Advanced Programming 2022

Introduction to Monads

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 13, 2022

Where are we?

- In first lecture, saw some general FP concepts and constructs:
  - (Pure) value-oriented computation paradigm
  - Functions as values
  - Algebraic datatypes and pattern matching
- In second, looked at more advanced, Haskell-specific features:
  - Type classes, including type-constructor classes
  - Laziness and equational reasoning
  - Purely functional IO
  - List comprehensions
- **Today:** functional programming with *monads*.
  - Conceptually simple idea (literally, just a few lines of code)
  - But profound impact on Haskell programming style
    - Even reflected in official language logo:
  - Draws upon many topics from previous two lectures

Why monads?

- **Misconception:** "selling out" on *pure* functional programming.
  - Allowing mutable store, exceptions, I/O, ... back in
  - ... after working so hard to banish them!
- **Actual goal:** organizing all "impurity" into distinct *effects*.
  - Type system keeps track of which functions have which effects.
    - Many still have have exactly none.
  - Can be as fine-grained or coarse-grained as programmer wishes
    - E.g., distinguish read-write vs. read-only vs. append-only state
- Each kind of effect is encapsulated as a *monad*.
  - Just another type class, like `Monoid` or `Functor`, with a few laws.
  - Programming and reasoning remain *completely pure*.
- Framework can directly accommodate problem-specific effects (e.g., backtracking search with *oracles*).
  - Same syntax as "standard" effects.
  - Bulk of program code doesn't need to change when adding or adjusting effect.

A bit of historical context for monads

- Concept first formulated 1960s–70s, in *category theory*.
  - Particularly abstract branch of mathematics, no apparent connection to (especially impure) programming.
- *Computational lambda calculus and monads* (E. Moggi, 1989)
  - Worked in context of *denotational semantics*, a formalism for describing language features using pure mathematical functions
  - Recognized that almost all common notions of effects were instances of the same *mathematical* pattern.
  - "Test of Time" award in 2009 for most influential paper from 1989 *Logic in Computer Science* conference.
- *Comprehending monads* (P. Wadler, 1990)
  - Recognized that almost all of Moggi's constructions could be reformulated in a pure functional language, not mathematics.
  - But presentation of monads still rooted in categorical concepts.
  - Proposed a generalization of *list-comprehension* notation, which eventually evolved into current `do`-notation.

A brief history of monads, continued

- *The essence of functional programming* (P. Wadler, 1992)
  - Tutorial introduction to “monadic style” of programming, especially for writing interpreters.
  - Relatively close to modern syntax and terminology, but not yet using type classes.
- Various developments, 1990s–2020s
  - Gradual evolution into current form.
  - Lots of generalizations and refinements (e.g., `Applicatives`)
  - Extensive and powerful (but also somewhat complicated) *Monad Transformer Library (MTL)* for GHC.
  - Staggering number (100+) of “monad tutorials” out there...
- *Advanced Programming*, 2022
  - Back to the essentials, but in modern context.
  - Really not that complicated, but don’t despair:
    - “Young man, in Mathematics we don’t *understand* things. We just get used to them.” – J. von Neumann, answering a student.

Motivating example 1: Exceptions/errors

- Consider function that may need to signal an error condition
  - Often, only one possible error, e.g., "key not found".
- **Observation:** a *partial* function of type *a* `->` *b* can be represented as *total* function of type *a* `-> Maybe` *b*
- Typical use of `lookup :: Eq a => a -> [(a, b)] -> Maybe b`:

```
case lookup k m of
  Just v -> ...   -- continue with computation, using v
  Nothing -> ...  -- deal with error
```

- When looking up two keys:

```
case lookup k1 m of
  Just v1 -> case lookup k2 m of
               Just v2 -> ... -- continue, using v1 and v2
               Nothing -> ... -- deal with error
  Nothing -> ...              -- deal with error
```

- *Exception-passing style:* explicitly check all subcomputation results and propagate failures.
  - Makes control flow apparent, but clutters everything.

