



Programación Funcional

Clases teóricas

por Pablo E. “Fidel” Martínez López

5. Tipos algebraicos

“Ged suspiraba a veces, pero no se quejaba. Sabía que en aquella insondable y polvorienta tarea de aprender el nombre verdadero de cada lugar, cada cosa y cada criatura, residía el poder ambicionado, como una gema en el fondo de un pozo seco. Porque en eso consistía la magia, conocer el nombre verdadero de cada cosa.”

Un mago de Terramar
Úrsula K. Le Guin





Motivación de tipos algebraicos

Motivación

- Presentamos un lenguaje y su sistema de tipos
- ¿Cómo representaríamos nuevos problemas?

Motivación

- ❑ Presentamos un lenguaje y su sistema de tipos
- ❑ ¿Cómo representaríamos nuevos problemas?
 - ❑ Solamente tenemos números, tuplas y funciones...
 - ❑ Ej. representar un dominio de helados
 - ❑ ¿Cómo podríamos resolver este problema?

Motivación

- ❏ Ej. representar un dominio de helados
 - ❏ Los helados pueden venir en vasito, cucurucho o pote y llevar helados de varios gustos
 - ❏ Hay restricciones
 - ❏ Los vasitos pueden tener 1 gusto de helado
 - ❏ Los cucuruchos, 2 gustos
 - ❏ Los potses, 3 gustos

Motivación

- Ej. representar un dominio de helados
 -
 - Los componentes se podrían representar con números
 - 1 - vasito, 2 - cucurucho, 3 - pote
 - 1 - chocolate, 2 - dulce de leche, 3 - sambayón, 4 - frutilla, etc.

Motivación

- Ej. representar un dominio de helados
 - Un helado podría representarse con una tupla
(contenedor, (gusto1, gusto2, gusto3))
 - Los componentes se podrían representar con números
 - 1 - vasito, 2 - cucurucho, 3 - pote
 - 1 - chocolate, 2 - dulce de leche, 3 - sambayón, 4 - frutilla, etc.
 - 0 - vacío (o nada)

Motivación

- ¿Qué representarían los siguientes elementos?
 - $(1, (2, 0, 0))$
 - $(2, (1, 3, 0))$
 - $(3, (1, 1, 1))$

Motivación

❑ ¿Qué representarían los siguientes elementos?

❑ $(1, (2, 0, 0))$

❑ $(2, (1, 3, 0))$

❑ $(3, (1, 1, 1))$

❑ ¿Y estos otros?

❑ $(0, (0, 0, 0))$

❑ $(42, (17, 13, 99))$

❑ $(1, (1, 1, 1))$

Motivación

❑ ¿Qué representarían los siguientes elementos?

❑ $(1, (2, 0, 0))$

❑ $(2, (1, 3, 0))$

❑ $(3, (1, 1, 1))$

❑ ¿Y estos otros?

❑ $(0, (0, 0, 0))$

❑ $(42, (17, 13, 99))$

❑ $(1, (1, 1, 1))$

❑ ¿Cómo distinguir elementos correctos de incorrectos?

Motivación

■ Problemas

- ¿Cómo se podría saber qué describe esta expresión?

(2, (1, 3, 0))

Motivación

■ Problemas

■ ¿Cómo se podría saber qué describe esta expresión?

$(2, (1, 3, 0))$

■ Y suponiendo que son helados, y que describe a mi helado favorito, ¿cómo saber cuál es?

Motivación

- ❑ Problemas
 - ❑ ¿Cómo se podría saber qué describe esta expresión?
 $(2, (1, 3, 0))$
 - ❑ Y suponiendo que son helados, y que describe a mi helado favorito, ¿cómo saber cuál es?
- ❑ Esta representación NO es suficientemente descriptiva

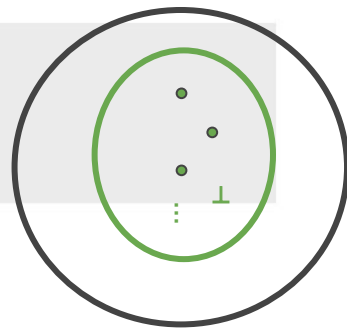
Motivación

- ❑ Problemas
 - ❑ ¿Cómo se podría saber qué describe esta expresión?
 $(2, (1, 3, 0))$
 - ❑ Y suponiendo que son helados, y que describe a mi helado favorito, ¿cómo saber cuál es?
- ❑ Esta representación NO es suficientemente descriptiva
 - ❑ Contradice la idea de denotación que se presentó
 - ❑ Un buen programa *debería comunicar intención*

Motivación

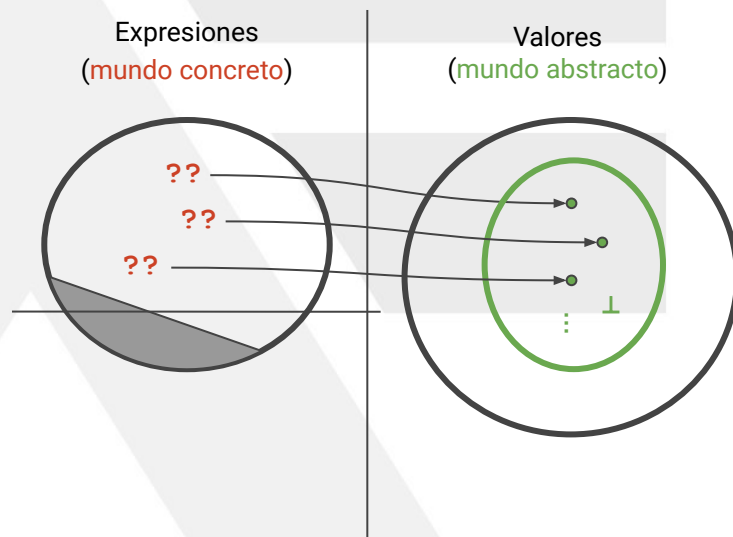
- ❏ ¿Qué queremos describir?
- ❏ Un conjunto de datos

Valores
(mundo abstracto)



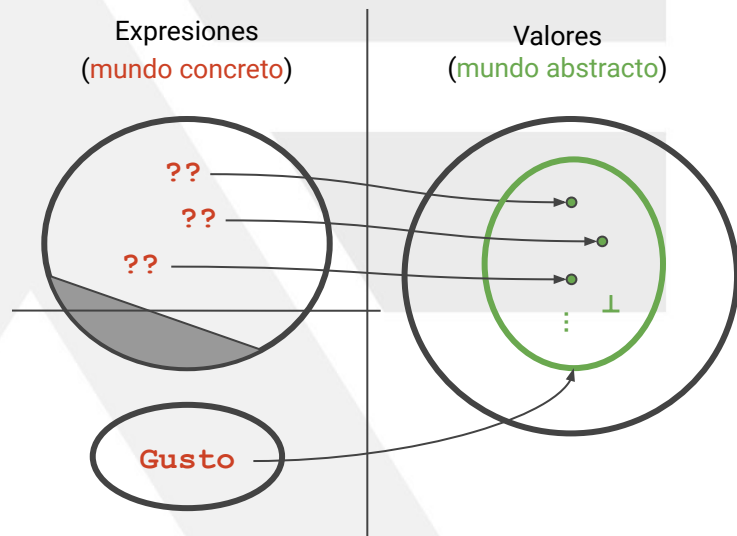
Motivación

- ❑ ¿Qué queremos describir?
 - ❑ Un conjunto de datos
- ❑ ¿Qué precisamos?
 - ❑ Expresiones
 - ❑ ¡Sintaxis!



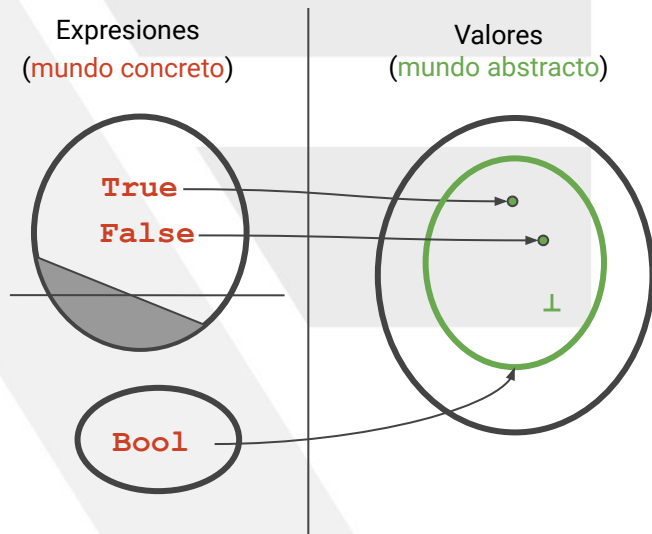
Motivación

- ❑ ¿Qué queremos describir?
 - ❑ Un conjunto de datos
- ❑ ¿Qué precisamos?
 - ❑ Expresiones
 - ❑ ¡Sintaxis!
 - ❑ Pero de tipos “básicos” nuevos...



Motivación

- ❑ ¿Cómo describimos los valores de verdad?
- ❑ **Bool, True y False**
- ❑ ¿Qué son **True** y **False**?



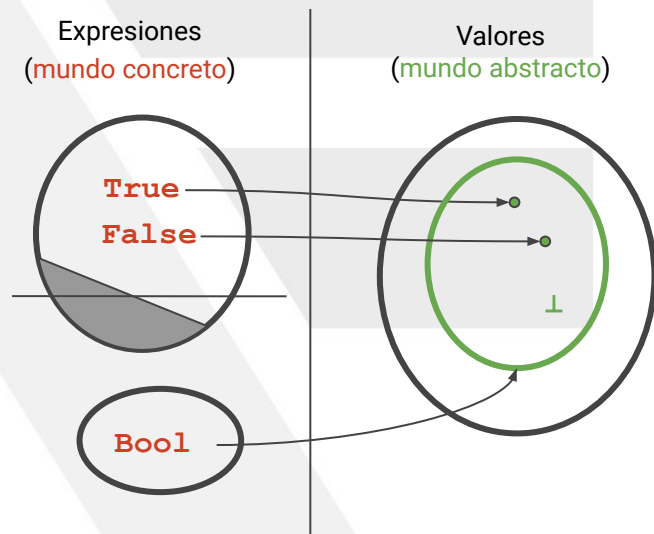
Motivación

❑ ¿Cómo describimos los valores de verdad?

