Combining Monads

David J. King Philip Wadler
University of Glasgow*

Abstract

Monads provide a way of structuring functional programs. Most real applications require a combination of primitive monads. Here we describe how some monads may be combined with others to yield a *combined monad*.

1 Introduction

Monads are taking root in the field of functional programming. Although their origins lay in the abstractions of category theory, they have a wide range of practical applications. Moggi [6] showed how they could be used to structure the semantics of computations. Since then Wadler [9, 10] adapted this idea to structure functional programs. When structuring functional programs like parsers, type checkers or interpreters, it is often the case that the monad needed is a combination of many, a so called *combined monad*.

For our purposes, we will think of a monad as a type constructor, together with three functions that must satisfy certain laws. For instance, we may have an interpreter and wish it to return, not just a value, but the number of reduction steps taken to reach the value. Later, we may want our interpreter either to return a value or an error message. Without using a monad or a similar discipline, it would be a messy undertaking to do in a purely functional programming language. If however our interpreter was written in a monadic style, we would merely have to change the monad to change what extra information the interpreter should return. An analogy can be drawn with a spreadsheet package which has rows relative to some formula. Instead of changing every element in a row we only have to change the formula to which the row is related.

For our interpreter to return the number of reductions and error messages as well it is appropriate that we use a combined monad with both properties. The construction of a combined monad is not always trivial, so it would be useful to have a systematic way of combining monads. Unfortunately, a method does not exist which combines any monad M with any monad N. However, it is conceivable that we may have a library of useful primitive monads, and for each one a technique for combining it with others. Perhaps our library could be divided into collections of monads which can be combined using the same recipe.

In this paper we will look at the class of monads containing trees, lists, bags and sets and consider how they may be combined with others. Lists will be studied first, and it will be shown that the list monad L can be combined

*Authors' address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, United Kingdom. Electronic mail: {`gnik, wadler`}`@dcs.glasgow.ac.uk`.

To appear in *Functional Programming, Glasgow 1992*, Ayr, Scotland, J. Launchbury and P.M. Sansom, editors, Springer Verlag, Workshops in Computing, 1993.

with any monad M to form the combined monad ML[1]. The list monad can be used to model a form of nondeterminism: instead of returning one value, a list of values are returned. The combined monad ML may, for instance, be state transformers of lists. Or ML may be sets of lists, perhaps a useful structure for a query system. Another practical use of this technique is to combine a parsing monad [8] with an exception monad (described in Section 2). A parsing monad has the type:

$$\textbf{type}\ Parse\ a\ =\ State \rightarrow L\ (a, State).$$

A parse may fail producing an empty list or succeed producing a list of possible parses. Combining the list part of this type with the exception monad will enable us to return either an error message or a list of possible parses.

All the work with lists is generalised in Section 7 to the tree, bag and set monads. It turns out that the monads which combine with bags and sets must hold certain properties, these are explained.

2 Combining monads

First let's look at two of the most commonly used monads and consider how they may be combined with others. Most of the notation used will be consistent with the Haskell language [4]. The monad of state transformers St has type:

$$\textbf{type}\ St\ a\ =\ State \rightarrow (a, State),$$

this says that a state transformer on type a is a function from a state to a pair consisting of a value of type a and a new state. For example, we may use a state in a type checker to hold the current substitution. The monad of exceptions Ex has the form:

$$\textbf{data}\ Ex\ a = Return\ a \mid Raise\ Exception,$$

this is a sum type with constructors $Return$ and $Raise$. This monad may be used to structure an interpreter so that it either returns a value or an error message.

Now we may combine the St monad with any other monad M:

$$\textbf{type}\ MSt\ a = State \rightarrow M\ (a, State),$$

thus forming MSt, note that M is textually substituted here. We may also combine the Ex monad with any other by defining:

$$\textbf{type}\ MEx\ a = M\ (Ex\ a)$$

There are now two distinct approaches to combining Ex with St, either:

$$\textbf{type}\ StEx\ a = State \rightarrow (Ex\ a, State)$$

or

$$\textbf{type}\ ExSt\ a = State \rightarrow Ex\ (a, State)$$

[1] The notation ML represents the combination of monad M with monad L and has nothing to do with the language Standard ML.

these have entirely different meanings. In the first model, an expression that raises an exception also returns a state. This monad may be thought of as modelling the language Standard ML, where when an exception is raised the state survives. In the second model, if an expression raises an exception then the state is not returned.

