Part 5: IO and explicit effects

Introduction to Haskell

Andres Löh

Well-Typed
The Haskell Consultants

Goals

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

Explicit effects

The original motivation for 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.

The classic approach

In most languages, **execution** of side effects is tied to **evaluation** of the side-effecting expression.

This is feasible for languages with eager evaluation, because the order in which expressions are written down corresponds closely to the resulting order of evaluation.

With lazy evaluation, this is not the case ...

Problematic programs

Assume for the time being:

```
getLine :: String
```

Problematic programs

Assume for the time being:

```
getLine :: String
```

Consider:

```
program1 =
  let
    x = getLine
    y = getLine
  in
    x ++ y
```

```
program2 =
  let
    x = getLine
  in
    x ++ x
```

```
program3 =
  let
    x = getLine
    y = getLine
  in
    y ++ x
```

Well-Typed

Problematic programs

Assume for the time being:

```
getLine :: String
```

Consider:

```
program1 =
  let
    x = getLine
    y = getLine
  in
    x ++ y
```

```
program2 =
  let
    x = getLine
  in
    x ++ x
```

```
program3 =
  let
    x = getLine
    y = getLine
  in
    y ++ x
```

If evaluation triggers the effect and evaluation is lazy, then when and how far we look at the resulting string will determine if and when lines are being read.

Problematic programs

Assume for the time being:

```
getLine :: String
```

Consider:

```
program1 =
  let
    x = getLine
    y = getLine
  in
    x ++ y
```

```
program2 =
  let
    x = getLine
  in
    x ++ x
```

```
program3 =
  let
    x = getLine
    y = getLine
  in
    y ++ x
```

If evaluation triggers the effect and evaluation is lazy, then when and how far we look at the resulting string will determine if and when lines are being read.

Using equational reasoning, all three programs should mean the same.

The Haskell approach

We do **not** tie the **execution** of side effects to **evaluation**.

The Haskell approach

We do **not** tie the **execution** of side effects to **evaluation**.

We introduce a new datatype `IO` and make **evaluation** and **execution** separate concepts!

Well-Typed

Evaluation vs. execution

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

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

Well-Typed

Evaluation vs. execution

```haskell
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.

The main program

```haskell
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.

The unit type

```
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 ()` .

Execution of effects via GHCi

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 ()
```

Explicit effects are a good idea

(Not just in Haskell, not just in a lazily evaluated language.)

- 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.

Constructing larger plans

Basic sequencing

```
(>>) :: 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.

Reading two lines

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

Well-Typed

Reading two lines

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

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

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
```

Modifying the result of a plan

```haskell
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.

```haskell
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

```haskell
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.

```haskell
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

```haskell
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
```

Well-Typed

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

Bind: letting an action use an earlier result

```
(>>=) :: 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
```

Shouting back

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 ()
```

Well-Typed

Shouting back twice

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

In GHCi

```
GHCi> shoutBack
Hello
Hello
```

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

Optioning out of doing IO

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

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

Optioning out of doing IO

```
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.

Escaping temporarily

```
(>>=) :: 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.

Bind is the most general sequencing function

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

Bind is the most general sequencing function

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

Well-Typed

Bind and return can implement lifting

```
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

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

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

do notation

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

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

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

A larger example

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

More about `do`

A corner case is a single IO action:

```
helloWorld = do
  putStrLn "Hello world"
```

is the same as writing

```
helloWorld =
  putStrLn "Hello world"
```

More about `do`

A corner case is a single IO action:

```
helloWorld = do
  putStrLn "Hello world"
```

is the same as writing

```
helloWorld =
  putStrLn "Hello world"
```

Remember that a `do` is never required. It is just “syntactic sugar” for a chain of `(>>=)` and `(>>)` applications.

Nested `do`

```
loop :: IO ()
loop = do
  putStrLn "Type 'q' to quit."
  c <- getChar   -- reads a single character
  if c == 'q'
    then putStrLn "Goodbye"
    else do
      putStrLn "Here we go again ..."
      loop
```

If we want to embed a sequence commands into a subexpression and use `do` -notation for that, we need another `do` .

Nested do

```haskell
loop :: IO ()
loop = do
  putStrLn "Type 'q' to quit."
  c <- getChar   -- reads a single character
  if c == 'q'
    then putStrLn "Goodbye"
    else do
      putStrLn "Here we go again ..."
      loop
```

If we want to embed a sequence commands into a subexpression and use `do` -notation for that, we need another `do` .

Note furthermore that there are lots of `do` blocks without `return` .

About `return`

- The purpose of `return` in Haskell is to embed computations into the `IO` type.
- As such, `return` can be used in many different places.
- It is fine to use `return` in the middle of a sequence of commands. It does **not** jump anywhere.

This is fine:

```
do
  n <- return 2
  print n
```

Functional programming with IO

Asking a question

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

Asking a question

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

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

Asking many questions

```haskell
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)
```

Mapping an IO action

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

Mapping an IO action

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

```haskell
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

```haskell
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.")
    )
```

Pick a car

```
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.")
```

Well-Typed

Asking a Boolean question

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

Traversing the tree interactively

```
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
```

Traversing the tree non-interactively

```
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

Whole-file IO

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

Well-Typed

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` :

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

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

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

Reading a file line by line

```
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.

A word of warning

Warning

Both `readFile` and `readFileLineByLine` are actually problematic for different reasons.
We will (probably) learn about better ways to process (in particular large) files later this week.

Exceptions

What happens if the file does not exist?

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

Well-Typed

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.