❑ **Bool, True y False**

❑ ¿Qué son **True** y **False**?

❑ Expresiones atómicas



Motivación

❑ ¿Cómo describimos los valores de verdad?

❑ **Bool, True y False**

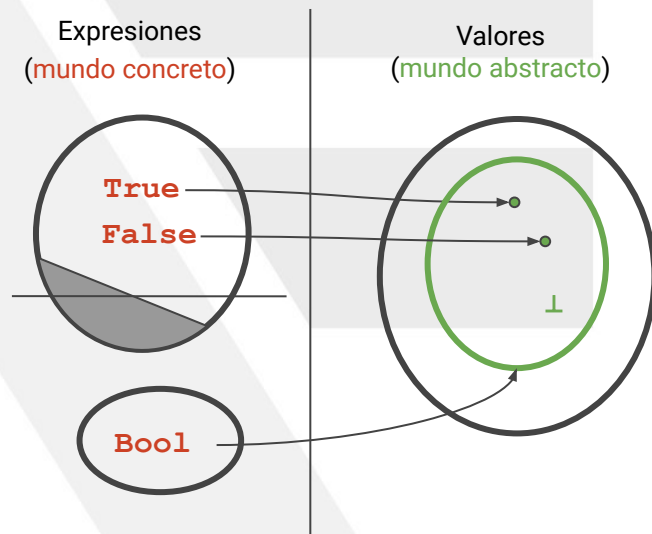
❑ ¿Qué son **True** y **False**?

❑ Expresiones atómicas

❑ ¿Y para un tipo nuevo?

❑ **Helado**

❑ ¿Y los datos?



Motivación

- Necesitamos una forma de definir nuevos tipos

Motivación

- ❑ Necesitamos una forma de definir nuevos tipos
 - ❑ Poder darle nombre al tipo
 - ❑ Poder dar expresiones atómicas para sus elementos

Motivación

- ❑ Necesitamos una forma de definir nuevos tipos
 - ❑ Poder darle nombre al tipo
 - ❑ Poder dar expresiones atómicas para sus elementos
 - ❑ Todo esto con sintaxis que mantenga las propiedades y sin perder la inferencia de tipos

Motivación

- ❑ Necesitamos una forma de definir nuevos tipos
 - ❑ Poder darle nombre al tipo
 - ❑ Poder dar expresiones atómicas para sus elementos
 - ❑ Todo esto con sintaxis que mantenga las propiedades y sin perder la inferencia de tipos
 - ❑ El sistema de tipos debe extenderse a demanda



Tipos algebraicos

Tipos algebraicos

- ❑ Veremos cómo hacer todo esto en Haskell
 - ❑ Como expresiones atómicas podemos usar identificadores con mayúsculas (como **True** y **False**)

Tipos algebraicos

- ▣ Veremos cómo hacer todo esto en Haskell
 - ▣ Como expresiones atómicas podemos usar identificadores con mayúsculas (como **True** y **False**)
 - ▣ Se denominarán ***constructores***

Tipos algebraicos

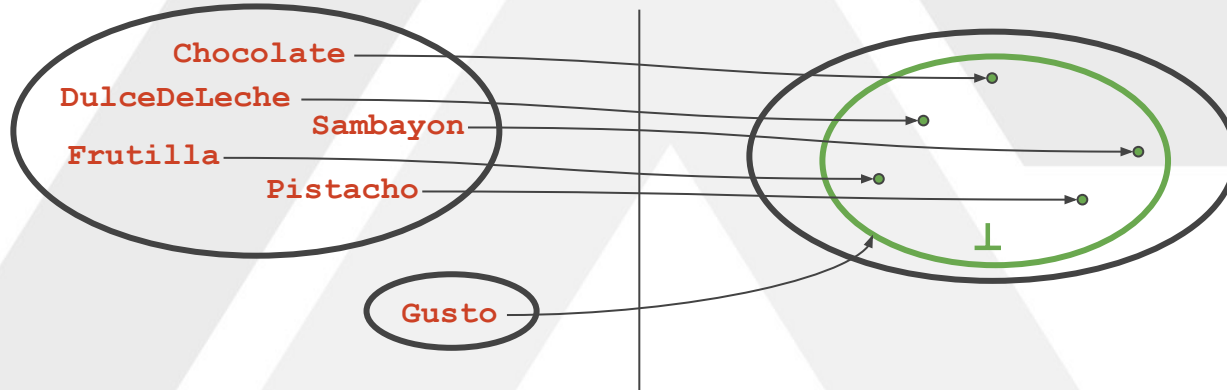
- ❑ Veremos cómo hacer todo esto en Haskell
 - ❑ Como expresiones atómicas podemos usar identificadores con mayúsculas (como **True** y **False**)
 - ❑ Se denominarán ***constructores***
 - ❑ ¿Cómo vincular estos identificadores con el tipo nuevo?

Tipos algebraicos

- ❑ Veremos cómo hacer todo esto en Haskell
 - ❑ Como expresiones atómicas podemos usar identificadores con mayúsculas (como **True** y **False**)
 - ❑ Se denominarán ***constructores***
 - ❑ ¿Cómo vincular estos identificadores con el tipo nuevo?
 - ❑ Hace falta una ***definición***
 - ❑ Utilizaremos para esto la palabra reservada **data**

Tipos algebraicos

- Ejemplo: gustos de helado
 - Se busca definir un conjunto de 5 elementos que represente los gustos de helado



Tipos algebraicos

- ❑ Ejemplo: gustos de helado
 - ❑ Una posible definición podría ser en castellano
 - ❑ **Gusto** es un tipo cuyos elementos satisfacen exclusivamente las siguientes propiedades
 - ❑ **Chocolate** es un elemento del tipo **Gusto**
 - ❑ **DulceDeLeche** es un elemento del tipo **Gusto**
 - ❑ **Sambayon** es un elemento del tipo **Gusto**
 - ❑ **Frutilla** es un elemento del tipo **Gusto**
 - ❑ **Pistacho** es un elemento del tipo **Gusto**

Tipos algebraicos

- ❑ Ejemplo: gustos de helado
 - ❑ Una posible definición podría ser en castellano
 - ❑ **Gusto** es un tipo cuyos elementos satisfacen exclusivamente las siguientes propiedades
 - ❑ **Chocolate** :: Gusto
 - ❑ **DulceDeLeche** :: Gusto
 - ❑ **Sambayon** :: Gusto
 - ❑ **Frutilla** :: Gusto
 - ❑ **Pistacho** :: Gusto

Tipos algebraicos

- La sintaxis para expresar esto en Haskell es

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```

- Gusto** es un tipo cuyos elementos satisfacen exclusivamente las siguientes propiedades
 - `Chocolate :: Gusto`
 - `DulceDeLeche :: Gusto`
 - `Sambayon :: Gusto`
 - `Frutilla :: Gusto`
 - `Pistacho :: Gusto`
- Son equivalentes esta definición y esa sintaxis

Tipos algebraicos

❏ Definición de un tipo algebraico

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla   | Pistacho
```

Tipos algebraicos

❑ Definición de un tipo algebraico

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla   | Pistacho
```

❑ La cláusula **data** comienza la definición

❑ Sigue el *nombre* del nuevo tipo

Tipos algebraicos

❑ Definición de un tipo algebraico

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla   | Pistacho
```

- ❑ La cláusula **data** comienza la definición
- ❑ Sigue el *nombre* del nuevo tipo
- ❑ Luego del igual, separados por barras verticales (|) van los constructores

Tipos algebraicos

❑ Definición de un tipo algebraico

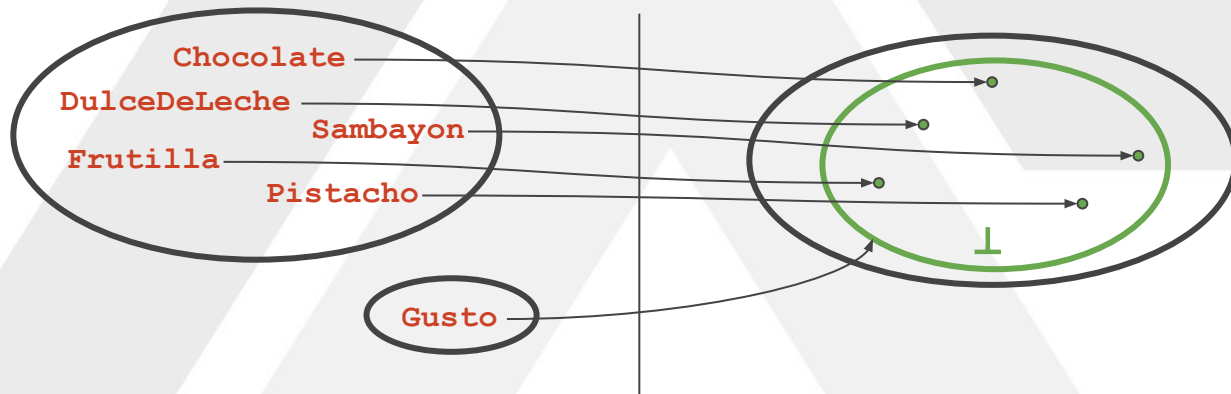
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla   | Pistacho
```

- ❑ La cláusula **data** comienza la definición
- ❑ Sigue el *nombre* del nuevo tipo
- ❑ Luego del igual, separados por barras verticales (|) van los constructores
 - ❑ Van con mayúsculas
 - ❑ Usaremos color rojo para distinguirlos

