Functional Programming

IO Monad

Jevgeni Kabanov

Department of Computer Science
University of Tartu

Overview

1. Functional IO
2. Monadic IO
3. Higher Order IO

Outline

1 Functional IO

2 Monadic IO

3 Higher Order IO

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Lazy Pure Haskell

Haskell is a *pure* and *lazy* language

Pure

- Purity means that every function result is uniquely determined by its parameters
- Compiler is free to inline any function as it pleases
- Runtime can cache any function call

Lazy

- Laziness means that values are computed on need
- If they are not needed, they are not computed
- Runtime is free to precompute them or reorder computations

Pure IO

The following function should read a character from standard input

$$getChar :: Char$$

Let's see an example:

$$get2chars = [getChar, getChar]$$

Questions

- How many characters will be read?
- In what order will they be read?

Pure IO

The following function should read a character from standard input

$$getChar :: Char$$

Let's see an example:

$$get2chars = [getChar, getChar]$$

Questions

- How many characters will be read?
- In what order will they be read?

Pure IO

The following function should read a character from standard input

$$getChar :: Char$$

Let's see an example:

$$get2chars = [getChar, getChar]$$

Questions

- How many characters will be read?
- In what order will they be read?

Pure IO

The following function should read a character from standard input

$$getChar :: Char$$

Let's see an example:

$$get2chars = [getChar, getChar]$$

Questions

- How many characters will be read?
- In what order will they be read?

Pure IO

Problem 1

Let's try to solve these problems:

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

Now the values are distinct, yet the order is still undefined

Problem 2

Also we now need to add the same *Int* argument to *get2chars* for the same reason:

$$get4chars :: Int \to [Char]$$
$$get4chars = get2chars\ 1 +\!\!+ get2chars\ 2$$

Pure IO

Problem 1

Let's try to solve these problems:

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

Now the values are distinct, yet the order is still undefined

Problem 2

Also we now need to add the same *Int* argument to *get2chars* for the same reason:

$$get4chars :: Int \to [Char]$$
$$get4chars = get2chars\ 1 +\!\!+ get2chars\ 2$$

Pure IO

Problem 1

Let's try to solve these problems:

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

Now the values are distinct, yet the order is still undefined

Problem 2

Also we now need to add the same *Int* argument to *get2chars* for the same reason:

$$get4chars :: Int \to [Char]$$
$$get4chars = get2chars\ 1 +\!\!+ get2chars\ 2$$

Lazy IO

Problem 3

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

- What will be called first?
- What we really need is some kind of dependency!
- The only kind of runtime dependency Haskell provides is value dependency:

$$getchar :: Int \to (Char, Int)$$
$$get2chars\ \_ = [a, b]\ \mathbf{where}\ (a, i) = getChar\ 1$$
$$(b, \_) = getChar\ i$$

Lazy IO

Problem 3

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

- What will be called first?
- What we really need is some kind of dependency!
- The only kind of runtime dependency Haskell provides is value dependency:

$$getchar :: Int \to (Char, Int)$$
$$get2chars\ \_ = [a, b]\ \mathbf{where}\ (a, i) = getChar\ 1$$
$$(b, \_) = getChar\ i$$

Lazy IO

Problem 3

$$getChar :: Int \to Char$$
$$get2chars = [getChar\ 1, getChar\ 2]$$

- What will be called first?
- What we really need is some kind of dependency!
- The only kind of runtime dependency Haskell provides is value dependency:

$$getchar :: Int \to (Char, Int)$$
$$get2chars\ \_ = [a, b]\ \mathbf{where}\ (a, i) = getChar\ 1$$
$$(b, \_) = getChar\ i$$

Lazy IO

Problem 4

The problem now is *get2chars* – the parameter is neither used nor unique and compiler might still reorder or omit it.

$$get4chars = [get2chars\ 1, get2chars\ 2]$$

What we need is a unique parameter associated with *every* IO action:

$$\begin{aligned}
&get2chars :: Int \to (String, Int)\\
&get2chars\ i0 = ([a, b], i2)\ \mathbf{where}\ (a, i1) = getChar\ i0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i2) = getChar\ i1\\
&get4chars\ i0 = (a +\!\!+ b, i2)\ \mathbf{where}\ (a, i1) = get2chars\ i0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i2) = get2chars\ i1
\end{aligned}$$

