Monads*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin, Christoph Sprenger and Simon Meier for slide material

Monads: What's it all about?

- Model various computational features in a uniform way.
  E.g. partiality, state, exceptions, non-determinism, I/O, ...
- Idea: separate values from computations producing the values:

  `f :: a -> b` ordinary function, returns value of type `b`

  `f :: a -> M b` monadic function, returns computation `M b`

- `M` is a type constructor satisfying certain properties (monad laws).
  By varying `M`, we can model different notions of computation.
- Every monad supports two basic operations: embedding a value into a computation and composing computations.
- Explains side effects in a functional context and helps designing controlled side effects.

Motivation: partial functions

Example: partial functions

- Consider integer division:

```
10 `div` 2 = 5                          -- OK
10 `div` 0 = ..                         -- exception
*** Exception: divide by zero
```

- This partiality can be captured with the Maybe type:

```
data Maybe a = Nothing | Just a

safeDiv :: Int -> Int -> Maybe Int
safeDiv n d
 | d /= 0    = Just (n `div` d)         -- successful computation
 | otherwise = Nothing                  -- failure
```

- A similar construction makes head safe:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing                -- failure
safeHead (x:_) = Just x                 -- successful computation
```

Computing with Maybe

Suppose we are given two Int lists xs and ys.
We would like to safely compute "(head xs) `div` (head ys) + 1".

Direct implementation

```
foo1 :: [Int] -> [Int] -> Maybe Int
foo1 xs ys = case safeHead xs of
  Just a  -> case safeHead ys of
    Just b  -> case safeDiv a b of
      Just c  -> Just (c + 1)
      Nothing -> Nothing
    Nothing -> Nothing
  Nothing -> Nothing
```

Many case distinctions.
Ugly and scales poorly.

Using some Haskell magic

```
foo2 :: ..(same type)..
foo2 xs ys = do
  a <- safeHead xs;
  b <- safeHead ys;
  c <- safeDiv a b;
  return (c + 1)
```

To be explained
here and now!

Composition is the magic

- Key observation is that we would like to compose partial functions.

```
maybe1; maybe2    ⇝    ?
```

- Possible interpretation:

```
Nothing; maybe2    ⇝   Nothing
maybe1;  Nothing   ⇝   Nothing
Just x1; Just x2   ⇝   Just x2
```

- We define `maybe1; maybe2` by `maybe1 ‘semi‘ maybe2` where

```
semi :: Maybe a -> Maybe b -> Maybe b
semi _          Nothing   = Nothing
semi Nothing    _         = Nothing
semi (Just x1) (Just x2) = Just x2
```

- Problem: the computation of `x2` may depend on `x1`.

Composition with value bindings

- Second computation needs to bind result of first.

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing   _ = Nothing
bind (Just x1) f = f x1
```

- We also define a function embedding a value in the `Maybe` type:

```
return :: a -> Maybe a
return x = Just x
```

- Thus we can now write foo2 as

```
foo2 :: [Int] -> [Int] -> Maybe Int
foo2 xs ys =
  safeHead xs `bind` (\a ->
  safeHead ys `bind` (\b ->
  safeDiv a b `bind` (\c ->
  return (c + 1))))
```

The Monad type class

- The Monad typeclass abstractly specifies `bind` and `return`

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b      -- bind
```

- The type constructor `Maybe` instantiates this class.

```
instance Monad Maybe where
  return x       = Just x
  Nothing  >>= _ = Nothing
  (Just x) >>= f = f x
```

- Hence our function foo2 becomes

```
foo2 xs ys =
  safeHead xs >>= (\a ->
  safeHead ys >>= (\b ->
  safeDiv a b >>= (\c ->
  return (c + 1))))
```

```
foo2' xs ys = do
  a <- safeHead xs
  b <- safeHead ys
  c <- safeDiv a b
  return (c + 1)
```

The do-notation is just syntactic sugar to improve readability.

The monad laws

- Monads are mathematical objects with additional properties.
- The monad operations must satisfy the following laws.

