

# **construyendo tipos de datos con containers**

---

Eugenia Simich

20 de octubre de 2016

XIV JCC



$$S \mathrel{\lhd} P = \Sigma [s \in S] \, P_s$$

# vistazo previo

1. motivación: programación genérica

algunos programas genéricos

universos

2. constructores de tipos

funtores

3. containers

4. morfismos de containers

esta charla se trata de

- programación genérica
- en Agda
- (o en cualquier lenguaje con tipos dependientes)

¿y qué es...

- ... la programación genérica?
- ... Agda?
- ... los tipos dependientes?

## **motivación: programación genérica**

---

# algunos tipos de datos

## booleanos

```
data Bool : Set where
  true  : Bool
  false : Bool
```

# algunos tipos de datos

## booleanos

```
data Bool : Set where
  true  : Bool
  false : Bool
```

## naturales

```
data Nat : Set where
  zero  : Nat
  suc   : Nat → Nat
```

## algunos programas

- programemos la función identidad...

## algunos programas

- programemos la función identidad...

**...de booleanos**

`idbool : Bool → Bool`

`idbool x = x`

# algunos programas

- programemos la función identidad...

**...de booleanos**

`idbool : Bool → Bool`

`idbool x = x`

**...y de naturales**

`idnat : Nat → Nat`

`idnat x = x`

# algunos programas

- programemos la función identidad...

...de booleanos

`idbool : Bool → Bool`

`idbool x = x`

...y de naturales

`idnat : Nat → Nat`

`idnat x = x`

- ¿podremos construir una función identidad que *funcione* en ambos casos?

# algunos programas

- programemos la función identidad...

...de booleanos

`idbool : Bool → Bool`

`idbool x = x`

...y de naturales

`idnat : Nat → Nat`

`idnat x = x`

- ¿podremos construir una función identidad que *funcione* en ambos casos?

¡simple!

`id : ∀{A : Set} → A → A`

`id x = x`

## algunos programas

---

- programemos ahora una función de comparación por igualdad...

- programemos ahora una función de comparación por igualdad...

**...para booleanos**

eqbool : Bool → Bool → Bool

eqbool true true = true

eqbool false false = true

eqbool \_ \_ = false

# algunos programas

- programemos ahora una función de comparación por igualdad...

...para booleanos

```
eqbool : Bool → Bool → Bool  
eqbool true true  = true  
eqbool false false = true  
eqbool _ _        = false
```

...y para naturales

```
eqnat : Nat → Nat → Bool  
eqnat zero zero    = true  
eqnat zero (suc _) = false  
eqnat (suc _) zero = false  
eqnat (suc x) (suc y) = eqnat x y
```

# algunos programas

- programemos ahora una función de comparación por igualdad...

...para booleanos

```
eqbool : Bool → Bool → Bool  
eqbool true true  = true  
eqbool false false = true  
eqbool _ _        = false
```

...y para naturales

```
eqnat : Nat → Nat → Bool  
eqnat zero zero    = true  
eqnat zero (suc _) = false  
eqnat (suc _) zero = false  
eqnat (suc x) (suc y) = eqnat x y
```

- ¿podremos construir una función de equivalencia **genérica**?  
es decir, ¿podremos construir una función `eq : A → A → Bool`?  
(al menos para  $A$  reemplazable por `Bool`, `Nat`)

# algunos programas

- programemos ahora una función de comparación por igualdad...

...para booleanos

```
eqbool : Bool → Bool → Bool  
eqbool true true  = true  
eqbool false false = true  
eqbool _ _        = false
```

...y para naturales

```
eqnat : Nat → Nat → Bool  
eqnat zero zero    = true  
eqnat zero (suc _) = false  
eqnat (suc _) zero = false  
eqnat (suc x) (suc y) = eqnat x y
```

- ¿podremos construir una función de equivalencia **genérica**?  
es decir, ¿podremos construir una función `eq : A → A → Bool`?  
(al menos para  $A$  reemplazable por `Bool`, `Nat`)
- ¿qué ganaríamos con hacerlo?

