Chapter 12 : Signaling Adversity

→ data Maybe a = Nothing | Just a : lets us return a default Nothing value, when we don't have any sensible values to return for our intended type a.

↓

For example : ifEvenAdd2 :: Integer → Integer

ifEvenAdd2 n = if even n then n+2 else ??

What if we want to the caller to know that there is no option to be performed on an odd no., rather than returning the odd no, which might not be clear idea about the same.

↓

we use maybe

function appln.

↓

highest precedence

ifEvenAdd2 :: Integer → Maybe Integer

ifEvenAdd2 n = if even n then Just (n+2) else Nothing

*) smart constructors for datatypes :

type Name = String

type Age = Int

data Person = Person Name Age deriving Show

↓

Handling empty String or -ve age values

↓

can be done using maybe

↓

```
mkPerson :: Name → Age → Maybe Person
mkPerson name age
  | name /= "" && Age >= 0 =
      Just $ Person name age
  | otherwise = Nothing
```

↓

returns nothing if the argument out of line.

mkPerson is a smart constructor, since they allow us to create a value for a type only when they meet certain criteria, and return an explicit signal when we don't.

→ Either : wanting to know which of the fields was invalid.

```
data Either a b = Left a | Right b
```

Start by making a sum type to enumerate failure modes:

```
data PersonInvalid = NameEmpty
                   | AgeTooLow
                   deriving (Eq, Show)
```

↓

Case expressions & pattern matching can work without the Eq instance, but guards using (==) will not.

```
module EqCaseGuard where

data PersonInvalid = NameEmpty
                   | AgeTooLow

toString :: PersonInvalid → String

toString NameEmpty = "NameEmpty"
toString AgeTooLow = "AgeTooLow"

instance Show PersonInvalid where
    show = toString
```

| modifying mkPerson fn type signature

```
mkPerson :: Name
         → Age
         → Either PersonInvalid Person
```

| modifying the logic in the fn.

```
mkPerson name age
  | name /= "" && age >= 0 =
      Right $ Person name age.

  | name == "" = Left NameEmpty
  | otherwise = Left AgeTooLow
```

↓

Left of Either is used to mark whatever case is going to stop the code from working.

*) When both cases fail, then we see NameEmpty only, which is not entirely correct. This can be rectified by creating separate checking functions and then combine the results.

↓ addn. of a type alias

```
type ValidatePerson a = Either [PersonInvalid] a
```

↓ checking fns.

```
ageOkay :: Age -> Either [PersonInvalid] Age.
ageOkay age = case age >= 0 of
    True -> Right age
    False -> Left [AgeTooLow]
```

↳ allows us to return both NameEmpty and AgeTooLow in cases both are true.

```
nameOkay :: Name -> Either [PersonInvalid] Name
nameOkay name = case name /= "" of
    True -> Right name
    False -> Left [NameEmpty]
```

↓

→ these still don't provide complete safety from bogous arguments like "42".

↓

gives type error for entering no.

```
mkPerson :: Name
         -> Age
         -> ValidatePerson Person
mkPerson name age =
    mkPerson' (nameOkay name) (ageOkay age)

mkPerson' :: ValidatePerson Name (result of nameOk)
          -> ValidatePerson Age  (result of ageOk)
```

→ validate Person Person (accordingly result of mkPerson)

```
mkPerson' (Right nameOk) (Right ageOk)
    = Right (Person nameOk ageOk)

mkPerson' (Left badName) (Left badAge)
    = Left (badName ++ badAge)
```

→ Left expected a list of PersonInvalid

```
mkPerson' (Left badName) _ = Left (badName)
mkPerson' _ (Left badAge) = Left (badAge)
```

→ Kinds, a thousand stars in your types: Kinds are types one level up. Used to describe the types of type constructors. Type constructors (higher kinded types) take more types as arguments.

↓

Type constant : no args. (Int, Bool, Char)

Type constructors : takes args.

data Example a = Blah | RoofGoats | woot a

(* → *) : Example must be applied to one type to become a concrete type represented by (*).

↓

The two-tuple takes 2 arguments, hence its kind signature : (* → * → *)

↓

Similarly for Maybe & Either.

) Lifted & unlifted types: '' : Lifted types
'#' : Unlifted types.

Lifted types: any type that could be defined ourself, and can be inhabited by bottom. Represented by a pointer that can include most of the datatypes in use or ones that can be encountered.

Unlifted types: types which cannot be inhabited by bottom. Types of kind # are often native m/c types and raw ptrs.

↓

newtypes are a special case, since their kind is '*', but are unlifted because their repr. is identical to the type they contain, so the newtype is not creating any new ptr. beyond that of the type it contains.

↓

newtypes can not be inhabited by bottom., but its underlying type can for sure.

↓

(→) signals need for application in both kind signature & type signature, meaning that we construct another type by applying to a type.

↓

data Maybe a = Nothing | Just a.

:k Maybe
(* → *)
: k Maybe Int → Maybe Int :: *

Maybe Maybe → won't work, because kinds don't match. i.e. the first argument of Maybe should have kind (*) and not (* → *).

↓

instead Maybe Maybe Int should work since Maybe Int reverts to '*' and that is exactly what Maybe needs.

```
(*) : k []
[] :: (* → *)

    : k [] Int
[] Int : *
```

(*) Data constructors are functions: Data constructors that take args do behave like functions. Like functions, their arguments are typechecked against the specification in the type.

```
>> fmap Just [1, 2, 3]
[Just 1, Just 2, Just 3]
```

→ highlighting the fact that Just also behaves like a fn.

