

Interactive Programming and IO (based on material due to Graham Hutton and Stephanie Weirich)



1

- Quiz 2 on Friday

- PS5 later this week; no new hw until Oct 15th
- Project proposal guidelines now available
 - Submission deadline: October 15th
 - Team up with a partner. A team of two is required
 - Please read carefully
 - Do some research and think through carefully

Programming in Haskell, A Milanova

2

2

Outline

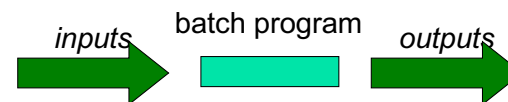
- IO actions
 - Writing on screen and reading from screen
 - File handles and files
- Monads!
- The Maybe and List monads
- Monadic bind and do notation
- Back to the IO monad
- Generic monadic functions

3

3

Introduction

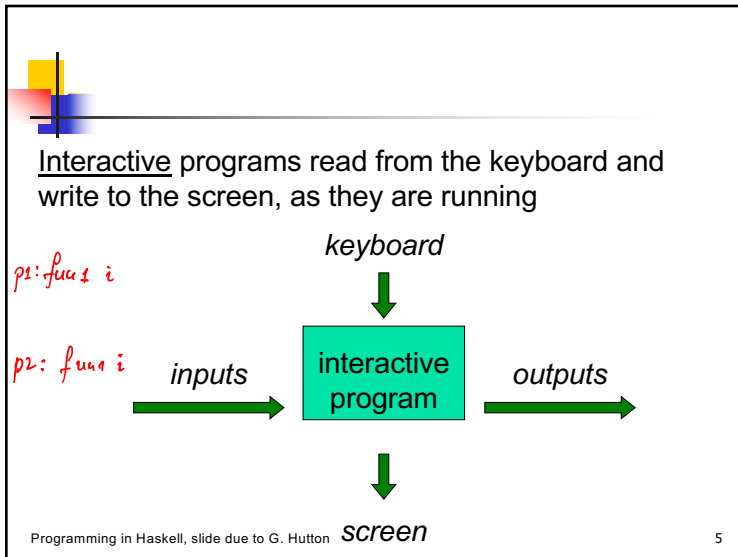
We have seen how Haskell can be used to write batch programs that take all their inputs at the start and give all their outputs at the end



Programming in Haskell, slide due to G. Hutton

4

4



5

The Problem

Haskell programs are pure mathematical functions:

- Haskell programs have no side effects

However, reading from the keyboard and writing to the screen are side effects:

- Interactive programs have side effects

Programming in Haskell, slide due to G. Hutton

6

The Solution

Interactive programs can be written in Haskell by using types to distinguish pure expressions from impure actions that may involve side effects

IO a

The type of actions that return a value of type a.

Programming in Haskell, slide due to G. Hutton

7

For example:

getChar :: IO Char

The type of actions that return a character.

putChar :: Char -> IO ()

The type of purely side effecting actions that return no result value.

Note:

- () is the type of tuples with no components

Programming in Haskell, slide due to G. Hutton

8

Basic Actions

The standard library provides a number of actions, including the following three primitives:

- The action `getChar` reads a character from the keyboard, echoes it to the screen, and returns the character as its result value:

→ `getChar :: IO Char`

9

- The action `putChar c` writes the character `c` to the screen, and returns no result value:

`putChar :: Char → IO ()`

- The action `return v` simply returns the value `v`, without performing any interaction:

→ `return :: a → IO a`

*> :: i return
Moved in ⇒ a → ma*

10

Sequencing

A sequence of actions can be combined as a single composite action using the keyword `do`

For example:

```
act :: IO (Char, Char)
act = do x ← getChar
        getChar
        y ← getChar
        return (x,y)
```

Char
IO Char
Char
IO (Char, Char)

11

Derived Primitives

- Reading a string from the keyboard:

```
getline :: IO String
getline = do x ← getChar
            if x == '\n' then
                return []
            else
                do xs ← getLine
                 return (x:xs)
```

Action 1
IO String
Action 2
IO String

12

- Writing a string to the screen:

```
putStr :: String → IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                  putStr xs
```

- Writing a string and moving to a new line:

```
putStrLn :: String → IO ()
putStrLn xs = do putStr xs Action 1
                 putChar '\n' Action 2
```

13

Example

We can now define an action that prompts for a string to be entered and displays its length:

```
strlen :: IO ()
strlen = do putStr "Enter a string: " Action 1
           xs ← getLine Action 2
           putStr "The string has " Action 3
           putStr (show (length xs)) Action 4
           putStrLn " characters" Action 5
```

14

For example:

```
> strlen
Enter a string: Haskell
The string has 7 characters
```

Note:

- Evaluating an action executes its side effects, with the final result value being discarded.

