IO

Haskell and Cryptocurrencies
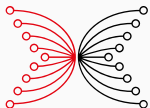
Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG
Antonio Ibarra, IOG

February, 2024

INPUT | OUTPUT

Goals

- Recap: explicit effects.
- Simple IO programs.
- Building larger IO programs.
- Reconciling IO and the functional style.

Explicit effects

- Given lazy evaluation as a strategy, the moment of evaluation is not easy to predict and hence not a good trigger for side-effecting actions.
- Even worse, it may be difficult to predict whether a term is evaluated at all.
- We would like to keep equational reasoning, and allow compiler optimisations such as
  - strictness analysis – evaluating things earlier than needed if they will definitely be needed, or
  - speculative evaluation – evaluating things even if they might not be needed at all.

Assume for the time being:

```
getLine :: String
```

Assume for the time being:

```
getLine :: String
```

```
program1 = getLine ++ getLine
program2 = (\ x -> x ++ x) getLine
program3 = (\ x y -> y ++ x) getLine getLine
```

Explicit effects are a good idea

- We can see via the type of a program whether it is guaranteed to have no side effects, or whether it is allowed to use effects.
- In principle, we can even make more fine-grained statements than just yes or no, by allowing just specific classes of effects.
- Encourages a programming style that keeps as much as possible effect-free.
- Makes it easier to test programs, or to run them in a different context.

```
data IO a   -- abstract
```

The type of plans to perform effects that ultimately yield an a .

```
data IO a   -- abstract
```

The type of plans to perform effects that ultimately yield an `a`.

- Evaluation does not trigger the actual effects. It will at most evaluate the plan.
- Execution triggers the actual effects. Executing a plan is not possible from within a Haskell program.

```
main :: IO ()
```

- The entry point into the program is a plan to perform effects (a possibly rather complex one).
- This is the one and only plan that actually gets executed.

```
data () = ()   -- special syntax
```

Constructor:

```
() :: ()
```

- A type with a single value (nullary tuple).
- Often used to parameterize other types.
- A plan for actions with no interesting result: `IO ()`.

For convenience, GHCi also executes IO actions:

```
GHCi> getLine
Some text.
"Some text."
```

```
getLine :: IO String
```

A plan that when executed, reads a line interactively and returns that line as a `String`.

Execution of effects with unit results in GHCi

GHCi does not print the final result of `IO ()`-typed actions:

```
GHCi> writeFile "test.txt" "Hello"
GHCi> putStrLn "two\nlines"
two
lines
```

```
writeFile :: FilePath -> String -> IO ()
putStrLn :: String -> IO ()
```

Constructing larger plans

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

Function that takes two plans and constructs a plan that first executes the first plan, discard its result, then executes the second plan, and returns its result.

```
getTwoLines :: IO String
getTwoLines = getLine >> getLine
```

```
getTwoLines :: IO String
getTwoLines = getLine >> getLine
```

```
GHCi> getTwoLines
Line 1.
Line 2.
"Line 2."
```

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

Takes a function and a plan. Constructs a plan that executes the given plan, but before returning the result, applies the function.

```
duplicateLine :: IO String
duplicateLine = liftM (\ x -> x ++ x) getLine
```

Modifying the result of a plan

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

Takes a function and a plan. Constructs a plan that executes the given plan, but before returning the result, applies the function.

```
duplicateLine :: IO String
duplicateLine = liftM (\ x -> x ++ x) getLine
```

```
GHCi> duplicateLine
Hello
"HelloHello"
```

Shouting

```
GHCi> :t toUpper
toUpper :: Char -> Char
GHCi> toUpper 'x'
'X'
GHCi> liftM (map toUpper) getLine
Hello
"HELLO"
```

Combining the output of two sequenced plans

```
liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c
```

Takes an operator and two plans. Constructs a plan that
executes the two plans in sequence, and uses the operator to
combine the two results.

```
joinTwoLines :: IO String
joinTwoLines = liftM2 (++) getLine getLine
```

Combining the output of two sequenced plans

```
liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c
```

Takes an operator and two plans. Constructs a plan that executes the two plans in sequence, and uses the operator to combine the two results.

```
joinTwoLines :: IO String
joinTwoLines = liftM2 (++) getLine getLine
```

```
GHCi> joinTwoLines
Hello
world
"Helloworld"
```

Joining and flipping two lines

```
flipTwoLines :: IO String
flipTwoLines =
  liftM2 (\ x y -> y ++ x) getLine getLine
```

Joining and flipping two lines

```
flipTwoLines :: IO String
flipTwoLines =
  liftM2 (\ x y -> y ++ x) getLine getLine
```

```
GHCi> flipTwoLines
Hello
world
"worldHello"
```

Revisiting the problematic examples

Wrong:

```
program1 = getLine ++ getLine
program2 = (\ x -> x ++ x) getLine
program3 = (\ x y -> y ++ x) getLine getLine
```

Revisiting the problematic examples

Wrong:

```
program1 = getLine ++ getLine
program2 = (\ x -> x ++ x) getLine
program3 = (\ x y -> y ++ x) getLine getLine
```