Motivating example 2: Mutable, global state

- Consider function that may need to access a global variable.
  - For concreteness, assume said variable has type `Int`.
  - E.g., random-number seed, allocation counter, ...
- **Observation:** a *side-effecting* function of type *a* `->` *b* can be represented as *pure* function of type `(`*a*`, Int) -> (`*b*`, Int)`, or equivalently (in curried style), *a* `-> Int -> (`*b*`, Int)`.
- Typical sequencing pattern, when computing `g (f a)` (as *effectful* functions):

```
let (b, i1) = f a i0   -- i0 is initial value of var.
    (c, i2) = g b i1
in ...                 -- i2 is final value of var.
```

- *State-passing style:* explicitly thread current value of global variable through all function calls.
  - Makes data flow and dependencies apparent, but clutters everything.

A unified view

- In both examples, same pattern: effectful function of type a `->` b corresponds to total, pure function of type a `-> M` b.
- `M` t is the type of *computations* returning a result of type t.
  - For errors, `type M t = Maybe t`
  - For state, `type M t = Int -> (t, Int)`
- Computation that just returns a given value: `unit :: a -> M a`
  - For errors: `unit a = Just a`
  - For state: `unit a = \s -> (a, s)`
- Computation that applies effectful function to result (if any) of effectful computation: `bind :: M a -> (a -> M b) -> M b`
  - For errors:
    ```
    bind (Just a) f = f a
    bind Nothing f = Nothing
    ```
  - For state: `bind m f = \s -> let (a,s1) = m s in (f a) s1`
- Such a triple `(M, unit, bind)` constitutes a *monad*.

The Monad class

- Class of *type constructors*, like `Functor`.

```
class Applicative m => Monad m where
  return :: a -> m a                      -- unit
  (>>=) :: m a -> (a -> m b) -> m b  -- bind
  (>>) :: m a -> m b -> m b    -- bind, ignoring first value
  m1 >> m2 = m1 >>= \_ -> m2  --   default implementation
  -- fail :: String -> m a     -- now only in MonadFail
  -- fail s = error s
```

  - (Ignore the `Applicative m` constraint for now.)

- Predefined instances in standard prelude, e.g.:

```
instance Monad Maybe where
  return a = Just a
  Just a  >>= f = f a
  Nothing >>= f = Nothing
  -- fail s = Nothing
```

Monad instances for states and exceptions

- Start by defining suitable new-type constructor (not synonym):

```
newtype IntState a = St {runSt :: Int -> (a, Int)}
instance Monad IntState where
  return a = St (\s -> (a, s))
  m >>= f = St (\s0 -> let (a,s1) = runSt m s0
                       in runSt (f a) s1)
```

- In fact, can generalize to arbitrarily typed state:

```
newtype State s a = St {runSt :: s -> (a, s)}
instance Monad (State s) where
  -- defs of return, (>>=) exactly as above
```

  - For any fixed s, (`State` s) itself is still a *type constructor*.
- Likewise, can define a general error monad parameterized by type of error values (exception data), where `Maybe a` $\approx$ `Either () a`:

```
data Either a b = Left a | Right b -- in standard prelude
instance Monad (Either e) where
  return a = Right a
  (Left e)  >>= f = Left e
  (Right a) >>= f = f a
```

Monad laws

- All instances M of `Monad` should satisfy the three *monad laws*:
  1. `return` v `>>=` f $\simeq$ $f\ v$
  2. m `>>= (\a -> return a)` $\simeq$ m
  3. `(`m `>>=` f`) >>=` g $\simeq$ m `>>= (\a -> (`f `a >>=` g`))`

  (for all types a, b, and c, and all values v `::` a, m `::` $M\ a$, f `::` a `->` $M\ b$, and g `::` b `->` $M\ c$.)
- Roughly say that composition of represented effectful functions behaves as expected (in particular, is associative).
- **Note:** these equations are between monadic-type expressions, which may not have `Eq` instances.
  - Recall that $\simeq$ means “behaves indistinguishably from”.
  - Can check laws by essentially mathematical reasoning about purely functional expressions, “replacing equals by equals”
- If `Monad` instance satisfies laws, clever optimizations possible.
  - Including using imperative implementation “under the hood”, such as destructive state updates.
  - Details beyond the scope of this course.

