Especificación de Costos de ListSeq y ArrSeq

Meli Sebastián. Rodríguez Jeremías.

Estructuras de Datos y Algoritmos II - Trabajo Práctico

2 de junio de 2015

Índice

1. Especificación de Costos para ListSeq

1.1. FilterS

La implementación presentada es:

```
1 filterS p []     = []
2 filterS p (x:xs) = if cond then (x:xs') else xs'
3   where (cond,xs') = (p x) ||| (filterS p xs)
```

La función FilterS recibe como argumento una lista de tamaño n, y un predicado p. De la implementación en Haskell, se desprende la siguiente ecuación de recurrencia:

- Suponiendo que $W_p \in \mathcal{O}(1)$, $S_p \in \mathcal{O}(1)$:

$$\begin{aligned} W_{filterS}(0) &= c_1 \\ W_{filterS}(n) &= c_2 + W_{filterS}(n-1) \end{aligned}$$

$$W_{filterS} \in \mathcal{O}(n)$$

$$\begin{aligned} S_{filterS}(0) &= c_1 \\ S_{filterS}(n) &= c_2 + S_{filterS}(n-1) \end{aligned}$$

$$S_{filterS} \in \mathcal{O}(n)$$

  En la última ecuación, el máximo entre $S_{filterS}$ (n-1) y una constante (aplicar p) es $S_{filterS}$ (n-1).

- En forma más general, si W_p, S_p son el trabajo y profundidad asociados a p; y la secuencia a filtrar es de la forma $\langle x_1, x_2, ..., x_n \rangle$, entonces:

$$W_{filterS} \in \mathcal{O}(\textstyle\sum_{i=1}^{n} W_p(x_i))$$

  Pues aplicamos n veces el predicado, más el costo constante de cons, etc.

$$S_{filterS} \in \mathcal{O}(n + \max_{1 \leq i \leq n} S_p(x_i))$$

  En cuanto a la profundidad, los cálculos de $p(x_i)$ se realizan todos en paralelo. Además tenemos n veces costos constantes, provenientes de realizar comparaciones, if then else, y usar cons para armar la nueva secuencia.

1.2. showT

La implementación que presentamos es:

```
1 showtS []       = EMPTY
2 showtS (x:[])   = ELT x
3 showtS xs       = let (l,r)= take n xs ||| drop n xs in (NODE l r)
4   where n=div (length xs) 2
```

Damos por asumido que take n xs, drop n xs tienen trabajo y profundidad $\mathcal{O}(n)$; y que length xs tiene trabajo y profundidad $\mathcal{O}(|xs|)$.

Luego, si n es la longitud del argumento de showtS:

$$W_{showtS}(n) = \underbrace{c_1}_{constantes} + \underbrace{c_2 * n}_{length} + \underbrace{c_3 * n}_{take} + \underbrace{c_4 * n}_{drop}$$

$$W_{showtS} \in \mathcal{O}(n)$$

En cuanto a la profundidad, la ecuación es de la forma:

$$S_{showtS}(n) = \underbrace{c_1}_{constantes} + \underbrace{c_3 * n}_{take|||drop} + \underbrace{c_4 * n}_{drop}$$

$$S_{showtS} \in \mathcal{O}(n)$$

1.3. reduceS

1.3.1. Funcion auxiliar

Nuestra implementación de reduce utiliza una función auxiliar contraer. En primer lugar analiaremos su trabajo y profundidad:

```
1 contraer op []         = []
2 contraer op (x:[])     =  x:[]
3 contraer op (x:y:xs)   = let (x',xs') = (op x y) ||| (contraer op xs) in (x':xs')
```

- Suponiendo op constante, y siendo n la longitud de la secuencia a contraer:

$$\begin{aligned} W_{contraer}(n) &= \underbrace{c_1}_{let,cons,etc} + W_{contraer}(n-2) + \underbrace{c_2}_{op\ x\ y} \\ S_{contraer}(n) &= \underbrace{c_1}_{let,cons,etc} + \underbrace{S_{contraer}(n-2)}_{max(c_2,S_{contraer}(n-2))} \end{aligned}$$

