Monads*

David Basin

Department of Computer Science
ETH Zurich

*Thanks to 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 the composing computations.

- Explains side-effects in a functional context and helps designing controlled side effects.

Outline

Part I – A gentle introduction to monads by examples

- Partial functions
- Monad type class and monad laws
- Input/output
- Stateful computations

Part II – Case study: monadic interpreters

- Standard and monadic interpreter for mini-Haskell
- Variant 1: improved error handling
- Variant 2: counting the number of evaluation steps
- Variant 3: non-deterministic computation
- Variant 4: tracing intermediate results

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)
 | otherwise = Nothing
```

- A similar construction makes head safe:

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

Computing with Maybe's

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

Direct implementation	Using some Haskell magic

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

Many case distinctions. Ugly and scales poorly.	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 – The 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 dependent on result of first one
  (>>) :: 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

Why is IO problematic?

- How would we write a program like the following in Haskell?

```
void main () {
  char name[20];
  printf ("Hi, I am HAL.  Who are you?");
  scanf ("%19s", name);
  printf ("Hello %s!", name);
}
```

- Assume there would be functions like `inputInt :: Int` in Haskell
  - What is the value of `inputInt - inputInt`?
  - Equational reasoning would no longer be sound
  - Result depends on order in which the arguments are evaluated
- This function is not side-effect free! So which state changes?

The IO type constructor

- Haskell uses a monad to distinguish between pure expressions and expressions that interact with The World.

  `IO a` type of computations performing I/O operations and returning a value of type `a`

- Examples:

```
inputInt    :: IO Int
inputString :: IO String
outputInt   :: Int -> IO ()   -- () is the unit type in Haskell
```

- You can think of `IO a` as the Haskell type

```
data IO a = InOut (TheWorld -> (a, TheWorld))
```

Basic actions

- Haskell (see `Prelude.hs`) provides IO primitives
  - `getChar :: IO Char`
    The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value.
  - `putChar :: Char -> IO ()`
    The action `putChar c` writes the character `c` to the screen and returns no result value.
  - `return :: a -> IO a`
    The action `return v` simply returns the value `v`, without performing any interaction.
- . . . and many others (reading & writing files, etc.)

Sequencing

- The order of the actions matters

```
read2 = do c1 <- getChar
           c2 <- getChar
           return (c1:c2:[])
```

```
read2' = do c2 <- getChar
            c1 <- getChar
            return (c1:c2:[])
```

What is the type of `read2`?

- Previous lecture: order of applying parsers also matters

```
pexpr = do token "("
           e <- expr
           token ")"
           return e
```

```
pexpr' = do token "("
            token ")"
            e <- expr
            return e
```

- The IO type constructor cannot be "opened". Hence, any function doing I/O will have range `IO a` for some type a.

- For example, `inputInt - inputInt` is incorrect. Why?

IO examples

- Printing a string on the screen:

```
putString :: String -> IO ()
putString ""     = return ()
putString (x:xs) = do putChar x; putString xs
```

- Reading a string from the keyboard:

