Monadic Methods

Wolfgang De Meuter - Theo D'Hondt
Vrije Universiteit Brussel - Computer Science Department
Pleinlaan 2 - 1050 Brussel - Belgium

Abstract

Object-oriented frameworks are often based on 'skeleton' methods that determine the overall control flow and that carry around 'contextual information'. Changing the signature of a skeleton method often induces a combinatorial explosion of changes to other methods spread over the different classes of the framework. Our work materialises these skeleton methods as *monadic methods* that carry around monadic information. The nature of the information and the way it flows through the code is determined by a so called monad. We investigate monadic methods, and explain how the monadic style limits the propagation of changes to skeleton methods.

1 Introduction

An object-oriented framework [4] is usually described as a well-designed set of co-operating classes that solve problems from an entire problem space. Instances of a framework are created by overriding (abstract) methods in subclasses, or by plugging in new classes at some previously specified sites. As such, a framework defines a skeleton for applications, which are created by furnishing the framework with the details of a particular problem in the problem space.

Although informal techniques like design patterns [2] are extremely valuable in the construction of good frameworks, they only *document* a framework. When really diving into the code, one still has to struggle with 'plain' language constructs such as message passing and assignments. These do not reveal whether one is using or changing an essential design choice of the framework, or whether one is manipulating detail code. At the code level, the only way to distinguish essentials from details is by a good choice of identifiers and by informal annotations. Therefore, recently, techniques have been proposed to formally express some of the laws that *define* a framework. One example are contracts [3] which allow one to express the communication laws between objects. Another example are reuse contracts [17] which allow the designer and the reuser of a class to interact by means of a formal contract.

This paper proposes *monadic methods* as a way to formally capture the 'skeleton' methods of a framework. Although every object-oriented program consists of objects that send messages to each other, one of the added values that comes with a framework is the identification of the crucial messages that really tie large parts of the framework together. These messages are often characterised by the fact that they carry around 'contextual information' that is crucial for the entire framework but cannot be stored in a global variable. Examples thereof are the environment parameter of an evaluator, a 'current user' in a multi-user system, and so on. Much of the definition of a framework lies in finding the right abstraction levels for these methods. Proof of this are the various behavioural design patterns documenting a traversal that is parametrised by contextual information. Methods that carry around such information will be called monadic methods.

Section 2 concretely motivates our work. Section 3 introduces monads and monadic style programming, which is an abstraction technique that originated in denotational semantics. Section 4 discusses how the monadic technique can be translated to the world of objects and messages. Section 5 illustrates how the monadic style of object-orientation allows us to write highly parametrised 'monadic' methods. We will argue that these methods are the ones framework developers often experience as the 'main threads' of their design.

2 Motivation

Suppose we are writing a system to create computer assisted learning applications. The interface of the system resembles that of a drawing program, but instead of manipulating rectangles and circles, a teacher is linking together graphics, texts, multiple choice and open questions. Once a lesson has been completely edited, it is saved as a stand alone application to be run by students.

A small part of this system models open questions. These are internally stored as a list of 'question chunks', which can be pieces of text, drawings or fill-in fields in which the student is supposed to answer. The application processes an open question in two phases. First, the chunks are preprocessed. In the case of textual and graphical chunks, this draws the information on the screen. Preprocessing a fill-in field might show an incorrect word and consult a database to find its corrected version. After an answer is given, each chunk is traversed in order to check it. Of course, checking graphics and text chunks always returns 'true'. Checking fill-in fields will match the answer of the student to the correct answer determined at preprocessing time.

Besides a hierarchy of questions, our system will contain a hierarchy of chunks. Each chunk defines a `preprocess()` method that preprocess the chunk and calls `preprocess` on its `nextChunk`. Each chunk also contains a `check()` method that checks it. If evaluation fails, 'false' is returned. Otherwise, `check` is sent to `nextChunk`.

Let us now adapt the system for doing physics problems. A question will therefore need 'variable' chunks that bind a name to a random value. These names are used by a chunk in which a formula resides. At preprocessing time, the values must be generated and bound to the names which are used by the formula to store the expected answer. At checking time, the formula chunk compares the stored answer to the answer of the student. This adaptation requires a drastic change. The `preprocess` method of every chunk class must be adapted such that a set of bindings is forwarded to the next chunk. In the new 'variable' chunks, a binding is added to the set. Preprocessing the new fill-in fields then evaluates the formula in the accumulated set of bindings.

Another requirement might be that a formula may occur at the start of the chunk list. Again, `preprocess` must forward a set of bindings through the list, but at the end, the set must be propagated backwards to the formula. Again, every `preprocess` method must be changed so that it gets the bindings, forwards them to `nextChunk` and returns the resulting set back to its caller.

In both changes, the only option seems to be a re-implementation of the entire chunk hierarchy in order to pass around the set of bindings. If there are many chunk classes, this becomes a time consuming adaptation of the code, while only two kinds of chunks really need that set: the chunks that declare a new variable, and the ones that need the variables.

