Programación Funcional en Haskell

Segunda parte (reforzada)

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

9 de abril de 2024

Anteriormente en PLP...

- Tipos algebraicos en Haskell
- Esquemas de recursión
- Currificación
- Aplicación parcial
- Alto orden
- ...

Hoy

- Razonamiento ecuacional
- Extensionalidad
- Inducción estructural
- Ejercicios más difíciles
- ...

Básicamente, vamos a ver cómo demostrar propiedades sobre nuestros programas, y cómo usar aplicación parcial para "recorrer dos estructuras a la vez".

Demostrando propiedades

Sean las siguiente definiciones:

```
doble :: Integer -> Integer
doble x = 2 * x
```

```
cuadrado :: Integer -> Integer
cuadrado x = x * x
```

¿Cómo probamos que `doble 2` $=$ `cuadrado 2`?

Solución:

`doble 2` $=_{\texttt{doble}}$ `2 * 2` $=_{\texttt{cuadrado}}$ `cuadrado 2` $\square$

Igualdad de funciones

Queremos ver que:

```
curry . uncurry = id
```

Tenemos:

```
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)
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
id :: a -> a
id x = x
```

¿Cómo hacemos?

Extensionalidad

Dadas `f, g :: a -> b`, probar f $=$ g se reduce a probar:

$$\forall \texttt{x::a}\,.\ \texttt{f x} = \texttt{g x}$$

Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

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

`F` $=$ `G`	$\Leftrightarrow$	$\forall$ `x::a . F x` $=$ `G x`
`F` $=$ `\x -> Y`	$\Leftrightarrow$	$\forall$ `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).

Volviendo al ejercicio...

Ahora probemos:

$$\texttt{curry . uncurry} = \texttt{id}$$

Tenemos:

```
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)
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
id :: a -> a
id x = x
```

Se define la siguiente función, que permite multiplicar pares y enteros entre sí (usando producto escalar entre pares).

```
prod :: Either Int (Int, Int) -> Either Int (Int, Int) ->
Either Int (Int, Int)
prod (Left x) (Left y) = Left (x * y)
prod (Left x) (Right (y, z)) = Right (x * y, x * z)
prod (Right (y, z)) (Left x) = Right (y * x, z * x)
prod (Right (w, x)) (Right (y, z)) = Left (w * y + x * z)
```

¿Podemos probar esto?

$$\forall \texttt{p::Either Int (Int, Int)}\,.\,\forall \texttt{q::Either Int (Int, Int)}\,.\; \texttt{prod p q} = \texttt{prod q p}$$

Pares y unión disjunta/tipo suma

Recordemos los principios de extensionalidad para pares y sumas.

Dado `p :: (a, b)`, siempre podemos usar el hecho de que existen `x :: a`, `y :: b` tales que `p = (x, y)`.

De la misma manera, dado `e :: Either a b`, siempre podemos usar el hecho de que:

- `e = Left x` con `x :: a`, o
- `e = Right y` con `y :: b`.

Pares y unión disjunta/tipo suma

Probemos enconces...

$$\forall \texttt{p::Either Int (Int, Int)}\,.\,\forall \texttt{q::Either Int (Int, Int)}\,.\\ \texttt{prod p q} = \texttt{prod q p}$$

Tenemos:

```
prod :: Either Int (Int, Int) -> Either Int (Int, Int) ->
Either Int (Int, Int)
prod (Left x) (Left y) = Left (x * y)
prod (Left x) (Right (y, z)) = Right (x * y, x * z)
prod (Right (y, z)) (Left x) = Right (y * x, z * x)
prod (Right (w, x)) (Right (y, z)) = Left (w * y + x * z)
```

Funciones como estructuras de datos

Se cuenta con la siguiente representación de conjuntos:
`type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si c es un conjunto y e un elemento, la expresión $c\ e$ devuelve `True` si e pertenece a c y `False` en caso contrario.

1. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.
2. Escribir las funciones `intersección`, `unión` y `diferencia` (todas de tipo `Conj a -> Conj a-> Conj a`).
3. Demostrar la siguiente propiedad:
   ∀ `c::Conj a` . ∀ `d::Conj a` . intersección d (diferencia c d) = vacío

Inducción en los naturales

- Pruebo $P(0)$
- Pruebo que **si** vale $P(n)$ **entonces** vale $P(n+1)$.

Inducción en listas

- Pruebo $P([\,])$
- Pruebo que **si** vale $P(\mathtt{xs})$ **entonces** para todo elemento $\mathtt{x}$ vale $P(\mathtt{x:xs})$.

