

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 : ifEvenAdd 2 :: Integer → Integer

ifEvenAdd2 n = if even n then n+2 else 

function appln.

↑
highest precedence

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

ifEvenAdd 2 :: 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 for 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 [Person|invalid] a

↓ checking fns.

ageOkay :: Age → Either [Person|invalid] 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 [Person|invalid] Name

nameOkay name = case name /= " " of
True → Right name
False → Left [NameEmpty]

→ these still don't provide complete safety from bogus arguments like "q2".

mkPerson :: Name

→ Age
→ validatePerson Person

↓
gives type error for entering no.

mkPerson name age =

mkPerson' (nameOkay name) (ageOkay age)

mkPerson' :: validatePerson Name (result of nameOk)

→ validatePerson Age (result of ageOk)

→ validatePerson 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 | RoofGoots | 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 oneself, 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.

Unglifted 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 $(* \rightarrow *)$.

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

$(*) : K []$
 $[] :: (* \rightarrow *)$
 $: k [] \text{Int}$
 $[] \text{Int} : *$

(*) 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.