



Programación Funcional

Clases teóricas

por Pablo E. “Fidel” Martínez López

4. Reducción

“Para transformar esta piedra en una gema tienes que ponerle otro nombre verdadero.”

Un mago de Terramar
Úrsula K. Le Guin





Computación por reducción

Computación por reducción

- ❏ ¿Cómo calcular mecánicamente el valor de una expresión?
 1. Repetir los pasos 2 y 3 hasta que no haya ninguna subexpresión que cumpla la condición dada por 2
 2. Encontrar una subexpresión que coincida con una instancia del lado izquierdo de una ecuación
 3. Reemplazarla por la correspondiente instancia del lado derecho
- ❏ Este proceso es llamado ***mecanismo de reducción***
 - ❏ O simplemente, **reducción**
 - ❏ Lo expresamos con →

Computación por reducción

■ Algunas definiciones

■ **Redex** (por *reducible expression*)

Subexpresión que coincide con una instancia del lado izquierdo de una ecuación

■ **Forma normal**

Expresión que no contiene redexes

■ La *reducción* reduce la distancia a la forma normal

Computación por reducción

❑ Mecanismo de reducción

Repetir hasta forma normal

Localizar un redex

Reemplazarlo

- ❑ Ya vimos ejemplos de reducciones en la clase 1
- ❑ ¿Qué propiedades tiene este mecanismo?

Propiedades de la reducción

- ❏ Propiedades

- ❏ **Confluencia**

- ¿la forma normal es única?

- ❏ **Normalización**

- ¿la forma normal siempre existe?

- ❏ **Orden de reducción**

- ¿importa en qué orden se eligen los redexes?



Confluencia

Confluencia

- ❑ **Confluencia** - ¿la forma normal es única?
 - ❑ ¿Qué implicaría si no fuese así?
 - ❑ ¡Habría más de un significado!
 - ❑ Es importante que lo sea para el significado
 - ❑ La forma de las definiciones permitidas influyen sobre la confluencia
 - ❑ En Haskell hay confluencia
 - ❑ No elaboraremos más sobre este punto



Normalización

Normalización

- ❑ Considerar las siguientes definiciones

```
inf :: ...
```

```
inf = inf + 1
```

```
detect :: ...
```

```
detect 42 = True
```

- ❑ ¿Tienen tipo?
- ❑ ¿Cómo reducen?

Normalización

❏ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: ...
```

```
inf = inf + 1
```

Normalización

❏ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: Int -- el resultado de la suma entre dos Ints
inf = inf + 1
```

Normalización

❏ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: Int -- el resultado de la suma entre dos Ints  
inf = inf + 1
```

inf → ...

Normalización

❏ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: Int -- el resultado de la suma entre dos Ints  
inf = inf + 1
```

`inf` → `inf+1` → ...

Normalización

❏ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: Int -- el resultado de la suma entre dos Ints  
inf = inf + 1
```

`inf` \rightarrow `inf+1` \rightarrow `(inf+1)``+1` \rightarrow ...

Normalización

- ❑ ¿Cuál es el tipo de `inf`? ¿Y cómo reduce?

```
inf :: Int -- el resultado de la suma entre dos Ints
inf = inf + 1
```

`inf` \rightarrow `inf+1` \rightarrow `(inf+1)``+1` \rightarrow ...

- ❑ ¡El redex nunca desaparece! La reducción no termina...
- ❑ ¿Es una ecuación orientada?

Normalización

❏ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect :: ...
```

```
detect 42 = True
```

Normalización

❏ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect ::      ->
```

```
detect 42 = True
```

Normalización

❏ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect :: Int -> Bool
```

```
detect 42 = True
```

Normalización

❏ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect :: Int -> Bool
```

```
detect 42 = True
```

```
detect 42 → ...
```

Normalización

- ❑ ¿Cuál es el tipo de `detect`? ¿Y cómo reduce?

```
detect :: Int -> Bool
```

```
detect 42 = True
```

```
detect 42 → True
```

Normalización

❑ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect :: Int -> Bool
```

```
detect 42 = True
```

```
detect 42 → True
```

```
detect 21 →
```

Normalización

- ❑ ¿Cuál es el tipo de **detect**? ¿Y cómo reduce?

```
detect :: Int -> Bool
```

```
detect 42 = True
```

```
detect 42 → True
```

```
detect 21 ↯ -- ¡no hay ecuaciones!
```

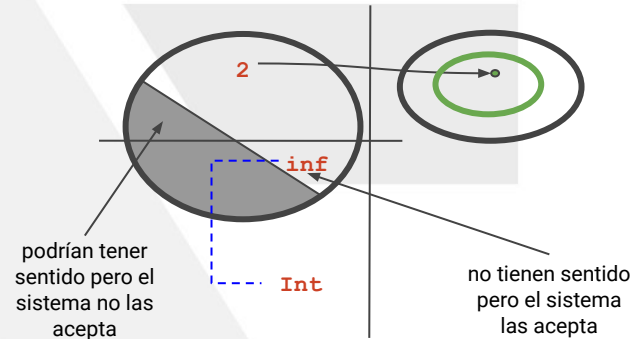
- ❑ ¡El redex no se puede reemplazar! Y no es forma normal...
 - ❑ ¿Es un redex?

Normalización

- ❑ **Normalización** - ¿La forma normal siempre existe?
 - ❑ ¡No!
 - ❑ ¿Cuáles son las formas de no llegar a forma normal?
 - ❑ Nunca termina de reducir (e.g. **inf**)
 - ❑ No puede seguir pero no llegó a forma normal (e.g. **detect**)
- ❑ Se llama *normalización* porque tiene que ver con la obtención de la forma *normal*
 - ❑ ¡No toda expresión tiene forma normal!

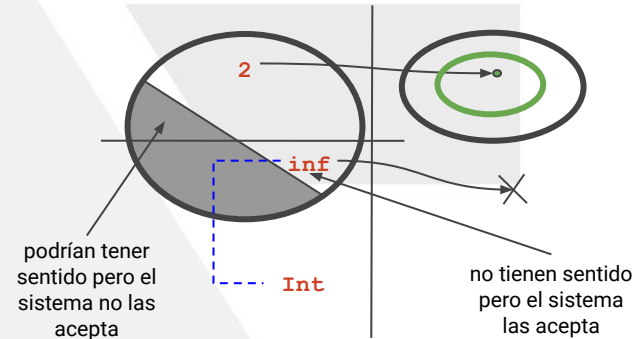
Normalización

- Consecuencias de la falta de formas normales
 - ¿Cuál es el valor de **inf**? ¿Existe?
 -



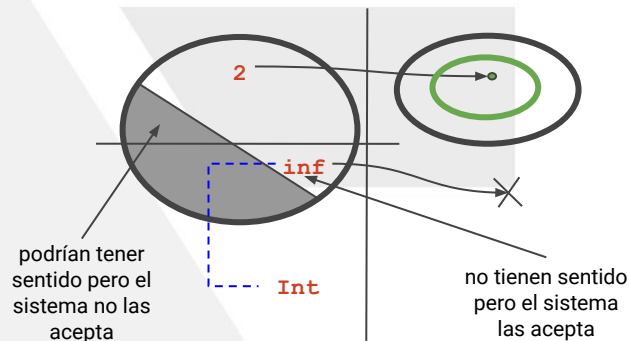
Normalización

- Consecuencias de la falta de formas normales
 - ¿Cuál es el valor de **inf**? ¿Existe?
 - Un número entero que sea igual a su sucesor



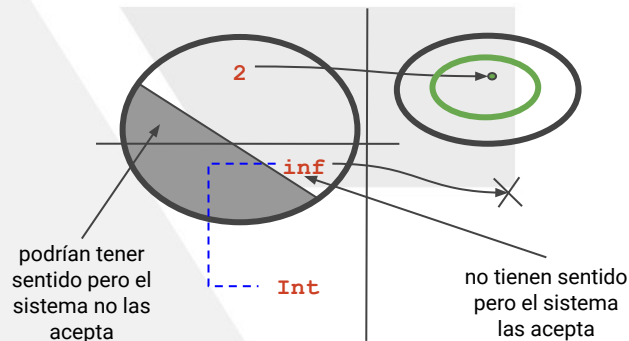
Normalización

- Consecuencias de la falta de formas normales
 - ¿Cuál es el valor de **inf**? ¿Existe?
 - Un número entero que sea igual a su sucesor
 - Pero **inf** tiene tipo...



