Monads in Common Lisp

Jim Newton

November 2016

Abstract

In this article we explain monads so they can be understood to the Lisp programmer. We base the explanation on a very clean explanation presented in the Scala programming language. We then proceed to represent the concepts using mostly simple Common Lisp concepts. We do not attempt to justify the motivation behind the definitions, and we do not attempt to give any examples of applications. Most notably, we do not attempt to explain the connection monads have to modeling side effects.

1 Scala Perspective

1.1 Scala for the Lisper in two paragraphs

Although the notation and syntax are quite different, a Lisp programmer should be able very quickly to understand a simple Scala program. Just as does Common Lisp [5], Scala [3, 2] comes equipped with a large library of accessories which of course have to be learned and pose difficulties to the beginner. Nevertheless, just as one can approach Common Lisp as a language whose evaluation model [1] is based on the *untyped lambda calculus* and extended by lots of libraries, Scala can be viewed as an extension of the *typed lambda calculus*[4] augmented by a large library.

The lisp programmer may think of a Scala *trait* as a class which can only be used for mix-in; it cannot be instantiated. A trait may provide method implementations, or it may provide specifications declarations for methods which have no implementation but do have type signatures. In such a case Scala requires that any instantiatable class inheriting from such a trait provide an implementation for any such specified method.

1.2 The monad in Scala

In the Scala programming language a *monad* is a parameterized type (represented as a trait) which implements a particular interface, consisting of a `flatMap` function and a `unit` function. Moreover, the interface obeys three monadic laws.

The interface for the monad `M[T]` looks like this.

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

`M` is a monad if, given the following:

- types `T`, `U`, and `V`,
- a function `f: T=>M[U]`,
- a function `g: U=>M[V]`,
- an object `x` of type `T`, and
- an object `m` of type `M[T]`,

the three laws are as follows.

Left unit: `f(x) == unit(x) flatMap f`
and is an object of type `M[U]`.

Right unit: `m == m flatMap unit`

Associativity: `m flatMap f flatMap g`
`== m flatMap (x => f(x) flatMap g)`
and is an object of type `M[V]`.

The syntax might seem bizarre to the Lisp programmer. In Scala certain binary functions can be written using either prefix notation or infix notation. In prefix notation, parentheses and commas are required, as `flatMap(f(x),g)`. In infix notation the same expression may be written without the commas, as `(f(x) flatMap g)`, in which case the outer parenthesis may be omitted according to the precedence rules.

Another bit of Scala syntax: `x => f(x) flatMap g` is the anonymous function syntax. Although there is no such `lambda` syntax in Scala, you can think of it as $\lambda x.(flatMap\ (f\ x)\ g)$

2 Lisp Perspective

2.1 Lisp types and notation

In Common Lisp a type is simply a set of values. The set may be empty, finite, or infinite. The behavior of a type is determined by the functions which are defined to accept or return values of this type. Some Common Lisp types have notation, and some do not. For example *NIL* denotes the empty set; *T* denotes the universal set of all values; *fixnum* denotes the set of integers which are not bignums. However, the set of lists of strings does not have a *standardized* type

name, and the Common Lisp specification does not directly define a notation to specify such a type.

The set of functions is denoted *function*. The set of binary functions which accept a *fixnum* and a *float* and return a *string* is denoted *(function (fixnum float) string)*.

In type theory literature it is more customary to denote function types using so-called *arrow* arrow notation. The Common Lisp type will hereinafter be referred to as $(fixnum, float) \rightarrow string$. The type of unary functions may be denoted with or without the parenthesis; the following are equivalent $(fixnum) \rightarrow string$ and $fixnum \rightarrow string$. And the type 0-ary functions returning an object of type *string* is denoted $() \rightarrow string$.

Some types are sets of containers such as *list* of $fixnum$, or *vector* of $function$ mapping $fixnum$ to *string*. Such types have no standard notation in Common Lisp, but we will refer to such types using so-called *bracket* notation such as $list[fixnum]$ or $vector[fixnum \rightarrow string]$.

It is customary in type theory literature to use T as a variable representing a type. However, we will refrain from using T to represent such a variable lest it be confused with the Common Lisp type universal type, also denoted T. Instead, we'll typically use letters such as S, U, and V to represent type variables.

To denote that an object *has a type* or *is an element of a type* we use the *colon* notation such as: $i{:}fixnum$ denotes i is of type $fixnum$, and $f{:}fixnum \rightarrow string$ denotes f is a function mapping $fixnum$ to *string*.

2.2 Uniform type monad

First we'll discuss the *uniform type monad* or *simple monad* over a type S, denoted $M[S]$ or simply M, is a container type but which is also associated with two well behaved auxiliary functions: a unary function called its $unit{:}S \rightarrow M$ function and a binary function called its $bind{:}(S \rightarrow M, M) \rightarrow M$ function. The *laws* which *unit* and *bind* must adhere to are as follows.

Left unit: $(f\ x)$ is equivalent to $(bind\ f\ (unit\ x))$ whenever $f{:}S \rightarrow S$ and $x{:}S$

Right unit: m is equivalent to $(bind\ unit\ m)$ whenever $m{:}M$

Associativity: $(bind\ g\ (bind\ f\ m))$ is equivalent to $(bind\ (\lambda x\ (bind\ g\ (f\ x)))\ m)$ whenever $m{:}M$, $f{:}S \rightarrow M$, and $g{:}S \rightarrow M$

These laws may seem somewhat difficult to grasp so an example may help to clarify. Consider the Common Lisp container type *list* of *fixnum*, which we'll denote as $list[fixnum]$. This container type is a simple monad if we can identify an appropriate *unit* and *bind* functions of the correct types and satisfying the three monadic laws.

