IO and Monads

Lecture 6 of CSE 3100 Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

Lecture plan

- Effectful programming with the `IO` type
- The `Monad` typeclass
- `Monad` examples: `Maybe`, `Either`, `[]`

The `IO` type


CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Impure operations

Recap. A function is called pure if its behaviour is fully described by how it maps its inputs to its outputs.

What are examples of things that violate purity?

Impure operations

Recap. A function is called pure if its behaviour is fully described by how it maps its inputs to its outputs.

What are examples of things that violate purity?

- Mutable variables
- Exceptions
- Input and output
- File system access
- Network communication

Question. How to write Haskell programs that interact with the world?

Question. How to write Haskell programs that interact with the world?

Answer. Use the builtin Haskell type `IO a`!

What is `IO`?

`IO a` is the type of programs that interact with the world and return a value of type `a`.

Examples.

- `putStrLn "Hello" :: IO ()`
- `getLine :: IO String`

Unlike other builtin types such as `[a]` or `(a,b)`, we cannot give a definition of `IO` ourselves: it is built into Haskell.

Executing IO actions

An expression of type `IO a` is called an action.

Actions can be passed around and returned like any Haskell type, but they are not executed except in specific cases:

- `main :: IO ()` is executed when the whole program is executed.
- GHCi will also execute any action it is given.
- Other actions are only executed when called by another action.

Separating the pure from the impure

Haskell programs are separated into a pure part (that does *not* use `IO`) and an impure (or *effectful*) part (that does use `IO`).

We can convert a pure value (of type `a`) to an impure one (of type `IO a`), but not the other way. Once a value is in `IO`, it is stuck there!

Advice. Most Haskell programs should only use `IO` in a small part of the program ($\sim$ 10%).

`IO` primitives: terminal I/O

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()

getChar :: IO Char
getLine :: IO String
readLn  :: Read a => IO a
```

`IO` primitives: reading and writing files

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

openFile   :: FilePath -> IOMode
           -> IO Handle
hPutStrLn  :: Handle -> String -> IO ()
hGetLine   :: Handle -> IO String
hIsEOF     :: Handle -> IO Bool
hClose     :: Handle -> IO ()
```

Other things you can do with `IO`

- Generate random numbers
- Read and write mutable variables (`IORef`)
- Throw and catch IO exceptions
- Write graphics on the display
- Listen to keyboard and mouse events
- Communicate over the network
- Start other processes
- …

Basically: anything that's not a pure function

`do` notation

Haskell has a special syntax for writing sequences of `IO` actions, known as `do` notation:

```
main :: IO ()
main = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ "!")
```

Anatomy of a `do` block

```haskell
f :: IO a
f = do
  v1 <- a1
  ...
  vn <- an
  f v1 ... vn
```

- Each `vi <- ai` is a statement with an action `ai :: IO bi`.
- The result `vi` of each statement can be used *as a pure value of type* `bi` in the rest of the do block.
- The final line must be an `IO` action of type `IO a` (not a statement).

The function `return` (1/2)

The function `return :: a -> IO a` turns a pure value into an `IO` action:

It is often used at the end of a `do` block:

```
f :: a -> IO (b,c)
f x = do
  y <- g x
  z <- h x
  return (y,z)
```

The function `return` (2/2)

Warning. The `return` function is unrelated to `return` in imperative languages.

```
main = do
  putStr "Spam"
  return ()
  putStr " and eggs"
  return ()
-- Output: Spam and eggs
```

A `return` in the middle of a `do` block does not terminate the function!

Conditional actions with `when`

`when :: Bool -> IO () -> IO ()`
executes an action only when the condition is `True`:

```
main = do
  putStrLn "Pick a password:"
  pwd <- getLine
  when (length pwd < 6) $
    putStrLn "Warning: too short"
  ... -- do some more stuff
```

Running a list of actions with `sequence`

`sequence :: [IO a] -> IO [a]` runs a list of `IO` actions and collect the results.

```
main = do
  putStrLn "Pick three colors"
  colors <- sequence
    [getLine, getLine, getLine]
  putStrLn "The sorted colors are:"
  sequence
    (map putStrLn (sort colors))
  return ()
```

`IO` is a functor

`fmap f act` applies the pure function `f` to the result of the `IO a` action `act`, producing a new `IO b` action.

Example. If `parse :: String -> Expr` then `fmap parse getLine :: IO Expr`.

So we can view `act :: IO a` as a 'box' holding a value of type `a` that will be available at run-time.

Applicative instance for `IO`

We could define an `Applicative IO` instance as follows:

```
instance Applicative IO where
  pure x = return x
  mf <*> mx = do
    f <- mf
    x <- mx
    return (f x)
```

This is not the real definition[1] but it gives the right idea.

[1] `do`-notation requires a `Monad` instance, which requires an

Live programming question

Write a small interactive Haskell program with a function `main` that stores a list of items. It should have a simple menu with three options:

1. List all current items on the list in sequence
2. Add a new item to the list
3. Remove the item at a given position in the list

Bonus: Add an option for saving/loading the list from a file.

The hidden backdoor: `unsafePerformIO`

The module `System.IO.Unsafe` provides a function `unsafePerformIO :: IO a -> a` that allow executing an `IO` action inside a pure function.

Warning (!!). Due to laziness, predicting if and when the action will be executed is **very hard!**

Only use `unsafePerformIO` for debugging or better understanding of laziness, but avoid using it in real programs.

Tracing Haskell functions

A common use of unsafe IO is the function `trace :: String -> a -> a`:

```haskell
fib :: Int -> Int
fib 0 = trace ("fib 0") 1
fib 1 = trace ("fib 1") 1
fib n = trace ("fib " ++ show n)
  (fib (n-1) + fib (n-2))
```

By adding `trace` to your functions, you can get a better understanding of how Haskell functions are evaluated at runtime.

