Functional programming in practice

The best ideas for the best practices

Ondra Pelech

Introduction to ideas

Languages converging to old and very influential language called ML

- Functions first class citizens
- Typed (parametric polymorphism aka generics)
- Expression-based
- ADTs + pattern matching

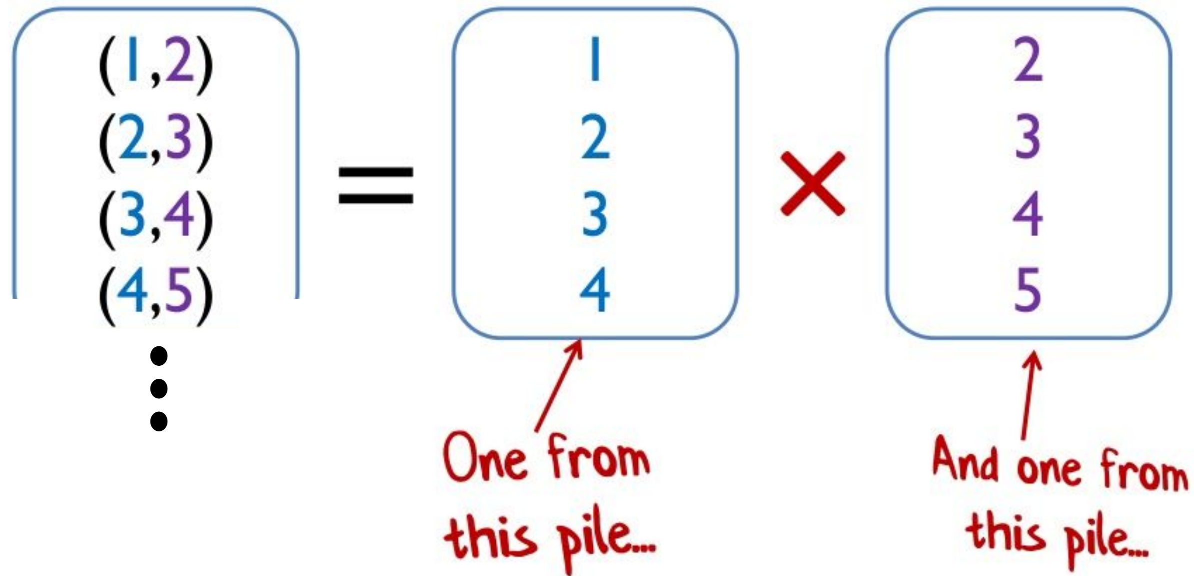
Not just Haskell

Practical for other languages

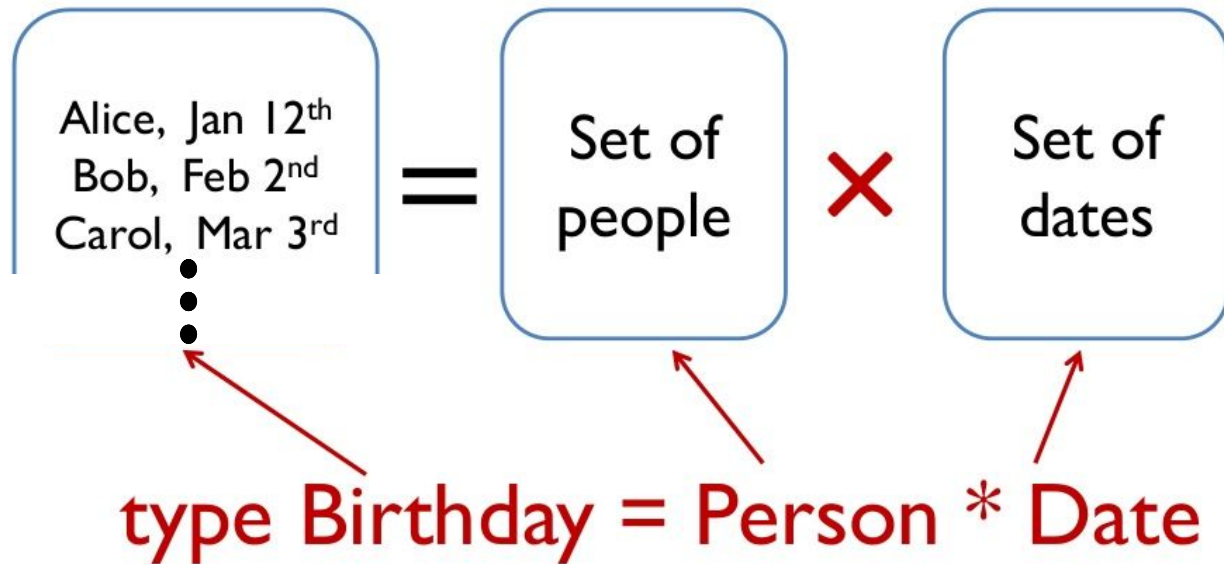
Algebraic Data Types (ADTs)

