Teórica 3 : Razonamiento Ecuacional e Inducción Estructural

Tomás Felipe Melli

April 10, 2025

Índice

1 Introducción **2**
1.1 Igualdades por definición . . . 2
1.1.1 Principio de reemplazo . . . 2

2 Inducción Estructural **3**
2.1 Inducción sobre Booleanos . . . 3
2.1.1 Principio de Inducción sobre Booleanos . . . 3
2.2 Inducción sobre Pares . . . 3
2.2.1 Principio de Inducción sobre Pares . . . 3
2.3 Inducción sobre Naturales . . . 3
2.3.1 Principio de Inducción sobre Naturales . . . 3
2.4 Caso General . . . 4
2.4.1 Principio de Inducción Estructural . . . 4
2.4.2 Principio de Inducción Estructural sobre Listas . . . 4
2.4.3 Principio de Inducción Estructural sobre Árboles Binarios . . . 5
2.4.4 Principio de Inducción Estructural sobre Árboles Polinomios . . . 5
2.5 Relación entre foldr y foldl . . . 5
2.6 Lemas de Generación . . . 5
2.6.1 Lema de Generación para Pares . . . 6
2.6.2 Lema de Generación para Sumas . . . 6

3 Extensionalidad **6**
3.1 Principio de Extensionalidad Funcional . . . 6
3.2 Resumen : Razonamiento Ecuancional . . . 7

4 Isomorfismos de tipos **7**

1 Introducción

La motivación de querer demostar que ciertas expresiones son equivalentes es que eso nos permitirá probar la **correctitud de un algoritmo** y también para **posibilitar optimizaciones**.
Para razonar sobre la equivalencia de expresiones vamos a **asumir**:

1. Que trabajamos con **estructuras de datos finitas**
2. Que trabajamos con **fucniones totales**, es decir, las ecuaciones deben *cubrir todos los casos* y la *recursión siempre termina*
3. Que el programa **no depende del orden** de las ecuaciones.

```haskell
vacia [] = True
vacia _ = False

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

1.1 Igualdades por definición

1.1.1 Principio de reemplazo

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

1. Reemplazar *cualquier instancia* de $e1$ por $e2$
2. Reemplazar *cualquier instancia* de $e2$ por $e1$

Si una igualdad se puede demostrar usando sólo este principio, decimos que la igualdad vale por definición.
Veamos un ejemplo :

```haskell
sucesor :: Int -> Int
sucesor n = n + 1 -- {SUC} nombramos la ecuaci n
```

$$sucesor\ (factorial\ 10) + 1$$
$$= (factorial\ 10 + 1) + 1\ por\ \{SUC\}$$
$$= sucesor(factorial\ 10 + 1)\ por\ \{SUC\}$$

Otro ejemplo :

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

Queremos ver que valga :

$$length\ [''a'', ''b''] = suma[1, 1]$$

entonces :

$$length\ [''a'', ''b'']$$
$$= 1 + length\ [''b'']) \ por\ \{L1\}$$
$$= 1 + (1 + 0)\ por\ \{L0\}$$
$$= 1 + (1 + suma[\,])\ por\ \{S0\}$$
$$= 1 + suma[1]\ por\ \{S1\}$$
$$= suma[1, 1]\ por\ \{S1\}$$

2 Inducción Estructural

2.1 Inducción sobre Booleanos

Ocurre que el principio visto no siempre nos sirve para probar las equivalencia que queremos. Miremos este ejemplo :

$$\{NT\}\ not\ True = False$$

$$\{NF\}\ not\ False = True$$

Podemos probar que $\forall x :: Bool.not\ (not\ x) = x$? El problema es que la expresión $not\ (not\ x)$ está **trabada**, y esto es porque *no podemos aplicar ninguna ecuación*. Es por ello que presentamos el siguiente *principio*.

2.1.1 Principio de Inducción sobre Booleanos

Si $P(True)$ *y* $P(False)$ *entonces* $\forall x :: Bool.P(x)$ Con esto en mente, probamos $\forall x :: Bool.not\ (not\ x) = x$

1. $not\ (not\ True) \overset{\text{NT}}{=} not\ False \overset{\text{NF}}{=} True$
2. $not\ (not\ False) \overset{\text{NF}}{=} not\ True \overset{\text{NT}}{=} False$

Es importante destacar que **cada tipo de datos tiene su principio de inducción**.

2.2 Inducción sobre Pares

Sean :

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

Se puede probar que $\forall p :: (a, b).fst\ p = snd\ (swap\ p)$? El tema es que $(fst\ p)$ y $(snd\ (swap\ p))$ estn trabadas y por ello surge el *principio de inducción sobre pares*

2.2.1 Principio de Inducción sobre Pares

El principio nos dice que :

$$Si\ \forall x :: a.\forall y :: b.P((x, y))\ entonces\ \forall p :: (a, b).P(p)$$

Probemos entonces lo anterior.

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

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

2.3 Inducción sobre Naturales

Definimos al tipo de datos **Nat**

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

2.3.1 Principio de Inducción sobre Naturales

$$Si\ P(Zero)\ y\ \forall n :: Nat.(\underbrace{P(n)}_{\text{Hipótesis inductiva}} \implies \underbrace{P(Suc\ n)}_{\text{Tesis inductiva}})\ entonces\ \forall n :: Nat.P(n)$$

Vemos el siguiente ejemplo..

$$\{SO\}\ suma\ Zerom = m$$

$$\{SO\}\ suma\ (Suc\ n)\ m = Suc\ (suma\ n)\ m$$

Queremos probar que $\forall n :: Nat.suma\ n\ Zero = n$

1. Probamos $P(0)$: $suma\ Zero\ Zero = Zero$ inmediato por $\{S0\}$
2. Planteamos entonces

$$\underbrace{suma\ n\ Zero = n}_{\text{H.I}} \implies \underbrace{suma\ (Suc\ n)\ Zero = Suc\ n}_{\text{T.I}}$$

Dicho esto, $suma\ (Suc\ n)\ Zero \overset{\text{S1}}{=} Suc\ (suma\ n\ Zero) \overset{\text{H.I}}{=} Suc\ n$

2.4 Caso General

