HIGHER KINDED DATA

Chris Penner

(he/him)

CHRIS PENNER

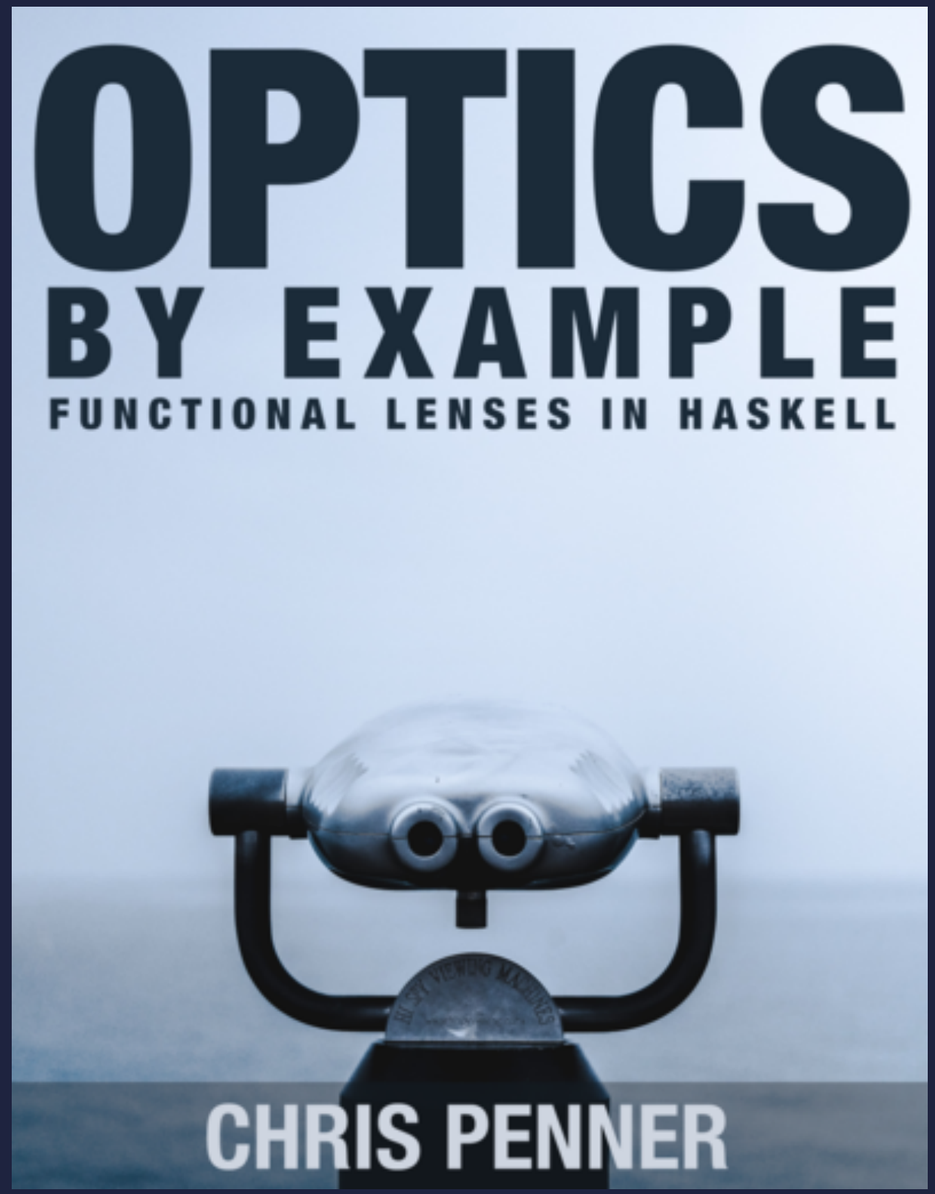
I write golang to pay the bills

I like doing handstands

I wrote "Optics By Example"

chrispenner.ca

@ChrisLPenner



LET'S START WITH THE BASICS

Here's a datatype representing a Github Repository.

```
data Repository = Repository
  { name  :: String
  , owner :: Email
  , stars :: Int
  }
```

THIS IS A PRETTY STANDARD FORM FOR THIS DATA

BUT DATA ISN'T ALWAYS

perfect....

