Building a Console Application in Haskell

Rebecca Skinner

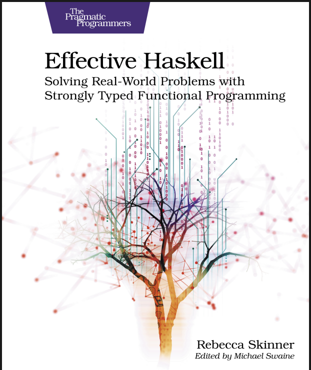
2022-12-07

Prelude

Hello, World

- About Me: Rebecca Skinner
  - Lead Software Engineer at Mercury
  - Author of Effective Haskell
- @cercerilla on Twitter and Cohost
- https://rebeccaskinner.net
- https://github.com/rebeccaskinner/

Effective Haskell



https://tinyurl.com/2744kfu7

Now in Beta!

About This Talk

During this talk we'll build basic command line tool in Haskell. As we go, you'll:

- Learn how Haskell programs use IO actions to deal with the real world
- Find out how to do simple terminal and file IO
- See examples of how to mix IO and pure functional code effectively
- Follow along with implementing pure functional code to work with text

Most importantly: You'll get an intuition for how to think about building Haskell programs that can serve as a basis for future learning.

HCat

HCat: A Haskell Pager

- Get a file name from the command line
- Print the contents of the file to the screen one page at a time
- After showing a page of text, wait for user input
- Format each page so it fits on the screen
- Allow the user to quit at any time

```
{-# LANGUAGE RecordWildCards #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TypeApplications #-}
module HCat where

import qualified System.IO.Error as IOError
import qualified Control.Exception as Exception

import qualified System.Environment as Environment
import qualified System.Exit as Exit
import qualified Data.List as List
import qualified Data.Maybe as Maybe
import qualified Data.Char as Char
import qualified Text.Printf as Printf
import qualified Data.Text as Text
import qualified Data.Text.IO as TextIO
import qualified Data.ByteString as BS
import System.IO ( openFile
                 , hPutStr
                 , hFlush
                 , hClose
                 , hFileSize
                 , hGetContents
                 , stdin
                 , stdout
                 , hGetChar
                 , hSetBuffering
                 , hGetBuffering
                 , hSetEcho
                 , BufferMode (..)
                 , Handle
                 , IOMode (..) )
import qualified System.Info
import qualified System.Process as Process
import Text.Read
import Control.Monad
import qualified Data.Time.Clock as Clock
import qualified Data.Time.Format as TimeFormat

-- import qualified System.Posix.User as PosixUser
import qualified System.Directory as Directory


--START:FileInfo
./src/HCat.hs | permissions: rw- | 19337 bytes | modified: 2022-01-25 04:15:03 | page: 1 of 15
```

Starting a New Project

Building HCat

Setting Up Your Development Environment

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Editor If you already use `emacs` or `vim` keep using them, otherwise `vscode`

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Editor If you already use `emacs` or `vim` keep using them, otherwise `vscode`

REPL `ghci` should be your best friend when you're learning Haskell.

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Editor If you already use `emacs` or `vim` keep using them, otherwise `vscode`

REPL `ghci` should be your best friend when you're learning Haskell.

Docs Learn `hoogle` early, and use it often

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Editor If you already use `emacs` or `vim` keep using them, otherwise `vscode`

REPL `ghci` should be your best friend when you're learning Haskell.

Docs Learn `hoogle` early, and use it often

IDE Tooling `HLS` with `vscode` will give you the most IDE-like experience

Setting Up Your Development Environment

Installation	Use `ghcup` to install Haskell.
Build Tooling	Use `cabal` to create one project per application you are building
Editor	If you already use `emacs` or `vim` keep using them, otherwise `vscode`
REPL	`ghci` should be your best friend when you're learning Haskell.
Docs	Learn `hoogle` early, and use it often
IDE Tooling	`HLS` with `vscode` will give you the most IDE-like experience
Testing	Test manually with `ghci` or use `hspec` but only test pure functions.

