Up to Our Gonads in Monads: Mommy, Where Does `>>=` Come From?

Sam Hopkins

December 7, 2012

Abstract

Technical perspective on [Moggi(1989)], for Dan Grossman's CSE 505, Fall 2012. The intended audience is young Haskell children who have been so brazen as to ask the (sub)titular question. *Disclaimer:* All the mathematics in this paper is only morally correct: the intuitions are right, but the technical details are often lacking or flat-out wrong.

Introduction Just what is a monad, anyway? Haskell programmers are used to using them to handle IO and state, but the relationship between monads and computation runs deeper than that. Well before the first release of the Glasgow Haskell Compiler, Eugenio Moggi's [Moggi(1989)] laid out the foundational relationship between monads and computation. We will survey the paper's main ideas in a fashion accessible to anybody with just some experience using, say the `IO` monad.

Abstract Nonsense We cannot do justice to Moggi's paper without a preliminary introduction to *category theory.* Category theory first arose in the 1940s from a branch of mathematics called, *algebraic topology* (which is also related to programming languages, see [Various(2012)]). It has grown into both a branch of mathematics unto itself and a spectacularly useful mathematical tool. As we will employ it here, category theory provides a mathematical language at the right level of abstraction (and as any programmer knows, a language at the right level of abstraction for the task at hand is a valuable thing indeed). The definitions that follow difficult to motivate without becoming acquainted with a great many examples: the interested computer scientist should consult [Pierce(1991)]; the mathematically-inclined should consult [Mac Lane(1998)].

Definition. A *category* $\mathcal{C}$ is a collection of objects, denoted $Ob(\mathcal{C})$, so that for each $A, B \in Ob(\mathcal{C})$ there is a set of *morphisms from A to B.* If f is such a morphism, we write $f : A \to B$. Furthermore, composition of morphisms is well-defined: if $f : A \to B$ and $g : B \to C$ then there is $f \circ g : A \to C$, and each object A has a special identity morphism $id_A : A \to A$ so that for any morphism $f : B \to A$, $id_A \circ f = f$, and for any morphism $g : A \to B$, $g \circ id_A = g$.

Moggi's observation is that, in the presence of not too much more structure, we can define a notion of computation in the setting of a general category. Moggi gives an (extremely simple) programming language and provides an *interpretation* of the programming language into an arbitrary category (with this additional structure): given a well-typed program P, a category $\mathcal{C}$, and assignments of all the basic types in the language (e.g. `int`) to objects in $\mathcal{C}$, and basic operations on those types (e.g. `+`) to appropriate morphisms in $\mathcal{C}$, we can find a morphism corresponding to P. The internal structure of the category: which morphisms there are, between which objects, reflects the structure of the set of programs in the language: what programs there are, and between what types.

We need to grapple with this extra structure (which should look very familiar to Haskell programmers).

Definition. A *Kleisi triple*[1] over $\mathcal{C}$ is a triple $(T, \eta, *)$, where T associates to each object A another object TA, η associates to each object A a morphism $\eta_A : A \to TA$, and $*$ is a unary operator on morphisms, so that if $f : A \to TB$ then $f^* : TA \to TB$[2], so that the following laws hold. We will express these laws in the

[1] From any Kleisi triple we can recover a *monad* over $\mathcal{C}$, which has a slightly different definition. Kleisi triples provide a better inuitive understanding of the structure.

[2] For the jargon-curious: T is almost what we would call a *functor* from $\mathcal{C}$ to itself (it needs to be extended to operate on morphisms, but the extension is straightforward), and η then becomes a *natural transformation* from the identity functor on $\mathcal{C}$ to T.

form of *commuting diagrams*,[3] but dear reader do not fear![4] We read the diagrams in the obvious fashion: the nodes in the graph are objects in $\mathcal{C}$, and the arrows are morphisms. The rule is that any path around the diagram (composing morphisms as you go around) must give the same morphism.

$$\begin{tikzcd}
TA \arrow[r, "f^*"] & TB \\
A \arrow[u, "\eta_A"] \arrow[ur, "f"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, bend left, "\eta_A^*"] \arrow[r, bend right, "id_{TA}"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[d, "f^*"'] \arrow[dr, "(g^* \circ f)^*"] & \\
TB \arrow[r, "g^*"'] & TC
\end{tikzcd}$$

We might think of a Kleisi triple as a world where computation can take place. Here's some intuition. Think of the objects as types. If A is an object, then TA is the object of A-computations, η_A represents a trivial program which given an A just returns the computation that, when run, gives back that A, and $*$ takes a program f which takes an A and modifies it to take a TA, by first running the A-computation and then feeding the result to f. It turns out (and we will see evidence for it) that the three laws above are enough to formalize these intuitions.

It is worth keeping in mind the special case (being careful not to get trapped in it!) where $\mathcal{C}$ is the category of sets and we are just interested in deterministic computation without side-effects. Then TA is just the set A augmented with a special element (usually written $\perp$) to denote non-haltingness; TA then captures all the possible behaviors of a program that "should" output an A. But it is important to remember that the notion is much more general than this (indeed, the objects need not even be sets, in which case the intuitive account above is questionable), and by defining $(T, \eta, *)$ appropriately we can capture all sorts of computation paradigms: nondeterministic computation, computation with side-effects, etc. (It is a worthwhile exercise to formulate the appropriate definitions for these two examples.[5])

Concrete Sense Let's connect these back to Haskell's `Monad`s. `Monad` defines two operations, `return` and `>>=` (bind). Clearly our η is just `return`: it injects values into computations.[6] Bind is very nearly our lifting operator $*$; we could define $*$ in Haskell with `star f = \x -> x >>= f`. `Monad`s in Haskell obey three laws:

```
          return a >>= k  =  k a                (1)
            m >>= return  =  m                  (2)
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h    (3)
```

The interested reader can verify (and it is instructive to do so) that these laws correspond (in order) exactly to our three diagrams above.