3 Monads and their comprehension

For our purposes, a monad consists of a type constructor M, an endofunctor $map^M :: (a \to b) \to M\ a \to M\ b$ and two natural transformations $unit^M :: a \to M\ a$ and $join^M :: M(M\ a) \to M\ a$, satisfying:

$$
\begin{array}{lll}
\text{(M-1)} & map^M\ id & = & id, \\
\text{(M-2)} & map^M\ (f \cdot g) & = & map^M\ f \cdot map^M\ g, \\
\\
\text{(M-3)} & map^M\ f \cdot unit^M & = & unit^M \cdot f, \\
\text{(M-4)} & map^M\ f \cdot join^M & = & join^M \cdot map^M\ (map^M f), \\
\\
\text{(M-5)} & join^M \cdot unit^M & = & id, \\
\text{(M-6)} & join^M \cdot map^M\ unit^M & = & id, \\
\text{(M-7)} & join^M \cdot map^M\ join^M & = & join^M \cdot join^M.
\end{array}
$$

Laws (M-1) and (M-2) show that map^M is an endofunctor, while (M-3) shows that $unit^M$ is a natural transformation and similarly (M-4) shows that $join^M$ is a natural transformation. Laws (M-5) and (M-6) are the left and right unitary identities, law (M-7) is the associative property for monads.

Wadler [9] describes a comprehension for monads which is sometimes more concise and clear than the above monad functions. The comprehension has the same form as list comprehension used in functional languages like Haskell. The difference is that the structures may not be just lists, hence we tag the square brackets with a letter indicating which monad is being referred to. A monad comprehension takes the form:

$$[\, t \mid q \,]^M,$$

where t is a term and q is a qualifier. A qualifier is either empty (Λ), a generator ($x \leftarrow u$), or the composition of two qualifiers (p, q).

Monad comprehension can be defined in terms of the monad functions as follows:

$$
\begin{array}{llll}
\text{(mc-1)} & [\, t \mid \Lambda \,]^M & = & unit^M\ t, \\
\text{(mc-2)} & [\, t \mid x \leftarrow u \,]^M & = & map^M\ (\lambda x.\ t)\ u, \\
\text{(mc-3)} & [\, t \mid p, q \,]^M & = & join^M\ [\, [\, t \mid q \,]^M \mid p \,]^M.
\end{array}
$$

There are two important facts about the qualifiers, firstly that their order matters. For instance, if we define two Cartesian product functions where the qualifiers are in a different order:

$$
\begin{array}{lll}
a \times b & = & [\, (x, y) \mid x \leftarrow a, y \leftarrow b \,], \\
a \times' b & = & [\, (x, y) \mid y \leftarrow b, x \leftarrow a \,],
\end{array}
$$

where a and b range over lists. Now both forms of Cartesian product will be applied to the same arguments:

$$
\begin{array}{lcl}
[1,2] \times [3,4] & = & [(1,3),(1,4),(2,3),(2,4)], \\
[1,2] \times' [3,4] & = & [(1,3),(2,3),(1,4),(2,4)].
\end{array}
$$

Hence, interchanging qualifiers changes the meaning that the comprehension denotes. We will see later on that if the order of qualifiers in a comprehension can be changed without changing the value that it denotes, then the monad is commutative. So, in the above case we have shown that the list monad is not commutative.

The second fact about qualifiers is that the variable part of a generator may be used in a later qualifier, but not in an earlier qualifier. For example,

$$[\, x \mid xs \leftarrow a, x \leftarrow xs \,]^M$$

the variable xs is bound by the first generator and used by the next (note, this is in fact the definition of $join^M\ a$). Reversing the qualifiers in the above comprehension will make xs a free variable.