Setting Up Your Development Environment

Installation Use `ghcup` to install Haskell.

Build Tooling Use `cabal` to create one project per application you are building

Editor If you already use `emacs` or `vim` keep using them, otherwise `vscode`

REPL `ghci` should be your best friend when you're learning Haskell.

Docs Learn `hoogle` early, and use it often

IDE Tooling `HLS` with `vscode` will give you the most IDE-like experience

Testing Test manually with `ghci` or use `hspec` but only test pure functions.

Linting Use `hlint`, but feel free to ignore anything you don't understand

Creating A New Project

It's best to create a new project for each application. Avoid globally installed dependencies.

Creating A New Project

It's best to create a new project for each application. Avoid globally installed dependencies.

```
user@host$ cabal init --interactive
```

Creating A New Project

It's best to create a new project for each application. Avoid globally installed dependencies.

```
user@host$ cabal init --interactive
```

As a rule of thumb, each project should define:

- One library, named after the project. This is where most of your code will live
- One (or, occasionally, more than one) executable. These usually have very little code
- A test suite
- Optionally, benchmarks

Dependencies

You can browse available packages on hackage. To add a new dependency to your project, add it to your cabal file.

```
library
  hs-source-dirs:       src
  exposed-modules:      HCat
  build-depends:        base, directory, process
  ghc-options:          -Wall
  default-language:     Haskell2010
```

Executables and Libraries

It's common to have a very minimal executable:

```haskell
module Main where
import HCat (runHCat)

main :: IO ()
main = runHCat
```

Executables and Libraries

It's common to have a very minimal executable:

```
module Main where
import HCat (runHCat)

main :: IO ()
main = runHCat
```

Plus a library is where most code is implemented:

```
module HCat (runHCat) where

runHCat :: IO ()
runHCat = print "stub"
```

Executables and Libraries

It's common to have a very minimal executable:

```haskell
module Main where
import HCat (runHCat)

main :: IO ()
main = runHCat
```

Plus a library is where most code is implemented:

```haskell
module HCat (runHCat) where

runHCat :: IO ()
runHCat = print "stub"
```

Why?

Executables and Libraries

It's common to have a very minimal executable:

```haskell
module Main where
import HCat (runHCat)

main :: IO ()
main = runHCat
```

Plus a library is where most code is implemented:

```haskell
module HCat (runHCat) where

runHCat :: IO ()
runHCat = print "stub"
```

Why?

- Testability
- Easy Refactoring
- Low overhead to add additional executables

Writing Our First Version

Let's start with an MVP. We'll make a program like echo that:

- reads the contents of a file
- then prints the contents to the screen

Writing Our First Version

Let's start with an MVP. We'll make a program like echo that:

- reads the contents of a file
- then prints the contents to the screen

There's just one problem: Haskell is a pure functional language. Doesn't that mean no side effects?

Understanding IO

The Trouble with IO

Haskell is a pure functional language, but most of the things we want our programs to do revolve around side effects!

The Trouble with IO

Haskell is a pure functional language, but most of the things we want our programs to do revolve around side effects!

- Reading and writing files

The Trouble with IO

Haskell is a pure functional language, but most of the things we want our programs to do revolve around side effects!

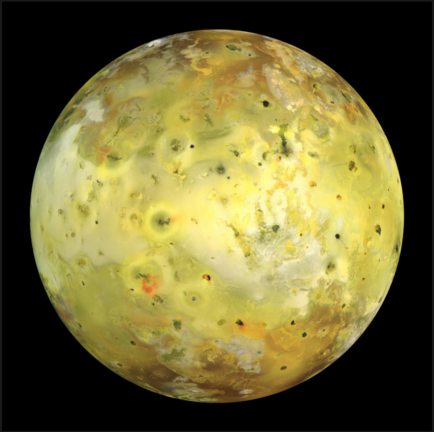
- Reading and writing files
- Printing text to the screen

The Trouble with IO

Haskell is a pure functional language, but most of the things we want our programs to do revolve around side effects!