FIELDS MIGHT BE

missing

FIELDS MIGHT BE

invalid

DATA TAKES ON A *journey*

HOW DO WE REPRESENT THESE

DATA VARIANTS

IN A *well-typed* WAY?

ONE RECORD PER USE-CASE

```
data Repository = Repository
  { name  :: String
  , owner :: Email
  , stars :: Int
  }
```

```
data PartialRepository = PartialRepository
  { name  :: Maybe String
  , owner :: Maybe Email
  , stars :: Maybe Int
  }

data ValidatedRepository = ValidatedRepository
  { name  :: Either Error String
  , owner :: Either Error Email
  , stars :: Either Error Int
  }
```

ONE RECORD PER USE-CASE

- Lots of boilerplate
- Need to duplicate all instances & utilities
- Need to keep all records in sync
- No structured relationship between fields

The Goal

- Less boilerplate & duplication
- A single shared structure
- Structured relationship between fields of related types

HOW DO WE GET THERE?

LET'S LOOK AT

polymorphic

RECORDS

POLYMORPHIC RECORDS

```
data Stock a = Stock
  { item :: a
  , units :: Int
  }
```

a is a type parameter
it could represent any type!

POLYMORPHIC RECORDS

```haskell
data Stock a = Stock
  { item :: a
  , units :: Int
  }
```

```haskell
newtype DVD = DVD String

terminatorStock :: Stock DVD
terminatorStock =
  Stock
  { item = DVD "The Terminator"
  , units = 12
  }
```

SAME RECORD, DIFFERENT ITEMS

```haskell
newtype DVD = DVD String

terminatorStock :: Stock DVD
terminatorStock =
  Stock
  { item = DVD "The Terminator"
  , units = 12
  }
```

```haskell
newtype Candy = Candy String

milkaStock :: Stock Candy
milkaStock =
  Stock
  { item = Candy "Milka Bar"
  , units = 36
  }
```

Is there a *pattern* here?

```
data Repository = Repository
  { name  :: String
  , owner :: Email
  , stars :: Int
  }
```

```
data PartialRepository = PartialRepository
  { name  :: Maybe String
  , owner :: Maybe Email
  , stars :: Maybe Int
  }

data ValidatedRepository = ValidatedRepository
  { name  :: Either Error String
  , owner :: Either Error Email
  , stars :: Either Error Int
  }
```

IT'S THE SAME RECORD WITH DIFFERENT FIELD WRAPPERS

CAN WE BE POLYMORPHIC OVER THE *wrapper?*

```haskell
data RepositoryHKD f = RepositoryHKD
  { name  :: f String
  , owner :: f Email
  , stars :: f Int
  }
```

f could be any wrapper type

HIGHER KINDED DATATYPE

```
>>> :kind RepositoryHKD
RepositoryHKD :: (Type → Type) → Type
                 \_______________/
                         |
                         f
```

HIGHER KINDED DATATYPE

An HKD is a type with kind

```
(Type → Type) → Type
```

Which wraps its fields in a polymorphic container

Polymorphic

```
data RepositoryHKD f = RepositoryHKD
  { name  :: f String
  , owner :: f Double
  , stars :: f Int
  }
```

f ~ Maybe

```
data RepositoryHKD Maybe = RepositoryHKD
  { name  :: Maybe String
  , owner :: Maybe Double
  , stars :: Maybe Int
  }
```

STANDARD RECORD SYNTAX

```
partialRepo :: RepositoryHKD Maybe
partialRepo = RepositoryHKD
  { name  = Just "HKD Conference Talk"
  , owner = Nothing
  , stars = Just 14
  }
```

WHAT ABOUT AN UNWRAPPED REPOSITORY?

unwrapped record

```
data Repository = Repository
  { name  :: String
  , owner :: Email
  , stars :: Int
  }
```

f ~ ???

```
data RepositoryHKD f = RepositoryHKD
  { name  :: f String
  , owner :: f Double
  , stars :: f Int
  }
```

WE NEED A

WRAPPER

WHICH DOESN'T DO ANYTHING

```
newtype Identity a = Identity a
```

```haskell
repo :: RepositoryHKD Identity
repo = RepositoryHKD
  { name  = Identity "HKD Conference Talk"
  , owner = Identity (Email "chris@chrispenner.ca")
  , stars = Identity 12
  }
```

TIME FOR AN

EXAMPLE

PATCH / REPO

PATCH /REPO ALLOWS UPDATING ANY FIELD OF A *repo*

If we were stuck with:

```
data Repository = Repository
  { name  :: String
  , owner :: Email
  , stars :: Int
  }
```

We can't update only a subset of our fields!

