# Especificación de costos

Estructuras de Datos y Algoritmos II
Trabajo Práctico 2

Marzorati Denise
marzorati.denise@gmail.com

Soncini Nicolás
soncininicolas@gmail.com

8 de Junio de 2016

**Docentes de la materia**

Mauro Jaskelioff
Cecila Manzino
Juan M. Rabasedas
Martin Ceresa

# Implementación con Listas

## `filterS`

La implementación de *filterS* con listas hace uso del paralelismo para reducir su profundidad lo más posible en base a las aplicaciones de la función que se le pasa.

```
filterS      = filterL

filterL :: (a -> Bool) -> [a] -> [a]
filterL _ [] = []
filterL f (x:xs) = let (a,b) = f x ||| (filter f xs)
                   in if a then x:b else b
```

El `trabajo` de la función se puede tomar de la siguiente forma:

$$W\left(filterS\ f\ xs\right) \in O\left(\sum_{i=0}^{|xs|-1} W\left(f\ xs_i\right)\right)$$

De esta forma, se calcula la aplicación de la función argumento en cada llamada recursiva de *filterS* hasta que la lista termine vacía. En el mejor de los casos, si $W\left(f\ xs_i\right) \in O\left(1\right)$, su sumatoria queda como $|xs|$, por lo tanto se puede omitir sumar este valor en la cota superior del trabajo.

La `profundidad` de la función se puede reducir gracias a la aplicación en paralelo de la función argumento, por lo tanto se puede deducir que:

$$S\left(filterS\ f\ xs\right) \in O\left(|xs| + \max_{i=0}^{|xs|-1} S\left(f\ xs_i\right)\right)$$

En este caso, como el máximo de todos los trabajos puede quedar constante, debemos sumar el trabajo de obtener cada elemento, lo cual suma $|xs|$.

## `showtS`

La implementación de *showtS* con listas divide la lista en dos mitades y con las funciones *takeS* y *dropS*.

```
takeS xs n   = take n xs

dropS xs n   = drop n xs

showtS       = showtL

showtL :: [a] -> TreeView a [a]
showtL []  = EMPTY
showtL [a] = ELT a
showtL xs  = let
               n      = quot (lengthS xs) 2
               (l, r) = (takeS xs n) ||| (dropS xs n)
             in NODE l r
```

Sabemos que las funciones utilizadas son lineales:

$$W/S\left(takeS\ xs\ n\right) \in O\left(|xs|\right)$$
$$W/S\left(dropS\ xs\ n\right) \in O\left(|xs|\right)$$

tanto en su trabajo como en su profundidad.

Luego, podemos concluir fácilmente que valen los siguientes costos ya que se aplica una vez cada función sobre la mitad de la lista.

$$W\left(showtS\ xs\right) \in O\left(|xs|\right)$$

$$S\left(showtS\ xs\right) \in O\left(|xs|\right)$$

## reduceS

La implementación de *reduceS* toma una función de costo desconocido, un elemento y una lista del mismo tipo y aplica el algoritmo de reduce de forma que opera con la función dada $\oplus$ los elementos de la lista de a pares sobre el resultado de cada llamada recursiva. Este orden de reducción (u operación) es el dado por el TAD.

```
reduceS       = reduceL

reduceL :: (a -> a -> a) -> a -> [a] -> a
reduceL _ n [] = n
reduceL f n xs = f n (reduceL' f xs)

reduceL' :: (a -> a -> a) -> [a] -> a
reduceL' _ [x]       = x
reduceL' f [x,y]     = f x y
reduceL' f xs        = reduceL' f (contraer f xs)
```

Ahora, a la hora de calcularlos para *reduceS*, nosotros debemos forzar el orden de reducción, ya que la implementación de *contraer* utiliza un orden que no cumple la especificación necesaria, pero es fácil identificar el rol que cumple al ser bien integrado a *reduceS*.

Como primera función auxiliar se encuentra *contraer*, que dada una función y una lista, opera de a pares sus elementos, tomando las llamadas recursivas y las aplicaciones de la función en forma paralela.

```
contraer :: (a -> a -> a) -> [a] -> [a]
contraer _ []        = []
contraer _ [x]       = [x]
contraer f (x:y:xs) = let (a,b) = (f x y) ||| (contraer f xs)
```

Debemos entonces calcular los costos de *contraer* para poder calcular los de la función pedida, ésta queda como el costo de recorrer toda la lista sumado a los costos de las operaciones dos a dos. Pero ésta ultima, en el caso que todas las operaciones sean constantes, ya queda acotada por la longitud de la lista, luego podemos obviar éste parametro al calcular su trabajo:

$$W\left(contraer\ \oplus\ xs\right) \in O\left(\sum_{i=0}^{(|xs/2|)-1} W\left(xs_{2i} \oplus xs_{2i+1}\right)\right)$$

