Igualdades por definición

Principio de reemplazo

Sea `e1 = e2` una ecuación incluida en el programa.
Las siguientes operaciones preservan la igualdad de expresiones:

1. Reemplazar cualquier instancia de `e1` por `e2`.
2. Reemplazar cualquier instancia de `e2` por `e1`.

Si una igualdad se puede demostrar usando sólo el principio de reemplazo, decimos que la igualdad vale **por definición**.

Ejemplo: principio de reemplazo

```
{L0}  length []       = 0
{L1}  length (_ : xs) = 1 + length xs
{S0}  suma []         = 0
{S1}  suma (x : xs)   = x + suma xs
```

Veamos que `length ["a", "b"] = suma [1, 1]`:

```
    length ["a", "b"]
=   1 + length ["b"]      por L1
=   1 + (1 + length [])   por L1
=   1 + (1 + 0)           por L0
=   1 + (1 + suma [])     por S0
=   1 + suma [1]          por S1
=   suma [1, 1]           por S1
```

Inducción sobre booleanos

Principio de inducción sobre booleanos

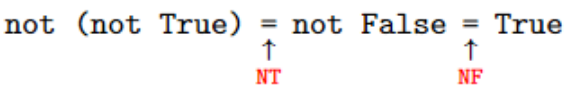
Si $\mathcal{P}$(`True`) y $\mathcal{P}$(`False`) entonces $\forall$`x :: Bool.` $\mathcal{P}$(`x`).

Ejemplo

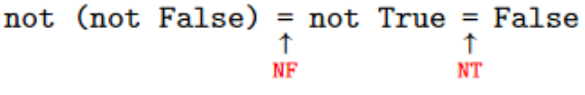
```
{NT} not True  = False
{NF} not False = True
```

Para probar `∀x :: Bool. not (not x) = x`
basta probar:

1. `not (not True) = True`.



2. `not (not False) = False`.



Inducción sobre pares

Principio de inducción sobre pares

Si $\forall$ `x :: a. ∀y :: b.` $\mathcal{P}$`((x, y))`
entonces $\forall$ `p :: (a, b).` $\mathcal{P}$`(p)`.

Ejemplo

```
{FST}  fst (x, _)  = x
{SND}  snd (_, y)  = y
{SWAP} swap (x, y) = (y, x)
```

Para probar `∀p :: (a, b). fst p = snd (swap p)`
basta probar:

- `∀x :: a. ∀y :: b. fst (x, y) = snd (swap (x, y))`

```
fst (x, y) = x = snd (y, x) = snd (swap (x, y))
             ↑   ↑           ↑
            FST SND         SWAP
```

Inducción sobre naturales

```
data Nat = Zero | Suc Nat
```

Principio de inducción sobre naturales

Si $\mathcal{P}$`(Zero)` y `∀n :: Nat. (` $\underbrace{\mathcal{P}(n)}_{\text{hipótesis inductiva}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suc n})}_{\text{tesis inductiva}}$ `)`,
entonces `∀n :: Nat.` $\mathcal{P}$`(n)`.

Ejemplo

```
{S0} suma Zero    m = m
{S1} suma (Suc n) m = Suc (suma n) m
```

Para probar `∀n :: Nat. suma n Zero = n`
basta probar:

1. `suma Zero Zero = Zero`.
   Inmediato por S0.
2. $\underbrace{\texttt{suma n Zero = n}}_{\text{H.I.}} \Rightarrow \underbrace{\texttt{suma (Suc n) Zero = Suc n}}_{\text{T.I.}}$.

```
suma (Suc n) Zero = Suc (suma n Zero) = Suc n
                    ↑                   ↑
                   S1                  H.I.
```

Inducción estructural

En el **caso general**, tenemos un tipo de datos inductivo:

$$\begin{aligned}\texttt{data T} \;=\; & \texttt{CBase}_1\ \langle \textit{parámetros} \rangle \\ & \ldots \\ \mid\; & \texttt{CBase}_n\ \langle \textit{parámetros} \rangle \\ \mid\; & \texttt{CRecursivo}_1\ \langle \textit{parámetros} \rangle \\ & \ldots \\ \mid\; & \texttt{CRecursivo}_m\ \langle \textit{parámetros} \rangle\end{aligned}$$

Principio de inducción estructural

Sea $\mathcal{P}$ una propiedad acerca de las expresiones tipo `T` tal que:

- $\mathcal{P}$ vale sobre todos los constructores base de `T`,
- $\mathcal{P}$ vale sobre todos los constructores recursivos de `T`, asumiendo como hipótesis inductiva que vale para los parámetros de tipo `T`,

entonces $\forall \texttt{x} :: \texttt{T}.\ \ \mathcal{P}(\texttt{x})$.

