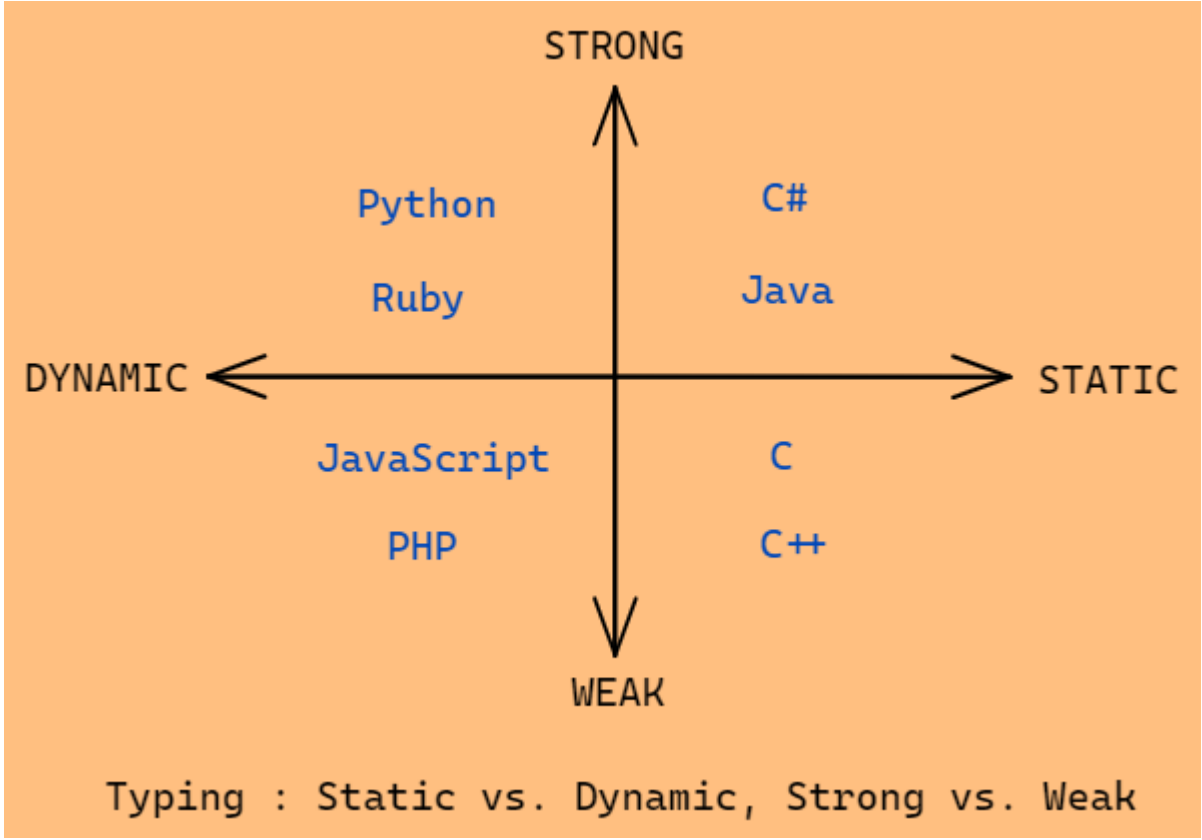
Repaso Haskell

Diego Acuña – TC – Marzo 2026

¿Qué es Haskell?

- Lenguaje de programación funcional puro
- Evaluación perezosa
- Tipado estático y fuerte



Sintaxis

- Definición de funciones

  ```
  suma :: Int -> Int -> Int
  suma a b  =  a + b
  ```

- Invocación de funciones

  ```
  suma 2 3
  ```

- Variables

  ```
  x = 10
  y = x +5
  ```

- Comentarios : -- para 1 línea y {- para bloque -}

Tipos

- Funcionales
- Abstractos de datos
- Árboles

Definir un tipo: data vs. type

data se usa para crear nuevos tipos de datos reales, con constructores propios y estructura internaEn cambio, type se usa para crear sinónimos de tipos, es decir, alias de tipos existentes, sin crear un nuevo tipo en el sistema de tipos.

Un tipo definido con data es distinto a nivel de tipos, mientras que un type es solo un nombre alternativo para un tipo ya existente.

Por tanto, data crea tipos nuevos con identidad propia y type solo renombra tipos existentes para mejorar legibilidad.

Definir un tipo: data vs. type

```
data N  where { O :: N ; S :: N -> N}

type Nat = N
```

¿Son lo mismo?

```
f :: N -> Nat        f :: Nat -> N        f :: N -> N        f:: Nat -> Nat
```

Crear un tipo nuevo

Notación de fundamentos:

```
data N  where { O :: N ; S :: N -> N}
```

Notación de teoría:

```
data N = O | S N
```

RECORDAR:

Tipos y sus constructores deben empezar con mayúscula

Funciones y variables con minúscula

Crear un tipo nuevo: newtype

newtype se utiliza para crear un nuevo tipo a partir de un tipo existente, pero sin el coste de memoria ni de ejecución que tiene data. A diferencia de type, newtype sí crea un tipo distinto en el sistema de tipos, pero solo puede tener un único constructor y un único campo. Se usa cuando se quiere seguridad de tipos (evitar confusiones entre tipos iguales internamente) sin penalización de rendimiento. Es común en TADs, wrappers semánticos y para dar significado a valores simples como Int o String.

