Haskell: Mistakes I've made

HaskellerZ

Jasper Van der Jeugt

March 31, 2016

About me

- I really like Haskell
- Contributed to and authored open source projects
- Did some consulting for various companies
- Worked at Better, currently at `fugue.co`

What this talk is about

Beginner Haskell "Mistakes" that can be avoided, signs of Haskell code smell.

What this talk is not about

We only talk about code that is accepted by the compiler.

```
fib :: Int -> Int
fib n =
  | n <= 1     = 1
  | otherwise =
      fib (n - 1) +
      fib (n - 2)
```

Intro: pick your battles

Pick your battles

It's often impossible to have perfectly clean code *everywhere* in large projects.

Pick your battles

- Aesthetically pleasing code?
- Just need to get the job done?
- Sacrifice clean code on the performance altar?
- Document top-level functions?

Pick your battles

What works for me: focus on *modules* as units.

Pick your battles: modules

The *exposed interface* of a module should be clean.

But inside you can `unsafePerformIO` *all you want.*

Pick your battles: modules

The module should have some documentation about what it is for and how one should use it.

But you don't necessarily need to write Haddock for all functions if it's clear what they do from the name.

Pick your battles: modules

There should be some tests that verify that the module works correctly.

But you don't need to test all the internals.

Mistake: not using `-Wall`

Mistake: not using `-Wall`

Always compile using `-Wall` .
Use `-Werror` for production/tests.

Low-risk `-Wall`

```
-fwarn-unused-imports,
-fwarn-dodgy-exports,
-fwarn-dodgy-imports,
-fwarn-monomorphism-restriction,
-fwarn-implicit-prelude,
-fwarn-missing-local-sigs,
-fwarn-missing-exported-sigs,
-fwarn-missing-import-lists,
-fwarn-identities
```

Medium-risk `-Wall`

`-fwarn-unused-binds`,
`-fwarn-unused-matches`,
`-fwarn-auto-orphans`

High-risk `-Wall`

```
-fwarn-incomplete-patterns
-fwarn-incomplete-uni-patterns
-fwarn-incomplete-record-updates
```

Mistake: wildcard pattern matches

Mistake: wildcard pattern matches

```
data Murderer
  = Innocent
  | Murderer

canBeReleased
  :: IsMurderer -> Bool
canBeReleased = \case
  Innocent -> True
  Murderer -> False
```

Mistake: wildcard pattern matches

```
fireSquad :: IsMurderer -> Bool
fireSquad = \case
  Innocent -> False
  _        -> True
```

Mistake: wildcard pattern matches

```
data IsMurderer
  = Innocent
  | Murderer
  | Like99PercentSure
```

Mistake: wildcard pattern matches

Note: this also appears
in other forms!

```
fireSquad :: IsMurderer -> Bool
fireSquad x = x /= Innocent
```

Mistake: wildcard pattern matches

It is often best to explicitly match on every constructor, unless you can be sure no further constructors will be added (`Maybe` , `Either` ...).

Mistake: you think you understand exceptions

Mistake: exceptions

Most common mistake: you think you understand how exceptions work in Haskell.

Alternatively: you falsely assume you will always pay attention to how they work.

Laziness + exceptions = hard

```
errOrOk <- try $
  return [1, 2, error "kaputt"]
case errOrOk of
  Left err -> print err
  Right x  -> print x
```

Laziness + exceptions = hard

```
errOrOk <- try $
  return (throw MyException)
case errOrOk of
  Left err -> print err
  Right x  -> print x
```

Async exceptions: very hard

Does this function close the socket?

```
foo :: (Socket -> IO a) -> IO a
foo f = do
  s <- openSocket
  r <- try $ f s
  closeSocket s
  ...
```

Async exceptions: very hard

How about:

```
foo :: (Socket -> IO a) -> IO a
foo f = mask $ \restore -> do
  s <- openSocket
  r <- try $ restore $ f s
  closeSocket s
  ...
```

Debugging async exceptions

1. Easy, this function shouldn't throw exceptions.
2. The exception comes from another place? Let's use `mask`?
3. Maybe we can set `unsafeUnmask` to allow an interrupt?
4. ...
5. Nothing works and programming is an eternal cycle of pain and darkness.
6. Go back to 1

Exceptions: avoiding the madness

Keep your code pure where possible.

Exceptions: avoiding the madness

Try to avoid throwing exceptions from pure code. If you have an exception lurking somewhere deep within a thunk, bad things will usually happen.

Exceptions: avoiding the madness

Use more predictable monad stacks:

- `IO (Either e a)`
- `Control.Monad.Except`

Exceptions: avoiding the madness

Use existing solutions such as `resource-pool`, `resourcet`, and existing patterns like `bracket`.

Mistake: avoiding GHC extensions

Mistake: avoiding GHC extensions

Before I wrote Haskell: C/C++ background. Lots of pain trying to get things compiling on MSVC++.

Mistake: avoiding GHC extensions

1. There is only GHC
2. Extensions widely used in "standard" packages
3. It's Haskell so refactoring is easy

Mistake: avoiding GHC extensions

Average of around 3 extensions per module. Commonly:

`OverloadedStrings`,
`TemplateHaskell`,
`DeriveDataTypeable`,
`ScopedTypeVariables`,
`GeneralizedNewtypeDeriving`,
`BangPatterns`

Mistake: avoiding GHC extensions

Extensions that improve readability are always a good idea!

`LambdaCase`, `ViewPatterns`, `PatternGuards`, `MultiWayIf`, `TupleSections`, `BinaryLiterals`

Mistake: avoiding GHC extensions

Extensions that write code for you are even better!

```
DeriveFunctor ,
DeriveFoldable ,
DeriveTraversable ,
RecordWildCards ,
DeriveGeneric
```

Mistake: avoiding GHC extensions

Further reading:

Oliver Charles:
24 days of GHC extensions

Mistake: not testing enough

Mistake: not testing enough

If it compiles, ship it.

Mistake: not testing enough

QuickCheck/SmallCheck

```
prop_revRev xs =
  reverse (reverse xs) == xs
```

Mistake: not testing enough

Lots of tutorials available on QuickCheck/SmallCheck, but many people only use this to test a *pure core* of their application.

Mistake: not testing enough

Test your entire application!

- `hspec-snap`
- `hspec-webdrivers`
- `tasty-golden`
- `...`

Mistake: not testing enough

Test your entire application!

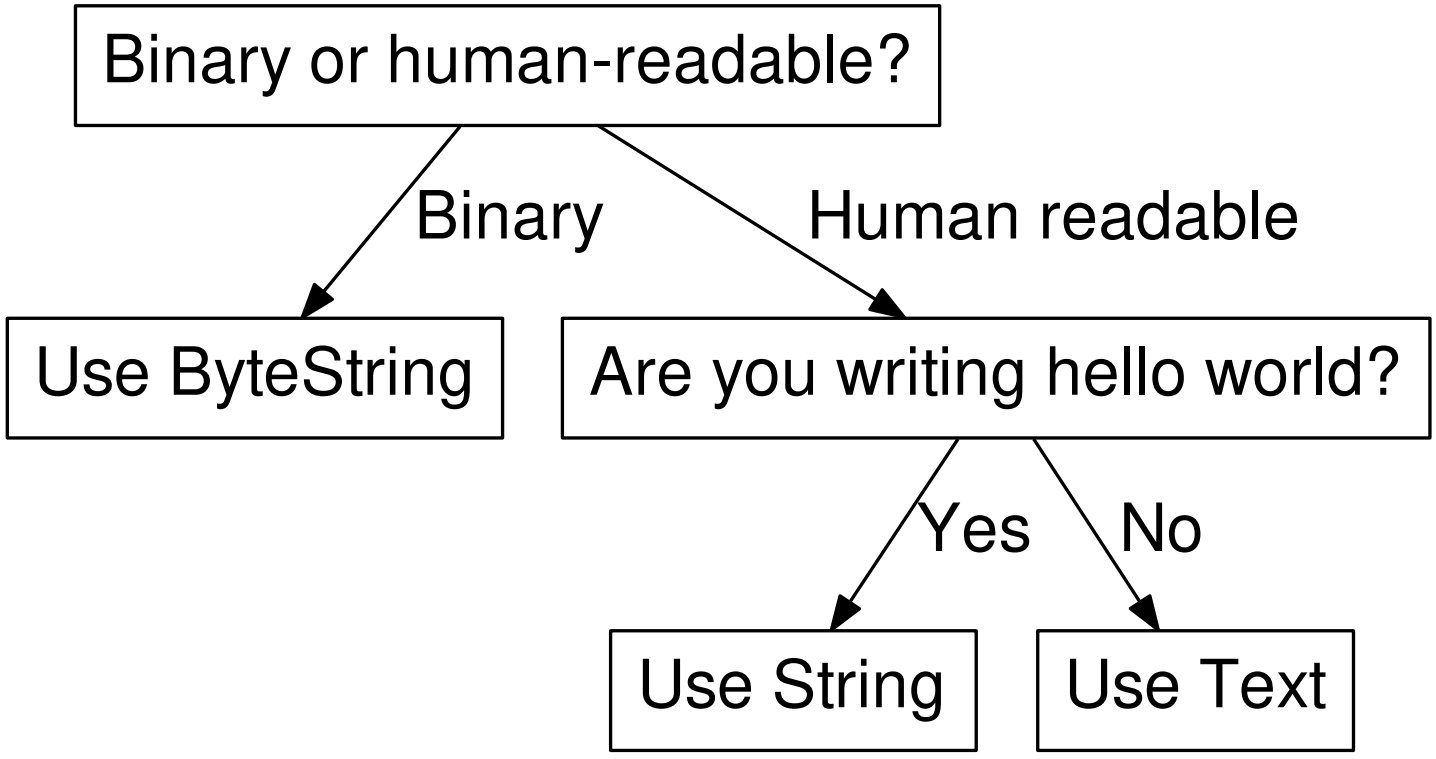
- Roll your own framework
- Bash
- Python
- JavaScript
- ...