Monads

Monads: impossible to understand?

A monad is just a monoid in the category of endofunctors, what's the problem?

…or completely trivial?

You already know everything there is to know about monads!

- Examples: `Maybe`, `Either`, `IO`, …
- Higher-order functions
- Type classes
- Functors and applicative functors
- `do`-notation

The Maybe monad

Walk the line[2]

```haskell
type Birds = Int
type Pole  = (Birds,Birds)

checkBalance :: Pole -> Maybe Pole
checkBalance (x,y)
  | abs (x-y) < 4 = Just (x,y)
  | otherwise     = Nothing

landL :: Birds -> Pole -> Maybe Pole
landL n (x,y) = checkBalance (x+n,y)

landR :: Birds -> Pole -> Maybe Pole
landR n (x,y) = checkBalance (x,y+n)
```

[2] http://learnyouahaskell.com/a-fistful-of-monads

Walk the line

```
landingSequence :: Pole -> Maybe Pole
landingSequence pole0 =
  case landL 1 pole0 of
    Nothing -> Nothing
    Just pole1 -> case landR 4 pole1 of
      Nothing -> Nothing
      Just pole2 -> case landL (-1) pole2 of
        Nothing -> Nothing
        Just pole3 -> case landR (-2) pole3 of
          Nothing -> Nothing
          Just pole4 -> Just pole4
```

Can't we do better??

Walk the line

Can't we do better?? Yes we can!

```haskell
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f = Nothing
Just x  >>= f = f x

landingSequence :: Pole -> Maybe Pole
landingSequence pole0 =
  Just pole0
  >>= landL 1
  >>= landR 4
  >>= landL (-1)
  >>= landR (-2)
```

The bind operator

The function `(>>=)` is called bind.

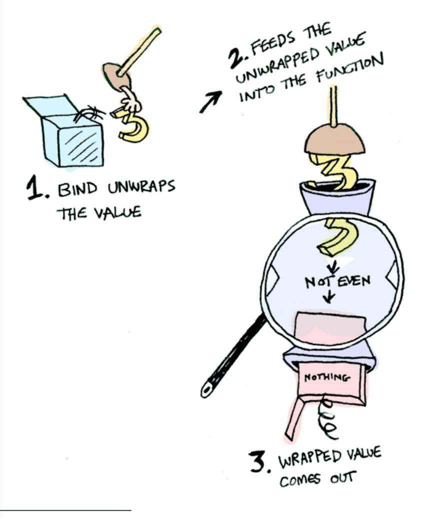
The general type of bind is:

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

where the type constructor `m` is a monad.

It is so useful, it made it into the Haskell logo:

The bind operator[3]



[3] https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

The Monad class

`Functor`, `Applicative`, and `Monad`

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

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

Two ways to think about monads

You can think about a monadic type `m a` as…

- … a container data structure that holds values of type `a`.
- … a computation that can perform some side effects before returning a value of type `a`.

Both ways are valid and can be useful in different situations!

Some terminology on monads

A monad is a type constructor that is an instance of the `Monad` type class.

A monadic type is a type of the form `m a` where `m` is a monad.

An action is an expression of a monadic type.

A monadic function is a function that returns an action.

The sequencing operator `(>>)`

The sequencing operator `(>>)` executes one action after another, ignoring the output of the first.

```
(>>) :: Monad m => m a -> m b -> m b
mx >> my = mx >>= (\_ -> my)
```

More monadic functions

```
-- do the operation if the boolean is `True`
when :: Applicative f => Bool -> f () -> f ()

-- do the operation if the boolean is `False`
unless :: Applicative f => Bool -> f () -> f ()

-- run the actions from left to right.
sequence :: Monad m => [m a] -> m [a]

-- run the monadic function to each element
-- in the list, combining the results.
traverse :: Applicative f =>
              (a -> f b) -> [a] -> f [b]
```

do notation for monads

We can use do notation for any monad!

```haskell
landingSequence :: Pole -> Maybe Pole
landingSequence pole0 = do
  pole1 <- landL 1 pole0
  pole2 <- landR 4 pole1
  pole3 <- landL (-1) pole2
  pole4 <- landR (-2) pole3
  return pole4
```

Desugaring of `do` notation

With `do`-notation

```haskell
do
  x <- f
  g x
  y <- h x
  return (p x y)
```

Without `do`-notation

```haskell
f >>= (\x ->
  g x >> (
    h x >>= (\y ->
      return (p x y)
    )
  )
)
```

Monads in other languages

Several other languages use monads too:

- The `std::optional` library defines the `Maybe` monad in C++
- The `flatMap` function in Scala is precisely `(>>=)` for the list monad
- Promises in JavaScript (and other languages) form a monad[4], with `.then()` acting as `(>>=)`.

[4]Almost, but not quite: see `https://hackernoon.com/functional-javascript-functors-monads-and-promises-679ce`

The Promise monad in JavaScript

`async`/`await` is `do`-notation:[5]

```
async function getBalances() {
  const accounts =
    await web3.eth.accounts();
  const balances =
    await Promise.all(
      accounts.map(web3.eth.getBalance));
  return Ramda.zipObject(balances, accounts);
}
```

[5] https://gist.github.com/MaiaVictor/bc0c02b6d1fbc7e3dbae838fb1376c80

Quiz: Which is which?

`Monad`		function
`IO ()`		action
`Maybe`	...is a ...	type
`return 42`		monad
`(>>=)`		type class

What's next?

Next lecture: The `State` and `Parser` monads

To do:

- Read the book:
  - Today: 10.1-10.5, 12.3
  - Next lecture: 13.1-13.8
- Finish week 3 exercises on Weblab