Normalización

- Consecuencias de la falta de formas normales
 - ¿Cuál es el valor de **inf**? ¿Existe?
 - Un número entero que sea igual a su sucesor
 - Pero **inf** tiene tipo...
 - Genera una discrepancia
 - ¿Cómo la arreglamos?



Normalización

Consecuencias de la falta de formas normales

- ¿Cuál es el valor de **inf**? ¿Existe?

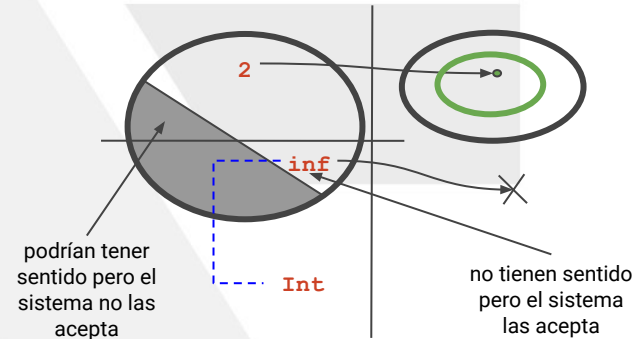
- Un número entero que sea igual a su sucesor

- Pero **inf** tiene tipo...

- Genera una discrepancia

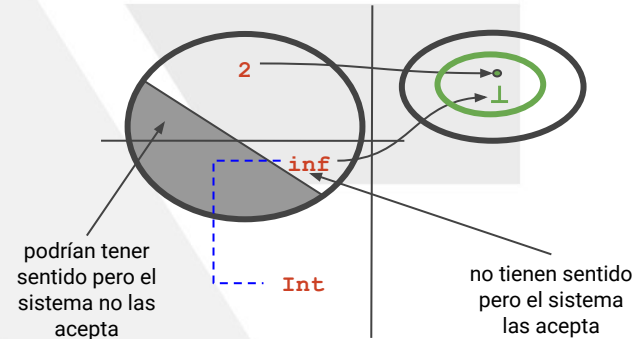
- ¿Cómo la arreglamos?

- Precisamos un
valor de error



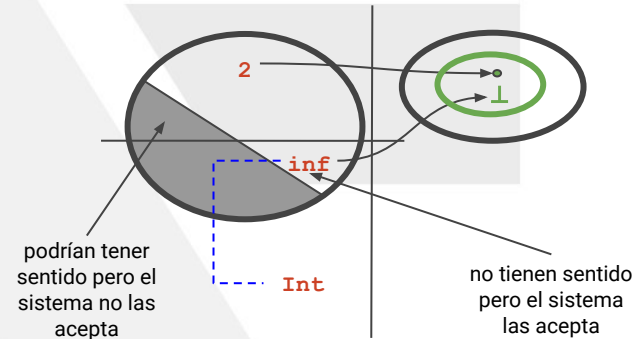
Normalización

- Consecuencias de la falta de formas normales
 - Valor de error: BOTTOM (\perp)**



Normalización

- Consecuencias de la falta de formas normales
 - Valor de error: BOTTOM (\perp)**
Valor teórico que representa a un error de cómputo



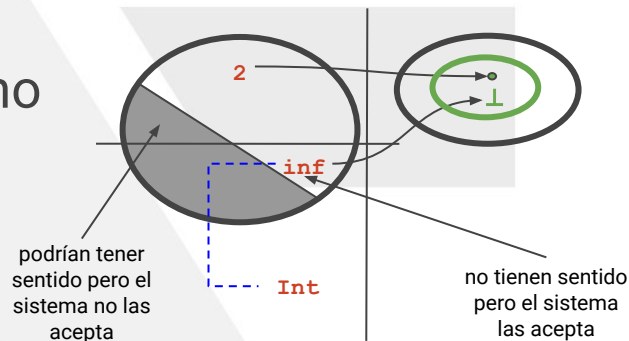
Normalización

Consecuencias de la falta de formas normales

Valor de error: BOTTOM (\perp)

Valor teórico que representa a un error de cómputo

- La reducción no termina
- La reducción no llega a destino



Normalización

Consecuencias de la falta de formas normales

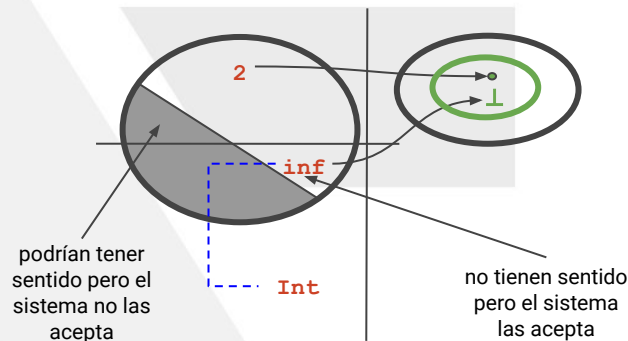
Valor de error: BOTTOM (\perp)

Valor teórico que representa a un error de cómputo

- La reducción no termina

- La reducción no llega

¡NO SE PUEDE manejar \perp de manera operacional!



Normalización

- ❑ Bottom es un valor especial

Normalización

- ❑ Bottom es un valor especial
 - ❑ Usamos el símbolo \perp para denotarlo

Normalización

- ❑ Bottom es un valor especial
 - ❑ Usamos el símbolo \perp para denotarlo
 - ❑ Observar que es verde
 - ❑ No existe por sí mismo en el mundo de la expresiones

Normalización

- ❑ Bottom es un valor especial
 - ❑ Usamos el símbolo \perp para denotarlo
 - ❑ Observar que es verde
 - ❑ No existe por sí mismo en el mundo de la expresiones
 - ❑ Es el valor de las expresiones que no terminan o no llegan a forma normal, un **valor de error**

Normalización

- ❑ Bottom es un valor especial
 - ❑ Usamos el símbolo \perp para denotarlo
 - ❑ Observar que es verde
 - ❑ No existe por sí mismo en el mundo de la expresiones
 - ❑ Es el valor de las expresiones que no terminan o no llegan a forma normal, un **valor de error**
 - ❑ No se puede observar operacionalmente si una expresión denota \perp (más de esto en un rato)

Normalización

■ ¿Por qué se llama bottom?

Normalización

- ¿Por qué se llama bottom?
 - En inglés significa “el fondo”, “lo de más abajo”

Normalización

- ❑ ¿Por qué se llama bottom?
 - ❑ En inglés significa “el fondo”, “lo de más abajo”
 - ❑ Es el valor con menos significado,
“el de más abajo” en la escala de significados

Normalización

- ❑ ¿Por qué se llama bottom?
 - ❑ En inglés significa “el fondo”, “lo de más abajo”
 - ❑ Es el valor con menos significado, “el de más abajo” en la escala de significados
 - ❑ En slang también se usa para otra cosa que puede servir en castellano...

