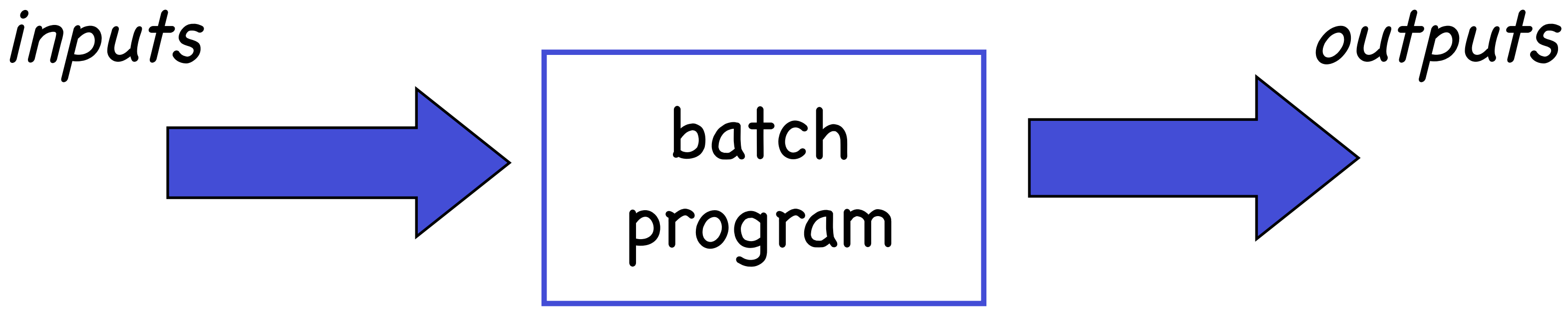
CSCE 314
Programming Languages

Interactive Programs: I/O and Monads

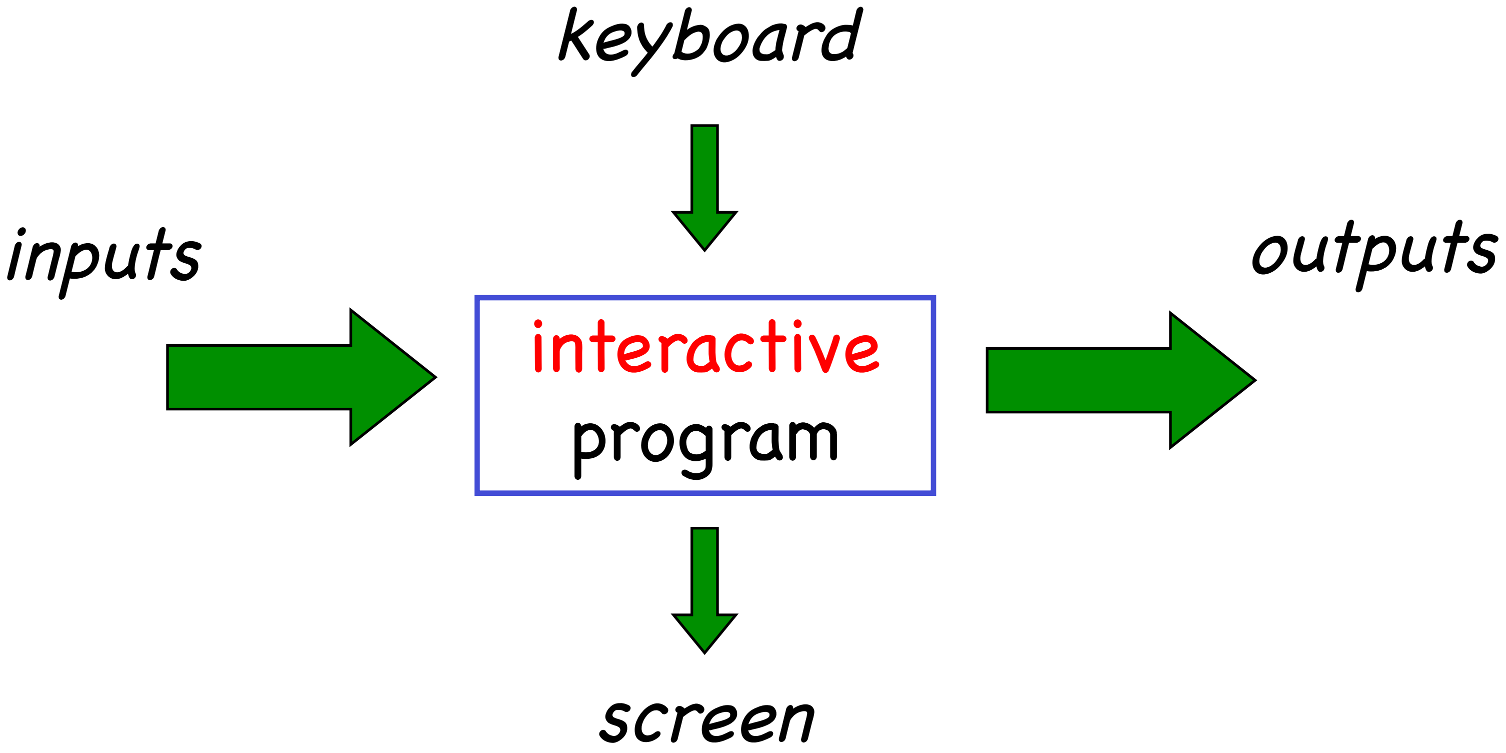
Dr. Hyunyoung Lee

Introduction

To date, 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.



However, we would also like to use Haskell to write interactive programs that read from the keyboard and write to the screen, as they are running.



The Problem: Haskell functions are pure mathematical functions

Haskell programs have no side effects.

referential transparency: called with the same arguments, a function always returns the same value

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

Interactive programs have side effects.

The Solution - The IO Type

Interactive programs can be viewed as a pure function whose domain and codomain are the current *state of the world*:

```
type IO = World -> World
```

However, an interactive program may return a result value in addition to performing side effects:

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

What if we need an interactive program that takes an argument of type b?
Use currying:

```
b -> World -> (a, World)
```

The Solution (Cont.)

Now, interactive programs (impure actions) can be defined using the IO type:

`IO a`

The type of actions that return a value of type a

For example:

`IO Char`

The type of actions that return a character

`IO ()`

The type of actions that return the empty tuple (a dummy value); purely side-effecting actions

Basic Actions (defined in the standard library)

1. 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
```

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

```
putChar :: Char -> IO ()
```

3. The action return v simply returns the value v, without performing any interaction:

```
return :: a -> IO a
```

Sequencing

A sequence of actions can be combined as a single composite action using the >>= or >> (binding) operators.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 a world1
   in (b, world2)
```

Apply action1 to world0, get a new action (action2 v), and apply that to the modified world

Compare it with:

```
(>>) :: IO a -> IO b -> IO b
(action1 >> action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 world1
   in (b, world2)
```

Derived Primitives

- Reading a string from the keyboard:

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

- Writing a string to the screen:

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

- Writing a string and moving to a new line:

```
putStrLn    :: String → IO ()
putStrLn xs = putStr xs >> putChar '\n'
```

Derived Primitives (do Notation)

- 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)
```

- 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
                 putChar '\n'
```