- Reading and writing files
- Printing text to the screen
- Handling user input

A True Color Photo of Side Effects



A side effect in its natural environment.

Can We Have a Little Bit of IO?

What if we cheat just a little?

Can We Have a Little Bit of IO?

What if we cheat just a little?

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

Can We Have a Little Bit of IO?

What if we cheat just a little?

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

- Nothing will happen until we evaluate `writeReadFile`

Can We Have a Little Bit of IO?

What if we cheat just a little?

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

- Nothing will happen until we evaluate `writeReadFile`
- When we evaluate `writeReadFile` we'll get whatever random contents were in `example.txt`

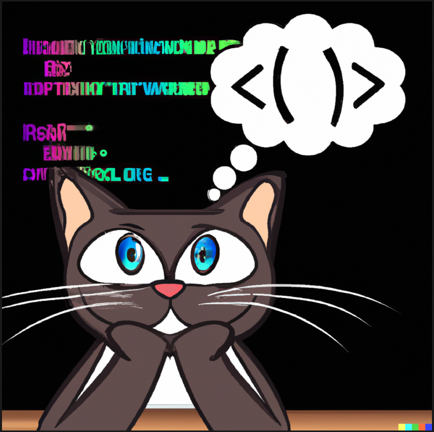
Can We Have a Little Bit of IO?

What if we cheat just a little?

```
writeReadFile =
  let
    _ = writeFile "example.txt" "Hello, Haskell"
    fileContents = readFile "example.txt"
  in print fileContents
```

- Nothing will happen until we evaluate `writeReadFile`
- When we evaluate `writeReadFile` we'll get whatever random contents were in `example.txt`
- We won't ever write "Hello, Haskell" to the file, because we're not using result of `writeFile`!

Let’s Dream of a Better Way



Let’s dream up a better way

IO, the Lazy Way

If we want to be lazy, we need to work for it by making sure every new side effect must depend on the previous one.

Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects:

Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects:

- Reading a file, then printing the contents

Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects:

- Reading a file, then printing the contents

More often, there isn't an obvious dependency:

Sometimes Things Are Easy

In some cases, there is a natural dependency between side effects:

- Reading a file, then printing the contents

More often, there isn't an obvious dependency:

- Writing a log message before opening a file
- Writing data to a file, then reading the contents
- Printing a message to the screen then waiting on user input

A Pointer To The Real World

We needed to sequence our side effects correctly because there's an implicit data dependency we haven't considered: the state of the real world.

We needed to sequence our side effects correctly because there's an implicit data dependency we haven't considered: the state of the real world.

```
data RealWorld
```

Welcome to the Real World

We can use a reference to the `RealWorld` to add a dependency between all of our calls:

Welcome to the Real World

We can use a reference to the `RealWorld` to add a dependency between all of our calls:

```
writeReadFile world0 =
  let
    (world1, _) = writeFile world0 "example.txt" "Hello, Haskell"
    (world2, fileContents) = readFile world1 "example.txt"
  in print world2 fileContents
```

Welcome to the Real World

We can use a reference to the `RealWorld` to add a dependency between all of our calls:

```
writeReadFile world0 =
  let
    (world1, _) = writeFile world0 "example.txt" "Hello, Haskell"
    (world2, fileContents) = readFile world1 "example.txt"
  in print world2 fileContents
```

But it sucks.

Typing IO Operations

Let's make a type!

Typing IO Operations

```
data SideEffect a =
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }
```

Side Effects Are Programs

Think of `SideEffect a` as a program that returns a value of type a.

`SideEffect String` : A program that runs and outputs a `String`

`SideEffect Int` : A program that runs and outputs an `Int`

`SideEffect` programs are not pure functional programs. They rely on, and change, the `RealWorld`.