- We can do algebraic operations on types
- Sum types
- Product types
- Great for expressiveness
- Great for business modelling

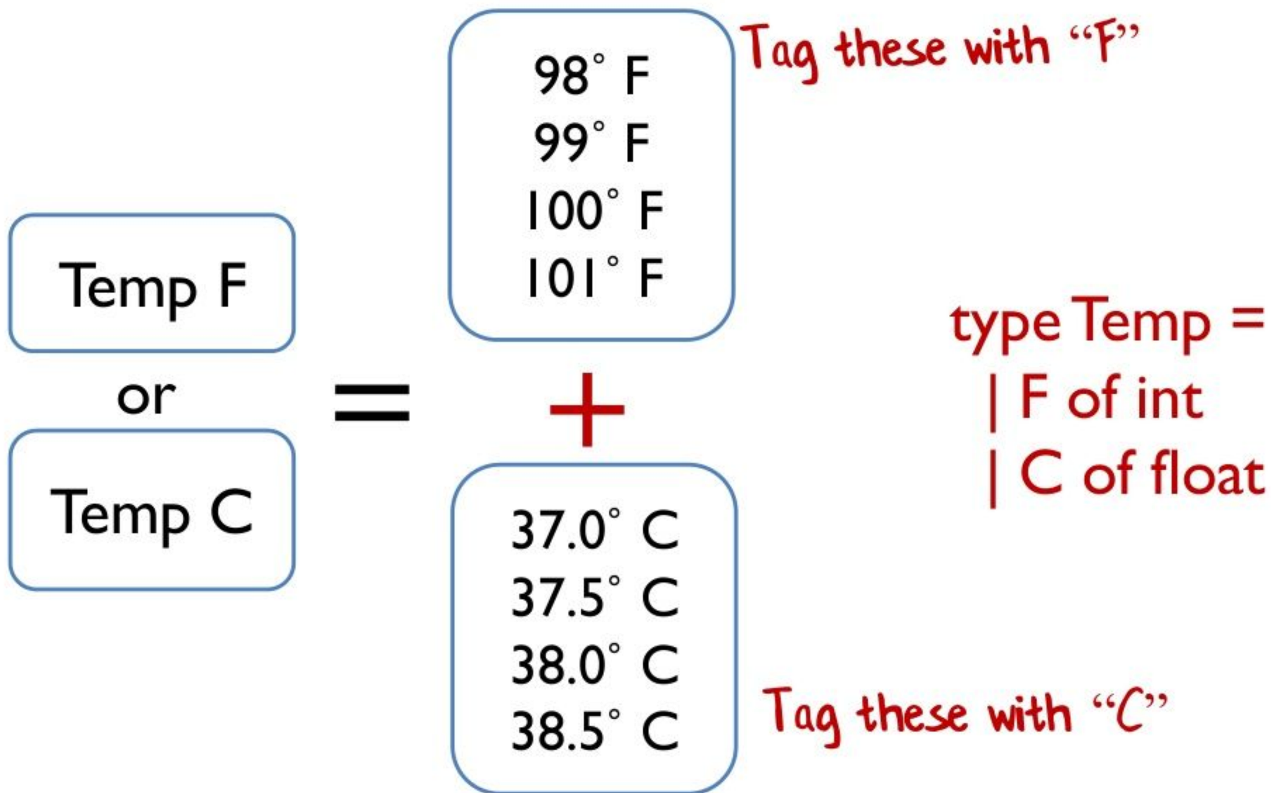
Representing pairs



Using tuples for data



Representing a choice



Domain Modelling

Domain Modelling

```
data Suit = Club | Diamond | Spade | Heart deriving (Show, Read, Eq, Ord)
data Rank = Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten
          | Jack | Queen | King | Ace deriving (Show, Read, Eq, Ord)
data Card = Card (Suit, Rank) deriving (Show, Read, Eq, Ord)
