Also, the weaver (Section 4.3) uses the `get-current-makeasp` function to construct aspects according to the current configuration. A monadic aspect semantics module must implement a specific interface.

3.2 Aspect Semantics Module

A module that defines monadic aspect semantics must at least export the following bindings:

- A type constructor `M`: the type constructor for the monad. For all types `A`, `(M A)` defines a new type.
- `M-return` with type $\forall$ `A` : `A` $\rightarrow$ `(M A)`.
- `M-bind` with type $\forall$ `A`, `B` : `(M A)` `(A` $\rightarrow$ `(M B))` $\rightarrow$ `(M B)`.
- `M-do`: form that behaves like `do`, but uses the `M-bind` binder instead of the binder of the current monad. We provide an internal macro to ease the definition of `M-do`.
- `MakeAspect`: the type of `M-make-aspect`. It is an alias of `(Pc Adv` $\rightarrow$ `(Pairof Pc Adv))`, where `Pc` is `JPStack` $\rightarrow$ `(M Any)` and `Adv` is `(Any` $\rightarrow$ `(M Any))` $\rightarrow$ `(Any` $\rightarrow$ `(M Any))` (see Section 4.1 for details on typing).
- `M-make-aspect`: function that defines the creation of pointcuts and advices in the semantics.
- `MakeAspectResult`: return type of `M-make-aspect`

The interface is designed to overcome some limitations of the Typed Racket type system. `M-return` and `M-bind` define the fundamental monad operations and are used to define the `return` and `do` forms, respectively. The `M-make-aspect` is used for aspect creation and deployment.

As any Racket module, a monadic aspect semantics module can provide supplementary procedures and syntactic extensions as needed, *e.g.* `up` and `run` in the `EL` monad.

3.3 Execution Level Monad

We show the encoding of the execution level monad [8] as a monadic aspect semantics for Monascheme[2].

Monadic definition. The execution level monad is defined (Listing 2) by its `EL-return` (line 4-7) and `EL-bind` functions (line 9-15), which rely on the `EL` type constructor (line 2). `EL` uses a support `ValueLevel` struct (line 1) that holds values, of any type A, and the execution level as an integer. A computation `(EL A)` is defined as a function with type `Integer` $\rightarrow$ `(ValueLevel A)`. `EL-return` lifts a value `a` into the monad by wrapping it into a closure with the appropriate type. The `EL-bind` binder first extracts the value of its computation argument (line 13) to obtain a (potentially) modified level of execution, and then passes the new level as the level at which the application of `f` is evaluated (line 14-15).

[2] A Racket module can rename bindings when exporting. This way we can use different internal names than those of Section 3.2

```
1 (: lookup (-> (EL Integer)))
2 (define (lookup) (EL (lambda: ([s : Integer])
3              ((EL-value (EL-return s)) s))))
4
5 (: inc-level (-> (EL Void)))
6 (define (inc-level) (EL (lambda: ([s : Integer])
7                (ValueLevel (void) (+ s 1)))))
8
9 (: dec-level (-> (EL Void)))
10 (define (dec-level) (EL (lambda: ([s : Integer])
11                (ValueLevel (void) (- s 1)))))
12
13 (define-syntax-rule (up e)
14  (do (inc-level)
15    (x <- e)
16    (dec-level)
17    (EL-return x)))
```

Listing 3. Level management functions.

Monadic operators. Some programs may be interested in querying or modifying the level of execution. To this end, the monadic aspect semantics module exports some functions and syntactic extensions. For instance, the level-shifting operator `up` is used to move the evaluation of an expression to the level above the current level. Its definition is shown in Listing 3 (lines 13-17). It uses the lower-level (not user-visible) functions `inc-level` and `dec-level`. `down` is defined similarly. Finally, the `lookup` function reifies the current level as a value (lines 1-3).

Aspect semantics. In execution levels, an aspect is deployed at a level n and can only advise join points emitted at that level. Join points are emitted one level above the computation they originated. Also, an aspect at level n evaluates its advice at level $n + 1$. This semantics is specified by the `EL-make-aspect` function (Listing 4). Given a pointcut `pc` and an advice `adv`, the function creates both a level-aware pointcut and advice. In both cases this is done by capturing `n` (line 3), the level of execution at definition time, using lexical scoping. The new pointcut first performs a level check (line 8), comparing the level of definition with the level of execution, and only if those levels match `pc` is applied one level above (line 9). The new advice returns a closure which executes `adv` one level above current computation, using `up`, wrapping `proceed` in a *level-capturing function* (line 13) to ensure that base computation is always executed at its original level. A level-capturing function is bound to a level of execution l. When applied, it is evaluated at level l and then the level goes back to its previous value. `lambda-at` is an internal expression to construct a level-capturing function bound at level n.

```
1 (: EL-make-aspect MakeAspect)
2 (define (EL-make-aspect pc adv)
3  (do (n <- (lookup) : Integer)
4   (EL-return
5    (cons
6     (lambda: ([jp : JPStack]) ;; New pointcut
7      (do (l <- (lookup) : Integer)
8       (if (= l n)
9        (up (pc jp))
10       (EL-return #f))))
11    (lambda: ([proceed : (Any -> (EL Any))]) ;; New advice
12     (lambda: ([arg : Any])
13      (up (adv (lambda-at n (args) (proceed args))) arg)))))))
```

