Monads as a theoretical foundation for AOP

Wolfgang De Meuter
(wdmeuter@vnet3.vub.ac.be)
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels - Belgium

1. Introduction

Looking at today's software engineering, the major abstraction technique is to divide systems into code units that are about a particular encapsulated ‘issue’ of the system, such that changes to a particular unit do not propagate through the entire system. However, this is only applicable as long as all the ‘issues’ of the system can indeed be classified in units. And that’s exactly where the shoe pinches. Some issues — called aspects — are system wide and cannot be put into a single unit. As a consequence, every piece of code that deals with these aspects is cluttered with them. Changing a property of an aspect requires one to rewrite all the code that deals with it. This problem is avoided if only we could program these aspects separately and afterwards weave them together with an ‘aspect weaver’. The activity of doing this is called aspect-oriented programming. In AOP, we separate the functional code of a system from all the additional aspects that surround it. But what exactly is an aspect and what power can one expect from the weaver ? The position of this paper is that much can be learned both about aspects and the aspect weaver if we think of the functional code as a monadic style program and we couch the different aspects into monads. The weaver then becomes a lifter to transform programs through different monads.

2. Monads

Monads come from category theory and were introduced in computer science by Moggi [6], but we had to wait for Wadler’s popularization [8] for monads to become widely used and for a ‘monad technology’ to arise. Originally, monads were introduced to solve the modularity problem of denotational semantics but nowadays they are heavily used to solve more technical issues such as the IO problem of functional programs. This problem is due to the fact that IO operators are actually side-effects (e.g. ‘write’ is a side effect on the screen) and thus complety demolish referential transparancy. For many years, this was a major obstacle to write functional applications with interactive IO. Functional programs were usually batch programs and IO was treated in a stepmotherly fashion precisely because the IO ‘aspect’ was all around and was considered a technical burden that completely ruined the beauty of a program. Until people learned how to encapsulate this system-wide aspect in a monad!
Another problem were in-place array operations. When a functional program uses arrays, conceptually these arrays have to be copied each time a position in the array is assigned. Indeed, assignment in arrays has to be modelled as a function and can therefore not have side effects. But of course, copying an array each time it is assigned yields hopelessly inefficient programs. Again, the problem was solved by putting the updatable array in a monad. Monadic programs that use the array can only access (and assign) it through so called monadic operations. This way, arrays are treated in a completely referential transparant way, but inside the monad, the operation can be safely programmed in a destructive way.

A monad is most effectively described as a computation that eventually returns a value. Monadic programming then consist of gluing together computations with an operator called 'bind' (denoted by `*`). Each monad defines such a `*` and `*` is allowed to see the

internal details of the monad. On top of this, there is an operation `unit` to inject any value into a zero computation that does nothing but returning the value:

```
c x = ...      /* c x denotes the type of computations
                  that will return a value of type x          */
* : c x -> ( x -> c y) -> c y
               /* glues together c x and c y parametrised
                  by the value of c x (i.e. x -> c y)
                  into c y                                    */
unit : x -> c x
```

Hence, `unit` injects any value into a zero computation. `*` glues together a computation c_1 of type `c x` and a computation c_2 of type `c y` that is parametrised by the value of c_1, into a resulting computation c_3. When we run c_3, the effect will be to run c_1, and then run c_2 thereby passing the value of c_1 to c_2. Although not part of the categorial definition of a monad, functional programmers provide a third operation called `run`. `run` actually runs the computation of type `c x` thereby returning its value (of type `x`). Of course all additional information about the computation will then be lost.

```
run : c x -> x
```

3. Monad Operations & Monadic Programming

Now that we now what a monad is, we have to discuss how monadic programs are written. Any monadic style program contains two kinds of code: code that does not depend on the details of the underlying monad and code that does depend on the details of the particular monad at hand.

