Programación Declarativa, 2021-1
Nota de clase 9: Programación Origami*

Javier Enríquez Mendoza Manuel Soto Romero

12 de noviembre de 2020
Facultad de Ciencias UNAM

Origami es el arte japonés de crear elegantes diseños usando dobleces en cualquier tipo de papel, de *ori* (doblez) y *kami* (papel). [5] Con base en este arte se crea un estilo de programación a partir de las llamadas funciones de plegado definidas sobre estructuras de datos recursivas.

El estudio de la programación origami es relevante, pues el desarrollo que se ha hecho sobre listas, ha demostrado ser de gran utilidad para mejorar la eficiencia de los programas, así como para simplificar el razonamiento ecuacional sobre éstos, tal y como se muestra en [1]. Revisaremos en esta nota la definición de dos funciones de plegado: `foldr` y `foldl` así como algunas propiedades sobre estos operadores.

9.1. Plegado en listas

Antes de definir algunas funciones de plegado sobre listas, analicemos brevemente los siguientes ejemplos:

Ejemplo 9.1. En el Código 1, se muestran dos funciones que suman y multiplican los elementos de una lista respectivamente. Las líneas 3 y 8 definen los casos base de las funciones, la suma en una lista vacía es 0 y el producto 1[1]. Los pasos recursivos se definen en las líneas 4 y 9, se suma o multiplica la cabeza de la lista con el resultado de la llamada recursiva al resto de la misma.

```
-- Suma los elementos de una lista de números enteros
sumaLst :: [Int] → Int
sumaLst [] = 0
sumaLst (x:xs) = x + (sumaLst xs)

-- Multiplica los elementos de una lista de números enteros
multLst :: [Int] → Int
multLst [] = 1
multLst (x:xs) = x * (multLst xs)
```

Código 1: Suma y multiplicación de los elementos de una lista

*Basada en *Programación Origami: Cómo doblar un Árbol*

```
Prelude> sumaLst [1,7,2,9]
19
Prelude> multLst [1,7,2,9]
126
```

Veamos una ejecución para estas llamadas:

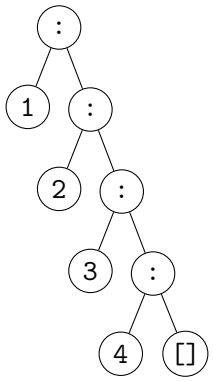
```
> sumaLst [1,7,2,9]
> 1 + sumaLst [7,2,9]
> 1 + 7 + sumaLst [2,9]
> 1 + 7 + 2 + sumaLst [9]
> 1 + 7 + 2 + 9 + sumaLst []
> 1 + 7 + 2 + 9 + 0
19


> multLst [1,7,2,9]
> 1 * multLst [7,2,9]
> 1 * 7 * multLst [2,9]
> 1 * 7 * 2 * multLst [9]
> 1 * 7 * 2 * 9 * multLst []
> 1 * 7 * 2 * 9 * 1
126
```

El patrón mostrado en el ejemplo anterior es muy común al definir funciones recursivas.

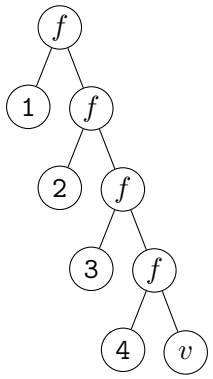
Ejercicio 9.1. Define la disyunción y conjunción de listas usando este patrón. Analiza detenidamente el patrón para la lista vacía.

Hint: Usa el neutro de cada operación.

Una forma de analizar este patrón es mediante su representación gráfica en forma de árbol que genera la construcción de una lista mediante el operador `cons` (`:`), por ejemplo, para la lista `[1,2,3,4]`, tenemos el siguiente árbol:



De esta forma, para definir funciones usando el patrón antes descrito, podemos reemplazar, la lista vacía por el neutro para dicha operación (v) y la operación `:` por la operación (función) correspondiente (f).



Observación 9.1. Es importante observar que la función f es binaria.