Side Effect Examples

Let's look at some examples of `SideEffect` programs. We'll imagine some internal helper functions that will do the unsafe low level IO operations:

```
readFile :: FilePath -> SideEffect String
readFile filename = SideEffect $ \realWorld ->
  let (realWorld', contents) = internalReadFile filename realWorld
  in (realWorld', contents)

writeFile :: FilePath -> String -> SideEffect ()
writeFile filename contents = SideEffect $ \realWorld ->
  let realWorld' = internalWriteFile filename contents realWorld
  in (realWorld', ())

print :: String -> SideEffect ()
print message = SideEffect $ \realWorld ->
  let realWorld' = internalPrint message realWorld
  in (realWorld', ())
```

Combining Side Effects

A `SideEffect` program can do things that have side effects, like reading from and writing to files, but that's pretty limiting. We can do a lot more if we can have a `SideEffect` program that executes other `SideEffect` programs and uses the results.

```
data SideEffect a =
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }

joinSideEffects :: SideEffect (SideEffect a) -> SideEffect a
joinSideEffects outerSideEffect = SideEffect $ \world ->
  let (world', innerSideEffect) = runSideEffects outerSideEffect world
  in runSideEffects innerSideEffect world'
```

First One, Then The Other

Most of the time, we want to write a SideEffect program that does one side effect and then does another one. It turns out that this is just another way of saying that we have one SideEffect program that calls the first effect, and uses it's value to call the second one:

```
data SideEffect a =
  SideEffect { runSideEffects :: RealWorld -> (RealWorld, a) }

sequenceSideEffects :: SideEffect a -> (a -> SideEffect b) -> SideEffect b
sequenceSideEffects sideEffect makeNextSideEffect =
  joinSideEffects $ SideEffect $ \world ->
    let (world', val) = runSideEffects sideEffect world
    in (world', makeNextSideEffect val)
```

Write, Read, Print

Let's to to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  `sequenceSideEffects` (\_ -> readFile "example.txt")
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

Write, Read, Print

Let's to to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  `sequenceSideEffects` (\_ -> readFile "example.txt")
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

- Every side effect depends on its predecessor, so they all happen in the right order

Write, Read, Print

Let's to to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  `sequenceSideEffects` (\_ -> readFile "example.txt")
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

- Every side effect depends on its predecessor, so they all happen in the right order
- Our code is focused on the work it needs to do, without having to explicitly pass around references to the real world

Write, Read, Print

Let's to to write our program again, using the things we've just built:

```
writeReadFile :: SideEffect ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  `sequenceSideEffects` (\_ -> readFile "example.txt")
  `sequenceSideEffects` (\contents -> print contents)
```

How does this version compare?

- Every side effect depends on its predecessor, so they all happen in the right order
- Our code is focused on the work it needs to do, without having to explicitly pass around references to the real world
- Our code program is still a pure functional program. Instead of doing side effects directly, we generate a program that would have side effects if it were run. The programs themselves are still pure values.

That's Not All

Before we get back to HCat

That's Not All

Before we get back to HCat

One more thing

That's No Side Effect

```haskell
sayHello :: IO ()
sayHello =
  putStrLn "Hello, World!"
```

```
U:@**-  Hello.hs      18% L5     (Haskell +8 FlyC ivy ElDoc) 12:02AM 0.36
```

Ceci n'est pas une side effect.

That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- Instead of `SideEffect a` we say `IO a`

That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- Instead of `SideEffect a` we say `IO a`
- Instead of `sequenceSideEffects` we say `>>=`

That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- Instead of `SideEffect a` we say `IO a`
- Instead of `sequenceSideEffects` we say `>>=`
- Instead of `SideEffect` program we say `IO` action

That's No Side Effect

It turns out our imaginary `SideEffect` type isn't entirely imaginary.