Chapter Exercises.

*) Determine the kinds

1) a :: *

2) a :: *
   f :: * → *

*) String processing

list of words

replace

↳

intersperse

1. replaceThe :: String → String

replaceThe s = concat $ intersperse " "

$ replace $ words s

```
replace :: [String] → [String]
replace [] = []
replace (x:xs)
  | x == "the" = "a" : replace xs
  | otherwise = x : replace xs
```

2. countTheBeforeVowel :: String → Integer

```
countTheBeforeVowel s = count $ words s

count :: [String] → Integer
count [] = 0
count (x:[]) = 0
count (x1:x2:xs)
  | x1 == "the" && ((x2!!0) `elem` "aeiou") = 1 + count (xs)
  | otherwise = count xs
```

*) validate the word

count vowels & constants. If vowels > constants, return nothing.

```
newtype Word' = Word' String
                deriving (Eq, Show)

vowels = "aeiou"

mkWord :: String -> Maybe Word'
mkWord s =
    if v > (length s - v)
    then Nothing
    else Just (Word' s)
    where v = countVowels s
```

*) It's only Natural

```
data Nat = Zero
         | Succ Nat
         deriving (Eq, Show)

natToInteger :: Nat -> Integer
natToInteger Zero = 0
natToInteger (Succ x) = 1 + natToInteger x

integerToNat :: Integer -> Maybe Nat
integerToNat x
  | x < 0  = Nothing
  | x == 0 = Zero
  | x > 0  = Succ $ integerToNat $ x-1
```

*) Small library for Maybe

Maybe catamorphism

```
maybee :: b → (a → b) → Maybe a → b

catMaybes :: [Maybe a] → [a]
catMaybes [] = []
catMaybes (Nothing : xs) = catMaybes xs
catMaybes (Just x : xs) = x : catMaybes xs

flipMaybe :: [Maybe a] → Maybe [a]
flipMaybe [] = Just []
flipMaybe (Nothing : xs) = Nothing
flipMaybe (Just x :- xs) = fmap (x :) (flipMaybe xs)
```

*) Small library for Either

```
lefts' :: [Either a b] → [a]
lefts' = foldr ((:) . (ifLeft)) []

partitionEithers' ::
    [Either a b] → ([a], [b])
partitionEithers' x =
    (lefts' x, rights' x)
```

```
foldr (+) z (x : xs)
(+) x (foldr (+) z xs)

ifLeft :: Either a b → [a]
ifLeft Left x = [x]
ifLeft _ = []

ifRight :: Either a b → [b]
ifRight Right x = [x]
ifRight _ = []
```

```
eitherMaybe' :: (b → c)
             → Either a b
             → Maybe c
eitherMaybe' _ (Left x) = Nothing
eitherMaybe' f (Right x) = Just (f x)

either' :: (a → c) → (b → c) → Either a b → c
either' f _ (Left x) = f x
either' _ g (Right x) = g x

eitherMaybe'' :: (b → c) → Either a b → Maybe c
eitherMaybe'' f w@(Right x) = Just (either' f id w)
eitherMaybe'' _ _ = Nothing
```

→ Unfolds: anamorphisms: opposite of catamorphisms in the way that they allow us to build structures up. There are a few ways to unfold a data structure. These can be used to create finite & ∞ data structures alike.

① iterate :: (a → a) → a → [a] (infinite)

```
take 10 $ iterate (+1) 0
```

② unfoldr (more general) :: (b → Maybe (a, b)) → b → [a]

```
take 10 $ unfoldr (\b → Just (b, b+1)) 0
```

The principle of abstracting out common patterns and giving them names applies as well to unfolds as it does to folds.

*) Detour

```
mehProduct :: Num a => [a] -> a
mehProduct xs = go 1 xs
  where go :: Num a => a -> [a] -> a
        go n [] = n
        go n (x:xs) = (go (n*x) xs)
```

→ write your own iterate and unfoldr

*) myIterate :: (a -> a) -> a -> [a]

```
myIterate f x = (f x) : myIterate f (f x)
```

*) myUnfoldr :: (b -> Maybe (a, b)) -> b -> [a]

use a go method in order to have an incremental value, & create a new variable by applying the func. i/p argument to the other i/p argument.

```
myUnfoldr f x = go f x (f x) where
        go f s Nothing = []
        go f s (Just (a, b)) = a : go f b (f b)
```

*) betterIterate :: (a -> a) -> a -> [a]

```
betterIterate f x = myUnfoldr
```

→ Binary Tree

```
data BinaryTree a =
    Leaf
  | Node (BinaryTree a) a (BinaryTree a)
  deriving ()
```

1. unfold for binarytree

```
unfold :: (a → Maybe (a, b, a))
       → a
       → BinaryTree b
unfold f x = go f x f x where
  go f x Nothing = Leaf
  go f x (Just (a, b, c)) = Node
      (go f b (f a)) (b) (go f c (f c))
```

```
unfold f x =
  case (f x) of
    Nothing → Leaf
    Just (l, m, r) → Node (unfold f l) m
                          (unfold f r)
```

2. Tree Builder

```
treeBuild :: Integer → BinaryTree Integer
treeBuild x
  | x < 0 = Leaf
```

```
| otherwise = unfold f 0
    where f k
            | k < x  = Just (k+1, k, k+1)
            | k == x = Nothing
```