Programación Declarativa, 2021-1
Nota de clase 10: Evaluación Perezosa*

Karla Ramírez Pulido Manuel Soto Romero Luis Fernando Loyola Cruz

22 de noviembre de 2020
Facultad de Ciencias UNAM

Hasta ahora hemos usado Haskell en la definición de funciones y evaluación de expresiones sin prestar mucha atención al régimen de evaluación que ocupa. Como mencionamos en las notas introductorias a este lenguaje, Haskell hace uso de un régimen de evaluación perezosa. En esta nota mostraremos la definición de este concepto así como sus principales ventajas.

10.1. Estrategias de evaluación

Como hemos visto a lo largo de estas notas, el modelo básico de computación del estilo de Programación Funcional es la aplicación de funciones. Por ejemplo, supongamos que se tiene la siguiente función:.

```
inc :: Int → