Predicate Monads:

A Framework for Proving Generic Properties of Monadic Programs via Rewriting

Edwin Westbrook
Galois Inc.
westbrook@galois.com

Gregory Malecha
gmalecha@gmail.com

1. Introduction

Monads are a simple, elegant, and powerful way to specify effectful programs. They provide a compositional framework for building program specifications that supports a wide variety of different sorts of effects, including mutable state, continuations, errors and exceptions, non-determinism, and many more (Moggi 1991). These effects are defined in a given monad M by exposing a set of *monadic operations* to implement these effects in M. For instance, if M supports mutable state it often exposes it through the following type signature:

$$\begin{array}{lll} \mathsf{returnM} & :: & \forall A.A \to M\ A \\ (\gg\!=) & :: & \forall A.\forall B.M\ A \to (A \to M\ B) \to M\ B \\ \mathsf{getM} & :: & M\ S \\ \mathsf{putM} & :: & S \to M\ \mathsf{unit} \end{array}$$

The first two of these, returnM and $\gg\!=$, are the usual monad operations, for building pure computations and for sequencing two computations together, while getM and putM define operations for reading and writing, respectively, the current value of the mutable state of type S. Note that we also use the standard abbreviation

$$m_1 \gg m_2 \triangleq m_1 \gg\!= \lambda\, x : \mathsf{unit} . m_2\ .$$

to sequecne a computation m_1 followed by a computation m_2 that does not inspect the return value of m_1.

Each monad also comes with a set of reasoning principles, called generally the *monad laws*, for reasoning about programs written in that monad. These monad laws take the form of a set of equalities on programs. For instance, in addition to the standard monad laws for returnM and $\gg\!=$, a monad that supports mutable state will generally satisfy the following *state monad laws*, stating that getM and putM behave as expected:

$$\begin{array}{lcl} \mathsf{putM}\ s \gg \mathsf{putM}\ s' & \approx & \mathsf{putM}\ s' \\ \mathsf{putM}\ s \gg\!= \mathsf{getM} & \approx & \mathsf{putM}\ s \gg \mathsf{returnM}\ s \\ \mathsf{getM} \gg\!= \mathsf{putM} & \approx & \mathsf{returnM}\ \mathsf{tt} \\ \mathsf{getM} \gg\!= \lambda s.\, \mathsf{getM} \gg\!= f\ s & \approx & \mathsf{getM} \gg\!= \lambda s.\, f\ s\ s \end{array}$$

The notation $\approx$ denotes the equality relation for the monad M, which may or may not correspond with the built-in equality of the meta-logic (i.e., with intensional equality in Coq); we assume every type comes equipped with a pre-order, which we formalize in Coq with a type class, and we write $\lessapprox$ for this pre-order and $(\approx) \triangleq (\lessapprox) \cap (\lessapprox)^{-1}$ as the equivalence relation derived from $\lessapprox$. Technically speaking, we formalize monads as functors over types plus orders, and the versions of the monad laws in our formulation include monotonicity constraints, i.e., they require that the various components be Proper, but we ignore this detail here for space reasons.

A monad that supports multiple effects will additionally include monad laws for how those effects interact; e.g., a monad with mutable state and errors might include the law

$$\mathsf{putM}\ s \gg \mathsf{failM} \approx \mathsf{failM}$$

to indicate that a failure eradicates all mutable state modifications. A similar monad with these effects, however, might not satisfy this law, if, for instance, failures can be recovered from.

Monad laws are a powerful tool that can be used to prove a wide variety of properties of effectful programs, by using them as rewrite rules. For instance, Gibbons and Hinze use this approach to verify that a number of effectful programs have the same behavior as their corresponding functional specifications (Gibbons and Hinze 2011). The key difficulty comes when we want to prove properties other than equalities. Monad laws cannot, for instance, prove that a function satisfies a given pre- and post-condition, because this is not a property that can be defined in terms of equality. Although there are approaches specifically for proving pre- and post-conditions, such as the Hoare monad (Nanevski et al. 2008) and the Dijkstra monad (Swamy et al. 2013), it is not clear that these approaches generalize to effects other than mutable state, or to combinations of effects. It is also not clear how to derive the Hoare monad or the Dijkstra monad structure for an arbitrary monad that does support mutable state.

In this talk, we will present ongoing work on a general framework, called *predicate monads*, for proving properties of effectful programs that overcomes these difficulties. The goal of this approach is to leverage the power and elegance of the monad laws, but apply them not to monadic programs themselves but to predicates over monadic computations. More technically, in our approach, each monad M is associated with a so-called predicate monad P_M which is, intuitively, a monad of predicates on computations in M. If we view M as a domain-specific semantics, then P_M is a *domain-specific logic*. The predicate monad P_M supports predicates over all of the monadic operations of M, allowing it to reason about arbitrary effects. This reasoning comes in the form of *predicate monad laws*, which allow predicates to be proved using rewriting. Additionally, if M is defined using monad transformers, a powerful and compositional way to build monads (Liang et al. 1995), then P_M can be built using the same monad transformers. This allows us to derive predicate monads for a wide variety of monads and effects in a straightforward manner.

2. Building Predicate Monads

To define predicate monads formally, we capture the notion as a new sort of computational effect, the "logic over M" effect. Just as with other sorts of effects, this means adding a number of monadic operations and a set of monad laws for those operations. The operations for predicate monads are:

$$\begin{array}{lll} \mathsf{forallP} & :: & \forall A.\forall B.(A \to P_M\ B) \to P_M\ B \\ \mathsf{existsP} & :: & \forall A.\forall B.(A \to P_M\ B) \to P_M\ B \\ \mathsf{impliesP} & :: & \forall A.P_M\ A \to P_M\ A \to P_M\ A \\ \mathsf{singleP} & :: & \forall A.M\ A \to P_M\ A \end{array}$$

Intuitively, the first three operations build predicates corresponding to universal quantification, existential quantification, and implication. Note that conjunction and disjunction operators andP and orP can be defined from $\mathsf{forallP}$ and $\mathsf{existsP}$, respectively, as can the greatest predicate trueP and the least predicate falseP. We define

$$\mathsf{assertP}\ (P : \mathsf{Prop}) : P_M\ A \triangleq \mathsf{existsP}(\lambda\, p{:}P\,.\,\mathsf{trueP})$$

as the predicate that holds iff P is provable. Finally, the fourth operation above, $\mathsf{singleP}$, builds the singleton predicate, i.e., the least predicate containing a given monadic computation.

Leastness here is with respect to the pre-order $\lesssim$ of P_M, which can be viewed as entailment over predicates; i.e., $p_1 \lesssim p_2$ means that any computations which satisfy p_1 will always satisfy p_2. The equality $\approx$ associated with P_M thus denotes logical equivalence of two predicates, so that $p_1 \approx p_2$ means that p_1 and p_2 hold for the same computations in M. To define the notion of which computations in M satisfy a predicate in P_M, we define

$$m \vDash p \triangleq \mathsf{singleP}\ m \lesssim p$$

to capture the notion that computation m of type $M\ A$ satisfies predicate p of type $P_M\ A$.

When we reason about predicate monads, we use the following laws which are stated in terms of the ordering relation on the predicate monad.

$$\begin{array}{rcl} \mathsf{singleP}\ (\mathsf{returnM}\ x) & \approx & \mathsf{returnM}\ x \\ \mathsf{singleP}\ (m \gg\!= f) & \approx & (\mathsf{singleP}\ m) \gg\!= (\lambda x.\,\mathsf{singleP}\ (f\ x)) \end{array}$$

$$\begin{array}{rcll} \mathsf{forallP}\ p & \lesssim & p\ x & \mathsf{forall}\ x \\ (\forall x.p \lesssim q\ x) & \to & p \lesssim \mathsf{forallP}\ q & \\ p\ x & \lesssim & \mathsf{existsP}\ p & \\ (\forall x.p\ x \lesssim q) & \to & \mathsf{existsP}\ p \lesssim q & \\ \mathsf{andP}\ p_1\ p_2 \lesssim p_3 & \leftrightarrow & p_1 \lesssim \mathsf{impliesP}\ p_2\ p_3 & \end{array}$$

$$\begin{array}{c} (\forall x.\forall y.x \lesssim y \to p\ x \lesssim q\ x) \to \mathsf{forallP}\ p \lesssim \mathsf{forallP}\ q \\ (\forall x.\forall y.x \lesssim y \to p\ x \lesssim q\ x) \to \mathsf{existsP}\ p \lesssim \mathsf{existsP}\ q \\ (p_1' \lesssim p_1 \wedge p_2 \lesssim p_2') \to \mathsf{impliesP}\ p_1\ p_2 \lesssim \mathsf{impliesP}\ p_1'\ p_2' \end{array}$$