The laws (M-5), (M-6) and (M-7) can be expressed in the comprehension:

$$
\begin{array}{llcl}
\text{(M-5)} & [\, t \mid \Lambda, q \,] & = & [\, t \mid q \,], \\
\text{(M-6)} & [\, t \mid q, \Lambda \,] & = & [\, t \mid q \,], \\
\text{(M-7)} & [\, t \mid (p,q), r \,] & = & [\, p, (q,r) \,].
\end{array}
$$

From (mc-1), (mc-2) and (mc-3) we can derive the following which are useful for manipulating the comprehension:

$$
\begin{array}{llcl}
\text{(mc-4)} & [\, f\ t \mid q \,]^M & = & map^M\ f\ [\, t \mid q \,]^M, \\
\text{(mc-5)} & [\, x \mid x \leftarrow u \,]^M & = & u, \\
\text{(mc-6)} & [\, t \mid p, x \leftarrow [\, u \mid q \,]^M, r \,]^M & = & [\, t_x^u \mid p, q, r_x^u \,]^M,
\end{array}
$$

where the notation t_x^u means replace all free occurrences of x in t with u.

4 Composing the list monad with any other

4.1 Lists

Lists are one of the most frequently used constructs by functional programmers, and are built in to most functional languages. Lists will be denoted with the letter L. Everything we want to do with lists here can be defined in terms of the empty list and four functions. The empty list will be denoted $nil^L :: L\ a$ (we may sometimes use the notation $[\,]$). The append operation $(+\!\!+) :: L\ a \rightarrow L\ a \rightarrow L\ a$ fuses two lists:

$$[\, a_1, a_2, \ldots, a_m \,] +\!\!+ [\, b_1, b_2, \ldots, b_n \,] = [\, a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n \,].$$

Append $(+\!\!+)$ is associative and has nil^L as identity. Hence $(L, +\!\!+, nil^L)$ forms a monoid (in fact it is the free monoid).

The function $unit^L :: a \rightarrow L\ a$ takes an element and gives back the singleton list, thus:

$$unit^L\ a = [a].$$

The function map^L applies a function f to every element of a list:

$$\begin{array}{lcl} map^L & :: & (a \to b) \to L\ a \to L\ b \\ map^L\ f\ nil^L & = & nil^L \\ map^L\ f\ (unit^L\ a) & = & unit^L\ (f\ a) \\ map^L\ f\ (as +\!\!+\ bs) & = & (map^L\ f\ as) +\!\!+ (map^L\ f\ bs). \end{array}$$

The function $fold^L$ takes three things: an associative operator $\oplus$, an identity element e and a list:

$$\begin{array}{lcl} fold^L & :: & (a \to a \to a) \to a \to L\ a \to a \\ fold^L\ (\oplus)\ e\ nil^L & = & e \\ fold^L\ (\oplus)\ e\ (unit^L\ a) & = & a \\ fold^L\ (\oplus)\ e\ (as +\!\!+\ bs) & = & (fold^L\ (\oplus)\ e\ as) \oplus (fold^L\ (\oplus)\ e\ bs). \end{array}$$

Hence,

$$fold^L\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n.$$

The function $join^L$ appends lists of lists (also known as concatenation) and can be defined in terms of $fold^L$:

$$\begin{array}{lcl} join^L & :: & L(L\ a) \to L\ a \\ join^L & = & fold^L\ (+\!\!+)\ nil^L. \end{array}$$

A useful property of $join^L$ is that it distributes through append, that is:

$$join^L\ (a +\!\!+\ b) = join^L\ a +\!\!+\ join^L\ b.$$

It is not difficult to show that L forms a monad by using the definitions given to check that the seven monad laws are satisfied.

Homomorphism lemma
A function h that satisfies the equations:

$$\begin{array}{lcl} h\ nil^L & = & e, \\ h\ (unit^L\ a) & = & g\ a, \\ h\ (as +\!\!+\ bs) & = & h\ as \oplus h\ bs, \end{array}$$

is a homomorphism if and only if $h = fold^L\ (\oplus)\ e \cdot map^L\ g$ for some function g and operator $\oplus$ with identity e, such that $g = h \cdot unit^L$. This definition can be found in Bird's paper [3], a more general definition can be found in Spivey's paper [7].