## programando genéricamente

---

en Agda se puede, con un *truco*: universos

# programando genéricamente

en Agda se puede, con un *truco*: universos

1. comenzemos definiendo un tipo Name:

```
data Name : Set where
  nat  : Name
  bool : Name
```

# programando genéricamente

en Agda se puede, con un *truco*: universos

1. comenzemos definiendo un tipo Name:

```
data Name : Set where
  nat   : Name
  bool  : Name
```

2. y una función:

```
[_] : Name → Set
[_ nat ] = Nat
[_ bool ] = Bool
```

# programando genéricamente

en Agda se puede, con un *truco*: universos

1. comenzemos definiendo un tipo Name:

```
data Name : Set where
  nat  : Name
  bool : Name
```

2. y una función:

```
[_] : Name → Set
[_ nat] = Nat
[_ bool] = Bool
```

- ahora podemos definir:

```
eq : (c : Name) → [_ c] → [_ c] → Bool
```

# programando genéricamente

en Agda se puede, con un *truco*: universos

1. comenzemos definiendo un tipo Name:

```
data Name : Set where
  nat  : Name
  bool : Name
```

2. y una función:

```
[_] : Name → Set
[_ nat] = Nat
[_ bool] = Bool
```

- ahora podemos definir:

```
eq : (c : Name) → [_ c] → [_ c] → Bool
eq nat =
eq bool =
```

# programando genéricamente

en Agda se puede, con un *truco*: universos

1. comenzemos definiendo un tipo Name:

```
data Name : Set where
  nat  : Name
  bool : Name
```

2. y una función:

```
[_] : Name → Set
[_ nat] = Nat
[_ bool] = Bool
```

- ahora podemos definir:

```
eq : (c : Name) → [_ c] → [_ c] → Bool
eq nat = eqnat
eq bool = eqbool
```



Ceci n'est pas une pipe.

# receta para construir universos

---

1. definir un tipo de *códigos sintácticos*, por ejemplo, U:

```
data U : Set where
  nat   : U
  bool  : U
  pair  : U → U → U
```

1. definir un tipo de *códigos sintácticos*, por ejemplo, U:

```
data U : Set where
  nat   : U
  bool  : U
  pair  : U → U → U
```

2. y una función de extensión:

```
[_] : U → Set
[_ nat]      = Nat
[_ bool]     = Bool
[_ pair t1 t2] = [_ t1] × [_ t2]
```

# receta para construir universos

1. definir un tipo de *códigos sintácticos*, por ejemplo, U:

```
data U : Set where
  nat   : U
  bool  : U
  pair  : U → U → U
```

2. y una función de *extensión*:

```
[_] : U → Set
[_ nat]      = Nat
[_ bool]     = Bool
[_ pair t1 t2] = [_ t1] × [_ t2]
```