Tipos algebraicos

❏ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```

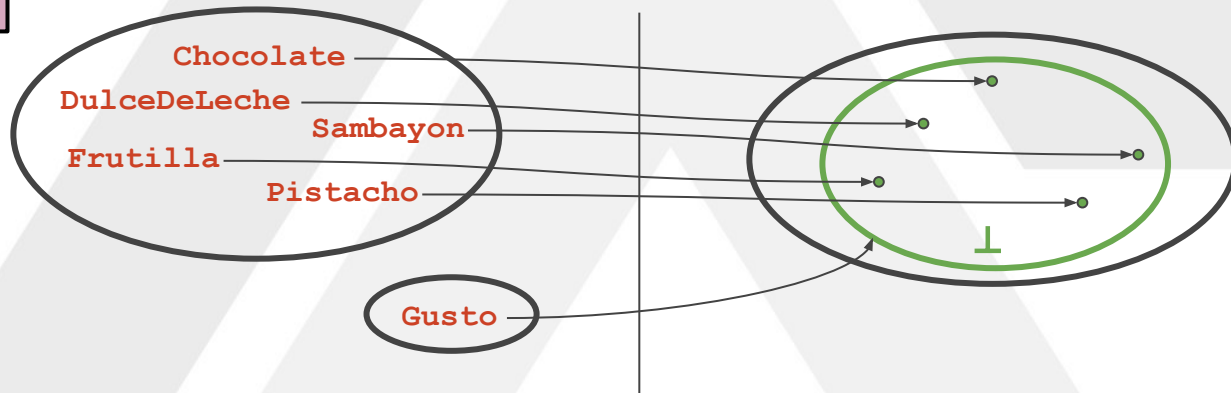


Tipos algebraicos

❏ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```

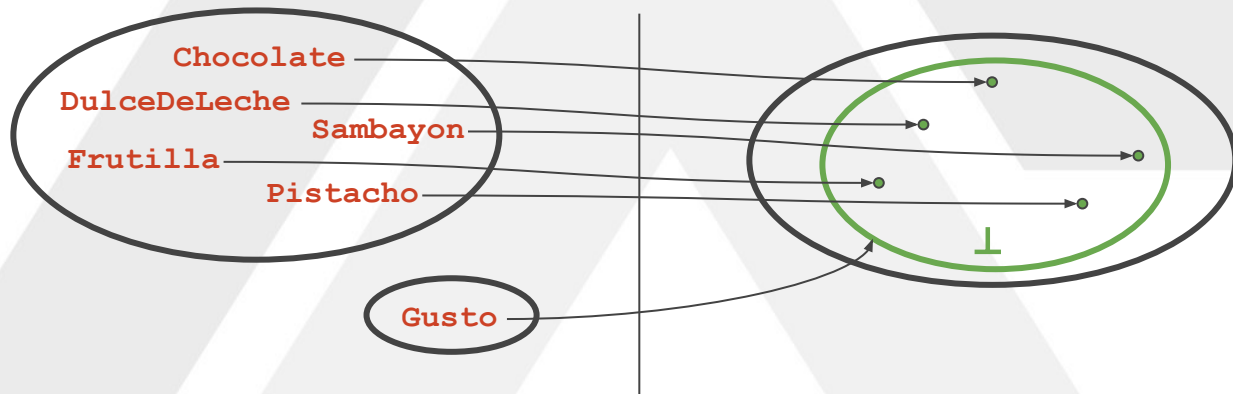
¡Es complejo
dibujar así!



Tipos algebraicos

❏ Ejemplo: gustos de helado

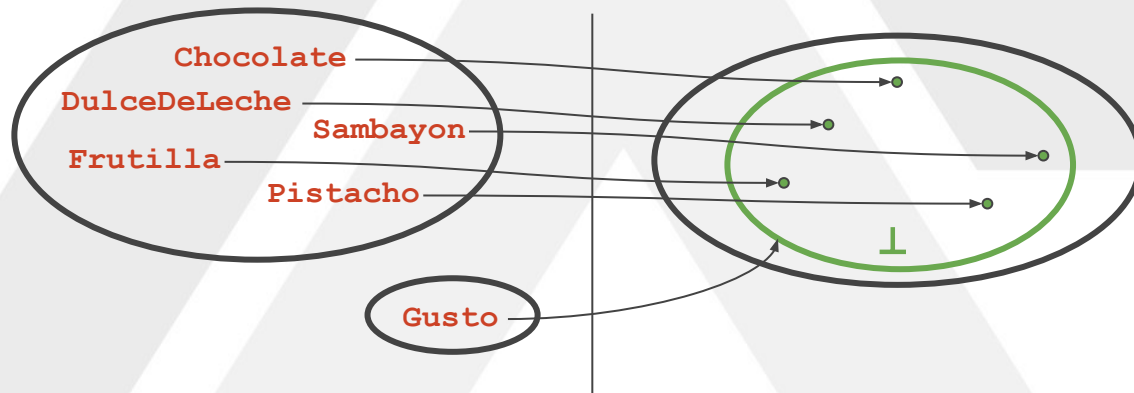
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

□ Ejemplo: gustos de helado

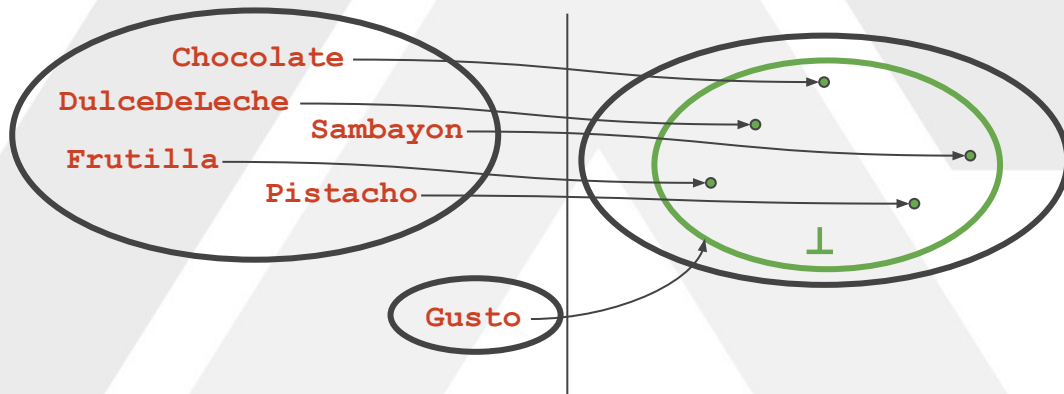
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

□ Ejemplo: gustos de helado

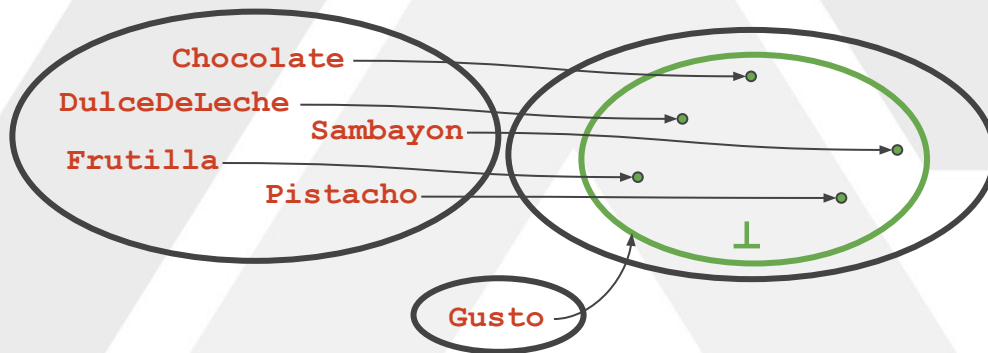
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

□ Ejemplo: gustos de helado

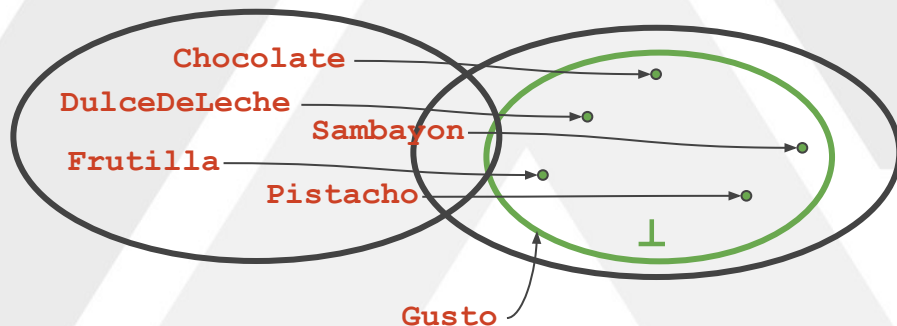
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

❏ Ejemplo: gustos de helado

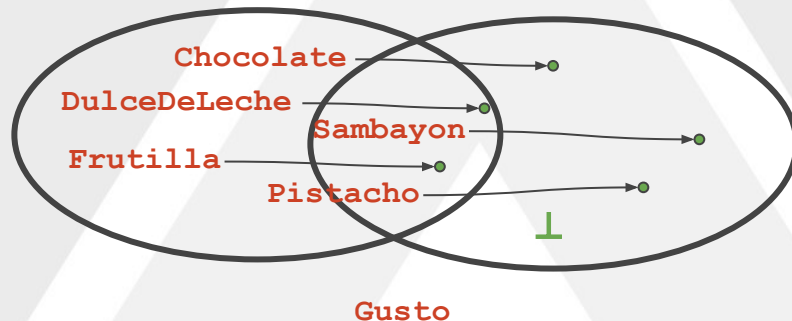
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

❏ Ejemplo: gustos de helado

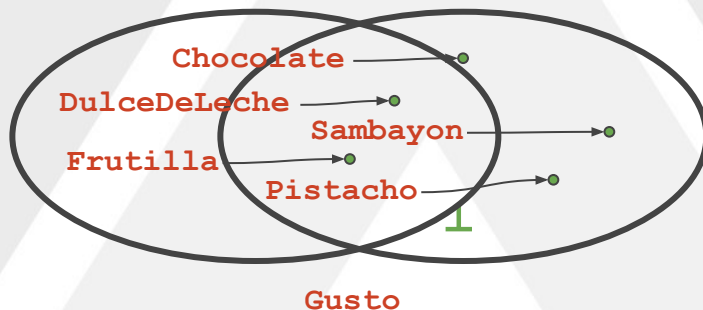
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

□ Ejemplo: gustos de helado

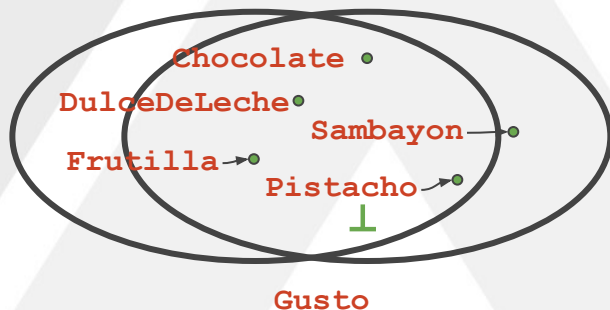
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

❏ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

□ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



Tipos algebraicos

❏ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



¡Por lo que abusaremos de la notación gráfica, así!