Building More Complex IO Actions

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

```
strlen :: IO ()
strlen  = putStr "Enter a string: ” >>
          getLine >>= \xs ->
          putStr "The string has ” >>
          putStr (show (length xs)) >>
          putStrLn " characters."
```

Building More Complex IO Actions (do)

Using the do natation:

```
strlen :: IO ()
strlen  = do putStr “Enter a string: ”
             xs <- getLine
             putStr “The string has ”
             putStr (show (length xs))
             putStrLn “ characters.”
```

IO Monad As An Abstract Data Type

Consider:

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
getChar :: IO Char
putChar :: Char -> IO ()
openFile :: [Char] -> IOMode -> IO Handle
```

- All primitive IO operations return an IO action
- IO monad is sticky: all functions that take an IO argument, return an IO action
- `return` offers a way *in* to an IO action, but no function offers a way *out* (you can bind a variable to the IO result by use of "<-")

The Type of main

A complete Haskell program is a single IO action. For example:

```
main :: IO ()
main = getLine >>= \cs ->
          putLine (reverse cs)
```

Typically, IO "contaminates" a small part of the program (outermost part), and a larger portion of a Haskell program does not perform any IO. For example, in the above definition of main, reverse is a non-IO function.

Monad (Roughly)

- Monad is a strategy for combining computations into more complex computations
- No language support, besides higher-order functions, is necessary
  - But Haskell provides the do notation
- Monads play a central role in the I/O system
  - Understanding the I/O monad will improve your code and extend your capabilities

Monad Example: Maybe

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

Reminder:

- Maybe is a <u>type constructor</u> and Nothing and Just are <u>data constructors</u>
- The polymorphic type <u>Maybe a</u> is the type of all computations that may return a value or Nothing – properties of the Maybe container
- For example, let f be a partial function of type a -> b, then we can define f with type:

```
f :: a -> Maybe b -- returns Just b or Nothing
```

Example Using Maybe

Consider the following function querying a database, signaling failure with Nothing

```
doQuery :: Query -> DB -> Maybe Record
```

Now, consider the task of performing a sequence of queries:

```
r :: Maybe Record
r = case doQuery q1 db of
      Nothing -> Nothing
      Just r1 -> case doQuery (q2 r1) db of
                   Nothing -> Nothing
                   Just r2 -> case doQuery (q3 r2) db of
                                Nothing -> Nothing
                                Just r3 -> . . .
```

Capture the pattern into a combinator