• The easiest code is the one that is independent of a particular monad. This code manipulates values by applying operations onto values. It does not depend on the nature of the computations that produced those values. Here is the general strategy: suppose we write a monadic program that calls n monadic programs p_i, that each produce a value v_i. The effect of our monadic program is to apply a function `f` onto the values that will be produced by the computations. Of course, the result of applying `f` must be a computation again so that we have to `unit` it:

$$p_1 * (\lambda v_1.\ p_2 * (\lambda v_2.\ \ldots.\ p_n * (\lambda v_n\ .\ \mathtt{unit}(\mathtt{f}(v_1,\ \ldots,\ v_n))) \ldots)$$

This pattern occurs so frequently that modern functional languages like Gofer [2] provide a special syntax for it:

$$\mathtt{do}\ \{\ v_1 \leftarrow p_1;\ v_2 \leftarrow p_2;\ \ldots;\ v_n \leftarrow p_n\}\ \mathtt{result}\{\ \mathtt{f}(v_1,\ldots,v_n)\ \}$$

• The second kind of code fragments are those that depend on the nature of the underlying monad. Surely these fragments will exist for otherwise it would not be very useful to write monadic programs after all. In order to access the interal information of the monad, we need monad operations. These are extra operations (besides `unit` and `*`) that are added to a particular monad. Any code that uses such an operation admits that it can only be executed in a monad that strong. Let us for example consider a very simple tracing monad, in which every computation takes a stream (onto which tracing happens) and returns a value and a resulting stream

```
c x = Stream -> (x,Stream)       /* (a,b) means a x b */
unit x = λstr . (x,str)
cmp * f = λstr . let (x,str') = cmp str
                 in f x str'
```

The computations of the monad return values thereby transforming a stream into a possibly modified stream. In order to get something onto the stream that is passed around by the computations, we need a monad operation `write`:

```
write : String -> m void
write s = λstr . (void,append str s)
```

The `write` operation takes a string. It returns a computation that takes an output stream and adds the string to it. Of course, code that has nothing to do with tracing, does not have to know about `write`.

4. Monads and Aspect Oriented Programming

The resemblances between MP and AOP are really remarkable:
1) Both monadic style programs and aspect-oriented programs give us the same feeling about writing *'layered' code*. The top layer consist of the 'functional layer', that is the 'actual code of the program'. The other layers are the aspects (or different monads) that help the actual code to accomplish its task. Join points (AOP terminology) allow different aspects to meet each other. In a monadic program the join points are those functions that acces information of (a) monad(s).
2) Both monads and aspects have *system wide repercussions*. When 'expanding' a functional monadic program (i.e. throwing away all the `unit`'s and `*`'s) we get code that is completely tangled with 'global information'. In functional programs, this technical cluttering might consist of system wide state (i.e. assignable memory) or IO streams that enter and leave every function, exception handling and so on. This is precisely the phenomenon described by the AOP papers, in which the 'global information' is called systemic information.
3) Both MP and AOP endorse the viewpoint that the compiler's job is integration rather than inspiration. Just like AOP, MP forces a programmer to separate the different 'aspects' into different monads himself. When all monads are described and the interactions between them [4] is known, they can fairly easily be integrated. It is not a compiler's job to define or extract the monads.

Of course, the only way to test wether monads can really serve as a theoretical basis for AOP is to try to formulate a few AOP instances in monadic style. To this extent, we have implemented a small monadic OOPL in Scheme. We have used the macro facilities of Scheme to implement the following syntactic constructs:

```
(class parent . declarations)

(var name value)

(Omethod name fargs . body)      -- definition of ordinary method
(Obegin . lst)                   -- body of ordinary method
(Oself)                          -- self reference
(Osuper)                         -- parent reference
(O! name value)                  -- assignment
(O? name)                        -- variable referencing
(O: receiver name . aargs)       -- message passing

(Mmethod name fargs . body)      -- definition or a monadic method
(Mbegin . lst)
(Mself)
(Msuper)
(M! name value)
(M? name)
(M: receiver name . aargs)
```