Tipos algebraicos

❏ Ejemplo: gustos de helado

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon  | Frutilla  | Pistacho
```



¡Por lo que abusaremos de la notación gráfica, así!

Tipos algebraicos

- ❑ Recordemos el dominio de helados
 - ❑ Los helados pueden venir en vasito, cucurucho o pote y llevar helados de varios gustos
 - ❑ Hay restricciones
 - ❑ Los vasitos pueden tener 1 gusto de helado
 - ❑ Los cucuruchos, 2 gustos
 - ❑ Los potes, 3 gustos

Tipos algebraicos

- ❑ Los helados tienen partes...



Tipos algebraicos

- ❑ Los helados tienen partes...
 - ❑ ... lo que podría modelarse con *muchos* constructores...

VasitoDeChocolate

VasitoDeDulceDeLeche

VasitoDeSambayon

VasitoDeFrutilla

CucuruchoDeChocolateYSambayon

CucuruchoDeChocolateYDulceDeLeche

...

Tipos algebraicos

- ❑ Los helados tienen partes...
 - ❑ ... lo que podría modelarse con *muchos* constructores...
 - ❑ ... ¡o con constructores parametrizados!

Vasito DulceDeLeche

Cucurucho Chocolate Sambayon

Tipos algebraicos

- ❑ Los helados tienen partes...
 - ❑ ... lo que podría modelarse con *muchos* constructores...
 - ❑ ... ¡o con constructores parametrizados!
Vasito DulceDeLeche
Cucurucho Chocolate Sambayon
- ❑ ¡Los constructores pueden ser funciones!

Tipos algebraicos

- ❑ Los helados tienen partes...
 - ❑ ... lo que podría modelarse con *muchos* constructores...
 - ❑ ... ¡o con constructores parametrizados!
Vasito DulceDeLeche
Cucurucho Chocolate Sambayon
- ❑ ¡Los constructores pueden ser funciones!
- ❑ Pero, incluso con argumentos, son expresiones atómicas

Tipos algebraicos

- ❑ Los helados tienen partes...
 - ❑ ... lo que podría modelarse con *muchos* constructores...
 - ❑ ... ¡o con constructores parametrizados!
Vasito DulceDeLeche
Cucurucho Chocolate Sambayon
- ❑ ¡Los constructores pueden ser funciones!
- ❑ Pero, incluso con argumentos, son expresiones atómicas
 - ❑ *No tienen asociada una regla de reducción*, o sea, NO forman redexes (comienzan con mayúsculas para indicar eso)

Tipos algebraicos

- ❏ Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

Tipos algebraicos

❑ Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

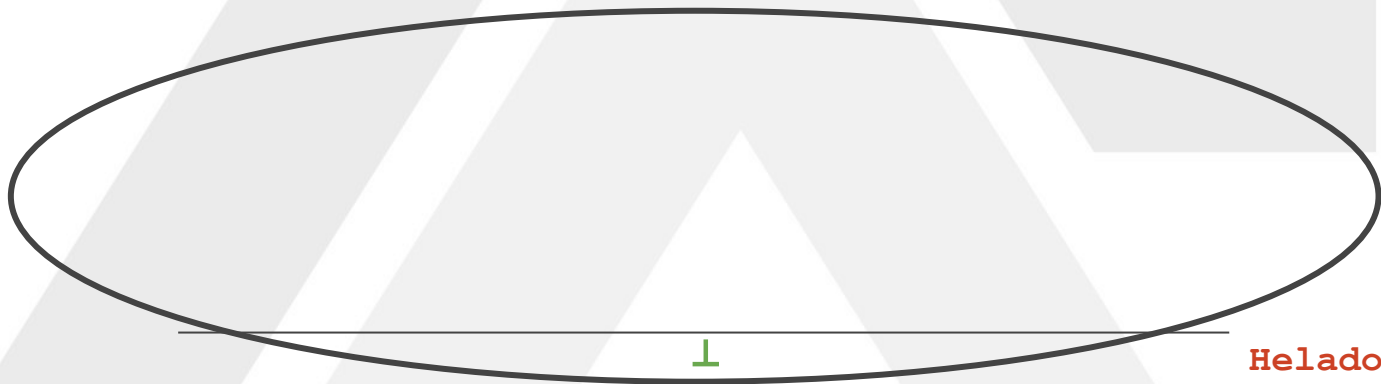
❑ **Helado** es un tipo cuyos elementos satisfacen exclusivamente las siguientes propiedades

- ❑ si `g :: Gusto`
entonces `Vasito g :: Helado`
- ❑ si `g1,g2 :: Gusto`
entonces `Cucurucho g1 g2 :: Helado`
- ❑ si `g1,g2,g3 :: Gusto`
entonces `Pote g1 g2 g3 :: Helado`

Tipos algebraicos

- Declaración con argumentos en los constructores

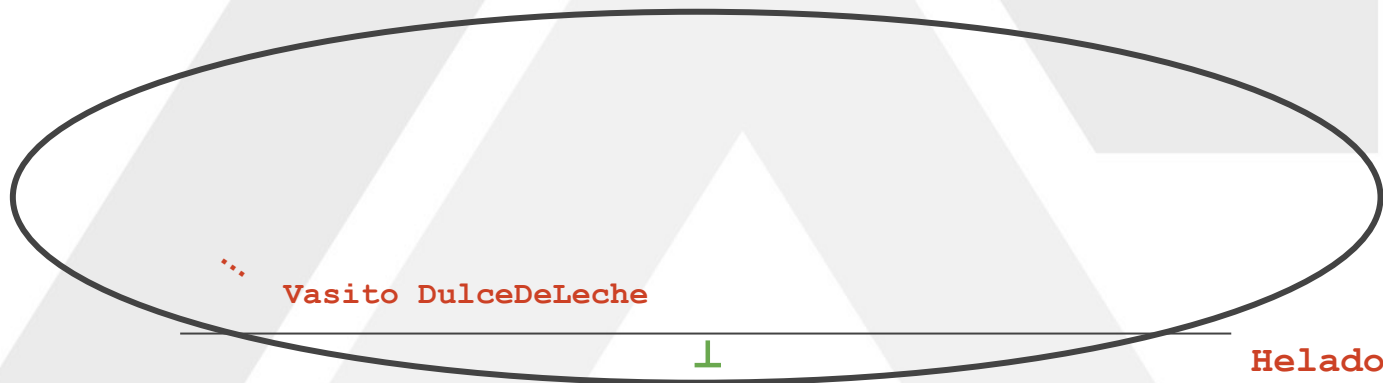
```
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

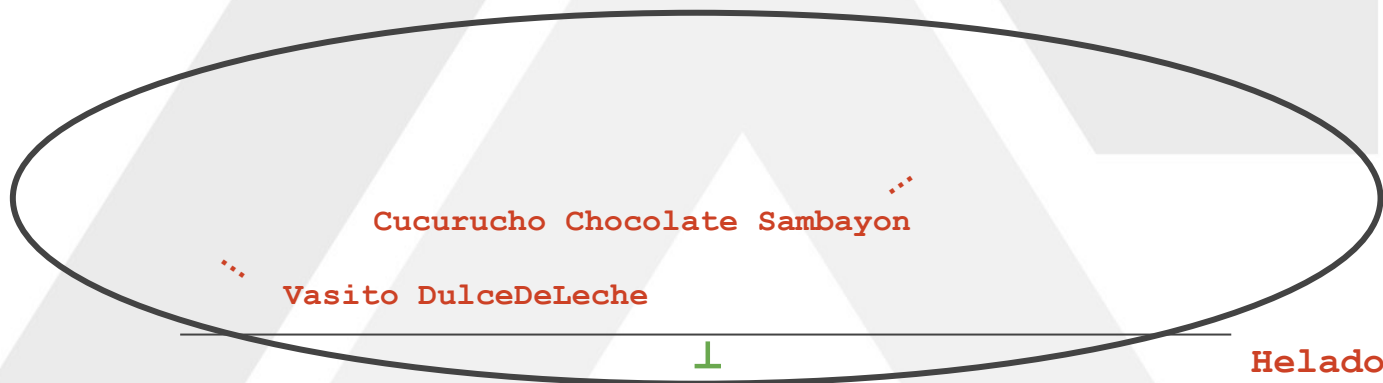
```
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

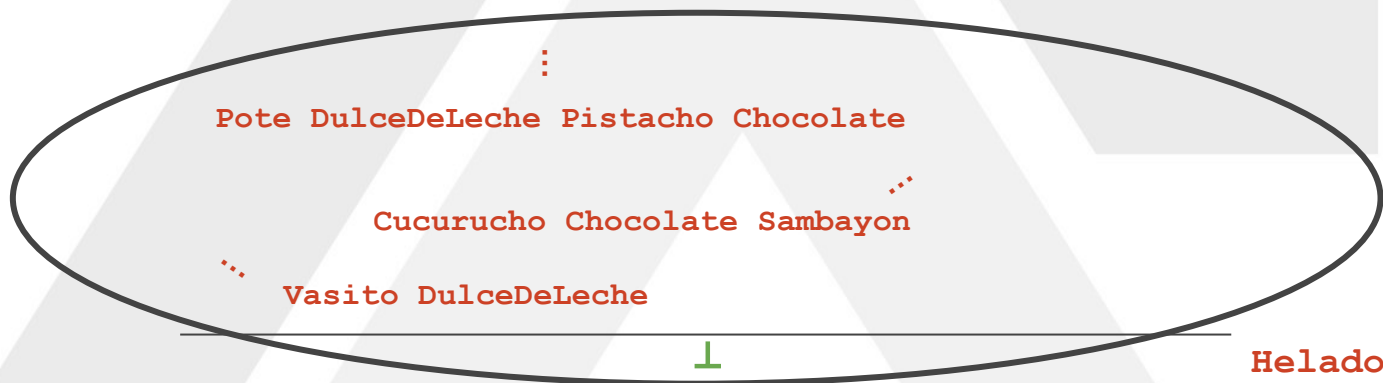
```
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