Ejemplo: principio de inducción sobre árboles binarios

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `AB a` tal que:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{i} :: \texttt{AB a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{d} :: \texttt{AB a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{i}) \wedge \mathcal{P}(\texttt{d}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin i r d})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{AB a}.\ \ \mathcal{P}(\texttt{x})$.

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `Poli a` tal que:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \texttt{a}.\ \ \mathcal{P}(\texttt{Cte k})$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suma p q})}_{\text{T.I.}})$$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{Poli a}.\ \ \mathcal{P}(\texttt{x})$.

Ejemplo: principio de inducción sobre listas

`data [a] = [] | a : [a]`

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `[a]` tal que:

- $\mathcal{P}(\texttt{[]})$
- $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\underbrace{\mathcal{P}(\texttt{xs})}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{x : xs})}_{\text{T.I.}})$

Entonces $\forall \texttt{xs} :: \texttt{[a]}.\ \mathcal{P}(\texttt{xs})$.

```
{M0} map f []       = []
{M1} map f (x : xs) = f x : map f xs
{A0} []       ++ ys = ys
{A1} (x : xs) ++ ys = x : (xs ++ ys)
```

Propiedad. Si `f :: a -> b`, `xs :: [a]`, `ys :: [a]`, entonces:

```
map f (xs ++ ys) = map f xs ++ map f ys
```

Por inducción en la estructura de `xs`, basta ver:

1. Caso base, $\mathcal{P}(\texttt{[]})$.
2. Caso inductivo, $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\mathcal{P}(\texttt{xs}) \Rightarrow \mathcal{P}(\texttt{x : xs}))$.

con $\mathcal{P}(\texttt{xs}) :\equiv$ (`map f (xs ++ ys) = map f xs ++ map f ys`).

Caso base:

```
    map f ([] ++ ys)
=   map f ys                 por A0
=   [] ++ map f ys           por A0
=   map f [] ++ map f ys     por M0
```

Caso inductivo:

```
    map f ((x : xs) ++ ys)
=   map f (x : (xs ++ ys))          por A1
=   f x : map f (xs ++ ys)          por M1
=   f x : (map f xs ++ map f ys)    por H.I.
=   (f x : map f xs) ++ map f ys    por A1
=   map f (x : xs) ++ map f ys      por M1
```

Extensionalidad

Usando el principio de inducción estructural, se puede probar:

Extensionalidad para pares

Si `p :: (a, b)`, entonces $\exists$`x :: a. ` $\exists$`y :: b. p = (x, y)`.

```
data Either a b = Left a | Right b
```

Extensionalidad para sumas

Si `e :: Either a b`, entonces:

- o bien $\exists$`x :: a. e = Left x`
- o bien $\exists$`y :: b. e = Right y`

Principio de extensionalidad funcional

Sean `f, g :: a -> b`.

Propiedad inmediata

Si `f = g` entonces ($\forall$`x :: a. f x = g x`).

Principio de extensionalidad funcional

Si ($\forall$`x :: a. f x = g x`) entonces `f = g`.