  Luego, $W_{contraer} \in \mathcal{O}(n),\ S_{contraer} \in \mathcal{O}(n)$

- En el caso general (op no es constante), si la secuencia argumento de reduce es de la forma $\langle x_1, x_2, ..., x_n \rangle$, las ecuaciones son de la forma:

$$\begin{aligned} W_{contraer}(n) &= \sum_{i=1}^{\lfloor n/2 \rfloor} (W_{op}(x_{2i}, x_{2i+1}) + c_i) \\ S_{contraer}(n) &= n + \max_{1 \le i \le \lfloor n/2 \rfloor} S_{op}(x_{2i}, x_{2i+1})) \end{aligned}$$

  Formalmente, usando la notación $\mathcal{O}_c$(contraer op s) = {aplicaciones de op al utilizar contraer}

$$\begin{aligned} W_{contraer}(n) &\in \mathcal{O}\left(\sum_{(op\ x\ y)\ \in\ \mathcal{O}_c(op\ s)} W(op\ x\ y) \right) \\ S_{contraer}(n) &\in \mathcal{O}\left(n + \max_{(op\ x\ y)\ \in\ \mathcal{O}_c(op\ s)} S(op\ x\ y) \right) \end{aligned}$$

  El n sumando que aparece en la profundidad proviene de que, aunque hagamos todos los cálculos de op en paralelo, necesariamente deberemos recorrer la lista para esto. Y ello implica un costo lineal.

1.3.2. Analisis de reduceS

Ahora analizamos la función reduceS:

```
1 reduceS op e []                = e
2 reduceS op e (x:[])            = op e x
3 reduceS op e xs                = reduceS op e (contraer op xs)
```

De la implementación resulta:

$$\begin{aligned} W_{reduce}(n) &= W_{contraer}(n) + W_{reduce}(\lfloor n/2 \rfloor) \\ S_{reduce}(n) &= S_{contraer}(n) + S_{reduce}(\lfloor n/2 \rfloor)) \end{aligned} \tag{1}$$

- En el caso op constante

  $$W_{reduce}(n) = W_{reduce}(\lfloor n/2 \rfloor) + c * n.$$

  Quitando los pisos (Podemos quitar los pisos por propiedad de suavidad), y aplicando el teorema maestro[1] :

  $$W_{reduce} \in \mathcal{O}(n)$$

  La profundidad es igual, pues contraer tiene el mismo orden de trabajo y profundidad.

- Si la operación no es constante, para calcular W_{reduce} observamos que si extendemos la ecuación (1), lo que estaremos haciendo es aplicar muchas veces la función contraer, cada una de estas aplicaciones a una lista de tamaño aproximadamente la mitad y sobre los resultados de la contracción anterior. Y cada llamada a la función contraer, aplicará en realidad muchas veces la función op mientras recorre la secuencia.

  Sea $\mathcal{O}_r(op, e, s) = \{$Aplicaciones de op durante el cálculo de reduce op e s$\}$, que es la unión de todos los conjuntos de operaciones realizadas en cada llamada a contraer (los llamamos $\mathcal{O}_c$ en la sección anterior). Usando esta notación podemos expresar W_{reduce} y S_{reduce} de la siguiente forma:

  $$\begin{aligned} W_{reduce}(n) &\in \mathcal{O}\left(n + \sum_{(op\ x\ y)\ \in\ \mathcal{O}_r(op\ e\ s)} W(op\ x\ y)\right) \\ S_{reduce}(n) &\in \mathcal{O}\left(n + lg\ n * \max_{(op\ x\ y)\ \in\ \mathcal{O}_r(op\ e\ s)} S(op\ x\ y)\right) \end{aligned}$$

  En el trabajo, lo que hacemos con reduce es una cantidad logaritmica de llamadas a contraer, cada una de ellas con una entrada de tamaño la mitad que la anterior. El trabajo total va a ser:

  1. La suma de todas las op realizadas en todas las llamadas a contraer
  2. El costo de recorrer cada secuencia s_i resultado de la contracción i-ésima, que es (n/1+n/2+n/4+n/8 ... + 1)<= c.n
  3. El costo (constante) de llamar recursivamente a reduce y contraer (pasar de una lista a otra), log n veces.

  El segundo y tercer item son el motivo del +n en el trabajo, y el primer item de la sumatoria.