- Instead of `SideEffect a` we say `IO a`
- Instead of `sequenceSideEffects` we say `>>=`
- Instead of `SideEffect` program we say `IO` action

```
writeReadFile :: IO ()
writeReadFile =
  writeFile "example.txt" "Hello, Haskell"
  >>= (\_ -> readFile "example.txt")
  >>= print
```

To do List

Writing a long chain of calls to »= gets tiresome. Instead we can use do notation:

```
writeReadFile :: IO ()
writeReadFile = do
  writeFile "example.txt" "Hello, Haskell"
  contents <- readFile "example.txt"
  print contents
```

To do List

Writing a long chain of calls to >>= gets tiresome. Instead we can use do notation:

```
writeReadFile :: IO ()
writeReadFile = do
  writeFile "example.txt" "Hello, Haskell"
  contents <- readFile "example.txt"
  print contents
```

- Each line in a do block corresponds to >>=

To do List

Writing a long chain of calls to »= gets tiresome. Instead we can use do notation:

```
writeReadFile :: IO ()
writeReadFile = do
  writeFile "example.txt" "Hello, Haskell"
  contents <- readFile "example.txt"
  print contents
```

- Each line in a do block corresponds to »=
- The <- arrow names the output of an IO action

To do List

Writing a long chain of calls to »= gets tiresome. Instead we can use do notation:

```
writeReadFile :: IO ()
writeReadFile = do
  writeFile "example.txt" "Hello, Haskell"
  contents <- readFile "example.txt"
  print contents
```

- Each line in a do block corresponds to »=
- The <- arrow names the output of an IO action
- When we run a Haskell program, the initial state of the real world is used to run an IO action named `main`.

HCat

Return of the HCat

Back To The Code

Now that we understand how to write code that has side effects and interacts with the real world, let's put it to practice with an MVP:

Back To The Code

Now that we understand how to write code that has side effects and interacts with the real world, let's put it to practice with an MVP:

```
module Main where

runHCat :: IO ()
main = readFile "example.txt" >>= putStrLn
```

The M-est of MVPs

Success! we can read a file and print it out to the screen!

The M-est of MVPs

Success! we can read a file and print it out to the screen!

. . . but only a single hard-coded file

The M-est of MVPs

Success! we can read a file and print it out to the screen!

. . . but only a single hard-coded file

. . . and it's not actually paginated

The M-est of MVPs

Success! we can read a file and print it out to the screen!

. . . but only a single hard-coded file

. . . and it's not actually paginated

. . . or formatted for our terminal window

The M-est of MVPs

Success! we can read a file and print it out to the screen!

. . . but only a single hard-coded file

. . . and it's not actually paginated

. . . or formatted for our terminal window

Let's take one problem at a time

Getting Into Arguments

we need to deal with arguments

Getting Into Arguments

We can use `getArgs` to get command line arguments, but we'll need to deal with user errors.

Getting Into Arguments

We can use `getArgs` to get command line arguments, but we'll need to deal with user errors.

```
module HCat where
import System.Environment

targetFileName :: IO FilePath
targetFileName = do
  args <- getArgs
  case args of
    [filename] ->
      pure filename
    _otherwise ->
      ioError $ userError "please provide a single filename"

runHCat :: IO ()
runHCat = do
  contents <- readFile =<< targetFileName
  putStrLn contents
```

I Am Error

Dealing with errors in IO actions can be complicated because there are a lot of options:

I Am Error

Dealing with errors in IO actions can be complicated because there are a lot of options:

- Plain IO Errors
- Using `Either` or `Maybe` values for failure
- Custom exceptions
- Monad Transformers

I Am Error

Dealing with errors in IO actions can be complicated because there are a lot of options:

- Plain IO Errors
- Using `Either` or `Maybe` values for failure
- Custom exceptions
- Monad Transformers

Opinion: Getting too fancy too early will cause more problems than it solves. Start with the simplest thing that can possibly work.

What About Libraries?

Why parse arguments directly instead of using a library?