A class consist of a parent class and a list of declarations. Each declaration is either an instance variable or a method. Methods can be ordinary or monadic. Within an ordinary method, all instructions must be ordinary. Within a monadic method, everything must be monadic. The reason why we distinguish between ordinary and monadic methods is that, in our work, we wanted to study the interactions between monadic code and non-monadic code. This is not really relevant for this paper. Just keep in mind that the monadic code denotes 'monadic objects', i.e. objects that have been injected in a monad using `unit`. The monadic constructions have the same semantics as the ordinary constructions, except that the values have become

computations and that, in the implementation of the language, these computations are bound together using `*`. When a monadic method calls a monadic method, the associated computations are bound together. When a monadic method calls an ordinary method, the ordinary method is executed and the result is returned after `unit'`ing it. When an ordinary method calls a monadic method the monadic method is '`run`'.

The following example is a number class built on top of Scheme numbers:

```
(define numberç
  (class rootç
    (var val 0)
    (var tmp '())
    (Omethod ! (i) (O! val i))
    (Omethod ? ()  (O? val))
    (Omethod + (i) (Obegin (O! tmp (numberç 'new))
                           (O: (O? tmp) ! (+ (O? val) (O: i ?)))
                           (O? tmp)))
    (Omethod - (i) (Obegin (O! tmp (numberç 'new))
                           (O: (O? tmp) ! (- (O? val) (O: i ?)))
                           (O? tmp)))
    (Omethod == (i) (Obegin (if (= (O? val) (O: i ?))
                           true
                           false)))
    (Mmethod fib ()
      (Mbegin
        (M: (M: (Mself) == (unit (num 0)))
          MifTrueIfFalse
           (Mblock0 (unit (num 1)))
           (Mblock0 (M: (M: (Mself) == (unit (num 1)))
                     MifTrueIfFalse
                      (Mblock (unit (num 1)))
                      (Mblock (M: (M: (M: (Mself) - (unit (num 1))) fib)
                                  +
                                   (M: (M: (Mself) - (unit (num 2))) fib)
     ))))))))
))

     (define (num number)
       (define i (numberç 'new))
       (O: i ! number)
       i)
```

The focus of our experiments was the `fib` monadic method. It is invoked by evaluating

```
(run (M: (unit (num 8)) fib))
```

By executing this method in the identity monad, we get a normal Fibonacci method[1]. The identity monad looks as follows. Its computations simply are values.

```
(define (unit x) x)
(define (bind x n f)
  (f x))
(define (run x) x)
```

The following experiment shows how `unit` and `bind` can be used to make the Fibonacci method more efficient. As everyone knows, `fib` is a tree recursive algorithm

[1]Notice that we slightly changed the definition of `bind`. Besides a computation (x) and a computation (f) parametrised by the value of the first computation, `bind` is paramtrised with the name (n) of the message for which `bind` was used. This way, we have a finer grained control over how the monad is used by the program. From this perspective, `bind` can be regarded as a very abstract notion of a MOP since it can intercept messages from the 'base level program'.

that does most work twice. This can be optimised by caching the computed Fibonacci numbers (a technique called memoizing):

```
(define **fiblist** '())

(define (unit x) x)

(define (bind x n f)
 (if (equal? n 'fib)
   (let ((aFib (assq (O: x ?) **fiblist**)))
    (if aFib
     (cdr aFib)
     (begin
       (set! aFib (f x))
       (set! **fiblist** (cons (cons (O: x ?) aFib) **fiblist**))
       aFib)))
   (f x)))

(define (run x) x)
```

In this variant of the state monad, we use the name of the message to intercept calls to `fib`. We then look up the receiver in a list of (n,fib(n)) pairs. When it is already in the list, we simply return its associated Fibonacci number. Otherwise, we actually perform the computation and store its result in the list for later usage.