9.1.1. `foldr`

Así, podemos definir una *función de plegado* que tome una función binaria `(a → b → b)`, un valor de regreso para la lista vacía (`b`) y la lista a plegar (`[a]`). Llamamos a esta función `fold`. Existen distintas variantes para esta función, la más popular es la típica `foldr` que pliega hacia la derecha (de aquí viene la `r`), su definición se muestra a continuación:

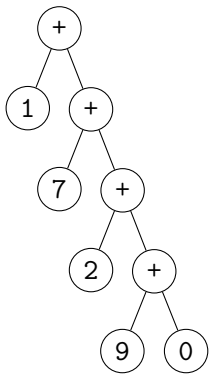
```
-- Función que aplica una función de forma encadenada a la derecha.
-- a los elementos de una lista dado un caso base.
foldr :: (a → b → b) → b → [a] → b
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

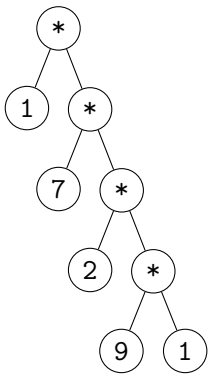
De esta forma, podemos reescribir nuestras funciones `sumaLst` y `multLst` como sigue:

```
-- Suma los elementos de una lista de números enteros
sumaLst :: [Int] → Int
sumaLst = foldr (+) 0

-- Multiplica los elementos de una lista de números enteros
multLst :: [Int] → Int
multLst = foldr (*) 1
```

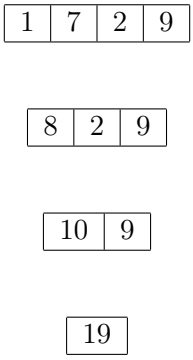
Notar que omitimos la lista debido a la currificación.

Se muestran a continuación los árboles de evaluación de ambas funciones con la lista `[1,7,2,9]`.





Si analizamos esta implementación, es como si hiciéramos origami con una hoja rectangular previamente doblada en partes iguales que corresponde con el número de elementos de una lista. Anotamos en cada parte el número correspondiente y cada que rehacemos el doblés, aplicamos la operación correspondiente. Por ejemplo, aplicando la suma a la lista `[1,7,2,9]`:



Ejercicio 9.2. Redefine las funciones para la disyunción y conjunción que definiste usando la función `foldr`.

9.1.2. `foldl`

Viendo este último ejemplo, es natural preguntarse, ¿Qué pasa si doblo la lista (hoja) hacia el lado contrario, es decir, de derecha a izquierda. Bueno, este patrón también existe y es una variante de nuestro ya estudiado `foldr`. Llamamos a esta función `foldl`. Su principal diferencia es por supuesto, el orden de evaluación, sin embargo, una peculiaridad de esta función es que es implementada usando la técnica de *recursión de cola* que nos permite optimizar el espacio en memoria al no generar excesivas llamadas a función pendientes y usar un único registro de activación a la vez.

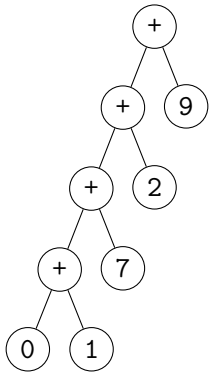
Se define a continuación la función `fold`, notar que los parámetros no cambian.

```
-- Función que aplica una función de forma encadenada a la derecha.
-- a los elementos de una lista dado un caso base.
foldl :: (a → b → b) → b → [a] → b
foldl _ v [] = v
foldr f v (x:xs) = foldl f (f v x) xs
```

Por ejemplo, veamos nuevamente el caso de la suma:

```
> foldl (+) 0 [1,7,2,9]
> foldl (+) (+ 0 1) [7,2,9]
> foldl (+) (+ (+ 0 1) 7) [2,9]
> foldl (+) (+ (+ (+ 0 1) 7) 2) [9]
> foldl (+) (+ (+ (+ (+ 0 1) 7) 2) 9) []
(+ (+ (+ (+ 0 1) 7) 2) 9)
19
```

Y el siguiente árbol de evaluación, observemos que el árbol de desarrolla en este caso hacia la izquierda.



Observación 9.2. El hecho de que estas dos funciones hayan regresado el mismo resultado con los ejemplos anteriores viene de que hemos usado operaciones que conmutan, como la suma y la multiplicación. Sin embargo, dependiendo de la operación (función) que usemos, los resultados no necesariamente serán los mismos.

Ejercicio 9.3. Redefine las funciones para la disyunción y conjunción que definiste usando la función `foldl`.

9.1.3. Ejemplos

Veamos otros ejemplos, usando estas definiciones.

`aplana` Aplana una lista de listas, es decir concatena las listas internas en una sola.

```
aplana :: [[a]] → [a]
aplana [] = []
aplana (x:xs) = x ++ aplana xs

aplana :: [[a]] → [a]
aplana = foldr (++) []

aplana :: [[a]] -> [a]
aplana = foldl (\x y -> x ++ y) []
```

`longitud` Calcula la longitud de una lista.

