301AA - Advanced Programming

Lecturer: Andrea Corradini

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

AP-20: Constructor Classes and Monads in Haskell

Summary

- Type Constructor Classes
- Functor and fmap
- Towards monads: Maybe and partial functions
- Monads as containers and as computations
- Introducing side effects with the IO monad
- Control structures on monads

Type Constructor Classes

- Type Classes are predicates over types ≡
- Allow to define overloaded functions common to several type constructors
- Example: map function useful on many Haskell types
 Lists:

```
map:: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

> map (\x->x+1) [1,2,4]
[2,3,5]
```

More examples of map function

```
data Tree a = Leaf a | Node(Tree a, Tree a)
    deriving Show

mapTree :: (a -> b) -> Tree a -> Tree b

mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node(1,r)) = Node (mapTree f 1, mapTree f r)

> t1 = Node(Node(Leaf 3, Leaf 4), Leaf 5)
> mapTree (\x->x+1) t1
Node (Node (Leaf 4, Leaf 5), Leaf 6)
```

```
data Maybe a = Nothing | Just a
  deriving Show 
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just x) = Just (f x)

> o1 = Just 10
> mapMaybe (\x->x+1) o1
Just 11
```

Constructor Classes

All map functions share the same structure

```
map :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
```

They can all be written as:

```
fmap:: (a -> b) -> g a -> g b
```

- where g is: ≡
 - [-] for lists, Tree for trees, and Maybe for options
- Note that g is a function from types to types, i.e.
 a type constructor ≡

Constructor Classes

 This pattern can be captured in a constructor class Functor: =

```
class Functor g where
fmap :: (a -> b) -> g a -> g b
```

- Simply a type class where the predicate is over a type constructors rather than on a type
- Compare with the definition of a standard type class:

```
class Eq a where
  (==) :: a -> a -> Bool
```

The Functor constructor class and some instances =

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node(t1,t2)) = Node(fmap f t1, fmap f t2)
instance Functor Maybe where
  fmap f (Just s) = Just(f s)
  fmap f Nothing = Nothing
```

The Functor constructor class and some instances (2)

Or by reusing the definitions map, mapTree, and mapMaybe:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap = mapTree

instance Functor Maybe where
  fmap = mapMaybe
```

Constructor Classes

 We can then use the overloaded symbol fmap to map over all three kinds of data structures:

```
*Main> fmap (\x->x+1) [1,2,3]
[2,3,4]
it :: [Integer]

*Main> fmap (\x->x+1) (Node(Leaf 1, Leaf 2))
Node (Leaf 2, Leaf 3)
it :: Tree Integer

*Main> fmap (\x->x+1) (Just 1)
Just 2
it :: Maybe Integer
```

 The Functor constructor class is part of the standard Prelude for Haskell

■ Towards Monads: The Maybe type constructor

Towards Monads

- Often type constructors can be thought of as defining "boxes" for values \(\box{\omega}\)
- Functors with fmap allow to apply functions inside "boxes"
- Monads are constructor classes introducing operations for
 - Putting a value into a "box" (return)
 - Compose functions that return "boxed" values (bind)

The Maybe type constructor

 Type constructor: a generic type with one or more type variables

```
data Maybe a = Nothing | Just a
```

- A value of type Maybe a is a possibly undefined value of type a
- A function f :: a -> Maybe b is a partial function from a to b ≡

Composing partial function

```
bothGrandfathers :: Person -> Maybe (Person, Person)
   bothGrandfathers p =
        case father p of
           Nothing -> Nothing
           Just dad ->
               case father dad of
                   Nothing -> Nothing
                    Just qf1 ->
                                                       -- found first grandfather
                       case mother p of
                           Nothing -> Nothing
                           Just mom ->
                               case father mom of
                                   Nothing -> Nothing
                                   Just qf2 -> -- found second grandfather
                                       Just (qf1, qf2)
```

Composing partial functions

 We introduce a higher order operator to compose partial functions in order to "propagate" undefinedness automatically

• The *bind* operator will be part of the definition of a *monad*.