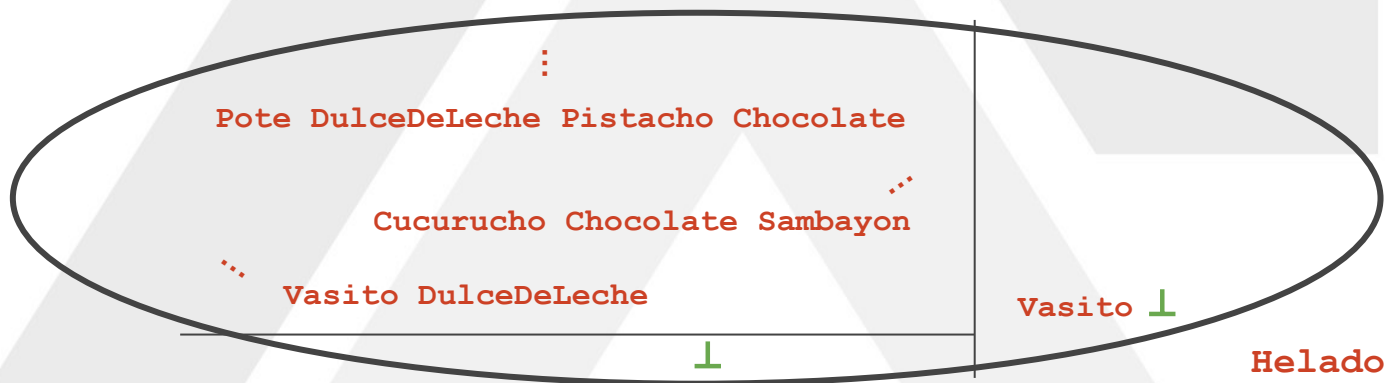
```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

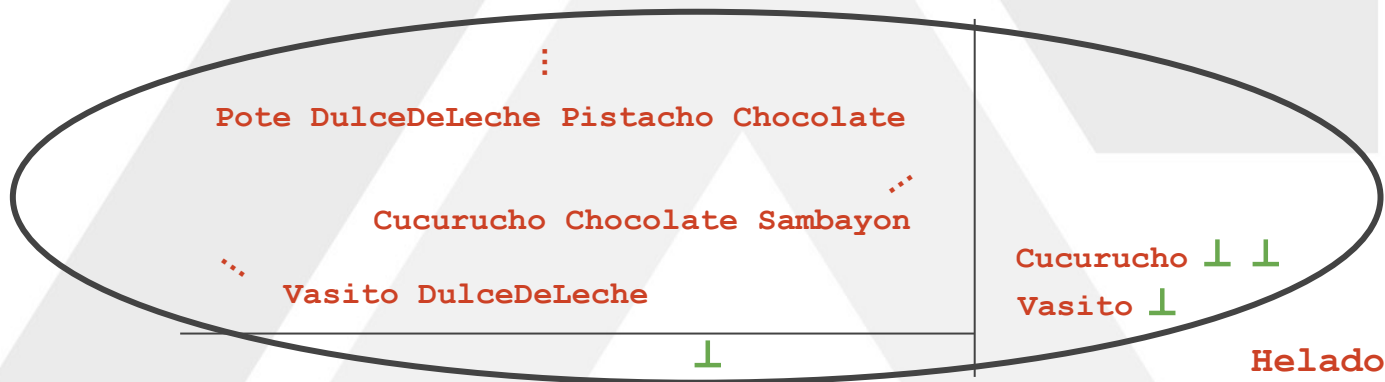
```
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

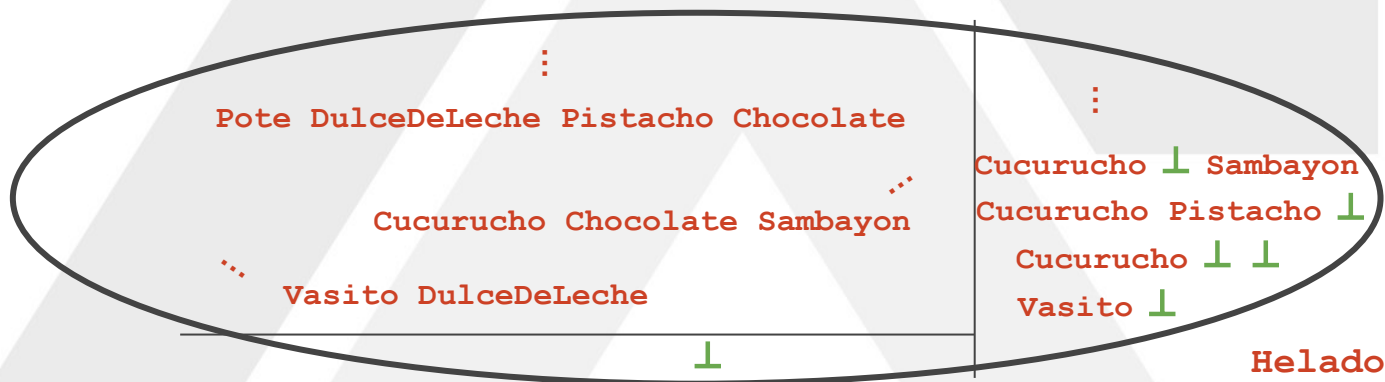
```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```



Tipos algebraicos

- Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Se pueden construir funciones y elementos

```
miHeladoFavorito :: ??
miHeladoFavorito = Cucurucho Chocolate
                  Sambayon

poteDeUnGusto :: ??
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Se pueden construir funciones y elementos

```
miHeladoFavorito :: Helado
miHeladoFavorito = Cucurucho Chocolate
                  Sambayon

poteDeUnGusto :: ??
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Se pueden construir funciones y elementos

```
miHeladoFavorito :: Helado
miHeladoFavorito = Cucurucho Chocolate
                  Sambayon

poteDeUnGusto ::      ->
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Declaración con argumentos en los constructores

```
data Helado = Vasito Gusto
            | Cucurucho Gusto Gusto
            | Pote Gusto Gusto Gusto
```

- Se pueden construir funciones y elementos

```
miHeladoFavorito :: Helado
miHeladoFavorito = Cucurucho Chocolate
                  Sambayon

poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

❏ ¿Se podrían modelar los helados así?

```
data Contenedor = Vasito | Cucurucho | Pote
data HeladoFeo  = HFeo Contenedor
                Gusto Gusto Gusto
```


Tipos algebraicos

- ❑ ¿Se podrían modelar los helados así?

```
data Contenedor = Vasito | Cucurucho | Pote
data HeladoFeo  = HFeo Contenedor
                  Gusto Gusto Gusto
```

- ❑ ¿Cuál sería la desventaja?

Tipos algebraicos

- ❑ ¿Se podrían modelar los helados así?

```
data Contenedor = Vasito | Cucurucho | Pote
data HeladoFeo  = HFeo Contenedor
                  Gusto Gusto Gusto
```

- ❑ ¿Cuál sería la desventaja?

- ❑ ¡No se puede modelar que el vasito tiene UN gusto!

HFeo Vasito DulceDeLeche ?? ??

Tipos algebraicos

- ❑ ¿Se podrían modelar los helados así?

```
data Contenedor = Vasito | Cucurucho | Pote
data HeladoFeo = HFeo Contenedor
                  Gusto Gusto Gusto
```

- ❑ ¿Cuál sería la desventaja?

- ❑ ¡No se puede modelar que el vasito tiene UN gusto!

HFeo Vasito DulceDeLeche ?? ??

- ❑ Por eso es mejor usar constructores parametrizados

Tipos algebraicos

- ❑ ¿Se podrían modelar los helados así?

```
data Contenedor = Vasito | Cucurucho | Pote
data HeladoFeo  = HFeo Contenedor
                  Gusto Gusto Gusto
```

- ❑ ¿Cuál sería la desventaja?

- ❑ ¡No se puede modelar que el vasito tiene UN gusto!

HFeo Vasito DulceDeLeche ?? ??

- ❑ Por eso es mejor usar constructores parametrizados

- ❑ ¡La *estructura* captura la restricción!

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: ??
```

```
chocoHelate consH = consH Chocolate
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: ->  
chocoHelate consH = consH Chocolate
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (    ->    ) ->  
chocoHelate consH = consH Chocolate
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto ->      ) ->  
chocoHelate consH = consH Chocolate
```


Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

```
poteDeUnGusto :: Gusto -> Helado  
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado)->Helado
```

```
chocoHelate consH = consH Chocolate
```

- ¿Y cómo reduce?

```
chocoHelate poteDeUnGusto
```

```
poteDeUnGusto :: Gusto -> Helado  
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado)->Helado  
chocoHelate consH = consH Chocolate
```

- ¿Y cómo reduce?

chocoHelate poteDeUnGusto



```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
chocoHelate consH = consH Chocolate
```

- ¿Y cómo reduce?

chocoHelate poteDeUnGusto

→ (def. de chocoHelate, con `consH <- poteDeUnGusto`)
`poteDeUnGusto Chocolate`

```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
chocoHelate consH = consH Chocolate
```

- ¿Y cómo reduce?

$\xrightarrow{\text{consH}}$ chocoHelate poteDeUnGusto (def. de chocoHelate, con $\text{consH} \leftarrow \text{poteDeUnGusto}$)
poteDeUnGusto Chocolate

```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

■ Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

■ ¿Y cómo reduce?

chocoHelate poteDeUnGusto

→ (def. de chocoHelate, con `consH` ← `poteDeUnGusto`)

poteDeUnGusto Chocolate

→

```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
chocoHelate consH = consH Chocolate
```

¿Y cómo reduce?

chocoHelate poteDeUnGusto

→ (def. de chocoHelate, con **consH** <- **poteDeUnGusto**)

poteDeUnGusto Chocolate

→ (def. de poteDeUnGusto, con **g** <- **Chocolate**)

Pote Chocolate Chocolate Chocolate

```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
chocoHelate consH = consH Chocolate
```

¿Y cómo reduce?

chocoHelate poteDeUnGusto

→ ^g (def. de chocoHelate, con consH <- poteDeUnGusto)

poteDeUnGusto Chocolate

→ (def. de poteDeUnGusto, con g <- Chocolate)

Pote Chocolate Chocolate Chocolate

^g

^g

^g


```
poteDeUnGusto :: Gusto -> Helado
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
chocoHelate consH = consH Chocolate
```

¿Y cómo reduce?

chocoHelate poteDeUnGusto

→ (def. de chocoHelate, con **consH** ← **poteDeUnGusto**)

poteDeUnGusto Chocolate

→ (def. de poteDeUnGusto, con **g** ← **Chocolate**)

Pote Chocolate Chocolate Chocolate

¡ES una formal
normal!

```
poteDeUnGusto :: Gusto -> Helado  
poteDeUnGusto g = Pote g g g
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado)->Helado
```

```
chocoHelate consH = consH Chocolate
```

- ¿Y qué denota?

