Teórica 2

Paradigmas de Programación

Razonamiento ecuacional
Inducción estructural

2do cuatrimestre de 2024
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Motivación

Validación y verificación de programas

¿Cómo mostramos que un programa hace lo que tiene que hacer?

- Existen diferentes técnicas para abordar este problema.
- En esta materia veremos **prueba formal de propiedades** para razonar sobre el comportamiento de programas.
- ∀xs :: [a]. `length xs = length (reverse xs)`
- ∀xs :: [a]. `reverse xs = reverse (reverse xs)`

Prueba formal de propiedades

Queremos demostrar que ciertas expresiones son equivalentes.

Usos

- Probar que un programa es correcto
- Razonar sobre optimizaciones/alternativas

```
f x + f x          =  2 * f x
map f (map g xs)   =  map (f . g) xs
```


Hipótesis de trabajo

Vamos a asumir que

1. Trabajamos con estructuras de datos **finitas**.
   Técnicamente: con tipos de datos **inductivos**.
2. Trabajamos con **funciones totales**.
   - Las ecuaciones deben cubrir todos los casos.
   - La recursión siempre debe terminar.
3. 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 del 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

Considerar el siguiente programa (damos nombre a las ecuaciones):

```
      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
```


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

Ejemplo

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

Para probar $\forall$`x :: Bool. not (not x) = x`, basta probar:

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

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

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

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

Inducción sobre pares

Inducción y tipos de datos

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

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

Inducción sobre naturales

Naturales

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

Principio de inducción sobre naturales

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

Inducción sobre naturales

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

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

Inducción estructural: Caso general

Tipos de datos inductivo

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

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

Inducción estructural

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:

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

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, $\mathcal{P}$(`[]`).
2. Caso inductivo, ∀`x` :: `a`. ∀`xs` :: `[a]`. ($\mathcal{P}$(`xs`) $\Rightarrow$ $\mathcal{P}$(`x : xs`)).

con $\mathcal{P}$(`xs`) :≡ (`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 $\mathcal{P}$(`[]`) es fácil.
Caso inductivo, ∀`x` :: `a`. ∀`xs` :: `[a]`. ($\mathcal{P}$(`xs`) $\Rightarrow$ $\mathcal{P}$(`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:

```
foldl g z (xs ++ [x]) = g (foldl g z xs) x
```


Extensionalidad

Usando el principio de inducción estructural, se puede probar:

Extensionalidad para pares

Si `p :: (a, b)`, entonces ∃`x` :: `a`. ∃`y` :: `b`. `p = (x, y)`.

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

Extensionalidad para sumas

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

- o bien ∃`x` :: `a`. `e = Left x`
- o bien ∃`y` :: `b`. `e = Right y`

Puntos de vista intensional vs. extensional

¿Vale la siguiente equivalencia de expresiones?

```
mergesort = insertionSort
```

Depende del punto de vista.

- **Punto de vista intensional.** (va con "s")
  Dos valores son iguales si están definidos de la misma manera.
- **Punto de vista extensional.**
  Dos valores son iguales si son indistinguibles al observarlos.

Ejemplo

`mergesort` 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 (∀`x :: a. f x = g x`).

Principio de extensionalidad funcional

Si (∀`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:

∀`p :: (a, b). (swap . swap) p = id p`

Por inducción sobre pares, basta ver:

∀`x :: a.` ∀`y :: b. (swap . swap) (x, y) = id (x, y)`

En efecto:

	`(swap . swap) (x, y)`	
=	`swap (swap (x, y))`	(por C)
=	`swap (y, x)`	(por S)
=	`(x, y)`	(por S)
=	`id (x, y)`	(por I)

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`?

Se puede demostrar

```
quickSort = insertionSort
```

pero `quickSort` e `insertionSort` no son el mismo código.

Corrección con respecto a observaciones

Si demostramos `e1 = e2 :: A`, entonces para toda posible "observación" `obs :: A -> Bool`.

`obs e1` ⇝ `True` si y sólo si `obs e2` ⇝ `True`

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
```


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 `A` $\simeq$ `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)`