Paradigmas de Programación

Razonamiento ecuacional
Inducción estructural

1er cuatrimestre de 2025

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Introducción

Inducción estructural

Extensionalidad

Isomorfismos de tipos

Casos de estudio

Motivación

Queremos demostrar que ciertas expresiones son equivalentes.
¿Para qué?

Para justificar que un algoritmo es correcto

Por ejemplo, si logramos demostrar que:

$$\forall \texttt{xs} :: \texttt{[Int]}.\ \texttt{quickSort xs = insertionSort xs}$$

esto nos da confianza relativa de un algoritmo con respecto al otro.

Para posibilitar optimizaciones

¿Siempre es correcto hacer las siguientes optimizaciones?

$$\begin{array}{lcl} \texttt{f x + f x} & \rightsquigarrow & \texttt{2 * f x} \\ \texttt{map f (map g xs)} & \rightsquigarrow & \texttt{map (f . g) xs} \end{array}$$

En un lenguaje funcional sí.
En un lenguaje imperativo **no**, ya que `f` y `g` pueden tener efectos.

Hipótesis de trabajo

Para razonar sobre equivalencia de expresiones vamos a asumir:

1. Que trabajamos con estructuras de datos **finitas**.

   Técnicamente: con tipos de datos **inductivos**.

2. Que trabajamos con **funciones totales**.
   - Las ecuaciones deben cubrir todos los casos.
   - La recursión siempre debe terminar.

3. Que el programa **no depende del orden** de las ecuaciones.

```
vacia [] = True            vacia []      = True
vacia _  = False     ⇝     vacia (_ : _) = False
```

Relajar estas hipótesis es posible pero más complejo.

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

Le damos nombre a las ecuaciones del programa:

```
      sucesor :: Int -> Int
{SUC} sucesor n = n + 1
```

```
     sucesor (factorial 10) + 1
  =  (factorial 10 + 1) + 1       por SUC
  =  sucesor (factorial 10 + 1)   por SUC
```

Igualdades 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
```

Introducción

Inducción estructural

Extensionalidad

Isomorfismos de tipos

Casos de estudio

Inducción sobre booleanos

El principio de reemplazo no alcanza para probar todas las equivalencias que nos interesan.

Ejemplo

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

¿Podemos probar $\forall$`x :: Bool. not (not x) = x`?

El problema es que la expresión

```
not (not x)
```

está "trabada": no se puede aplicar ninguna ecuación.

Inducción sobre booleanos

Principio de inducción sobre booleanos

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

Ejemplo

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

Para probar $\forall \texttt{x} :: \texttt{Bool}.\ \texttt{not (not x) = x}$
basta probar:

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

$$\texttt{not (not True)} \underset{\texttt{NT}}{\overset{\uparrow}{=}} \texttt{not False} \underset{\texttt{NF}}{\overset{\uparrow}{=}} \texttt{True}$$

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

$$\texttt{not (not False)} \underset{\texttt{NF}}{\overset{\uparrow}{=}} \texttt{not True} \underset{\texttt{NT}}{\overset{\uparrow}{=}} \texttt{False}$$

Inducción sobre pares

Cada tipo de datos tiene su propio principio de inducción.

Ejemplo

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

¿Podemos probar $\forall$`p :: (a, b). fst p = snd (swap p)`?

Las expresiones `(fst p)` y `(snd (swap p))` están “trabadas”.

Inducción sobre pares

Principio de inducción sobre pares

Si $\forall$`x :: a.` $\forall$`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 $\forall$`p :: (a, b). fst p = snd (swap p)`
basta probar:

- $\forall$`x :: a.` $\forall$`y :: b. fst (x, y) = snd (swap (x, y))`

$$\texttt{fst (x, y)} \underset{\texttt{FST}}{=} \texttt{x} \underset{\texttt{SND}}{=} \texttt{snd (y, x)} \underset{\texttt{SWAP}}{=} \texttt{snd (swap (x, y))}$$

Inducción sobre naturales

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

Principio de inducción sobre naturales

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

entonces $\forall\texttt{n} :: \texttt{Nat}.\ \mathcal{P}(\texttt{n})$.

Inducción sobre naturales

Ejemplo

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

Para probar $\forall$`n` :: `Nat. suma n Zero = n`
basta probar:

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

$$\texttt{suma (Suc n) Zero} \underset{\texttt{S1}}{\overset{\uparrow}{=}} \texttt{Suc (suma n Zero)} \underset{\text{H.I.}}{\overset{\uparrow}{=}} \texttt{Suc n}$$

Inducción estructural

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

$$
\begin{array}{llll}
\texttt{data T} & = & \texttt{CBase}_1 & \langle \textit{parámetros} \rangle \\
 & & \ldots & \\
 & | & \texttt{CBase}_n & \langle \textit{parámetros} \rangle \\
 & | & \texttt{CRecursivo}_1 & \langle \textit{parámetros} \rangle \\
 & & \ldots & \\
 & | & \texttt{CRecursivo}_m & \langle \textit{parámetros} \rangle
\end{array}
$$

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

Inducción estructural

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

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

Inducción estructural

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:

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

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

Ejemplo: inducción sobre listas

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

Ejemplo: inducción sobre listas

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
```