In general, each object-oriented program consists of objects sending messages to each other. The added value of a framework is that one identifies certain *crucial* messages that tie the parts of the framework together. These messages forward the context of the framework to other parts of the framework and are responsible for propagating back results. A problem that occurs over and over when using a framework, is that somewhere inside new code, suddenly an extra object is needed that was not originally transported through the framework by the crucial messages. A similar problem is that unexpectedly, additional results must be propagated back from the crucial messages. In most cases, both problems require many code to be changed [9].

Section 4 and 5 explain how this problem can be overcome by seeing the skeleton methods of a framework as abstract computations called monads. Monads were introduced in computer science by Moggi [11], [12] as a way to deal with the modularity problem of denotational semantics. Because of the categorial nature of Moggi's work, we had to wait for Wadler's popularisation [19] for monads to become widespread in functional programming. The following section summarises monads.

3 Monads

3.1 The Modularity Problem of Denotational Semantics

Consider a toy language that consists of additive numeric expressions. Its parse trees contain constant nodes like $(const\ 4)$, and nodes representing additions $(plus\ p_1\ p_2)$, where p_1 and p_2 are trees again. An evaluator for this language is an *eval* function which traverses a tree and produces a numerical result:

$$\begin{array}{lcl} eval & : & ParseTree \to Int \\ eval(const\ c) & = & c \\ eval(plus\ p_1\ p_2) & = & eval(p_1) + eval(p_2) \end{array}$$

Experiment 1: Let us extend this language with lexical variables so that the parse trees can contain variable reference nodes like $(ref\ x)$ and let-bindings of the form $(let\ x\ p_1\ p_2)$, where x is a variable and p_1 and p_2 are parse trees. Evaluating these nodes requires the evaluator to be parametrised by an environment parameter. Unfortunately, this requires the existing evaluator to be rewritten since the other nodes must also pass the environment down the tree[1]:

$$\begin{array}{lcl} eval & : & ParseTree \times Env \to Int \\ eval(const\ c, e) & = & c \\ eval(plus\ p_1\ p_2, e) & = & eval(p_1, e) + eval(p_2, e) \\ eval(let\ x\ p_1\ p_2, e) & = & eval(p_2, add(e, x, eval(p_1, e))) \\ eval(ref\ x, e) & = & lookup(x, e) \end{array}$$

Experiment 2: Now consider adding statements to our language. This requires new parse trees for assignments $(set!\ x\ p)$ and for sequencing $(begin\ p^*)$[2]. Extending the evaluator for these requires that *eval* is parametrised by an environment as in the previous experiment. However, as assignments might alter this environment, *eval* must now return a pair consisting of a value and a (possibly) changed environment. Again, in order to propagate these pairs back, the existing evaluator must be completely rewritten.

[1] In the code we assume a function $add(env, symb, val)$ that extends an environment and returns the resulting environment. $lookup(symb, env)$ should be self-explaining.

[2] p^* means a list of p's

3.2 Monadic Style

The core of the modularity problem of denotational semantics is that when a language feature is added whose evaluation needs information that was not passed down the tree, the evaluator must be rewritten [13]. In the same vein, if the new part returns more than expected, everything must be rewritten since the result should be dealt with in the old part of the evaluator.

This problem can be avoided by writing the evaluators in *monadic style*, using a so called *monad*. A monad can best be described as 'a computation that eventually returns a value'. Since the type of this value is not hardcoded in the monad, a monad is parametrised by a type variable t representing the type of values over which the computations act. A monad thus is a type constructor, much in the spirit of a template class in C++.

$$compOver(t) = ...t...$$

Each monad implements two polymorphic operations called *unit* and $\star$ (pronounced 'bind'). *unit* injects any value into a zero computation that, when executed, just returns that value. Hence, the type of *unit* is:

$$unit : t \rightarrow compOver(t)$$

$\star$ is a polymorphic operation to glue together a computation c_1 and a function representing a computation c_2 that is parametrised by the value of the first computation. $\star$ returns a computation that, when performed, executes the first computation and passes its value to the parametrised second computation right before executing the second computation. Hence, its type is:

$$\star : compOver(t_1) \times (t_1 \rightarrow compOver(t_2)) \rightarrow compOver(t_2)$$

Writing a program in monadic style means that the program has no knowledge of the particular monad and only uses *unit* and $\star$. As an example, we can rewrite our very first evaluator in monadic style:

$$\begin{array}{lcl}
eval & : & ParseTree \to compOver(Int) \\
eval(const\ c) & = & unit(c) \\
eval(plus\ p_1\ p_2) & = & eval(p_1) \star \lambda v_1.(eval(p_2) \star \lambda v_2.unit(v_1 + v_2))
\end{array}$$

eval now delivers computations that, when executed, return the numeric value of the expression. A monadic style program typically consists of a number of occurrences of $\star$ thereby passing λ-expressions which contain nested occurrences of $\star$. These can use the value of the enclosing occurrences through lexical scoping:

$$c_1 \star \lambda v_1.(c_2 \star \lambda v_2.(\ldots \lambda v_n.unit(doIt(v_1, v_2, \ldots, v_n))))$$