data Hand = Hand [Card] deriving (Show, Read, Eq)
data Deck = Deck [Card] deriving (Show, Read, Eq)
data Player = Player { name:: String, hand :: Hand }
              deriving (Show, Read, Eq)
data Game = Game { deck :: Deck, players :: [Player] }
            deriving (Show, Read, Eq)
data Deal = Deal (Deck -> (Deck, Card))
data PickupCard = PickupCard ((Hand, Card) -> Hand)
```

Type system

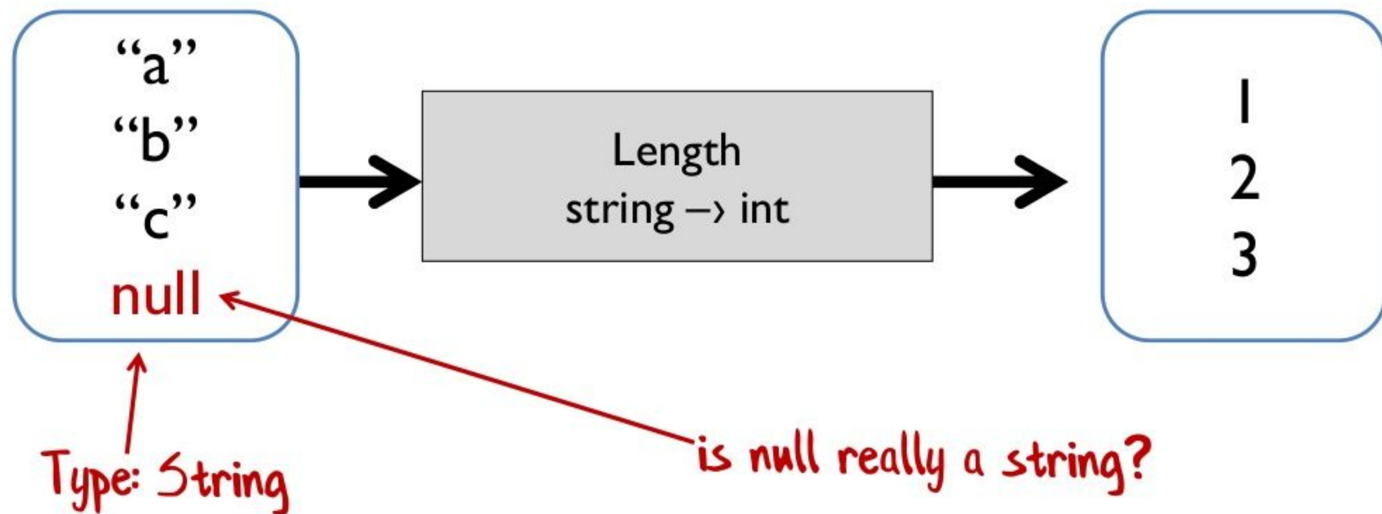
- Your ally
- Let it do its job
- No cheating around
  - Optional values
  - Immutable values
  - `null` and mutability and type casting ruin everything