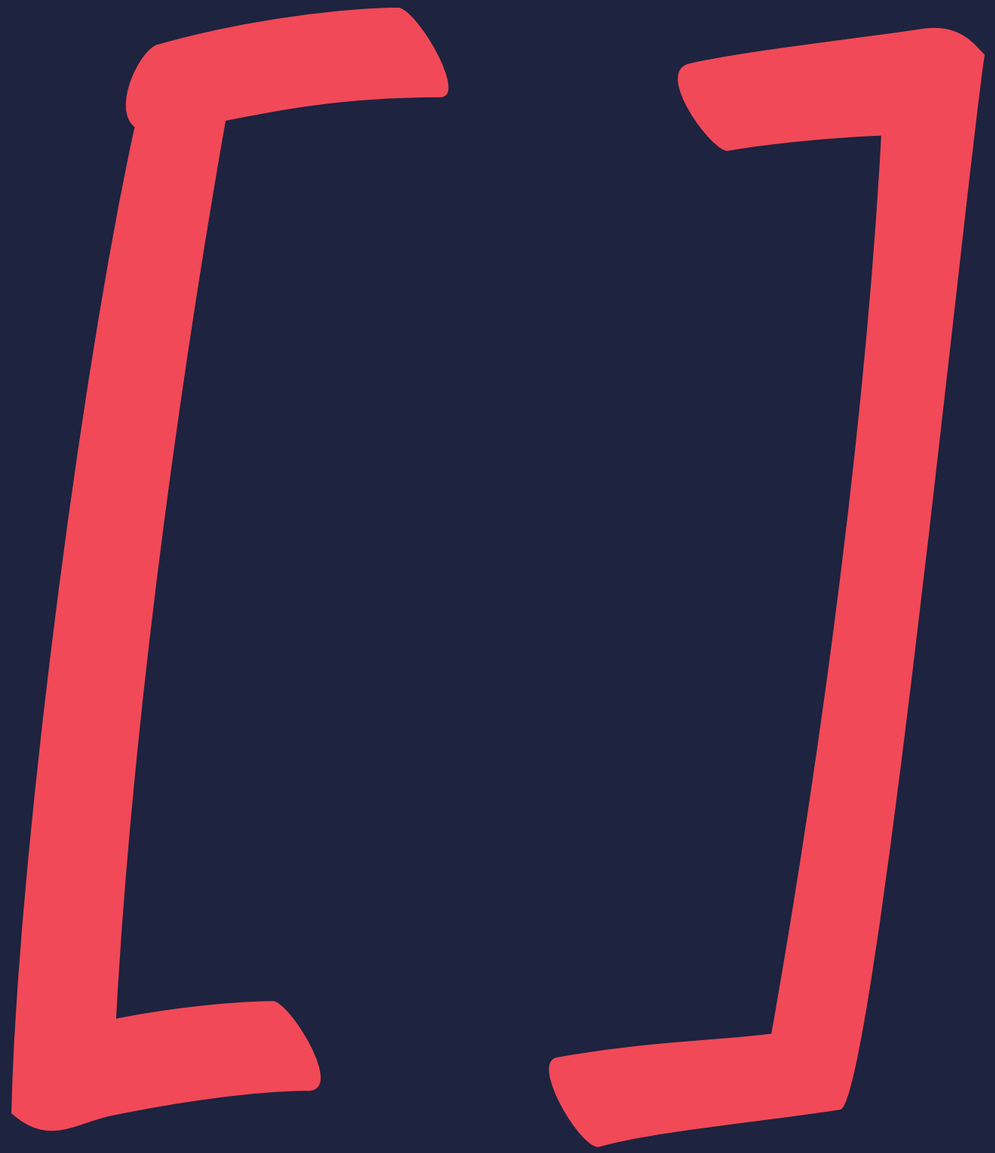
Partial Records to the rescue!

```
PATCH ChrisPenner/hkds

{
   "name": "Reducing Boilerplate with HKDs"
}

----------------------------------------

repoUpdates :: RepositoryHKD Maybe
repoUpdates = RepositoryHKD
  { name  = Just "Reducing Boilerplate with HKDs"
  , owner = Nothing
  , stars = Nothing
  }
```