Experiment 1: Let us now redo the first experiment. Since the evaluator for *ref* requires an environment, it will have to assume that the computations carry around an environment. It will therefore require a computation

$$getEnv : compOver(Env)$$

that retrieves the environment that is carried around. The evaluator for *let* will also need an opposite operation to let a computation use an updated environment:

$$setEnv : Env \times compOver(t) \to compOver(t)$$

Using these operations to get and set the environment that is carried around in the computation, the evaluator for *let* and *ref* is implemented as follows:

$$\begin{array}{lcl}
eval(ref\ x) & = & getEnv \star \lambda e.unit(lookup(x, e)) \\
eval(let\ x\ p_1\ p_2, e) & = & eval(p_1) \star \lambda v.(getEnv \star \lambda e.setEnv(add(e, x, v), eval(p_2)))
\end{array}$$

The nice thing about writing the evaluator clauses for *let* and *ref* is that we didn't have to modify the existing clauses for expressions. However, the new evaluator fragment requires that computations are parametrised by an environment and that there are two operations to get and set this environment. In jargon, we say that the new evaluator clauses assume that the evaluation monad is the so called *environment monad*, which looks as follows:

$$\begin{array}{lcl} compOver(t) & = & Env \to t \\ unit & : & t \to (Env \to t) \\ unit(x) & = & \lambda e.x \\ \star & : & (Env \to t_1) \times (t_1 \to Env \to t_2) \to (Env \to t_2) \\ c \star f & = & \lambda e.f(c(e), e) \end{array}$$

unit injects a value in a zero computation. The result of $\star$ is a computation that performs the first one $c(e)$ and passes its value to the function. In the environment monad, *setEnv* and *getEnv* are implemented as follows:

$$\begin{array}{lcl} getEnv & = & \lambda e.e \\ setEnv(e, c) & = & \lambda e'.c(e) \end{array}$$

getEnv is a computation that returns the underlying environment. *setEnv* takes an environment e and a computation c. It returns a computation that neglects its environment, but instead performs c in e.

Experiment 2: The second experiment required the evaluator to take an environment and yield a numeric value together with a (possibly updated) environment. This is achieved in the so called *state monad* which looks as follows:

$$\begin{array}{lcl} compOver(t) & = & Env \to (t \times Env) \\ unit & : & t \to (Env \to (t \times Env)) \\ unit(x) & = & \lambda e.(x, e) \\ \star & : & (Env \to (t_1 \times Env)) \times (t_1 \to Env \to (t_2 \times Env)) \to (Env \to (t_2 \times Env)) \\ c \star f & = & \lambda e.let\ (v, e') = c(e)\ in\ f(v, e') \end{array}$$

unit returns a zero computation that returns a value and an unmodified environment. $c \star f$ is a computation executing c in the given environment. The resulting environment e' is fed into f after feeding it with the value v of c. Adding evaluator clauses for assignment and statement sequencing and writing *setEnv* and *getEnv* for the state monad is a fairly easy functional programming assignment. Important is that the existing evaluator can be reused completely.

Remarks

1) Operations like *setEnv* and *getEnv* are called *monad operations*. By using *setEnv* and *getEnv*, the evaluator for *let* and *ref* admits that it must be exe-

cuted in a monad which has at least the complexity of the environment monad.
2) An operation that is not part of the categorial definition of monads, but that is assumed by most functional programmers is $run : compOver(t) \rightarrow t$. In both the environment and the state monad, *run* starts a computation by feeding it with an empty environment. In the case of the state monad, the final environment is 'thrown away' and the value is returned.
3) Although the environment and the state monad are most frequently used, many other monads exist. We refer to [19] and [18].

4 Monadic Methods

This section investigates how a pure OOPL based on objects and messages can be enriched with monadic programming constructs. In section 4.1 we establish the basic object model we assume. Based on this, we define monadic methods and explain how monadic and ordinary methods interact. One of the driving forces behind our proposal is to hide *unit*, $\star$ and *run* as much as possible for the programmer. Of course, the evaluation rules[3] for the constructs will use them extensively.

4.1 A Simple Model of Objects and Message Passing

Every OOPL evaluator uses a way to represent objects. Both [10] and [16] show how this representation can vary between several kinds of languages. In this paper, we assume the simplest model in which objects are records containing methods. These methods are functions that take a list of argument objects and return an object. The domain of object representations can thus be modelled as

$$o = Record(o^* \rightarrow o)$$

[3] We will use a 'denotational semantics'-like meta-language. However in order to keep our notation readable, we assume the language has state and error handling.

Although some evaluators implement several operations on their object domain (e.g. 'copy'), they all at least implement message passing . Hence it is reasonable to assume the operation

$$pass_o : o \times Ident \times o^* \to o$$

on the elements of our object domain o. $pass_o$ takes a receiver, an ID for the message and a list of arguments, and it returns an object. In our record model, $pass_o$ is implemented as method selection followed by method invocation.

4.2 Monadic Methods

The general idea of our work is to allow objects to declare so called *monadic methods* besides the 'ordinary' methods described above.

Monadic methods are methods whose body is evaluated in a monad. Hence, the type of monadic methods is $o^* \to m(o)$ where m(t) is a monad[4]. That is, instead of returning an object, monadic methods return a computation whose execution will eventually result in an object. Such a computation of type $m(o)$ will be referred to as a *monadic object*. Allowing objects to contain monadic methods besides ordinary methods, means that the object domain of section 4.1 must be adjusted to

$$o = Record(o^* \to o \oplus o^* \to m(o))$$

where $\oplus$ is the disjoint union operator for domains.