```
data T  = CBase1 <par metros>
        ...
        | CBaseN <par metros>
        | CRecursivo1 <par metros>
        ...
        | CRecursivoM <par metros>
```

2.4.1 Principio de Inducción Estructural

Sea **P** una **propiedad acerca de las expresiones de tipo T** tal que :

- **P vale sobre todos los constructores de base T**
- **P vale sobre todos los constructores recursivos de T**, asumiendo como HI que vale para parámetros de tipo T

Entonces $\forall x :: T.P(x)$
Veamo un ejemplo

2.4.2 Principio de Inducción Estructural sobre Listas

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

Sea **P** una propiedad sobre expresiones del tipo $[a]$ tal que :

- $P([\,])$
- $\forall x :: a.\forall xs :: [a].(\underbrace{P(xs)}_{\text{H.I}} \implies \underbrace{P(x : xs)}_{\text{T.I}})$ Entonces $\forall xs :: [a].P(xs)$

Ejemplo :

$$\{M0\}\ map\ f\ [\,] = [\,]$$
$$\{M1\}\ map\ f\ (x : xs) = f\ x :\ map\ f\ xs$$
$$\{A0\}\ [\,]\ ++\ ys = ys$$
$$\{A0\}\ (x : xs)\ ++\ ys = x : (xs ++ ys)$$

Propiedad : si $f :: a \to b, xs :: [a], ys :: [a], entonces$

$$map\ f\ (xs ++ ys) = map\ f\ xs ++ map\ f\ ys$$

Por inducción en la estructura de xs, basta ver :

1. Caso base, $P([\,])$
2. Caso inductivo, $\forall x :: a.\forall xs :: [a].(\underbrace{P(xs)}_{\text{H.I}} \implies \underbrace{P(x : xs)}_{\text{T.I}})$ con

$$P(xs) \equiv (map\ f\ (xs ++ ys) = map\ f\ xs ++ map\ f\ ys)$$

Caso Base

$$map\ f\ ([\,] ++ ys)$$
$$= map\ f\ ys\ \{A0\}$$
$$= [\,] ++ map\ f\ ys\ \{A0\}$$
$$= map\ f\ [\,] ++ map\ f\ ys\ \{M0\}$$

Caso Inductivo

$$map\ f\ ((x : xs) ++ ys)$$
$$= map\ f\ (x : (xs ++ ys))\ \{A1\}$$
$$= f\ x :\ map\ f\ (xs ++ ys)\ \{M1\}$$
$$= f\ x :\ (map\ f\ xs ++ map\ f\ ys)\ \{H.I\}$$
$$= (f\ x :\ map\ f\ xs) ++ map\ f\ ys\ \{A1\}$$
$$= map\ f\ (x : xs) ++ map\ f\ ys\ \{M1\}$$

2.4.3 Principio de Inducción Estructural sobre Árboles Binarios

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

Sea **P** una propiedad sobre expresiones del tipo $AB\ a$ tal que :

- $P(Nil)$
- $\forall i :: AB\ a.\forall r :: a.\forall b :: AB\ a$

$$(\underbrace{(P(i) \wedge P(d))}_{\text{H.I}} \implies \underbrace{P(Bin\ i\ r\ d)}_{\text{T.I}})$$

Entonces $\forall x :: AB\ a.P(x)$

2.4.4 Principio de Inducción Estructural sobre Árboles Polinomios

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

Sea **P** una propiedad sobre expresiones de tipo $Poli\ a$ tal que :

- $P(X)$
- $\forall k :: a.P(Cte\ k)$
- $\forall p :: Poli\ a.\forall q :: Poli\ a.$

$$(\underbrace{(P(p) \wedge P(q))}_{\text{H.I}} \implies \underbrace{P(Suma\ p\ q)}_{\text{T.I}})$$

- $\forall p :: Poli\ a.\forall q :: Poli\ a.$

$$(\underbrace{(P(p) \wedge P(q))}_{\text{H.I}} \implies \underbrace{P(Prod\ p\ q)}_{\text{T.I}})$$

Entonces $\forall x :: Poli\ a.P(x)$

2.5 Relación entre foldr y foldl

Propiedad: *Si* $f :: a \to b \to b, z :: b, xs :: [a]$ entonces :

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

Por **inducción** en la estructura de **xs**. El **caso base** $P([\,])$ es fácil. Y el **caso inductivo** $\forall x :: a.\forall xs :: [a].(P(xs) \implies P(x : xs))$

$$foldr\ f\ z\ (x : xs)$$
$$= f\ x\ (foldrf\ z\ xs)\ \{Def\ foldr\}$$
$$= f\ x\ (foldl\ (flip\ f)\ z\ (reverse\ xs))\ \{H.I\}$$
$$= flip\ f\ (foldl\ (flip\ f)\ z\ (reverse\ xs))\ x\ \{Def\ flip\}$$
$$= foldl\ (flip\ f)\ z\ (reverse\ xs ++ x)\ \{(\mathbf{??})\}$$