En el caso general (inducción estructural)

- Pruebo P para el o los caso(s) base (para los constructores no recursivos).
- Pruebo que **si** vale $P(Arg_1), \ldots, P(Arg_k)$ **entonces** vale $P(C\ Arg_1\ \ldots\ Arg_k)$ para cada constructor C y sus argumentos recursivos $Arg_1,\ \ldots,\ Arg_k$.
  (Los argumentos no recursivos quedan cuantificados universalmente).

Pasos a seguir

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.
- Plantear el esquema de inducción.
- Plantear y resolver el o los caso(s) base.
- Plantear y resolver el o los caso(s) inductivo(s).

Desplegando foldr

Veamos que estas dos definiciones de `length` son equivalentes:

```
length1 :: [a] -> Int
length1 [] = 0
length1 (_:xs) = 1 + length1 xs
```

```
length2 :: [a] -> Int
length2 = foldr (\_ res -> 1 + res) 0
```

Recordemos:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Demostrando implicaciones

Queremos probar que:

$$\texttt{Ord a} \Rightarrow \forall\, \texttt{e::a} \,.\, \forall\, \texttt{ys::[a]} \,.\, \texttt{elem e ys} \Rightarrow e \leq \texttt{maximum ys}$$

Antes que nada, ¿quién es P?
¿En qué estructura vamos a hacer inducción?

$$P(\texttt{ys}) = \texttt{Ord a} \Rightarrow \forall\, \texttt{e::a} \,.\, \texttt{elem e ys} \Rightarrow e \leq \texttt{maximum ys}$$

Ahora bien, si no vale `Ord a`, la implicación de afuera es trivialmente verdadera (recordar que las implicaciones asocian a derecha). Además, si vale `Ord a`, también vale `Eq a` (por la jerarquía de clases de tipos en Haskell).

Suponemos que todo eso vale y vamos a probar lo que nos interesa.

Demostrando implicaciones (continúa)

Tenemos:

```
elem :: Eq a => a -> [a] -> Bool
elem e [] = False
elem e (x:xs) = (e == x) || elem e xs
```

```
maximum :: Ord a => [a] -> a
maximum [x] = x
maximum (x:xs) = if x < maximum xs then maximum xs else x
```

Sabemos que valen `Eq a` y `Ord a`. Queremos ver que, para toda lista `ys`, vale:

$$\forall \mathtt{e::a} \,.\, \mathtt{elem\ e\ ys} \Rightarrow e \leq \mathtt{maximum\ ys}$$

Seguimos en el pizarrón.

Otra vuelta de tuerca

Dadas las siguientes definiciones:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x:xs) = foldl f (f ac x) xs

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []

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

Queremos probar que:

∀`ys::[a] . length ys` = `length (reverse ys)`

...que, por `reverse`, es lo mismo que:

∀`ys::[a] . length ys` = `length (foldl (flip (:)) [] ys)`

Avancemos hasta que nos trabemos.

Generalizando propiedades

$$P(\texttt{ys}) = \texttt{length ys} = \texttt{length (foldl (flip (:)) [] ys)}$$

En el caso inductivo (`ys = x:xs`) nuestra Hipótesis Inductiva es:

$$\texttt{length xs} = \texttt{length (foldl (flip (:)) [] xs)}$$

Pero lo que necesitamos es:

$$1 + \texttt{length xs} = \texttt{length (foldl (flip (:)) (x:[]) xs)}$$

¿Qué podemos hacer?

Respuesta: demostrar una propiedad más general.
Probemos: $\forall \texttt{ys::[a]}.\forall \texttt{zs::[a]}.\texttt{length zs} + \texttt{length ys} =$
$\texttt{length (foldl (flip (:)) zs ys)}$
Luego, tomando $\texttt{zs} = \texttt{[]}$ y sabiendo que $\texttt{length []} = 0$, obtenemos lo que buscábamos.

Una más difícil

Escribir la función `take :: Int -> [a] -> [a]` usando `foldr`.

Pista: definir `flipTake` tal que `take = flip flipTake`[1].

[1] En sus casas pueden probar definir `take` sin `flip`, usando `foldNat`.

¿Está bien lo que hicimos?

Dada esta versión alternativa con recursión explícita:

```
take' :: [a] -> Int -> [a]
take' [] _ = []
take' (x:xs) n = if n == 0 then [] else x : take' xs (n-1)
```

¿Podemos probar que `take'` $=$ `flipTake`?

¿Está bien lo que hicimos?

Tenemos:

```
take' :: [a] -> Int -> [a]
take' [] _ = []
take' (x:xs) n = if n == 0 then [] else x : take' xs (n-1)

flipTake :: [a] -> Int -> [a]
flipTake = foldr
     (\x rec n -> if n == 0 then [] else x : rec (n-1))
     (const [])

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] _ = z
foldr f z (x:xs) = f x (foldr f z xs)

const :: (a -> b -> a)
const = (\x ->\_ -> x)
```

Probemos que `take'` $=$ `flipTake`.