Lazy IO

Problem 4

The problem now is *get2chars* – the parameter is neither used nor unique and compiler might still reorder or omit it.

$$get4chars = [get2chars\ 1, get2chars\ 2]$$

What we need is a unique parameter associated with *every* IO action:

$$\begin{aligned}
&get2chars :: Int \to (String, Int)\\
&get2chars\ i0 = ([a, b], i2)\ \mathbf{where}\ (a, i1) = getChar\ i0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i2) = getChar\ i1\\
&get4chars\ i0 = (a +\!\!+ b, i2)\ \mathbf{where}\ (a, i1) = get2chars\ i0\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (b, i2) = get2chars\ i1
\end{aligned}$$

Lazy IO

Problem 5

However there is still one problem left, can you see it?:

$$get2chars\ i0 = ([a, b], i2)\ \textbf{where}\ (a, i1) = getChar\ i0$$
$$(b, i2) = getChar\ i0$$

A simple mistake and our carefully built up IO is ruined!

NB!

What we need is a way for compiler to ensure that the value is unique!

Lazy IO

Problem 5

However there is still one problem left, can you see it?:

$$get2chars\ i0 = ([a, b], i2)\ \textbf{where}\ (a, i1) = getChar\ i0 \\ (b, i2) = getChar\ i0$$

A simple mistake and our carefully built up IO is ruined!

NB!

What we need is a way for compiler to ensure that the value is unique!

Lazy IO

Problem 5

However there is still one problem left, can you see it?:

$$get2chars\ i0 = ([a, b], i2)\ \textbf{where}\ (a, i1) = getChar\ i0$$
$$(b, i2) = getChar\ i0$$

A simple mistake and our carefully built up IO is ruined!

NB!

What we need is a way for compiler to ensure that the value is unique!

Outline

1. Functional IO
2. Monadic IO
3. Higher Order IO

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Some History

When Haskell was created three main ways for doing pure, lazy IO were known.

1. Input-output streams
2. Continuations
3. Threaded unique value

Of course Haskellers chose the fourth way!

Clean

Interestingly enough a Haskell-like language Clean chose to introduce *uniqueness typing* that made the third way possible directly

Real World IO

We continue the previous example by interpreting the threaded value as the token of the world state:

$$getChar :: RealWorld \to (Char, RealWorld)$$

In that case the program is of type

$$main :: RealWorld \to ((), RealWorld)$$

We can introduce a type synonym and rewrite it as:

$$\mathbf{type}\ IO\ a = RealWorld \to (a, RealWorld)$$

$$main :: IO\ ()$$

$$getChar :: IO\ Char$$

Real World IO

We continue the previous example by interpreting the threaded value as the token of the world state:

$$getChar :: RealWorld \rightarrow (Char, RealWorld)$$

In that case the program is of type

$$main :: RealWorld \rightarrow ((), RealWorld)$$

We can introduce a type synonym and rewrite it as:

$$\mathbf{type}\ IO\ a = RealWorld \rightarrow (a, RealWorld)$$

$$main :: IO\ ()$$

$$getChar :: IO\ Char$$

Hello, Monad!

Let's see a program reading two characters from input:

$$
\begin{aligned}
&main\ world0 = \mathbf{let}\ (a, world1) = getChar\ world0 \\
&\quad (b, world2) = getChar\ world1 \\
&\quad \mathbf{in}\ ([a, b], world2)
\end{aligned}
$$

We could hide the token from the programmer by threading it automatically:

$$
\begin{aligned}
&(>\!\!>\!\!=) :: IO\ a \to (a \to IO\ b) \to IO\ b \\
&(action1 >\!\!>\!\!= action2)\ world0 = \\
&\quad \mathbf{let}\ (a, world1) = action1\ world0 \\
&\qquad (b, world2) = action2\ a\ world1 \\
&\quad \mathbf{in}\ (b, world2) \\
&main = \\
&\quad getChar >\!\!>\!\!= \lambda a \to getChar >\!\!>\!\!= \lambda b \to (\lambda w \to ([a, b], w))
\end{aligned}
$$

Hello, Monad!