4.2 An ML monoid

We define a special operation denoted $\otimes$ which is a kind of Cartesian product,

$$\begin{array}{lcl} (\otimes) & :: & ML\ a \to ML\ a \to ML\ a \\ a \otimes b & = & [\,x +\!\!+\ y \mid x \leftarrow a, y \leftarrow b\,]^M. \end{array}$$

The identity element for $\otimes$ is $unit^M\ nil^L$. Using our comprehension, $\otimes$ can be given in terms of the monad functions.

$$
\begin{aligned}
& a \otimes b \\
& \quad = [\, x +\!\!+\, y \mid x \leftarrow a, y \leftarrow b\,]^M && \text{(by definition)} \\
& \quad = join^M \; [\,[\, x +\!\!+\, y \mid y \leftarrow b\,]^M \mid x \leftarrow a\,]^M && \text{(mc-3)} \\
& \quad = join^M \; (map^M \; (\lambda x.\; [\, x +\!\!+\, y \mid y \leftarrow b\,]^M)\; a) && \text{(mc-2)} \\
& \quad = join^M \; (map^M \; (\lambda x.\; (map^M \; (\lambda y.\; x +\!\!+\, y)\; b))\; a) && \text{(mc-2)}
\end{aligned}
$$

Proposition
$(ML, \otimes, e)$ forms a monoid.

Properties of $\otimes$
For each monad function there will be a property relating it to $\otimes$, these will be useful for later proofs.

$$(\otimes\text{-}1) \quad unit^M \; a \otimes unit^M \; b \quad = \quad unit^M \; (a +\!\!+\, b).$$

A list homomorphism h must distribute through append, that is:

$$h \; (a +\!\!+\, b) = h \; a +\!\!+\, h \; b.$$

The monad functions $map^L \; f$ and $join^L$ are list homomorphisms, however $unit^L$ is not. Using the above homomorphism lemma every such homomorphism can be expressed as the following composition $h = join^L \cdot map^L \; g$ where $g \; a = h \; [a]$. If f is a list homomorphism then:

$$(\otimes\text{-}2) \quad map^M \; f \; a \otimes map^M \; f \; b \quad = \quad map^M \; f \; (a \otimes b).$$

For $(\otimes$-3$)$ another kind of homomorphism must first be defined:

$$h \; (a \otimes b) = h \; a \otimes' h \; b,$$

where $\otimes'$ is defined:

$$a \otimes' b \quad = \quad [\, x \otimes y \mid x \leftarrow a, y \leftarrow b\,]^M.$$

Now if f is such a homomorphism then:

$$(\otimes\text{-}3) \quad join^M (f \; a) \otimes join^M (f \; b) \quad = \quad join^M \; (f \; (a \otimes b)).$$

4.3 Folding with $\otimes$

As $(ML, \otimes, e)$ forms a monoid, we are at liberty to define a $fold^L$ operation in terms of these, it will be called $prod$.

$$
\begin{aligned}
prod &\;::\; L(ML \; a) \rightarrow ML \; a \\
prod &\;=\; fold^L \; (\otimes) \; e.
\end{aligned}
$$

This function is related to the functions for monads in the following ways:

Properties of $prod$

$$
\begin{aligned}
&\text{(prod-1)} \; prod \cdot unit^L && = \; id, \\
&\text{(prod-2)} \; prod \cdot map^L \; unit^M && = \; unit^M \cdot join^L, \\
&\text{(prod-3)} \; prod \cdot map^L \; (map^M \; (map^L \; f)) && = \; map^M \; (map^L \; f) \cdot prod, \\
&\text{(prod-4)} \; prod \cdot map^L \; (join^M \cdot map^M \; prod) && = \; join^M \cdot map^M \; prod \cdot prod, \\
&\text{(prod-5)} \; prod \cdot map^L \; prod && = \; prod \cdot join^L.
\end{aligned}
$$

Interestingly, if (prod-1) and (prod-5) hold then $(ML, prod)$ forms an L-algebra as described in Barr and Wells [2]. We can prove the last four properties of $prod$ by using the homomorphism lemma again. To prove that $h = prod \cdot map^L\ g$ for some given h we just have to show that h satisfies the three equations for a homomorphism.

4.4 The ML monad

Now given any monad M we can define a new monad ML:

$$\begin{aligned} unit^{ML} &= unit^M \cdot unit^L, \\ map^{ML}\ f &= map^M\ (map^L\ f), \\ join^{ML} &= join^M \cdot map^M\ prod. \end{aligned}$$

The fact that M and L are monads together with the properties of $prod$ are just what we need to prove that ML forms a monad.