Normalización

- ❑ ¿Por qué se llama bottom?
 - ❑ En inglés significa “el fondo”, “lo de más abajo”
 - ❑ Es el valor con menos significado, “el de más abajo” en la escala de significados
 - ❑ En slang también se usa para otra cosa que puede servir en castellano...
 - ❑ El BOOM de Gobstones es bottom
 - ❑ Pero esto fue meramente accidental

Normalización

- Hay expresiones que no terminan de todos los tipos

```
inf = inf+1
```

```
bb = not bb
```

```
bc = if False then 'a' else bc
```

Normalización

- Hay expresiones que no terminan de todos los tipos

```
inf = inf+1
```

```
bb = not bb
```

```
bc = if False then 'a' else bc
```

- ¿A qué tipo pertenece, entonces, \perp ?

Normalización

- Hay expresiones que no terminan de todos los tipos

```
inf = inf+1
```

```
bb = not bb
```

```
bc = if False then 'a' else bc
```

- ¿A qué tipo pertenece, entonces, \perp ?

- TODOS los tipos contienen este valor

Normalización

- Hay expresiones que no terminan de todos los tipos

```
inf = inf+1
```

```
bb = not bb
```

```
bc = if False then 'a' else bc
```

- ¿A qué tipo pertenece, entonces, \perp ?
- TODOS los tipos contienen este valor
- ¿Se puede definir una expresión polimórfica que dé \perp ?

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que dé \perp ?

`bottom :: ...`

`bottom = ...`

Normalización

- ❏ ¿Se puede definir una expresión polimórfica que de \perp ?

`bottom :: a`

`bottom = bottom`

- ❏ Para no terminación, se puede

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

bottom :: **a**

bottom = **bottom**

- ❑ Para no terminación, se puede
 - ❑ De hecho, \perp es el único valor con tipo **a**

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

`bottom :: a`

`bottom = bottom`

- ❑ Para no terminación, se puede
 - ❑ De hecho, \perp es el único valor con tipo `a`
 - ❑ En Haskell se llama `undefined` en lugar de `bottom`

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

bottom :: a

bottom = bottom

- ❑ Para no terminación, se puede
 - ❑ De hecho, \perp es el único valor con tipo **a**
 - ❑ En Haskell se llama **undefined** en lugar de **bottom**
- ❑ ¿Y se puede definir una que no llegue a forma normal?
 - ❑ Debería no tener ecuaciones...

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

error :: ...

error = ...

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

error :: ...

error = ...

- ❑ Que no llegue a forma normal, no se puede

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

```
error :: String -> a  
-- Predefinida en Haskell
```

- ❑ Que no llegue a forma normal, no se puede
 - ❑ Debe ser parte del lenguaje

Normalización

- ❑ ¿Se puede definir una expresión polimórfica que de \perp ?

```
error :: String -> a
-- Predefinida en Haskell
```

- ❑ Que no llegue a forma normal, no se puede
 - ❑ Debe ser parte del lenguaje
- ❑ ¿Para qué sirve el **String** del argumento?

Normalización

- ❑ Función de **error** `:: String -> a`
¿Para qué sirve el **String** del argumento?

Normalización

- ❑ Función de `error :: String -> a`
¿Para qué sirve el `String` del argumento?
 - ❑ El valor producido es el mismo, \perp
`error "Todo mal" = \perp`
`error "Se pudrió la momia" = \perp`

Normalización

- ❑ Función de `error :: String -> a`

¿Para qué sirve el `String` del argumento?

- ❑ El valor producido es el mismo, \perp

`error "Todo mal" = \perp`

`error "Se pudrió la momia" = \perp`

- ❑ ¡Pero cambia el comportamiento operacional!

- ❑ El mensaje de error es diferente
- ❑ Comprobarlo en el intérprete

Normalización

- ❑ ¿Qué quiere decir que no se puede observar operacionalmente si una expresión denota \perp ?
- ❑ El valor \perp no es observable operacionalmente

Normalización

❑ ¿Qué quiere decir que no se puede observar operacionalmente si una expresión denota \perp ?

❑ El valor \perp no es observable operacionalmente

`hpDiscr :: a -> Int`

`hpDiscr x = if (x==bottom) then 1 else 0`

Normalización

- ¿Qué quiere decir que no se puede observar operacionalmente si una expresión denota \perp ?

- El valor \perp no es observable operacionalmente

`hpDiscr :: a -> Int`

`hpDiscr x = if (x==bottom) then 1 else 0`

- ¿Cuánto vale `hpDiscr 2`?
- ¿Y `hpDiscr bottom`?
- ¿Y `hpDiscr (error "")`?

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

(def. de hpDiscr, con $x \leftarrow 2$)

if (2==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

(def. de hpDiscr, con $x \leftarrow 2$)

if (2==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

(def. de hpDiscr, con $x \leftarrow 2$)

if (2==bottom) then 1 else 0

→

(def. de bottom)

if (2==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr 2

→

(def. de hpDiscr, con $x \leftarrow 2$)

if (2==bottom) then 1 else 0

→

(def. de bottom)

if (2==bottom) then 1 else 0

→ ...

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

- El valor \perp no es observable operacionalmente

hpDiscr 2

→

(def. de hpDiscr, con $x \leftarrow 2$)

if (2==bottom) then 1 else 0

→

(def. de bottom)

if (2==bottom) then 1 else 0

→ ...

- ¡Nunca produce una forma normal!

hpDiscr 2 = \perp

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

`hpDiscr bottom`

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr bottom

→

(def. de hpDiscr, con $x \leftarrow \text{bottom}$)

if (bottom==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr bottom

→

(def. de hpDiscr, con $x \leftarrow \text{bottom}$)

if (bottom==bottom) then 1 else 0

→

(def. de bottom)

if (bottom==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr bottom

→

(def. de hpDiscr, con $x \leftarrow \text{bottom}$)

if (bottom==bottom) then 1 else 0

→

(def. de bottom)

if (bottom==bottom) then 1 else 0

→ ...

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

- ❑ El valor \perp no es observable operacionalmente

hpDiscr bottom

→

(def. de hpDiscr, con $x \leftarrow \text{bottom}$)

if (bottom==bottom) then 1 else 0

→

(def. de bottom)

if (bottom==bottom) then 1 else 0

→ ...

- ❑ ¡Tampoco produce nunca una forma normal!

hpDiscr bottom = \perp

Normalización

- ❏ El valor \perp no es observable operacionalmente

```
hpDiscr (error "")
```

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr (error "")

→

(def. de hpDiscr, con $x \leftarrow \text{error ""}$)

if (error ""==bottom) then 1 else 0

→

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

❏ El valor \perp no es observable operacionalmente

hpDiscr (error "")

→

(def. de hpDiscr, con $x \leftarrow \text{error ""}$)

if (error ""==bottom) then 1 else 0

↯

(def. de error)

Normalización

```
hpDiscr :: a -> Int
hpDiscr x = if (x==bottom)
              then 1
              else 0
```

- El valor \perp no es observable operacionalmente

hpDiscr (error "")

→

(def. de hpDiscr, con $x \leftarrow \text{error ""}$)

if (error ""==bottom) then 1 else 0

↯

(def. de error)

- ¡Nuevamente nunca produce una forma normal!

hpDiscr (error "") = \perp

Normalización

- El valor \perp no es observable operacionalmente

```
hpDiscr :: a -> Int
```

```
hpDiscr x = if (x==bottom) then 1 else 0
```

- `hpDiscr 2 = \perp`
- `hpDiscr bottom = \perp`
- `hpDiscr (error "") = \perp`

Normalización

- El valor \perp no es observable operacionalmente

```
hpDiscr :: a -> Int
```

```
hpDiscr x = if (x==bottom) then 1 else 0
```

- `hpDiscr 2 = \perp`

- `hpDiscr bottom = \perp`

- `hpDiscr (error "") = \perp`

- Por lo tanto, `hpDiscr = \x -> \perp`

¡Observar el abuso de notación!