Notice that the arguments of a monadic method are ordinary objects. The reason is that we choose the standard applicative order model of object-orientation in which arguments are evaluated before a message is sent. The alternative is to model monadic methods as $(m(o))^* \to m(o)$. This induces a normal order model since arguments are not run before method invocation. Investigating normal order methods is beyond the scope of the paper.

[4] In order to make our notation more explicit, we index the necessary operations by m yielding $unit_m$, $\star_m$ and run_m. We also assume that m satisfies the classic monad laws [19].

We assume that the body of ordinary methods is a sequence (in a `beginO` construct) of message expressions[5]. Its return value is the value of the last message expression that occurs in it. Likewise, monadic methods are also expected to contain a sequence of message expressions. But unlike ordinary methods, the expressions are expected to yield monadic objects. The task of the sequencing construct then is to glue them all together. Monadic methods will therefore use a different sequencing construct than ordinary methods. We will call it `beginM`. Hence, the result of a monadic method is a monadic object, which is a computation that, once performed, executes all the computations in the sequence and returns the value of the last expression occurring in that sequence. The evaluation rule[6] for both ordinary and monadic method bodies is given below[7]:

$$\begin{array}{lcll}
evalBegin(lst) & = & eval(hd(lst)) & |lst| = 1 \\
 & = & first\ eval(hd(lst)) & \\
 & & then\ evalBegin(tl(lst)) & |lst| > 1 \\
evalBegin_M(lst) & = & eval_M(hd(lst)) & |lst| = 1 \\
 & = & eval_M(hd(lst)) \star_m \lambda v.evalBegin_M(tl(lst)) & |lst| > 1
\end{array}$$

Hence, when evaluating an ordinary method, all the expressions occurring in the `beginO` of that method are evaluated. When evaluating a monadic method, a computation is returned which consists of binding together all the computations occurring in the `beginM` of that monadic method.

4.3 Object Meets Monad

In order to study the interactions between monadic and ordinary methods, we take a closer look at the message expressions occurring in the above sequencing constructs. Syntactically, each message expression consists of a receiver, a name and a list of actual arguments. Of course the receiver and the arguments can be message expressions in their turn.

[5] We assume a 'pure' object-oriented language (like Smalltalk) in which everything but statement sequencing and assignment is accomplished via message passing. Assignment is postponed until section 4.4

[6] Since we assumed a meta-language with state, it has a sequencing construct 'first ... then ...' as well

[7] hd, tl and $|.|$ respectively return the head, the tail and the length of a list. We distinguish between $eval$ and $eval_M$. The former yields ordinary objects, the latter yields monadic ones.

Since instance variables are postponed until section 4.4, each object visible inside a method is either an actual argument, the receiving object denoted by `self` or a result of a message expression. Concerning the nature of these objects we can say that:

1. because of our applicative order assumption, all actual arguments of a (monadic) method are ordinary objects of type o.
2. the same goes for the receiver since the 'real' object value of an expression must be computed for otherwise, we couldn't have sent it a message.

Hence, the only objects that still need investigation are results from messages that are sent from within a method. Here we must make a distinction between the two kinds of methods:

Ordinary Methods. First consider the situation of an ordinary method. Since `begin0` does not know how to handle monadic objects, the result of a message expression occurring in it must be an ordinary object. This together with (1) and (2) above implies that all objects that can ever enter an ordinary method will be of type o. Therefore, the message passing operator to be used by ordinary methods will have the type

$$pass_o : o \times Ident \times o^* \to o$$

As explained in section 4.1, this operator will be implemented as a combination of method selection and invocation. However, due to our adjusted object domain from section 4.2, the method that will be found in the selection process can be an ordinary method or a monadic method. In the case of an ordinary method, there is no problem and we can simply invoke the method using ordinary function application. In the case of a monadic method, the computation returned by the monadic method must be run_m in order to get a 'real' object of type o. Hence, the message passing operator to be used *by ordinary methods* is defined as follows:

$pass_o(rec, msg, args) = apply(lookup(rec, msg), args)$
if $lookup(rec, msg) \in o^* \to o$
$pass_o(rec, msg, args) = run_m(apply(lookup(rec, msg), args))$
if $lookup(rec, msg) \in o^* \to m(o)$

When ordinary methods send messages implemented by ordinary methods, function application is used. When calling monadic methods from ordinary methods, the resulting computation must be run.

Monadic Methods. The other case to be considered is the evaluation of the message expressions occurring inside a monadic method. Because of the evaluation rule for `beginM`, all the message expressions occurring inside `beginM` must yield a monadic object. Hence, the result type of the message passing operator to be used must be $m(o)$. But since the receiver and the arguments of a message expression can also be message expressions, they must also be of type $m(o)$. Thus, the message passing operator to be used *by monadic methods* has the following type:

$$pass_{m(o)} : m(o) \times Ident \times (m(o))^* \to m(o)$$

Hence, a monadic method requires that all visible objects are monadic . But since (1) and (2) described above dictate that all actual arguments and the receiver of the monadic method will be ordinary objects, we must implicitly inject these into a zero computation using $unit_m$ such that they can be used in the monadic method.