15

Hangman

Consider the following version of hangman:

- One player secretly types in a word.
- The other player tries to deduce the word, by entering a sequence of guesses.
- For each guess, the computer indicates which letters in the secret word occur in the guess.
- The game ends when the guess is correct.

16

We adopt a top down approach to implementing hangman in Haskell, starting as follows:

```
hangman :: IO ()
hangman = do putStrLn "Think of a word: "
            word ← sgetLine
            putStrLn "Try to guess it:"
            play word
```

Programming in Haskell, slide due to G. Hutton

17

17

The action sgetLine reads a line of text from the keyboard, echoing each character as a dash:

```
sgetLine :: IO String
sgetLine = do x ← getch
              if x == '\n' then
                do putStrLn x
                 return []
              else
                do putStrLn '-'
                 xs ← sgetLine
                 return (x:xs)
```

Programming in Haskell, slide due to G. Hutton

18

18

The action getCh reads a single character from the keyboard, without echoing it to the screen:

```
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False
           x ← getChar
           hSetEcho stdin True
           return x
```

Programming in Haskell, slide due to G. Hutton

19

19

The function play is the main loop, which requests and processes guesses until the game ends.

```
play :: String → IO ()
play word =
  do putStrLn "? "
     guess ← getLine
     if guess == word then
       putStrLn "You got it!"
     else
       do putStrLn (match word guess)
        play word
```

Programming in Haskell, slide due to G. Hutton

20

20

The function `match` indicates which characters in one string occur in a second string:

```
match :: String -> String -> String
match xs ys =
  [if elem x ys then x else '-' | x <- xs]
```

For example:

```
> match "haskell" "pascal"
"-as--ll"
```

Programming in Haskell, slide due to G. Hutton

21

21

Exercise

Implement the game of `nim` in Haskell, where the rules of the game are as follows:

- The board comprises five rows of stars:

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

Programming in Haskell, slide due to G. Hutton

22

22

- Two players take turn about to remove one or more stars from the end of a single row.
- The winner is the player who removes the last star or stars from the board.

Hint:

Represent the board as a list of five integers that give the number of stars remaining on each row. For example, the initial board is `[5,4,3,2,1]`.

Programming in Haskell, slide due to G. Hutton

23

23

Files and File Handles

```
import System.IO -- Standard IO

openFile :: FilePath -> IOMode -> Handle
hClose :: Handle -> () -- void
hIsEOF :: Handle -> Bool
hGetChar :: Handle -> Char
```

These operations break "referentially transparency". For example, `hGetChar` typically returns different value when called twice in a row.

Programming in Haskell, A Milanova

24

24

```

getFileContents :: String -> IO String
getFileContents filename =
  do h <- openFile filename ReadMode
     putStrLn filename
     reversed_cs <- readFileContents h []
     hClose h
     return (reverse reversed_cs)

readFileContents :: Handle -> String -> IO String
readFileContents h rcs =
  do b <- hIsEOF h
     if (b) then return rcs
     else do { c <- hGetChar h; readFileContents h (c:rcs) }

```

→ another notation for sequencing!

25

Other useful functions:

```

// reads entire file into one string:
readFile :: FilePath -> IO String
// writes entire string into a file:
writeFile :: FilePath -> String -> IO ()

```

String

```

main = do
  [f,g] <- getArgs
  s <- readFile f
  writeFile g s

```

26

Outline

- IO actions
 - Writing on screen and reading from screen
 - File handles and files
- **Monads!**
- The Maybe and List monads
- Monadic bind and do notation
- Back to the IO monad
- Generic monadic functions

27

Example: Cloned Sheep

data Maybe a = Nothing | Just a

```

type Sheep = ...
father :: Sheep -> Maybe Sheep
father = ...
mother :: Sheep -> Maybe Sheep
mother = ...

```


Defining the function that takes a sheep and returns the maternal grandfather of the sheep:

```

maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
  Nothing -> Nothing
  Just m -> father m

```

28



What if we wanted to go one generation up:


```

mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
  Nothing -> Nothing
  Just m -> case (father m) of
    Nothing -> Nothing
    Just gf -> father gf
  
```

Tedious, unreadable, difficult to maintain!

Programming in Haskell, A Milanova (All About Monads tutorial on Haskell.org) 29

29



Monads!