Lo mismo no sucede en su profundidad, ya que se suma el máximo de las profundidades de las operaciones, con lo cual éstas pueden quedar constantes, pero el trabajo está acotado por el trabajo de recorrer la lista (que es lineal sobre su longitud):

$$S\left(contraer\ \oplus\ xs\right) \in O\left(|xs| + \max_{i=0}^{(|xs/2|)-1} S\left(xs_{2i} \oplus xs_{2i+1}\right)\right)$$

Es claro entonces el costo que aporta adecuar esta función para que trabaje sobre cada resultado de un llamado recursivo de si mismo, cumpliendo así el orden de reducción pedido por el TAD de Secuencias para *reduceS*.

Ahora sí podemos calcular el trabajo de *reduceS*, el cual queda de la siguiente manera:

$$W\left(reduceS \oplus\ b\ xs\right) \in O\left(|xs| + \sum_{(xs_i \oplus xs_j) \in \mathcal{O}_r(\oplus, b, xs)} W\left(xs_i \oplus xs_j\right)\right)$$

Tanto su `trabajo` como su `profundidad` se ven afectadas por el reordenamiento de reducción sobre el resultado de *contraer*, con lo cual la paralelización de ésta no puede ser aprovechada, y obtenemos:

$$S\left(reduceS \oplus\ b\ xs\right) \in O\left(|xs| + lg\ |xs| \max_{(xs_i \oplus xs_j) \in \mathcal{O}_r(\oplus, b, xs)} S\left(xs_i \oplus xs_j\right)\right)$$

## scanS

La implementación de `scanS` hace uso de varias funciones auxiliares para su correcto funcionamiento y su fácil comprensión.

```
scanS         = scanL

scanL :: (a -> a -> a) -> a -> [a] -> ([a], a)
scanL f n xs = (scanL' f n xs) ||| (reduceS f n xs)

scanL' :: (a -> a -> a) -> a -> [a] -> [a]
scanL' _ n [_] = [n]
scanL' f n xs  = expandir f xs (scanL' f n (contraer f xs))
```

Para obtener el costo del mismo debemos primero describir y especificar los costos de las funciones que lo auxilian.

Como primera función auxiliar se encuentra *contraer*, que dada una función y una lista, opera de a pares sus elementos, tomando las llamadas recursivas y las aplicaciones de la función en forma paralela. (Se detallan los costos en la especificación de costos de reduceS).

Luego hacemos uso de la función *expandir*, que dada una función y dos listas (la primera siendo la lista argumento de *scanS*, y la segunda el resultado de aplicarle a la primera la función *contraer*) retorna una única lista que combina ambas:

```
expandir :: (a -> a -> a) -> [a] -> [a] -> [a]
expandir _ [] []           = []
expandir _ [_] zs          = zs
expandir f (x:_:xs) (z:zs) = let (a,b) = (f z x) ||| (expandir f xs zs)
```

$$W\left(expandir\ \oplus\ xs\ zs\right) \in O\left(\sum_{i=0}^{(|xs|)/2-1} W\left(xs_{\lfloor (2i+1)/2 \rfloor} \oplus zs_{2i}\right)\right)$$

$$S\left(expandir\ \oplus\ xs\ zs\right) \in O\left(|xs| + \max_{i=0}^{(|xs|)/2-1} S\left(xs_{\lfloor (2i+1)/2 \rfloor} \oplus zs_{2i}\right)\right)$$

Podemos concluir de esta forma, dado que la función *scanS* para listas llama a expandir sobre una recursión de aplicar contraer a la lista dada, que su `trabajo` es por lo menos lineal, que se calcula como:

$$W\left(scanS \oplus\ b\ xs\right) \in O\left(|xs| + \sum_{(xs_i \oplus xs_j) \in \mathcal{O}_s(\oplus, b, xs)} W\left(xs_i \oplus xs_j\right)\right)$$

Y dado que las profundidades de *expandir* y *contraer* son como mínimo lineales, la `profundidad` de *scanS* queda como la suma del tamaño de la lista y el máximo de las profundidades de la aplicación de la función sobre las sub-aplicaciones en el orden de reducción dado. Con lo cual tenemos:

$$S\left(scanS \oplus\ b\ xs\right) \in O\left(|xs| + lg\ |xs| \max_{(xs_i \oplus xs_j) \in \mathcal{O}_s(\oplus, b, xs)} S\left(xs_i \oplus xs_j\right)\right)$$