Thus, $pass_{m(o)}$ sends a message to a monadic object with a suite of monadic parameters. It is implemented by binding all the computations together with $\star_m$ and seq_m[8]. Then the method is selected in the record. If the result happens to be an ordinary method, that method is invoked and its result is injected in a zero computation that can be glued between the computations of the calling monadic method. If on the other hand, the method is a monadic method,

[8]the seq_m operation transforms a list of computations into a computation that returns the list of values. It works for every monad since its implementation uses only $unit_m$ and $\star_m$.

its computation is glued between those of the calling monadic method. The implementation of $pass_{m(o)}$ is given below:

$$\begin{array}{lll}
\multicolumn{3}{l}{pass_{m(o)}(rec, msg, args) = rec \star_m \lambda r.(seq(args) \star_m \lambda a.unit_m(apply(lookup(r, msg), a)))} \\
\multicolumn{3}{l}{\quad \text{if } lookup(r, msg) \in o^* \to o} \\
\multicolumn{3}{l}{pass_{m(o)}(rec, msg, args) = rec \star_m \lambda r.(seq(args) \star_m \lambda a.apply(lookup(r, msg), a))} \\
\multicolumn{3}{l}{\quad \text{if } lookup(r, msg) \in o^* \to m(o)} \\
seq_m & : & (m(t))^* \to m(t^*) \\
seq_m([]) & = & unit_m([]) \\
seq_m([m, ms]) & = & m \star_m \lambda o.(seq_m(ms) \star_m \lambda os.unit_m([o, os]))
\end{array}$$

Message Passing Summarised

The complete message sending semantics is outlined in Fig.1. The message

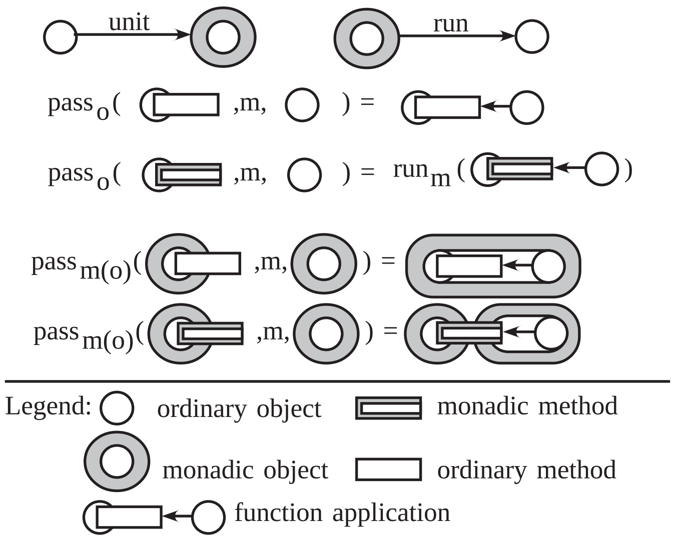

Figure 1: Message Passing

passing operator $pass_o$ used from within ordinary methods takes and returns ordinary objects. If the method corresponding to a message expression is an ordinary method, we simply invoke it. If it happens to be a monadic method, the resulting computation must be run_m in order to yield an ordinary object. The message passing operator $pass_{m(o)}$ used by monadic methods takes and yields monadic objects. If the method corresponding to a message expression is an ordinary method, the monadic information flows 'over' that method. If the method happens to be monadic, the monadic information flows 'into' (and perhaps back out of) that method.

4.4 Instance Variables

In the previous sections, we omitted instance variables. Accessing and assigning instance variables can be accomplished in two ways[9]:

1) The **first option** is to allow only ordinary objects to be stored in instance variables. Ordinary methods can then read and write instance variables as usual. Reading an instance variable by a monadic method requires its value to be implicitly $unit_m$-ed. Assigning instance variables `var := rhs` in monadic methods is more subtle. Since all objects that enter a monadic method are monadic, `rhs` yields a monadic object. This object must be $\star_m$-ed in order to store its 'real' value v:

$$eval_M(x \;:=\; rhs) \quad = \quad eval_M(rhs) \star_m \lambda v. \begin{array}{l} \text{first assign } v \text{ to } x \\ \text{then return } unit_m(v) \end{array}$$

This treatment of instance variables will be called the *o-approach.* The *o*-approach has the advantage that it is very easy to think about from a programmers point of view: the *o*-approach treats instance variables 'as expected'.

2) The **second option** is allow both kinds of objects to be stored. This option is called the $(m(o) \oplus o)$-*approach* and is more difficult to think about. The evaluation rule for assignments `var := rhs` is very subtle. Assignment by ordinary methods is trivial since `rhs` then yields an ordinary object. However, when the assignment appears in a monadic method, `rhs` is monadic. The assignment may not glue this monadic object in the computation of the assigning

[9]Notice that instance variables were not included in our record model. In order to be complete, we would have to use the standard denotational technique of stores and locations.

monadic method since this would cause the computation to be done twice; once when evaluating the assigning method and once when reading the instance variable. Hence, the evaluation rule for assignments in monadic methods using the $(m(o) \oplus o)$-approach looks as follows:

$$\begin{array}{rcl} eval_M(x \ := \ rhs) & = & \text{first assign } eval_M(rhs) \text{ to } x \\ & & \text{then return } unit_m(o_{undefined})) \end{array}$$

Reading a variable is quite easy in the $(m(o) \oplus o)$-approach. Ordinary (resp. monadic) methods will implicitly run_m (resp. $unit_m$) monadic (resp. ordinary) objects stored in variables. However, when monadic objects in variables must be run_m, there are two options again. The first one is to use the resulting ordinary object and let the instance variable still point to the computation it contains. But then referring to the variable twice will run the computation twice, which might give unexpected results. In the second option, the monadic object is run and the instance variable is updated with the result thereof. In this case, the computation is executed and each future reference to the variable will use the resulting ordinary object.