```
(1) return x >>= f  = f x                          (left unit)
(2) m >>= return    = m                            (right unit)
(3) (m >>= f) >>= g = m >>= (\x -> (f x >>= g))    (associativity)
```

  These laws enable equational reasoning about monadic programs.

- Exercise: check that these laws hold for the `Maybe` monad.

  Also check this for all other monads in this lecture.

The monad type class — full story
Two additional ingredients

```
class Monad m where

  -- return and bind are the mathematical core
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  -- shortcut for convenience; when second computation
  -- does not depend on result of first
  (>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>= (\_ -> m2)

  -- not part of mathematical concept of a monad
  -- called on pattern matching errors in do-notation
  fail :: String -> m a
```

Input/Output

IO revisited

- `IO` tags expressions that interact with the `Real World`.

```
main :: IO ()                          -- putStrLn :: String -> IO ()
main = do                              -- getLine  :: IO String
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn ("Hi, " ++ name ++ "!")
```

- You can think of `IO a` as an **abstract** datatype

```
data IO a = IO (RealWorld -> (a, RealWorld))
```

- The functions `getLine` and `putStrLn` transform the `RealWorld`.

  Use the monad operators to compose them.

IO's `Monad` instance

The combinator "bind" passes the world around.

Let's pretend `IO` was a type synonym.

```
type IO a =     RealWorld -> (a, RealWorld)
```

```
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  (f >>= g) =     \w -> let (a, w') =      f w in        g a  w'

  return :: a -> IO a
  return x =      \w -> (x, w)

  fail = ...
```

`(>>=)` ensures that each world is **used exactly once**.

IO's `Monad` instance

The combinator "bind" passes the world around.

```
data IO a = IO (RealWorld -> (a, RealWorld))

unIO :: IO a -> RealWorld -> (a, RealWorld)    -- not exported
unIO (IO f) = f

instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  (f >>= g) = IO (\w -> let (a, w') = unIO f w in unIO (g a) w')

  return :: a -> IO a
  return x = IO (\w -> (x, w))

  fail = ...
```

(>>=) ensures that each world is **used exactly once**.

IO monad example

```
data IO a = IO (RealWorld -> (a, RealWorld))
unIO (IO f) = f
(f >>= g) = IO (\w -> let (a, w') = unIO f w in unIO (g a) w')
```

- ```
  main = do
    putStrLn "Enter your name:"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")
  ```

- Desugar to (>>=)

  ```
  main =
    putStrLn "Enter your name:" >>= (_ ->
    getLine >>= (\name ->
    putStrLn ("Hello, " ++ name ++ "!")))
  ```

- Unfold definition of (>>=)

  ```
  main = IO (\w1 ->
    let (_, w2) = unIO (putStrLn "Enter your name:") w1
    in unIO (IO (\w2 ->
       let (name, w3) = unIO getLine w2
       in unIO (putStrLn ("Hello, " ++ name ++ "!")) w3)) w2)
  ```
  ```

IO monad example

```
type IO a =     RealWorld -> (a, RealWorld)

(f >>= g) =     \w -> let (a, w') =      f w in        g a  w'
```

- ```
  main = do
    putStrLn "Enter your name:"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")
  ```

- Desugar to (>>=)

```
main =
  putStrLn "Enter your name:" >>= (_ ->
  getLine >>= (\name ->
  putStrLn ("Hello, " ++ name ++ "!")))
```