Optional values

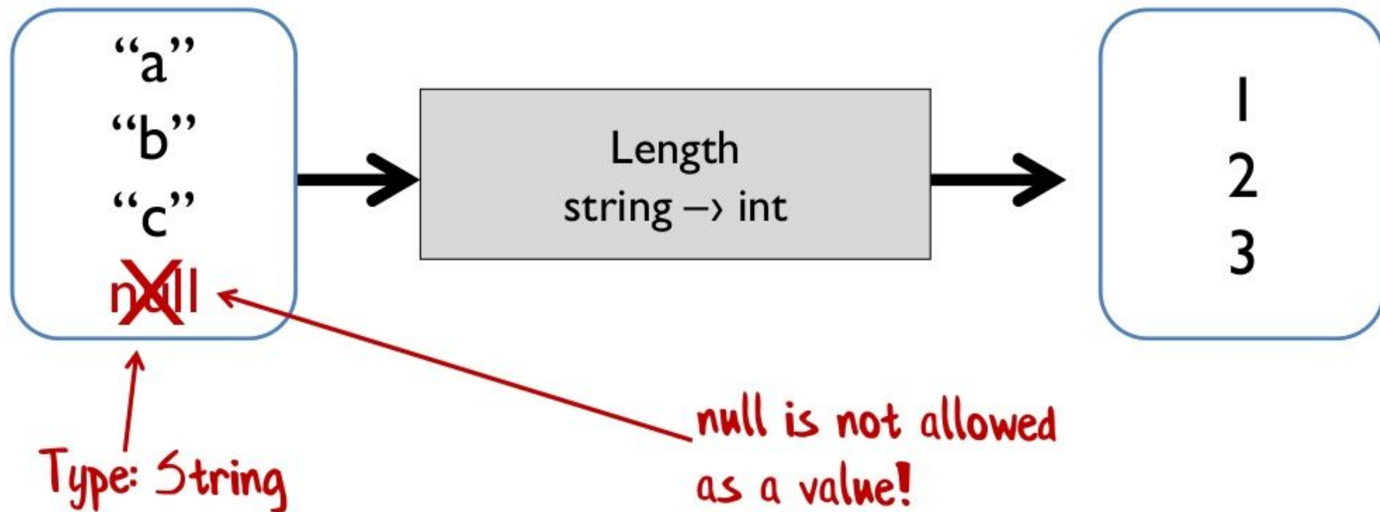
- Better dealt with using ADTs
- `null` is evil
- `null` subverts the type system

`null` is evil!!!