  En cuanto a la profundidad: de cada llamada a contraer (entre la cantidad logarítmica de ellas que hay), solo sumamos el máximo de esa llamada. Podemos acotar este máximo de cada llamada a contraer por el máximo de todas las op realizadas. De allí sale el (log n) multiplicado por el máximo de todas las op. Además, el mismo +n que arrastramos por el item 2 del trabajo.

[1] $T(n) = aT(\frac{n}{b}) + f(n)$ donde $f(n) \in \mathcal{O}(n)$, $a = 1$, $b = 2$. Debemos comparar $n^{log_b a} = n^0$ con n^1. Como $n^1 \in \Omega(n^{0+\epsilon})$ para $\epsilon = 1$,y $\frac{n}{2} \leq n \Rightarrow T(n) \in \mathcal{O}(n)$.

1.4. scanS

1.4.1. Implementación

```
1  scanS op e [] =  ([],e)
2  scanS op e [x]=  ([e],op e x)
3  scanS op e s  =  (xs, snd s')
4    where (s',n)              = scanS op e (contraer op s) ||| lengthS s
5    xs                 = e
6
7  contraer op []         = []
8  contraer op (x:[])     =  x:[]
9  contraer op (x:y:xs)  = let (x',xs') = (op x y) ||| (contraer op xs) in (x':xs')
10
11 expandir _ _ _ i n             | i==n   = []
12 expandir op (c:cs) ss 0 n                = c : (expandir op (c:cs) ss 1 n)
13 expandir op (c:cs) ss i n      | even i = c : (expandir op (c:cs) (drop 1 ss) (i+1)
       n )
14 expandir op (c:cs) (s:ss) i n | odd i  = (op c s) : (expandir op cs ss (i+1) n)
```

La idea de como funciona nuestra función scanS aplicada a una secuencia s=$\langle x_1, x_2, \ldots, x_n \rangle$, operación $\oplus$ y caso base b es: (Pensamos en $\oplus$ asociativa y n par en el ejemplo)

1. Contraer s, obteniendo una secuencia de la forma:

$$s_c = \langle x_1 \oplus x_2,\ x_3 \oplus x_4, \ldots,\ x_{n-1} \oplus x_n \rangle$$

2. Recursivamente, aplicar scanS a s_c, obteniendo s':

$$s' = (\langle b, (b \oplus x_1 \oplus x_2), (b \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4), \ldots,\ (b \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-2}) \rangle, b \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \oplus x_n)$$

3. Generamos, usando la función expandir en s' y s, el resultado final de scanS s. Esto lo hacemos observando que, snd (scanS s) = snd s'; y fst (scanS s) es una secuencia tal que su elemento en la posición i-ésima es:

   - Si i es par, el elemento i/2 de s'
   - Si i es impar, el elemento $\lfloor i/2 \rfloor$ de s' operado con el elemento $\lfloor i/2 \rfloor$ de s.

   expandir va generando los elementos de scanS s uno a la vez. El argumento adicional que recibe, un numero entero, indica el índice del elemento que se esta generando en esa llamada. Por ese motivo es llamada en 0 desde scanS, para que genere la secuencia desde el primer elemento y recursivamente genere el resto de la secuencia. Está comentada en el archivo .hs con más detalle.

1.4.2. Complejidad de funciones auxiliares

El trabajo y la profundidad de contraer fueron analizados en la sección de reduceS.
En cuanto a expandir, de forma análoga a contraer llegamos al siguiente resultado:

- De forma general, siendo n la longitud de la secuencia sin contraer:

$$\begin{aligned} W_{expandir}(n) &\in \mathcal{O}\left(n + \sum_{(x\oplus y)\ \in\ \mathcal{O}_e(\oplus\ s\ s')} W(x \oplus y)\right) \\ S_{expandir}(n) &\in \mathcal{O}\left(n + \max_{(x\oplus y)\ \in\ \mathcal{O}_e(\oplus\ s\ s')} S(x \oplus y)\right) \end{aligned}$$

  Donde $\mathcal{O}_e(\oplus\ s\ s')$ son todas las aplicaciones de $\oplus$ que se hacen al llamar a extender con s s' $\oplus$ y 0. (Las operaciones que se realizan para armar los elementos de índice impar en la secuencia de sumas parciales).