- Unfold definition of (>>=)

```
main =      \w1 ->
  let (_, w2) =       putStrLn "Enter your name:"  w1
  in
     let (name, w3) =       getLine w2
     in        putStrLn ("Hello, " ++ name ++ "!")  w3
```
```

More IO actions

- Haskell (see `Prelude.hs`) provides IO primitives
  - `getLine :: IO String`
    Read a line from the keyboard, echo it to the screen and return the string.
  - `putStrLn :: Char -> IO ()`
    Write a string and a newline to the screen.
  - `readFile :: FilePath -> IO String`
    Read a file and return contents as a lazy string.
  - `writeFile :: FilePath -> String -> IO ()`
    Write a string to a file.
- The Haskell libraries provide many others (sockets, concurrency, . . .)

Stateful computation

Example: collecting type inference constraints

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \to \tau}\ \textit{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \textit{App}$$

1. Start with judgement $\vdash t :: \tau_0$ with type variable τ_0.
2. Build derivation tree bottom-up by applying the matching rule. Introduce fresh type variables and collect constraints if needed.

$$\frac{\dfrac{\dfrac{}{z : \tau_2 \vdash z :: \tau_3}\ \textit{Var}}{\vdash \lambda z.\, z :: \tau_1 \to \tau_0}\ \textit{Abs} \qquad \dfrac{\dfrac{}{x : \tau_4 \vdash x :: \tau_5}\ \textit{Var}}{\vdash \lambda x.\, x :: \tau_1}\ \textit{Abs}}{\vdash (\lambda z.\, z)\ (\lambda x.\, x) :: \tau_0}\ \textit{App}$$

$$\begin{aligned} \tau_2 \to \tau_3 &= \tau_1 \to \tau_0 \\ \tau_2 &= \tau_3 \\ \tau_1 &= \tau_4 \to \tau_5 \\ \tau_4 &= \tau_5 \end{aligned}$$

Idea: use an accumulator to keep track of used indices

Type inference: fixing the types

- Terms are built from variables, abstractions, and applications.

```
type Variable = String
data Term     = Var Variable
              | Lam Variable Term
              | App Term Term

data Type     = TVar Int
              | Fun Type Type
```

- Model constraints as a pair of left- and right-hand side.

```
type Constraint = (Type, Type)
collectConstraints :: Term -> [Constraint]
```

- It suffices to store the highest used index in the accumulator.

```
type Context = [(Variable, Type)]
collect :: Context -> Term -> Type -> Int -> ([Constraint], Int)
```

Implementing constraint collection

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \mathit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \to \tau}\ \mathit{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \mathit{App}$$

```
collectConstraints :: Term -> [Constraint]
collectConstraints t = collect [] t (TVar 0) 0
```
Start with $\vdash t :: \tau_0$

```
collect :: Context -> Term -> Type -> Int -> ([Constraint], Int)
collect gamma (Var x) ty n =
  case lookup gamma x of                          -- lookup defined in Prelude
    Just tyX -> ([(tyX, ty)], n)                      add equality constraint

collect gamma (Lam x t) ty n =
  let sigma    = TVar (n + 1)                         generate fresh
      tau      = TVar (n + 2)                         type variables
      (cs, n') = collect ((x, sigma) : gamma) t tau (n + 2)
  in ((ty, Fun sigma tau) : cs, n')                   add equality constraint

collect gamma (App t1 t2) ty n =
  let sigma     = TVar (n + 1)                        fresh type variable σ
      (cs1, n1) = collect gamma t1 (Fun sigma ty) (n + 1)
      (cs2, n2) = collect gamma t2 sigma n1
  in (cs1 ++ cs2, n2)
```

Example: collecting type inference constraints

```
type Variable   = String
data Term       = Var Variable | Lam Variable Term | App Term Term
data Type       = TVar Int     | Fun Type Type
type Context    = [(Variable, Type)]
type Constraint = (Type, Type)

collectConstraints :: Term -> [Constraint]
collectConstraints t = collect [] t (TVar 0) 0

collect :: Context -> Term -> Type -> Int -> ([Constraint], Int)
collect gamma (Var x) ty n =
  case lookup gamma x of Just tyX -> ([(tyX, ty)], n)
collect gamma (Lam x t) ty n =
  let sigma    = TVar (n + 1)
      tau      = TVar (n + 2)
      (cs, n') = collect ((x, sigma) : gamma) t tau (n + 2)
  in ((ty, Fun sigma tau) : cs, n')
collect gamma (App t1 t2) ty n =
  let sigma     = TVar (n + 1)
      (cs1, n1) = collect gamma t1 (Fun sigma ty) (n + 1)
      (cs2, n2) = collect gamma t2 sigma n1
  in (cs1 ++ cs2, n2)
```

Ugly plumbing needed to thread state through recursive calls.

Constructing the state monad

Type constructor for stateful computations

```
data State s a = State (s -> (a, s))
```

Idea: computation takes a state of type `s` and transforms it into a result of type `a` and a successor state of type `s`.

Compare with `IO`:

```
data IO a = IO (RealWorld -> (a, RealWorld))
```

State access read current value of state without changing it