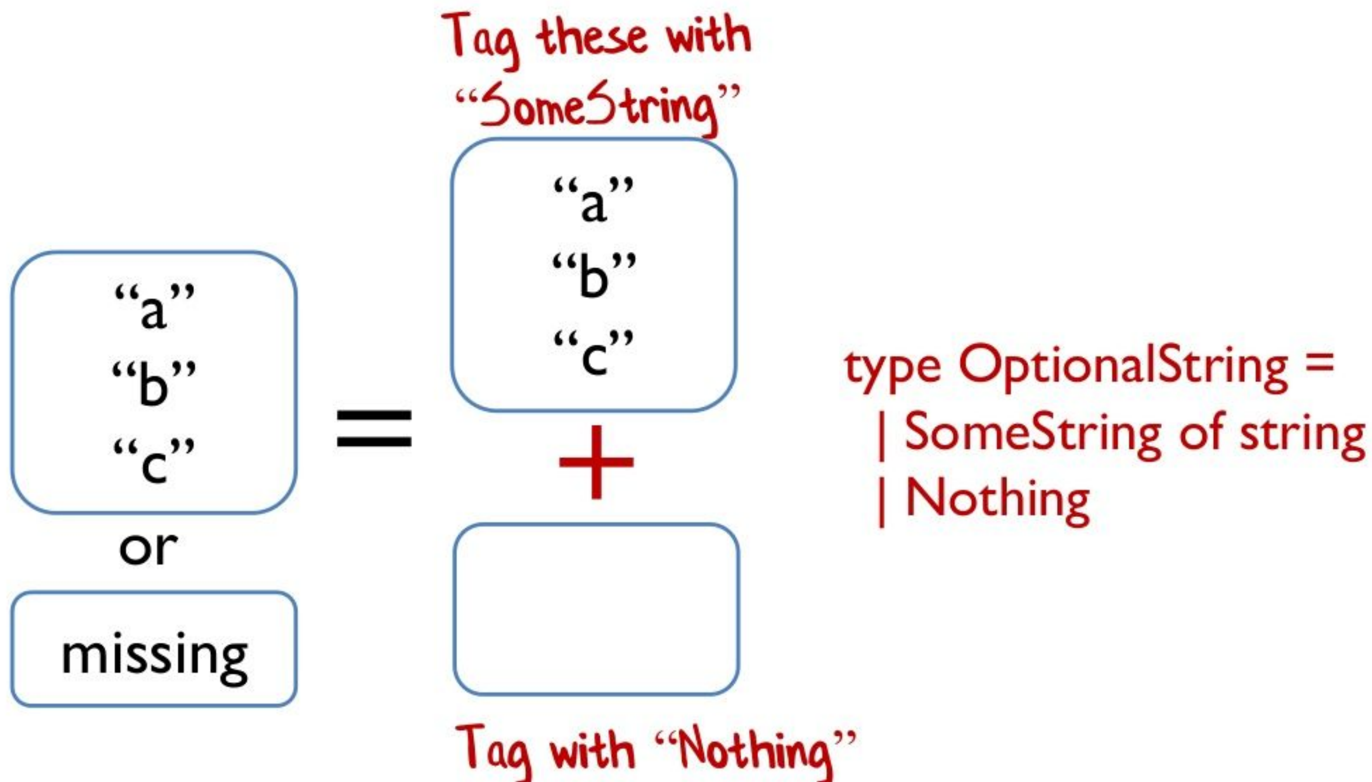
Null is not the same as "optional"



Null is not allowed



A better way for optional values



Make illegal states unrepresentable

```
data Contact = Contact
  { name        :: String
  , emailInfo   :: Maybe Email
  , postalInfo  :: Maybe String
  }
```

Make illegal states unrepresentable

```haskell
data ContactInfo =
    EmailOnly Email
  | PostOnly String
  | EmailAndPost Email String

data Contact = Contact
  { name        :: String
  , contactInfo :: ContactInfo
  }
```

Make illegal states unrepresentable

```
data ConnectionState =
    Connecting
  | Connected
  | Disconnected
```

```
data ConnectionInfo = ConnectionInfo
  { state              :: ConnectionState
  , server             :: InetAddr
  , lastPingTime       :: Maybe Time
  , lastPingId         :: Maybe Int
  , sessionId          :: Maybe String
  , whenInitiated      :: Maybe Time
  , whenDisconnected   :: Maybe Time
  }
```

Make illegal states unrepresentable

```haskell
data Connecting = Connecting
  { whenInitiated :: Time
  }

data Connected = Connected
  { lastPing  :: Maybe (Time, Int)
  , sessionId :: String
  }

data Disconnected = Disconnected
  { whenDisconnected :: Time
  }
```

```haskell
data ConnectionState =
    ConnectingState   Connecting
  | ConnectedState    Connected
  | DisconnectedState Disconnected

data ConnectionInfo =
ConnectionInfo
  { state   :: ConnectionState
  , server  :: InetAddr
  }
```

Pattern matching

- Goes very well with ADTs
- Better than any crazy `if` party
    - More readable
    - Succinct
    - Compiler checks all cases are covered

Smart constructors

```haskell
module Email
  ( Email  -- abstract, hiding constructors
  , create -- only way to create an `Email` instance
  ) where

newtype Email = Email String

create :: String -> Maybe Email
create s = if matchesEmailRegex s
             then Just $ Email s
             else Nothing
```

Functor

- `fmap :: Functor f => (a -> b) -> f a -> f b`

```
f a -> f b

    ↑

  a -> b
```

- f: List, Maybe, Future, IO, ...

