¿POR QUÉ TIPOS DE DATO ALGEBRAICOS?

Sum Type y búsqueda de patrones sobre ellos

Run Reset EDIT ON ScalaFiddle

```
sealed trait Triple
case class L(l: Boolean) extends Triple
case class M(m: Boolean) extends Triple
case class R(r: Int) extends Triple

val r : Triple = L(true)

val result = r match {
  case L(true) => "L_True"
  case M(false) => "M_False"
  case R(1) => "R_1"
  case _ => "Perl6"
```

RESULT

run ▶ open in repl.it

main.hs

```
1  data Triple = L Bool | M Bool | R Int
2  f :: Triple -> String
3  f x = case x of L True -> "L True"
```

```
GHCi, version 8.6.3
```

Product Type y búsqueda de patrones

Run Reset EDIT ON ScalaFiddle

```scala
case class Pair[A, B] (a: A, b: B)

def processPair[A,B](pair: Pair[A,B]) = {
  pair match {
    case Pair(fst: Int, snd :Int) => "Pair (Int, Int) "
    case Pair(fst: Int, snd :String) => "Pair (Int, String)"
    case Pair(fst: Boolean, snd :Boolean ) => "Pair (Bool, Bool)"
    case _ => "Pair of something else"
  }
}
// println(processPair(Pair(1,"2")))
```

RESULT

run ▶ open in repl.it

main.hs

```haskell
1  data Pair a b = P a b deriving Show
2  pairBoolBool :: Pair Bool Bool
3  pairBoolBool = P True True
```

```
GHCi, version 8.6.3
```

Exponential type

Run Reset EDIT ON ScalaFiddle

```
type BoolToBoolET = Boolean => Boolean
val notF : BoolToBoolET = (b: Boolean) => ! b

// Additional :: Partial Type Application via alias
type T[A] = Option[Map[Int, A]]
val t: T[String] = Some(Map(1 -> "abc", 2 -> "xyz"))
```

RESULT

run ▶ open in repl.it

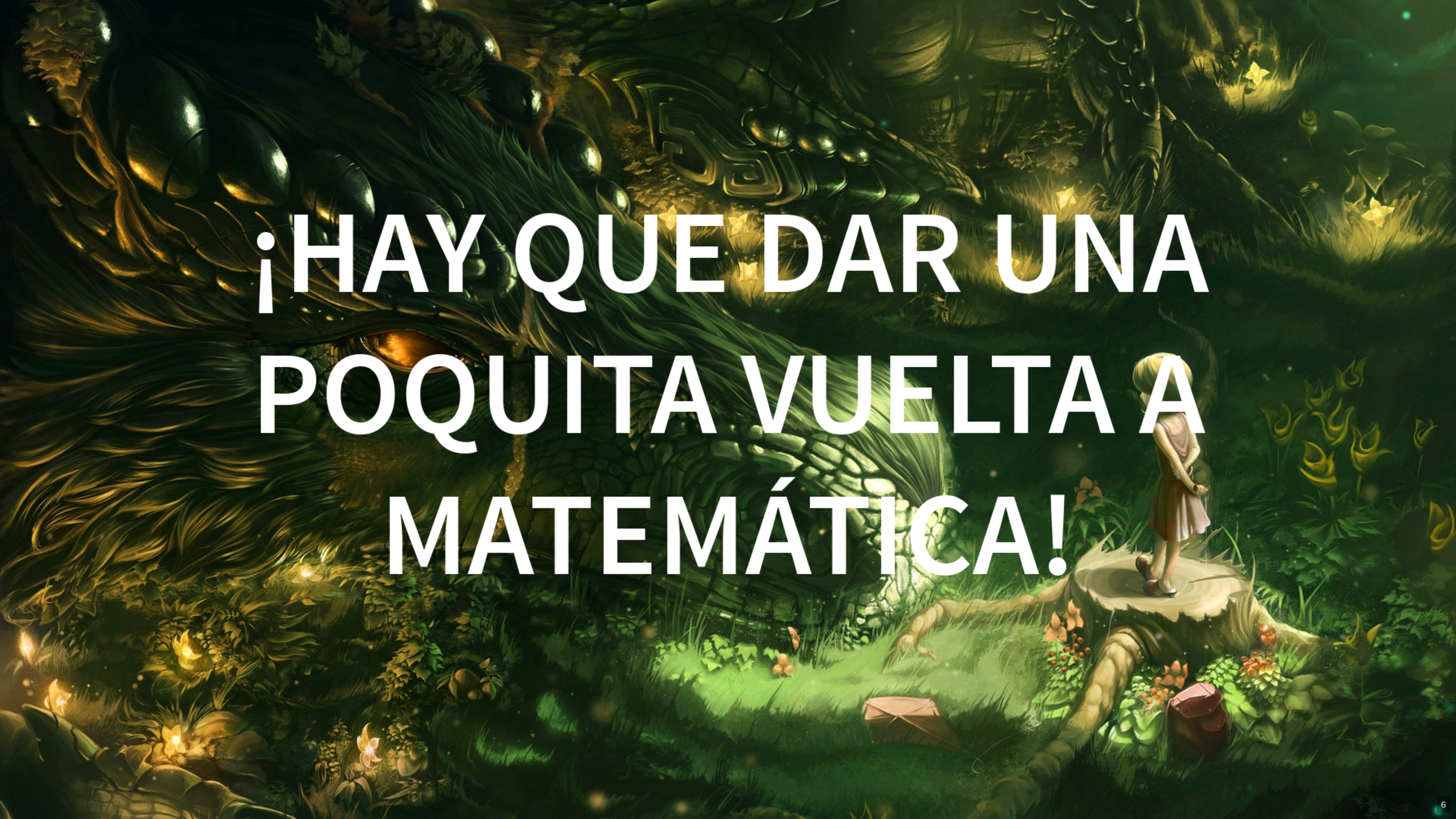
main.hs

```
1  type BoolToBool = Bool -> String
2  notAsString :: BoolToBool
3  notAsString True = "Fasle"
```

```
GHCi, version 8.6.3
```

