LPP - 3CFU : COSA

Sollecitare l'uso delle Type Class

(Typeclassopedia)

* FUNCTOR
* APPLICATIVE
* MONAD ← Perché

Computational effects in pure functional style

LPP - 3CFU : coME

* Definiamo un linguaggio
  {costrutti numerici, una operazione}

* ⟦Costante numerica⟧ = numero binario

⟦operazione⟧ =

```
div :: BN -> BN -> BN
div N0 _ = N0
div n d | n (<) d = N0
        | otherwise = succ(div (sub n d) d)
```

div ?

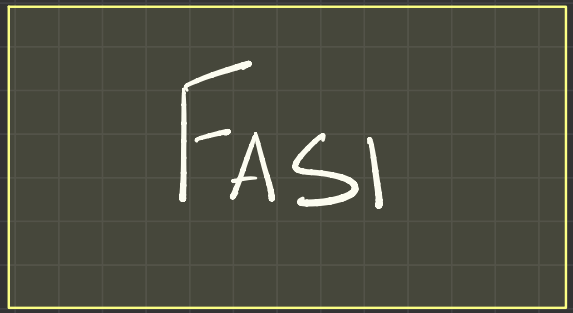
```
div :: BN -> BN -> BN
div N0 _ = N0
div n d | n (<) d = N0
        | otherwise = succ (div (sub n d) d)
```

Tipo Binary Numbers

BN dovrà essere istanza di Ord

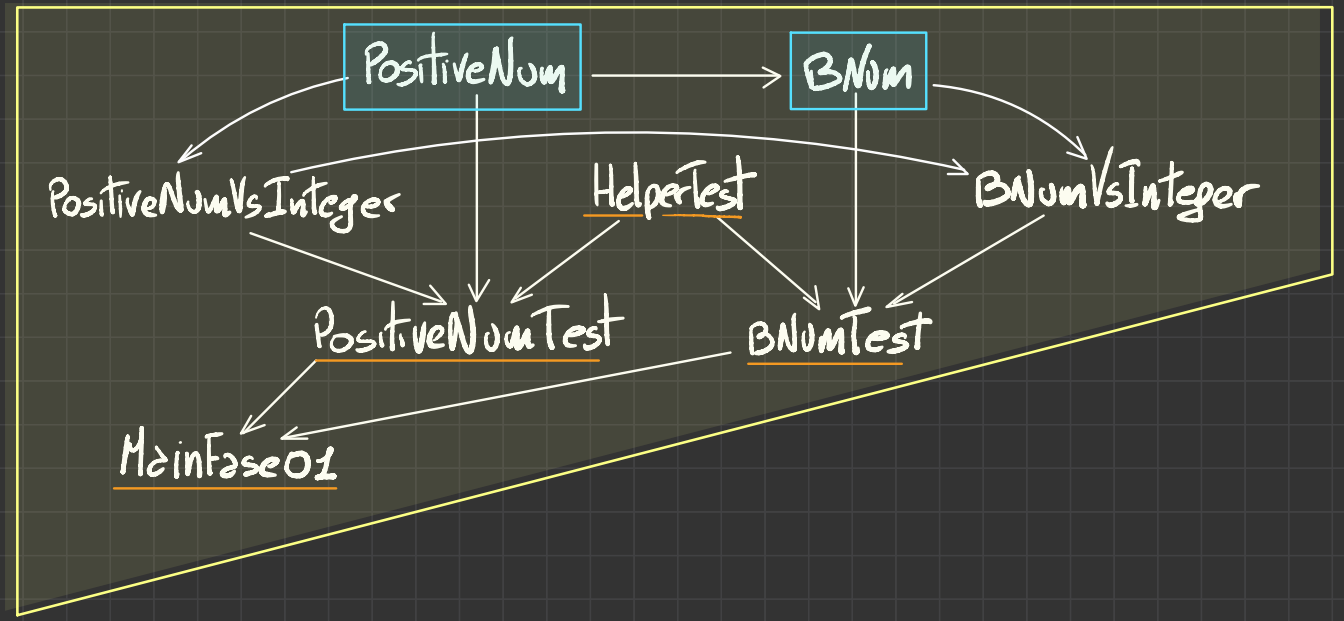
necessarie per rendere BN istanza di Num

∃ una "migliore" per i nostri obiettivi ?