Bottom y funciones

Bottom y funciones

■ ¿Cuál es la relación entre \perp y las funciones?

Bottom y funciones

- ¿Cuál es la relación entre \perp y las funciones?
 - ¿Puede devolver \perp al recibir algo totalmente definido?
 - ¿Puede devolver algo distinto de \perp cuando recibe \perp ?

Bottom y funciones

- ❑ ¿Cuál es la relación entre \perp y las funciones?
 - ❑ ¿Puede devolver \perp al recibir algo totalmente definido?
 - ❑ Valor \perp en los resultados
 - ❑ ¿Puede devolver algo distinto de \perp cuando recibe \perp ?
 - ❑ Valor \perp en los argumentos

Bottom y funciones

- ❑ ¿Cuál es la relación entre \perp y las funciones?
 - ❑ ¿Puede devolver \perp al recibir algo totalmente definido?
 - ❑ Valor \perp en los resultados
 - ❑ **Funciones parciales y totales**
 - ❑ ¿Puede devolver algo distinto de \perp cuando recibe \perp ?
 - ❑ Valor \perp en los argumentos
 - ❑ **Funciones estrictas y no estrictas**

Bottom y funciones

❏ ¿Cuál es la relación entre \perp y las funciones?

Resultado Argumento	$\neq \perp$	$= \perp$
$\neq \perp$ (totalmente definido)	¿Siempre? Función total	¿Puede ser? Función parcial
$= \perp$	¿Puede ser? Función no estricta	¿Siempre? Función estricta

Funciones parciales y totales

❏ Considerar

```
succ :: Int -> Int  
succ x = x+1
```

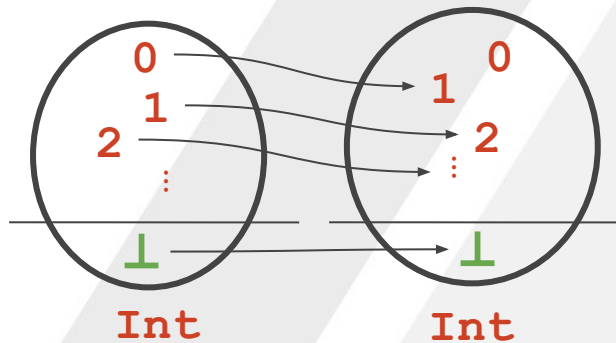
```
detect :: Int -> Bool  
detect 42 = True
```


Funciones parciales y totales

Considerar

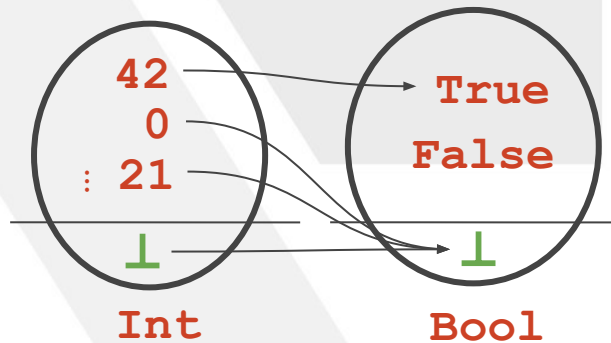
$\text{succ} :: \text{Int} \rightarrow \text{Int}$

$\text{succ } x = x+1$



$\text{detect} :: \text{Int} \rightarrow \text{Bool}$

$\text{detect } 42 = \text{True}$



Funciones parciales y totales

❏ Considerar

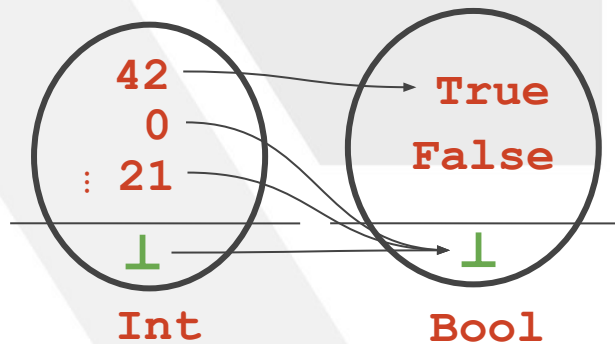
succ :: Int \rightarrow Int
succ x = x+1

❏ Para todo $n \neq \perp$,

succ n $\neq \perp$

❏ **succ** es TOTAL

detect :: Int \rightarrow Bool
detect 42 = True



Funciones parciales y totales

❏ Considerar

succ :: Int -> Int
succ x = x+1

❏ Para todo $n \neq \perp$,

succ n $\neq \perp$

❏ **succ** es TOTAL

detect :: Int -> Bool
detect 42 = True

❏ Existen $n \neq \perp$,

detect n = \perp

❏ **detect** es PARCIAL

Funciones parciales y totales

- Definiciones

- Función total**

- Nunca* da \perp si recibe valores totalmente definidos

- Función parcial**

- Puede dar \perp al recibir valores totalmente definidos

- En matemáticas no hay funciones parciales...

- ¿Por qué? ¿Qué es diferente en este enfoque?

Funciones parciales y totales

❏ ¿Qué diferencia hay entre estas funciones?

```
detect 42 = True
```

```
detectE 42 = True
```

```
detectE x = error ""
```

```
detectB 42 = True
```

```
detectB x = bottom
```

Funciones parciales y totales

❏ ¿Qué diferencia hay entre estas funciones?

`detect 42 = True`

`detectE 42 = True`

`detectB 42 = True`

`detectE x = error ""`

`detectB x = bottom`

❏ `detect 42 = detectE 42 = detectB 42 = True`

Funciones parciales y totales

❏ ¿Qué diferencia hay entre estas funciones?

`detect 42 = True`

`detectE 42 = True`

`detectB 42 = True`

`detectE x = error ""`

`detectB x = bottom`

❏ `detect 42 = detectE 42 = detectB 42 = True`

❏ `detect e = detectE e = detectB e = ⊥, si e ≠ 42`

Funciones parciales y totales

- ¿Qué diferencia hay entre estas funciones?

`detect 42 = True`

`detectE 42 = True`

`detectB 42 = True`

`detectE x = error ""`

`detectB x = bottom`

- `detect 42 = detectE 42 = detectB 42 = True`

- `detect e = detectE e = detectB e = ⊥`, si `e ≠ 42`

- ¡Por lo tanto, `detect = detectE = detectB`!

- ¿Pero cómo es su comportamiento operacional?

