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.

![Diagram](inputs -> batch program -> outputs)
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*:

\[
\text{type IO} = \text{World} \rightarrow \text{World}
\]

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

\[
\text{type IO a} = \text{World} \rightarrow (a, \text{World})
\]

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

Use currying:

\[
b \rightarrow \text{World} \rightarrow (a, \text{World})
\]
The Solution (Cont.)

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

\[ \text{IO } a \]

The type of actions that return a value of type \( a \)

For example:

\[ \text{IO } \text{Char} \]

The type of actions that return a character

\[ \text{IO } () \]

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:

   ```haskell
   getChar :: IO Char
   ```

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

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

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

   ```haskell
   return :: a -> IO a
   ```
Sequencing

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

\[
(\ggg) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\]
\[
\text{let } (a, \text{world1}) = \text{action1 } \text{world0}
\]
\[
(b, \text{world2}) = \text{action2 } a \text{ world1}
\]
\[
in (b, \text{world2})
\]

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

Compare it with:

\[
(\gg) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b
\]
\[
\text{let } (a, \text{world1}) = \text{action1 } \text{world0}
\]
\[
(b, \text{world2}) = \text{action2 } \text{world1}
\]
\[
in (b, \text{world2})
\]
Derived Primitives

- Reading a string from the keyboard:

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

- Writing a string to the screen:

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

- Writing a string and moving to a new line:

  ```haskell
  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:

```plaintext
strlen :: IO ()
strlen = putStrLn "Enter a string: " >>
        getLine >>= \xs ->
        putStrLn "The string has " >>
        putStrLn (show (length xs)) >>
        putStrLn " characters."
```
Building More Complex IO Actions (do)

Using the do notation:

```haskell
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:

```haskell
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 type constructor and Nothing and Just are data constructors
- The polymorphic type `Maybe a` 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:
  
  \[ f :: a \rightarrow \text{Maybe } b \rightarrow \text{Just } b \text{ or Nothing} \]
Example Using Maybe

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

\[
\text{doQuery} :: \text{Query} \to \text{DB} \to \text{Maybe Record}
\]

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

\[
r :: \text{Maybe Record}
\]

\[
r = \text{case doQuery } q1 \text{ db of}
\]

\[
\text{Nothing} \to \text{Nothing}
\]

\[
\text{Just } r1 \to \text{case doQuery } (q2 \ r1) \text{ db of}
\]

\[
\text{Nothing} \to \text{Nothing}
\]

\[
\text{Just } r2 \to \text{case doQuery } (q3 \ r2) \text{ db of}
\]

\[
\text{Nothing} \to \text{Nothing}
\]

\[
\text{Just } r3 \to \ldots
\]
Capture the pattern into a combinator

\[
\begin{align*}
\text{thenMB} &:: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b \\
&\text{mB `thenMB` f = case mB of} \\
&\quad \text{Nothing} \rightarrow \text{Nothing} \\
&\quad \text{Just } a \rightarrow f a
\end{align*}
\]

This allows the following rewrite to doQuery

\[
\begin{align*}
\text{r} &:: \text{Maybe Record} \\
\text{r} &= \text{doQuery q1 db `thenMB` \(\text{\textbackslash r1}\) ->} \\
&\quad \text{doQuery (q2 r1) db `thenMB` \(\text{\textbackslash r2}\) ->} \\
&\quad \text{doQuery (q3 r2) db `thenMB` . . .}
\end{align*}
\]
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:

\[
\text{instance Monad [] where}
\]

\[
m >>= f = \text{concatMap} f m
\]

\[
\text{return } x = [x]
\]

The type of (>>=):

\[(>>=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]\]

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

\[
\text{concatMap :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]}
\]
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 \((M, \text{return}, \ggg)\) consisting of a type constructor \(M\) and two polymorphic functions:

\[
\text{return} :: a \rightarrow M\ a \\
(\ggg) :: M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b
\]

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

\[
\begin{align*}
\text{return } x \ggg f & = f\ x \quad \text{-- left identity} \\
\text{m } \ggg \text{return} & = m \quad \text{-- right identity} \\
(m \ggg f) \ggg g & = m \ggg (\lambda x \rightarrow f\ x \ggg g) \quad \text{-- associativity}
\end{align*}
\]
What is the practical meaning of the monad laws?

Let us rewrite the laws in do-notation:

**Left identity:**

```haskell
do { x' <- return x;
     f x'             ==  do { f x }
 }
```

**Right identity:**

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

**Associativity:**

```haskell
do { y <- do { x <- m;  ==  do { x <- m;
    f x               y <- f x;
  }                   g y
     g y                      }
 }
```
The Monad Type Class

```haskell
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:

```haskell
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

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:

```
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

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 \( a \rightarrow b \) is converted to a function of type \( a \rightarrow M b \)
- \( M \) 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
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:

```haskell
hangman :: IO ()
hangman  =
  do putStrLn "Think of a word: "
     word ← sgetLine
     putStrLn "Try to guess it:"
     guess word
```
The action \texttt{sgetLine} reads a line of text from the keyboard, echoing each character as a dash:

\begin{verbatim}
  sgetLine :: IO String
  sgetLine  = do x ← getCh
                if x == '\n' then
                  do putChar x
                  return []
                else
                  do putChar '-'
                      xs ← sgetLine
                  return (x:xs)
\end{verbatim}
Hangman (Cont.)

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

```haskell
import System.IO

getCh :: IO Char
getCh = do hSetEcho stdin False
          c ← getChar
          hSetEcho stdin True
          return c
```
Hangman (Cont.)

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

```haskell
guess :: String → IO ()
guess word =
  do putStr "> 
    xs ← getline
    if xs == word then
      putStrLnLn "You got it!"
    else
      do putStrLnLn (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"
```