Use of **bind** of the **Maybe** monad to compose partial functions

```
father :: Person -> Maybe Person -- partial function
mother :: Person -> Maybe Person -- (lookup in a DB)

maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
    case mother p of
    Nothing -> Nothing
    Just mom -> father mom
```



The Monad Constructor Class and the Maybe monad

```
class Monad m where =
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    ... -- + something more
```

- m is a type constructor
- m a is the type of monadic values \(\equiv \)

• bind (>>=) shows how to "propagate" undefinedness

Alternative, imperative-style syntax: do

```
bothGrandfathers p =
      father p >>=
           (\dad -> father dad >>=
               (\gf1 -> mother p >>=
                    (\mom -> father mom >>=
                        (\gf2 -> return (gf1,gf2
bothGrandfathers p = do {
                               bothGrandfathers p = do
     dad <- father p;</pre>
                                       dad <- father p
       gf1 <- father dad;</pre>
                                       qf1 <- father dad
                                       mom <- mother p
       mom <- mother p;</pre>
       gf2 <- father mom; □
                                       gf2 <- father mom
                                       return (gf1, gf2)
       return (gf1, gf2);
```

do syntax is just syntactic sugar for >>= ≡

Monad	Imperative semantics
Maybe	Exception (Anonymous) 틎
Error	Exception (with error description)
State	Global state
Ю	Input/output
[] (lists)	Non-determinism
Reader	Environment
Writer	Logger

Understanding Monads as containers

```
class Monad m where -- definition of Monad type class
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    ... -- + something more + a few axioms
```

- The monadic constructor can be seen as a container: let's see this for lists
- Getting bind from more basic operations

Understanding Monads as computations

```
class Monad m where -- definition of Monad type class
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    (>>) :: m a -> m b -> m b -- "then"
    ... -- + something more + a few axioms
```

- A value of type m a is a computation returning a value of type a ≡
- For any value, there is a computation which "does nothing" and produces that result. This is given by function return
- Given two computations x and y, one can form the computation
 x >> y which intuitively "runs" x, throws away its result, then runs y returning its result
- Given computation x, we can use its result to decide what to do next. Given f: a → m b, computation x >>= f runs x, then applies f to its result, and runs the resulting computation. ≡

```
Note that we can define then using bind:

x >> y = x >>= (\_ -> y)
```

Understanding Monads as computations (2)

```
class Monad m where -- definition of Monad type class
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b -- "bind"
    (>>) :: m a -> m b -> m b -- "then"
    ... -- + something more + a few axioms
```

- return, bind and then define basic ways to compose computations
- They are used in Haskell libraries to define more complex composition operators and control structures (sequence, for-each loops, ...)
- If a type constructor defining a library of computations is monadic, one gets automatically benefit of such libraries

Example: MAYBE

- f:a -> Maybe b is a partial function
- bind applies a partial function to a possibly undefined value, propagating undefinedness

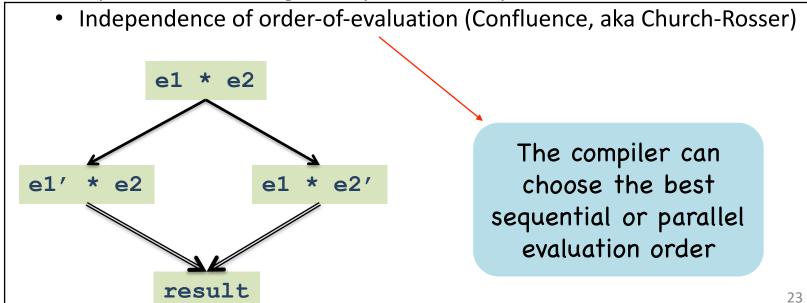
Example: LISTS

- **f**:a -> [b] is a *non-deterministic* function
- bind applies a non-deterministic function to a list of values, collecting all possible results

Contaminating Haskell with side effects: Towards the IO monad

Pros of Functional Programming

- Functional programming is beautiful:
 - Concise and powerful abstractions
 - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
 - Close correspondence with mathematics
 - Semantics of a code function is the mathematical function
 - Equational reasoning: if x = y, then f x = f y



Problems...

- But to be useful, a language must be able to manage "impure features":
 - Input/Output
 - Imperative update
 - Error recovery (eg, timeout, divide by zero, etc.)
 - Foreign-language interfaces
 - Concurrency control

The whole point of a running a program is to interact with the external environment and affect it

The Direct Approach

- Just add imperative constructs "the usual way"
 - I/O via "functions" with side effects:

```
putchar 'x' + putchar 'y'
```

– Imperative operations via assignable reference cells:

```
z = ref 0; z := !z + 1;
f(z);
w = !z    (* What is the value of w? *)
```

- Error recovery via exceptions
- Foreign language procedures mapped to "functions"
- Concurrency via operating system threads
- - Ocaml, Standard ML are good examples of this approach

But what if we are "lazy"?

In a lazy functional language, like Haskell, the order of evaluation is undefined.