The final experiment consists of allowing both branches of the Fibonacci recursion tree to be computed in parallel. We therefore use the special form `(future ...)` which starts the (Scheme) computation `...` in parallel with the rest of the computation. Only when its value is needed, the interpreter will wait for the parallel computation to be finished. Here are the implementations of `unit`, `bind` and `run`:

```
(define (unit x) x)

(define (bind x n f)
  (future (f x)))

(define (run x) x)
```

Gluing together `x` and `f` now consists of returning `(f x)` but allowing the interpreter to compute this in parallel with the remainder of the computation (which is the monadic method whose implementation called `bind` in the first place).

The above experiments strongly indicate that monadic style programs and AOP have a lot in common. By using another monad, we get system-wide changes. Furthermore, our code only contains the functional aspects of a normal Fibonacci implementation as taught in high school. Both the caching and parallellism aspects can be put in a monad without cluttering up the code. An argument in favour of monads is that our implementation uses Scheme which is rather close to denotational semantics. Another argument is that only very recently, monads have been used to split objects into several 'features' using a method quite different from ours [7]. In this work, objects have features (aspects?) and adding more features is done by using a richer monad. As such, our experiments and those of Prehofer [7] indicate that the monad concept might be a very good candidate to give formal semantics to aspect-oriented programming languages. This can in its turn improve our understanding of AOP.

As can be seen from our experiments, `bind` is certainly a major join-point of the code. As a matter of fact, we didn't use any monad operations at all. Hence, a natural reaction would be to separate every aspect in a single monad and combining their `bind`'s into a single `bind` to be used by the monadic program. Unfortunately, monads do not compose[3]. If we have a monad `c x` and a monad `c' x`, `c(c' x)` is not necessarily a monad. The problem is that in general, it is not possible to implement `bind` for `c(c' x)` solely in terms of `unit` and `bind` of `c` and `c'`.

However, monads *do* transform using so called monad transformers. A monad transformer is a monad `t` parametrised by another monad `m`. Programming `unit` and `bind` for a monad transformer is exactly the same as programming them for a monad, except that one can use the `unit` and `bind` of the monad `m` with which the transformer is parametrised. For an extensive discussion of the technicalities concerning (the relatively new) monad transformer technology, we refer to [4] and [1].

5. Conclusions and Future Research

The experiments described in this short paper illustrate a very strong simularity between AOP and MP. Of course, much more research needs to be done to completely understand that relation. At the time of writing, the status of our work at least raises the following questions:

• What is the relation between our approach to AOP and Prehofer's feature oriented programming ?

• How can monads (and the experience with monads in the functional community) be of any help to identify aspects in a program or in a problem statement ?

• Much of the monad technology used in functional programming involves the implementation of visitors (the simplest example being an evaluator). What is the relation with Adaptive Programming [5] as described by Lieberherr ?

• As already indicated in the text, our version of `bind` is slightly more general than the conventional one because it is extra parametrised by the name of a message. This formulation of monads immediatly raises the question what the relation is between monads and meta-object protocols. As described by the AOP papers, the latter is strongly related to AOP.

• At first sight, monads seem to be able to capture a variety of completely different aspects and it seems to be a very interesting research question to see just how far we can take this.

7. References

[1] D. A. Espinosa. Semantic Lego. PhD. Thesis, Columbia University, 1995.

[2] M. Jones. Introduction to Gofer 2.20. Ftp from `nebula.cs.yale.edu` in `pub/haskell/gofer`, September 1991.

[3] M. Jones and L. Duponcheel. Composing Monads. Technical Report YALEU/DCS/RR-1004, Yale University, 1993.

[4] S. Liang, P. Hudak, and M. Jones. Monad Transformers and modular interpreters. In Conference Record of POPL'95, page 472, ACM Press, 1995.

[5] K. J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, 1996.

[6] E. Moggi. A modular Approach to Denotational Semantics. Category Theory and Computer Science - Lecture Notes in Computer Science, 530:138, 1991.

[7] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects, to appear in ECOOP'97 Proceedings, Springer Verlag, 1997.

[8] P. Wadler. The essense of functional programming. In Conference Record of POPL'92, ACM Press, 1992.