Ejemplo: relación entre `foldr` y `foldl`

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

$$\underbrace{\texttt{foldr f z xs = foldl (flip f) z (reverse xs)}}_{\mathcal{P}(\texttt{xs})}$$

Por inducción en la estructura de `xs`. El caso base $\mathcal{P}(\texttt{[]})$ es fácil.
Caso inductivo, $\forall\texttt{x :: a}.\ \forall\texttt{xs :: [a]}.\ (\mathcal{P}(\texttt{xs}) \Rightarrow \mathcal{P}(\texttt{x : xs}))$:

	`foldr f z (x : xs)`	
=	`f x (foldr f z xs)`	(Def. `foldr`)
=	`f x (foldl (flip f) z (reverse xs))`	(H.I.)
=	`flip f (foldl (flip f) z (reverse xs)) x`	(Def. `flip`)
=	`foldl (flip f) z (reverse xs ++ [x])`	**(???)**
=	`foldl (flip f) z (reverse (x : xs))`	(Def. `reverse`)

Para justificar el paso faltante **(???)**, se puede demostrar:
Lema. Si `g :: b -> a -> b`, `z :: b`, `x :: a`, `xs :: [a]`, entonces:

$$\texttt{foldl g z (xs ++ [x]) = g (foldl g z xs) x}$$

Lemas de generación

Usando el principio de inducción estructural, se puede probar:

Lema de generación para pares

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

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

Lema de generación para sumas

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

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

Introducción

Inducción estructural

Extensionalidad

Isomorfismos de tipos

Casos de estudio

Puntos de vista intensional vs. extensional

¿Vale la siguiente equivalencia de expresiones?

```
quickSort = insertionSort
```

Depende del punto de vista:

Punto de vista intensional. (va con "s")
Dos valores son iguales si están construidos de la misma manera.

Punto de vista extensional.
Dos valores son iguales si son indistinguibles al observarlos.

Ejemplo

`quickSort` e `insertionSort`

- **no** son intensionalmente iguales;
- **sí** son extensionalmente iguales: computan la misma función.

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`.

Principio de extensionalidad funcional

Ejemplo: extensionalidad funcional

```
{I} id x = x                  {C} (g . f) x = g (f x)
{S} swap (x, y) = (y, x)
```

Veamos que `swap . swap = id` :: `(a, b) -> (a, b)`.
Por extensionalidad funcional, basta ver:

$$\forall p :: (a, b).\ (\texttt{swap . swap})\ p = \texttt{id}\ p$$

Por inducción sobre pares, basta ver:

$$\forall x :: a.\ \forall y :: b.\ (\texttt{swap . swap})\ (x, y) = \texttt{id}\ (x, y)$$