Equivalence and Existence It is reasonable to ask why in the world we have put all this work into formulating wild abstractions when all we got back were `Monad`s with some pretty diagrams for the (yet-to-be-explained) monad laws. Moggi seeks to capture the intutive notion of a computation in the most general setting possible. A worthwhile criterion of success in this endeavor is whether the intutive notions of equivalence and halting acquire natural definitions in this setting. The categorical approach permits this investigation.

When working over a general category with Moggi's simplified language, the definition of equivalence becomes extremely natural: programs are equivalent just in case they correspond to the same morphism. The notion of halting becomes a special case (for deterministic, non-side-effecting computation) of something more general that Moggi terms "existence". A program exists if the morphism $f : A \to TB$ to which it corresponds can be expressed as $f = \eta_B \circ h$ for some $h : A \to B$. Roughly, this says that there is a well-defined non-computational morphism that the program represents in computational form. That existence and equivalence are so readily definable suggests that far from just being a way to hide I/O or statefulness, `Monad`s and their attendant laws express the conditions satisfied by a *computation strategy*.

[3]Can't have a paper about categories without some commuting diagrams!

[4]I'm looking at you, Dan.

[5]Hints: powerset, Cartesian product, respectively.

[6]Indeed, were we being more mathematically correct we would have required that η_A be mono for all A. When the category is **Set**, mono means injective.

Expanding the Language We have managed thus far to avoid the main technical contribution of [Moggi(1989)] (everything up till now is just kid stuff!), but no longer. The simplified language allows for an arbitrary collection of basic types and operations on values of those types, but the only way to compose those operations is with `let` (i.e. `let x = e1 in e2`), and, even worse, the only variable that `e2` can use which it does not define itself is `x`. This means that in the program

```
let x = e1 in let y = e2 in e3
```

the expression `e3` cannot use `x`. These restrictions make it very straightforward to define the correspondence between `let` and composition in the Kleisi triple, but they reveal that the generality of this of notion of computation is makes it in some ways considerably less expressive than a full-fledged programming language.

The main technical contribution of [Moggi(1989)] is to extend these monadic semantics to a fuller programming language, which has functions, and in which expressions can use more than one variable that they do not define themselves. The general categorical technique for the interpretation of functions was known well before [Moggi(1989)]; sparing the reader the categorical details we will just mention that there is a good categorical defnition (i.e. in terms just of morphisms and diagrams) of "the object of morphisms from A to B," denoted B^A. A function type `t1 -> t2` then corresponds to $(T[\![\texttt{t1}]\!])^{[\![\texttt{t2}]\!]}$, where $[\![\texttt{t1}]\!]$ is the object corresponding to type `t1` and similarly for $[\![\texttt{t2}]\!]$. This gives a nice categorical restriction on our choice of Kleisi triple if we want to be able to define a full programming language: the underlying category must have the appropriate exponential objects.

The other challenge, dealing with multiple undefined variables in an expression, is a little more accessible. We would like for an expression `e` using two undefined variables `x` and `y` to correspond to a morphism $A \times B \to TC$.[7] Then running the program with one input of the type corresponding to A and a second of the type corresponding to B would give a C-computation. In order to make this play nice with monadic composition, we need a way to move between $A \times TB$ and $T(A \times B)$, and "good" morphisms between those two objects are not guaranteed to exist in general. Moggi defines the properties of the desired morphisms and in the restricted class of structures in which they are present (*strong monads* with some special exponential objects to deal with the interaction of functions and multiple variables) provides the appropriate interpretation of a fully-powered programming language. Out of this falls some augmentation of the definition of program equivalence and existence in the setting of a full-power programming language; the augmented logic for equivalence and existence is the *λ_C-calculus.*

One last question arises. Haskell's `Monad`s are clearly strong enough to represent full-powered programming langauges (indeed, it is common to use them to implement interpreters and compilers). Why do we not need to provide the additional structure with which Moggi augemnts his Kleisi triples when we instantiate `Monad`? The answer is not so obvious, and this author has sketched only a preliminary argument, but it appears that other features of the type system guarantee the existence of the required extra structure; this in turn follows from a condition given by Moggi for the existence and uniqueness of strong monads.

Termination Let's do one last survey of the lay of the land. Category-theoretic monads (we have used a slightly different formulation, Kleisi triples) provide a general setting for computation; the fact that they capture a wide variety of computational paradigms (nondeterministic, probabilistic, etc.) while simultaneously allowing for extremely natural definitions of equivalence and existence is evidence that monads give us the right level of structural abstraction. [Moggi(1989)] offers the interpretation of a programming language into *one* monad, but Haskell *reifies* monads with `Monad`. *Reification* is taking a concept used to describe the semantics of a language and inserting it into the language itself. `Monad` lets Hakell programmers define their own computational universes. It is a convenient property of this reification that `Monad`s are isolated from each other and the core program; this allows the classic use of `Monad`s to isolate impure computation strategies from the (pure) Haskell core.

References

[Mac Lane(1998)] S. Mac Lane. *Categories for the working mathematician*, volume 5. springer, 1998.

[7] Again, we will not bother with the details of products $A \times B$ in general categories. The concerned reader may assume that we're working over **Set**.

[Moggi(1989)] Eugenio Moggi. Computational lambda-calculus and monads. *LICS*, 1989.

[Pierce(1991)] B.C. Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

[Various(2012)] Various. Homotopy type theory, 2012. URL `http://homotopytypetheory.org/`.