PERO...

¿POR QUÉ SON ALGEBRAICOS?


¡HAY QUE DAR UNA POQUITA VUELTA A MATEMÁTICA!

CADA ADT LLEVA SU OPERADOR PARA REALIZAR SU ÁLEGEBRA

$$a \mid b = |a| + |b|$$

$$(a, b) = |a| * |b|$$

$$a \to b = |b| \,\hat{}\, |a|$$

PARTE UNO

LOS TIPOS Y SUS CARDINALIDADES

```
|Void|   = 0
|()|     = 1
|Bool|   = 2
-- De aquí ya podemos realizar lo que siguiente
type Product = |(Bool, Bool)|      = ???
type Sum = |Either Bool Bool| = ???
type Exponential = |(Bool -> Bool)|    = ???
```

La correspondencia unívoca tipos A y B con las mismas cardinalidades son isomorfismos

TODO absurd :: Ex falso sequitur quodlibet

ISOMORPHISMO CURRY-HOWARD

Álgebra	ADT
a + b <=> ∑	Either a b
a * b	(a, b)
b^a <=> ∏ b	a -> b
1	( )
0	void

PARTE DOS - DADA LA DEMOSTRACIÓN

ISOMORFISMO ENTRE ENTRE MATEMÁTICA Y TIPOS

```
(a * b)^c = a^c * b^c
a * ( b + c ) = a * b + a * c
-- not that easy
(a^b)^c = a ^ (b * c)
-- TODO show maxBound
-- TODO show Data.Void
-- curry/uncurry
```

LA FORMA CANÓNICA DE ADT

$$t = \sum \prod t\ m,\ n\ m\ n$$

```
-- canónicas ::
Either a b
a -> b
Either a (Either b, (c,d))
-- no canónicas ::
(a, Either b c)
(a, Bool)
```

LAS VARIANTES DEL TIPOS

“ A Profunctor is a Contravariant Functor on its first type parameter and a Functor on its second type parameter. ”

data Variance = Covariant | Contravariant | Invariant

import Data.Functor

A Variance of type T a is fully specified by positive, negative (or mix of both) type variable position.

```haskell
newtype F1 a = F1 ( Int -> a ) -- + => + Covariant
newtype F2 a = F2 ( a -> a ) --  +/- => +/- Invariant
newtype F3 a = F3 ((Int -> a) -> Int ) -- Contravariant - * + =>
newtype F4 a = F4 ((a -> Int) -> Int ) -- Covariant - * - => +
```

SCALA -> (HASKELL, IDRIS, PERL 6)