```
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
mB `thenMB` f = case mB of
                  Nothing -> Nothing
                  Just a -> f a
```

This allows the following rewrite to doQuery

```
r :: Maybe Record
r = doQuery q1 db           `thenMB` \r1 ->
    doQuery (q2 r1) db      `thenMB` \r2 ->
    doQuery (q3 r2) db      `thenMB` . . .
```

Another Example: The List Monad

The common Haskell type constructor, [] (for building lists), is also a monad that encapsulates a strategy for combining computations that can return 0, 1, or multiple values:

```
instance Monad [] where
    m >>= f = concatMap f m
    return x = [x]
```

The type of (>>=):

```
    (>>=) :: [a] -> (a -> [b]) -> [b]
```

The binding operation creates a new list containing the results of applying the function to all of the values in the original list.

```
    concatMap :: (a -> [b]) -> [a] -> [b]
```

Combinators controlling parameter passing and computational flow

Many uses for the kind of programming we just saw

- Data Structures: lists, trees, sets
- Computational Flow: Maybe, Error Reporting, non-determinism
- Value Passing: state transformer, environment variables, output generation
- Interaction with external state: IO, GUI programming
- Other: parsing combinators, concurrency, mutable data structures

There are instances of Monad for all of the above situations

Monad Definition

Monad is a triple (M, return, >>=) consisting of a type constructor M and two polymorphic functions:

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

which satisfy the monad laws (note, checking these is up to the programmer):

```
return x >>= f    == f x      -- left identity
m >>= return      == m        -- right identity
(m >>= f) >>= g   ==
     m >>= (\x -> f x >>= g) -- associativity
```

What is the practical meaning of the monad laws?

Let us rewrite the laws in do-notation:

<u>Left identity</u>:

```
do { x’ <- return x;
     f x’                  ==   do { f x }
   }
```

<u>Right identity</u>:

```
do { x <- m;
     return x              ==   do { m }
   }
```

<u>Associativity</u>:

```
do  { y <- do { x <- m;   ==   do { x <- m;
                f x                 y <- f x;
              }                     g y
      g y                         }
    }
```

The Monad Type Class

```
class Monad m where
    >>= :: m a -> (a -> m b) -> m b
    >>  :: m a -> m b -> m b
    return :: a -> m a
    m >> k = m >>= \_ -> k
```

- >> is a shorthand for >>= ignoring the result of first action
- Any type with compatible combinators can be made to be an instance of this class. For example:

```
data Maybe a = Just a | Nothing
thenMB :: Maybe a -> (a -> Maybe b) -> Maybe b
instance Monad Maybe where
   (>>=) = thenMB
   return a = Just a
```

Utilizing the Monad Type Class

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

    m >> k = m >>= \_ -> k
```

- The type class gives a common interface for all monads
- Thus, we can define functions operating on all monads.
- For example, execute each monadic computation in a list:

```
sequence          :: Monad m => [m a] -> m [a]
sequence []        = return []
sequence (c:cs) = c             >>= \x ->
                   sequence cs >>= \xs ->
                   return (x:xs)
```

Running a Monad

- Most monadic computations (such as IO actions) are functions of some sorts

- Combining computations with bind creates ever more complex computations, where some state/world/. . . is threaded from one computation to another, but essentially a complex computation is still a function of some sorts

- A monadic computation is "performed" by applying this function

Monad Summary

Converting a program into a monadic form means:

- A function of type a -> b is converted to a function of type a -> M b
- M then captures whatever needs to be captured, environment, state, . . .
- and can be changed easily

Going into, staying in, and getting out?

- Roughly, return gets a value into a monad
- Bind keeps us in the monad and allows to perform computations within
- There's nothing to get us out! -- This is crucial in the IO monad for not "leaking" side effects to otherwise purely functional program

Hangman

Consider the following version of hangman:

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

Hangman (Cont.)

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:"
      guess word
```

Hangman (Cont.)

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 putChar x
                     return []
                else
                  do putChar '-'
                     xs ← sgetLine
                     return (x:xs)
```

Hangman (Cont.)

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
            c ← getChar
            hSetEcho stdin True
            return c
```

Hangman (Cont.)

The function <u>guess</u> is the main loop, which requests and processes guesses until the game ends.

```
guess      :: String → IO ()
guess word =
   do putStr "> "
      xs ← getLine
      if xs == word then
         putStrLn "You got it!"
       else
         do putStrLn (diff word xs)
            guess word
```

Hangman (Cont.)

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

```
diff         :: String → String → String
diff xs ys =
   [if elem x ys then x else '-' | x ← xs]
```

For example:

```
> diff "haskell" "pascal"

"-as--ll"
```