Maybe

REPRESENTS PARTIAL RECORDS

WHAT DO *other* WRAPPERS DO?



A RECORD WHERE EACH FIELD HAS *zero or more* VALUES

```haskell
data WebServerConfigHKD f =
  WebServerConfigHKD
  { preferredPort   :: f Int
  , preferencesYaml :: f FilePath
  , databasePath    :: f String
  }
```

[ ] AS A WRAPPER

```
config :: WebServerConfigHKD []
config = WebServerConfigHKD
  { preferredPort = [8080, 4200]
  , preferencesYaml = ["./prefs.yaml", "~/app/.preferences"]
  , databasePath = ["app.com/postgres:5432"]
  }
```

IO

ALLOWS DELAYING WORK

UNTIL A SPECIFIC FIELD IS NEEDED

```haskell
configLoader :: WebServerConfigHKD IO
configLoader = WebServerConfigHKD
  { preferredPort = read <$> getEnv "PORT"
  , preferencesYaml = fmap (!! 1) <$> getArgs
  , databasePath = fetchDBConfig
  }
```

Const String

FOR PER-FIELD DOCUMENTATION

```haskell
newtype Const a b = Const a

documentation :: forall field. Const String field
documentation = Const "Some text"
```

```haskell
configDocs :: WebServerConfigHKD (Const String)
configDocs = WebServerConfigHKD
  { preferredPort   = Const "The port where your app will be served."
  , preferencesYaml = Const "Where you store your preferences."
  , databasePath    = Const "The URL for your postgres database."
  }
```

```haskell
{-# LANGUAGE OverloadedStrings #-}
configDocs :: WebServerConfigHKD (Const String)
configDocs = WebServerConfigHKD
  { preferredPort   = "The port where your app will be served."
  , preferencesYaml = "Where you store your preferences."
  , databasePath    = "The URL for your postgres database."
  }
```

(,) String

FOR ANNOTATING REAL DATA

YOU CAN ALSO DEFINE YOUR OWN WRAPPERS

SO HOW DO WE USE THESE RECORDS?

```
defaultConfig :: WebServerConfigHKD Identity
                      +
userOverrides :: WebServerConfigHKD Maybe
                      =
actualConfig  :: WebServerConfigHKD Identity
```

SINGLE FIELD

```
applyFieldOverride :: forall field.
                      Identity field
                   -> Maybe    field
                   -> Identity field
applyFieldOverride (Identity def) maybeOverride =
  case maybeOverride of
    Just override -> Identity override
    Nothing       -> Identity def
```

```
applyFieldOverride
  :: forall a.     Identity a
                -> Maybe a
                -> Identity a

applyConfigOverride
  :: WebServerConfigHKD Identity
  -> WebServerConfigHKD Maybe
  -> WebServerConfigHKD Identity
```

```haskell
applyConfigOverride
  :: WebServerConfigHKD Identity
  -> WebServerConfigHKD Maybe
  -> WebServerConfigHKD Identity
applyConfigOverride defaultConfig overrideConfig =
  WebServerConfigHKD
  { preferredPort   =
      applyFieldOverride (preferredPort defaultConfig)   (preferredPort overrideConfig)
  , preferencesYaml =
      applyFieldOverride (preferencesYaml defaultConfig) (preferencesYaml overrideConfig)
  , databasePath    =
      applyFieldOverride (databasePath defaultConfig)    (databasePath overrideConfig)
  }
```

BOILERPLATE ALERT

ZIPPING RECORD FIELDS

- Know each field has the same underlying type
  - Wrapper types are known
  - Just need a way to line them up!

