Monadic IO

CS 440: Programming Languages and Translators
Lecture 23, Mon 4/22

Simple IO in Haskell

- (Reference: The Haskell wikibook)
- Syllogism?
  - Haskell does things purely and lazily.
  - Input/Output is not pure and not lazy.
  - Therefore Haskell can't do I/O.
- Haskell does support IO, but in a context that keeps us from noticing the side-effecting and eager nature of IO.
- IO operations are referentially opaque
  - As value-producing computations, they can produce different values, so allowing == on them would break referential transparency.
  - E.g., the getline routine reads a string from the keyboard, but we can't allow the test getline == getline because different calls of getline might produce different results.
- Haskell handles this by creating a Monad type operator\(^1\), IO, and restricting how I/O operations work.
  - So getline returns an IO String, not a String. You can't just use IO String values as strings (you can't ++, for example), but you can use the monad operators to access the string and do things with it.
  - In particular, you can use getline >>= \var -> .... At runtime, var will get bound to the string read in by getline.
  - An unperformed getline is said to be an IO String action
    - It's a promise that at some future time, we'll get a String value.
    - Once we perform the getline, (Performed action => past time.)
    - But it has to yield the string in way that doesn't allow you to use referential transparency.
- The IO monad has the feature that once you have an IO \( t \) value (any type \( t \)), there's no useful way to get the type \( t \) value out of it. (There are useless ways like constant-returning functions: \( \text{\io\_t\_val} \rightarrow 17 \).)
  - The IO monad types are not instances of Eq, so you can't use == on them.
  - This is the main reason why you can create values of a type like IO String but you can't convert them to values of plain old type String: String supports == but IO String can't.
  - You can't just ++ two IO String values. but you can take ++ and use to build an operation that works on two IO String values, say getline >>= \str1 -> getline >>= \str2 -> ...
  - Once you start using IO values, you can manipulate them into other IO values.

\(^1\) Recall generally, type operations take a type/types and yields a new type. Other type operations include ->, "tuple-ing", as in \((\ldots)\), and "list of": \([\ ]\)
In fact, with IO, there's no way to stop using IO values.

There's a pure :: t -> m t function but no unsafe :: m t -> t (except for constant functions like \
x -> 17)

More generally, an IO t value means that at runtime, you'll get a value of type t through some opaque means.

The putStrLn function is of type :: String -> IO ()

In some sense, the IO part holds a printer object [not in OO sense]

Saying this another way

I/O actions are modeled as IO monad values; actions aren't functions, so they don't need to be lazy or pure.

The IO operations are referentially opaque, but there's no way to use == to realize that's true.

That's why we can't ask getline == getline.

If this really bothers you, think of it this way:

Each I/O call and returned value has a clock time attached to it.

The clock part is not visible to us; there's also a clock -> set-of-all-device-states function that's not visible to us.

By definition, the time changes with each call or result.

E.g. getline at time 1 returns "Hi" time 2, getline at time 3 returns "Hi" time 4

So the two getline results are not equal.

Printing Output

putStrLn "some string" prints the string and returns a value of type IO ()

The () is because printing doesn't return a value. The type () (pronounced nil) has just one value, also spelled ()

Printing "modifies" the printer by taking printer with output so far and yielding a (similar) printer with more output. The printer object is hidden (so we don't have to deal with it).

Similarly, getline function yields value of type IO String.

The keyboard is hidden, and the read-in string read is returned, but not as a string.

Can't use getline as if it returns a string:

putStrLn getline fails because putStrLn is of type String -> IO () but getline is of type IO String.

IO is a Monad, so to get at the String from an IO String, we can use >>=

Example: read line, print it out twice (on same line) surrounded by "**"

gline >>= \str -> putStrLn ("**" ++ str ++ " " ++ str ++ "**")

Haskell also allows some syntactic sugar to make it easier.
**Do block**

- Holds a sequence of **actions**; instead of >>=, it uses a syntax reminiscent of assignment
  - Instead of \( \text{exp}_1 >>= \var \rightarrow \text{exp}_2 \), we use \( \text{do} \ var \leftarrow \text{exp}_1 \ ; \ \text{exp}_2 \)
  - E.g., use \( \var \leftarrow \text{getLine} \) to read a line and bind the string to a variable.
- Example
  
  ```hs
  getline >>= \str \rightarrow \text{putStrLn} \ ("**" ++ \str ++ " " ++ \str ++ "**")
  ```
  becomes
  