FASI

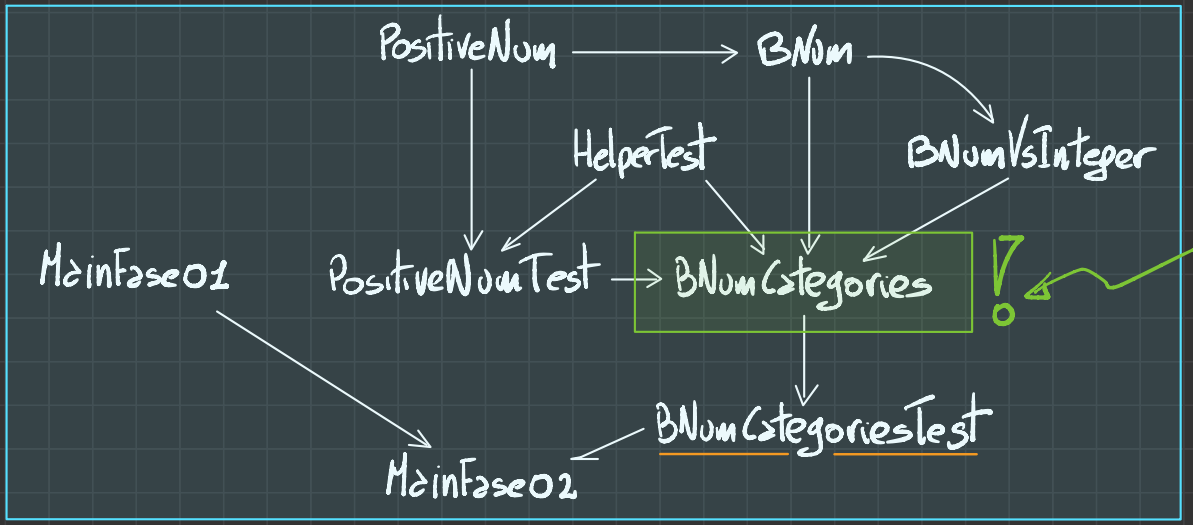
1 ma

Numeri binari
Operazioni
} istanza di Num



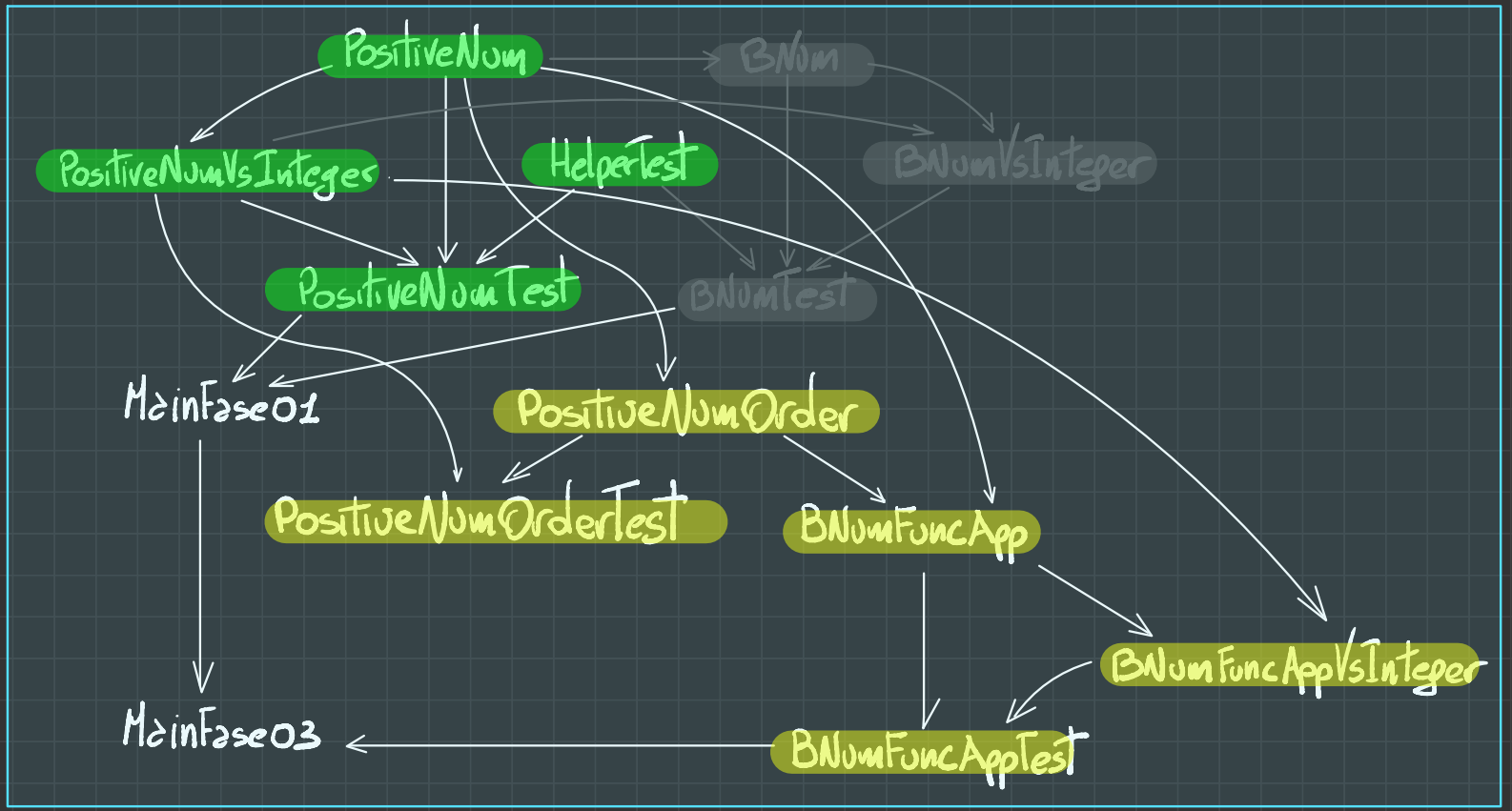
2da