- Example: res = putchar 'x' + putchar 'y'
 - Output depends upon the evaluation order of (+).
- Example: ls = [putchar 'x', putchar 'y']
 - Output depends on how list is used
 - If only used in length ls, nothing will be printed because length does not evaluate elements of list

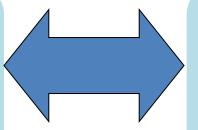
Fundamental question

- Is it possible to add imperative features without changing the meaning of pure Haskell expressions?
- Yes! Exploiting the concept of monad
 - The IO monad defines monadic values which are called actions, and prescribes how to compose them sequentially

Monadic Input and Output The IO Monad

Problem

A functional program defines a pure function, with no side effects



The whole point of running a program is to have some side effect

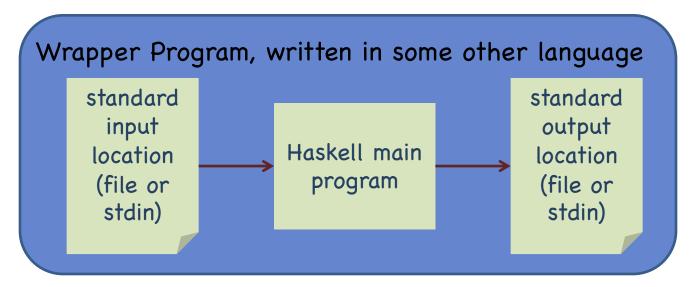
The term "side effect" itself is misleading

Before Monads

- Streams
 - Program sends stream of requests to OS, receives stream of responses
- Continuations
 - User supplies continuations to I/O routines to specify how to process results
- Haskell 1.0 Report adopted Stream model
 - Stream and Continuation models were discovered to be inter-definable

Stream Model: Basic Idea

- Move "side effects" outside of functional program
- Haskell main :: String -> String



But what if you need to read more than one file?
 Or delete files? Or communicate over a socket?

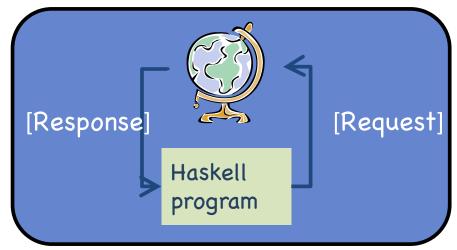
30

Stream Model

 Enrich argument and return type of main to include all input and output events.

- Wrapper program interprets requests and adds responses to input.
- Move side effects outside of functional program

Stream Model: main::[Response] -> [Request]



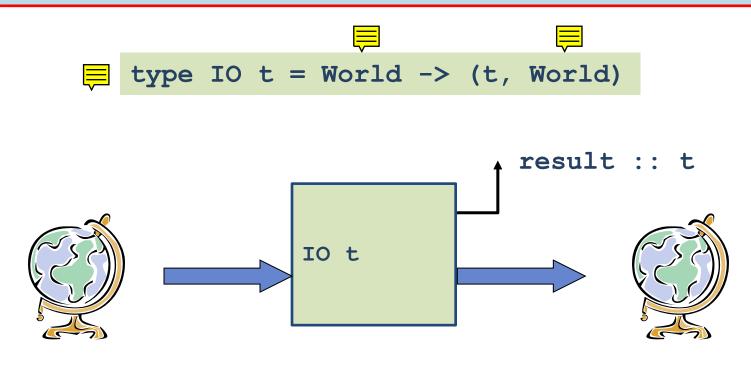
- Problem: Laziness allows program to generate requests prior to processing any responses.
- Hard to extend
 - New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response \(\equiv \)
 - easy to get "out-of-step," which can lead to deadlock
- Not composable
 - no easy way to combine two "main" programs ≡
- ... and other problems!!!

Monadic I/O: The Key Ideas

- IO is a type constructor, instance of Monad
- A value of type (IO t) is a computation or "action" that, when performed, may do some input/output before delivering a result of type t
- return returns the value without making I/O
- then (>>) [and also bind (>>=)] composes two actions sequentially into a larger action ≡
- The only way to perform an action is to call it at some point, directly or indirectly, from Main.main

A Helpful Picture

A value of type (IO t) is an "action." When performed, it may do some input/output before delivering a result of type t.