Better:

```
joinTwoLines1 = liftM2 (++) getLine getLine
joinTwoLines2 = (\ x -> liftM2 (++) x x) getLine
joinTwoLines3 =
  (\ x y -> liftM2 (++) y x) getLine getLine
duplicateLine = liftM (\ x -> x ++ x) getLine
flipTwoLines =
  liftM2 (\ x y -> y ++ x) getLine getLine
```

Actions that depend on the results of earlier actions

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

Shouting back

Transforms the result (but does not print it back):

```haskell
shout :: IO String
shout = liftM (map toUpper) getLine
```

Transforms the result (but does not print it back):

```
shout :: IO String
shout = liftM (map toUpper) getLine
```

```
shoutBack :: IO ()
shoutBack = shout >>= putStrLn
```

Shouting back

Transforms the result (but does not print it back):

```
shout :: IO String
shout = liftM (map toUpper) getLine
```

```
shoutBack :: IO ()
shoutBack = shout >>= putStrLn
```

```
(>>=)             :: IO a -> (a -> IO b) -> IO b
shout             :: IO String
putStrLn          :: String -> IO ()
shout >>= putStrLn :: IO ()
```

```
shoutBackTwice :: IO ()
shoutBackTwice =
  shout >>= \ x -> putStrLn x >> putStrLn x
```

```
GHCi> shoutBack
Hello
HELLO
```

```
GHCi> shoutBackTwice
can you hear me?
CAN YOU HEAR ME?
CAN YOU HEAR ME?
```

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

An plan that when executed, perform no effects and returns
the given result.

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

An plan that when executed, perform no effects and returns the given result.

- Intuitively, `IO a` says that we may use effects to obtain an `a`. We are not required to.
- On the other hand, `a` says that we must not use effects to obtain an `a`.

No escape from IO!

There is no[1] function

```
runIO :: IO a -> a
```

---

[1]There actually is one, called `unsafePerformIO` , but its use is generally not justified.

No escape from IO!

There is no[1] function

```
runIO :: IO a -> a
```

If a value requires effects to obtain, we should not ever
pretend that it does not.

---

[1]There actually is one, called `unsafePerformIO`, but its use is generally
not justified.

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

- Gives us access to the `a` that results from the first action.
- But wraps it all up in another `IO` action.

```
(>>) :: IO a -> IO b -> IO b
a1 >> a2 = a1 >>= \ _ -> a2
```

```
(>>) :: IO a -> IO b -> IO b
a1 >> a2 = a1 >>= \ _ -> a2
```

Or:

```
(>>) :: IO a -> IO b -> IO b
ioa >> iob = ioa >>= const iob

const :: a -> b -> a
const a b = a
```

```
liftM :: (a -> b) -> IO a -> IO b
liftM f ioa = ioa >>= \ a -> return (f a)
```

```
liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c
liftM2 f ioa iob =
  ioa >>= \ a -> iob >>= \ b -> return (f a b)
```

do notation

```
liftM2 :: (a -> b -> c) -> IO a -> IO b -> IO c
```

```
liftM2 f ioa iob =
  ioa >>= \ a ->
  iob >>= \ b ->
  return (f a b)
```

do notation

```
liftM2 :: (a -> b -> c ) -> IO a -> IO b -> IO c
```

```
liftM2 f ioa iob =          liftM2 f ioa iob = do
  ioa >>= \ a ->              a <- ioa
  iob >>= \ b ->              b <- iob
  return (f a b)             return (f a b)
```

A larger example

```haskell
greeting :: IO ()
greeting =
  putStrLn "What is your name?"    >>
  getLine                          >>= \ name ->
  putStrLn ("Hello, " ++ name ++ "!") >>
  putStrLn "Where do you live?"    >>
  getLine                          >>= \ loc ->
  let
    answer
      | loc == "Kenya"  = "Fantastic!"
      | loc == "Uganda" = "Outstanding!"
      | otherwise       = "Sorry, don't know that."
  in
    putStrLn answer
```

A larger example

```haskell
greeting :: IO ()
greeting = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
  putStrLn "Where do you live?"
  loc <- getLine
  let
    answer
      | loc == "Kenya" = "Fantastic!"
      | loc == "Uganda" = "Outstanding!"
      | otherwise       = "Sorry, don't know that."
  putStrLn answer
```

Functional programming with IO

```haskell
ask :: String -> IO String
ask question = do
  putStrLn question
  getLine
```

```
ask :: String -> IO String
ask question = do
  putStrLn question
  getLine
```

```
GHCi> ask "What is your name?"
What is your name?
Lars
"Lars"
```

```
askMany :: [String] -> IO [String]
askMany []       = return []
askMany (q : qs) = do
  answer  <- ask q
  answers <- askMany qs
  return (answer : answers)
```

The standard design pattern on lists is back!