Ejemplo: newtype Natural = N Nat

Lambda notation and cases

Fundamentos:

```
par = \n -> case n of {
                O -> True;
                S x -> not (par x) }
```

capaz alguno:

```
par n = case n of {
                O -> True;
                S x -> not (par x) }
```

Pattern matching

Fundamentos:

```
par n = case n of {
                    O -> True;
                    S x -> not (par x) }
```

Teoría:

```
par O = True
par (S x) = not (par x)
```

Cases vs Guardas

```
filter  :: (a-> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = case p x of {
                        True -> x: (filter p xs);
                        False ->filter p xs}
```

```
filter  :: (a-> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
 | p x = x : (filter p xs)
 | otherwise = filter p xs
```

Existe el if then else también!

Clases

Una type class define comportamientos; los tipos implementan esos comportamientos como instancias.

elem :: Eq a => a -> [a] -> Bool

```
class  Eq a  where
  (==), (/=)              :: a -> a -> Bool
  x /= y                  =  not (x == y)
```

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

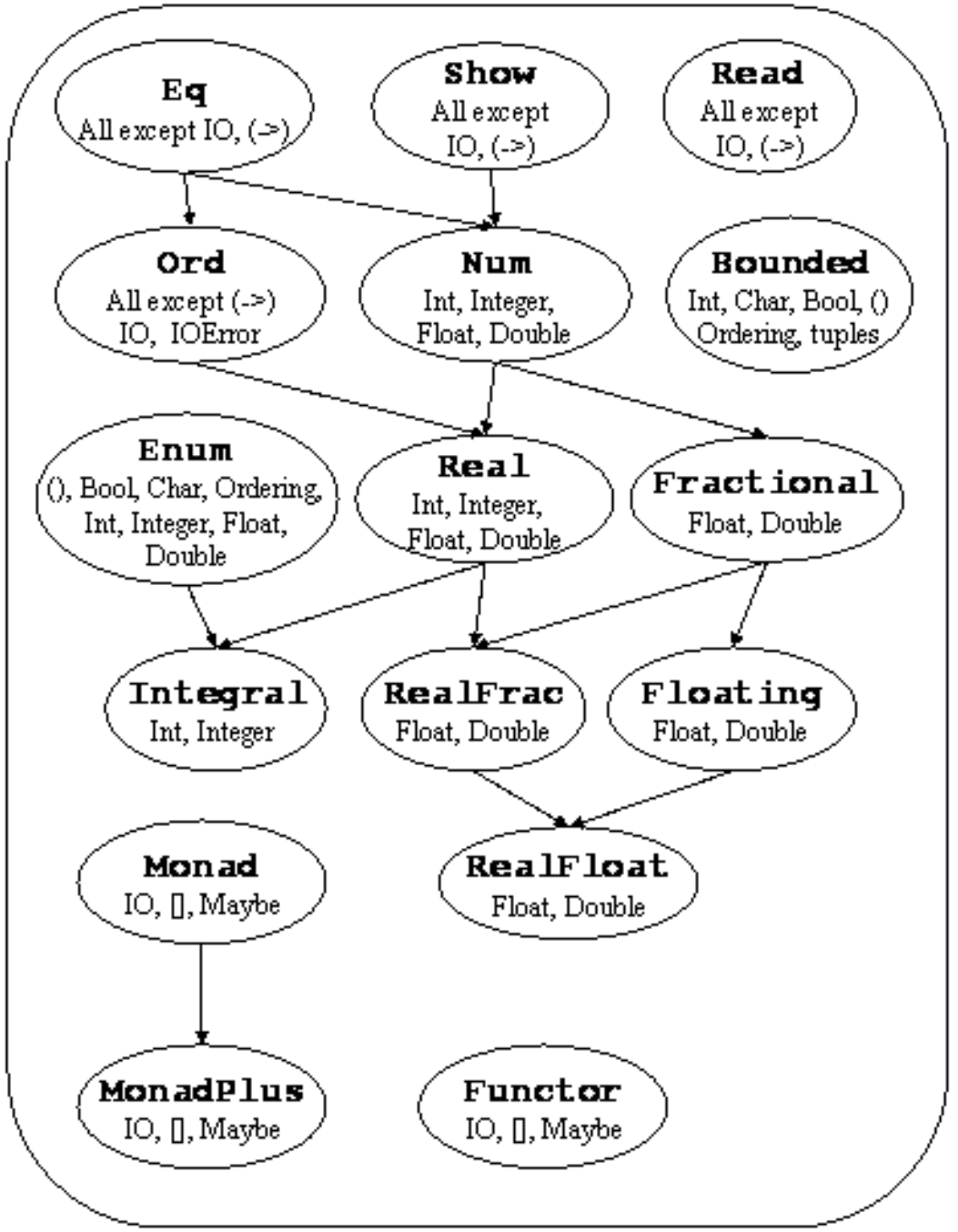
Clases

Los que vamos a usar:

- Show
- Eq
- Ord

Ej:

data N = O | S N deriving Show



Data Maybe

Se utiliza para representar valores que pueden existir o no existir. En lugar de usar null, -1 o valores inválidos, Haskell usa Maybe para modelar la ausencia de datos de forma segura.

data Maybe a = Nothing | Just a

Cláusula where

Ejemplos

```
notaFinal parcial defensas trabajo
  | promedio >= 70 = "Aprobado"
  | otherwise     = “Eliminado"
  where
    promedio = (parcial * 0.3) + (defensas * 0.5) + (trabajo * 0.2)
```

Cláusula let

```
precioFinal precio =
  let iva = precio * 0.22
      total = precio + iva
  in total
```

```
distancia (x1,y1) (x2,y2) =
  sqrt (dx*dx + dy*dy)
  where
    dx = x2 - x1
    dy = y2 - y1
```