Mistake: string types

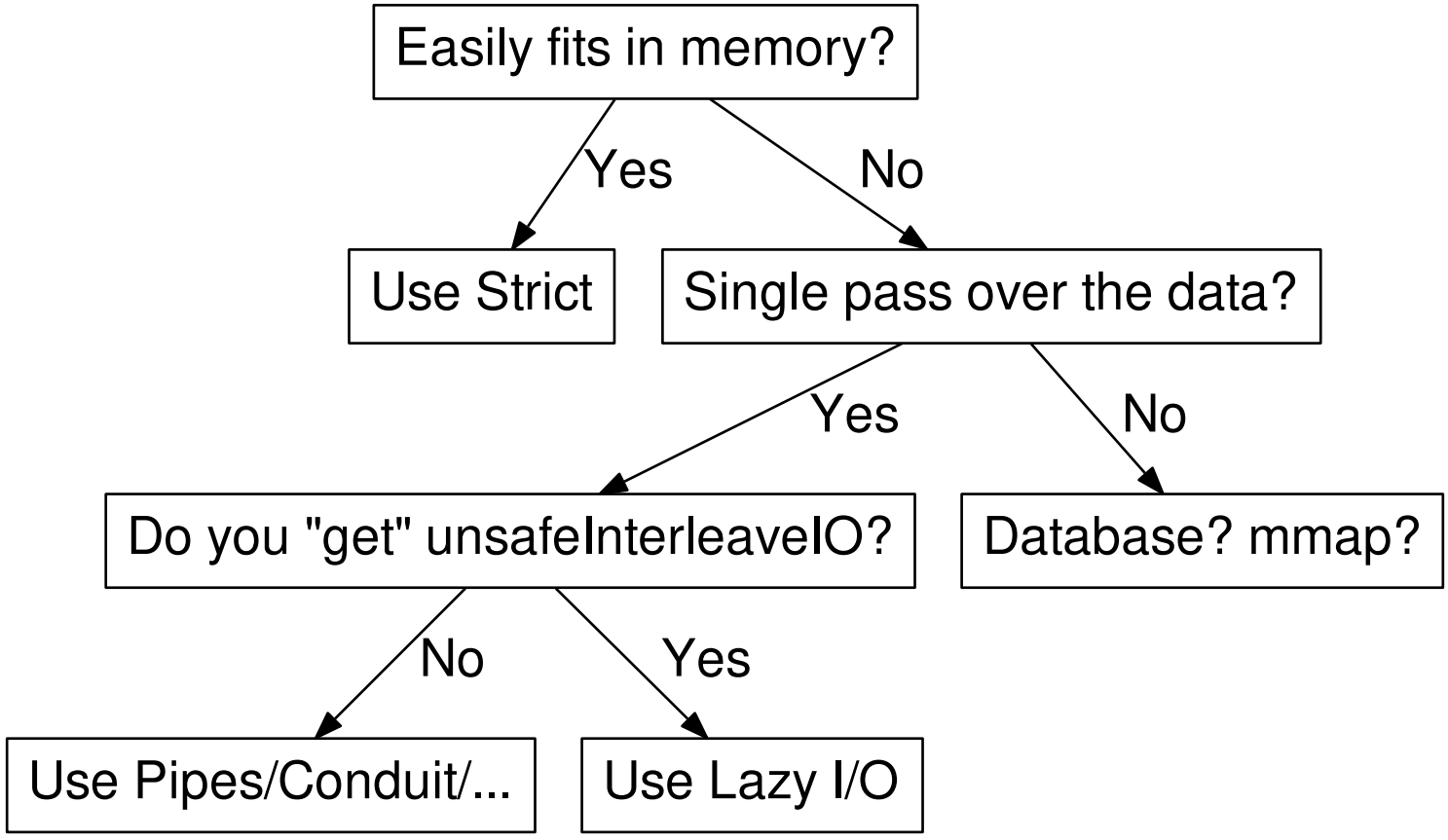
Mistake: string types

1. Write some code
2. It's pretty slow
3. Find a stackoverflow thread from 2008 that tells me to use `ByteString`
4. Z[?]rich