At the time of writing, it is unclear which approach to prefer. An OOPL might even allow them both by having two kinds of variables. Experience with monadic software will have to teach us how the approaches relate in practice.

4.5 Summary

In section 4, we defined monadic methods and explained how they can be added to a pure OOPL with a record-like object model. In each subsection, the evaluation of a feature was explained. Putting all the rules together gives us a complete monadic OOPL evaluator that incorporates objects, messages, (monadic) methods and assignments. The following section explains how these constructs can be used to solve the problems outlined in section 2.

5 Monadic Object-Oriented Programming

5.1 A Monadic OOPL

We implemented [10] the rules of section 4 in Scheme using a syntax-extension facility resulting in a system with the following syntax constructs. Notice that we have only one kind of variables for which we choose the *o*-approach because of its simplicity.

```
<class> -> (class <name> <parentname> (var <name>)* <method>*)

<method> -> (methodO <name> <formal>* (beginO <expressionO>*))
         |  (methodM <name> <formal>* (beginM <expressionM>*))

<expressionO> -> (passO <expressionO> <name> <expressionO>*)
              |  (:=O <name> <expressionO>)
              |  (selfO)
<expressionM> -> (passM <expressionM> <name> <expressionM>*)
              |  (:=M <name> <expressionM>)
              |  (selfM) -- "unit"-ed version of (selfO)
```

In the evaluation rules, we need $unit_m$, $\star_m$ and run_m. In our system, we implemented them using ordinary Scheme lambda's which we store in global Scheme variables. Of course this implies that the current status of our work does not allow different classes to use different monads unless we explicitly implement 'monad switching' Scheme code in our methods. This problem was never raised by the functional community and solving it requires more research on monadic OOPL and monads in general. We discuss this in our future work section.

Now that we know the syntax and semantics of *general* monadic object-oriented programs, we turn our attention to particular monads and describe how to incorporate their monad operations in an OOPL. In this paper, we focus on the environment and state monads because these are so frequently used. We refer to [19] and [18] for the definition of other monads. Some of them, such as the error monad, will translate quite easy to object-orientation. For others , such as the continuation monad, this is less clear.

[10] See `http://progwww.vub.ac.be`.

In a pure object-oriented setup, monadic information such as environments are also objects. Thus, the environment monad will be $m(o) = o \rightarrow o$ where the first o is the environment parameter and the second o is the resulting value. Likewise, the state monad will be $m(o) = o \rightarrow o \times o$ where the left part of the pair indicates the 'value' of the computation and the right part is the resulting environment.

In order to access the monadic information, programs need monad operations like $getEnv$ which is a computation that once executed, returns the 'current' environment. Since computations correspond to monadic objects, we need a syntactic construct to denote a monadic object whose value is bound to the environment carried around. We propose to use a pseudo variable, similar to the use of `self` in ordinary OOPL's. A monadic program carrying around an environment can give access to the environment through the pseudo variable `envM?`. We thus add `(envM?)` to our list of `<expressionM>`'s. Its semantics is $getEnv$ as implemented in section 3. Of course, we also need a syntactic variant for $setEnv$, with which we can update the environment flowing through monadic code. We use the 'parametrised pseudo variable' `(envM! envM expM)` for this purpose. It's precise semantics is $eval_M(\texttt{envM}) \star_m \lambda e.setEnv(e, eval_M(\texttt{expM}))$ because $setEnv$'s first parameter is an ordinary value (see section 3) while all objects in a monadic method are monadic. Notice that `(envM! envM expM)` has a different meaning in the environment monad than in the state monad. In the environment monad, `envM` enters `expM` but the original environment is restored. In the state monad, `envM` enters `expM` and its resulting environment is used in the remainder of the method. We will use this in the next section.

5.2 The Example

Let us now return to the 'question chunk' example discussed in section 2. The only task of the checking method `check` is to check each chunk and return the corresponding boolean value. When checking a new kind of chunk requires extra information, our design dictates that it was the job of `preprocess` to make sure the information got to the chunk somehow. This is made tangible by explicitly declaring **preprocess** as a monadic method. The other code of the chunk hierarchy consist of ordinary methods of that manipulate ordinary objects.

```
(class Chunk  Root
   (var nextChunk)
   (methodO check ()
       (beginO (passO nextChunk check))))

(class Graphic Chunk
   ...
   (methodO show () ...)
   (methodM preprocess ()
       (beginM (passM (selfM) show)
               (passM nextChunk preprocess))))

...
```

In the **preprocess** method, the monadic information enters **beginM** and goes through both message expressions. In the first one, an ordinary method **show** is called such that the information will flow over that method. In the second one, the information enters **preprocess** of **nextChunk** since **preprocess** is a monadic method. Notice that, due to our evaluation rules, the monadic code looks just like the ordinary code.

