
will not use fancy types
will not use fancy types
will not use fancy types
will not use fancy types
will not use fancy ty
will not use fancy ty
@_alpacaaa #haskellX

# STICK TO SIMPLE HASKELL

@_alpacaaa #haskellX


@_alpacaaa #haskellX

# HASKELL

@_alpacaaa #haskellX

# 1ST APRIL 1990

# 2ND APRIL 1990

# 1998

# HASKELL98

## LAZY
## DATA TYPES
## TYPE CLASSES

# GOALS

TEACHING
RESEARCH
APPLICATIONS

# INNOVATION

# TYPES

# EXTENSIONS

```
{-# LANGUAGE DataKinds                #-}
{-# LANGUAGE FlexibleInstances        #-}
{-# LANGUAGE FlexibleContexts         #-}
{-# LANGUAGE KindSignatures           #-}
{-# LANGUAGE GADTs                    #-}
{-# LANGUAGE ScopedTypeVariables      #-}
{-# LANGUAGE StandaloneDeriving       #-}
{-# LANGUAGE TypeFamilies             #-}
{-# LANGUAGE TypeOperators            #-}
{-# LANGUAGE UndecidableInstances #-}
```

# DEPENDENT TYPES

@_alpacaaa #haskellX

# TYPES | VALUES

# TYPES | VALUES

# FANCY TYPES

# RECAP

## 30 YEARS OLD
## AMBITIOUS GOALS
## TOOL FOR RESEARCH

# APPLICATIONS

# HOW FANCY SHOULD YOU GO?

@_alpacaaa #haskellX

# LOOK GOOD ON PAPER

PUN INTENDED

@_alpacaaa #haskellX

# INCLUSIVITY

# MARGINAL BENEFITS

# SO WE IGNORE THE PAST 20 YEARS?

# ERGONOMICS. MAKE THE LANGUAGE NICER, NOT THE TYPE SYSTEM.

# SIMPLE HASKELL

## HASKELL 98

## − BOILERPLATE (GENERICS)

## + ERGONOMIC EXTENSIONS

@_alpacaaa #haskellX

```
{-# LANGUAGE BlockArguments     #-}
{-# LANGUAGE LambdaCase         #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE TypeApplications   #-}
```

# APPLICATION ARCHITECTURE WITHOUT FANCY TYPES?

@_alpacaaa #haskellX

# FUNCTIONAL CORE
# IMPERATIVE SHELL

# IO IS FINE

```haskell
app :: Env -> IO a
```

```haskell
data Env
  = Env
      { usersCache :: TVar [User]
      , postgresConnection :: PG.Connection
      , log :: Severity -> Text -> IO ()
      , fetchUser :: UserId -> IO (Maybe User)
      , storeFile :: Filename -> ByteString -> IO (Either Text ())
      }
```

```haskell
data Env
  = Env
      { usersCache :: TVar [User]
      , postgresConnection :: PG.Connection
      , log :: Severity -> Text -> IO ()
      , fetchUser :: UserId -> IO (Maybe User)
      , storeFile :: Filename -> ByteString -> IO (Either Text ())
      }
```

```haskell
data Env
  = Env
      { usersCache :: TVar [User]
      , postgresConnection :: PG.Connection
      , log :: Severity -> Text -> IO ()
      , fetchUser :: UserId -> IO (Maybe User)
      , storeFile :: Filename -> ByteString -> IO (Either Text ())
      }
```

```haskell
data Env
  = Env
      { usersCache :: TVar [User]
      , postgresConnection :: PG.Connection
      , log :: Severity -> Text -> IO ()
      , fetchUser :: UserId -> IO (Maybe User)
      , storeFile :: Filename -> ByteString -> IO (Either Text ())
      }
```

```haskell
data UserService
  = UserService
      { fetchUser  :: UserId -> IO (Maybe User)
      , updateUser :: User -> IO (Either UserServiceError ())
      , deleteUser :: UserId -> IO (Either UserServiceError ())
      }

data Env
  = Env
      { userService :: UserService
      , ...
      }
```

```haskell
data UserService
  = UserService
      { fetchUser  :: UserId -> IO (Maybe User)
      , updateUser :: User -> IO (Either UserServiceError ())
      , deleteUser :: UserId -> IO (Either UserServiceError ())
      }

data Env
  = Env
      { userService :: UserService
      , ...
      }
```

```haskell
data UserService
  = UserService
      { fetchUser  :: UserId -> IO (Maybe User)
      , updateUser :: User -> IO (Either UserServiceError ())
      , deleteUser :: UserId -> IO (Either UserServiceError ())
      }

data Env
  = Env
      { userService :: UserService
      , ...
      }
```

```haskell
app :: Env -> IO a
```

# ReaderT Design Pattern

https://www.fpcomplete.com/blog/2017/06/readert-design-pattern

# EXCEPTIONS ARE GOOD

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
data Config = Config { ... }

data LoadConfigError
  = ConfigFileNotFound
  | ConfigInvalidJSON String

loadConfig :: FilePath -> IO (Either LoadConfigError Config)
loadConfig file = do
  exists <- System.Directory.doesFileExist file
  if exists
    then do
      content <- ByteString.Lazy.readFile file
      case Aeson.eitherDecode content of
        Left err -> pure $ Left (ConfigInvalidJSON err)
        Right config -> pure (Right config)
    else
      pure (Left ConfigFileNotFound)
```

```haskell
loadConfig :: FilePath -> IO Config
loadConfig file = do
  content <- ByteString.Lazy.readFile file
  case Aeson.eitherDecode content of
    Left err ->
      Exception.throwString ("Failed to decode config: " <> show err)
    Right config ->
      pure config
```

```haskell
loadConfig :: FilePath -> IO Config
loadConfig file = do
  content <- ByteString.Lazy.readFile file
  case Aeson.eitherDecode content of
    Left err ->
      Exception.throwString ("Failed to decode config: " <> show err)
    Right config ->
      pure config
```

```haskell
loadConfig :: FilePath -> IO Config
loadConfig file = do
  content <- ByteString.Lazy.readFile file
  case Aeson.eitherDecode content of
    Left err ->
      Exception.throwString ("Failed to decode config: " <> show err)
    Right config ->
      pure config
```

```haskell
loadConfig :: FilePath -> IO Config
loadConfig file = do
  content <- ByteString.Lazy.readFile file
  case Aeson.eitherDecode content of
    Left err ->
      Exception.throwString ("Failed to decode config: " <> show err)
    Right config ->
      pure config
```

# SIMPLE HASKELL

@_alpacaaa #haskellX

# SIMPLE / BORING

# Boring technology

## Dan McKinley

# GO & ELM

A TOOL FOR RESEARCH

A LANGUAGE TO WRITE APPLICATIONS

Any problem you have, you could turn into a PhD thesis.

Ryan Thinkle

# WANT TO USE HASKELL IN PRODUCTION?

# YOU DON'T NEED A PHD.

# YOU DON'T NEED EFFECT LIBRARIES.

# YOU DON'T NEED FANCY TYPES.

Dependent Haskell is the future

https://serokell.io/blog/why-dependent-haskell

# STICK TO SIMPLE HASKELL

@_alpacaaa #haskellX

# Stick to Simple Haskell

Marco Sampellegrini
Habito

@_alpacaaa