Feels like a map

A `map` has the wrong result type:

```
askMany' :: [String] -> [IO String]
askMany' = map ask
```

Feels like a map

A `map` has the wrong result type:

```
askMany' :: [String] -> [IO String]
askMany' = map ask
```

But we can sequence a list of plans:

```
sequence :: [IO a] -> IO [a]
sequence []       = return []
sequence (x : xs) = do
  a  <- x
  as <- sequence xs
  return (a : as)
```

```
mapM :: (a -> IO b) -> [a] -> IO [b]
mapM f xs = sequence (map f xs)
```

```
mapM :: (a -> IO b) -> [a] -> IO [b]
mapM f xs = sequence (map f xs)
```

```
askMany :: [String] -> IO [String]
askMany questions = mapM ask questions
```

Traversing a tree interactively

A tree of yes-no questions

```
data Interaction =
    Question String Interaction Interaction
  | Result String
```

Constructors:

```
Question ::
  String
   -> Interaction -> Interaction -> Interaction
Result   :: String -> Interaction
```

Pick a language

```
pick :: Interaction
pick =
  Question "Do you like FP?"
    (Question "Do you like static types?"
      (Result "Try OCaml.")
      (Result "Try Clojure.")
    )
    (Question "Do you like dynamic types?"
      (Result "Try Python.")
      (Result "Try Rust.")
    )
```

```
ford :: Interaction
ford =
  Question "Would you like a car?"
    (Question "Do you like it in black?"
      (Result "Good for you.")
      ford
    )
    (Result "Never mind then.")
```

Asking a Boolean question

```
askBool :: String -> IO Bool
askBool question = do
  putStrLn (question ++ " [yn]")
  x <- getChar
  putStrLn ""
  return (x `elem` "yY")
```

```haskell
interaction :: Interaction -> IO ()
interaction (Question q y n) = do
  b <- askBool q
  if b then interaction y else interaction n
interaction (Result r) = putStrLn r
```

```haskell
simulate :: Interaction -> [Bool] -> Maybe String
simulate (Question _ y _) (True  : bs) =
  simulate y bs
simulate (Question _ _ n) (False : bs) =
  simulate n bs
simulate (Result r) []                 = Just r
simulate _              _              = Nothing
```

Acquiring and releasing resources

```haskell
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Handle-based file IO

All in `System.IO`:

```
hGetLine   :: Handle -> IO String
hPutStrLn :: Handle -> String -> IO ()
hIsEOF     :: Handle -> IO Bool
```

Handle-based file IO

All in `System.IO`:

```haskell
hGetLine   :: Handle -> IO String
hPutStrLn  :: Handle -> String -> IO ()
hIsEOF     :: Handle -> IO Bool
```

```haskell
withFile ::
  FilePath -> IOMode
   -> (Handle -> IO r)  -- continuation (aka callback)
   -> IO r
```

```haskell
data IOMode =
    ReadMode | WriteMode
  | AppendMode | ReadWriteMode
```

Reading a file line by line

```haskell
readFileLineByLine :: FilePath -> IO [String]
readFileLineByLine file =
  withFile file ReadMode readFileHandle

readFileHandle :: Handle -> IO [String]
readFileHandle h = do
  eof <- hIsEOF h
  if eof
    then return []
    else do
      line <- hGetLine h
      lines <- readFileHandle h
      return (line : lines)
```

Handle is automatically released at end of continuation.

Warning

Both *readFile* and *readFileLineByLine* are actually problematic for different reasons.

We will learn about better ways to process (in particular large) files later in the course.

Exceptions

GHCi> readFileLineByLine "doesnotexist"
*** Exception: doesnotexist: openFile: does not exit
(No such file or directory)

Exceptions in effectful vs effect-free code

Exceptions in pure code (via `error`, missing patterns, …) are bad:

- It is unclear when exactly, or if, they will be triggered,
- It is therefore also unclear where or when to best handle them,
- Explicitly handling failure via `Maybe` or similar is almost always the better solution.

Exceptions in effectful vs effect-free code

Exceptions in pure code (via `error`, missing patterns, …) are bad:

- It is unclear when exactly, or if, they will be triggered,
- It is therefore also unclear where or when to best handle them,
- Explicitly handling failure via `Maybe` or similar is almost always the better solution.

Exceptions in effectful (`IO`) code are different:

- Execution order is explicit, and handling is easier.
- There are *many* things that go wrong.

Catching IO errors

From `System.IO.Error`:

```
catchIOError :: IO a -> (IOError -> IO a) -> IO a
```

Catching IO errors

From `System.IO.Error`:

```
catchIOError :: IO a -> (IOError -> IO a) -> IO a
```

```
readFileLineByLine' ::
  FilePath -> IO (Maybe [String])
readFileLineByLine' file =
  catchIOError
    (liftM Just (readFileLineByLine file))
    (const (return Nothing))
```

Testing it

```
GHCi> writeFile "test" "foo\nbar"
GHCi> readFileLineByLine' "test"
Just ["foo", "bar"]
GHCi> removeFile "test"
GHCi> readFileLineByLine' "test"
Nothing
```

From `System.Directory`:

```
removeFile :: FilePath -> IO ()
```

Recap

- The role of the `IO` type.
- Composing `IO` functions.
- Higher-order `IO` functions (`sequence`, `mapM`).
- File IO.
- Resources.
- Exceptions.