```
longitud :: [a] → Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

longitud :: [a] → Int
longitud = foldr (\_ y-> 1+y) 0

longitud :: [a] → Int
longitud = foldl (\x _-> x+1) 0
```

`concatena` Define la concatenación de dos listas.

```
concatena :: [a] → [a] → [a]
concatena [] l2 = l2
concatena (x:xs) l2 = x:(concatena xs l2)

concatena :: [a] → [a] → [a]
concatena l1 l2 = foldr (:) l2 l1

concatena :: [a] -> [a] -> [a]
concatena l1 l2 = foldl snoc l1 l2
```

9.1.4. Funciones que no se pueden reescribir con `fold`

Por supuesto no todas las funciones encapsulan estos dos patrones, de forma tal que no podemos reescribir todas las funciones recursivas como `foldr` o `foldl`. El más claro ejemplo se da en la función `rota` que, como su nombre indica, rota los elementos de una lista, por ejemplo `rota [1,2,3] = [2,3,1]`, y se define como sigue:

```
rota :: [a] → [a]
rota [] = []
rota (x:xs) = xs ++ [x]
```

Esta función no puede traducirse a una instancia de `foldr` ni `foldl` pues no se trata de una función ni siquiera de una función recursiva. Formalizaremos este problema más adelante en otra nota.

9.2. Propiedades del operador `foldr`

El operador `foldr` tiene un número importante de propiedades, para los fines de este curso, mostramos dos de ellas: La Propiedad Universal y El Principio de Fusión. Éstas son de suma importancia pues con ellas se puede generalizar cualquier otra propiedad sobre `foldr`. [5]

En general, cuando se habla de `fold`, se entiende que se hace referencia a la función `foldr`, sin importar el lenguaje de programación. A partir de este punto, escribiremos simplemente `fold`. Todo esto puede aplicarse de forma equivalente a `foldl`.

9.2.1. Propiedad Universal

La Propiedad Universal de la función `fold` tiene su origen en la Teoría de la Recursión.

Proposición 9.1. (Propiedad Universal de `fold` para listas) *La propiedad Universal se puede enunciar con la siguiente equivalencia entre las definiciones de una función g sobre listas:*

$$\begin{array}{lcl} \texttt{g []} & = & v \\ \texttt{g (x:xs)} & = & \texttt{f x (g xs)} \end{array} \iff \texttt{g = fold f v}$$

Demostración. Mostremos la equivalencia de estas definiciones.

($\Rightarrow$)

Por inducción estructural sobre listas

Caso base:

$$\begin{array}{rcll} \texttt{g []} & = & \texttt{fold f v []} & \\ \texttt{v} & = & \texttt{fold f v []} & def. \\ \texttt{v} & = & \texttt{v} & def. \end{array}$$

Hipótesis de Inducción

Supongamos que la propiedad se cumple para `xs`, es decir:

```
g xs = fold f v xs
```

Paso Inductivo

Por demostrar que `g (x:xs) = fold f v (x:xs)`

$$\begin{array}{rcll} \texttt{g (x:xs)} & = & \texttt{fold f v (x:xs)} & \\ \texttt{f x (g xs)} & = & \texttt{fold f v (x:xs)} & def. \\ \texttt{f x (fold f v xs)} & = & \texttt{fold f v (x:xs)} & h.i. \\ \texttt{f x (fold f v xs)} & = & \texttt{f x (fold f v xs)} & def. \end{array}$$

□

Ejercicio 9.4. Completa la demostración anterior mostrando que el regreso ($\Leftarrow$) se cumple.

Intuitivamente, lo que dice la Propiedad Universal del operador `fold` es que toda función definida sobre listas con este patrón recursiva es equivalente a una instancia del operado `fold`, tal y como vimos en los ejemplos anteriores.

El uso práctico de la Propiedad Universal es como un principio de prueba. De esta forma se puede evitar en algunos casos, el uso de Inducción para demostrar propiedades de expresiones que usen el operado `fold`.

Ejemplo 9.2. Demostrar la siguiente igualdad [5]

```
(1+) . suma = foldr (+) 1
```

Esta igualdad establece que es lo mismo sumar los elementos de una lista e incrementar el resultado que plegar la lista con `(+)` partiendo del valor 1. Dado que la igualdad es una instancia de la propiedad universal:

$$\underbrace{\texttt{(1+) . suma}}_{\texttt{g}} \;=\; \texttt{foldr}\;\; \underbrace{\texttt{(+)}}_{\texttt{f}}\;\; \underbrace{\texttt{1}}_{\texttt{v}}$$

basa con demostrar las precondiciones de ésta, es decir:

```
((1+) . suma) []     = 1
((1+) . suma) (x:xs) = x + (((1+) . suma) xs)
```

Eliminando la composición, tenemos:

```
1 + suma []     = 1
1 + suma (x:xs) = x + (1 + suma xs)
```

que pueden ser verificadas fácilmente

$$\begin{array}{rcll} \texttt{1 + suma []} & = & \texttt{1 + suma 0} & \textit{def.} \\ & = & \texttt{1 + 0} & \textit{def.} \\ & = & \texttt{1} & \textit{arit} \end{array}$$

$$\begin{array}{rcll} \texttt{1 + suma (x:xs)} & = & \texttt{1 + (x + suma xs)} & \textit{def.} \\ & = & \texttt{x + (1 + suma xs)} & \textit{conm.} \end{array}$$

En general, es posible demostrar la equivalencia de dos funciones sobre listas usando la Propiedad Universal si una de ellas puede ser expresada usando `foldr`.

9.2.2. Principio de Fusión

Una ecuación muy común al trabajar con funciones que operan sobre listas que pueden ser expresadas usando `foldr` es la siguiente:

```
h . foldr f w = foldr f z
```

La parte izquierda de la ecuación representa una función que primero realiza un pliegue con `f` y un valor neutro (base) `w` y al resultado de éste, le aplica la función `h`. Es claro que esta propiedad no es siempre cierta, sin embargo a partir de la Propiedad Universal de `foldr`, podemos verificar las propiedades necesarias para que esta ecuación sea cierta, es decir, las condiciones para `h`, `f`, `w` y `z`.

Dado que el lado derecho es un plegado, tenemos la siguiente equivalencia:

```
(h . foldr f w) []     = z
(h . foldr f w) (x:xs) = f x (((h . foldr f w) xs)
```

Aplicando la composición:

```
h (foldr f w [])     = z
h (foldr f w (x:xs)) = f x (h (foldr f w xs))
```

De la primera ecuación tenemos:

$$\texttt{h (foldr f w [])} \;=\; \texttt{z} \iff \texttt{h w = z} \quad def.$$

De la segunda ecuación tenemos:

$$\texttt{h (foldr f w (x:xs))} \;=\; \texttt{f x (h (foldr f w xs))}$$

$$\Longleftrightarrow$$

$$\texttt{h (f x (foldr f w xs)) = f x (h (foldr f w xs))}$$

reescribiendo

$$\texttt{h (f x ys) = f x (h ys)}$$

De esta forma, usando la Propiedad Universal, hemos llegado a una igualdad para listas finitas:

Teorema 9.1. (Teorema de fusión de `foldr`)

$$\begin{array}{lcl} h\ w & = & z \\ h\ (f\ x\ ys) & = & f\ x\ (h\ ys) \end{array} \Longrightarrow h\ .\ foldr\ f\ w = foldr\ f\ z$$

El Teorema se denomina de Fusión pues se fusiona la función de plegado `foldr f w` seguido de la función `h` en un único plegado `(foldr f z)`. Por ejemplo, veamos la siguiente ejecución:

```
> (h . foldr f w) [a,b,c]
> h (foldr f w [a,b,c])
> h (f a (f b (f c w)))
> f a (h (f b (f c w)))
> f a (f b (h (f c w)))
> f a (f b (f c (h w)))
> f a (f b (f c z))
> foldr f z [a,b,c]
```

Ejercicio 9.5. Realiza la fusión de la siguiente ecuación en un único plegado:

```
foldr f u . map g = foldr g z
```

Es decir, encuentra las incógnitas `g` y `z` para que la propiedad sea cierta.

Referencias

[1] Javier Enríquez, *Programación Origami: Cómo doblar un árbol*, Tesis de Licenciatura, Facultad de Ciencias UNAM, 2020.

[2] Manuel Soto, *Manual de Prácticas para la Asignatura de Programación Declarativa,* Facultad de Ciencias UNAM, Proyecto de Titulación, 2019.

[3] Jeremy Gibbons, *Origami Programming*, The Fun of Programming, 2011.

[4] Graham Hutton, *A tutorial on the university and expressivneness of fold.*, J. Functional Programming, 1999.

[5] José E. Gallardo, *Funciones de Plegado sobre listas,* Lección Magistral.