- Si $\oplus$ es O(1), entonces:

$$\begin{aligned} W_{expandir}(n) &\in \mathcal{O}(n) \\ S_{expandir}(n) &\in \mathcal{O}(n) \end{aligned}$$

1.4.3. Complejidad de scanS

Nuestra implementación ya presentada de scanS es:

```
1 scanS op e [] =  ([],e)
2 scanS op e [x]=  ([e],op e x)
3 scanS op e s  =  (xs, snd s')
4   where (s',n)             = scanS op e (contraer op s) ||| lengthS s
5   xs                 = expandir op (fst s') s 0 n
```

De donde resulta la siguiente recurrencia, siendo n la longitud de la secuencia a la que se le aplica scan:

$$\begin{aligned} W_{scanS}(n) &= \underbrace{c_1 * n}_{lengthS,etc} + W_{contraer}(n) + W_{expandir}(n) + W_{scanS}(\lfloor n/2 \rfloor) \\ S_{scanS}(n) &= \underbrace{c_1}_{constantes} + S_{contraer}(n) + S_{expandir}(n) + S_{scanS}(\lfloor n/2 \rfloor) \end{aligned}$$

- Si $\oplus$ es constante, las ecuaciones se reducen a:

$$\begin{aligned} W_{scanS}(n) &= \bar{c_1} * n + W_{scanS}(\lfloor n/2 \rfloor) \\ S_{scanS}(n) &= \bar{c_2} * n + S_{scanS}(\lfloor n/2 \rfloor) \end{aligned}$$

  De donde se concluye:

$$\begin{aligned} W_{scanS}(n) &\in \mathcal{O}(n) \\ S_{scanS}(n) &\in \mathcal{O}(n) \end{aligned}$$

- Si $\oplus$ es una operación cualquiera, y $\mathcal{O}_s(\oplus\ s\ b)$ son todas las operaciones de $\oplus$ en scan:

$$
\begin{aligned}
W_{scanS}(n) &\in \mathcal{O}\left(n + \sum_{(x \oplus y) \in \mathcal{O}_s(\oplus\ s\ b)} S(x \oplus y)\right) \\
S_{scanS}(n) &\in \mathcal{O}\left(n + lg\ n * \max_{(x \oplus y) \in \mathcal{O}_s(\oplus\ s\ b)} S(x \oplus y)\right)
\end{aligned}
$$

El razonamiento es el mismo que con reduce. La diferencia es que aquí utilizmos además una función expander por cada llamada a la función contraer. Como contraer y expander tienen los mismos costos, aplicar ambas significa un aumento en la constante oculta pero no en el orden de complejidad de scan respecto al orden de complejidad de reduce.

2. Especificación de costos para arreglos persistentes

2.1. Filter

```
1  filterS f s    = joinS (tabulateS g n)
2                   where
3                     n    = lengthS s
4                     g i = if f elem then (singletonS elem) else emptyS
5                      where elem = (s!i)
```

Pensando en predicado constante:

$$W_{filter}(n) = \underbrace{n}_{tabulate} + \underbrace{c_1}_{constantes,length,singleton,!} + \underbrace{n}_{join} \Rightarrow \mathcal{O}(n)$$
$$S_{filter}(n) = c_1 + \underbrace{lg\ n}_{join} \Rightarrow \mathcal{O}(lg\ n)$$

Si el predicado no es constante, la diferencia con el caso constante está en que el tabulate tendrá como trabajo la sumatoria de las aplicaciones del predicado; y en la profundidad el máximo.

$$W_{filter}(n) \in \mathcal{O}(\textstyle\sum_{i=1}^{n} W(px_i))$$
$$S_{filter}(n) \in \mathcal{O}(lg\ n + \text{máx}_{i=1}^{n}\, S(px_i))$$

2.2. ShowT

```
1  showtS s       | n==0 = EMPTY
2                 | n==1 = ELT (s!0)
3                 | n> 1 = NODE l r
4                     where n = lengthS s
5                           l = takeS s (div n 2)
6                           r = dropS s (div n 2)
```

ShowT utiliza solo funciones de trabajo constante una cantidad constante de veces, por lo que es constante. No paraleliza nada, su profundidad es también constante.