```
getString :: IO String
getString = do c <- getChar
               if c == '\n'
                 then return ""
                 else do cs <- getString
                         return (c:cs)
```

- A "hello world" program in Haskell:

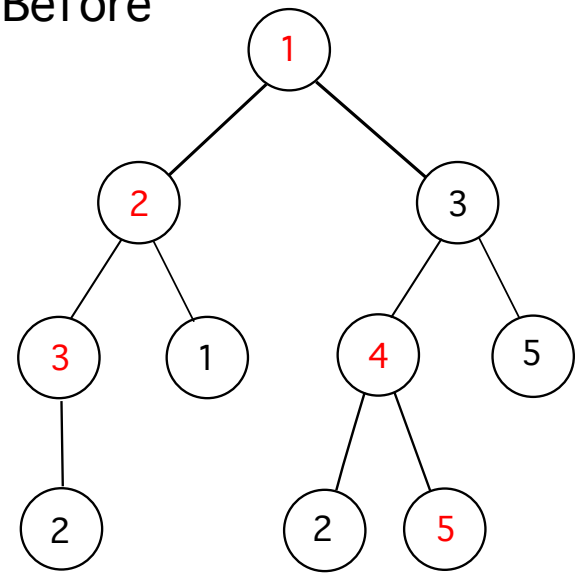
```
main :: IO ()
main = do putString "Hi, I am HAL. Who are you?\n"
          name <- getString
          putString ("Hello " ++ name ++ "!\n")
```

Stateful computation

Example: renaming of tree nodes

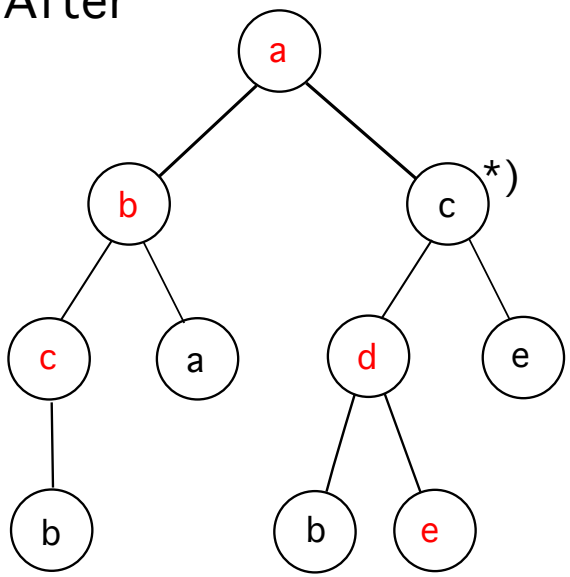
- We want to consistently rename tree nodes in preorder fashion.
- New names are given as a list that is assumed to be long enough.
- Idea: Use an accumulator to keep track of two things: the list of remaining names and a table of current name translations.

Before



names = [a, b, c, d, e]
red indicates first occurrence (in preorder)

After



*) When reaching this node the accumulator contains ([d, e], [(1, a), (2, b), (3, c)])

Implementation of tree node renaming

The type of accumulator (names, table) is `([b],[(a,b)])`.

```
rename :: Eq a => [b] -> Tree a -> Tree b
rename names = fst . go (names, [])

go :: Eq a => ([b],[(a,b)]) -> Tree a -> (Tree b, ([b],[(a,b)]))
go state           Leaf          = (Leaf, state)
go (names, table) (Node x l r) =
  case lookup x table of
    Nothing -> let x'                = head names
                   state'            = (tail names, (x, x'):table)
                   (l', state'')     = go state'  l
                   (r', state''')    = go state'' r
               in (Node x' l' r', state''')

    Just x' -> let (l', state')  = go state  l
                   (r', state'') = go state' r
               in (Node x' l' r', state'')
```

Ugly plumbing needed to thread state through two 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.

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 binding

```
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.

Tree renaming using the state monad

```
rename :: Eq a => [b] -> Tree a -> Tree b
rename names t = fst $ runState (renameTree t) (names, [])

renameTree :: Eq a => Tree a -> State ([b],[(a,b)]) (Tree b)
renameTree Leaf = return Leaf
renameTree (Node x l r) = do
  (names, table) <- get
  case lookup x table of
    Nothing -> do
      let x' = head names
      put (tail names, (x, x'):table)
      l' <- renameTree l
      r' <- renameTree r
      return (Node x' l' r')
    Just x' -> do
      l' <- renameTree l
      r' <- renameTree r
      return (Node x' l' r')
```

The state monad takes care of all the plumbing!

Tree renaming the way you want it

- Renaming can be made a bliss ...

```
renameTree' :: Eq a => Tree a -> State ([b],[(a,b)]) (Tree b)
renameTree' Leaf = return Leaf
renameTree' (Node x l r) = do
  x' <- translate x
  l' <- renameTree' l
  r' <- renameTree' r
  return (Node x' l' r')
```

- ... by abstracting the pattern of looking up a translation

```
translate :: Eq a => a -> State ([b],[(a,b)]) b
translate x = do
  (names, table) <- get
  case lookup x table of
    Nothing -> do
      let x' = head names
      put (tail names, (x, x'):table)
      return x'
    Just x' -> do
      return x'
```

Summary (Part I)

- Monads are a powerful concept which helps understanding and modeling computations with side effects.
- Contrary to imperative languages, where side effects are the rule, monads promote the use of side effects in a controlled way (you usually have a good reason to use a monad).
- Construct, combine, and use monads that exactly fit the structure of your problem (fine-grained control of side effects).
- New monads can model computational effects that are not present in imperative languages (e.g., non-determinism, continuations)
- Monadic computations are first-class values that can be composed as needed.