# Implementación con Arreglos Persistentes

## `filterS`

La función $filterS$ toma una lista y una función sobre elementos de la misma y hace uso de $tabulateS$ para aplicar la función sobre cada elemento y $joinS$ para unir los resultados sueltos de la aplicación anterior.

```
filterS     = filterA

filterA  :: (a -> Bool) -> A.Arr a -> A.Arr a

filterA f xs = joinS (tabulateS func (lengthS xs))
               where func = (\i -> if f (nthS xs i) then singletonS (nthS xs i) else emptyS)
```

Sabemos que los costos de $tabulateS$ y $joinS$ para arreglos persistentes son:

$$W\left(tabulateS\ f\ n\right) \in O\left(\sum_{i=0}^{n-1} W\left(f\ i\right)\right)$$

$$S\left(tabulateS\ f\ n\right) \in O\left(\max_{i=0}^{n-1} S\left(f\ i\right)\right)$$

y

$$W\left(joinS\ as\right) \in O\left(|as| + \sum_{i=0}^{|as|-1} \left(|as_i|\right)\right)$$

$$S\left(joinS\ \ as\right) \in O\left(\ lg\ |as|\ \right)$$

De ambas se puede calcular que el `trabajo` de $filterS$ queda la suma de la longitud del arreglo y la sumatoria del costo de trabajo de las aplicaciones de la función a cada elemento. En el mejor de los casos, la función utilizada es de costo constante, con lo cual ya tendríamos el costo lineal sobre la longitud del arreglo en ese caso. De esto se puede omitir el tamaño del arreglo, con lo que nos quedaría:

$$W\left(filterS\ f\ as\right) \in O\left(\sum_{i=0}^{|as|-1} W\left(f\ as_i\right)\right)$$

Ahora, al momento de calcular su `profundidad` es una simple suma de las profundidades de $joinS$ y $tabulateS$, y no puede omitirse ninguna ya que cualquiera de ellas puede tomar un valor mayor a la otra, dependiendo del costo de las funciones que utiliza tabulate. Luego podemos observar que:

$$S\left(filterS\ f\ as\right) \in O\left(\ lg\ |as| + \max_{i=0}^{|as|-1} S\left(f\ as_i\right)\right)$$

## showtS

Para la implementación de *showtS* para este formato debemos partir al arreglo en dos partes de forma que nos de una rapida idea de la forma en árbol del mismo. Para llevar a cabo esto hacemos uso de las funciones ya definidas para arreglos persistentes *takeS* y *dropS*, que devuelven un arreglo con los primeros o últimos elementos del arreglo según una cantidad arbitraria respectivamente.

```
takeS xs n = A.subArray 0 n xs

dropS xs n = A.subArray n (lengthS xs - n) xs

showtA :: A.Arr a -> TreeView a (A.Arr a)
showtA xs
    | lengthS xs == 0 = EMPTY
    | lengthS xs == 1 = ELT (nthS xs 0)
    | otherwise       = NODE (takeS xs len) (dropS xs len)
      where len = quot (lengthS xs) 2
```

Para considerar los costos de *showtS* sabemos como primera instancia que los costos de las funciones auxiliares son constantes:

$$W/S\,(takeS\ as\ n) \in O\,(1)$$

$$W/S\,(dropS\ as\ n) \in O\,(1)$$

Y como tomar su longitud también lo es (útil para el cálculo de $2^{ilog(|as|-1)}$), tenemos entonces que los costos para la función *showtS* también quedan constantes:

$$W\,(showtS\ as) \in O\,(1)$$

$$S\,(showtS\ as) \in O\,(1)$$

## reduceS

Para calcular los costos de la función *reduceS* tenemos que hacerlo en función a los costos que posee *contraer*, ya que es la única función auxiliar que llama en su recursión y la única operación (de costo significante) que realiza.

```
reduceS     = reduceA

reduceA f n xs
    | lengthS xs == 0 = n
    | otherwise       = f n (reduceA' f xs)

reduceA' :: (a -> a -> a) -> A.Arr a -> a
reduceA' f xs
    | lengthS xs == 1  = nthS xs 0
    | otherwise        = reduceA' f (contraer f xs)

contraer f xs
    | even len   = tabi
    | odd len    = appendS tabi (singletonS (nthS xs (len-1)))
      where len  = lengthS xs
            tabi = tabulateS (\i -> f (nthS xs (2*i)) (nthS xs (2*i + 1))) (quot len 2)
```

El costo de la función *contraer* se calcula, para su `trabajo` como la suma de las aplicaciones de la función binaria, ya que viene dada por un *tabulateS* cuyo costo es éste mismo, y por la misma razon la `profundidad` es su máximo:

$$W\left(contraer\ \oplus\ as\right) \in O\left(\sum_{i=0}^{(|as/2|)-1} W\left(as_{2i} \oplus as_{2i+1}\right)\right)$$

$$S\left(contraer\ \oplus\ as\right) \in O\left(\max_{i=0}^{(|as/2|)-1} S\left(as_{2i} \oplus as_{2i+1}\right)\right)$$

