Tipos Recursivos

- ¿Podemos definir un tipo con infinitos valores?
- En Matemáticas los conjuntos infinitos suelen definirse por inducción.
- En Haskell se pueden definir tipos infinitos declarando tipos recursivos

Todo número natural puede representarse con un valor del tipo natural

- data Nat = Cero|Suc Nat deriving Show
- uno = Suc Cero
- dos = Suc(Suc Cero)

```
Main> uno
Suc Cero
Main> :t Cero
Cero :: Nat
Main> :t uno
uno :: Nat
Main> :t dos
dos :: Nat
```

{ Cero, Suc Cero, Suc (Suc Cero), . . . }

Las evaluaciones que producen error o que no terminan (divergen) se reducen a _|_ (se lee botton)

- Entonces, los valores del tipo Nat que no representan ningún natural, debido a que el valor undefined pertenece a cualquier tipo.
- `Main> :t undefined`
- `undefined :: a`
- Podemos hacer :
- indefinidoN :: Nat
- indefinidoN = undefined

- `Main> indefinidoN`

- `Program error: Prelude.undefined`

Al ser indefinidoN un valor del tipo Nat también lo son Suc indefinidoN , . . .

- `Main> :t Suc indefinidoN`
- `Suc indefinidoN :: Nat`
- `Main> :t Suc (Suc indefinidoN)`
- `Suc (Suc indefinidoN) :: Nat`

- **Podemos decir que** `Suc indefinidoN esta algo mas definido que indefinidoN`

La evaluación perezosa permite no evaluar completamente el argumento :

- esCero :: Nat -> Bool
- esCero Cero = True
- esCero _ = False

- `Main> esCero indefinidoN`

- `Program error: Prelude.undefined`

- `Main> esCero (Suc indefinidoN)`
- `False`

También se puede definir el numero natural mayor que cualquier otro

- infinitoN :: Nat
- infinitoN = Suc infinitoN

- ```
  Main> infinitoN
  ```
- ```
  Suc (Suc (Suc (Suc (Suc
  (S{Interrupted!}
  ```

Las funciones que actúan con datos recursivos se definen elegantemente

- esPar :: Nat -> Bool
- esPar Cero = True
- esPar (Suc x) = not (esPar x)

- `Main> esPar Cero`
- `True`
- `Main> esPar uno`
- `False`
- `Main> esPar (Suc (Suc (Suc (Cero))))`
- `False`

El operador +> ha sido definido de modo recursivo sobre su segundo argumento

- infixl 6 +>
- (+>) :: Nat -> Nat -> Nat
- (+>) m Cero = m
- (+>) m (Suc n) = Suc ((+>) m n)

- `Main> uno +> dos`
- `Suc (Suc (Suc Cero))`

El operador <+ ha sido definido de modo recursivo sobre su primer argumento

- infixl 6 <+
- (<+) :: Nat -> Nat -> Nat
- (<+) Cero n = n
- (<+) (Suc m) n = Suc ((<+) m n)

- `Main> uno <+ dos`
- `Suc (Suc (Suc Cero))`

Funciones de plegado (fold)

- Son funciones de orden superior que capturan la recursión sobre tipos recursivos.

- Muchas de las funciones definidas sobre el tipo
- Nat siguen un mismo patrón recursivo .
- Las funciones esPar , +> , siguen la siguiente plantilla :

- fun :: Nat -> a
- fun Cero = e
- fun (Suc n) = f (fun n)

La función foldNat es la función de plegado para el tipo Nat

- foldNat :: (a -> a) -> a -> Nat -> a
- foldNat f e Cero = e
- foldNat f e (Suc n) = f (foldNat f e n)

- Estas funciones de plegado también son conocidas como catamorfismos, o recursores.

- esPPar = foldNat not True
- `Main> esPPar (Suc (Suc (Suc (Cero))))`
- `False`

- (+>>) m =foldNat Suc m

- `Main> uno +>> dos`
- `Suc (Suc (Suc Cero))`

Tipos Polimórficos o Parametrizados

- Algunos tipos predefinidos son polimórficos. El programador también puede definir esta clase de tipos.
- data Par a = UnPar a a deriving Show

```
Main> UnPar 3 7
UnPar 3 7
Main> :t UnPar 3 7
UnPar 3 7 :: Num a => Par a
Main> :t UnPar True False
UnPar True False :: Par Bool
```

El tipo polimórfico predefinido Either

- data Either a b = Left a | Right b
  deriving (Eq , Ord , Read , Show)

```
listaMixta1 :: [Either Char Int]
listaMixta1 = [Left 'a' ,Left 'c' , Right 3, Left 'g', Right 3]

listaMixta2 :: [Either String Bool]
listaMixta2 = [Left “computación" , Right True , Left
   "discretas", Right  False]
```

El tipo polimórfico predefinido Either

- listaMixta3 :: [Either Int Int]
- listaMixta3 = [Left 5 , Right 5 , Left 8, Right 6]

- listaPoli :: [Either Bool a]
- listaPoli = [Left False, Left True, Left True]

El tipo polimórfico predefinido Maybe

- Se usa como resultado de funciones parcialmente definidas

- Data Maybe a = Nothing | Just a
  deriving (Eq , Ord , Read , Show)

  La idea es definir una función parcial con resultado de tipo a como una función totalmente definida con resultado de tipo Maybe a que devuelva Nothing cuando la función original no este definida y Just x cuando el resultado este definido y sea x.

El tipo polimórfico predefinido Maybe

- reciproco :: Float -> Maybe Float
- reciproco 0 = Nothing
- reciproco x = Just (1/x)

- `Main> reciproco 2`
- `Just 0.5`
- `Main> reciproco 0`
- `Nothing`
- Maybe evita el uso de la función error y obtiene un resultado incluso cuando la función definida no tenga sentido

- type Nombre = String
- type Telefono = Integer
- data Agenda = Ag [(Nombre, Telefono)]

- miAgenda :: Agenda
- miAgenda = Ag [("Luis" ,9335588),("Maria", 9566669),("Carlia",98876664)]

- ageLista :: Agenda ->[(Nombre,Telefono)]
- ageLista (Ag xs) =xs

- buscar :: Nombre -> Maybe Telefono
- buscar nom

```
| [y| (x,y) <-  (ageLista miAgenda), x== nom]==[] = Nothing
|otherwise  = Just (head [y| (x,y) <-  (ageLista miAgenda),
   x== nom
```

```
Main> buscar  "Luis"
Just 9335588
Main> buscar  "blas"
Nothing
```