Funciones parciales y totales

```
detect  42 = True
detectE 42 = True
detectE x = error ""
detectB 42 = True
detectB x = bottom
```

❏ ¿Cuál es el comportamiento operacional? ($e \neq 42$)

❏ **detect** $e \rightarrow ??$

❏ **detectE** $e \rightarrow ??$

❏ **detectB** $e \rightarrow ??$

Funciones parciales y totales

```
detect  42 = True
detectE 42 = True
detectE x  = error ""
detectB 42 = True
detectB x  = bottom
```

❏ ¿Cuál es el comportamiento operacional? ($e \neq 42$)

❏ `detect e` $\not\rightarrow$

❏ `detectE e` \rightarrow `error ""` $\not\rightarrow$

❏ `detectB e` \rightarrow `bottom` $\rightarrow \dots$

Funciones parciales y totales

```
detect  42 = True  
detectE 42 = True  
detectE x = error ""  
detectB 42 = True  
detectB x = bottom
```

- ❏ ¿Cuál es el comportamiento operacional? ($e \neq 42$)
 - ❏ `detect e` \nrightarrow
 - ❏ `detectE e` \rightarrow `error ""` \nrightarrow
 - ❏ `detectB e` \rightarrow `bottom` $\rightarrow \dots$
- ❏ Los 3 comportamientos operacionales son *diferentes*

Funciones parciales y totales

```
detect  42 = True
detectE 42 = True
detectE x = error ""
detectB 42 = True
detectB x = bottom
```

- ❑ ¿Cuál es el comportamiento operacional? ($e \neq 42$)
 - ❑ `detect e` $\not\rightarrow$
 - ❑ `detectE e` \rightarrow `error ""` $\not\rightarrow$
 - ❑ `detectB e` \rightarrow `bottom` $\rightarrow \dots$
- ❑ Los 3 comportamientos operacionales son *diferentes*
- ❑ Sin embargo, las 3 formas definir denotan la *misma función*

Funciones estrictas y no estrictas

❏ Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

Funciones estrictas y no estrictas

- Considerar la siguiente función

`const :: a -> b -> a`

`const x y = x`

- ¿Cuál es el valor de `const bottom 2`?

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

def. de const,
x <- bottom,
y <- 2

- ¿Cuál es el valor de **const bottom 2**?

const bottom 2 → bottom → ...

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

def. de const,
x <- bottom,
y <- 2

- ¿Cuál es el valor de **const bottom 2**?

const bottom 2 → bottom → ...

- Entonces, **const bottom 2 = ⊥**

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

def. de const,
 $x \leftarrow \text{bottom}$,
 $y \leftarrow 2$

- ¿Cuál es el valor de $\text{const bottom } 2$?

$\text{const bottom } 2$ \rightarrow bottom $\rightarrow \dots$

- Entonces, $\text{const bottom } 2 = \perp$

- El valor de x es necesario para el resultado
 - La función **const** es *estricta* en su 1er argumento

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de **const 2 bottom**?

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de `const 2 bottom`?

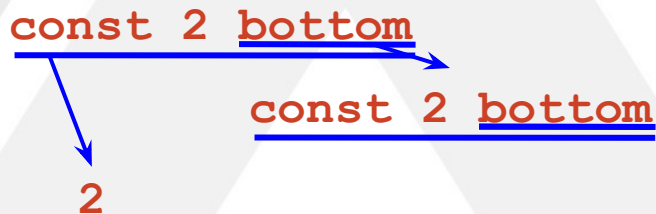
`const 2 bottom`

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de `const 2 bottom`?

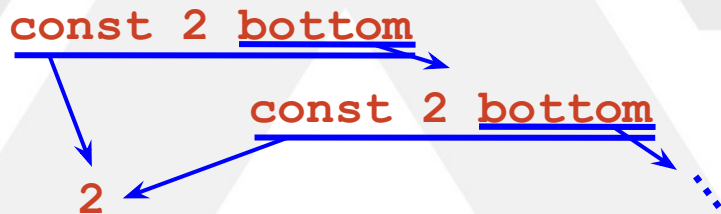


Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de `const 2 bottom`?



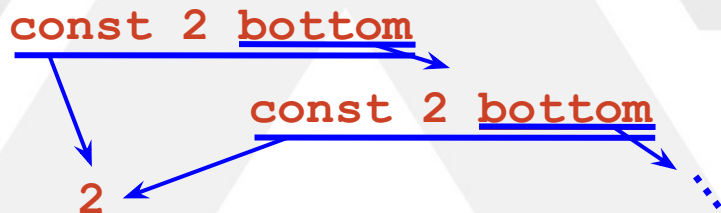
Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a
const x y = x
```

- ¿Y el valor de `const 2 bottom`?

- ¿`const 2 bottom = 2`? ¿`const 2 bottom = ⊥`?



```
const :: a -> b -> a  
const x y = x
```

Funciones estrictas y no estrictas

- Disgresión: ¿cómo sería **const** en C?

```
int const(int x, y) { return y; }  
int bottom() { while true; return 0; }
```

- ¿Cuál es el valor de **const(2, bottom())** en C?
 - ¿**const(2, bottom())** = 2?
 - ¿**const(2, bottom())** = \perp ?

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de `const 2 bottom`?
 - ¿`const 2 bottom = 2`? ¿`const 2 bottom = ⊥`?
 - ¿El valor de `y` es necesario para el resultado?

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Y el valor de `const 2 bottom`?
 - ¿`const 2 bottom = 2`? ¿`const 2 bottom = ⊥`?
 - ¿El valor de `y` es necesario para el resultado?
 - No... ¡Hay dos posibles elecciones!

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a
const x y = x
```

- ¿Y el valor de **const 2 bottom**?
 - ¿**const 2 bottom** = 2? ¿**const 2 bottom** = \perp ?
 - ¿El valor de **y** es necesario para el resultado?
 - No... ¡Hay dos posibles elecciones!
 - ¡Existen DOS funciones **const**, una *estricta* y una *no estricta*!! ¿De qué depende?

Funciones estrictas y no estrictas

- Considerar la siguiente función

```
const :: a -> b -> a  
const x y = x
```

- ¿Existen DOS funciones **const**,
una *estricta* y una *no estricta*!! ¿De qué depende?
- ARMAR UN GRÁFICO DE AMBAS FUNCIONES CONST

Funciones estrictas y no estrictas

- Definiciones

- **Función no estricta**

- Puede dar algo $\neq \perp$ si recibe \perp

- **Función estricta**

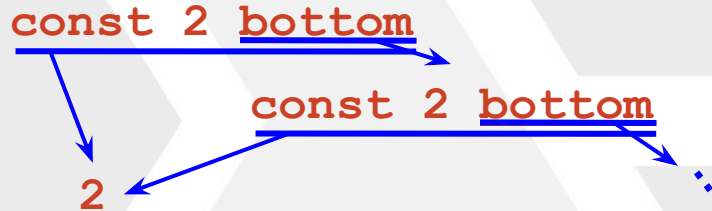
- Da \perp al recibir \perp

- Las funciones que no precisan su argumento tienen DOS versiones: una estricta y otra no estricta

Órdenes de reducción

Órdenes de reducción

- ❑ No da igual reducir cualquier redex



- ❑ ¿Cuál redex elegir si hay más de uno?
 - ❑ ¡Sin decidir cuál redex elegir en caso de haber más de uno, el resultado puede variar!
 - ❑ Da un resultado definido, o da \perp

Órdenes de reducción

- ❑ Orden de reducción
 - ❑ Regla para elegir qué redex reducir
- ❑ Órdenes más comunes
 - ❑ Si están al mismo nivel, elige el de más a la izquierda
 - ❑ Si están uno dentro de otro, hay dos extremos
 - ❑ **Orden APLICATIVO**
 - ❑ **Orden NORMAL**

Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos

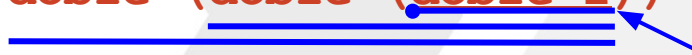
`doble (doble (doble 2))`

Órdenes de reducción

Orden de reducción APLICATIVO

- Se elige el redex más interno de todos

```
double (double (double 2))
```



Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos

`doble (doble (doble 2))`



→

(def. de doble, con $x \leftarrow 2$)

`doble (doble (2+2))`

Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos

`doble (doble (doble 2))`

→

(def. de doble, con $x \leftarrow 2$)

`doble (doble (2+2))`

→

Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos

`doble (doble (doble 2))`

→

(def. de doble, con $x \leftarrow 2$)

`doble (doble (2+2))`

→

(def. de suma)

`doble (doble 4)`

Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos

`double (double (double 2))`



→

(def. de doble, con **x=2**)

`double (double (2+2))`



→

(def. de suma)

`double (double 4)`



→ ...