```
chocoHelate poteDeUnGusto
```

```
= Pote Chocolate Chocolate Chocolate
```

Tipos algebraicos

```
poteDeUnGusto :: Gusto -> Helado  
poteDeUnGusto g = Pote g g g
```

❏ Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

❏ **Chocolate :: Gusto**

❏ **Vasito Chocolate :: Helado**

❏ **Vasito :: ??**

Tipos algebraicos

Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

Chocolate :: Gusto

Vasito Chocolate :: Helado

Vasito :: Gusto -> Helado

¡Porque está
aplicada, ES
una función!

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

- Se puede usar un constructor como argumento

```
chocoHelate Vasito
```

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

- Se puede usar un constructor como argumento

```
chocoHelate Vasito
```



Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

- Se puede usar un constructor como argumento

```
chocoHelate Vasito
```

→ (def. de chocoHelate, con consH <- Vasito)
Vasito Chocolate

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado  
chocoHelate consH = consH Chocolate
```

- Se puede usar un constructor como argumento

chocoHelate Vasito

→ (def. de chocoHelate, con **consH** = **Vasito**)
Vasito Chocolate

Es una formal normal

Tipos algebraicos

- Tipos algebraicos y orden superior

```
chocoHelate :: (Gusto->Helado) -> Helado
```

```
chocoHelate consH = consH Chocolate
```

- Se puede usar un constructor como argumento

```
chocoHelate Vasito
```

```
= Vasito Chocolate
```

Tipos algebraicos

❏ Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

❏ **Vasito :: Gusto -> Helado**

❏ **Cucurucho :: ??**

❏ **Pote :: ??**

Tipos algebraicos

❏ Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

❏ **Vasito :: Gusto -> Helado**

❏ **Cucurucho :: -> ->**

❏ **Pote :: ??**

Tipos algebraicos

❏ Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

❏ **Vasito :: Gusto -> Helado**

❏ **Cucurucho :: Gusto -> Gusto -> Helado**

❏ **Pote :: ??**

Tipos algebraicos

❏ Constructores con argumentos

```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

- ❏ **Vasito :: Gusto -> Helado**
- ❏ **Cucurucho :: Gusto -> Gusto -> Helado**
- ❏ **Pote :: Gusto->Gusto->Gusto->Helado**

Tipos algebraicos

❏ Constructores con argumentos

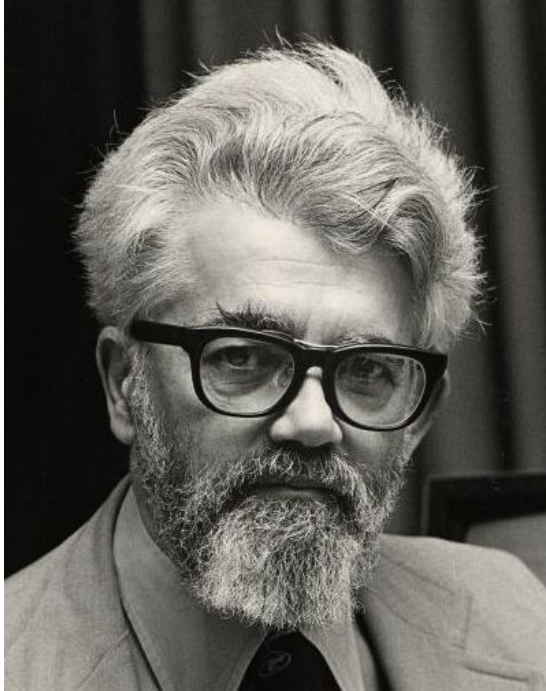
```
data Gusto = Chocolate | DulceDeLeche  
           | Sambayon | Frutilla | Pistacho  
data Helado = Vasito Gusto  
            | Cucurucho Gusto Gusto  
            | Pote Gusto Gusto Gusto
```

❏ `Vasito :: Gusto -> Helado`

❏ `Cucurucho :: Gusto -> (Gusto -> Helado)`

❏ `Pote :: Gusto -> (Gusto -> (Gusto -> Helado))`

John McCarthy



John McCarthy

(4 de septiembre 1927 – 24 de octubre 2011) es un científico de la computación estadounidense que desarrolló su carrera en la Universidad de Stanford. Tomó clases con John von Neumann en Caltech, quién lo inspiró en su carrera.

Desarrolló el lenguaje LISP en 1958, inventó el concepto de *garbage collection* en 1959, participó en la creación de ALGOL y también fue uno de los fundadores de la disciplina de Inteligencia Artificial (junto con Alan Turing, Marvin Minsky y otros), de la que es responsable por acuñar el término en 1955. Fue quién propuso el uso de recursión y de if-then-else en lenguajes de programación, e introdujo el uso de funciones de orden superior en programación para LISP.

Robin Milner



Arthur John Robin Gorell Milner

(13 de enero 1934 – 20 de marzo 2010) es un científico de la computación británico, fundador del Laboratorio de Fundamentos de Ciencias de la Computación (LFCS) de la Universidad de Edimburgo.

Desarrolló la *Lógica para Funciones Computables* (**LCF**) en 1972, una de las primeras herramientas para demostración automatizada de teoremas, el lenguaje **ML** (Meta-Language) en 1973, primer lenguaje con inferencia de tipos polimórfica (el sistema de tipos Hindley-Milner), pensado para implementar la LCF, y el *Cálculo de Sistemas Comunicantes* (**CCS**) en 1980 y su sucesor, el **π -cálculo** en 1992, para analizar sistemas concurrentes.



Pattern matching

Tipos algebraicos y *pattern matching*

- ❑ Los constructores se usan para describir elementos
- ❑ ¿Pueden usarse para algo más?

Tipos algebraicos y *pattern matching*

- ❑ Los constructores se usan para describir elementos
- ❑ ¿Pueden usarse para algo más?
 - ❑ ¡Para *preguntar* a un elemento si fue construido con él!
 - ❑ ***Pattern matching***

Tipos algebraicos y *pattern matching*

- ❑ Los constructores se usan para describir elementos
- ❑ ¿Pueden usarse para algo más?
 - ❑ ¡Para *preguntar* a un elemento si fue construido con él!
 - ❑ ***Pattern matching***
 - ❑ Es para preguntar en lugar de describir valores (“*desarmar*” -- o “*deconstruir*” -- en lugar de “*construir*”)
 - ❑ ¿Cómo lo distinguimos?

Tipos algebraicos y *pattern matching*

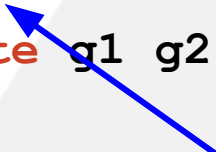
- ❑ Los constructores se usan para describir elementos
- ❑ ¿Pueden usarse para algo más?
 - ❑ ¡Para *preguntar* a un elemento si fue construido con él!
 - ❑ ***Pattern matching***
 - ❑ Es para preguntar en lugar de describir valores (“*desarmar*” -- o “*deconstruir*” -- en lugar de “*construir*”)
 - ❑ ¿Cómo lo distinguimos?
 - ❑ Se usa un constructor del lado de los parámetros (con ciertas restricciones)

Tipos algebraicos y *pattern matching*

■ *Pattern matching*

- Se usa un constructor en posición de parámetro

```
esPote :: Helado -> Bool
esPote (Vasito g)           = False
esPote (Cucurucho g1 g2)    = False
esPote (Pote g1 g2 g3)      = True
```



Constructores usados
para preguntar

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

■ *Pattern matching*

■ ¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

■ *Pattern matching*

■ ¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

■ Pattern matching

■ ¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

Este es el redex que
habría que reducir

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

¿Cómo funciona?

`esPote (chocoHelate poteDeUnGusto)`

x

Pero como no hay ecuaciones,
cede la prioridad por **una**
reducción

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)
→ (def. de **chocoHelate**, con **consH** < **poteDeUnGusto**)
esPote (poteDeUnGusto Chocolate)

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

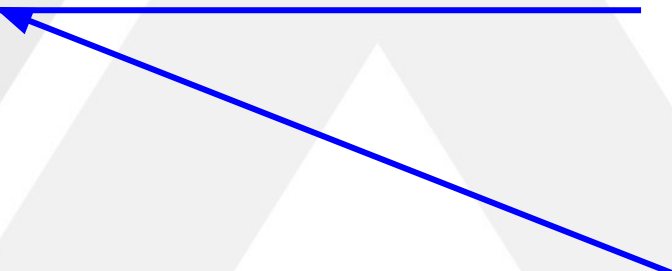
¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

→ (def. de **chocoHelate**, con **consH** < **poteDeUnGusto**)

esPote (poteDeUnGusto Chocolate)

→



Vuelve a suceder
lo mismo

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

→ (def. de chocoHelate, con **consH** ← **poteDeUnGusto**)

esPote (poteDeUnGusto Chocolate)

→ (def. de poteDeUnGusto, con **g** ← **Chocolate**)

esPote (Pote Chocolate Chocolate Chocolate)

Vuelve a suceder
lo mismo

Tipos algebraicos y *pattern matching*

```
esPote :: Helado -> Bool
esPote (Vasito g)           = False
esPote (Cucurucho g1 g2)    = False
esPote (Pote g1 g2 g3)      = True
```

Pattern matching

¿Cómo funciona?

esPote (chocoHelate poteDeUnGusto)

→ (def. de chocoHelate, con **consH** < **poteDeUnGusto**)

esPote (poteDeUnGusto Chocolate)

→ (def. de poteDeUnGusto, con **g** ← **Chocolate**)

esPote (Pote Chocolate Chocolate Chocolate)

g3

¡Ahora coincide con la 3era ecuación!

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

¿Cómo funciona?