What About Libraries?

Why parse arguments directly instead of using a library?

- Handling arguments yourself is good practice while learning
- Some good libraries use language features you probably haven't learned yet

Terminal Size

The size of our terminal will determine our page count. We can get the terminal size with the `tput` program on *nix systems.

Terminal Size

The size of our terminal will determine our page count. We can get the terminal size with the `tput` program on *nix systems.

```haskell
module HCat where
import System.Process
data TerminalDimension = TerminalLines | TerminalCols
data ScreenDimensions =
  ScreenDimensions {screenRows :: Int, screenColumns :: Int}

getTerminalSize :: IO ScreenDimensions
getTerminalSize = do
  termLines <- tput TerminalLines
  termCols <- tput TerminalCols
  pure ScreenDimensions
    { screenRows = termLines
    , screenColumns = termCols }

tput :: TerminalDimension -> IO Int
tput dimension = do
  outputData <- readProcess "tput" [cmd] ""
  pure . read . head . lines $ outputData
  where
    cmd = case dimension of
      TerminalLines -> "lines"
      TerminalCols -> "cols"
```

Word Wrapping

Given the size of our terminal, we can wrap the text to fit.

Word Wrapping

Given the size of our terminal, we can wrap the text to fit.

```haskell
wordWrap :: Int -> String -> [String]
wordWrap lineLength lineText =
  case splitAt lineLength lineText of
    (fullLine, "") -> [fullLine]
    (hardwrappedLine, rest) ->
      let (nextLine, remainder) = softWrap hardwrappedLine
       in nextLine : wordWrap lineLength (remainder <> rest)
  where
    softWrap hardWrapped =
      let (rest, wrappedText) = break isSpace $ reverse hardWrapped
       in (reverse wrappedText, reverse rest)

runHCat :: IO ()
runHCat = do
  contents <- readFile =<< targetFileName
  termSize <- getTerminalSize
  let wrapped = wordWrap (screenColumns termSize) contents
  putStrLn $ unlines wrapped
```

Architecture

A Lesson On Building Things

Let's talk about Architecture

A Tale of Two Word Wraps

We only need the terminal width to word wrap. Maybe we should combine them?

A Tale of Two Word Wraps

We only need the terminal width to word wrap. Maybe we should combine them?

```haskell
wordWrap :: String -> IO [String]
wordWrap lineText = do
  lineLength <- tput TerminalCols
  case splitAt lineLength lineText of
    (fullLine, "") ->
      pure [fullLine]
    (hardwrappedLine, rest) -> do
      let (nextLine, remainder) = softWrap hardwrappedLine
      wrappedRemainder <- wordWrap (remainder <> rest)
      pure (nextLine : wrappedRemainder)
  where
    softWrap hardWrapped =
      let (rest, wrappedText) = break isSpace $ reverse hardWrapped
       in (reverse wrappedText, reverse rest)
```

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API
- Doesn't create "extra" functions

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API
- Doesn't create "extra" functions

Unfortunately. . .

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API
- Doesn't create "extra" functions

Unfortunately. . .

- It can only be called by other IO actions

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API
- Doesn't create "extra" functions

Unfortunately. . .

- It can only be called by other IO actions
- We don't know what it might do. Perhaps it makes a network request to a word wrap server?

A Tale of Two Word Wraps

Coming from impure languages, mixing IO and pure code feels natural:

- Hides implementation details about getting the terminal width
- Provides a simpler and more automated API
- Doesn't create "extra" functions

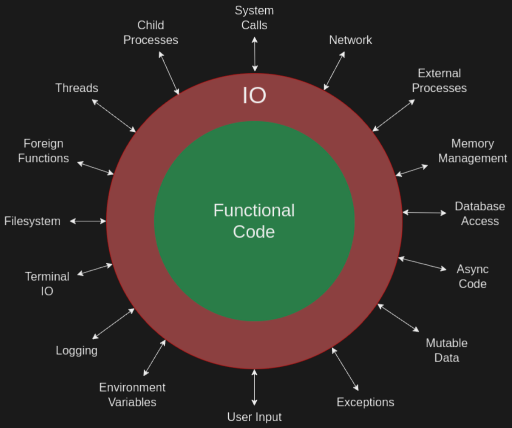
Unfortunately...