En efecto:

$$
\begin{array}{lll}
 & (\texttt{swap . swap})\ (x, y) & \\
= & \texttt{swap}\ (\texttt{swap}\ (x, y)) & (\text{por } C) \\
= & \texttt{swap}\ (y, x) & (\text{por } S) \\
= & (x, y) & (\text{por } S) \\
= & \texttt{id}\ (x, y) & (\text{por } I)
\end{array}
$$

Resumen: razonamiento ecuacional

Razonamos ecuacionalmente usando tres principios:

1. **Principio de reemplazo**
   Si el programa declara que `e1 = e2`, cualquier instancia de `e1` es igual a la correspondiente instancia de `e2`, y viceversa.
2. **Principio de inducción estructural**
   Para probar $\mathcal{P}$ sobre todas las instancias de un tipo `T`, basta probar $\mathcal{P}$ para cada uno de los constructores
   (asumiendo la H.I. para los constructores recursivos).
3. **Principio de extensionalidad funcional**
   Para probar que dos funciones son iguales, basta probar que son iguales punto a punto.

Corrección del razonamiento ecuacional

Supongamos que logramos demostrar que `e1 = e2`.
¿Qué nos asegura eso sobre `e1` y `e2`?

Cuidado: no necesariamente dan el mismo resultado
Por ejemplo, se puede demostrar:

```
quickSort = insertionSort
```

pero `quickSort` e `insertionSort` no dan el mismo resultado.

Corrección con respecto a observaciones

Si demostramos `e1 = e2` :: `A`, entonces:

$$\texttt{obs e1} \rightsquigarrow \texttt{True} \quad \text{si y sólo si} \quad \texttt{obs e2} \rightsquigarrow \texttt{True}$$

para toda posible "observación" `obs` :: `A -> Bool`.

Demostración de desigualdades

¿Cómo demostramos que **no** vale una igualdad `e1 = e2 :: A`?

Por la contrarrecíproca de la anterior, basta con encontrar una observación `obs :: A -> Bool` que las distinga.

Ejemplo

Demostrar que **no** vale la igualdad:

```
id = swap :: (Int, Int) -> (Int, Int)
```

```
obs :: ((Int, Int) -> (Int, Int)) -> Bool
obs f = fst (f (1, 2)) == 1
```

```
obs id    ⇝  True
obs swap  ⇝  False
```

Introducción

Inducción estructural

Extensionalidad

Isomorfismos de tipos

Casos de estudio

Misma información, distinta forma

¿Qué relación hay entre los siguientes valores?

```
("hola", (1, True)) :: (String, (Int, Bool))
((True, "hola"), 1) :: ((Bool, String), Int)
```

Representan la misma información, pero escrita de distinta manera.

Podemos transformar los valores de un tipo en valores del otro:

```
f :: (String, (Int, Bool)) -> ((Bool, String), Int)
f (s, (i, b)) = ((b, s), i)

g :: ((Bool, String), Int) -> (String, (Int, Bool))
g ((b, s), i) = (s, (i, b))
```

Se puede demostrar que:

```
g . f = id     f . g = id
```

Isomorfismos de tipos

Definición

Decimos que dos tipos de datos `A` y `B` son **isomorfos** si:

1. Hay una función `f :: A -> B` total.
2. Hay una función `g :: B -> A` total.
3. Se puede demostrar que `g . f = id :: A -> A`.
4. Se puede demostrar que `f . g = id :: B -> B`.

Escribimos $\mathtt{A} \simeq \mathtt{B}$ para indicar que `A` y `B` son isomorfos.

Ejemplo de isomorfismo: currificación

Ejemplo

Veamos que `((a, b) -> c)` $\simeq$ `(a -> b -> c)`.