2.3. Reduce

```
reduceS op e s | n == 0 = e
               | n == 1 = op e (s!0)
               | n >  1 = reduceS op e t
                 where
                     n = lengthS s
                     t = contraer op s

contraer op s = tabulate g m
              where n   = lengthS s
                    m   = div (n + 1) 2
                    g i | i==(m-1)  = if (odd n) then (s!(n-1)) else elem
                        | otherwise = elem
                              where elem = op (s!(2*i)) (s!(2*i+1))
```

La idea de reduce es exactamente la que hicimos en listas: contraer la secuencia argumento y aplicar recursivamente reduce.

- Contraer solo utiliza en su implementación un tabulate cuya función argumento aplica op a dos elementos. Tendrá como trabajo la suma del trabajo del cálculo de op en cada uno de los pares; y profundidad al máximo de esos cálculos.
- Luego, lo que hacemos en reduce es aplicar sucesivamente contraer sobre el resultado de la contracción anterior hasta quedar con un solo elemento. De esto concluímos:

  1. El trabajo será la suma de todas las operaciones realizadas en la reducción. El costo constante de las llamadas recursivas se realiza log n veces, pero esto es menor a la suma de todas las operaciones, por lo que:

  $$W_{reduce}(s\ e\ \oplus) \in \mathcal{O}(\sum_{x \oplus y \in \mathcal{O}_r(s\ e\ \oplus)} W(x \oplus y))$$

  2. En cada contracción, todas las operaciones se realizan en paralelo. Tenemos una cantidad logarítmita de contracciones, y en cada una de ellas sumamos el máximo de los costos de las operaciones en esa contracción. Podemos acotar esta suma de log n sumandos hallando el máximo de todos los costos de las aplicaciones de $\oplus$ en la reducción:

  $$S_{reduce}(s\ e\ \oplus) \in \mathcal{O}(lg\ n * \text{máx}_{x \oplus y \in \mathcal{O}_r(s\ e\ \oplus)} S(x \oplus y))$$

  Obs: Seguimos usando la notación $\mathcal{O}_r(s\ e\ \oplus)$ = aplicaciones de $\oplus$ en la reducción de s. La idea es la misma que en listas, pero los costos son mejores con arreglos porque no arrastramos el costo de recorrer una lista.

2.4. Scan

```
scanS op e s    | n == 0 = (emptyS,e)
                | n == 1 = (singletonS e, op e (s!0) )
                | n  > 1 = xs ||| snd t
                   where n  = lengthS s
                         t  = scanS op e (contraer op s)
                         xs = expander op s (fst t)

contraer op s = tabulate g m
                where n   = lengthS s
                      m   = div (n + 1) 2
                      g i | i==(m-1)  = if (odd n) then (s!(n-1)) else elem
                          | otherwise = elem
                                where elem = op (s!(2*i)) (s!(2*i+1))
expander op s s' = tabulate f n
                where   n    = lengthS s
                        f i | even i    = elem
                            | otherwise = op elem (s!(i-1))
                              where elem = s'!(div i 2)
```

En Scan para arreglos, al igual que en reduce para arreglos; hacemos exáctamente lo mismo que en listas: contraer - aplicar recursivamente - expander.

Contraer y expander tienen como trabajo la suma de las respectivas operaciones que realizan, y como profundidad la máxima profundidad de ellas. (Igual que en listas pero restando el costo de recorrer la secuencia)

Reduce va a tener como trabajo el cálculo de todas las operaciones de todos los contaer y expandir que se realicen. En la especificación de costos esperados hay un n sumando, pero consideramos que la sumatoria es mayor (y la abarca) aún cuando la operación sea constante, pues deberemos hacer (más) de n operaciones.

$$W_{scan}(s\ e\ \oplus) \in \mathcal{O}(\sum_{x \oplus y \in \mathcal{O}_s(s\ e\ \oplus)} W(x \oplus y))$$

En cuanto a la profundidad, se realizarán log n expansiones y log n contracciones. En cada contracción y cada expansión, el costo será el máximo de todas las operaciones que se realicen en la llamada. Todos estos máximos podemos acotarlos por el máximo de toda operación realizada en el cálculo de scan. Luego tendremos 2*log n*(costo de la máxima operación), lo que nos dá la profundidad de scan deseada.

$$S_{scan}(s\ e\ \oplus) \in \mathcal{O}(lg\ n * \text{máx}_{x \oplus y \in \mathcal{O}_s(s\ e\ \oplus)} S(x \oplus y))$$