Let's see a program reading two characters from input:

$$
\begin{aligned}
&main\ world0 = \mathbf{let}\ (a, world1) = getChar\ world0 \\
&\quad (b, world2) = getChar\ world1 \\
&\quad \mathbf{in}\ ([a, b], world2)
\end{aligned}
$$

We could hide the token from the programmer by threading it automatically:

$$
\begin{aligned}
&(>\!\!>\!=) :: IO\ a \to (a \to IO\ b) \to IO\ b \\
&(action1 >\!\!>\!= action2)\ world0 = \\
&\quad \mathbf{let}\ (a, world1) = action1\ world0 \\
&\quad\quad (b, world2) = action2\ a\ world1 \\
&\quad \mathbf{in}\ (b, world2) \\
&main = \\
&\quad getChar >\!\!>\!= \lambda a \to getChar >\!\!>\!= \lambda b \to (\lambda w \to ([a, b], w))
\end{aligned}
$$

Doing It with Class

Now we can make $IO\ a$ a monad!

```
instance Monad (IO a) where
  (>>=) :: IO a → (a → IO b) → IO b
  (action1 >>= action2) world0 =
    let (a, world1) = action1 world0
        (b, world2) = action2 a world1
    in (b, world2)
  return :: a → IO a
  return a = (λworld → (a, world))
```

We can now use the *do*-notation so the previous example becomes

```
main = do
  a ← getChar
  b ← getChar
  return [a, b]
```

Doing It with Class

Now we can make $IO\ a$ a monad!

$$
\begin{array}{l}
\textbf{instance}\ Monad\ (IO\ a)\ \textbf{where} \\
\quad (\gg\!=) :: IO\ a \to (a \to IO\ b) \to IO\ b \\
\quad (action1 \gg\!= action2)\ world0 = \\
\qquad \textbf{let}\ (a, world1) = action1\ world0 \\
\qquad\quad (b, world2) = action2\ a\ world1 \\
\qquad \textbf{in}\ (b, world2) \\
\quad return :: a \to IO\ a \\
\quad return\ a = (\lambda world \to (a, world))
\end{array}
$$

We can now use the *do*-notation so the previous example becomes

$$
\begin{array}{l}
main = \textbf{do} \\
\quad a \leftarrow getChar \\
\quad b \leftarrow getChar \\
\quad return\ [a, b]
\end{array}
$$

Interlude

Reflections

- It is logical to make $IO\ a$ abstract, so that programmer would be protected from making mistakes.
- Then a value of $IO\ a$ is nothing more than an *action* or a *computation* to be executed.
- However this also means that all IO functions must be predefined with priority access to IO monad.

IO functions

IO module necessarily contains functions for working with:

- Files and file systems
- Mutable variables
- Random numbers
- System clock

Which makes it pretty big!

Interlude

Reflections

- It is logical to make $IO\ a$ abstract, so that programmer would be protected from making mistakes.
- Then a value of $IO\ a$ is nothing more than an *action* or a *computation* to be executed.
- However this also means that all IO functions must be predefined with priority access to IO monad.

IO functions

IO module necessarily contains functions for working with:

- Files and file systems
- Mutable variables
- Random numbers
- System clock

Which makes it pretty big!

Interlude

Reflections

- It is logical to make $IO\ a$ abstract, so that programmer would be protected from making mistakes.
- Then a value of $IO\ a$ is nothing more than an *action* or a *computation* to be executed.
- However this also means that all IO functions must be predefined with priority access to IO monad.

IO functions

IO module necessarily contains functions for working with:

- Files and file systems
- Mutable variables
- Random numbers
- System clock

Which makes it pretty big!

Interlude

Reflections

- It is logical to make $IO\ a$ abstract, so that programmer would be protected from making mistakes.
- Then a value of $IO\ a$ is nothing more than an *action* or a *computation* to be executed.
- However this also means that all IO functions must be predefined with priority access to IO monad.

IO functions

IO module necessarily contains functions for working with:

- Files and file systems
- Mutable variables
- Random numbers
- System clock

Which makes it pretty big!