The first two laws state that applying $\mathsf{singleP}$ to a monadic operation in M always yields an application of the same monadic operation in P_M. That is, $\mathsf{returnM}\ x$ in P_M builds a predicate for recognizing the computation $\mathsf{returnM}\ x$ in M, while $p \gg\!= q$ in P_M builds a predicate for recognizing computations that are equivalent, modulo the monad laws, to $m \gg\!= f$ in M for some $m \vDash p$ and some f such that $\forall x.fx \vDash Px$. The second set of laws state that $\mathsf{forallP}$, $\mathsf{existsP}$, and $\mathsf{impliesP}$ satisfy usual introduction and elimination rules for the corresponding connectives in first-order logic. Note that, viewed differently, the laws for $\mathsf{forallP}$ and $\mathsf{existsP}$ state that $\lesssim$ is a complete lattice, while the law for $\mathsf{impliesP}$ states that $\lesssim$ is a Heyting algebra. Finally, the third set of rules express monotonicity constraints, also known in Coq as Proper constraints, on the predicate monad operators. Not shown, for space reasons, are laws stating that the logical connectives commute with $\gg\!=$; e.g., $(\mathsf{forallP}\ f) \gg\!= g \approx \mathsf{forallP}\ (\lambda x.\ f\ x \gg\!= g)$.

In order to build predicate monads with this structure, we start by defining the predicate monad P_{Identity} for the simplest possible monad, the $\mathsf{Identity}$ monad. It turns out that the Set monad, defined as $\mathsf{Set}\ A \triangleq A \to \mathsf{Prop}$ has the desired structure, by defining $\mathsf{forallP}$, $\mathsf{existsP}$, and $\mathsf{impliesP}$ to be the straightforward application of the corresponding combinators in Prop, and by defining

$$p \lesssim q \triangleq \forall x.p\ x \to \exists y.x \lesssim y \wedge q\ y$$

meaning that entailment in the Set predicate monad corresponds to a "covering" property, where each element of the lesser set has a greater element in the greater set. It also turns out that, for many standard monad transformers $\mathcal{T}$, we can define $P_{\mathcal{T}(M)} \triangleq \mathcal{T}(P_M)$, though, again, we omit the details here. This means that, we can build up predicate monads for a wide variety of different sorts of effects, using the same "transformer stack" used to build the underlying monad.

3. Proving Properties using Predicate Monads

We now briefly consider how to formulate and prove Hoare-style pre- and post-conditions using predicate monads, as an illustrative example. We start with this definition:

$$\begin{array}{l} \mathsf{HoareP}(\phi : S \to \mathsf{Prop})(\psi : S \to A \to S \to \mathsf{Prop})(p : P_M\ A) \triangleq \\ \quad \mathbf{do}\ s \leftarrow \mathsf{getM} \\ \qquad \mathsf{impliesP}\ (\mathsf{assertP}\ (\phi\ s)) \\ \qquad\quad (\mathbf{do}\ x \leftarrow \mathsf{catchM}\ p\ (\lambda e.\,\mathsf{falseP}) \\ \qquad\qquad s' \leftarrow \mathsf{getM} \\ \qquad\qquad \mathsf{andP}\ \mathsf{assertP}\ (\psi\ s\ x\ s')\ (\mathsf{returnM}\ x)) \end{array}$$

If we pass trueP for the argument p, then this intuitively captures total correctness for pre-condition ϕ and post-condition ψ. It is satisfied by computations equivalent to one that first reads the input state s, and, if $\phi\ s$ holds, performs any computation m with return value(s) x such that $\mathsf{catchM}\ m\ f \approx m$ — i.e., any computation with no errors — and then gets the output state s' and returns x, such that $\psi\ s\ x\ s'$ holds. Note that this definition says nothing about the internals of either M or P_M, as long as M and P_M have the expected operations for mutable state and error effects. This allows us to define Hoare-style pre- and post-conditions generically, and also to prove properties — such as the transitivity rule for Hoare logic — without reference to the definition of M.

The reason to have the p, rather than always just using trueP for p, is that HoareP is monotonic in p, so we can prove $m \vDash \mathsf{HoareP}\ \phi\ \psi\ \mathsf{trueP}$ by proving $m \lesssim \mathsf{HoareP}\ \phi\ \psi\ (\mathsf{singleP}\ m)$, which we can do by rewriting via both the monad and the predicate monad laws. For instance, if m has the form $\mathsf{getM} \gg\!= \lambda s.\,\mathsf{putM}\ (f\ s) \gg \mathsf{returnM}\ (g\ s)$, then rewriting this formula and then applying the Proper-ness rules for $\gg\!=$ yields

$$\forall s.\phi\ s \to \psi\ s\ (g\ s)\ (f\ s)$$

which is exactly what we would expect; e.g., $\mathsf{catchM}\ m\ (\lambda e.\,\mathsf{falseP})$ in this case rewrites to m.

References

J. Gibbons and R. Hinze. Just do it: Simple monadic equational reasoning. In *ICFP*, 2011.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, 2008.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, 2013.