Our first change to the system concerned adding variable and formula chunks. As explained, preprocessing them requires a set of bindings to flow through chunk lists. Since we designed **preprocess** as a monadic method, this will now happen automatically when running the system in the environment monad. All we have to do is implement the new chunk classes and let their **preprocess** method access the set of bindings. The code for all other chunks remains valid.

```
(class Variable Chunk
   (var symbol)
   ...
   (methodM preprocess ()
       (beginM (envM! (passM (envM?) add symbol (passM float newRandom))
                      (passM nextChunk preprocess)))))

(class Formula Chunk
   (var itsParseTree)
(var suppliedAnswer)
(var expectedAnswer)
...
(methodM preprocess ()
    (beginM (:=M expectedAnswer (passM itsParseTree eval (envM?)))
            (passM nextChunk preprocess)))
(methodO check ()
    (beginO (passO (passO suppiedAnswer = expectedAnswer)
                   and (passO nextChunk check)))))
```

Variable chunks are preprocessed by reading the bindings from the monad using `(env?M)` and preprocess the next chunk in an updated environment. A formula reads the bindings from the monad in order to evaluate its parse tree. If we design `eval` as a monadic method, the set does not even have to be accessed in the formula chunk since it would automatically enter `eval`. In order to get our second change (i.e. formulas may appear before variables in the chunk list), we simply run the system in the state monad. We must also switch the message expressions in the `preprocess` method of the formula chunk because the state must first visit all the chunks before the formula is evaluated. However, this is a very local change in code that explicitely uses the monad operations.

This example was deliberately kept small for didactic purposes. Our set of bindings is a very 'pure' application of the environment monad. A monad more common in object-oriented software that is actually a generalisation of the environment monad, could be called the *the context monad*. In the context monad, an aggregate of several objects flows through the methods. Many frameworks implement it by 'by hand' by passing a context object to every important method. The disadvantage of passing around a context 'by hand' is that it blurs many methods that actually don't need the context but have to forward it to methods that *do* need it. Our approach only requires these methods to be monadic: it is not visible in their signature and code what kind of information is actually passed around, and more importantly, how it is passed around. This can be particularly useful for passing around information through bought libraries . Another disadvantage of the context by hand approach is that it only parametrises the input parameters of a method. If unexpectedly, more results must be returned from a chain of method calls, everything must be rewritten.

In several cases, this also requires one to put `if`-clauses in the code that test one of the results in order to decide whether to do something or continue the returning process. This is merely a variant of the error monad.

5.3 Designing Skeleton Methods

If an ordinary object-oriented program is seen as a graph of objects sending messages to each other, a monadic program can be seen as the same graph, underneath which a system of tubes is plumbed that transparently transports information from one part of the graph to another. The precise contents of the tubes and the way they are connected to each other is determined by the particular monad. Objects that need to access the tubes can do this by the monad operations. Sometimes, it is important not to have unconnected segments in the tube system. This happens when a monadic part of the system M_1 calls ordinary methods which in their turn call methods in a monadic part M_2. Because the monad information will not flow through the ordinary methods, the tubes of M_1 are not connected to those of M_2. In other cases, it is most important to separate the two tube systems, simply because they have nothing to do with each other or because they are not compatible with each other as they depend on different monads.

Thinking about monads in terms of tubes can be useful, for instance, to study their interaction with inheritance, a topic that was not touched in this paper. What happens when monadic methods are overridden by ordinary methods, or vice versa? Suppose we have three monadic methods m_1, m_2 and m_3 that call each other in a sequence. If we override m_2 by an ordinary method, we separate the tubes of m_1 and m_3 which results in a loss of monadic information. Conversely, if m_2 originally was an ordinary method, and we override it by a monadic one that still calls m_3, we link together two separate tube systems. Although this is not even always technically possible (see future work), it is an important decision.

Although it requires much more research (and experience!), this discussion indicates that the ‘monadicness’ of a method is important design information. By declaring a method as ‘ordinary’, we actually say that it only depends on ‘local’ information like actual arguments, instance variables and results from sending other messages. If on the other hand, we declare a method as ‘monadic’, we express that it is capable of transparently transporting system-wide information that it relies on, or that might be needed by some parts of the program that it activates. Furthermore, by declaring a method as ‘ordinary’, we express that the method will always be hooked on the standard object-oriented evaluation scheme consisting of method invocation, statement sequencing and returning from a method. In a monadic method on the other hand, the $\star_m$ operation might neglect complete computations such that it can have influential effects on how the computation proceeds, much in the style of exception handling and CPS. These arguments convince us that a language making the distinction between monadic and ordinary methods forces a programmer to thoroughly consider whether she is dealing with a local detail method or with a ‘skeleton’ method whose design can have system-wide repercussions. However, this can only be known for certain by extensive experience with monadic OOPL’s.