Monads are a way to cleanly compose actions

- E.g., `f` may return a value of type `a` or `Nothing`

Composing actions becomes tedious


```

case (f s) of
  Nothing -> Nothing
  Just m -> case (f m) ...
  
```

In Haskell, monads model IO and State

Programming in Haskell, A Milanova 30

30



Monad

- Monad is a higher-order type class (actually, right term is a higher-kinded type class):

```


class Monad m where
  -- | Sequentially compose two actions, passing any
  -- value that is produced by first action to the second
  (>>=) :: m a -> (a -> m b) -> m b

  -- | Inject a value into a monad type
  return :: a -> m a

  -- More functions that we'll leave for now.
  
```

Programming in Haskell, A Milanova 31

31



Maybe Monad

- What are some instances of Monad?

instance Monad Maybe where

```

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

-- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
Just x >>= f = f x
  
```

Programming in Haskell, A Milanova 32

32

Example: Cloned Sheep

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
  Nothing -> Nothing
  Just m -> case (father m) of
    Nothing -> Nothing
    Just gf -> father gf
```

Becomes just this:

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s =
  (return s) >>= mother >>= father >>= father
```

If at any point a function returns Nothing, it cleanly propagates, no need to check.

33

33

Example: Lambda Calculus

```
data LExpr = ...
oneStepAppNF :: LExpr -> Maybe LExpr
oneStepAppNF expr = case expr of
  (Atom v) -> Nothing
  ...
```

Defining the function that does two applicative order reduction steps:

```
twoSteps :: LExpr -> Maybe LExpr
twoSteps expr = (return expr) >>= oneStepAppNF >>= oneStepAppNF
```

Programming in Haskell, A Milanova

34

34

Exercise

- Define firstJust which takes two Maybes and returns the first Just, if any:

```
-- firstJust (Just 1) (Just 2) yields (Just 1)
-- firstJust Nothing (Just 2) yields (Just 2)
-- firstJust (Just 1) Nothing yields (Just 1)
-- firstJust Nothing Nothing yields Nothing
```

```
firstJust :: Maybe a -> Maybe a -> Maybe a
```

```
firstJust x.y = case x of
  Nothing -> y
  _ -> x
```

Programming in Haskell, A Milanova

35

35

Exercise

- Define seqJust which takes two Maybes and if both are Just returns the first, otherwise Nothing:

```
-- seqJust (Just 1) (Just 2) yields (Just 1)
-- seqJust Nothing (Just 2) yields Nothing
-- seqJust (Just 1) Nothing yields Nothing
-- seqJust Nothing Nothing yields Nothing
```

```
seqJust :: Maybe a -> Maybe b -> Maybe a
```

```
seqJust x.y = case y of
  Just _ -> x
  Nothing -> Nothing
```

```
seqJust' x y = x >>= (1 -> y) >>= (1 -> x)
```

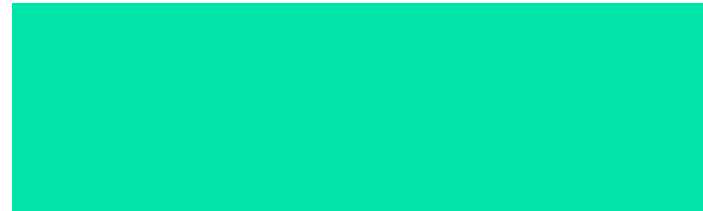
Programming in Haskell, A Milanova

36

36

Exercise

- Can we define either `firstJust` or `seqJust` using the bind operator `>>=`? Why or why not? Remember the definition we gave of `>>=` for `Maybe`.

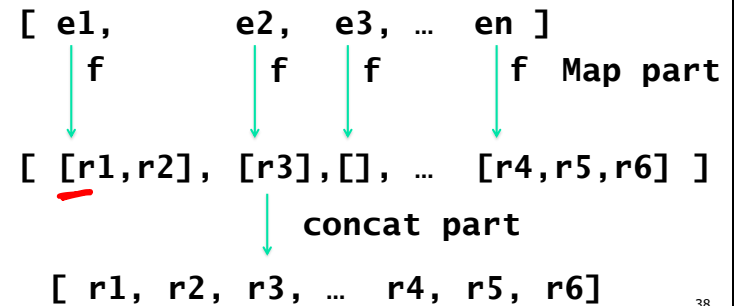


37

List is Monadic

```
> concatMap f xs
```

$xs \gg= f = \text{concatMap } f \text{ } xs$



38

The List Monad

- List type constructor is an instance of the Monad type class:

```
instance Monad [] where
  -- return :: a -> [a]
  return x = ?
```

```
-- (>>=) :: [a] -> (a -> [b]) -> [b]
li >>= f = concatMap f li
```

