Advanced Programming 2017

Introduction to Monads

Andrzej Filinski
andrzej@di.ku.dk

Department of Computer Science
University of Copenhagen

September 12, 2017

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
  - 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

Functional programming with effects

- Imperative (and non-pure functional) languages often support a wide variety of *effectful* features, intermixed with simple expression evaluation:
  - Mutable state, both explicit and implicit (e.g., PRNG seed)
  - Exceptions (with or without handlers)
  - Nondeterministic search and backtracking (SNOBOL, Prolog)
  - Generalized control ("long jumps", "first-class continuations")
  - Concurrency ($=$ multi*programming*, $\neq$ multi*processing*)
  - Interactive I/O and communication
- Generally deeply embedded into language definition and compiler.
- Effects often important for writing concise code, but also severely complicate reasoning about programs.
- Long seemed fundamentally incompatible with *purely* functional programming...
  - ... until monads came along!

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)
  - Work in context of *denotational semantics*, a formalism for describing language features using pure mathematical functions
  - Recognized that virtually 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 observations could be reformulated in a pure functional language.
  - But presentation of monads still rooted in categorical tradition.
  - 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 interpreters.
  - Relatively close to modern syntax and terminology, but not yet using type classes.
- Various refinements, 1990s–2010s
  - Gradual evolution into current form
  - Lots of embellishments and refinements
  - Extensive and general (but also complicated) *Monad Transformer Library (MTL)* for GHC
- *Advanced Programming*, 2017
  - 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 pattern:

```
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
  - POSIX C API: must check almost all functions for error returns.

Motivating example 2: Mutable, global state

- Consider function that may silently modify 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 >>= \a -> m2  -- default implementation

  fail :: String -> m a
  fail s = error s            -- default implementation
```

- (Ignore the `Applicative m` constraint for now.)
- `fail` may have non-default definition for monads representing potentially failing computations; will see relevance later.
- Predefined instances, e.g.:

```
instance Monad Maybe where
  return a = Just a

  Just a  >>= f = f a
  Nothing >>= f = Nothing

  fail s = Nothing
```

A Monad instance for state

- 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` ≈ `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 `==` f v
  2. m `>>= (\a -> return a) ==` m
  3. `(`m `>>=` f`) >>=` g `==` 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.
  - Interpret "==" as "behaves indistinguishably from"
  - Can verify 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  (accessor . 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 (x,y) = (e1,e2) in e3 == e3[x := e1, y := 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 . accessor == 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)
```

  - If `M` satisfies the monad laws, `fmap` will satisfy functor laws.
- GHC 7.10 actually made `Monad` a *subclass* of `Functor`.
  - Must have `instance Functor M` to even be able to declare `instance Monad M`.
  - Old 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 (pure) function:

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

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

```
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)
```

- Exactly same code will work whether `m` represents partial or state-manipulating computations (or any other monad).
- Had we not used monads, we would have to write, for state-passing:

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

- And something very different for an error-passing `pairE`.

Effecful 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.:

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

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

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

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

do-notation

- For increased readability, Haskell offers a convenient notation for writing long monadic computations.
- Syntax: `do` $bind_1$ `;` … $bind_n$ `;` $mexp_0$ $(n \geq 0)$
  - Each $bind_i$ is of form "pat_i `<-` $mexp_i$", "$mexp_i$", or "`let` pat `=` exp_i", where all $mexp_i$ are of monadic type (with same monad).
  - The semicolons a normally replaced by newlines + indentation.
- All uses of `do`-notation are simplified ("desugared", $\rightsquigarrow$) into standard monad operations early in the compilation process.
- First step: bring all `do`'s into form with exactly one $bind$:

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

  $\texttt{do}\ bind_1; bind_2; \ldots; bind_n; mexp_0 \ \rightsquigarrow$

  $\quad \texttt{do}\ bind_1; (\texttt{do}\ bind_2; \ldots; bind_n; mexp_0)$ (use repeatedly while $n \geq 2$)
- Then only have to deal with `do`'s of form "`do` $bind_1$ `;` $mexp_0$".

Desugaring do-notation, continued

- More desugaring rules:

`do` x `<-` $mexp_1$ `;` $mexp_0$	$\rightsquigarrow$	$mexp_1$ `>>= \x ->` $mexp_0$
`do` $mexp_1$ `;` $mexp_0$	$\rightsquigarrow$	$mexp_1$ `>>` $mexp_0$
`do let` pat `=` exp `;` $mexp_0$	$\rightsquigarrow$	`let` pat `=` exp `in` $mexp_0$

- But monadic bindings with (refutable) patterns are a bit special:

  `do` pat `<-` $mexp_1$ `;` $mexp_0$ $\rightsquigarrow$

  ```
  let f pat = mexp0
      f _   = fail "Pattern matching failure at ..."
  in mexp1 >>= f
  ```

  - `fail` is defined in the `Monad`, not necessarily as `error`
  - A bit obscure feature, but used sometimes

- **Example:**