Outline

1. Functional IO
2. Monadic IO
3. Higher Order IO

Console Operations

Definition

Most of console functions are self-explanatory:

$$putChar :: Char \to IO\ ()$$
$$putStr :: String \to IO\ ()$$
$$putStrLn :: String \to IO\ ()$$

$$print :: Show\ a \Rightarrow a \to IO\ ()$$
$$readLn :: Read\ a \Rightarrow IO\ a$$

$$getChar :: IO\ Char$$
$$getLine :: IO\ String$$
$$getContents :: IO\ String$$

Console Operations

Example

```
main = do
  putStr "What is your name?"
  a ← readLn
  putStr "How old are you?"
  b ← readLn
  print (a, b)
```

IO Actions as Values

Example

Let's define a list of IO actions:

$$
\begin{aligned}
&ioActions :: [IO\ ()] \\
&ioActions = \\
&\quad [(print\ \texttt{"Hello!"}), \\
&\quad (putStr\ \texttt{"just kidding"}), \\
&\quad (getChar \gg return\ ())]
\end{aligned}
$$

Remember, that $IO\ a = RealWorld \to (a, RealWorld)$, so these are usual functional values.

IO Actions as Values

Example

However these IO actions can also be used directly:

```
main = do
  head ioActions
  ioActions !! 1
  last ioActions
```

Output:

```
"Hello!"
just kidding(Wait for  input)
```

IO Action Lists

Definition

Sequencing a list of actions can be reused (*foldr* is just a reminder):

$$
\begin{aligned}
&\mathit{foldr} && :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ z\ [\,] = && z \\
&\mathit{foldr}\ f\ z\ (x : xs) = && f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sequence}\_ :: \mathit{Monad}\ m \Rightarrow [m\ a] \to m\ () \\
&\mathit{sequence}\_ = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ())
\end{aligned}
$$

A lot more interesting monad combinators are available and we will examine them later.

IO Action Lists

Definition

The previous example then becomes:

```
ioActions :: [IO ()]
ioActions =
  [(print "Hello!"),
  (putStr "just kidding"),
  (getChar >> return ())]
main = do sequence_ ioActions
```

Output:

```
"Hello!"
just kidding(Wait for  input)
```

File Operations

Definition

Files are opened as *Handles* and can be used accordingly:

$$
\begin{aligned}
&\mathit{openFile} :: \mathit{String} \to \mathit{IOMode} \to \mathit{IO}\ \mathit{Handle} \\
&\mathit{hSeek} :: \mathit{Handle} \to \mathit{SeekMode} \to \mathit{Integer} \to \mathit{IO}\ () \\
&\mathit{hGetChar} :: \mathit{Handle} \to \mathit{IO}\ \mathit{Char} \\
&\mathit{hPutChar} :: \mathit{Handle} \to \mathit{Char} \to \mathit{IO}\ () \\
&\mathit{hClose} :: \mathit{Handle} \to \mathit{IO}\ ()
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{data}\ \mathit{SeekMode} \\
&\quad = \mathit{AbsoluteSeek} \\
&\quad \mid \mathit{RelativeSeek} \\
&\quad \mid \mathit{SeekFromEnd}
\end{aligned}
$$

File Operations

Example

Let's define a function reading one character from a given handle and position:

```
readi h i = do
  hSeek h i AbsoluteSeek
  hGetChar h
```

Next we want to open the file and just read characters from it:

```
readfilei :: String → IO (Integer → IO Char)
readfilei name = do
  h ← openFile name ReadMode
  return (readi h)
```

File Operations

Example

Let's define a function reading one character from a given handle and position:

```
readi h i = do
  hSeek h i AbsoluteSeek
  hGetChar h
```

Next we want to open the file and just read characters from it:

```
readfilei :: String → IO (Integer → IO Char)
readfilei name = do
  h ← openFile name ReadMode
  return (readi h)
```

File Operations

Example

Since IO functions can be higher order we can apply *readi* repeatedly:

$$
\begin{aligned}
&main = \mathbf{do} \\
&\quad myfile \leftarrow readfilei\ \texttt{"test"} \\
&\quad a \leftarrow myfile\ 0 \\
&\quad b \leftarrow myfile\ 1 \\
&\quad print\ (a, b)
\end{aligned}
$$