```
get :: State s s
get = State (\s -> (s, s))
```

State update write a new state value, ignoring the current state

```
put :: s -> State s ()
put t = State (\s -> ((), t))
```

Return and bind

Run is an auxiliary function that opens the monad and runs the computation from the initial state s0

```
runState :: (State s a) -> s -> (a, s)
runState (State m) s0 = m s0
```

Return embeds a value into a stateful computation

```
return :: a -> State s a
return x = State (\s -> (x, s))
```

Bind composes two stateful computations with value binding

```
(>>=) :: State s a -> (a -> State s b) -> State s b
m >>= k = State (\s -> let (x, t) = runState m s
                       in runState (k x) t)
```

Note: The operator (>>) defined as `m1 >> m2 = m1 >>= (\_ -> m2)` is essentially the sequential composition (;) in imperative programming languages.

Understanding the state monad

- `x := x + 1` in state monad

```
tick :: State Int ()
tick = do
  x <- get
  put (x + 1)
```

with explicit bind

```
tick :: State Int ()
tick =
  get >>= (\x ->
  put (x + 1))
```

- Stepwise evaluation of `tick`

```
tick = State (\s ->
         let (x, t) = runState get s
         in runState (put (x + 1)) t)

     = State (\s ->
         let (x, t) = (\s -> (s, s)) s
         in (\s -> ((), x + 1)) t)

     = State (\s -> ((), s + 1))
```

  - The state monad encapsulates program composition.
  - To run the program: invoke `runState tick s0` where `s0` is some initial state.

Constraint collection using the state monad

```
freshTVar = do
  n <- get
  let n' = n + 1
  put n'
  return (TVar n')
```

We only ever need to get a fresh type variable. Thanks to the state monad, we can encapsulate this in an operation.

```
collectConstraints :: Term -> [Constraint]
collectConstraints t = fst (runState (collect [] t (TVar 0)) 0)

collect :: Context -> Term -> Type -> State Int [Constraint]
collect gamma (Var x) ty =
  case lookup gamma x of Just tyX -> return [(tyX, ty)]
collect gamma (Lam x t) ty = do
  sigma <- freshTVar
  tau   <- freshTVar
  cs    <- collect ((x, sigma) : gamma) t tau
  return ((ty, Fun sigma tau) : cs)
collect gamma (App t1 t2) ty = do
  sigma <- freshTVar
  cs1   <- collect gamma t1 (Fun sigma ty)
  cs2   <- collect gamma t2 sigma
  return (cs1 ++ cs2)
```

A zoo of monads

Monad	Type constructor
Partiality	`Maybe a = Nothing | Just a`
Exceptions	`Exc e a = Exception e | Success a`
State	`State s a = State (s -> (a, s))`
Input/Output	`IO a = IO (RealWorld -> (a, RealWorld))`
Nondeterminism	`[a] = [] | a : [a]`
Parsers	`Parser a = Parser (String -> [(a, String)])`
⋮	⋮

Conclusions

Summary Using monads we can ...

- write functional programs with a variety of controlled side effects in a uniform, abstract, and flexible way
- obtain a deeper understanding of the meaning of side effects

Combining monads

- Q: How can I model language AFX (All Fancy effeXts)?
- A: Use monad transformers to modularly combine monads. E.g., the parser monad is a non-deterministic state monad, we can also define state-exception monads, etc.

Reasoning about monads two possibilities:

- equational reasoning using definitions of monadic functions and monad laws (verify these for any monads you may invent), or
- pre-/post-condition reasoning using a monadic Hoare logic

Bibliography

- Monad tutorials recommended at http://www.haskell.org/haskellwiki/Tutorials#Using_monads
- Philip Wadler, *The essence of functional programming*, POPL 92, 1992. [Nice series of interpreters with monadic effects.]
- Sheng Liang, Paul Hudak, and Mark Jones, *Monad transformers and modular interpreters*, POPL 95, 1995. [Series of interpreters obtained in a modular fashion using monad transformers. This goes beyond this course, but is very readable.]
- Nick Benton, John Hughes, and Eugenio Moggi, *Monads and Effects*, 2002. [Covers both theoretical aspects and programming.]