- An action is a first-class value
- Evaluating an action has no effect; performing the action has the effect

Implementation of the IO monad

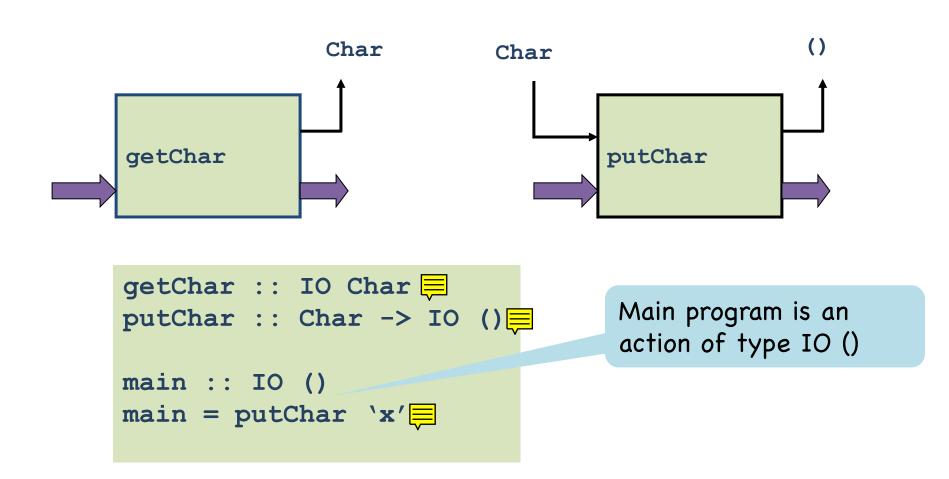
GHC uses "world-passing semantics" for the IO monad

```
type IO t = World -> (t, World)
```

 It represents the "world" by an un-forgeable token of type World, and implements bind and return as:

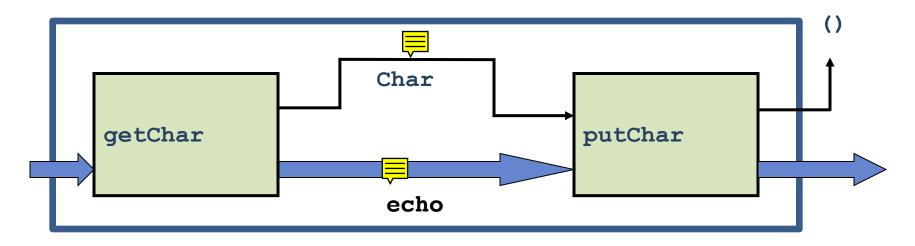
- Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.
- The code generator then converts the code to modify the world "in-place."

Simple I/O actions



The Bind Combinator (>>=)

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```



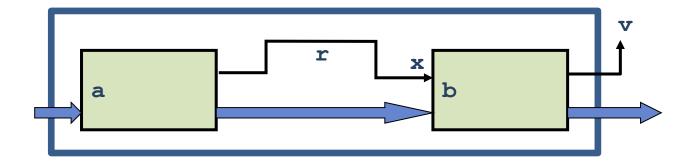
 We have connected two actions to make a new, bigger action.

```
getChar :: IO Char
putChar :: Char -> IO ()
echo :: IO ()
echo = getChar >>= putChar
```

The (>>=) Combinator

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

- Operator is called bind because it binds the result of the left-hand action in the action on the right
- Performing compound action a >>= \x->b: ≡
 - performs action a, to yield value r ≡
 - applies function \x−>b to r ≡
 - performs the resulting action b{x <- r} ≡</p>
 - returns the resulting value ▼ ≡



The (>>) Combinator

 The "then" combinator (>>) does sequencing when there is no value to pass:

```
(>>) :: IO a -> IO b -> IO b ==
-- defined from bind
(>>=) :: IO a -> (a -> IO b) -> IO b
m >> n = m >>= (\_ -> n) ==
```

```
echoDup :: IO () = 
echoDup = getChar >>= \c ->
    putChar c >>
    putChar c
```

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

The return Combinator

 The action (return v) does no IO and immediately returns v:

```
return :: a -> IO a

return
```



The "do" Notation

 The "do" notation adds syntactic sugar to make monadic code easier to read.

```
-- Plain Syntax
getTwoChars :: IO (Char,Char)
getTwoChars = getChar >>= \c1 ->
              getChar >>= \c2 ->
              return (c1,c2)
-- Do Notation
getTwoCharsDo :: IO(Char,Char)
getTwoCharsDo = do { c1 <- getChar ;</pre>
                      c2 <- getChar ;
                      return (c1,c2) }
```

do syntax designed to look imperative.

Desugaring "do" Notation

The "do" notation only adds syntactic sugar:

The scope of variables bound in a generator is the rest of the "do" expression.

The following are equivalent:

```
do { x1 <- p1; ...; xn <- pn; q }
do x1 <- p1; ...; xn <- pn; q
```

```
do x1 <- p1
...
xn <- pn
q
```

Bigger Example

The getLine function reads a line of input:

Note the <u>"regular" code mixed with the monadic operations</u> and the nested "do" expression.