Verifying the monad laws

For example, checking Law 1 for the `State` monad:

```
  return v >>= f
≃ -- def. of >>= for State
  St (\s0 -> let (a,s1) = runSt (return v) s0 in runSt (f a) s1)
≃ -- def. of return for State
  St (\s0 -> let (a,s1) = runSt (St (\s -> (v, s))) s0
             in runSt (f a) s1)
≃ -- runSt (St h) ≃ h  (projection . constructor ≃ id)
  St (\s0 -> let (a,s1) = (\s -> (v, s)) s0 in runSt (f a) s1)
≃ -- (\x -> e₁) e₂ ≃ e₁[x ↦ e₂] (subst. actual for formal)
  St (\s0 -> let (a,s1) = (v, s0) in runSt (f a) s1)
≃ -- let (x₁,x₂) = (e₁,e₂) in e₃ ≃ e₃[x₁ ↦ e₁, x₂ ↦ e₂]
  St (\s0 -> runSt (f v) s0)
≃ -- \x -> e x ≃ e, if no occurrences of x in e
  St (runSt (f v))
≃ -- St (runSt m) ≃ m  (constructor . projection ≃ id)
  f v
```

Monads are Functors and Applicatives

- Category-theoretically, every monad is also a functor.
- For any `Monad` instance *M*, can derive a `Functor` instance by:

```
instance Functor M where
  fmap f m = m >>= \a -> return (f a)  -- or fmap = liftM
```

  - **Fact:** If *M* satisfies the monad laws, `fmap` will satisfy functor laws.
- GHC 7.10 (2015) actually made `Monad` a *subclass* of `Functor`.
  - Must have `instance Functor` *M* to even be able to declare `instance Monad` *M*.
  - Older code and textbook examples no longer compile as written!
  - Fortunately, can still define `fmap` using `return` and `>>=`, as above.
    - Can just include above instance declaration verbatim (with *M* replaced by the name of your Monad), to satisfy compiler.
- Likewise, `Monad` is now a subclass of `Applicative`.
  - Generally enough to include following magic phrase (for each *M*):

```
instance Applicative M where
  pure = return; (<*>) = ap  -- from Control.Monad
```

  - Will not say more about `Applicative` operations here, but you're welcome to use them if you like.

General programming with effectful functions

- Using `return` and `>>=`, can now combine computations in generic way, even when they have effects.
- Consider simple higher-order function:

```
pair :: (a -> b) -> (a -> c) -> a -> (b, c)
pair f g a = let b = f a; c = g a in (b, c)
```

- What if `f` and `g` may have effects? Then so will their pairing:

```
pairM :: Monad m => (a -> m b) -> (a -> m c) -> a -> m (b, c)
pairM f g a = f a >>= \b -> g a >>= \c -> return (b, c)
```

- **Note:** Same code works, whether `m` represents error-signaling or state-manipulating computations (or *any other* monad).
- Had we not used monads, would need *separate* definitions:

```
pairE f g a =
  case f a of Just b -> case g a of Just c -> Just (b,c)
                                    Nothing -> Nothing
            Nothing -> Nothing

pairS f g a =
  \s0 -> let (b,s1) = f a s0 in let (c,s2) = g a s1 in ((b,c), s2)
```