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

```
> concatMap (return . product) [[1,2],[3,4],[5,6]]
```

39

Exercises

```
> f x = [x+1]
> [1,2,3] >>= f

> [1,2,3] >>= \x -> return (x+1)

> [1,2,3] >>= return . (+1)

> [1,2,3] >>= return . (+1) . (*5)
```

40

Exercises

- Define maybeToList:

```
maybeToList :: Maybe a -> [a]
maybeToList x =
```

- Define parents, which returns the list of parents (or parent) of a cloned sheep:

```
parents :: Sheep -> [Sheep]
parents s =
```

- Now define grandParents:

```
grandParents :: Sheep -> [Sheep]
grandParents s =
```

41

Outline

- IO actions
 - Writing on screen and reading from screen
 - File handles and files
- Monads!
 - The Maybe and List monads
 - Monadic bind and do notation
 - Back to the IO monad
 - Generic monadic functions

42

42

Monad Laws

- return is an identity to >>=:

```
1. return x >>= f == f x
```

```
2. m >>= return == m
```

- Associativity (kind of) of >>=:

```
3. (m >>= f) >>= g == m >>= (\x -> f x >>= g)
```

- Remember, do notation correct only if associativity holds

- Exercise: Show Maybe and List obey the monad laws

Programming in Haskell, A Milanova

43

43

do Notation

- do notation is just syntactic sugar for monadic bind >>=!

```
> return sh >>= mother >>= father
```

By identity monad law:

```
> mother sh >>= father
```

Making argument explicit by η -expansion:

```
> mother sh >>= (\m -> father m)
```


Which has direct rewrite into do-notation!

```
do m <- mother sh
   father m
```

Programming in Haskell, A Milanova

44

44



- do notation is just syntactic sugar for monadic bind >>=!

```
> (return sh) >>= mother >>= father
```

We can make encapsulated element explicit (η -expansion):

```
> (return sh) >>= (\s -> mother s) >>= (\m -> father m)
```

Which is equivalent to (by associativity monad law):


```
> (return sh) >>= (\s -> mother s >>= (\m -> father m))
```

Which has a direct rewrite in do-notation:

```
do s <- return sh
   m <- mother s
   father m
```

Programming in Haskell, A Milanova 45

45



- Start with “straightforward” application of bind >>= :

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather sh =
  return sh >>= mother >>= father >>= father
```

By monad laws:


```
return sh >>= (\s -> mother s >>= (\m -> father m
  >>= (\gf -> father gf)))
```

And the do notation:

```
do
  s <- return sh
  m <- mother s
  gf <- father m
  father gf
```

Programming in Haskell, A Milanova 46

46



- do notation is just syntactic sugar for monadic bind >>=!

```
> [1,2,3] >>= return . (+1) >>= return . (*5)
```

We can make encapsulated element explicit:

```
> [1,2,3] >>= (\x -> [x+1]) >>= (\y -> [y*5])
```

Which is equivalent to (by monad laws):


```
> [1,2,3] >>= (\x -> [x+1] >>= (\y -> [y*5]))
```

Which has a direct rewrite in do-notation:

```
v = do x <- [1,2,3]
      y <- [x+1]
      [y*5]
```

Programming in Haskell, A Milanova 47

47



List Comprehension

- List comprehension is syntactic sugar over do notation. In other words, list comprehensions are rewritten internally in terms of >>=

```
v = do x <- [1,2,3]
      y <- [x+1]
      [y*5]
```

Direct rewrite into comprehension:

```
v = [ y*5 | x <- [1,2,3], y <- [x+1] ]
```

Programming in Haskell, A Milanova 48

48

Monad

- >> composes actions without worrying about value produced by first action:

```
class Monad m where
  -- | Sequentially compose two actions, passing any
  -- value that is produced by first action to the second
  (>>=) :: m a -> (a -> m b) -> m b

  -- | Inject a value into a monad type
  return :: a -> m a

  -- | Ignore value produced by first action
  (>>) :: m a -> m b -> m b
```

49

- Second action does not need input from first action:

```
(>>) :: m a -> m b -> m b
m >> f = m >>= (\_ -> f)
```

```
v = do
  x <- doSomething
  doSomethingElse -- ?
  y <- doMore
  f x y
```

```
doSomething >>= (\x ->
  doSomethingElse >>
  doMore >>= (\y ->
  (f x y)))
```

50

Exercise

- List comprehension is syntactic sugar over do notation