BNum ISTANZA DI { Funtore ? / Applicative ? }



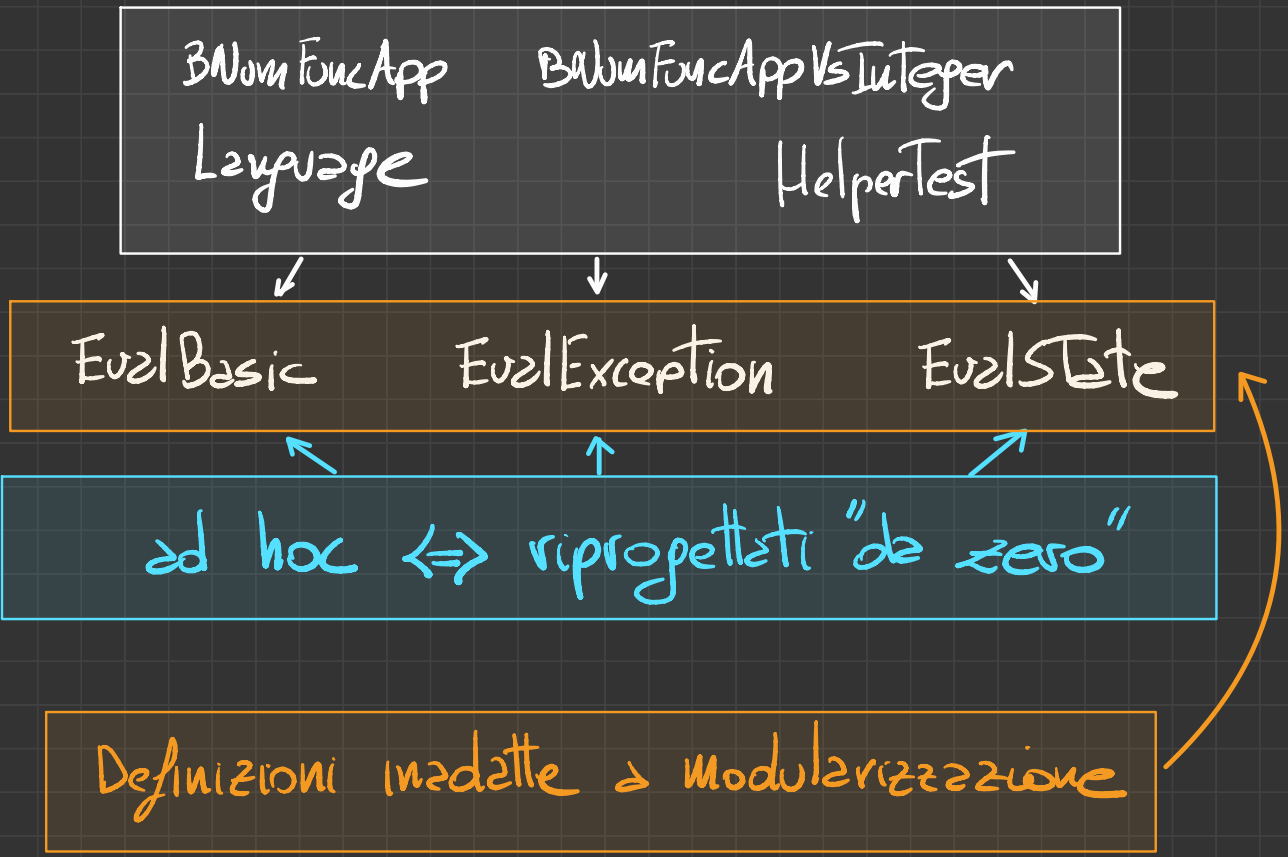
$$\frac{\text{Categoria}}{\text{Funtore}} \simeq \frac{\text{Haskell}}{\text{TypeClass}}$$