3. descansar, acabamos de crear un universo

## constructores de tipos

---

## algunos constructores de tipos de datos

listas

data List ( $X : \text{Set}$ ) : Set where

nil : List  $X$

cons :  $X \rightarrow \text{List } X \rightarrow \text{List } X$

## algunos constructores de tipos de datos

**listas**

```
data List (X : Set) : Set where
    nil      : List X
    cons   : X → List X → List X
```

**maybe**

```
data Maybe (X : Set) : Set where
    nothing  : Maybe X
    just     : X → Maybe X
```

## algunos constructores de tipos de datos

**listas**

```
data List (X : Set) : Set where
    nil      : List X
    cons   : X → List X → List X
```

**maybe**

```
data Maybe (X : Set) : Set where
    nothing  : Maybe X
    just     : X → Maybe X
```

**árboles**

```
data Tree (X : Set) : Set where
    leaf    : X → Tree X
    node   : Tree X → Tree X → Tree X
```

## algunos constructores de tipos de datos

**listas**

```
data List (X : Set) : Set where
    nil      : List X
    cons   : X → List X → List X
```

**maybe**

```
data Maybe (X : Set) : Set where
    nothing  : Maybe X
    just     : X → Maybe X
```

**árboles**

```
data Tree (X : Set) : Set where
    leaf    : X → Tree X
    node   : Tree X → Tree X → Tree X
```

**streams**

```
data Stream (A : Set) : Set where
    cons  : A → Stream A → Stream A
```

listas infinitas

# map

- programemos funciones de mapeo...

...para listas

`maplist :  $\forall\{A\ B\} \rightarrow (A \rightarrow B) \rightarrow \text{List } A \rightarrow \text{List } B$`

`maplist f nil = nil`

`maplist f (cons x l) = cons (fx) (maplist f l)`

`map f [a, b, c] = [fa, fb, fc]`

# map

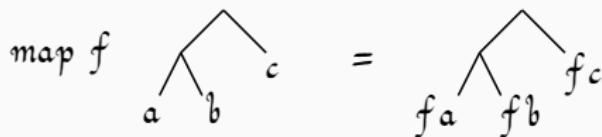
- programemos funciones de mapeo...

...para árboles

`maptree : ∀{A B} → (A → B) → Tree A → Tree B`

`maptree f (leaf x) = leaf (fx)`

`maptree f (node l r) = node (maptree f l) (maptree f r)`



# map

- programemos funciones de mapeo...

...para streams

`mapstream` :  $\forall\{A\ B\} \rightarrow (A \rightarrow B) \rightarrow \text{Stream } A \rightarrow \text{Stream } B$

`mapstream f (cons x s) = cons (fx) (mapstream fs)`

$$\text{map } f(\alpha_1, \alpha_2, \dots) = (f\alpha_1, f\alpha_2, \dots)$$

# map

- programemos funciones de mapeo...

...para maybe

`mapmaybe : ∀{A B} → (A → B) → Maybe A → Maybe B`

`mapmaybe f nothing = nothing`

`mapmaybe f (just x) = just (f x)`

$$\text{map } f (\text{just } x) = \text{just } (f x)$$

- las listas, los árboles, los streams, entre otros constructores de tipos son **mapeables**
- llamaremos **funtores** a estos constructores  
(deberán preservar identidades y composiciones)
- ¿podremos construir una **map** genérica?

## **containers**

---

los **containers** son una representación alternativa de ciertos tipos de datos

¿qué tipos de datos?

aquellos que *contienen* otros datos

y pueden pensarse como **formas con posiciones**

veamos algunos ejemplos...

## ejemplo: listas

---

- las listas tienen las siguientes formas:

[ ] [ O ] [ O , O ] [ O , O , O ]

## ejemplo: listas

- las listas tienen las siguientes formas:

[ ] [ O ] [ O , O ] [ O , O , O ]

*la forma de una lista  
es su longitud*

## ejemplo: listas

- las listas tienen las siguientes formas:

[ ] [ O ] [ O , O ] [ O , O , O ]

*la forma de una lista  
es su longitud*

- y cada forma tiene un conjunto de posiciones a ser llenadas con datos:

[ ] [ O<sub>0</sub> ] [ O<sub>0</sub>, O<sub>1</sub> ] [ O<sub>0</sub>, O<sub>1</sub>, O<sub>2</sub> ]

## ejemplo: listas

- las listas tienen las siguientes formas:

[ ] [ O ] [ O , O ] [ O , O , O ]

*la forma de una lista  
es su longitud*

- y cada forma tiene un conjunto de posiciones a ser llenadas con datos:

[ ] [ O<sub>0</sub> ] [ O<sub>0</sub>, O<sub>1</sub> ] [ O<sub>0</sub>, O<sub>1</sub>, O<sub>2</sub> ]