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

WE WANT TO

ZIP

THE FIELDS

```
hkdZipWith :: forall (hkd :: (Type → Type) → Type)
            . (forall field. f field -> g field -> h field)
            -> hkd f
            -> hkd g
            -> hkd h
```

```
zipWith :: (a -> b -> c)
        -> [a]
        -> [b]
        -> [c]

hkdZipWith :: forall (hkd :: (Type → Type) → Type)
           . (forall field. f field -> g field -> h field)
           -> hkd f
           -> hkd g
           -> hkd h
```

SPECIALIZE THE CONTAINERS

```
hkdZipWith :: forall (hkd :: (Type → Type) → Type)
            . (forall field. Identity field -> Maybe field -> Identity field)
            -> hkd Identity
            -> hkd Maybe
            -> hkd Identity
```

SPECIALIZE THE HKD

```
hkdZipWith :: (forall field. Identity field -> Maybe field -> Identity field)
           -> WebServerConfigHKD Identity
           -> WebServerConfigHKD Maybe
           -> WebServerConfigHKD Identity
```

```haskell
hkdZipWith :: (forall field. Identity field -> Maybe field -> Identity field)
           -> WebServerConfigHKD Identity
           -> WebServerConfigHKD Maybe
           -> WebServerConfigHKD Identity

applyFieldOverride
  :: forall field. Identity field -> Maybe field -> Identity field

applyConfigOverride
  :: WebServerConfigHKD Identity
  -> WebServerConfigHKD Maybe
  -> WebServerConfigHKD Identity
applyConfigOverride =
  hkdZipWith applyFieldOverride
```

```haskell
applyConfigOverride defaultConfig overrideConfig =
  WebServerConfigHKD
  { preferredPort   =
      applyFieldOverride (preferredPort defaultConfig)   (preferredPort overrideConfig)
  , preferencesYaml =
      applyFieldOverride (preferencesYaml defaultConfig) (preferencesYaml overrideConfig)
  , databasePath    =
      applyFieldOverride (databasePath defaultConfig)    (databasePath overrideConfig)
  }

-------------------------------------------------------------------------------

applyConfigOverride =
  hkdZipWith applyFieldOverride
```

IT

AUTOMAGICALLY

HANDLES ANY NEW FIELDS

How does `hkdZipWith` work?

INTRODUCING

Barbies

BY DANIEL GORIN

https://hackage.haskell.org/package/barbies

“Types that are parametric on a functor are like Barbies that have an outfit for each role.

This package provides the basic abstractions to work with them comfortably.”

Barbies

Utilities for building, transforming, and combining HKDs

TYPECLASSES

STANDARD POLYMORPHIC TYPES

```
[] :: TYPE → TYPE
```

STANDARD TYPECLASSES

Functor

Applicative

Traversable

HIGHER KINDED POLYMORPHIC TYPES

```
WebServerConfigHKD :: (Type → Type) → Type
```

HIGHER KINDED TYPECLASSES

FunctorB

ApplicativeB

TraversableB

UTILITIES FROM BARBIES

```
bmap :: FunctorB hkd =>
        (forall a. f a → g a)
        → hkd f
        → hkd g
```

```
bmap :: (forall a. Either Error a → Maybe a)
        → WebServerConfigHKD (Either Error)
        → WebServerConfigHKD Maybe
```

```
bzipWith :: ApplicativeB hkd =>
            (forall a. f a → g a → h a)
            → hkd f
            → hkd g
            → hkd h
```

```haskell
bsequence' :: (Applicative e, TraversableB hkd)
           ⇒ hkd e
           → e (hkd Identity)
```

```haskell
sequenceWebServerConfig
  :: WebServerConfigHKD IO
  -> IO (WebServerConfigHKD Identity)
sequenceWebServerConfig = bsequence'
```

```
bFoldMap :: (TraversableB hkd, Monoid m)
         ⇒ (forall a. f a -> m)
         → hkd f
         → m
```

```haskell
collectDocumentation
  :: (forall a. Const String a -> Docs)
  → WebServerConfigHKD (Const String)
  → Docs
collectDocumentation = bFoldMap
```