  ```hs
  do \str \leftarrow \text{getLine}; \text{putStrLn} \ ("**" ++ \str ++ " " ++ \str ++ "**")
  ```
- On multiple lines (recall we need \{ ... \} if in ghci)

  ```hs
  \{ \\
  do \str \leftarrow \text{getLine}; -- note semicolon here \\
  \text{putStrLn} \ ("**" ++ \str ++ " " ++ \str ++ "**") \\
  \}
  ```
  (indentation note: \p \text{in} \text{putStrLn} \text{lines up with} \s \text{in} \str \text{above})

  or even

  ```hs
  \{ \\
  do \str \leftarrow \text{getLine} -- note semicolon on next line \\
  ; \text{putStrLn} \ ("**" ++ \str ++ " " ++ \str ++ "**") \\
  \}
  ```
  (indentation note: ; \text{lines up with} \s \text{in} \str \text{above})
- The ; isn't needed if you put the <- and putStrLn on different lines:

  ```hs
  \{ \\
  do \str \leftarrow \text{getLine} \\
  \text{putStrLn} \ ("**" ++ \str ++ " " ++ \str ++ "**") \\
  \}
  ```
- We can break up the print into multiple lines if we want.

  ```hs
  \{ \\
  do \str \leftarrow \text{getLine} \\
  \text{putStr} "**" -- \text{putStr} \text{doesn't print a line end like putStrLn} \\
  \text{putStr} \ \str \\
  \text{putStr} " " \\
  \text{putStr} \ \str \\
  \text{putStrLn} "**" \\
  \}
  ```
- **Actions** are things that produce monadic values, and you can only use the var <- exp notation on a monadic value, since \( \text{non-monadic}\_\text{exp} >>= \ \var \rightarrow \ldots \) has a type error.
- To use a non-monadic value, use \textit{let}

  ```hs
  do \str \leftarrow \text{getLine} ; \text{out} \leftarrow \str ++ " " ++ \str ; \text{putStrLn} \ \text{out} \ -- \text{fails}
  ```
do str <- getLine ; let out = str ++ " " ++ str in putStrLn out -- OK

- On multiple lines:
  :
  do str <- getLine
     let out = str ++ " " ++ str in
     putStrLn out
  :
- You can use if / then / else inside a do, but the branches need to be actions.
- If a branch of a conditional needs to sequence actions, use a nested "do"
  :
  do
     putStrLn "enter: "
     s <- getLine
     if s == ""
        then putStrLn "empty?"
     else do putStrLn "not empty"
     putStrLn ("length " ++ show (length s))
  :

- The sides of a conditional have to have the same type, so they need to be actions of the same type.
  - In the example above, if one side had ended in a getChar, then the two sides would have had different types (IO String and IO ()).
  - The way around that is to use the monadic return expr. E.g., return () would be used as an IO () value. (In detail, it would start out as a SomeMonadType () value. When the type checker sees that the other conditional arm has type IO (), then it would decide to use IO for SomeMonadType.)

IO as a Functor

- It turns out that IO is a Functor: It supports \texttt{fmap :: (a -> b) -> IO a -> IO b}. For example, we can use \texttt{fmap} to take an IO String value from \texttt{getLine} and run a String -> something function.
  
  **Example:** Read a line and append " there". We can't use \texttt{getLine ++ " there"} because the types don't match (IO String ++ String fails). We can use \texttt{fmap}, however.

  -- append_there :: IO [Char]
  append_there = fmap (++ " there") getline

- The result of \texttt{append_there} is an IO String; luckily, the interpreter can deal with a value of IO String (it prints the string).

> append_there

  hi           -- my input
  "hi there"   -- the IO String result
• **Example:** Read in an integer and multiply it by 6. Note \((\ast\ 6)\ .\ \text{read}\) is the function that composes the "multiply by 6" function with the \text{read} function: It takes a \text{String}, runs it through \text{read} to get a number, and multiplies the number by 6. I.e., it is equivalent to (\str \rightarrow (\text{read str}) \ast 6).

```haskell
-- mult_6 :: IO Int
mult_6 = fmap ((\* 6) . read) getLine
> mult_6
 7 -- my input
42 -- IO Int result
```

• We must use the result of \text{mult}_{6} as a monadic value: If we want to use the actual number, we need \text{<-}.

```haskell
> do n <- mult_6 ; putStrLn ("aha! " ++ show (n*3))
7
aha! 126
```

• To get an \text{IO String} from a \text{String}, use function \text{pure} or \text{return}. (For monads, \text{pure} and \text{return} are the same. For a type that is a functor but not a monad, use \text{pure}.)

```haskell
> fmap (++ " there") (pure "hi")
"hi there"
```

• \text{IO} is an applicative functor: It supports an \(<*\) operation that's similar to \text{fmap} and lets you work with \text{IO} values.

\[(<*\) :: IO (a \rightarrow b) \rightarrow IO a \rightarrow IO b\]

• (Compare with \text{fmap} :: \((a \rightarrow b) \rightarrow IO a \rightarrow IO b\).)

• The type system uses \text{IO} types to sequester all the side-effecting operations.
  • Since there's no useful way to get from an \text{IO a} to a non-monic value, all \text{IO-based} computations and their results are separate from the completely pure part of the language.
  • Within code that uses \text{IO}, we can still use referential transparency to reason about pure manipulations of values obtained by side effects.