322 Numeri binari istanze di Num e Orol

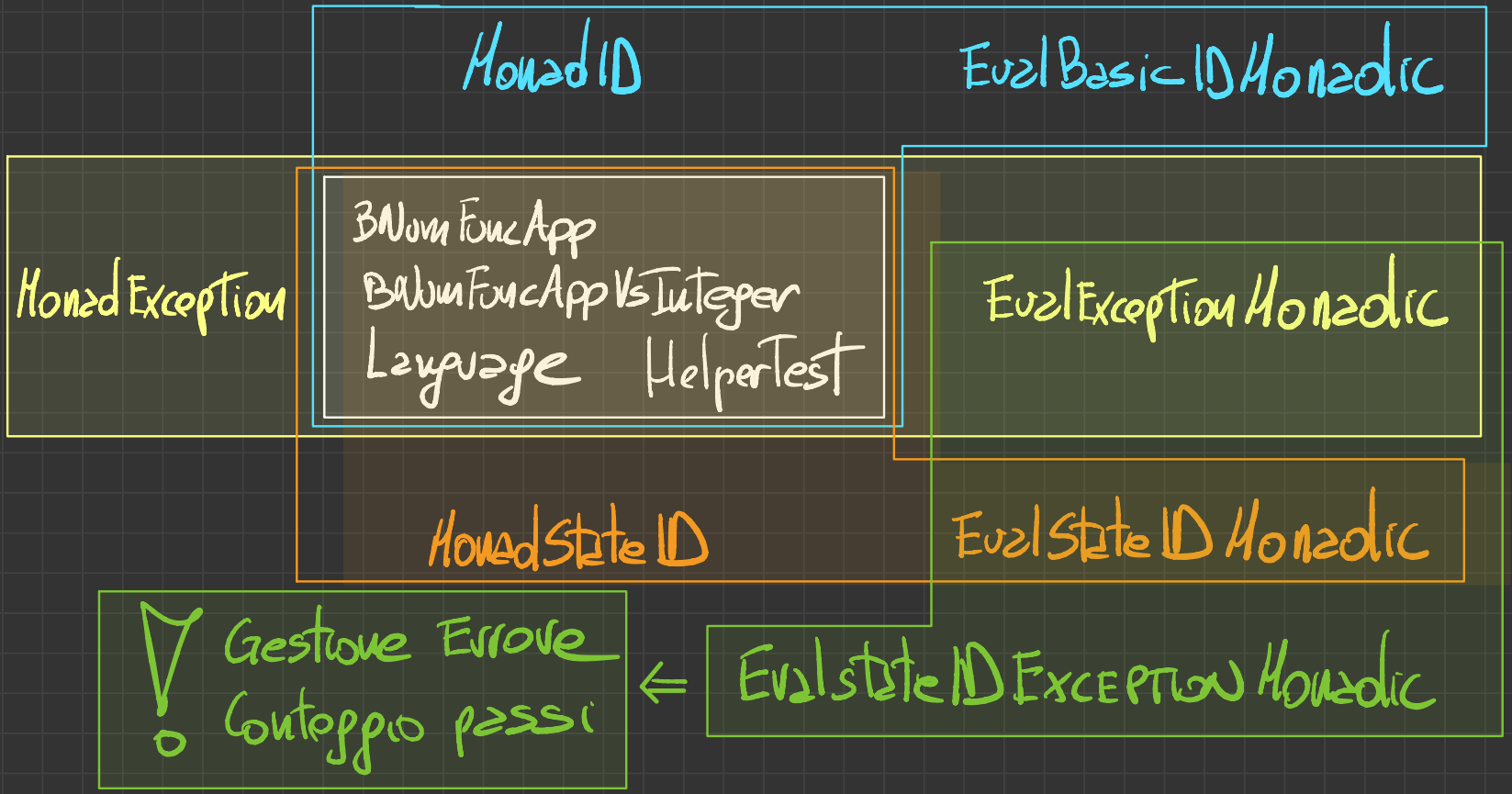


4ta

INTERPRETI "OVVI"



5 tz — Interpreti "fattorizzati" e modulari



6T2 Da Haskell a Proof-Assistant per dimostrare proprietà

Tipo a Istanza di:

Num <=> { + :: a → a → a ; − :: a → a → a ; * :: a → a → a ; negate :: a → a } ∪ div

Ord <=> a istanza di Eq
≤ :: a → a → Bool
! compare :: a → a → Ordering

Eq <=> = :: a → a → Bool
≠ :: a → a → Bool

Interpreti modulari conseguenza di:

class Functor f
fmap :: (a → b) → f a → f b
+ functor laws

class Applicative f
pure :: a → f a
<*> :: f (a → b) → f a → f b
t.c. fmap g x = pure g <*> x

class Monad f
>>= :: f a → (a → f b) → f b
return = pure
t.c. return identità sx, dx
etc.

hs-to-coq ? (Tutorial: Verify Haskell Programs with hs-to-coq)