Effectful operations in a monad

- Have seen how to combine (e.g.) state-manipulating functions in general way.
- But how do we actually *access* the state?
- Following instance declaration, we can also define additional functions, specific to the particular monad, e.g.:

```
newtype IntState a = St {runSt :: Int -> (a, Int)}
instance Monad IntState where ...        -- as before

getState :: IntState Int                 -- read the state
getState = St (\i -> (i, i))

putState :: Int -> IntState ()           -- write the state
putState i' = St (\i -> ((), i'))

modifyState :: (Int -> Int) -> IntState () -- "bump" state
modifyState f = St (\i -> ((), f i))
```

- Generalizes directly to `Monad (State` s`)`, for any s.
- Will see operations for other monads shortly

do-notation

- For increased readability, Haskell offers a convenient notation for writing chains of monadic computations, e.g.:

```
pairM :: Monad m => (a -> m b) -> (a -> m c) -> a -> m (b, c)
pairM f g a = do b <- f a; c <- g a; return (b, c)
```

- Cf. `let`-notation for non-effectful pairing:

```
pair :: (a -> b) -> (a -> c) -> a -> (b, c)
pair f g a = let b = f a; c = g a in (b, c)
```

- General syntax: `do` $stmt_1$ `;` … $stmt_n$ `;` $mexp_0$ $(n \geq 0)$
  - Each $stmt_i$ may be of one of three forms:
    - pat_i `<-` $mexp_i$
    - $mexp_i$
    - `let` pat_i `=` exp_i
  - All $mexp_i$ must be of monadic type (for the same monad).
  - The semicolons are normally replaced by newlines + indentation.
- All uses of `do`-notation are simplified ("desugared", $\rightsquigarrow$) into standard monad operations early in the compilation process.

Desugaring do-notation, continued

- First step: bring all `do`s into form with exactly one *stmt*:

  $\texttt{do}\ mexp_0 \ \rightsquigarrow\ mexp_0$

  $\texttt{do}\ stmt_1; stmt_2; \ldots; stmt_n; mexp_0 \ \rightsquigarrow$

  $\quad \texttt{do}\ stmt_1; (\texttt{do}\ stmt_2; \ldots; stmt_n; mexp_0)$ (use repeatedly while $n \geq 2$)

- (So now only need to deal with: `do` *stmt* `;` $mexp_0$)

- Next: desugaring rules for each kind of *stmt*:

  $\texttt{do}\ pat\ \texttt{<-}\ mexp_1\texttt{;}\ mexp_0 \ \rightsquigarrow\ mexp_1\ \texttt{>>=}\ \backslash pat\ \texttt{->}\ mexp_0$

  If matching against *pat* cannot fail: x or (x, y) are OK; $[x, y]$ is not.

  $\texttt{do}\ mexp_1\texttt{;}\ mexp_0 \ \rightsquigarrow\ mexp_1\ \texttt{>>}\ mexp_0$

  $\texttt{do let}\ pat\ \texttt{=}\ exp\texttt{;}\ mexp_0 \ \rightsquigarrow\ \texttt{let}\ pat\ \texttt{=}\ exp\ \texttt{in}\ mexp_0$

- **Example:**

```
do c <- getChar
   let c’ = toUpper c
   putStr ("Hi "++[c’])
   return $ ord c
```

$\rightsquigarrow$

```
getChar >>= \c ->
  let c’ = toUpper c in
    putStr ("Hi "++[c’]) >>
      return $ ord c
```

A few more, frequently useful monads

- Have already seen general state (`State s`) and exception (`Either e`) monads.
- Let's see a few others:
  - Read-only state (`Reader s`)
  - Accumulating state (`Writer s`)
  - Nondeterminism/backtracking (`[]`)
  - I/O (`SimpleIO`) (next time)

Read-only state

- Useful for parameterizing (sub)computation with some additional data that will stay constant throughout computation.
- A cut-down variant of the `State` monad constructor:

```
newtype Reader d a = Rd {runRd :: d -> a}

instance Monad (Reader d) where
  return a = Rd (\d -> a)
  m >>= f = Rd (\d -> let a=runRd m d in runRd (f a) d)

ask :: Reader d d
ask = Rd (\d -> d)

local :: (d -> d) -> Reader d a -> Reader d a
local f m = Rd (\d -> runRd m (f d))
  -- often used as: local (const d') m
```