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

Ejemplo de isomorfismo: currificación

Veamos que

`uncurry . curry = id  :: ((a, b) -> c) -> (a, b) -> c`

Por extensionalidad funcional, basta ver que si `f :: (a, b) -> c`:

`(uncurry . curry) f = id f  :: (a, b) -> c`

Por extensionalidad funcional, basta ver que si `p :: (a, b)`:

`(uncurry . curry) f p = id f p  :: c`

Por inducción sobre pares, basta ver que si `x :: a`, `y :: b`:

`(uncurry . curry) f (x, y) = id f (x, y)  :: c`

En efecto:

```
     (uncurry . curry) f (x, y)
  =  uncurry (curry f) (x, y)     (Def. (.))
  =  curry f x y                  (Def. uncurry)
  =  f (x, y)                     (Def. curry)
  =  id f (x, y)                  (Def. id)
```

(Y vale también `curry . uncurry = id`).

Más isomorfismos de tipos

`(a, b)`	$\simeq$	`(b, a)`
`(a, (b, c))`	$\simeq$	`((a, b), c)`
`a -> b -> c`	$\simeq$	`b -> a -> c`
`a -> (b, c)`	$\simeq$	`(a -> b, a -> c)`
`Either a b -> c`	$\simeq$	`(a -> c, b -> c)`

Introducción

Inducción estructural

Extensionalidad

Isomorfismos de tipos

Casos de estudio

Ejemplo — Necesidad de usar lemas auxiliares

Asumimos las definiciones usuales para `(.)` y `(++)` y la siguiente para `reverse`:

```
{R0} reverse []       = []
{R1} reverse (x : xs) = reverse xs ++ [x]
```

Consideremos además la siguiente definición:

```
     ceros :: [a] -> [Int]
{Z0} ceros []       = []
{Z1} ceros (_ : xs) = 0 : ceros xs
```

Demostremos que `ceros . reverse = reverse . ceros`.

¿Qué ocurre?

Necesitamos un **lema auxiliar**:

```
∀xs ys :: [a]. ceros (xs ++ ys) = ceros xs ++ ceros ys
```

Ejemplo — Necesidad de generalizar el predicado inductivo

Consideremos la siguiente definición, usando recursión iterativa:

```
      suma :: Int -> [Int] -> Int
{S0}  suma k []           = k
{S1}  suma k (x : xs)     = suma (x + k) xs
```

Demostremos que para `k :: Int` y `xs :: [Int]` vale:

```
suma k (xs ++ ys) = suma (suma k xs) ys
```

¿Qué ocurre?

Necesitamos **generalizar** el predicado inductivo de $\mathcal{P}$ a $\mathcal{Q}$:

$\mathcal{P}(\texttt{xs}) \equiv \boxed{\texttt{suma k (xs ++ ys) = suma (suma k xs) ys}}$

$\mathcal{Q}(\texttt{xs}) \equiv \boxed{\forall\texttt{k' :: Int. suma k' (xs ++ ys) = suma (suma k' xs) ys}}$

Ejemplo — Necesidad de generalizar el predicado inductivo

Definimos funciones para acumular una lista, usando recursión iterativa y estructural:

```
{L0} acumL k []       = []
{L1} acumL k (x : xs) = (x + k) : acumL (x + k) xs

{R0} acumR []       = []
{R1} acumR (x : xs) = x : map (+ x) (acumR xs)
```

Demostremos que `acumL 0 = acumR`.

¿Qué ocurre?

Necesitamos **generalizar** el predicado inductivo de $\mathcal{P}$ a $\mathcal{Q}$:

$$\mathcal{P}(\mathtt{xs}) \equiv \boxed{\mathtt{acumL\ 0\ xs\ =\ acumR\ xs}}$$

$$\mathcal{Q}(\mathtt{xs}) \equiv \boxed{\forall \mathtt{k} :: \mathtt{Int}.\ \mathtt{acumL\ k\ xs\ =\ map\ (+\ k)\ (acumR\ xs)}}$$

(La demostración completa requiere algunos lemas auxiliares más).

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Lectura recomendada

Capítulo 6 del libro de Bird.

Richard Bird. *Thinking functionally with Haskell*
Cambridge University Press, 2015.