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

doble (doble (doble 2))

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

`dob1e (dob1e (dob1e 2))`

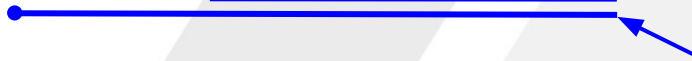


Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

`doble (doble (doble 2))`



(def. de doble, con $x \leftarrow (\text{doble} (\text{doble } 2))$)

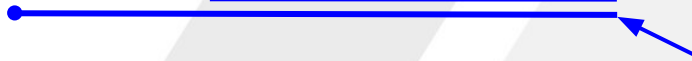
`doble (doble 2) + doble (doble 2)`

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

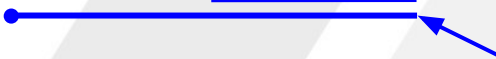
`doble (doble (doble 2))`



→

(def. de doble, con $x \leftarrow (\text{doble (doble 2)})$)

`doble (doble 2) + doble (doble 2)`



→

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

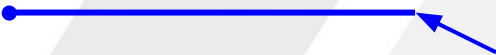
`doble (doble (doble 2))`



→

(def. de doble, con $x \leftarrow$ (doble (doble 2)))

`doble (doble 2) + doble (doble 2)`



→

(def. de doble, con $x \leftarrow$ (doble 2))

`(doble 2 + doble 2) + doble (doble 2)`

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

doble (doble (doble 2))



→ (def. de doble, con $x = (\text{doble } (\text{doble } 2))$)

doble (doble 2) + doble (doble 2)



→ (def. de doble, con $x = (\text{doble } 2)$)

(doble 2 + doble 2) + doble (doble 2)



→ ... 16

Órdenes de reducción

- ❏ Órdenes de reducción más comunes
 - ❏ *Si están al mismo nivel, elige el de más a la **izquierda***
 - ❏ Si están uno dentro de otro, hay dos extremos
 - ❏ **Orden APLICATIVO**
 - ❏ Elige el redex más interno (a la izquierda)
 - ❏ **Orden NORMAL**
 - ❏ Elige el redex más externo (a la izquierda)

Órdenes de reducción

❏ Orden de reducción APLICATIVO

- ❏ Se elige el redex más interno de todos (a la izquierda)

`const 2 (const 2 bottom)`



→ (def. de bottom)

`const 2 (const 2 bottom)`



→ ...

Órdenes de reducción

❏ Orden de reducción NORMAL

- ❏ Se elige el redex más externo de todos (a la izquierda)

`const 2 (const 2 bottom)`



→

(def. de const, con **x** <- 2,

y <- `const 2 bottom`)

2

Órdenes de reducción

❏ Órdenes de reducción y bottom

- ❏ Al reducir **(const 2 bottom)** con orden aplicativo da **⊥**
- ❏ Al reducir **(const 2 bottom)** con orden normal da **2**

Órdenes de reducción

❏ Órdenes de reducción y bottom

- ❏ Al reducir **(const 2 bottom)** con orden aplicativo da **⊥**
- ❏ Al reducir **(const 2 bottom)** con orden normal da **2**
- ❏ ¿Cuál orden encuentra la forma normal si existe?

Órdenes de reducción

❏ Órdenes de reducción y bottom

- ❏ Al reducir **(const 2 bottom)** con orden aplicativo da **⊥**
- ❏ Al reducir **(const 2 bottom)** con orden normal da **2**
- ❏ ¿Cuál orden encuentra la forma normal si existe?
 - ❏ El orden *normal* (de ahí su nombre)

Órdenes de reducción

❏ Órdenes de reducción y bottom

- ❏ Al reducir **(const 2 bottom)** con orden aplicativo da **⊥**
- ❏ Al reducir **(const 2 bottom)** con orden normal da **2**
- ❏ ¿Cuál orden encuentra la forma normal si existe?
 - ❏ El orden *normal* (de ahí su nombre)
- ❏ ¿Y por qué el nombre de *aplicativo* para el otro?

Órdenes de reducción

❏ Órdenes de reducción y bottom

- ❏ Al reducir `(const 2 bottom)` con orden aplicativo da \perp
- ❏ Al reducir `(const 2 bottom)` con orden normal da `2`
- ❏ ¿Cuál orden encuentra la forma normal si existe?
 - ❏ El orden *normal* (de ahí su nombre)
- ❏ ¿Y por qué el nombre de *aplicativo* para el otro?
 - ❏ Porque reduce el argumento antes de *aplicar* la función

Órdenes de reducción

- ❏ Órdenes de reducción y funciones no-estrictas
 - ❏ ¿Se observa la relación entre el orden de reducción y el hecho de que una función sea estricta o no?

Órdenes de reducción

- ❏ Órdenes de reducción y funciones no-estrictas
 - ❏ ¿Se observa la relación entre el orden de reducción y el hecho de que una función sea estricta o no?
 - ❏ Usar *orden aplicativo* para reducir implica que todas las funciones son *estrictas*

Órdenes de reducción

- ❑ Órdenes de reducción y funciones no-estrictas
 - ❑ ¿Se observa la relación entre el orden de reducción y el hecho de que una función sea estricta o no?
 - ❑ Usar *orden aplicativo* para reducir implica que todas las funciones son *estrictas*
 - ❑ Usar *orden normal* para reducir implicar que existen funciones *no estrictas*
 - ❑ Las que no precisan su argumento

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?
 - Con orden normal no se puede predecir el orden exacto en que se ejecutarán los redexes
 - Por lo tanto no es posible saber los efectos laterales que se ejecutarán

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

```
int const(int x, y) { return x; }  
int printSucc(int x)  
    { printf("Sumé 1"); return (x+1); }
```

- ¿Qué imprimiría `const(2, printSucc(3))`?

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

```
int const(int x, y) { return x; }  
int printSucc(int x)  
    { printf("Sumé 1"); return (x+1); }
```

- ¿Qué imprimiría `const(2, printSucc(3))`?
 - Con orden aplicativo, "Sumé 1"
 - Con orden normal, nada

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

```
int const(int x, y) { return x; }  
int printSucc(int x)  
    { printf("Sumé 1"); return (x+1); }
```

- ¿Qué imprimiría `f(2, printSucc(3))`?
 - Con orden aplicativo, "Sumé 1"
 - Con orden normal, no puedo saber, sin saber quién es `f`

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

```
int const(int x, y) { return x; }  
int infinito() { return (infinito()+1); }
```

- ¿Qué da `const(2, infinito())`?

Órdenes de reducción

- En un lenguaje con efectos laterales
ES NECESARIO usar orden aplicativo. ¿Por qué?

```
int const(int x, y) { return x; }  
int infinito() { return (infinito()+1); }
```

- ¿Qué da `const(2, infinito())`?
- En C *todas las funciones son estrictas*
 - ¡Si una parte falla, TODO falla, aunque no se use!

Órdenes de reducción

- En un lenguaje *puro* SE PUEDE ELEGIR qué orden usar
 - Es una decisión de diseño
tener funciones estrictas y no estrictas en el lenguaje
 - Distintos lenguajes puros eligen diferente
 - **Haskell** elige orden normal
 - **Clean** elige orden aplicativo

R. John M. Hughes



R. John M. Hughes

(15 de julio 1958 – ...) es un científico de la computación, profesor de la Universidad de Tecnología de Chalmers, Gotemburgo, Suecia, y cofundador y CEO de QuviQ AB, empresa que ofrece una versión comercial de QuickCheck, de la que es también uno de los autores.