5 The distributive laws for monads

The definition of $join^{ML}$ defined previously was given with no justification of where it came from. In fact, $join^{ML}$ was initially defined in terms of a natural transformation $cp :: L(M\ a) \rightarrow ML\ a$ which is a Cartesian product.

$$cp = prod \cdot map^L\ (map^M\ unit^L).$$

The function $join^{ML}$ can now be defined terms of this,

$$join^{ML} = map^M\ join^L \cdot join^M \cdot map^M\ cp.$$

This can be shown to be equivalent to the previous definition of $join^{ML}$.

Properties of cp

$$\begin{array}{lllll} \text{(cp-1)} & cp \cdot unit^L & = & map^M\ unit^L, \\ \text{(cp-2)} & cp \cdot map^L\ unit^M & = & unit^M, \\ \text{(cp-3)} & cp \cdot map^L\ (map^M\ f) & = & map^M\ (map^L\ f) \cdot cp, \\ \text{(cp-4)} & cp \cdot map^L\ join^M & = & join^M \cdot map^M\ cp \cdot cp, \\ \text{(cp-5)} & cp \cdot join^L & = & map^M\ join^L \cdot cp \cdot map^L\ cp. \end{array}$$

The (cp-3) property tells us that cp is a natural transformation, the others are known as the distributive laws and are discussed in [1]. We can prove these properties of cp by using the $prod$ properties. Conversely, we can prove the properties of $prod$ using the properties of cp and defining $prod$ as:

$$prod = map^M\ join^L \cdot cp.$$

6 Monad properties

The monad comprehension provides a meaningful way of describing monad properties as we showed before with the associative property for monads. In Section 3 we mentioned that interchanging qualifiers could be done with a commutative monad, thus commutativity is represented:

$$\text{(Commutativity)} \qquad [\, t \mid x \leftarrow u, y \leftarrow v\,]^M = [\, t \mid y \leftarrow v, x \leftarrow u\,]^M,$$

where x is not free in v, and y is not free in u.

We say that a monad is idempotent when:

$$\text{(Idempotence)} \qquad [\, t \mid x \leftarrow u, y \leftarrow u\,]^M = [\, t^x_y \mid x \leftarrow u\,]^M.$$

Category theorists, for example, Barr and Wells [1] use the same name for a different concept: they define a monad to be idempotent when $join^M$ is an isomorphism. With our definition the exception monad is idempotent, whereas it is not with the category theorist's definition. Bag and set monads are commutative but not idempotent. The above properties will be used later on.

7 From the Boom hierarchy to a monad hierarchy

Similar to lists, we can define bags in terms of: nil^B, $unit^B$, $+\!\!+^B$, map^B and $fold^B$. The function $fold^B$ is the same as $fold^L$ with the proviso that its operator is associative and symmetric, and the bag union operator $+\!\!+^B$ is also associative and symmetric. Similarly, for sets the operators are associative, symmetric and idempotent. On the other end of the scale, for trees the operators need not even be associative, and need not have *nil* as an identity. These structures are represented in the following hierarchy, known as the Boom hierarchy, see [5].

	Trees	Lists	Bags	Sets
Associative	$\times$	$\surd$	$\surd$	$\surd$
Symmetric	$\times$	$\times$	$\surd$	$\surd$
Idempotent	$\times$	$\times$	$\times$	$\surd$

These structures are all finite and all form free algebraic structures. For instance, bags form the free symmetric monoid.

In the following sections we will look at the conditions needed to form MB and MS. Trivially, the combined monad MT can be formed with no extra conditions, in the same way as we formed ML. We will discover that there are a hierarchy of conditions needed on M in order to combine it with bags and sets.

7.1 Bags

To form the monad MB, where B is the bag monad and M is any monad, we must first define the Cartesian product operator on MB.

$$\begin{array}{lcl} (\otimes^B) & :: & M(B\ a) \to M(B\ a) \to M(B\ a) \\ a \otimes^B b & = & [\, x +\!\!+^B y \mid x \leftarrow a, y \leftarrow b\,]^M. \end{array}$$

To define,

$$\begin{array}{lcl} prod^B & :: & B(MB\ a) \to MB\ a \\ prod^B & = & fold^B\ (\otimes^B)\ e, \end{array}$$

where $e = unit^M\ nil^B$, the operator $\otimes^B$ must be associative and symmetric, to be used with $fold^B$. Its associativity proof is the same as the proof for $\otimes$.