- It can only be called by other IO actions
- We don't know what it might do. Perhaps it makes a network request to a word wrap server?
- We'll have a harder time testing it

The Lesson

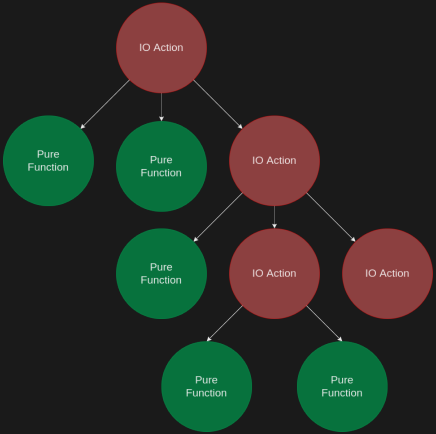
As much as possible, have IO actions gather data then pass it into pure functions for computation.

Procedural Shell, Functional Core



The "procedural shell, functional core" model is an over-simplification of a good guideline

Happy Little Trees



IO Actions and pure functions more closely resemble a tree

Back to HCat

Back to HCat

Back to our regularly scheduled HCat Presentation

Pagination

Our pager has one big problem right now: It doesn't paginate.

Pagination

Our pager has one big problem right now: It doesn't paginate.

```
paginate :: ScreenDimensions -> String -> [String]
paginate dimensions text = pages
  where
    rows = screenRows dimensions
    cols = screenColumns dimensions
    wrappedLines = concatMap (wordWrap cols) (lines text)
    pages = map (unlines . padTo rows) $ groupsOf rows wrappedLines
    padTo lineCount rowsToPad =
      take lineCount $ rowsToPad <> repeat ""
    groupsOf n elems
      | null elems = []
      | otherwise =
        let (hd, tl) = splitAt n elems
        in hd : groupsOf n tl
```

The Event Loop

If we want to show our user a page at a time, we need to do a few things:

The Event Loop

If we want to show our user a page at a time, we need to do a few things:

- Get some user input

The Event Loop

If we want to show our user a page at a time, we need to do a few things:

- Get some user input
- Loop over each page, displaying them

The Event Loop

If we want to show our user a page at a time, we need to do a few things:

- Get some user input
- Loop over each page, displaying them
- Exit cleanly if the user wants to quit

Getting User Input

```haskell
data ContinueCancel
  = Continue
  | Cancel
  deriving stock (Eq, Show)

getContinue :: IO ContinueCancel
getContinue = do
  hSetBuffering stdin NoBuffering
  hSetEcho stdin False
  input <- getChar
  case input of
    ' ' -> return Continue
    'q' -> return Cancel
    _ -> getContinue
```

Taking User Input for a Loop

IO actions feel like a procedural language. Sometimes it's tempting to fall back on familiar patterns. We even have access to things like for loops that make it easier to think this way.

Taking User Input for a Loop

IO actions feel like a procedural language. Sometimes it's tempting to fall back on familiar patterns. We even have access to things like for loops that make it easier to think this way.

```
showPages :: [String] -> IO ()
showPages allPages =
  for_ allPages $ \page -> do
    putStr "\^[[1J\^[[1;1H"
    putStr page
    cont <- getContinue
    -- ...
```

Taking User Input for a Loop

IO actions feel like a procedural language. Sometimes it's tempting to fall back on familiar patterns. We even have access to things like for loops that make it easier to think this way.

```
showPages :: [String] -> IO ()
showPages allPages =
  for_ allPages $ \page -> do
    putStr "\^[[1J\^[[1;1H"
    putStr page
    cont <- getContinue
    -- ...
```

Unfortunately, this can make things more difficult instead of easier.