Usando el principio de inducción estructural, se puede probar:

Extensionalidad para pares

Si `p :: (a, b)`, entonces $\exists$`x :: a. ` $\exists$`y :: b. p = (x, y)`.

```
data Either a b = Left a | Right b
```

Extensionalidad para sumas

Si `e :: Either a b`, entonces:

- o bien $\exists$`x :: a. e = Left x`
- o bien $\exists$`y :: b. e = Right y`

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

```
∀F :: a -> b . ∀G :: a -> b . ∀Y :: b . ∀Z :: a .
```

`F = G`	$\Leftrightarrow$	`∀x::a . F x = G x`
`F = \x -> Y`	$\Leftrightarrow$	`∀x::a . F x = Y`
`(\x -> Y) Z`	$=_\beta$	`Y` reemplazando `x` por `Z`
`\x -> F x`	$=_\eta$	`F`

`F`, `G`, `Y` y `Z` pueden ser expresiones complejas, siempre que la variable `x` no aparezca libre en `F`, `G`, ni `Z` (más detalles cuando veamos Cálculo Lambda).

Funciones Haskell

- **foldr**
  - **Descripción**: Aplica una función a cada elemento de una lista, desde el final hacia el principio, acumulando un resultado.
  - **Definición**: foldr :: (a -> b -> b) -> b -> [a] -> b
  - **Implementación**:
    - foldr _ z []   = z
    - foldr f z (x:xs) = f x (foldr f z xs)
  - **Ejemplo**:

```
foldr (+) 0 [1, 2, 3] -- Resultado: 6
```

- **foldl**
  - **Descripción**: Aplica una función a cada elemento de una lista, desde el principio hacia el final, acumulando un resultado.
  - **Definición**: foldl :: (b -> a -> b) -> b -> [a] -> b
  - **Implementación**:
    - foldl _ z []   = z
    - foldl f z (x:xs) = foldl f (f z x) xs
  - **Ejemplo**:

```
foldl (+) 0 [1, 2, 3] -- Resultado: 6
```

- **recr**
  - **Definición**:
    - recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
    - recr f z [] = z
    - recr f z (x : xs) = f x xs (recr f z xs)

```
recr (\x xs' acc -> (length xs - length xs') 0 [1,2,3,4] -- Resultado: 10
```

- **foldr1**
  - **Descripción**: Similar a `foldr`, pero no necesita un valor inicial; usa el último elemento de la lista como el acumulador inicial.
  - **Ejemplo**:

```
foldr1 (+) [1, 2, 3] -- Resultado: 6
```

- **foldl1**
  - **Descripción**: Similar a `foldl`, pero no necesita un valor inicial; usa el primer elemento de la lista como el acumulador inicial.
  - **Ejemplo**:

```
foldl1 (+) [1, 2, 3] -- Resultado: 6
```

- **map**
  - **Descripción**: Aplica una función a cada elemento de una lista y retorna una nueva lista con los resultados.
  - **Ejemplo**:

```
map (*2) [1, 2, 3] -- Resultado: [2, 4, 6]
```

- **zipWith**
  - **Descripción**: Combina dos listas aplicando una función a pares de elementos correspondientes.
  - **Definición**:
    - zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
    - zipwith f = foldr (\x rec ->
    - (\ys -> if null ys then [] else
    - (f x (head ys)) : (rec (tail ys)))) []
  - **Ejemplo**:

```
zipWith (+) [1, 2, 3] [4, 5, 6] -- Resultado: [5, 7, 9]
```

- **all**
  - **Descripción**: Retorna `True` si todos los elementos de la lista satisfacen un predicado.
  - **Ejemplo**:

```
all even [2, 4, 6] -- Resultado: True
```

- **any**
  - **Descripción**: Retorna `True` si al menos un elemento de la lista satisface un predicado.
  - **Ejemplo**:

```
any even [1, 3, 4] -- Resultado: True
```

- **null**
  - **Descripción**: Verifica si una lista está vacía.
  - **Ejemplo**:

```
null [] -- Resultado: True
```

- **nub**
  - **Descripción**: Elimina los elementos duplicados de una lista.
  - **Ejemplo**:

```
nub [1, 2, 2, 3, 3] -- Resultado: [1, 2, 3]
```

- **sort**
  - **Descripción**: Ordena los elementos de una lista.
  - **Ejemplo**:

```
sort [3, 1, 2] -- Resultado: [1, 2, 3]
```

- **sortBy**
  - **Descripción**: Ordena los elementos de una lista usando una función de comparación.
  - **Ejemplo**:

```
import Data.List (sortBy)
import Data.Ord (comparing)
sortBy (comparing length) ["abc", "a", "ab"] -- Resultado: ["a", "ab",
"abc"]
```