Case study: Monadic Interpreters

Language: A variant of mini-Haskell

- Language we consider here:

$$\begin{aligned} Term \quad ::= \quad & Identifier \mid Number \mid \\ & \lambda x.\, Term \mid Term \;\; Term \mid Term + Term \end{aligned}$$

  This core language could be extended with other arithmetic operations, predicates, if-then-else, recursion, ...

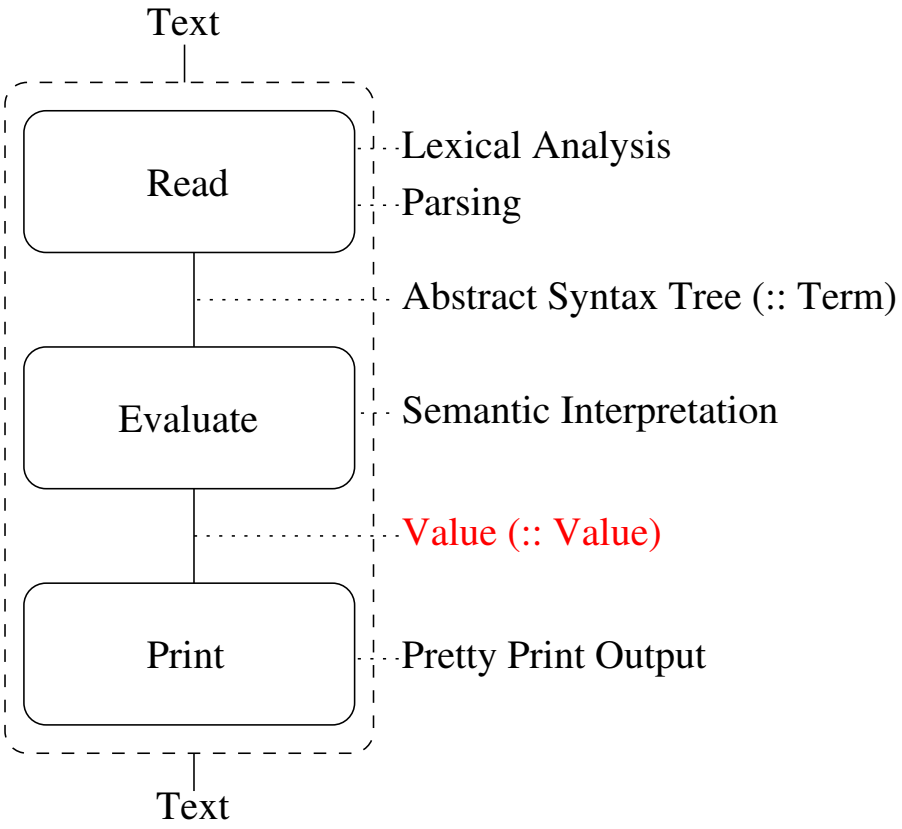
- Data types for syntax:

```
type Name = String
data Term = Var Name
          | Lit Int
          | Lam Name Term
          | App Term Term
          | Add Term Term
```

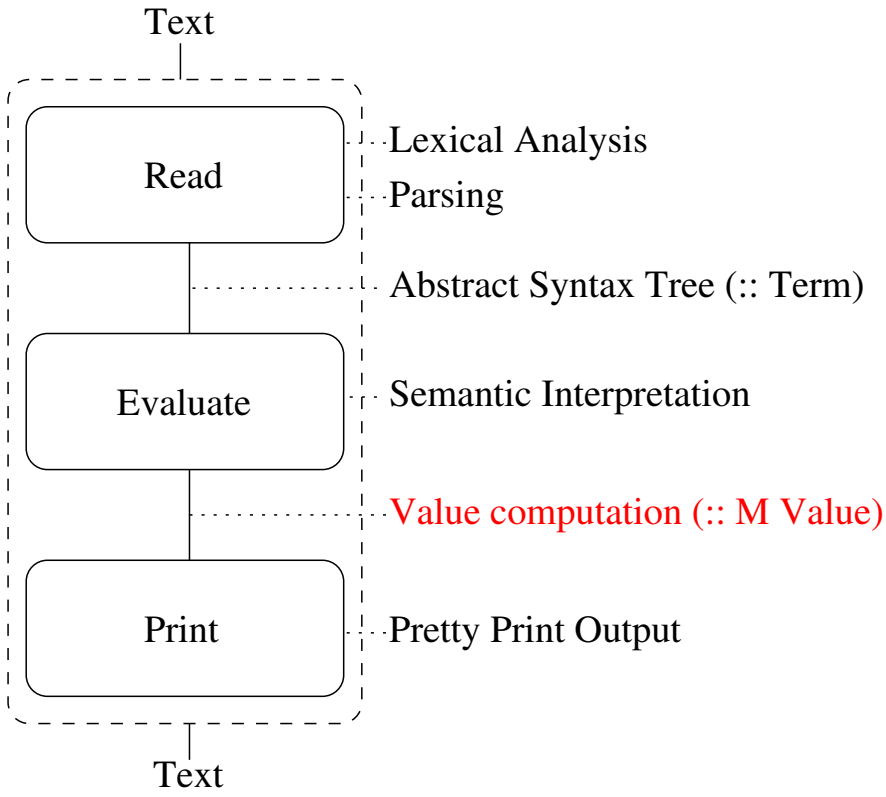
- Only consider evaluation, for parsing see previous module.