*una lista de tamaño n  
tiene n posiciones*

## ejemplo: listas

- las listas tienen las siguientes formas:

[ ] [ O ] [ O , O ] [ O , O , O ]

*la forma de una lista  
es su longitud*

- y cada forma tiene un conjunto de posiciones a ser llenadas con datos:

[ ] [ O<sub>0</sub> ] [ O<sub>0</sub>, O<sub>1</sub> ] [ O<sub>0</sub>, O<sub>1</sub>, O<sub>2</sub> ]

*una lista de tamaño n  
tiene n posiciones*

- ▶ el tipo **Nat** representa de forma unívoca a las formas de listas
- ▶ dada una forma  $n$  (una longitud  $n$ ), el conjunto de posiciones es  $\{0, \dots, n - 1\}$

## ejemplo: streams

---

- todos los streams tienen la misma forma:

(  $\bigcirc$  ,  $\bigcirc$  ,  $\bigcirc$  , ... )

## ejemplo: streams

---

- todos los streams tienen la misma forma:

$$( \bigcirc , \bigcirc , \bigcirc , \dots )$$

- y  $\aleph_0$  posiciones a ser llenadas con datos:

$$( \bigcirc_0, \bigcirc_1, \bigcirc_2, \dots )$$

## ejemplo: streams

- todos los streams tienen la misma forma:

( ○ , ○ , ○ , ... )

- y  $\aleph_0$  posiciones a ser llenadas con datos:

( ○<sub>0</sub>, ○<sub>1</sub>, ○<sub>2</sub>, ... )

- ▶ el tipo **Unit** (con un único habitante) representa el conjunto de formas de streams:

**data Unit : Set where**

**tt : Unit**

- ▶ el conjunto de posiciones es **Nat**  
(para toda forma)

## ejemplo: maybe

---

- un habitante de maybe puede tener dos formas:

nothing    just ○

## ejemplo: maybe

---

- un habitante de maybe puede tener dos formas:

nothing    just ○

- la forma “nothing” no tiene posiciones
- la forma “just” tiene una posición

## ejemplo: maybe

- un habitante de maybe puede tener dos formas:

nothing    just ○

- la forma “nothing” no tiene posiciones
  - la forma “just” tiene una posición
- 
- ▶ el tipo **Bool** (con dos habitantes) puede representar al conjunto de formas del container maybe
  - ▶ los tipos **Ø** (conjunto vacío) y **Unit** representan las posiciones, para cada forma

- representación alternativa de (algunos) constructores de tipos (como las listas, los árboles, etc)
- segregá **estructura** de **contenido**
- las listas, los árboles, los streams, pueden ser vistos como **contenedores** de otros datos

- representación alternativa de (algunos) constructores de tipos (como las listas, los árboles, etc)
- segregá **estructura de contenido**
- las listas, los árboles, los streams, pueden ser vistos como **contenedores** de otros datos

## definición

un **container** es un conjunto de formas, y por cada forma un conjunto de posiciones

si  $S$  es un conjunto de formas y  $P_x$  el conjunto de posiciones dentro de la forma  $x \in S$ :

$$S \triangleleft P$$

es un container

## retomando los ejemplos

---

un container es un par  $S \triangleleft P$  donde

- $S$  es un conjunto de formas
- $P_s$  es una familia de conjuntos de posiciones (un conjunto por cada forma)

## retomando los ejemplos

un container es un par  $S \triangleleft P$  donde

- $S$  es un conjunto de formas
- $P_s$  es una familia de conjuntos de posiciones (un conjunto por cada forma)

### listas

cList : Cont

cList = Nat  $\triangleleft (\lambda n \rightarrow \text{Fin } n)$

$\text{Fin } n = \{0, \dots, n-1\}$

## retomando los ejemplos

un container es un par  $S \triangleleft P$  donde

- $S$  es un conjunto de formas
- $P_s$  es una familia de conjuntos de posiciones (un conjunto por cada forma)