Proposition
The operator $\otimes^B$ is symmetric, if M is a commutative monad.

Proof

$$\begin{array}{ll} a \otimes^B b & \\ \quad = [\, x +\!\!+^B y \mid x \leftarrow a, y \leftarrow b\,]^M & \text{(by definition)} \\ \quad = [\, y +\!\!+^B x \mid x \leftarrow a, y \leftarrow b\,]^M & (+\!\!+^B \text{ symmetric}) \\ \quad = [\, y +\!\!+^B x \mid y \leftarrow b, x \leftarrow a\,]^M & (M \text{ commutative}) \\ \quad = b \otimes^B a & \text{(by definition)} \end{array}$$

Now we can define the monad $(map^{MB}, unit^{MB}, join^{MB})$ as was done for lists, using $prod^B$ given above instead of $prod$. Therefore, we can form the combined monad MB so long as M is a commutative monad.

7.2 Sets

Using the same recipe for a third time, the Cartesian operator will be defined:

$$\begin{array}{lcl} (\otimes^S) & :: & M(S\ a) \to M(S\ a) \to M(S\ a) \\ a \otimes^S b & = & [\, x +\!\!+^S y \mid x \leftarrow a, y \leftarrow b\,]^M. \end{array}$$

where the operator $+\!\!+^S$ will be set union, but we will keep to the same notation. Now to use $fold^S$, the operator $\otimes^S$ will have to be associative, symmetric and idempotent. Proof of associativity and symmetry are the same as for lists and bags.

Proposition
The operator $\otimes^S$ is idempotent, if M is an idempotent monad.

Proof

$$\begin{array}{ll} a \otimes^S a & \\ \quad = [\, x +\!\!+^S y \mid x \leftarrow a, y \leftarrow a\,]^M & \text{(by definition)} \\ \quad = [\, x +\!\!+^S x \mid x \leftarrow a\,]^M & (M \text{ idempotent}) \\ \quad = [\, x \mid x \leftarrow a\,]^M & (+\!\!+^S \text{ idempotent}) \\ \quad = a & \text{(mc-5)} \end{array}$$

Hence, from this the monad MS can be formed so long as M is a commutative, idempotent monad.

Acknowledgements

Thanks to the Science and Engineering Research Council for funding the first author. Many thanks also to the referees: Mark Jones, André Santos and Simon Thompson for their constructive comments.

References

[1] M. Barr and C. Wells, *Toposes, Triples, and Theories.* Springer Verlag, 1985.

[2] M. Barr and C. Wells, *Category Theory for Computing Science.* Prentice Hall, 1990.

[3] R. Bird, An Introduction to the Theory of Lists. In *Logic of Programming and Calculi of Discrete Design*, Springer Verlag, 1987.

[4] P. Hudak, S. Peyton Jones and P. Wadler, editors, Report on the functional programming language Haskell, Version 1.2, *SIGPLAN Notices*, Vol. 27, No. 5, May 1992.

[5] L. Meertens, Algorithmics - towards programming as a mathematical activity. In J. deBakker, M. Hazewinkel and L. Lenstra, editors, *CWI Symposium on Mathematics and Computer Science*, Vol. 1, CWI monographs, North Holland, 1986.

[6] E. Moggi, Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, June 1989.

[7] M. Spivey, A Categorical Approach to the Theory of Lists. In *Mathematics of Program Construction*, LNCS 375, Springer Verlag, 1989.

[8] P. Wadler, How to Replace Failure by a List of Successes. In *Proceedings of Functional Programming and Computer Architecture*, Springer-Verlag, LNCS 201, September 1985.

[9] P. Wadler, Comprehending Monads. In *ACM Conference of Lisp and Functional Programming*, June 1990.

[10] P. Wadler, The essence of functional programming. In *19'th ACM Symposium on Principles of Programming Languages*, January 1992.