Standard and Monadic Interpreters

Standard Interpreters



Monadic interpreters



- for calculuator in previous lecture: $Value = Int$;
- for λ-calculus interpreter: $Value = Term$.

Standard interpreter

- Comparison with λ-interpreter from previous lecture:
  - eager evaluation: evaluate function arguments and under λ's
  - evaluate only closed terms, as is usual in programming
  - use Haskell's substitution instead of implementing it ourselves

- Output: structured type of values

```
data Value = Wrong                      -- error
           | Num Int                    -- integer
           | Fun (Value -> Value)       -- closure
```

  Error + λ-abstractions evaluate to closures.

- An environment binds free variables to values

```
type Environment = [(Name, Value)]
```

Auxiliary functions

```
data Value = Wrong                      -- error
           | Num Int                    -- integer
           | Fun (Value -> Value)       -- closure

type Environment = [(Name, Value)]


lookup  :: Name -> Environment -> Value
lookup x []           = Wrong
lookup x ((y, v):es) = if x == y then v else lookup x es

add :: Value -> Value -> Value
add (Num x) (Num y) = Num (x + y)
add _       _       = Wrong

apply :: Value -> Value -> Value
apply (Fun k) a = k a
apply _       _ = Wrong
```

Standard Evaluation

```
data Value = Wrong                      -- error
           | Num Int                    -- integer
           | Fun (Value -> Value)       -- closure

type Environment = [(Name, Value)]


eval  :: Term -> Environment -> Value
eval (Var v)   e = lookup v e
eval (Lit x)   e = Num x
eval (Lam x t) e = Fun (\a -> eval t ((x, a):e))
eval (App t u) e = apply (eval t e) (eval u e)
eval (Add t u) e = add (eval t e) (eval u e)

run :: Term -> Value
run t = eval t []

instance Show Value where
  show Wrong   = "<wrong>"
  show (Num i) = show i
  show (Fun _) = "<function>"
```

Standard interpreter – Examples

- Consider the terms (in abstract syntax)

$$\begin{array}{lcll} t_0 & = & (\lambda x.\, x + x)\ (10 + 11) & \hookrightarrow \quad 42 \\ t_1 & = & (\lambda x.\, x) + 12 & \hookrightarrow \quad \text{error} \\ t_2 & = & \lambda x.\, x + 1 & \hookrightarrow \quad \text{function} \end{array}$$

- In concrete syntax:

```
t0 = App (Lam "x" (Add (Var "x") (Var "x"))) (Add (Lit 10) (Lit 11))
t1 = Add (Lam "x" (Var "x")) (Lit 10)
t2 = Lam "x" (Add (Var "x")) (Lit 1)
```

- Evaluation in ghci:

```
*Main> run t0
42

*Main> run t1
<wrong>

*Main> run t2
<function>
```

Example: $(\lambda x.x)(1+2)$

```
run (App (Lam "x" (Var "x")) (Add (Lit 1) (Lit 2)))

  = eval (App (Lam "x" (Var "x")) (Add (Lit 1) (Lit 2))) []

  = apply (eval (Lam "x" (Var "x")) []) (eval (Add (Lit 1) (Lit 2)) [])

  = apply (Fun (\a -> eval (Var "x") [(x,a)])
           (add (eval (Lit 1)[]) (eval (Lit 2) []))

  = (\a -> eval (Var "x") [("x",a)]) (add (Num 1) (Num 2))

  = eval (Var "x") [("x",(add (Num 1) (Num 2)))]

  = lookup "x" [("x",(add (Num 1) (Num 2)))]

  = add (Num 1) (Num 2)

  = Num 3
```

N.B. Multiple reductions compressed.

