Introduction to monads

Mayer Goldberg

May 20, 2013

Contents

1 Preliminaries 1

2 Threading a computation 1

3 Abstract monadic operations 2

4 Defining the monadic operators in the λ-calculus 3

1 Preliminaries

The **ordered n-tuple** $\langle A_1, \ldots, A_n \rangle$ is defined as follows:

$$\langle A_1, \ldots, A_n \rangle \quad = \quad \lambda x.xA_1 \cdots A_n$$

The **selector** σ_j^n takes n arguments and returns the j-th:

$$\sigma_j^n \quad = \quad \lambda x_1 \cdots x_n.x_j$$

The **projection** π_j^n takes an ordered n-tuple and returns it's j-th tuple:

$$\pi_j^n \quad = \quad \lambda v.v\sigma_j^n$$

2 Threading a computation

For the sake of this discussion, we shall use the term *action* to refer to a computation that involves side effects, as opposed to *functions*, which have no side effects.

We have already seen how side-effects can be removed by "completing" the domain and the range of the actions that make up the computation, so that they become functions in the mathematical sense. We shall refer to [the Cartesian product of] the added values in the domain and range as a *store*. This store is *passed around* between actions. We descibe this relation between the actions and the store by saying that the store *threads the actions*, or *threads the computation*.

Extending the domain and ranges of each action complicates the API, and makes programming unnecessarily verbose and complex.

We can hide the complexity by *Currying* on the additional arguments that extend the domain and range of the actions. Currying doesn't eliminate these arguments, but puts them "out of sight". The next step is to define algebraic operations over these actions, so that actions can be composed with respect to the store that threads them: If two actions are parameterized by a store, then the composition of these actions is also parameterized by a store, so that the store returned by the first computation gets passed to the second.

3 Abstract monadic operations

A **monad** is an abstract computational object that maps a store to a pair consisting of some *value* and a [possibly new] store:

$$\textbf{Monad} : \mathit{Store} \to \big\langle \mathit{Value}, \mathit{Store}' \big\rangle$$

An **action** is an abstact computational object that maps a value to a monad:

$$\textbf{Action} : \mathit{Value} \to \underbrace{\mathit{Store} \to \big\langle \mathit{Value}', \mathit{Store}' \big\rangle}_{\text{some } \mathit{monad}}$$

The simplest action does nothing with either the value of the store, returning them exactly as they were received. This operation corresponds to the identity function, and is called **Unit**. Faithful to the above formualtion of an action, we characterize **Unit** as follows:

$$\textbf{Unit} : \mathit{Value} \to \mathit{Store} \to \langle \mathit{Value}, \mathit{Store} \rangle$$

It is useful to define the **Lift** operator, which maps functions to actions:

$$\textbf{Lift} \;=\; \underbrace{(\mathit{Value} \to \mathit{Value}')}_{\substack{\text{some}\\ \text{function}}} \to \underbrace{\mathit{Value} \to \mathit{Store} \to \langle \overbrace{\mathit{Value}'}^{((\mathit{Value} \to \mathit{Value}')\ \mathit{Value}')\ =}, \mathit{Store}\rangle}_{\text{action that computes the given function}}$$

Note that the **Unit** action is simply a lifted identity function.

Actions can be added to the end of a monad, resulting in a new, larger monad. If you think of a monad as a piping system, then an action becomes yet-another pipe that can be attached onto the end of the piping system. The result is a new, longer piping system. The algebraic operation of attaching a pipe onto a piping system is known as *pipe* or *bind*:

$$\textbf{Pipe} : \langle \textbf{Monad}, \textbf{Action} \rangle \to \textbf{Monad}'$$

The **Pipe** operation is how sequencing is performed using monads, and therefore, it corresponds to, and can be used to provide a semantics for **begin** $\cdots$ **end** blocks in structured programming languages.

In this section, and in the context of abstract monadic operations, when we mention *pairs*, we do not have in mind a specific implementation of pairs, either in some set-theoretic or computational formalism, or in some programming language. Rather, these pairs are to be understood as an abstract, ordered "joining" of two things: Plato's `CONS`, if you will. Later, when we implement monads in some formalism or programming language, we shall have the freedom to do so in a way that is natural for that domain.

4 Defining the monadic operators in the λ-calculus

Monads are abstract computational objects that are parameterized by a store. A computation is encoded starting from a very simple monad, which serves to initialize the store, and actions that have been added onto it, in a pipeline fashion, to form larger monads. The **Pipe** operation is designed so that the store that threads each action shall thread the entire combined monad.

Defining monads in the λ-calculus (and in any programming language) requires that we define the **Pipe**, **Unit**, and **Lift** operations, which are general, and can be used for many purposes. We then need to define actions that are specific to the computation we wish to perform.

Defining the **Unit** is straightforward, and follows from the commonly-used representation of ordered-pairs in the λ-calculus, and the abstract definition of **Unit** in the previous section:

$$\mathbf{Unit} \quad = \quad \lambda vsx.xvs \;\equiv\; \langle \llcorner\!\lrcorner, \llcorner\!\lrcorner \rangle$$

The definition of **Pipe** is trickier. **Pipe** takes a monad and an action, and returns another monad, which means that it returns a function of a store. Our definition shall therefore start with $\lambda mas.\cdots$, where m is a monad, a is an action, s is a store. We now need to think about what the new monad actually does.

The new monad should apply the old monad to the store, getting a pair of a value and a new store. The value should be passed onto the action, followed by the store, and the resulting monad is the result of the computation. Applying the monad to the store is given by (ms). Since the result is a pair, we apply it to $\lambda v's'.\cdots$, where v' stands for the resulting value, and s' stands for the resulting store. What shall we do with v', s' ? Looking at the abstract definition for **Action**, in the previous section, we see that it expects a value and a store, so what we want to return is $(av's')$. Putting all this together, we get a definition for **Pipe** as follows: $\lambda mas.ms(\lambda v's'.av's')$. Notice that we have two opportunities for η-reduction, giving us the final form of:

$$\mathbf{Pipe} \quad = \quad \lambda mas.msa$$

Notice that **Pipe** is none other than the **C**-combinator.

For completeness, we now define the **Lift** operator. This is a straightforward encoding of the abstract definition given in the previous section:

$$\mathbf{Lift} \quad = \quad \lambda fvsx.x(fv)s$$