Listing 4. Aspect creation in the execution level monad

```
1 (define-type Exception (Pairof 'Exception Integer))
2 (define-type LA (All (A) (M (U A Exception))))
3
4 (: LA-return (All (A) (A -> (LA A))))
5 (define (LA-return a) (M-return a))
6
7 (: LA-bind (All (A B)
8   ((LA A) (A -> (LA B)) -> (LA B))))
9 (define (LA-bind ma f)
10  (M-bind ma
11   (lambda: ([a : (U A Exception)])
12    (if (Exception? a)
13     (M-return (cons 'Exception (cdr a)))
14     (f a)))))
```

Listing 5. Monadic definition of the level-aware exception transformer

3.4 Level-Aware Exceptions Monad Transformer

We define a special-purpose exception monad transformer to provide level-aware exception semantics to our language (as in [4]). By accessing the level of execution stored in the monadic context, the transformer can perform the necessary checks to only catch exceptions whose level matches those of the handlers. Note that this transformer requires a monad `M` that implements the `lookup` operation.

As a monadic aspect semantics module, it must adhere to the interface of Section 3.2. The transformer takes a monadic aspect semantics to be transformed, and exports the bindings required by Monascheme. Listing 5 describes the monadic definition of the transformer. This part is similar to the traditional Exception Monad Transformer. An `Exception` type (line 1) is defined as a pair of a constant symbol `'Exception` (we use a simple version where exceptions contain no extra information) and an integer representing the level at which the exception was raised. We define the type `LA A` (line 2) as an alias for a computation of the original monad with the union type `(U A Exception)`, denoting a value of type either `A` or `Exception`. To lift values into the new setting, `LA-return` (line 4-5) uses the original `M-return` (line 5) to create a `LA` computation. `LA-bind` (line 7-14) behaves as the usual exception monad transformer binder.

Having established the context in which exceptions propagate adequately, we define (Listing 6) the `throw` and `try-with` forms, to raise and catch exceptions, respectively. To raise exceptions, we get the current level `n` (line 3) using the `lookup` operation provided by `M`, and return an exception

```
1 (: throw (All (A) (A -> (M Exception))))
2 (define (throw a)
3  (M-do (n <- (lookup) : Integer)
4      (M-return (cons 'Exception n))))
5
6 (: try-with ((LA Any) (LA Any) -> (LA Any)))
7 (define (try-with e1 e2 )
8  (M-do (result <- e1 : (U Any Exception))
9      (if (Exception? result)
10       (M-do (n <- (lookup) : Integer)
11          (if (= n (cdr result))
12            e2
13            (M-return (cons 'Exception (cdr result)))))
14       (M-return result))))
```

Listing 6. `throw` and `try-with` forms tag and check the execution level respectively

```
1 (define-syntax-rule (deploy-fluid pc adv body ...)
2   (do (v-pc <- pc : (JPStack -> (M Any)))
3     (v-adv <- adv : ((Any -> (M Any)) -> (Any -> (M Any))))
4     (asp-pair <- ((get-current-makeasp)
5             v-pc v-adv) : MakeAspectResult)
6     (aspect <- (return
7            (Aspect (car asp-pair)
8                (cdr asp-pair))) : Aspect)
9     (handle <- (add-dynamic! aspect) : Undeployer)
10    (result <- (do body ...) : Any)
11    (remove-dynamic! handle)
12    (return result)))
```

Listing 7. `deploy-fluid` implementation

at that level using `M-return` (line 4). The `try-with` takes two computations `e1` and `e2`. It extracts the value of `e1` into `result` (line 8) using the binder of the monad `M`. Then, if `result` is an exception (line 9) it checks if the level of the handler matches that of the exception (line 11-13). If they match, `e2` is returned (line 12). Otherwise, the exception is propagated (line 13). If `result` is not an exception, it is returned (line 14).

4. Aspect Deployment and Weaving

In this section, we describe in detail the implementation of aspect deployment and weaving in Monascheme. The management of the join point stack is as in AspectScheme [3].

4.1 Typing in Monascheme

The monadic aspect weaver enables Monascheme to be parametric with respect to the aspect semantics. For now, the type system only checks monadic types. All values have type `Any`, and all non-advice functions have type `Any` $\rightarrow$ `(M Any)`. This is due to two limitations of Typed Racket: absence of type casts, and erasure of polymorphic types.