### listas

cList : Cont

cList = Nat  $\triangleleft (\lambda n \rightarrow \text{Fin } n)$

$\text{Fin } n = \{0, \dots, n-1\}$

### stream

cStream : Cont

cStream = Unit  $\triangleleft (\lambda \{\_ \rightarrow \text{Nat}\})$

## retomando los ejemplos

un container es un par  $S \triangleleft P$  donde

- $S$  es un conjunto de formas
- $P_s$  es una familia de conjuntos de posiciones (un conjunto por cada forma)

### listas

cList : Cont

cList = Nat  $\triangleleft (\lambda n \rightarrow \text{Fin } n)$

$\text{Fin } n = \{0, \dots, n-1\}$

### stream

cStream : Cont

cStream = Unit  $\triangleleft (\lambda \{\_ \rightarrow \text{Nat}\})$

### maybe

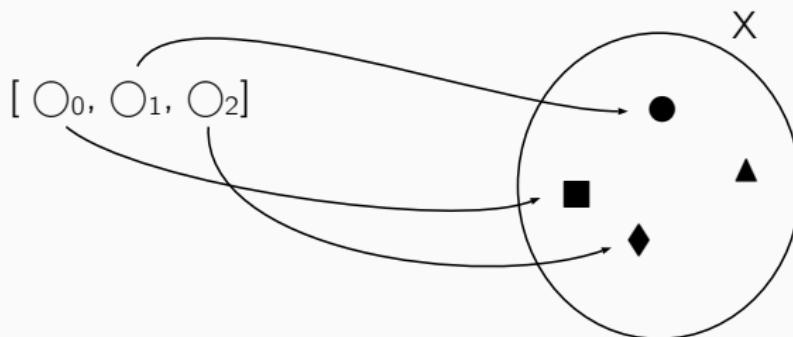
cMaybe : Cont

cMaybe = Bool  $\triangleleft (\lambda \{ \text{false} \rightarrow \emptyset ; \text{true} \rightarrow \text{Unit} \})$

- los containers representan la **estructura** de los tipos de datos  
¿cómo hacer para llenar la estructura con contenido?

- los containers representan la **estructura** de los tipos de datos  
¿cómo hacer para llenar la estructura con contenido?  
**asignando valores a las posiciones**

- los containers representan la **estructura** de los tipos de datos  
¿cómo hacer para llenar la estructura con contenido?  
**asignando valores a las posiciones**
- tomemos por ejemplo la lista  $[\bullet, \square, \diamond]$  de tipo  $X$



## extensión de containers

---

los containers son un universo para representar constructores de tipos de datos, su *extensión* le dará el significado original

la extensión **llena** el container

## extensión de containers

los containers son un universo para representar constructores de tipos de datos, su *extensión* le dará el significado original

la extensión **llena** el container

### definición

Dado un container  $S \triangleleft P$  su **extensión**  $\llbracket S \triangleleft P \rrbracket$  es

- el conjunto de formas  $S$
- por cada forma de  $S$ , una función de posiciones en valores

### extensión de containers

$$\llbracket \_ \rrbracket : \text{Cont} \rightarrow \text{Set} \rightarrow \text{Set}$$

$$\llbracket S \triangleleft P \rrbracket X = \Sigma [ s \in S ] (P s \rightarrow X)$$

$$\llbracket S \triangleleft P \rrbracket X = \{ (s, f) \mid s : S, f : Ps \rightarrow X \}$$

## volviendo a las listas originales

---

- la función de extensión reconstruye el tipo original

$$[\![ \text{cList} ]\!] \cong \text{List}$$

- la función de extensión reconstruye el tipo original

$$[\![ \text{cList} ]\!] \cong \text{List}$$

- ejemplo: la lista  $[\blacksquare, \bullet, \bullet]$  de símbolos se representa:

$I : [\![ \text{cList} ]\!] \text{Symbol}$

$I = 3 , (\lambda \{ \text{zero} \rightarrow \blacksquare ; (\text{suc zero}) \rightarrow \bullet ; (\text{suc} (\text{suc zero})) \rightarrow \bullet ; (\text{suc} (\text{suc} (\text{suc} ())))) \})$

## volviendo a las listas originales

- la función de extensión reconstruye el tipo original

$$[\![ \text{cList} ]\!] \cong \text{List}$$

- o mejor, reconstruimos las funciones:

$$\text{nil} : \forall \{X\} \rightarrow [\![ \text{cList} ]\!] X$$

$$\text{nil} = \text{zero} , (\lambda () )$$

$\lambda()$  es la función vacía

$$\text{cons} : \forall \{X\} \rightarrow X \rightarrow [\![ \text{cList} ]\!] X \rightarrow [\![ \text{cList} ]\!] X$$

$$\begin{aligned}\text{cons } x (n , f) = ( \text{suc } n , (\lambda \{ \text{zero} \rightarrow x \\ ; (\text{suc } i) \rightarrow f i \}) )\end{aligned}$$

## volviendo a las listas originales

- la función de extensión reconstruye el tipo original

$$[\![ \text{cList} ]\!] \cong \text{List}$$

- o mejor, reconstruimos las funciones:

$$\text{nil} : \forall\{X\} \rightarrow [\![ \text{cList} ]\!] X$$

$$\text{nil} = \text{zero} , (\lambda () \quad \curvearrowleft \lambda () \text{ es la función vacía})$$

$$\text{cons} : \forall\{X\} \rightarrow X \rightarrow [\![ \text{cList} ]\!] X \rightarrow [\![ \text{cList} ]\!] X$$

$$\begin{aligned}\text{cons } x (n , f) = ( \text{ suc } n , (\lambda \{ \text{ zero } \rightarrow x \\ ; (\text{ suc } i) \rightarrow f i \}) )\end{aligned}$$

- así la lista [■, ●, ●] es simplemente:

$$l' : [\![ \text{cList} ]\!] \text{Symbol}$$

$$l' = \text{cons } \blacksquare (\text{cons } \bullet (\text{cons } \bullet \text{nil}))$$

algunos containers triviales

- el container identidad representa al funtor identidad:

$$\mathcal{Id} = \{ \circ \}$$

algunos containers triviales

- el container identidad representa al funtor identidad:

cld : Cont

$$\mathcal{Id} = \{ \circ \}$$

cld = Unit  $\triangleleft (\lambda \_ \rightarrow \text{Unit})$

## más ejemplos

algunos containers triviales

- el container identidad representa al funtor identidad:

`cld : Cont`

$$\mathcal{Id} = \{ \circ \}$$

`cld = Unit \triangleleft (\lambda _ \rightarrow Unit)`

entonces

$$[\![ cld ]\!] X \cong X$$

algunos containers triviales

- el container identidad representa al funtor identidad:

`cld : Cont`

$$\text{Id} = \{ \circ \}$$

`cld = Unit ▷ (\lambda _ \rightarrow Unit)`

entonces

$$[\![ \text{cld} ]\!] X \cong X$$

- el container constante ignora el argumento y siempre construye un tipo constante:

$$\mathcal{A} = \{ a_1, a_2, a_3, \dots \}$$

## más ejemplos

algunos containers triviales

- el container identidad representa al funtor identidad:

$$cld : \text{Cont}$$

$$\mathbb{I}d = \{ \circ \}$$

$$cld = \text{Unit} \triangleleft (\lambda \_ \rightarrow \text{Unit})$$

entonces

$$[\![ cld ]\!] X \cong X$$

- el container constante ignora el argumento y siempre construye un tipo constante:

$$cK : \text{Set} \rightarrow \text{Cont}$$

$$\mathcal{A} = \{ a_1, a_2, a_3, \dots \}$$

$$cK A = A \triangleleft (\lambda \_ \rightarrow \emptyset)$$

## más ejemplos

algunos containers triviales

- el container identidad representa al funtor identidad:

$$\text{cld} : \text{Cont}$$

$$\text{Id} = \{ \circ \}$$

$$\text{cld} = \text{Unit} \triangleleft (\lambda \_ \rightarrow \text{Unit})$$

entonces

$$[\![ \text{cld} ]\!] X \cong X$$

- el container constante ignora el argumento y siempre construye un tipo constante:

$$\text{cK} : \text{Set} \rightarrow \text{Cont}$$

$$\mathcal{A} = \{ a_1, a_2, a_3, \dots \}$$

$$\text{cK } A = A \triangleleft (\lambda \_ \rightarrow \emptyset)$$

entonces

$$[\![ \text{cK } A ]\!] X \cong A$$

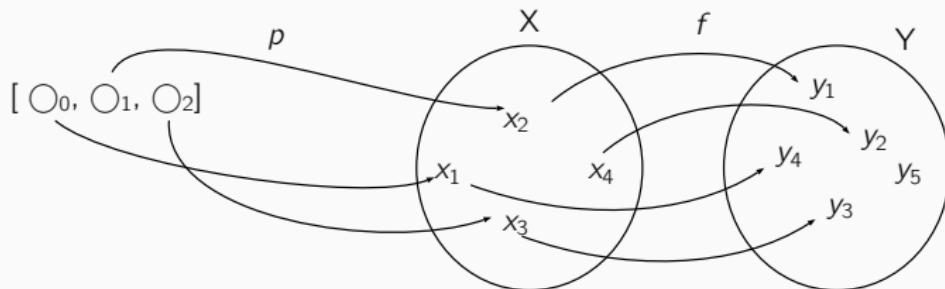
## map de containers

---

- si  $C$  es un container,  $\llbracket C \rrbracket : \text{Set} \rightarrow \text{Set}$
- $\llbracket C \rrbracket$  es nuevamente un constructor de tipos..
- ...pero, ¿es functorial? ¿podremos construir una función `map` de containers?

## map de containers

- si  $C$  es un container,  $\llbracket C \rrbracket : \text{Set} \rightarrow \text{Set}$
- $\llbracket C \rrbracket$  es nuevamente un constructor de tipos..
- ...pero, ¿es functorial? ¿podremos construir una función **map** de containers?
- en el caso de las listas, por ejemplo:



transformamos la lista  $[x_1, x_2, x_3]$  en la lista  $[y_4, y_1, y_3]$  simplemente *componiendo* las funciones

## map de containers

### map

$\text{map} : \forall \{X Y C\} \rightarrow (f : X \rightarrow Y) \rightarrow \llbracket C \rrbracket X \rightarrow \llbracket C \rrbracket Y$

$\text{map } f(s, p) = s, (f \circ p)$

## map de containers

### map

$\text{map} : \forall\{X\ Y\ C\} \rightarrow (f : X \rightarrow Y) \rightarrow [\![C]\!] X \rightarrow [\![C]\!] Y$   
 $\text{map } f(s, p) = s, (f \circ p)$

¡funciona para todo container! ¡obtuvimos nuestra **map** genérica!

podemos construir también:

- producto de containers
- suma de containers (Either de Haskell)
- exponencial de containers (funciones desde una constante)
- composición (para obtener listas de árboles, árboles de streams, etc)

## **morfismos de containers**

---

# morfismos de containers

containers = esqueletos

morfismos = funciones entre esqueletos

- Hay funciones que *hacen cosas con la estructura sin tocar el contenido* (**sólo lo reubican**)
  - decapitar listas
  - aplanar árboles
  - obtener subconjuntos finitos de un stream
- ¿podremos construir un universo para representar estas funciones?

## ejemplo: cabeza de lista

---

¿cómo la expresamos la función `head` de listas en términos de funciones entre los containers `cList` y `cMaybe`?