Result `Num 3` is output (with `show`) as 3.

Monadic interpreter – Ideas

- Replace all functions with result type `Value` by functions with monadic result type `M Value`

- Values and function signatures:

```
data Value = Wrong                          -- error
           | Num Int                        -- integer
           | Fun (Value → M Value)          -- closure

eval        ::  Term → Environment → M Value
run         ::  Term → M Value

lookup      ::  Name → Environment → M Value
apply, add  ::  Value → Value → M Value
```

- **Win**: By varying the definition of the monad `M`, we obtain different computational effects. (Different defs. upcoming!)

Monadic interpreter – Auxiliary functions

```
data Value = Wrong
           | Num Int
           | Fun (Value -> M Value)

type Environment = [(Name, Value)]

lookup :: Name -> Environment -> M Value
lookup x []           = return Wrong
lookup x ((y, v):es) = if x == y then return v else lookup x es

apply :: Value -> Value -> M Value
apply (Fun k) a = k a
apply _       _ = return Wrong

add :: Value -> Value -> M Value
add (Num x) (Num y) = return (Num (x + y))
add _       _       = return Wrong

instance Show (M Value) =
  show m = ...                    -- depends on monad M
```

Monadic interpreter – Evaluation

```
data Value = Wrong
           | Num Int
           | Fun (Value -> M Value)

type Environment = [(Name, Value)]

eval ::  Term -> Environment -> M Value
eval (Var v)   e = lookup v e
eval (Lit x)   e = return (Num x)
eval (Lam x t) e = return (Fun (\a -> eval t ((x, a):e)))
eval (App t u) e = do f <- eval t e
                      a <- eval u e
                      apply f a
eval (Add t u) e = do a <- eval t e
                      b <- eval u e
                      add a b

run ::  Term -> M Value
run t = eval t []
```

Monadic interpreter – Instances

- Recall: by varying the monad `M` we obtain interpreters with different computational effects.

- We consider the following instances of the monad `M`:
  - Identity monad: recover standard interpreter
  - Exception monad: improved error handling
  - State monad: count number of evaluation steps
  - Nondeterministic monad: compute with choices
  - Output monad: output intermediate results

- Abstraction is main benefit of using monads: Only small changes are necessary in each case, basic structure remains the same.

Identity monad: Standard interpreter

- The identity monad:

```
data Id a = Id a

instance Monad Id where
  return x      = Id x          -- identity function
  (Id m) >>= k = k m            -- function application
```

- For example, forgetting about the constructor `Id`, the clauses

```
eval (Lit i)   e = return (Num i)
eval (App t u) e = eval t e >>= (\f ->
                   eval u e >>= (\a ->
                   apply f a))
```

simplify to

```
eval (Lit i)   e = Num i
eval (App t u) e = apply (eval t e) (eval u e)
```

Improving error handling

- Current solution has a only single error message: `<wrong>`
- First attempt to improve situation (in standard interpreter):

```
data Value = Wrong String
           | Num Int
           | Fun (Value -> Value)
```

- Specify source of error as argument to `Wrong`:

```
lookup x [] = Wrong ("Unbound variable: " ++ x)
apply  v _  = Wrong ("Not a function: " ++ show v)
add    v w  = Wrong ("Not a number: " ++ show v
                     ++ " or " ++ show w)
```

- Does not behave as intended:

```
*Main> run (App (Var "x") (Lit 10))
Not a function: Unbound variable: x
```

Exception Monad

- Type constructor and basic monad operations:

```
data Exc a = Success a
           | Exception String

instance Monad Exc where
  return x             = Success x
  (Success a)   >>= k = k a              -- on success: continue
  (Exception e) >>= k = Exception e      -- on exception: abort
```

- Monad-specific operations: throw and catch exceptions

```
throw :: String -> Exc a
throw e = Exception e                    -- raise exception

catch :: Exc a -> (a -> Exc a) -> Exc a
catch (Success a)   h = (Success a)  -- normal execution
catch (Exception e) h = h e          -- call exception handler
```

- Straightforward function `show` shows value or exception.