```
esPote (chocoHelate poteDeUnGusto)
→ (def. de chocoHelate, con consH<-poteDeUnGusto)
esPote (poteDeUnGusto Chocolate)
→ (def. de poteDeUnGusto, con g<-Chocolate)
esPote (Pote Chocolate Chocolate Chocolate)
→ (def. de esPote, con g1, g2, g3<-Chocolate)
True
```

Tipos algebraicos y *pattern matching*

- ❑ ***Pattern matching***: ¿cómo funciona?
 - ❑ El argumento se reduce lo mínimo necesario
 - ❑ Hasta saber qué ecuación usar
 - ❑ La determinación se chequea en el *orden de las ecuaciones*
 - ❑ Cuando el argumento coincide con el *pattern* (está hecho con ese constructor), los nombres del *pattern* toman como valor las partes correspondientes del argumento
 - ❑ Por eso deben ser todos distintos
 - ❑ Se realiza la reducción, usando esas coincidencias

Tipos algebraicos y pattern matching

```
esPote :: Helado -> Bool
esPote (Vasito g)      = False
esPote (Cucurucho g1 g2) = False
esPote (Pote g1 g2 g3)  = True
```

Pattern matching

- ¿Para qué sirve el *orden de las ecuaciones* al calcular?
- Mejorar la expresividad

```
esPote :: Helado -> Bool
esPote (Pote _ _ _) = True
esPote _            = False
```

Patterns especiales,
que indican ignorar el
argumento

Tipos algebraicos y *pattern matching*

❏ *Pattern matching*

- ❏ ¿Para qué sirve el orden de las ecuaciones al calcular?
- ❏ Mejorar la expresividad (**OJO: el orden importa**)

```
esPoteMal :: Helado -> Bool
esPoteMal _ = False
esPoteMal (Pote _ _ _) = True
```

¡NO ES EQUIVALENTE
a la anterior!

Tipos algebraicos y *pattern matching*

■ ***Pattern matching***: ¿por qué el nombre?

Tipos algebraicos y *pattern matching*

- ***Pattern matching***: ¿por qué el nombre?
 - *Pattern* (esquema, “patrón”)
 - Expresión hecha de constructores y variables *distintas*
 - Usada solamente como parámetro (NO argumento)
 - Funciona como pregunta

Tipos algebraicos y *pattern matching*

- ❏ ***Pattern matching***: ¿por qué el nombre?
 - ❏ *Pattern* (esquema, “patrón”)
 - ❏ Expresión hecha de constructores y variables *distintas*
 - ❏ Usada solamente como parámetro (NO argumento)
 - ❏ Funciona como pregunta
 - ❏ *Matching* (coincidencia, “correspondencia”)
 - ❏ Operación asociada a un *pattern*
 - ❏ Inspecciona una expresión, y coincide o no
 - ❏ Si coincide, liga las variables

Tipos algebraicos y *pattern matching*

- ❑ ***Pattern matching***: ¿por qué el nombre?
 - ❑ *Pattern* (Esquema, “patrón”)
 - ❑ Expresión hecha de constructores y variables *distintas*
 - ❑ Usada solamente como parámetro (NO argumento)
 - ❑ Funciona como pregunta
 - ❑ *Matching* (coincidencia, “correspondencia”)
 - ❑ Operación asociada a un *pattern*
 - ❑ Inspecciona una expresión, y coincide o no
 - ❑ Si coincide, liga las variables
 - ❑ Coincidencia de esquemas (“correspondencia de patrones”)

Tipos algebraicos y *pattern matching*

❏ *Pattern matching*

❏ Uso de variables en el pattern

```
gustoSerio :: Gusto -> Gusto  
gustoSerio Frutilla = Chocolate  
gustoSerio g       = g
```

Estas son preguntas

Estas son descripciones

Tipos algebraicos y pattern matching

Pattern matching

Uso de variables en el pattern

```
heladoSerio :: Helado -> Helado
heladoSerio (Vasito g)          = Vasito (gustoSerio g)
heladoSerio (Cucurucho g1 g2) = Cucurucho (gustoSerio g1)
                                   (gustoSerio g2)
heladoSerio (Pote g1 g2 g3)    = Pote (gustoSerio g1)
                                   (gustoSerio g2)
                                   (gustoSerio g3)
```

Estas son preguntas

Estas son descripciones

Tipos algebraicos y pa

```
gustoSerio Frutilla = Chocolate
gustoSerio g      = g

heladoSerio (Vasito g) = Vasito (gustoSerio g)
heladoSerio (Cucurucho g1 g2) = ...
...
```

Pattern matching

- Uso de variables en el pattern

heladoSerio (Vasito ^gFrutilla)

→ (def. de heladoSerio, con $g <- \text{Frutilla}$)

Vasito (gustoSerio Frutilla)

→ (def. de gustoSerio)

Vasito Chocolate

Tipos algebraicos y pa

```
gustoSerio Frutilla = Chocolate
gustoSerio g      = g

heladoSerio (Vasito g) = Vasito (gustoSerio g)
heladoSerio (Cucurucho g1 g2) = ...
...
```

Pattern matching

- Uso de variables en el pattern

heladoSerio (Cucurucho ^{g1}Frutilla ^{g2}Sambayon)

→ (def. de heladoSerio, con $g1 \leftarrow \text{Frutilla}$, $g2 \leftarrow \text{Sambayon}$)
Cucurucho (gustoSerio Frutilla) (gustoSerio Sambayon)

→ (def. de gustoSerio) ^g
Cucurucho Chocolate (gustoSerio Sambayon)

→ (def. de gustoSerio, con $g \leftarrow \text{Sambayon}$)
Cucurucho Chocolate Sambayon

Tipos algebraicos con parámetros

Tipos algebraicos con parámetros

- ❏ ¿Cómo definir varios tipos con estructura común?

Tipos algebraicos con parámetros

❏ ¿Cómo definir varios tipos con estructura común?

```
data ParDeInts    = DosInts  Int  Int
```

```
data ParDeBools   = DosBools Bool  Bool
```

```
data ParDeGustos  = DosGustos Gusto Gusto
```

Tipos algebraicos con parámetros

- ❏ ¿Cómo definir varios tipos con estructura común?

```
data ParDeInts    = DosInts  Int  Int
```

```
data ParDeBools   = DosBools Bool Bool
```

```
data ParDeGustos  = DosGustos Gusto Gusto
```

- ❏ Nuevamente, tenemos el problema de la reiteración

Tipos algebraicos con parámetros

- ❑ ¿Cómo definir varios tipos con estructura común?

```
data ParDeInts    = DosInts  Int  Int
```

```
data ParDeBools   = DosBools Bool Bool
```

```
data ParDeGustos  = DosGustos Gusto Gusto
```

- ❑ Nuevamente, tenemos el problema de la reiteración
 - ❑ Antes se arregló con parametrización de constructores...

Tipos algebraicos con parámetros

- ❑ ¿Cómo definir varios tipos con estructura común?

```
data ParDeInts      = DosInts  Int  Int
```

```
data ParDeBools     = DosBools Bool Bool
```

```
data ParDeGustos    = DosGustos Gusto Gusto
```

- ❑ Nuevamente, tenemos el problema de la reiteración
 - ❑ Antes se arregló con parametrización de constructores...
 - ❑ ¿Se podrá hacer algo similar con los tipos?

Tipos algebraicos con parámetros

- ¿Cómo definir varios tipos con estructura común?
 - ¡Parámetros en el nivel de tipos!
 - ¿Con qué sintaxis? ¡La misma!

Tipos algebraicos con parámetros

- ❏ ¿Cómo definir varios tipos con estructura común?
 - ❏ ¡Parámetros en el nivel de tipos!
 - ❏ ¿Con qué sintaxis? ¡La misma!
 - ❏ Una declaración de tipo puede estar parametrizada
- ```
data Par a = DosCosas a a
```

# Tipos algebraicos con parámetros

- ❏ ¿Cómo definir varios tipos con estructura común?
  - ❏ ¡Parámetros en el nivel de tipos!
  - ❏ ¿Con qué sintaxis? ¡La misma!
- ❏ Una declaración de tipo puede estar parametrizada

```
data Par a = DosCosas a a
```

  - ❏ ¿Qué elementos tiene `Par Int`? ¿Y `Par Bool`?

# Tipos algebraicos con parámetros

- ❏ ¿Cómo definir varios tipos con estructura común?
  - ❏ ¡Parámetros en el nivel de tipos!
  - ❏ ¿Con qué sintaxis? ¡La misma!
- ❏ Una declaración de tipo puede estar parametrizada
  - `data Par a = DosCosas a a`
  - ❏ ¿Qué elementos tiene `Par Int`? ¿Y `Par Bool`?
  - ❏ ¿Qué otros conjuntos `Par` puede haber?

# Tipos algebraicos con parámetros

- ❑ ¿Cómo definir varios tipos con estructura común?
  - ❑ ¡Parámetros en el nivel de tipos!
  - ❑ ¿Con qué sintaxis? ¡La misma!
- ❑ Una declaración de tipo puede estar parametrizada
  - `data Par a = DosCosas a a`
  - ❑ ¿Qué elementos tiene `Par Int`? ¿Y `Par Bool`?
  - ❑ ¿Qué otros conjuntos `Par` puede haber?
    - ❑ `Par Gusto, Par Helado, Par (Par Int),  
Par (Int -> Int), Par (Gusto->Helado), ...`

# Tipos algebraicos con parámetros

❏ ¿Qué elementos tiene `Par Bool`?

```
data Par a = DosCosas a a
```

`DosCosas False True`

`DosCosas False False`

`DosCosas True True`

`DosCosas True False`

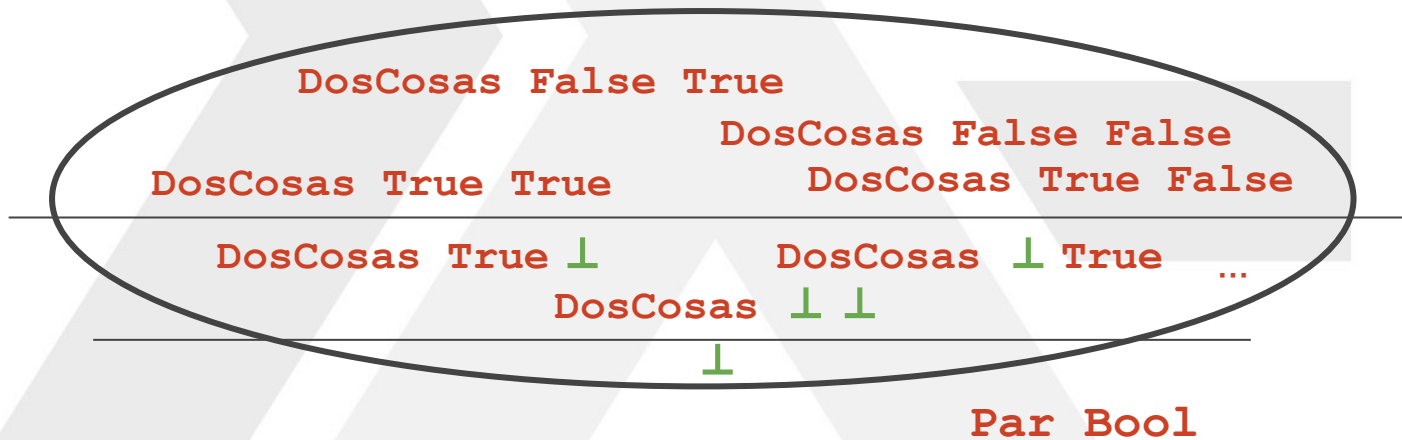
⊥

`Par Bool`

# Tipos algebraicos con parámetros

❏ ¿Qué elementos tiene `Par Bool`?