- **mod**
    - **Descripción**: Calcula el resto de la división entre dos números.
    - **Ejemplo**:

```
10 `mod` 3 -- Resultado: 1
```

- **odd**
    - **Descripción**: Verifica si un número es impar.
    - **Ejemplo**:

```
odd 3 -- Resultado: True
```

- **even**
    - **Descripción**: Verifica si un número es par.
    - **Ejemplo**:

```
even 4 -- Resultado: True
```

- **(++)**
    - **Descripción**: Concatena dos listas.
    - **Ejemplo**:

```
[1, 2] ++ [3, 4] -- Resultado: [1, 2, 3, 4]
```

- **head**
    - **Descripción**: Retorna el primer elemento de una lista.
    - **Ejemplo**:

```
head [1, 2, 3] -- Resultado: 1
```

- **tail**
    - **Descripción**: Retorna una lista sin el primer elemento.
    - **Ejemplo**:

```
tail [1, 2, 3] -- Resultado: [2, 3]
```

- **init**
    - **Descripción**: Retorna una lista sin el último elemento.
    - **Ejemplo**:

```
init [1, 2, 3] -- Resultado: [1, 2]
```

- **last**
    - **Descripción**: Retorna el último elemento de una lista.
    - **Ejemplo**:

```
last [1, 2, 3] -- Resultado: 3
```

- **length**
    - **Descripción**: Retorna la longitud de una lista.
    - **Ejemplo**:

```
length [1, 2, 3] -- Resultado: 3
```

- **replicate**
    - **Descripción**: Crea una lista repitiendo un elemento un número específico de veces.
    - **Ejemplo**:

```
replicate 3 0 -- Resultado: [0, 0, 0]
```

- **repeat**
  - **Descripción**: Genera una lista infinita repitiendo un elemento.
  - **Ejemplo**:

```
take 3 (repeat 1) -- Resultado: [1, 1, 1]
```

- **iterate**
  - **Descripción**: Genera una lista infinita aplicando repetidamente una función a un valor inicial.
  - **Ejemplo**:

```
take 3 (iterate (+1) 0) -- Resultado: [0, 1, 2]
```

- **filter**
  - **Descripción**: Filtra los elementos de una lista que satisfacen un predicado.
  - **Ejemplo**:

```
filter even [1, 2, 3, 4] -- Resultado: [2, 4]
```

- **take**
  - **Descripción**: Toma los primeros n elementos de una lista.
  - **Ejemplo**:

```
take 2 [1, 2, 3, 4] -- Resultado: [1, 2]
```

- **drop**
  - **Descripción**: Elimina los primeros n elementos de una lista.
  - **Ejemplo**:

```
drop 2 [1, 2, 3, 4] -- Resultado: [3, 4]
```

- **elem**
  - **Descripción**: Verifica si un elemento está en una lista.
  - **Ejemplo**:

```
3 `elem` [1, 2, 3] -- Resultado: True
```

- **find**
  - **Descripción**: Busca el primer elemento de una lista que satisface un predicado.
  - **Ejemplo**:

```
import Data.Maybe (fromJust)
fromJust (find even [1, 2, 3, 4]) -- Resultado: 2
```

- **isNothing**
  - **Descripción**: Verifica si un valor es `Nothing`.
  - **Ejemplo**:

```
import Data.Maybe (isNothing)
isNothing Nothing -- Resultado: True
```

- **fromJust**
  - **Descripción**: Extrae el valor de un `Maybe` que es `Just`.
  - **Ejemplo**:

```
import Data.Maybe (fromJust)
```

```
fromJust (Just 3) -- Resultado: 3
```

- **maybe**
  - **Descripción**: Desempaqueta un `Maybe` con un valor por defecto.
  - **Ejemplo**:

```
maybe 0 (+1) (Just 3) -- Resultado: 4
```

- **lookup**
  - **Descripción**: Busca un valor en una lista de pares clave-valor.
  - **Ejemplo**:

```
lookup 2 [(1, "a"), (2, "b")] -- Resultado: Just "b"
```