Control Structures on Monads

 Exploiting the monadic combinators, we can define control structures that work for any monad \(\equiv \)

```
repeatN 0 x = return ()
repeatN n x = x >> repeatN (n-1) x ==
repeatN :: (Num a, Monad m, Eq a) => a -> m a1 -> m ()

Main> repeatN 5 (putChar 'h') ==
```

```
Main> for [1..10] (\x -> putStr (show x))
```

A list of IO actions. ■

Sequencing

An IO action returning a list.

• Example use:

```
Main> sequence [getChar, getChar] \equiv
```

IO Provides Access to Files

- The IO Monad provides a large collection of operations for interacting with the "World."
- For example, it provides a direct analogy to the Standard C library functions for files:

```
openFile :: FilePath -> IOMode -> IO Handle ==
hPutStr :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
hClose :: Handle -> IO ()
```

References

- The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion.
- Idea: We can leverage the sequential nature of the IO monad to do other imperative things

```
data IORef a -- Abstract type
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

 A value of type IORef a is a reference to a mutable cell holding a value of type a.

Example Using References

```
import Data.IORef -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
   { r <- newIORef 0;
     addToN r 1 }
 where
    addToN :: IORef Int -> Int -> IO Int
    addToN r i | i > n = readIORef r
               | otherwise = do
                  { v <- readIORef r</pre>
                  ; writeIORef r (v + i)
                  ; addToN r (i+1)}
```

This is terrible: Contrast with: sum [1..n].

The IO Monad as ADT

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b

getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...
openFile :: [Char] -> IOMode -> IO Handle
... more operations on files ...
newIORef :: a -> IO (IORef a)
... more operations on references ...
```

- All operations return an IO action, but only bind (>>=) takes one as an argument.
- Bind is the only operation that combines IO actions, which forces sequentiality.
- In pure Haskell, there is no way to transform a value of type IO a into a value of type a

Unreasonable Restriction?

- In pure Haskell, there is no way to transform a value of type
 IO a into a value of type a
- Suppose you wanted to read a configuration file at the beginning of your program:

```
configFileContents :: [String]
configFileContents = lines (readFile "config") -- WRONG!
useOptimisation :: Bool
useOptimisation = "optimise" 'elem' configFileContents
```

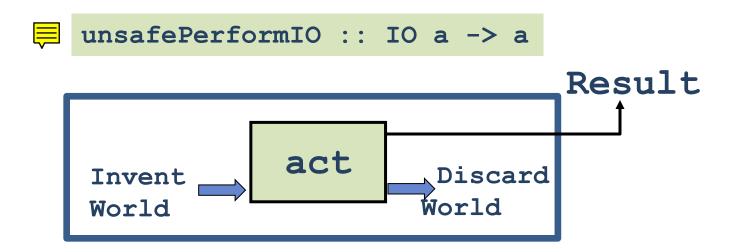
- The problem is that readFile returns an IO String, not a String.
- **Option 1**: Write entire program in IO monad. But then we lose the simplicity of pure code.
- Option 2: Escape from the IO Monad using a function from IO String -> String. But this is disallowed!

Type-Unsafe Haskell Programming

- Reading a file is an I/O action, so in general it matters when we read the file.
- But we know the configuration file will not change during the program, so it doesn't matter when we read it.
- This situation arises sufficiently often that Haskell implementations offer one last unsafe I/O primitive: unsafePerformIO.

```
unsafePerformIO :: IO a -> a
configFileContents :: [String]
configFileContents = lines(unsafePerformIO(readFile "config"))
```

unsafePerformIO



- The operator has a deliberately long name to discourage its use.
- Its use comes with a proof obligation: a promise to the compiler that the timing of this operation relative to all other operations doesn't matter.

unsafePerformIO

 Warning: As its name suggests, unsafePerformIO breaks the soundness of the type system.

```
r = unsafePerformIO (newIORef (error "urk")) 

r :: IORef a = -- Type of the stored value is generic

cast x = unsafePerformIO (do {writeIORef r x; readIORef r = })

> :t (\x > cast x)
(\x > cast x) :: a1 > a2

> cast 65:: Char = 'A'
```

- So claims that Haskell is type safe only apply to programs that don't use unsafePerformIO.
- Similar examples are what caused difficulties in integrating references with Hindley/Milner type inference in ML.

Summary on Mondas

- A complete Haskell program is a single IO action called main. Inside IO, code is single-threaded.
- Big IO actions are built by gluing together smaller ones with bind (>>=) and by converting pure code into actions with return.
- IO actions are first-class.
 - They can be passed to functions, returned from functions, and stored in data structures.
 - So it is easy to define new "glue" combinators.
- The IO Monad allows Haskell to be pure while efficiently supporting side effects.
- The type system separates the pure from the effectful code.