Monad

```
example input =
 let x = doSomething input in
 if isJust x then
     let y = doSomethingElse (fromJust x) in
     if isJust y then
         let z = doAThirdThing (fromJust y) in
         if isJust z then
             let result = fromJust z in
             Just result
           else Nothing
       else Nothing
   else Nothing
```

Monad

```
exampleBind input = doSomething input
                >>= doSomethingElse
                >>= doAThirdThing
```

```
exampleDo input = do
  x <- doSomething input
  y <- doSomethingElse x
  doAThirdThing y
```

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- People say
  - Pure functions may be easy
  - Side-effecting functions are difficult
  - Haskell makes I/O difficult
  - Programming in a language which doesn’t distinguish between pure and non-pure code is easier

BUT

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- But
  - We want to support 100k clients
  - OS kernel threads are expensive

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- But
  - We want to support 100k clients
  - OS kernel threads are expensive
  - Callbacks, yay!

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- But
  - We want to support 100k clients
  - OS kernel threads are expensive
  - Callbacks, yay!
    - Callback hell

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

icompile.eladkarako.com

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- But
  - We want to support 100k clients
  - OS kernel threads are expensive
  - Languages are growing solutions similar to Haskell's `IO`
    - Promise (JavaScript), Future (Java), Task (C#), ...
  - cons are similar to Haskell's `IO`, not as many pros as Hakell's `IO`

Pure functional programming
& I/O (side effects)
& Asynchronous programming

- Solution: Haskell's `IO`
  - Support 100k - 1M clients
    - RAM is the limit, so maybe even 10M
  - Lazy, in contrast to Promise (JavaScript), Future (Java), Task (C#)
    - Easier to reason about, combine, restart, etc
  - Clearly separates pure code from side-effecting code
    - e.g. STM
  - So not a burden, but an advantage!

Software Transactional Memory

```haskell
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
atomically :: STM a -> IO a

basicTransfer :: Int
              -> TVar Int
              -> TVar Int
              -> STM ()
basicTransfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  toQty   <- readTVar toBal
  writeTVar fromBal (fromQty - qty)
  writeTVar toBal   (toQty + qty)
```

```haskell
transferTest :: Int
             -> TVar Int
             -> TVar Int
             -> STM (Int, Int)
transferTest qty fromBal toBal = do
  basicTransfer qty fromBal toBal
  liftM2 (,) (readTVar fromBal) (readTVar toBal)

action :: IO ((Int, Int), (Int, Int))
action = do
  alice <- atomically (newTVar 12)
  bob   <- atomically (newTVar 4)
  let res1 = atomically (transferTest 7 alice bob)
  let res2 = atomically (transferTest 5 alice bob)
  concurrently res1 res2
```

Generic programming

- Create functions which work on any data type
- Reduces boilerplate
- e.g. encoders/decoders to JSON
- Similar to macros, **but**
    - Macros based on manipulating AST
    - Generic programing based on shape of ADTs
- Very useful even for consumes of libraries that use Generic programming
- Wait for demo

Property based testing

- Let the computer generate test cases
- More comfortable for the programmer
- Computer can think of cases the programmer would not
- Started in Haskell as QuickCheck
- Wait for demo

Summary of ideas

Types

- ADTs, pattern matching
- Domain modelling
- Make illegal states unrepresentable
- Smart constructors
- Separate side-effects from pure functions
- Let the compiler work for you (Generic, Property testing)

Not just Haskell

Where to go from here

- Stay with Haskell
  - The the best functional programming language, still practical, huge community
- JVM
  - Scala
- .NET
  - F#
- Web browser
  - Elm, PureScript
- C/C++
  - Rust
- Objective-C
  - Swift

Great resources about FP

- Online
  - haskell.fpcomplete.com/learn
  - FsharpForFunAndProfit.com
  - blog.janestreet.com/effective-ml-revisited/
  - Reddit
- Meetups
  - https://www.meetup.com/FSharping/
  - https://www.meetup.com/functional-jvm-meetup/

Functional programming is *practical*

Use it practice!