Adapting the interpreter

- Modification of data structures and interpreter

```
data Value = Num Int                          -- removed: Wrong
           | Fun (Value → ExcM Value)

lookup x [] = throw ("Unbound variable: "++x)
apply  v _  = throw ("Not a function: "++show v)
add    v w  = throw ("Not a number: "++show v++" or "++show w)
```

- Examples: (using abstract syntax)

```
*Main> run (λx. x + x) (10 + 11)
42

*Main> run (x 10)
Unbound variable: x

*Main> run (λx. x) + 10
Not a number: <function> or 10

*Main> run 99 (λx. x)
Not a function: 99
```

State monad: Counting evaluation steps

- The state monad (as seen before):

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

instance Monad (State s) where
  return x = State (\s -> (x, s))
  m >>= k  = State (\s -> let (a, t) = runState m s
                          in runState (k a) t)
```

- Monad-specific operations for state manipulation:

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

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

- Specific to application: step counting

```
tick :: State Int ()
tick = do s <- get; put (s + 1)         -- increment counter
```

Adapting the interpreter

- We add `tick`'s to addition (and similarly to application):

```
eval (Add t u) e = do a <- eval t e
                      b <- eval u e
                      r <- add a b
                      tick              -- count addition
                      return r
```

- The show function runs the monad with counter initialized to 0.

```
instance Show (State Int Value) where
  show m = let (a, c) = runState m 0
           in "Value:  " ++ show a ++ "; " ++
              "Count:  " ++ show c ++ "."
```

- Examples:

```
*Main> run (λx. x + x) (10 + 11)
Value 42; Count 3.

*Main> run (λx. x + 1) ((λx. x + x) (10 + 11))
Value 43; Count 5.
```

Nondeterministic monad: Allowing choices

- The nondeterministic monad (aka list monad):

```
data Alt a = Alt [a]

runAlt :: Alt a -> [a]
runAlt (Alt l) = l

instance Monad Alt where
  return a = Alt [a]
  m >>= k  = Alt [b | a <- runAlt m, b <- runAlt (k a)]
```

  Idea: computation may produce several possible results.

- We also need the following monad-specific operations.

```
failure :: Alt a
failure = Alt []

choice :: Alt a -> Alt a -> Alt a
choice xs ys = Alt (runAlt xs ++ runAlt ys)
```

Adapting the interpreter

- Extend the term language with failure and choice operations:

```
data Term = ...
          | Fail              -- written ⊥ below
          | Or Term Term      -- (Or t u) written t | u below
```

- The interpreter is extended as follows:

```
eval Fail      e = failure
eval (Or t u)  e = eval t e `choice` eval u e
```

- Examples:

```
*Main> run ((λx. x + x) (10 + 11))
[42]

*Main> run ((λx y. x + x + y) (10 | 20) (1 | 5))
[21,25,41,45]

*Main> run ((7 | 10) + ⊥)
[]
```

Output Monad: Tracing intermediate results

- Output monad chains computations and concatenates output:

```
data Out a = Out (a, String)

runOut :: Out a -> (a, String)
runOut (Out x) = x

instance Monad Out where
  return x = (x, "")
  m >>= k  = let (a, r) = runOut m; (b, s) = runOut k a
             in Out (b, r ++ s)
```

- Monad-specific output function:

```
out :: Show a => a -> OutM ()
out a = ((), show a ++ "; ")
```

- Show monad value:

```
instance Show a => Show (OutM a) where
  show (Out (a, s)) = "Output:  " ++ s ++
                      "Value:  " ++ show a
```

Adapting the interpreter for output tracing

- Extend the term language with a Show operation:

```
Term = ...
     | Show Term          -- Show t written #t below
```

- Extension of the interpreter:

```
eval (Show t) e = do r <- eval t e     -- evaluate t
                     out r             -- display its value
                     return r          -- and return it
```

- Examples:

```
*Main> run ((λx. x + x) #(10 + 11))
Output: 21; Value: 42

*Main> run (#(#(1 + 2) + 3) + 4)
Output: 3; 6; Value: 10

*Main> run (1 + #(2 + #(3 + 4)))
Output: 7; 9; Value: 10
```

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 =` `  InOut (TheWorld-> (a, TheWorld))`
Nondeterminism	`Alt a = Alt [a]`
Output	`Out a = Out (a,String)`
Parsers	`Parser a =` `  Parser (String -> [(a, String)])`

(*) The definition of the Haskell IO monad is a conceptual one.

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.]