Deriving Barbie Classes

```haskell
data WebServerConfigHKD f =
  WebServerConfigHKD
  { preferredPort   :: f Int
  , preferencesYaml :: f FilePath
  , databasePath    :: f String
  } deriving stock Generic
    deriving anyclass (FunctorB, ApplicativeB, TraversableB, ConstraintsB)
```

PRACTICAL USES

- Partial Serialization/Deserialization
- Database DSLs and models
- Web-form libraries

Scala

Typescript

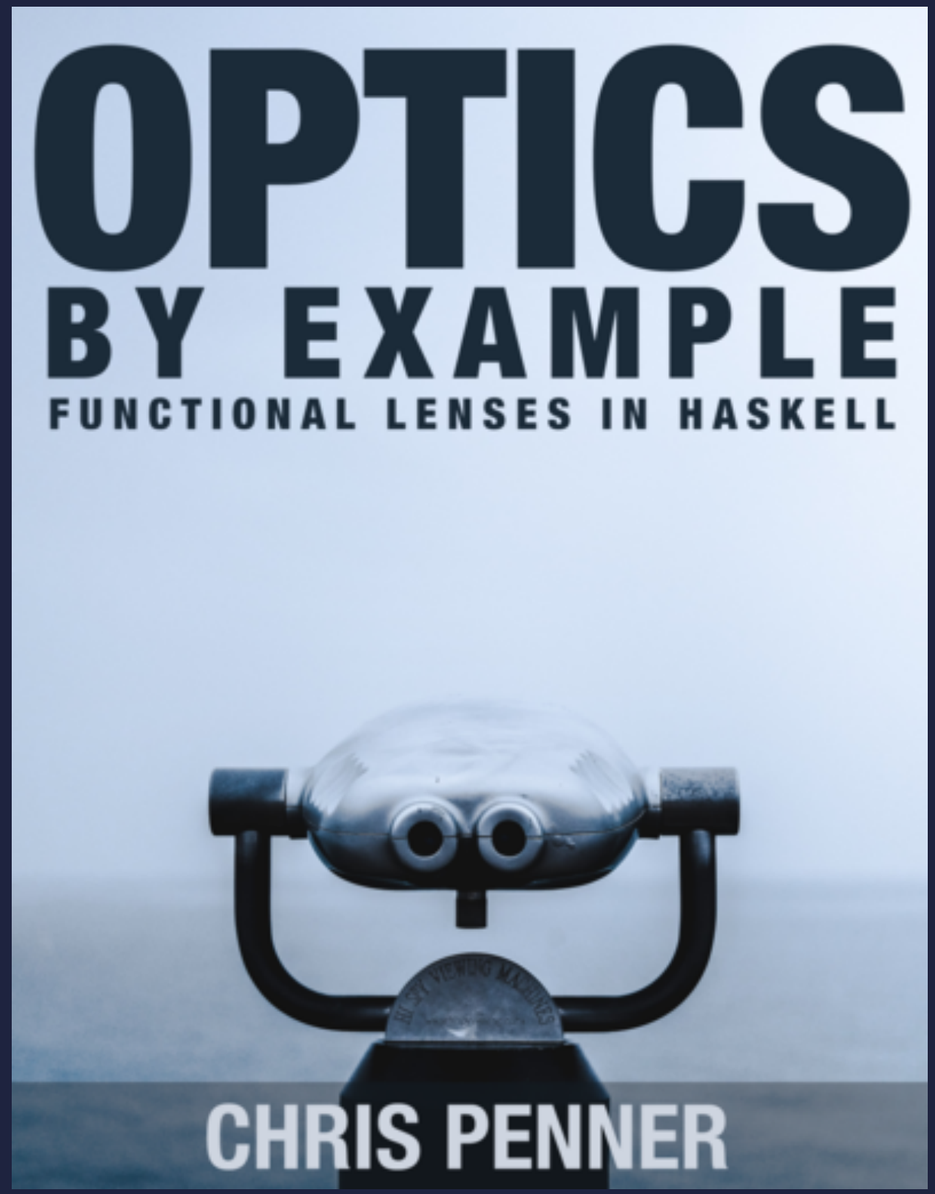
CHRIS PENNER

chrispenner.ca

github.com/ChrisPenner

@ChrisLPenner

leanpub.com/optics-by-example



Questions?