```
data Par a = DosCosas a a
```



# Tipos algebraicos con parámetros

- ❑ Pensemos en el siguiente diálogo
  - *¿Cuántas monedas tenés en el bolsillo?*
  - *Cero.*
  - *Tomá, guardá estas dos monedas.*
  - *Pero no tengo bolsillo...*
- ❑ ¿Cuáles son las respuestas posibles a la pregunta?
- ❑ ¿Qué significa la respuesta “cero”?
- ❑ ¿Cómo distinguir los casos con respuesta “cero”?



# Tipos algebraicos con parámetros

- Analicemos la pregunta: *¿Cuántas monedas tenés en el bolsillo?*
  - ¿Alcanza con una respuesta numérica?
  - Hay más de un caso para no tener monedas
    - Con bolsillo o sin bolsillo
  - ¡La respuesta es más compleja que un solo número!
    - **Puede** ser un número, pero no siempre
    - ¿Cómo representar esto con datos?  
`¿cuantasMonedas :: Persona -> Int?` ¡No!

# Tipos algebraicos con parámetros

- ❑ ¡La respuesta es más compleja que solo un número!

```
data Maybe a = Nothing | Just a
cuantasMonedas :: Persona -> Maybe Int
```

- ❑ ¿Qué elementos hay en `Maybe Int`?

- ❑ `Nothing`, no hay bolsillo
- ❑ `Just 0`, hay bolsillo pero no monedas
- ❑ `Just 1`, hay bolsillo y una moneda
- ❑ ...
- ❑ `⊥`, ¡BOOM!
- ❑ `Just ⊥`, se mira el bolsillo y ¡BOOM!



# **Expresividad de los tipos algebraicos**

# Tipos algebraicos y expresividad

- ❏ ¿Por qué se llaman tipos algebraicos?
  - ❏ Por sus características
    - ❏ toda combinación válida de constructores y valores es un elemento del tipo (y solamente ellas lo son)
      - ❏ No hay restricciones a los constructores
      - ❏ No hay elementos extra
    - ❏ dos elementos son iguales únicamente si están contruídos exactamente igual
      - ❏ La igualdad es por construcción *estructural*

# Tipos algebraicos y expresividad

- ❑ ¿Por qué se llaman tipos algebraicos?
  - ❑ Por sus características
    - ❑ toda combinación válida de constructores y valores es un elemento del tipo (y solamente ellas lo son)
      - ❑ No hay restricciones a los constructores
      - ❑ No hay elementos extra
    - ❑ dos elementos son iguales únicamente si están contruídos exactamente igual
      - ❑ La igualdad es por construcción estructural
  - ❑ *Álgebra libre* generada por los constructores

# Tipos algebraicos y expresividad

- Números complejos
    - toda combinación de dos reales es un complejo
    - dos complejos son iguales si tienen las mismas partes
- ```
data Complex = C Float Float -- C parteReal parteImaginaria

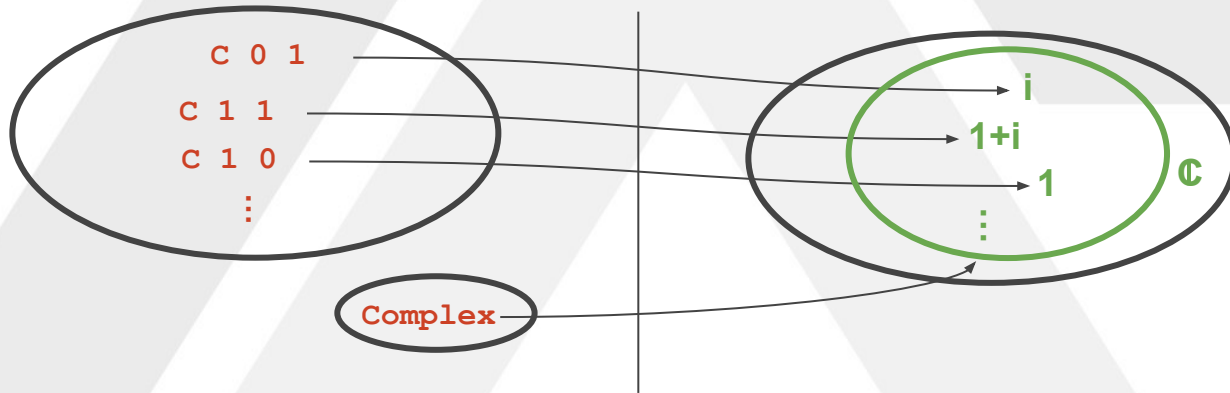
realPart (C r i) = r
imagePart (C r i) = i

addC (C r1 i1) (C r2 i2) = C (r1+r2) (i1+i2)

mkPolar rd a = C (rd * cos a) (rd * sin a)
```

Tipos algebraicos y expresividad

- Números complejos
 - toda combinación de dos reales es un complejo
 - dos complejos son iguales si tienen las mismas partes
- data Complex = C Float Float**



Tipos algebraicos y expresividad

- Números racionales
 - no todo par de enteros es un racional ($1/0$)
 - hay racionales iguales con partes distintas ($6/4 = 3/2$)
- ```
data WrongRational = R Int Int -- R numerador denominador

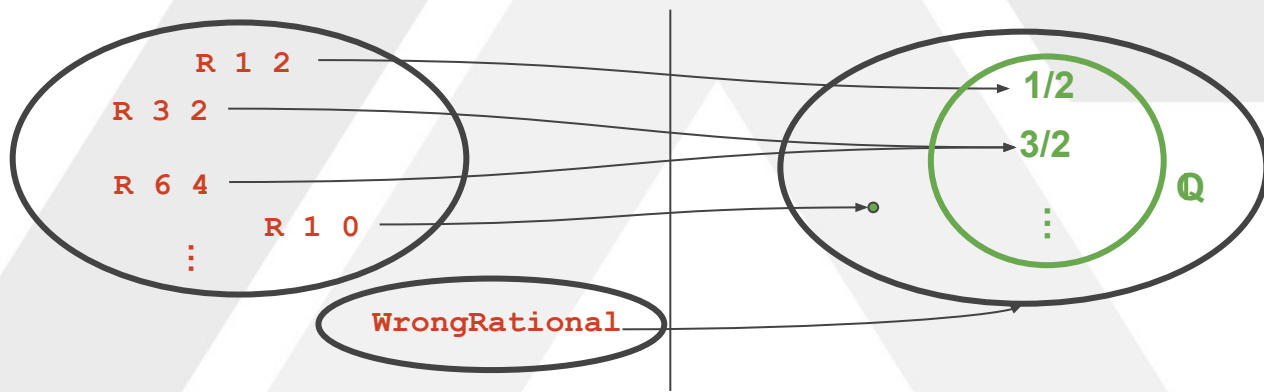
numerador (R n d) = n
denominador (R n d) = d
```
- ¡Los racionales NO SON algebraicos!



# Tipos algebraicos y expresividad

- Números racionales
  - no todo par de enteros es un racional ( $1/0$ )
  - hay racionales iguales con partes distintas ( $6/4 = 3/2$ )

```
data WrongRational = R Int Int
```



# Tipos algebraicos y expresividad

- ❑ Hay tipos que NO se pueden representar directamente como tipos algebraicos (p.ej. los Racionales)
  - ❑ Son necesarias más herramientas
    - ❑ ¡Tipos abstractos de datos!
    - ❑ Existen enfoques algebraicos a los TADs
  - ❑ Exceden el alcance de este curso

# Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?
  - ❑ Tipos enumerativos
  - ❑ Tipos producto (o registros)
  - ❑ Tipos variantes (o sumas)
  - ❑ Tipos recursivos estructurales
  - ❑ Otros tipos combinando con otras características
    - ❑ Orden superior
    - ❑ Recursión general

# Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?

- ❑ ***Tipos enumerativos***

- ❑ Solo constructores sin argumentos

- ❑ E.g. **Gusto**, **Bool**, etc.

```
data Gusto = Chocolate
 | DulceDeLeche
 | Sambayon | Frutilla
```

- ❑ ¿Qué otros enumerativos pueden pensarse?

- ❑ Días de la semana, meses, etc.

# Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?

- ❑ *Tipos producto*

- ❑ Un único constructor, varios argumentos

- ❑ E.g. **Par**, **Pokemon**, etc.

- `data Pokemon =`

- `Catchem Nombre Tipo Nivel`

- ❑ ¿Qué otros productos pueden pensarse?

- ❑ ¡Registros!

# Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?

- ❑ *Tipos variante (o sumas)*

- ❑ Muchos constructores con argumentos

- ❑ E.g. **Helado**, **Maybe**, etc.

- ```
data Shape = Circle Float
           | Rectangle Float Float
```

- ❑ ¿Qué otras sumas pueden pensarse?

- ❑ Cuentas bancarias, etc.

Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?
 - ❑ ***Tipos recursivos estructurales***
 - ❑ El tipo definido aparece como argumento
 - ❑ Listas, árboles
 - ❑ Los estudiaremos más en detalle en breve

Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?
 - ❑ **Otros tipos combinando con otras características**
 - ❑ E.g. combinando con orden superior

```
data Set a = S (a -> Bool)

pares, enterosPositivos :: ??
pares = S (\n -> esPar n)
enterosPositivos = S (\n -> n>0)
```
 - ❑ Hay universos para explorar...

Tipos algebraicos y expresividad

- ❑ ¿Qué se puede construir con tipos algebraicos?

- ❑ *Otros tipos combinando con otras características*

- ❑ E.g. combinando con orden superior

```
data Set a = S (a -> Bool)
```

```
pares, enterosPositivos :: Set Int
```

```
pares = S (\n -> esPar n)
```

```
enterosPositivos = S (\n -> n>0)
```

- ❑ Hay universos para explorar...



Resumen

Resumen

- ❑ Mecanismo de tipos algebraicos
 - ❑ Constructores de orden superior
 - ❑ Pattern matching
 - ❑ Tipos algebraicos con parámetros
 - ❑ Expresividad de tipos algebraicos
 - ❑ Clasificación y ejemplos