## ejemplo: cabeza de lista

¿cómo la expresamos la función `head` de listas en términos de funciones entre los containers `cList` y `cMaybe`?

recordemos:

`cList : Cont`

`cList = Nat ▷ (λ n → Fin n)`

`cMaybe : Cont`

`cMaybe = Bool ▷ (λ { false → ∅  
; true → Unit })`

## ejemplo: cabeza de lista

¿cómo la expresamos la función `head` de listas en términos de funciones entre los containers `cList` y `cMaybe`?

recordemos:

`cList : Cont`

`cList = Nat ▷ (λ n → Fin n)`

`cMaybe : Cont`

`cMaybe = Bool ▷ (λ { false → ∅ ; true → Unit })`

la función `head` transforma un `cList` en un `cMaybe`.

`[]` ↢ nothing

`[○, ○, ○]` ↢ just ○



en resumen,

- la forma `zero` se transforma en `false`
- la forma `suc n` se transforma en `true`
- lo que guarda la única posición de la forma `true` proviene de la *primera* posición de la lista

## definición

un morfismo de container entre los containers  $(S_1 \triangleleft P_1)$  y  $(S_2 \triangleleft P_2)$  es un par de funciones  $(f_s, f_p)$  donde:

- $f_s : S_1 \rightarrow S_2$  transforma las formas
- $f_p : (x : S_1) \rightarrow P_2 (f_s x) \rightarrow P_1 x$  reubica las posiciones

notamos  $C \Rightarrow D$  a los morfismos entre los containers  $C$  y  $D$ .

## ejemplo: cabeza de lista (continuación)

para concluir con el ejemplo

### chead

```
chead : cList ⇒ cMaybe
chead = (λ { zero → false ; (suc _) → true }) ,
        (λ { {zero} () ; {suc _} _ → zero })
```

## ejemplo: cabeza de lista (continuación)

para concluir con el ejemplo

### chead

```
chead : cList ⇒ cMaybe  
chead = (λ { zero → false ; (suc _) → true }) ,  
        (λ { {zero} () ; {suc _} _ → zero })
```

también podemos *llenar* los esqueletos de funciones, con la extensión de morfismos  $\llbracket \_ \rrbracket_m$

### head

```
head : ∀{X} → [ cList ] X → [ cMaybe ] X  
head = [ chead ]_m
```

punto de vista categórico: los containers y sus morfismos forman una categoría

- la categoría es bicartesiana cerrada (cuenta con productos, coproductos, exponentiales, inicial y terminal)
- la extensión de un container es un endofunctor en Set
- la extensión de los morfismos, las transformaciones naturales  
*(todas y sólo todas las transf. naturales)*

hemos introducido a los containers y algunas propiedades:

- forma alternativa de representación de (ciertos) tipos de datos
- universo de constructores de tipos
- útiles para la programación genérica
- morfismos de containers como funciones entre esqueletos

además se puede ver que es posible

- representar funtores polinomiales
- representar tipos estrictamente positivos (útiles en la programación total, i.e. garantizando terminación)

algunas extensiones posibles son:

- representar funtores de más de un argumento
- representar funtores indexados (para construir tipos dependientes)

**¡gracias! ¿preguntas?**

**pueden encontrar este (y más) material en**

**<https://www.github.com/eugeniasimich/containers>**

## referencias

- ▶ M. Abbott.  
**Categories of Containers.**  
PhD thesis, University of Leicester, 2003.
- ▶ M. Abbott, T. Altenkirch, and N. Ghani.  
**Containers: Constructing strictly positive types.**  
*Theoretical Computer Science*, 342:3–27, 2005.  
Applied Semantics: Selected Topics.
- ▶ T. Altenkirch, C. McBride, and P. Morris.  
**Generic programming with dependent types.**  
In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007.
- ▶ U. Norell.  
**Dependently typed programming in agda.**  
In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.