Mistake: string types



Mistake: string types



Mistake: not enough pure code

Mistake: not enough pure code

```
main :: IO ()
main = do
  x <- readLn :: IO Int
  y <- readLn :: IO Int
  -- wizzling
  let wizzled =
        x + y * y * 3 + 9
  print wizzled
```

Pure code: sometimes it's easy

```
wizzle :: Int -> Int -> Int
wizzle x y =
  x + y * y * 3 + 9
```

Pure code: sometimes it's easy

```
main :: IO ()
main = do
  x <- readLn :: IO Int
  y <- readLn :: IO Int
  print $ wizzled x y
```

Pure code: sometimes it's easy

1. Read your input
2. Do your stuff
3. Write your output

Pure code: sometimes it's hard

- Can't read all input because of memory constraints
- Input reading depends on temporary results
- Need for threads, continuous output
- ...

Pure code: example

```
data Request = Request
  { rqPath :: String
  , ...
  }

data Response
  = Response200 String
  | Response404
```

Pure code: example

```
handler
  :: Request -> IO Response
handler = do
  rq <- readRequest
  case rqPath rq of
    "/index" ->
      return $ Response200 "ok"
    _        ->
      return Response404
```

Pure code: example

Startup idea: People can't be expected to reverse their own strings if they are on a low-powered device like an IoT-enabled toaster. We should have a cloud service for reversing a string.

Pure code: example

```
data Request = Request
  { rqPath :: String
  , rqBody :: String
  , ...
  }

data Response
  = Response200 String
  | Response404
```

Pure code: example

```
handler = do
  rq <- readRequest
  case rqPath rq of
    "/rev"   ->
      return $ Response200 $
        reverse (rqBody rq)
    ...
```

Pure code: example

```
data Request = Request
  { rqPath :: String
  , ...
  }

data Response
  = Response200 String
  | Response404
  | ReadBody
      (String -> Response)
```

Pure code: example

```
handler = do
  rq <- readRequest
  case rqPath rq of
    "/rev"   -> return $
      ReadBody $ \body ->
        Response200 $
        reverse body
```

Pure code: example

```
run (Response200 str) =
  sendResponse str
run Response404 =
  error "404"
run (ReadBody f) = do
  body <- readRequestBody
  run (f body)
```

Pure code: free

```
data Free (f :: * -> *) a
  = Free (f (Free f a))
  | Pure a
```

Pure code: free

```
data Response
  = Response200 String
  | Response404

data HandlerF a
  = ReadBodyF (String -> a)
  deriving (Functor)

type Handler =
    Free HandlerF
```

Pure code: free

```
getBody :: Handler String
getBody = Free $
    ReadBodyF return

ok200
  :: String
  -> Handler Response
ok200 str = Pure $
    Response200 str
```

Pure code: free

```
handler
  :: Request
  -> Handler Response
handler rq =
  case rqPath rq of
    "/rev" -> do
      str <- getBody
      ok200 $ reverse str
```

Pure code: free

```
run :: Handler Response -> IO ()
run (Pure (Response200 str)) =
  sendResponse str
run (Pure (Response404)) =
  error "404"
run (Free (ReadBodyF f)) = do
  body <- readRequestBody
  run (f body)
```

Mistake: overdoing abstraction

Mistake: overdoing abstraction

```
zygoHistoPrepro
  :: (Unfoldable t, Foldable t)
  => (Base t b -> b)
  -> (forall c. Base t c -> Base t c)
  -> (Base t (EnvT b (Stream (Base t))
        -> a)
  -> t -> a
zygoHistoPrepro f g t =
  -- unless you want a generalized
  -- zygomorphism.
  gprepro (distZygoT f distHisto) g t
```

Mistake: overdoing abstraction

```
getProp k =
  case getProperty k m of
    Left  err -> fail err
    Right val ->
      return (show val)
```

Mistake: overdoing abstraction

Or do you prefer this version?

```
getProp =
  either fail (return . show) .
  flip getProperty
```

Mistake: Premature abstraction

Mistake: premature abstractions

Example: the resource provider in *Hakyll*.

An abstaction around the file system to list and obtain resources as various types. It provides some caching and access to metadata.

Mistake: premature abstractions

- Zero or one instances
- Lots of code working around restrictions
- Making assumptions about specific behaviours
- Too general
- Too abstract

Mistake: premature abstractions

(Historically incorrect) example:

```
class Functor m => Monad m where
  return :: a -> m a
  join   :: m (m a) -> m a

  -- wat
  fail   :: String -> m a
```

Mistake: premature abstractions

Abstractions are very hard to get right.

In general, just writing and repeating the non-abstract code until you *really* see the pattern helps.

Questions?