En 1984 recibió su doctorado de la Universidad de Oxford con la tesis *"The design and implementation of Programming Languages"*. Miembro del grupo de Programación Funcional en Chalmers, es uno de los miembros del comité que desarrolló el lenguaje Haskell. Dirigió numerosos doctorados de personalidades de PF, y de algún otro que no tanto.

Simon L. Peyton Jones



Simon L. Peyton Jones

(18 de enero 1958 – ...) es un científico de la computación británico, que investiga la implementación y aplicaciones de lenguajes funcionales, en particular lenguajes no estrictos. En 1980 se graduó en la Universidad de Cambridge (Trinity College) y enseñó en el University College de Londres y en la Universidad de Glasgow en Inglaterra. Desde 1998 trabaja para el grupo de investigación de Microsoft Research en Cambridge, Inglaterra. Es uno de los miembros del comité que desarrolló el lenguaje Haskell, y uno de sus principales desarrolladores, siendo el diseñador principal del Glasgow Haskell Compiler (GHC) y autor de numerosísimos trabajos en el área.

Evaluación lazy

- ❑ ¿Cuántas reducciones implica cada orden?

quin x = x+x+x+x+x

- ❑ Supongamos que **(fib 22)** precisa ~1 millón de reducciones en cualquiera de los órdenes
- ❑ ¿Cuántas reducciones precisa **quin (fib 22)**?
- ❑ ¿Y **const 2 (fib 22)**?
- ❑ ¡Depende del orden!

Evaluación lazy

❏ ¿Cuántas reducciones implica cada orden?

```
quin x = x+x+x+x+x
```

Orden aplicativo

```
quin (fib 22)
```

→ ...

Evaluación lazy

❏ ¿Cuántas reducciones implica cada orden?

```
quin x = x+x+x+x+x
```

Orden aplicativo

```
quin (fib 22)
```

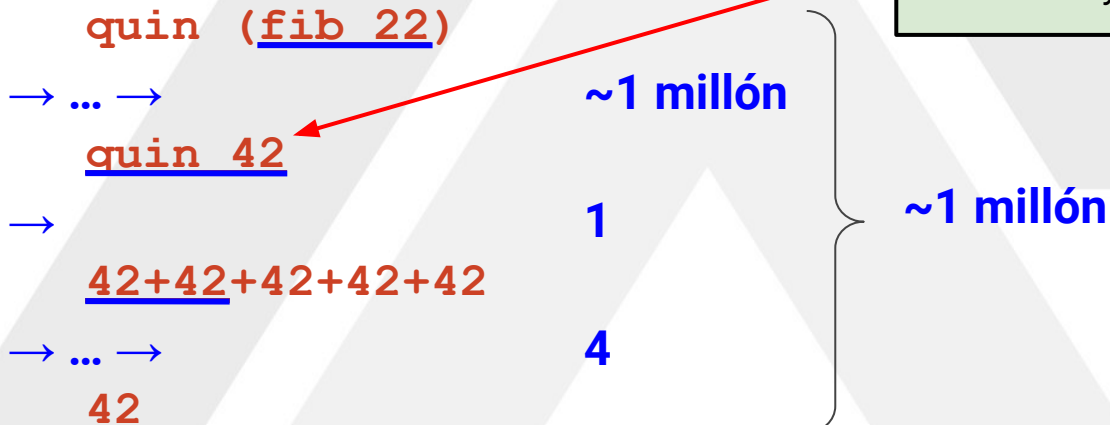
→ ...

Evaluación lazy

- ¿Cuántas reducciones implica cada orden?

quin x = x+x+x+x+x

Orden aplicativo



42 es la respuesta a la vida,
el universo y todo lo demás

Evaluación lazy

❏ ¿Cuántas reducciones implica cada orden?

quin x = x+x+x+x+x

Orden normal

quin (fib 22)

→

Evaluación lazy

- ❏ ¿Cuántas reducciones implica cada orden?

quin x = x+x+x+x+x

Orden normal

quin (fib 22)

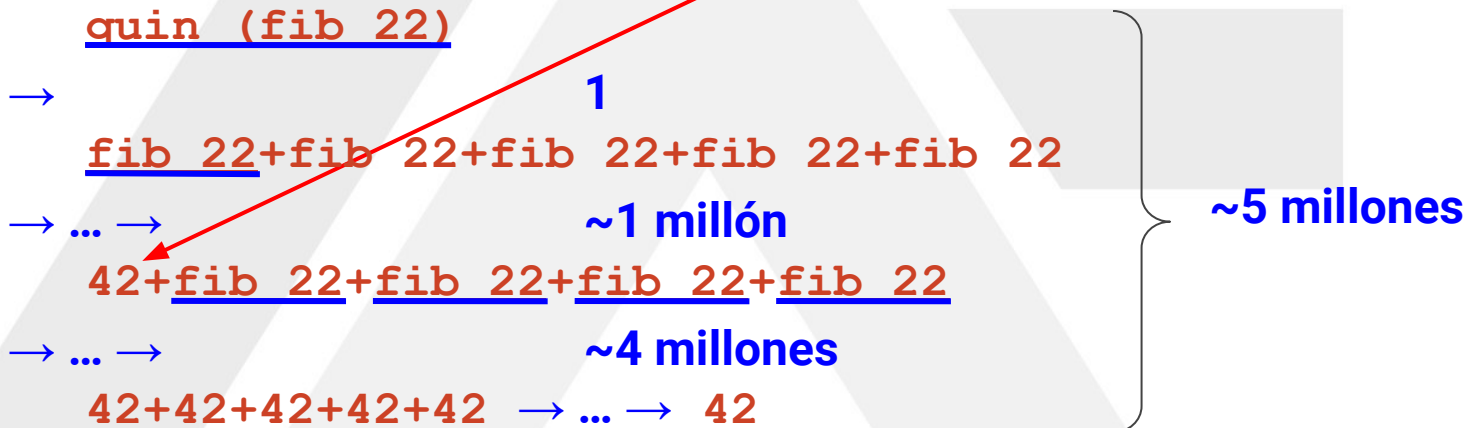
→

Evaluación lazy

- ¿Cuántas reducciones implica cada orden?

quin x = x+x+x+x+x

Orden normal



Evaluación lazy

- ¿Cuántas reducciones implica cada orden?
 - ¿Cuántas reducciones precisa `quin (fib 22)`?
- ¿Y `const 2 (fib 22)`?

Evaluación lazy

- ❑ ¿Cuántas reducciones implica cada orden?
 - ❑ ¿Cuántas reducciones precisa `quin (fib 22)`?
 - ❑ Con orden aplicativo:
 - ❑ Con orden normal:
 - ❑ ¿Y `const 2 (fib 22)`?
 - ❑ Con orden aplicativo:
 - ❑ Con orden normal:

Evaluación lazy

- ❑ ¿Cuántas reducciones implica cada orden?
 - ❑ ¿Cuántas reducciones precisa **quin (fib 22)**?
 - ❑ Con orden aplicativo: ~1 millón de reducciones
 - ❑ Con orden normal: ~5 millones de reducciones
 - ❑ ¿Y **const 2 (fib 22)**?
 - ❑ Con orden aplicativo: ~1 millón de reducciones
 - ❑ Con orden normal: 1 (una, solamente UNA)

Evaluación lazy

- ❑ ¿Cuántas reducciones implica cada orden?
 - ❑ ¿Cuántas reducciones precisa **quin (fib 22)**?
 - ❑ Con orden aplicativo: ~1 millón de reducciones
 - ❑ Con orden normal: ~5 millones de reducciones
 - ❑ ¿Y **const 2 (fib 22)**?
 - ❑ Con orden aplicativo: ~1 millón de reducciones
 - ❑ Con orden normal: 1 (una, solamente UNA)
- ❑ ¿Cuál orden es mejor?