- **reverse**
  - **Descripción**: Invierte una lista.
  - **Ejemplo**:

```
reverse [1, 2, 3] -- Resultado: [3, 2, 1]
```

- **concat**
  - **Descripción**: Concatena una lista de listas en una sola lista.
  - **Ejemplo**:

```
concat [[1, 2], [3, 4]] -- Resultado: [1, 2, 3, 4]
```

- **union**
  - **Descripción**: Retorna la unión de dos listas, eliminando duplicados.
  - **Ejemplo**:

```
union [1, 2] [2, 3] -- Resultado: [1, 2, 3]
```

- **intersect**
  - **Descripción**: Retorna la intersección de dos listas.
  - **Ejemplo**:

```
intersect [1, 2, 3] [2, 3, 4] -- Resultado: [2, 3]
```

- **(>>=)**
  - **Descripción**: Operador de encadenamiento para monadas (bind).
  - **Ejemplo**:

```
Just 3 >>= \x -> Just (x + 1) -- Resultado: Just 4
```

- **span**
  - **Descripción**: Divide una lista en dos partes: los elementos que cumplen un predicado y los que no.
  - **Ejemplo**:

```
span (<3) [1, 2, 3, 4] -- Resultado: ([1, 2], [3, 4])
```

- **takeWhile**
  - **Descripción**: Toma elementos de una lista mientras se cumpla un predicado.
  - **Ejemplo**:

```
takeWhile (<3) [1, 2, 3, 4] -- Resultado: [1, 2]
```

- **dropWhile**
  - **Descripción**: Elimina elementos de una lista mientras se cumpla un predicado.
  - **Ejemplo**:

```
dropWhile (<3) [1, 2, 3, 4] -- Resultado: [3, 4]
```

- **concatMap**
  - **Descripción**: Aplica una función a cada elemento de una lista y concatena los resultados.
  - **Ejemplo**:

```
concatMap (\x -> [x, x]) [1, 2] -- Resultado: [1, 1, 2, 2]
```

- **sum**
  - **Descripción**: Suma los elementos de una lista.
  - **Ejemplo**:

```
sum [1, 2, 3] -- Resultado: 6
```

- **max**
  - **Descripción**: Retorna el mayor de dos valores.
  - **Ejemplo**:

```
max 3 4 -- Resultado: 4
```

- **maximum**
  - **Descripción**: Retorna el mayor valor de una lista.
  - **Ejemplo**:

```
maximum [1, 2, 3] -- Resultado: 3
```

  -
- **maximumBy**
  - **Descripción**: Retorna el mayor valor de una lista según una función de comparación.
  - **Ejemplo:**

```
import Data.Ord (comparing)
maximumBy (comparing length) ["a", "ab", "abc"] -- Resultado: "abc"
```

- **min**
  - **Descripción**: Retorna el menor de dos valores.
  - **Ejemplo**:

```
min 3 4 -- Resultado: 3
```

  -
- **minimum**
  - **Descripción**: Retorna el menor valor de una lista.
  - **Ejemplo**:

```
minimum [1, 2, 3] -- Resultado: 1
```

- **minimumBy**
  - **Descripción**: Retorna el menor valor de una lista según una función de comparación.
  - **Ejemplo**:

```
import Data.Ord (comparing)
minimumBy (comparing length) ["a", "ab", "abc"] -- Resultado: "a"
```

- **compare**
    - **Descripción**: Compara dos valores y retorna `LT`, `EQ` o `GT`.
    - **Ejemplo**:

```
compare 3 4 -- Resultado: LT
```

- **comparing**
    - **Descripción**: Crea una función de comparación basada en una función transformadora.
    - **Ejemplo**:

```
import Data.Ord (comparing)
comparing length "abc" "a" -- Resultado: GT
```

- **ord**
    - **Descripción**: Retorna el valor Unicode de un carácter.
    - **Ejemplo**:

```
import Data.Char (ord)
ord 'A' -- Resultado: 65
```

- **chr**
    - **Descripción**: Retorna el carácter correspondiente a un valor Unicode.
    - **Ejemplo**:

```
import Data.Char (char)
chr 65 -- Resultado: ‘A’
```