```
do c <- getChar           getChar >>= \c ->
   putStr "Hi"      ~>      putStr "Hi" >>
   return [c]                 return [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`)

Read-only state

- Useful for parameterizing computation with some additional data that will stay constant throughout computation.
- A simplification of the `State` monad constructor:

```haskell
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

```haskell
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 d <- ask; return (d 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 (\d -> \y -> if y == x then n1 else d 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 string 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 erase it:

```haskell
newtype Writer s a = Wr {runWr :: (a, s)}
instance Monoid s => Monad (Writer s) where
  return a = Wr (a, mempty)
  m >>= f = let (a, s1) = runWr m
                (b, s2) = runWr (f a)
            in Wr (b, s1 `mappend` s2)

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

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

Nondeterminsm

- Sometimes want to express that several 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]
  fail s = []
```

- List comprehensions become simply `do`-notation:
  - `[exp |` $qual_1$`, ...,` $qual_n$`]` $\rightsquigarrow$ `do` $qual_1$`; ...;` $qual_n$`; [`*exp*`]`
    - Note: using `[`*exp*`]` instead of `return` *exp* to force list monad.
  - Generators *x* `<-` *lexp* simply kept as monadic bindings.
  - Boolean guards in qualifiers become refutable bindings: *bexp* $\rightsquigarrow$ `True <- [`*bexp*`]`
    - Or: *bexp* $\rightsquigarrow$ `if` *bexp* `then return () else fail ""`

Monadic I/O

- Defined last time:

```
data SimpleIO a = Done a
                | PutChar Char (SimpleIO a)
                | GetChar (Char -> SimpleIO a)
```

- Can organize as a monad:

```
instance Monad SimpleIO where
  return a = Done a
  Done a      >>= f = f a
  PutChar c m >>= f = PutChar c (m >>= f)
  GetChar h   >>= f = GetChar (\c -> h c >>= f)

myPutChar :: Char -> SimpleIO ()
myPutChar c = PutChar c (return ())

myGetChar :: SimpleIO Char
myGetChar = GetChar (\c -> return c)
```

I/O, continued

- Can define further I/O operations by normal monadic sequencing

```
myPutStr :: String -> SimpleIO ()
myPutStr s = mapM_ myPutChar s
  -- using mapM_ :: Monad m => (a -> M b) -> [a] -> m ()
```

- In particular, can check that
  `myPutStr "AP" == PutChar 'A' (PutChar 'P' (Done ()))`

- Can even turn `SimpleIO` computations into “real” `IO`.

```
perform :: SimpleIO a -> IO a
perform (Done a) = return a
perform (PutChar c m) = putChar c >> perform m
perform (GetChar h) = getChar >>= \c -> perform (h c)
```

Combining monads

- Have seen a range of basic monadic effects
  - Maybe 2–3 more are commonly used, of similar complexity.
- But how do we deal with programs that use *multiple* effects?
  - In general, have to create custom-tailored monad for any particular combination.
- **Example:** exceptions and (*persistent*) state

```
newtype ExnState s e a =
  ExSt {runExSt :: s -> (Either e a, s)}
instance Monad (ExnState s e) where
  return a = ExSt (\s -> (Right a, s))
  m >>= f = ExSt (\s -> case runExSt m s of
                          (Left e, s') -> (Left e, s')
                          (Right a, s') -> runExSt (f a) s')

putState :: s -> ExnState s e ()
putState s' = ExSt (\s -> (Right (), s'))
...
```

Combining monads, continued

- Also possible to combine exceptions and state with a *transactional* semantics.
  - State modifications discarded when error signaled.
- Subtle modification of monad type and operations:

```
newtype StateExn s e a = StEx {runStEx:: s -> Either e (a,s)}
instance Monad (StateExn s e) where
  return a = StEx (\s -> Right (a, s))
  m >>= f = StEx (\s -> case runStEx m s of
                          Left e -> Left e
                          Right (a, s') -> runStEx (f a) s')

putState :: s -> StateExn s e ()
putState s' = StEx (\s -> Right ((), s'))
```

- GHC comes with *Monad Transformer Library*, a way of building complex monads out of building blocks.
  - A monad transfomer extends a monad with new features.
  - For *AP*, usually manageable to build combined monad by hand.

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 for this lecture.
  - May want to start with Wadler paper.
- Assignment 0 due tomorrow
  - Do run it past OnlineTA before submitting!
- Start looking at this week's excercises and assignments
  - Assignment 1 due **Wednesday 20/9 at 20:00**
- Next lecture: monads recap + introduction to property-based testing.
  - Recommended readings are up.
  - Look at associated exercises *before* the lecture.
    - E.g., in lab sessions today, 11–12 (DIKU 1-0-{10,14,18}, Lille UP1)