Evaluación lazy

- ❏ ¿Cuál orden es mejor?
 - ❏ Ambos a veces llevan menos y a veces más
 - ❏ El orden normal permite tener más terminación

Evaluación lazy

- ❑ ¿Cuál orden es mejor?
 - ❑ Ambos a veces llevan menos y a veces más
 - ❑ El orden normal permite tener más terminación
 - ❑ ¿Podremos implementar el orden normal para que sea en el peor caso tan eficiente como el aplicativo?

Evaluación lazy

- ❑ ¿Cuál orden es mejor?
 - ❑ Ambos a veces llevan menos y a veces más
 - ❑ El orden normal permite tener más terminación
 - ❑ ¿Podremos implementar el orden normal para que sea en el peor caso tan eficiente como el aplicativo?
 - ❑ Orden de *reducción perezoso* (**lazy evaluation**)

Evaluación lazy

- ❏ Orden de *reducción perezoso* (***lazy evaluation***)
 - ❏ Orden normal, pero...

Evaluación lazy

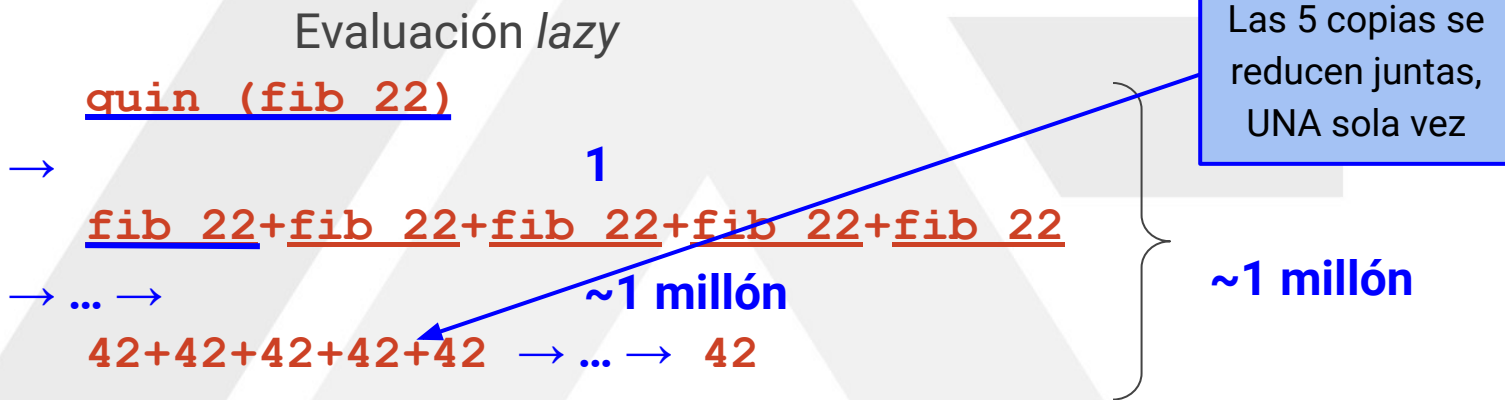
- ❑ Orden de *reducción perezoso* (***lazy evaluation***)
 - ❑ Orden normal, pero...
 - ❑ Se recuerda que las copias de un argumento son idénticas, y se reducen a lo sumo UNA vez
 - ❑ Un argumento no necesariamente se reduce por completo
 - ❑ Solamente las partes que se precisan son evaluadas

Evaluación lazy

- ❑ Orden de *reducción perezoso* (***lazy evaluation***)
 - ❑ Las copias de un argumento se reducen a lo sumo UNA vez

Evaluación lazy

- ❑ Orden de *reducción perezoso* (**lazy evaluation**)
 - ❑ Las copias de un argumento se reducen a lo sumo UNA vez



Evaluación lazy

- Orden de *reducción perezoso* (***lazy evaluation***)
 - Un argumento no necesariamente se reduce por completo

Evaluación lazy

- ❑ Orden de *reducción perezoso* (***lazy evaluation***)
 - ❑ Un argumento no necesariamente se reduce por completo

```
test = (3, quin (fib 22))
```

Evaluación lazy

- ❑ Orden de *reducción perezoso* (***lazy evaluation***)
 - ❑ Un argumento no necesariamente se reduce por completo

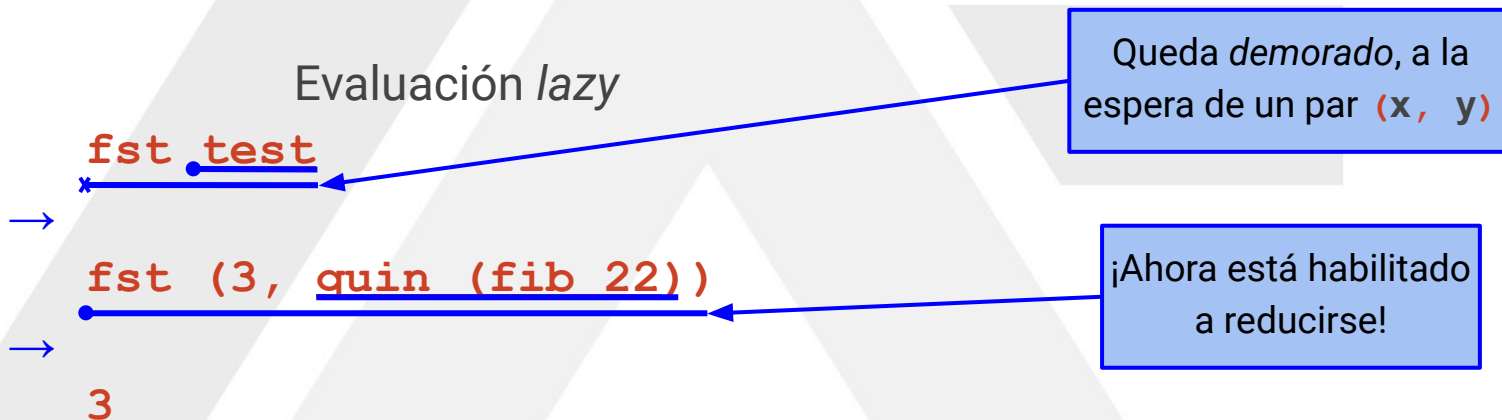
```
test = (3, quin (fib 22))
```

```
fst test
```

Evaluación lazy

- ❑ Orden de *reducción perezoso* (**lazy evaluation**)
 - ❑ Un argumento no necesariamente se reduce por completo

```
test = (3, quin (fib 22))
```



Evaluación lazy

- ❑ Ventajas de la *evaluación lazy*
 - ❑ Usualmente mayor eficiencia
 - ❑ Mejores condiciones de terminación
 - ❑ No hay necesidad de estructuras intermedias al componer funciones
 - ❑ Se pueden manipular estructuras y cálculos infinitos
- ❑ Desventajas de la *evaluación lazy*
 - ❑ Es MUY difícil calcular el costo de ejecución (pues depende del contexto en que se usa...)



Resumen

Resumen

- ❏ Miramos de más cerca el mecanismo de reducción
- ❏ Vimos propiedades
 - ❏ Confluencia, SÍ
 - ❏ Normalización, NO. Bottom, \perp
 - ❏ Funciones parciales y totales
 - ❏ Funciones estrictas y no estrictas
 - ❏ Órdenes de reducción
 - ❏ Aplicativo y normal
 - ❏ *Evaluación lazy*