4.2 Aspect Deployment

Monascheme features scoped aspect deployment, for all aspect semantics. It relies on the function `M-make-aspect` to construct adequate pointcut and advice functions, and then stores them in an internal `Aspect` structure inside the corresponding aspect environment. The implementation of `deploy-fluid` is given in Listing 7. The other deployment constructs, `deploy-static` and `deploy-top`, are implemented in a similar way.

```
1 ;; (All (A B) ((M (A −> (M B))) JPStack (Listof Aspect) −> (M (A −> (M B)))))
2 (define (weave fun jp* aspects)
3  (foldr
4   (lambda (aspect proceed)
5    (do (proceed-fun <- proceed)
6      (pointcut-result <- ((aspect-pc aspect) jp*))
7      (if pointcut-result
8        (return (lambda (arg)
9          (do (advice <- ((aspect-adv aspect) proceed-fun))
10           (advice arg))))
11       (return proceed-fun))))
12  fun aspects))
```

Listing 8. Monadic Aspect Weaver

4.3 Monadic Weaving Process

The abstract weaving function (Listing 8) is implemented in an untyped module (its polymorphic type is given in comments). It expects a computation `fun`, a join point stack `jp*` and an aspect environment `aspects` containing all aspects that can apply and returns a new woven function. Aspects are processed sequentially, and in each step a `proceed-fun` function is generated (line 5). `proceed-fun` corresponds either to the next advice in the chain, or to the original `fun` function. Advice is inserted in the resulting function (line 7-11) only when the pointcut matches (lines 8-10). The `aspect-pc` (line 6) and `aspect-adv` (line 9) functions are accessors to the internal aspect structure used during aspect deployment. Note that the algorithm uses `do` and `return` to work in the monadic setting, but is independent of any specific monad.

5. Controlling Aspects with Monads

So far, our motivation has been the separation of concerns between a core aspect language and concrete aspect semantics specifications. However, since monads are typically used to control effects in programs, we want to mix monads to control the effects available to base and aspect code. To make the combination possible, it is necessary to provide a lifting between base and aspect monads (in both directions). For instance, we have developed the *agnostic execution level* (AEL) monad, in which no explicit level-shifting is allowed. We can then define a *mixed* AEL+EL monad in which level-shifting is forbidden in advices.

But there are many more combinations that remain to be explored and are part of ongoing work. For instance, it is possible to support *sandboxed aspects*. This includes scenarios like providing aspects with *e.g.* read-only capabilities to the underlying store, or using the State monad so that aspects have their private store. This can be achieved by using two variants of the state monad M_{base} and M_{asp}, with the same underlying type constructor but with different effects: when executing base computation in M_{base}, only effects (such as read or write) on S_{base} can be used, and when executing advice computation in M_{asp}, only effects on S_{asp} are accessible. In that case, lifting a computation from one monad to another is simply given by the identity. The key to specifying the allowed effects lies in the `M-make-aspect` function. By dynamically adjusting the monadic aspect semantics, the execution of advice can be wrapped in a completely different monadic setting.

6. Related work

A monadic aspect weaver enables modular variations of the language semantics, and also makes it possible to control the effects of aspects. Work on extensible compilers (*e.g.* abc [2]) relate to the first point. The advantages of using monads for defining language variations are well-known; in particular, monads are modular, composable, and they bring opportunities for reasoning about effects. For instance, if execution levels are implemented in abc, it is impossible to know if a method is "pure" with respect to the execution level, without manually customizing the type system.

EffectiveAdvice (EA) [7] is the most related proposal. EA is inspired by Open Modules [1], but with full support for reasoning about effects using monads. EA is based on open recursion, and on application of a fixpoint combinator as the weaving process. EA provides strong guarantees about non-interference and harmless advice. There are two main differences with our work: first, EA supports neither quantification nor obliviousness, instead it uses explicit advice points and explicit composition of aspects and programs. In contrast, our work supports the full-fledged pointcut-advice mechanism. On the other hand, the monadic encoding proposed by Tabareau modifies the actual weaving process, whereas the monadic translation of EA only applies to advices. Hence our approach supports language extensions.

The relation between aspects and monads has been discussed several times. Hofer and Ostermann [5] clarified that they are two different beasts. Our work does not attempt at unifying them, but rather at extending monads to AOP languages in order to build extensible aspect languages with reasoning support.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005*, pages 144–168.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Transactions on AOSD*, vol. 3880, pages 293–334, 2006.

[3] 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, 2006.

[4] I. Figueroa and É. Tanter. A semantics for execution levels with exceptions. In *FOAL 2011*.

[5] C. Hofer and K. Ostermann. On the relation of aspects and monads. In *FOAL 2007*.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353.

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

[8] N. Tabareau. A monadic interpretation of execution levels and exceptions for AOP. In *AOSD 2012*.

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

[10] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL 2008*, pages 395–406.