In fact if we take the `list` function as *unit* and the *mapcan* function as the *bind* we can verify that the types conform and the laws are satisfied. First, consider the types.

- S — $fixnum$

- $M[S]$ — $list[fixnum]$
- $unit{:}S{\rightarrow}M$ — $list{:}fixnum{\rightarrow}list[fixnum]$
- $bind{:}(S{\rightarrow}M, M){\rightarrow}M$ — $mapcan : (fixnum{\rightarrow}list[fixnum], list[fixnum]){\rightarrow}list[fixnum]$

To see this in action lets define a few lisp objects, then test the laws. m is defined as a $list[fixnum]$. x is defined as a $fixnum$. f and g are both defines as $fixnum{\rightarrow}list[fixnum]$.

```
(defvar m '(1 3 5 -7 -11))
(defvar x -12)
(defun f (i) (list (* i 10)))
(defun g (i) (list (+ i 1)))
(defun unit (i) (list i))
(defun bind (fun data) (mapcan fun data))
```

The *Left unit* law says that $(f\ x)$ is equivalent to $(bind\ f\ (unit\ x))$, and we can see that is verified in the following test.

```
CL-USER> (f x)
(-11)
CL-USER> (bind #'f (unit x))
(-11)
CL-USER> (equal (f x)
                (bind #'f (unit x)))
T
```

The *right unit* law says that m is equivalent to $(bind\ unit\ m)$, and we can see that is verified in the following test.

```
CL-USER> m
(1 3 5 -7 -11)
CL-USER> (bind #'unit m)
(1 3 5 -7 -11)
CL-USER> (equal m
                (bind #'unit m))
T
```

The *Associativity* law says $(bind\ g\ (bind\ f\ m))$ is equivalent to $(bind\ (\lambda x\ (bind\ g\ (f\ x)))\ m)$, and we can see that is verified in the following test.

```
CL-USER> (bind #'g (bind #'f m))
(11 31 51 -69 -109)
CL-USER> (bind (lambda (x) (bind #'g (f x))) m)
(11 31 51 -69 -109)
CL-USER> (equal (bind #'g (bind #'f m))
                (bind (lambda (x) (bind #'g (f x))) m))
T
```

2.3 Arbitrary type monad

In the above example of the simple type monad we looked at the list containing of fixnums. However, this concept can be made more general to be more permissive with regard to the types of the objects in the container. Consider the following example.

```
(defun f (num)
 (declare (type fixnum num))
 (list (format nil "~D" num)))
```

```
(defun g (str)
 (declare (type string str))
 (list (float (with-input-from-string (stream str)
                (read stream)))))
```

Notice that f:$fixnum \rightarrow list[string]$ and g:$string \rightarrow list[float]$.

```
CL-USER> (f 3)
("3")
CL-USER> (g "3")
(3.0)
```

The functions, `f` and `g` have been constructed so that they no longer compose with each other as in the previous example of the simple monad. Nevertheless, they maintain their behavior with respect to their interaction with `mapcan`.

```
CL-USER> (mapcan #'g (f 3))
(3.0)
```

Given any list of fixnums we can produce the corresponding list of strings with `mapcan`.

```
CL-USER> (mapcan #'f (list 1 3 5 -7 -11))
("1" "3" "5" "-7" "-11")
```

And given a such list of strings we can further produce the corresponding list of floats with `mapcan`.

```
CL-USER> (mapcan #'g (list "1" "3" "5" "-7" "-11"))
(1.0 3.0 5.0 -7.0 -11.0)
```

Notice that the three monadic laws are still satisfied.

The *Left unit* law says that $(f\ x)$ is equivalent to $(bind\ f\ (unit\ x))$, and we can see that is verified in the following test.

```
CL-USER> (defvar i 12)
I
CL-USER> (f i)
("12")
CL-USER> (mapcan #'f (list i))
("12")
CL-USER> (equal (f i)
                (mapcan #'f (list i)))
T
```

The *right unit* law is unchanged from before. It says that m is equivalent to $(bind\ unit\ m)$, and we can see that is verified in the following test.

```
CL-USER> (defvar m (list 1 3 5 -7 -11))
M
CL-USER> m
(1 3 5 -7 -11)
CL-USER> (mapcan #'list m)
(1 3 5 -7 -11)
CL-USER (equal m
               (mapcan #'list m))
T
```

The *Associativity* law says $(bind\ g\ (bind\ f\ m))$ is equivalent to $(bind\ (\lambda x\ (bind\ g\ (f\ x)))\ m)$, and we can see that is verified in the following test.

```
CL-USER> (defvar m (list 1 3 5 -7 -11))
M
```

```
CL-USER> (mapcan #'g (mapcan #'f m))
(1.0 3.0 5.0 -7.0 -11.0)
CL-USER> (mapcan #'(lambda (i) (mapcan #'g (f i))) m)
(1.0 3.0 5.0 -7.0 -11.0)
CL-USER> (equal (mapcan #'g (mapcan #'f m))
                (mapcan #'(lambda (i)
                            (mapcan #'g (f i)))
                        m))
T
```

3 Conclusion

Although monads are generally seen as obtuse, in this article we have presented a very simple description in terms of straightforward Common Lisp mechanism. We have not made any attempt to justify any of the axioms and made no effort to give examples of their applications.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] P. Chiusano and R. Bjarnason. *Functional Programming in Scala*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[3] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[4] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[5] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.