=> PATTERN MATCHING - POWER OF THE TYPES

=> TYPE CLASSES VS. INTERFACES

=> KINDS - AS A LABELS FOR THE TYPES

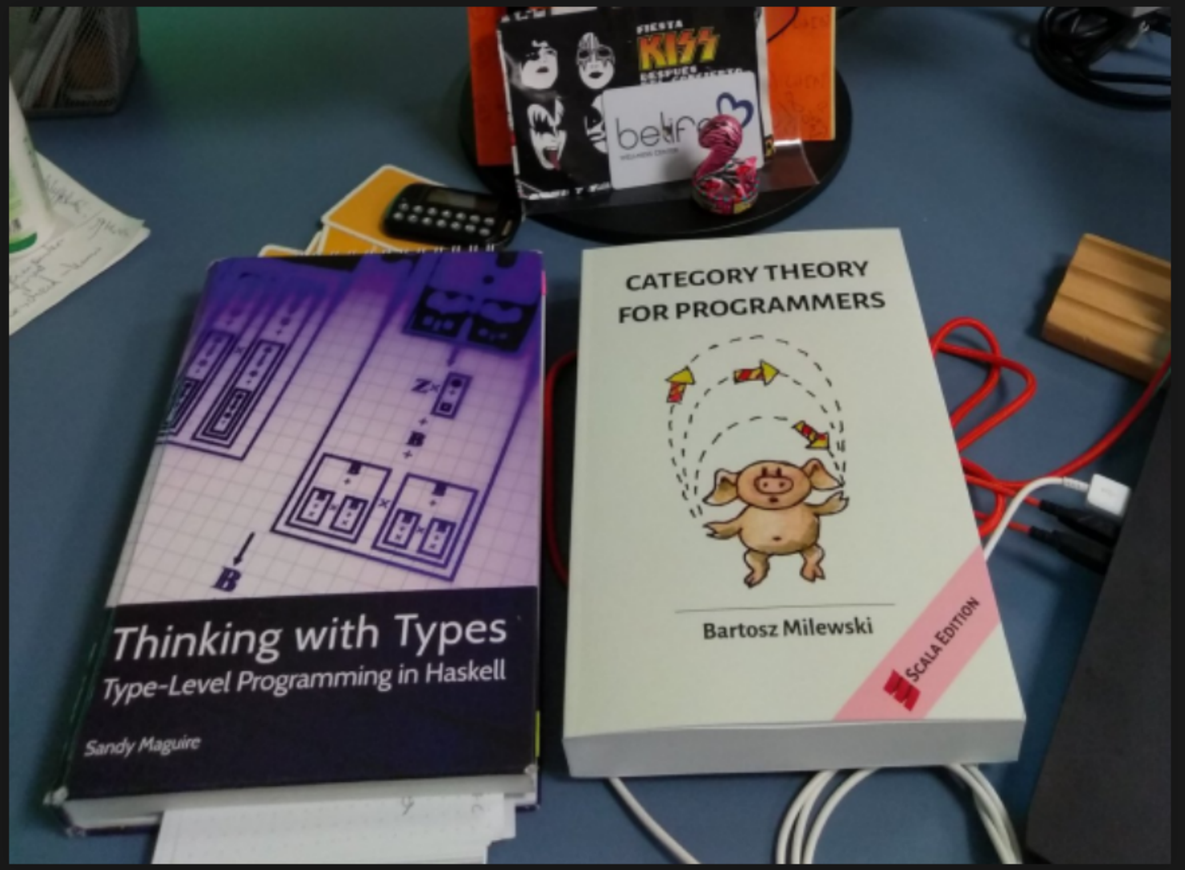
=> GADTS - GENERALIZED ADT

=> PARTIAL TYPE APPLICATION/TYPE LEVEL PROGRAMMING

=> A BETTER SCALA REPL AMMONITE.IO

=> TYPELEVEL CATS LIBRARY

=> SCALAZ SCALAZ-ZIO


KISS
Thinking with Types
Type-Level Programming in Haskell
Sandy Maguire
CATEGORY THEORY
FOR PROGRAMMERS
Bartosz Milewski
SCALA EDITION

¿Why?

Becasue writing like this we programming by giving proof.

¡Think about it!

¡¡EXTRA!! GADTS

```
role Maybe[::A] { }
sub just(::A $x --> Maybe[::A]) { class Just does Maybe[::A] { has $.V = $x; }.new }
sub nothing(Any:U $t --> Maybe[$t]) { class Nothing does Maybe[$t] { }.new }
```

```
data MyMaybe a where
        MyJust    :: a -> MyMaybe a
        MyNothing :: MyMaybe a
```

¡MUCHAS GRACIAS!

THANKS!