Un breve repaso

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

```
take' :: [a] -> Int -> [a]
take' [] _ = []
take' (x:xs) n = if n==0 then [] else x : take' xs (n-1)
```

```
listasQueSuman :: Int -> [[a]]
listasQueSuman 0 = [[]]
listasQueSuman n | n > 0 =
               [x : xs | x <- [1..], xs <- listasQueSuman (n-x)]
```

```
fact :: Int -> Int
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

```
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n | n > 1 = fibonacci (n-1) + fibonacci (n-2)
```

Funciones sobre árboles

Dado el tipo de datos:

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

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

```
insertarABB :: Ord a => a -> AB a -> [a]
insertarABB x Nil = Bin Nil x Nil
insertarABB x (Bin i r d) = if x < r
          then Bin (insertarABB x i) r d
          else Bin i r (insertarABB x d)
```

```
truncar :: AB a -> Int -> AB a
truncar Nil _ = Nil
truncar (Bin i r d) n = if n == 0 then Nil else
          Bin (truncar i (n-1)) r (truncar d (n-1))
```

Tarea: prueben escribir estas funciones con los esquemas de recursión correspondientes.

Demostrando propiedades sobre árboles

Dadas las siguientes funciones:

```
cantNodos :: AB a -> Int
cantNodos Nil = 0
cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
```

```
inorder :: AB a -> [a]
inorder Nil = []
inorder (Bin i r d) = (inorder i) ++ (r:inorder d)
```

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)
```

Queremos probar:

$$\forall \texttt{t::AB a}\,.\ \texttt{cantNodos t} = \texttt{length (inorder t)}$$

¿Y ahora qué hacemos?

¡Necesitamos un lema!

$\forall\,$`xs::[a]`$\,.\,\forall\,$`ys::[a]`$\,.\,$`length (xs++ys)` $=$ `length xs + length ys`

Una propiedad sobe árboles... y un lema sobre listas

Ahora sí:

```
cantNodos :: AB a -> Int
cantNodos Nil = 0
cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
```

```
inorder :: AB a -> [a]
inorder Nil = []
inorder (Bin i r d) = (inorder i) ++ (r:inorder d)
```

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)
```

Lema:

$\forall$ `xs::[a]` . $\forall$ `ys::[a]` . `length (xs++ys)` $=$ `length xs + length ys`

Queremos probar:

$\forall$ `t::AB a` . `cantNodos t` $=$ `length (inorder t)`

¡No nos olvidemos de probar el lema!

Demostremos el lema

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)
```

Lema:

$\forall$ `xs::[a]` . $\forall$ `ys::[a]` . `length (xs++ys)` $=$ `length xs + length ys`

Ejercicio integrador

Dadas las siguientes funciones:

```
altura :: AB a -> Int
altura Nil = 0
altura (Bin i r d) = 1 + max (altura i) (altura d)
```

```
truncar :: AB a -> Int -> AB a
truncar Nil _ = Nil
truncar (Bin i r d) n = if n == 0 then Nil else
        Bin (truncar i (n-1)) r (truncar d (n-1))
```

Y los siguientes lemas:

1. $\forall$ `t::AB a` . altura t $\geq$ 0 (Pueden probarlo en casa.)
2. $\forall$ `x::Int` . $\forall$ `y::Int` . x $\geq$ y $\Rightarrow$ min x y = min y x = y
3. $\forall$ `x::Int` . $\forall$ `y::Int` . $\forall$ `z::Int` . max (min x y) (min x z) = min x (max y z)
4. $\forall$ `x::Int` . $\forall$ `y::Int` . $\forall$ `z::Int` . z + min x y = min (z+x) (z+y)

Demostrar que $\forall$ `t::AB a` . $\forall$ `n::Int` . `n` $\geq$ `0` $\Rightarrow$
`(altura (truncar t n))` = `min n (altura t))`

Últimas preguntas

Si quisiéramos demostrar una propiedad sobre el tipo `Árbol23 a b` mediante inducción estructural:

```
data Árbol23 a b =
     Hoja a
   | Dos b (Árbol23 a b) (Árbol23 a b)
   | Tres b b (Árbol23 a b) (Árbol23 a b) (Árbol23 a b)
```

Para demostrar que vale $\forall\, \texttt{q::Árbol23 a b} \,.\, P(\texttt{q})$:

¿Cuál es o cuáles son los casos base?
¿Cuál es o cuáles son los casos inductivos? ¿Y la(s) hipótesis inductiva(s)?

Fin

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?