Como la función *reduceS* se llama recursivamente sobre el resultado de *contraer*, la cual devuelve un arreglo con la mitad de elementos, podemos ver que ésta debe entonces recorrer el arreglo en cada paso recursivo, dejándo asi un costo lineal sobre la longitud del mismo en su costo de `trabajo`, sumado obviamente a los costos de trabajo de cada aplicación de la función binaria:

$$W\left(reduceS \oplus\ b\ as\right) \in O\left(|as| + \sum_{(as_i \oplus as_j) \in \mathcal{O}_r(\oplus, b, as)} W\left(as_i \oplus as_j\right)\right)$$

A la hora de calcular su `profundidad`, sabemos que llama a la función contraer $lg\ |as|$ veces, a esto sumamos la máxima profundidad de las aplicaciones del argumento primero sobre el orden de reducción dado por sus llamadas recursivas. Con lo cual nos queda:

$$S\left(reduceS \oplus\ b\ as\right) \in O\left(lg\ |as| * \max_{(as_i \oplus as_j) \in \mathcal{O}_r(\oplus, b, xs)} S\left(as_i \oplus as_j\right)\right)$$

## scanS

Por último debemos calcular las cotas superiores de *scanS*, para esto tenemos que entender el procedimiento que realiza. La función *scanS* en cada paso recursivo hace una expansión de llamarse a ella misma sobre el resultado de realizar una contracción (esto nos asegura que se realizarán igual cantidad de expansiones y contracciones).

```
scanS       = scanA

scanA :: (a -> a -> a) -> a -> A.Arr a -> (A.Arr a, a)
scanA f n xs
    | lengthS xs == 0 = (singletonS n, n)
    | lengthS xs == 1 = (singletonS n, f (nthS xs 0) n)
    | otherwise       = expandir f xs (scanA f n (contraer f xs))
```

Durante la contracción se toma al arreglo de valores y se lo opera dos a dos, dejando así un arreglo de la mitad de la longitud. El costo de esta función se expresa en un punto anterior.

En la expansión se toma un arreglo y una tupla (que las llamadas recursivas dentro de *scanS* dejan llena con un arreglo en la primera posición y la operación de todos los elementos en la segunda) y los opera y reordena según un algoritmo ya visto. De esta forma deja un resultado intermedio útil para otra llamada de expandir, o el resultado final de *scanS* en la última llamada.

```
expandir :: (a -> a -> a) -> A.Arr a -> (A.Arr a, a) -> (A.Arr a, a)
expandir f xs (zs, z) = (tabulateS func (lengthS xs), z)
    where func = (\i -> if even i then (nthS zs (quot i 2)) else f (nthS zs (quot i 2)) (nthS xs (i-1)))
```

Como *expandir* utiliza un tabulate con funciones constantes en su argumento, podemos deifnir sus costos como:

$$W\left(expandir\ \oplus\ as\ bs\right) \in O\left(\sum_{i=0}^{(|as|)/2-1} W\left(as_{\lfloor (2i+1)/2 \rfloor} \oplus bs_{2i}\right)\right)$$

$$S\left(expandir\ \oplus\ as\ bs\right) \in O\left(\max_{i=0}^{(|bs|)/2-1} S\left(as_{\lfloor(2i+1)/2\rfloor} \oplus bs_{2i}\right)\right)$$

Tenemos entonces, dado que cada llamada de *scanS* sobre un arreglo de una longitud dada realiza dos llamadas de funciones (una de *contraer* y otra de *expandir*) con costos idénticos, y una llamada recursiva sobre un arreglo de la mitad de la longitud, que su costo de `trabajo` es:

$$W\left(scanS \oplus\ b\ as\right) \in O\left(|as| + \sum_{(as_i \oplus as_j) \in \mathcal{O}_r(\oplus, b, as)} W\left(as_i \oplus as_j\right)\right)$$

Con el mismo criterio, se llama a ambas funciones en forma del árbol de reduccion, con lo cual obtenemos una profundidad del máximo de todas las aplicaciones de éste árbol (que realizan en conjunto *contraer* y *expandir*) multiplicado por la profundidad del mismo (logarítmica):

$$S\left(scanS \oplus\ b\ as\right) \in O\left(lg\ |as|\ \max_{(as_i \oplus as_j) \in \mathcal{O}_r(\oplus, b, as)} S\left(as_i \oplus as_j\right)\right)$$