Read-only state, continued

- Sample use: expression evaluator

```
data Expr = Const Int | Var String | Plus Expr Expr
          | Let String Expr Expr

eval :: Expr -> Reader (String -> Int) Int
eval (Const n) = return n
eval (Var x) = do r <- ask; return (r x)
eval (Plus e1 e2) =
  do n1 <- eval e1; n2 <- eval e2; return (n1 + n2)
eval (Let x e1 e2) =
  do n1 <- eval e1
     local (\r -> \y -> if y == x then n1 else r y)
           (eval e2)

evalTop :: Expr -> Int
evalTop e = runRd (eval e)
                  (\x -> error $ "unbound variable: " ++ x)
```

Accumulating state

- Sometimes computations can only "add" to an accumulator:
  - Append line to log
  - Increment event counter
  - Possibly adjust "high-water mark" to new maximum
- Want to make it manifest from types that computations can neither *read* from the accumulator, nor *reset* it:

```
newtype Writer w a = Wr {runWr :: (a, w)}
instance Monoid w => Monad (Writer w) where
  return a = Wr (a, mempty)
  m >>= f = let (a, w1) = runWr m
                (b, w2) = runWr (f a)
            in Wr (b, w1 <> w2)

tell :: w -> Writer w ()
tell w = Wr ((), w)

runWr (do tell "foo"; tell "bar"; return 5) -- (5, "foobar")
```

- **Fact:** if *w* satisfies `Monoid` laws, `Writer` *w* will satisfy `Monad` ones.

Nondeterminsm

- Sometimes want to express that several *alternative* results are possible from a computation.
  - E.g., a function returning an arbitrary element of a list or set
- Standard prelude declares a `Monad` instance for lists:

```
instance Monad [] where  -- remember: type [t] is ([] t)
  return a = [a]
  m >>= f = concat (map f m)  -- concat :: [[a]] -> [a]
                -- actually, concatMap f m: pre-composed
```

- List comprehensions become simply `do`-notation:
  - `[exp | ` $qual_1$`, ..., `$qual_n$`]` $\rightsquigarrow$ `do` $stmt_1$`; ...; `$stmt_n$`; [`*exp*`]`
    - Note: using `[`*exp*`]` instead of `return` *exp* to force *list* monad.
  - Generator-qualifers *x* `<-` *lexp* simply kept as monadic bindings.
  - Guard-qualifiers *bexp* are like trivial/impossible bindings: *bexp* $\rightsquigarrow$ `if` *bexp* `then return () else []`
    - Or: *bexp* $\rightsquigarrow$ `_ <- if` *bexp* `then [()] else []`

Summary: monads from a SE perspective

- Monads abstract out the "plumbing" inherent to any notion of effectful computations.
- Not really *essential*: could just write programs explicitly in state-pasing, error-passing, etc. style.
- But doing so loses key benefits of abstraction:
  - More concise, readable code
  - Less room for manual error
  - Subsequent fixes, changes, or extensions to effect only have to be done in one place.
    - Cf. Wadler's interpreter examples.
- Any non-trivial piece of Haskell code you are likely to encounter will probably already make heavy use of monads.

What's next

- If you haven't yet, do the recommended readings and quiz for this lecture.
  - May want to start with Wadler paper.
- Exercise classes 10-12 (1 hr); see Abasalon (same rooms as last Tuesday)
- **Study-start test due by 12:00 today!**
  - If new MSc student, see your study-start course room in Absalon
- Assignment 1 due Friday
  - Do run it past OnlineTA before submitting!
- Start looking at this week's excercises and assignments
  - Assignment 2 out Thursday, due **Friday, 23/9 at 22:00**
- Thursday's lecture:
  1. Monads, continued
  2. Testing basics