$$= foldl\ (flip\ f)\ z\ (reverse\ (x : xs))\ \{Def\ reverse\}$$

Para justificar el paso faltante (**??**) se puede demostrar con el siguiente **Lema** : *Si* $g :: b \to a \to b, z :: b, x :: a, xs :: [a]$ entonces

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

2.6 Lemas de Generación

Un lema de generación afirma que todo valor de un tipo inductivo fue construido usando alguno de los constructores de ese tipo. Usando el principio de inducción estructural, se puede probar

2.6.1 Lema de Generación para Pares

La intuición nos dice que Si p es un par, entonces fue construido como un par, es decir, hay un primer elemento x y un segundo y tales que p = (x, y).

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

2.6.2 Lema de Generación para Sumas

La intuición nos dice que no hay otra forma de construir un valor del tipo Either a b que no sea usando alguno de los dos constructores (Left o Right).

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

Si $e :: Either\ a\ b,\ entonces :$

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

3 Extensionalidad

La extensionalidad es un principio lógico o matemático que dice que dos objetos son iguales si no se pueden distinguir por su comportamiento externo.
Es una forma de definir la igualdad que se basa en lo que los objetos hacen, no en cómo están construidos o representados internamente.
Nos preguntamos : vale la siguiente equivalencia de expresiones ?

$$quickSort = insertionSort$$

Depende del punto de vista

- **Punto de vista Intensional** (con 's') : dos valores son iguales si están construidos de la misma manera
- **Punto de vist Extensional** : dos valores son iguales si son indistiguibles al observarlos

En la equivalencia anterior, **no son intensionalmente iguales, pero sí extensionalmente (computan la misma función)**

3.1 Principio de Extensionalidad Funcional

$Sean\ f, g :: a \to b$ Inmediatamente podemos decir que $Si\ (\forall x :: a.\ f\ x = g\ x)\ entonces\ f = g.$ Veamos un ejemplo :

$$\{I\}\ id\ x = x$$

$$\{S\}\ swap\ (x, y) = (y, x)$$

$$\{C\}\ (g\ .\ f)\ x = g\ (f\ x)$$

Vemos que $swap\ .\ swap = id :: (a, b) \to (a, b)$ Por extensionalidad funcional basta ver :

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

Por *inducción sobre pares* :

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

En efecto :

$$(swap\ .\ swap)\ (x, y)$$

$$= swap\ (swap\ (x, y)\ \{C\}$$

$$= swap\ (y, x)\ \{S\}$$

$$= (x, y)\ \{S\}$$

$$= id\ (x, y)\ \{I\}$$

3.2 Resumen : Razonamiento Ecuancional

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 **P** sobre todas las instancias de un tipo **T**, basta con probar **P** sobre **cada uno de sus constructores** (asumimos H.I para los constructores recursivos)
3. **Principio de extensionalidad funcional** : para probar que dos fucniones son iguales, basta probar que son **iguales en un punto**

4 Isomorfismos de tipos

Qué relación existe entre : ?

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

Representan la misma información, pero escrita diferente
Como podemos transformar lo valores de un tipo en 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))
```

Podemos demostrar que

$$g \ . \ f = id \quad f \ . \ g = id$$

Veamos un ejemplo de isomorfismo con currificación.
Queremos ver 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
```