```
v = do x <- [1,2,3]
      y <- [6,5,4]
      [(x,y)]
```

Direct rewrite into comprehension:



51

Back to IO. IO is a Monad!

IO a: Computation that does some **IO** producing a value of type **a**. E.g., **(IO Char)**, **(IO String)**

Unlike other monads (e.g., **Maybe** and **List**) there is no way to make an **IO a** into an **a**

The monad encapsulates “mutable” IO state

... and, there is no “rep exposure” of this state!

Access to state is only through well-defined monadic operations (e.g., **hGetChar**)

52

Turn do sequence back into a monadic bind expression:

```
act :: IO (Char,Char)
act = do x <- getChar
        y <- getChar
        return (x,y)
```

Bind expression:

```
act' = getChar >>= (\x -> getChar >>= (\y -> return (x,y)))
```

53

Exercise

Turn do sequence back into a monadic bind expression:

```
getLine' :: IO String
getLine' = do
  x <- getChar
  if x == '\n' then
    return []
  else do
    xs <- getLine'
    return (x:xs)
```

Bind expression:

```
getLine' :: IO String
getLine' = getChar >>= (\x ->
  if x == '\n'
  then return []
  else getLine' >>= (\xs -> return (x:xs)))
```

54

Exercise

Turn back into an equivalent monadic bind expression:

```
simpleProgram :: IO ()
simpleProgram = do
  putStrLn "A simple program that does IO."
  putStrLn "What is your name?"
  inpStr <- getLine
  putStrLn ("Welcome to Haskell, " ++ inpStr ++ "!")
```

Bind expression using >>=:

```
putStrLn s1 >>= (\x' -> putStrLn s2 >>= (\x'' -> getLine >>=
  (\inpStr -> putStrLn (s3 ++ inpStr))))
```

Bind expression using >>:

```
putStrLn s1 >> (putStrLn s2 >> (getLine >>= (\inpStr ->
  putStrLn (s3 ++ inpStr))))
```

55

Outline

- IO actions
 - Writing on screen and reading from screen
 - File handles and files
- Monads!
- The Maybe and List monads
- Monadic bind and do notation
- Back to the IO monad
- **Generic monadic functions**

56

56

Generic Monadic Functions

- Define some useful generic functions. What does map do?

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f xs =
```

- What are some use cases with Maybe and List monads?
- Two use cases are shown in slides. Add an additional use case for mapM

57

Exercise

- Define fun1 in terms of mapM and maybeUpper:

```
import Data.Char

maybeUpper :: Char -> Maybe Char
maybeUpper x = if isAlpha x
                 then Just (toUpper x)
                 else Nothing

-- fun1 "abAcD" yields Just "ABACD"
-- fun1 "ab1cd" yields Nothing
fun1 :: String -> Maybe String
fun1 =
```

58

Exercise

- Define fun2 in terms of mapM and onlyUpper:

```
import Data.Char

onlyUpper :: String -> Maybe String
onlyUpper = filter isUpper

-- fun2 ["ABC", "Ac"] yields ["AA", "BA", "CA"]
-- fun2 ["ABC", "ac"] yields []
fun2 :: [String] -> [String]
fun2 =
```

59

- What does foldM do?

```
-- foldM safeDiv 16 [2,2] yields Just 4
-- foldM safeDiv 16 [2,2,0] yields Nothing

foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM op i xs =
```

60

■ What does sequence do? Add an additional use case!

```
-- sequence [[1::Int,2],[3],[4,5]] yields
--           [[1,3,4],[1,3,5],[2,3,4],[2,4,5]]
-- sequence [Just (3::Int), Nothing, Just 4] yields
--           Nothing
```

```
sequence :: Monad m => [m a] -> m [a]
sequence xs =
```

61

■ What does Kleisli fish operator do? Add a use case

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(>=>) =
```

62

■ What does join do? Add an additional use case

```
-- join [[1::Int,2],[3,4]] yields [1,2,3,4]
-- join (Just (Just (3::Int))) yields Just 3
```

```
join :: Monad m => m (m a) -> m a
join =
```

63

■ What does liftM function do?

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM =
```

■ What does liftM2 do?

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 =
```

64



■ Folding patterns

■ Lists

- Regular lists
- Divide lists

```
instance Foldable DivideList where
  -- (a -> SortedList a) -> DivideList a -> SortedList a
  foldMap f xs =
    case divide xs of
      (DL [], DL []) -> mempty
      (DL [], DL [x]) -> f x
      (d11, d12) -> (foldMap f d11) <> (foldMap f d12)
```

■ Trees

Programming in Haskell, A Milanova

65