IORef

Definition

$IORef\ a$ allows to create mutable variables:

$$\begin{aligned}
&\textbf{data}\ IORef\ a \\
&newIORef :: a \to IO\ (IORef\ a) \\
&readIORef :: IORef\ a \to IO\ a \\
&writeIORef :: IORef\ a \to a \to IO\ () \\
&modifyIORef :: IORef\ a \to (a \to a) \to IO\ ()
\end{aligned}$$

IORef

Example

$$
\begin{aligned}
&main = \mathbf{do} \\
&\quad varA \leftarrow newIORef\ 0 \\
&\quad a0 \leftarrow readIORef\ varA \\
&\quad writeIORef\ varA\ 1 \\
&\quad a1 \leftarrow readIORef\ varA \\
&\quad print\ (a0, a1)
\end{aligned}
$$

Haskell Objects

Figure

Let's try to define a *Figure* object:

$$\begin{aligned}&\textbf{data } Figure = Figure\{\\&\quad draw :: IO\ (), move :: Displacement \to IO\ ()\}\\&\textbf{type } Displacement = (Int, Int)\end{aligned}$$

We will define only one figure – a circle:

$$\begin{aligned}&circle :: Point \to Radius \to IO\ Figure\\&\textbf{type } Point = (Int, Int) \quad \text{-- point coordinates}\\&\textbf{type } Radius = Int \quad \text{-- circle radius in points}\end{aligned}$$

Haskell Objects

Drawing

We can test the objects when we define them with this code:

```
main = do
  figures ← sequence [circle (10,10) 5,
    rectangle (10,10) (20,20)]
  drawAll figures
  mapM _ (λfig → move fig (10,10)) figures
  drawAll figures

drawAll :: [Figure] → IO ()
drawAll figures = do
  putStrLn "Drawing figures:"
  mapM _ draw figures
```

Circle

```
circle center radius = do
  centerVar ← newIORef center
  let drawF = do
    center ← readIORef centerVar
    putStrLn ("   Circle at " ++ show center
      ++ " with radius " ++ show radius)
  let moveF (addX, addY) = do
    (x, y) ← readIORef centerVar
    writeIORef centerVar (x + addX, y + addY)
  return $ Figure{ draw = drawF, move = moveF }
```

Rectangle

```
rectangle from to = do
  fromVar ← newIORef from
  toVar ← newIORef to
  let drawF = do
    from ← readIORef fromVar
    to ← readIORef toVar
    putStrLn ("   Rectangle " ++ show from
      ++ "-" ++ show to)
  let moveF (addX, addY) = do
    (fromX, fromY) ← readIORef fromVar
    (toX, toY) ← readIORef toVar
    writeIORef fromVar (fromX + addX, fromY + addY)
    writeIORef toVar (toX + addX, toY + addY)
  return $ Figure{ draw = drawF, move = moveF }
```

Higher Order IO

Lazy IO

How lazy is the IO monad?

```
list1, list2 :: IO [Int]
list1 = return ∘ repeat $ 0
list2 = sequence ∘ repeat ∘ return $ 0

main = do
  lst ← list1
  putStrLn ∘ show ∘ take 10 $ lst
  readLn
  lst2 ← list2
  putStrLn ∘ show ∘ take 10 $ lst2
```

Higher Order IO

Lazy IO

How lazy is the IO monad?

```
list1, list2 :: IO [Int]
list1 = return ∘ repeat $ 0
list2 = sequence ∘ repeat ∘ return $ 0

main = do
  lst ← list1
  putStrLn ∘ show ∘ take 10 $ lst
  readLn
  lst2 ← list2
  putStrLn ∘ show ∘ take 10 $ lst2
```

The first list will show the first 10 elements, while the second one will go infinite.

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad

Conclusions

Advantages

- IO in Haskell is pure and lazy
- The order of operations that end up executed is guaranteed by the compiler
- IO actions are treated in Haskell first-class, they can be passed around, combined and curried

Disadvantages

- Every new function that needs to do side-effects needs access to IO monad implementation
- IO actions tend to contaminate pure code, lifting it to IO monad because some particular parts need to do IO
- Since IO operations are the fastest, a large portion of a program may end up under IO monad