Recursive IO Actions

You can use recursion in IO actions just like you would for pure functions.

Recursive IO Actions

You can use recursion in IO actions just like you would for pure functions.

```
showPages :: [String] -> IO ()
showPages [] = pure ()
showPages (page:pages) = do
  putStr "\^[[1J\^[[1;1H"
  putStr page
  cont <- if null pages
          then pure Cancel
          else getContinue
  when (Continue == cont) $
    showPages pages
```

Recursive IO Actions

You can use recursion in IO actions just like you would for pure functions.

```
showPages :: [String] -> IO ()
showPages [] = pure ()
showPages (page:pages) = do
  putStr "\^[[1J\^[[1;1H"
  putStr page
  cont <- if null pages
          then pure Cancel
          else getContinue
  when (Continue == cont) $
    showPages pages
```

This is a good starting spot for implementing the effectful logic in your programs.

Continued Action

As your programs grow, it's a good idea to think about making your IO actions compose. This can make your code a bit more verbose at first, but it buys you flexibility later.

Continued Action

`onContinue` lets us to do any IO action when the user continues:

Continued Action

`onContinue` lets us to do any IO action when the user continues:

```
onContinue :: IO () -> IO ()
onContinue ioAction = do
  cont <- getContinue
  case cont of
    Cancel -> pure ()
    Continue -> ioAction
```

Continued Action

`onContinue` lets us to do any IO action when the user continues:

```
onContinue :: IO () -> IO ()
onContinue ioAction = do
  cont <- getContinue
  case cont of
    Cancel -> pure ()
    Continue -> ioAction
```

`forPages` separates looping application logic with a continuation:

Continued Action

`onContinue` lets us to do any IO action when the user continues:

```haskell
onContinue :: IO () -> IO ()
onContinue ioAction = do
  cont <- getContinue
  case cont of
    Cancel -> pure ()
    Continue -> ioAction
```

`forPages` separates looping application logic with a continuation:

```haskell
forPages :: (String -> IO ()) -> [String]  -> IO ()
forPages ioAction pages  =
  case pages of
    [] -> pure ()
    (page:rest) -> do
      ioAction page
      onContinue (forPages ioAction rest)
```

Continued Action

`onContinue` lets us to do any IO action when the user continues:

```
onContinue :: IO () -> IO ()
onContinue ioAction = do
  cont <- getContinue
  case cont of
    Cancel -> pure ()
    Continue -> ioAction
```

`forPages` separates looping application logic with a continuation:

```
forPages :: (String -> IO ()) -> [String]  -> IO ()
forPages ioAction pages  =
  case pages of
    [] -> pure ()
    (page:rest) -> do
      ioAction page
      onContinue (forPages ioAction rest)
```

`showPages` composes benefits from the work we've done

Continued Action

`onContinue` lets us to do any IO action when the user continues:

```
onContinue :: IO () -> IO ()
onContinue ioAction = do
  cont <- getContinue
  case cont of
    Cancel -> pure ()
    Continue -> ioAction
```

`forPages` separates looping application logic with a continuation:

```
forPages :: (String -> IO ()) -> [String]  -> IO ()
forPages ioAction pages  =
  case pages of
    [] -> pure ()
    (page:rest) -> do
      ioAction page
      onContinue (forPages ioAction rest)
```

`showPages` composes benefits from the work we've done

```
showPages :: [String] -> IO ()
showPages = forPages $ \page -> do
  putStr "\^[[1J\^[[1;1H"
  putStr page
```

Putting It All Together

```haskell
runHCat :: IO ()
runHCat = do
  contents <- readFile =<< targetFileName
  termSize <- getTerminalSize
  showPages $ paginate termSize contents
```

Questions?

Questions?

Want to know more?

Follow the QR Code for a chance to win a copy of Effective Haskell.