6 Related Work

Although the idea of monadic object-oriented programming is completely new, the motivation for our work was inspired by previous research. In **adaptive programming** [9], methods can be spread over different class structures using a so called ‘propagation pattern’, supplemented by ‘transportation patterns’ that determine how objects are transported through an object structure. While propagation patterns seem pretty orthogonal to monadic methods, there seems to be a strong correspondence between monads and transportation patterns. Another technique to get a similar parametrisation are **open implementations** [7] in which systems have a base-level interface to ‘use’ it, and a meta-level interface to adapt it ‘from above’. In fact, the research described in this paper originated from conceptualising reflection in the Agora framework [15] which is an open implementation for evaluating reflective OOPL's.

7 Future Research

Although this paper describes a clean-cut technical treatment of monadic OOPL, it also raises many questions:

Static Type Checking. The paper only focuses on the 'meta-types' of the evaluator and doesn't pay any attention to the interfaces of the objects. An important question to be answered is how to know the interface of an object put in a monad 5 million lines away ? Perhaps the recent integration of constructor classes [5] in an OOPL [14] can play a role in this.

Implementing Monadic OOPL. The first thing we plan is to integrate monadic methods in an *existing* OOPL such that more practical experience can be gained with them. A question that was not addressed in our Scheme implementation is how monads themselves can be implemented with objects. We therefore plan to adjust the Smalltalk compiler since Smalltalk allows us to simulate lambda abstraction via block closures.

Monad Combinations. The mechanisms described in this paper work fine when an entire framework is written in the same monad. As indicated, 'monad management' by hand also allows different parts of a program to be written in different monads. However, this is only valid as long as these monads never 'meet each other', that is, as long as two 'incompatible tubes' are always separated by an ordinary method. When a message is sent from within a $n(o)$ and it is implemented by $o \rightarrow m(o)$, we run in trouble as neither $\star_m$ nor $\star_n$ can bind different kinds of computations. A related issue is the combination of monads which means writing methods in $m(n(o))$ if $m(t)$ and $n(t)$ are monads. This is a hard problem as monads in general do not compose [6]. A solution might be the insight of [8] that monads *do* transform, or Espinosa's proposal to stratify monads [1].We consider the translation of these to OOPL's as the most important technical open problem for monadic methods to become practically usable.

8 Acknowledgements

We sincerely thank Carine Lucas, Kim Mens, Kris De Volder, Tom Mens and Patrick Steyaert for reading early versions of the paper.

9 Conclusions

This paper launches the idea of monadic style object-oriented programming. Its technical contribution is a clean incorporation of monads in a pure object-oriented model, by defining monadic methods as methods whose body is to be executed in a monad. Based hereupon, we performed a careful analysis of the interactions between monadic and non-monadic methods, leading to a message passing semantics that is easy to understand and implement. This together with our proposal to access monadic information through pseudo variables makes monadic object-oriented programs look as natural as ordinary object-oriented code. As a second contribution, we have argued that the difference between monadic methods and ordinary methods makes the difference between local detail code and skeleton code that determines a system wide control flow and transparently transports contextual information through a design. Still, a lot of research and experience concerning monadic OOPL's is required in order to fully understand how far monadic methods can further demystify abstract and informal notions like 'architectures' and 'frameworks'.

References

[1] D. Espinosa. Modular Denotational Semantics. Unpublished Manuscript, 1993.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[3] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of Joint ECOOP/OOPSLA '90 Conference*, ACM Sigplan Notices, page 169. ACM Press, 1990.

[4] R. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA '92 Conference*, ACM Sigplan Notices, page 63. ACM Press, 1992.

[5] M. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorfism. In *FPCA'93 Conference on Functional Programming and Computer Architecture*, page 52. ACM Press, 1993.

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

[7] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of IMSA Workshop on Reflection and Meta-Level Architectures*, 1992.

[8] S. Liang, P. Hudak, and M. Jones. Monad Transformers and modular interpreters. In *Conference Record of POPL'95:22st ACM Symposium on Principles of Programming Languages*, page 472. ACM Press, 1995.

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

[10] J. Malenfant. On the Semantic Diversity of Delegation-based Programming Languanges. In *Proceedings of the OOPSLA'95 Conference*, ACM Sigplan Notices, page 215. ACM Press, 1995.

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

[12] E. Moggi. Notions of Computations and Monads. *Information and Computations*, 93:55, 1991.

[13] P. Moses. Denotational semantics. In J. Gunter, van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North Holland, 1990.

[14] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice . In *To appear in Proceedings of POPL '97 Conference*, 1997.

[15] P. Steyaert. *Open design of object-oriented languages, a foundation for specialisable reflective language frameworks.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1994.

[16] P. Steyaert and W. De Meuter. A Marriage of Class and Object-based Inheritance Without Unwanted Children. In *Proceedings of the ECOOP'95 9th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, page 127. Springer Verlag, 1995.

[17] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 Conference*, ACM Sigplan Notices, page 268. ACM Press, 1996.

[18] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi, Proceedings of the Marktoberdorf Summer School.* 1992.

[19] P. Wadler. The Essense of Functional Programming. In *Conference Record of POPL'92:19st ACM Symposium on Principles of Programming Languages*, page 1. ACM Press, 1992.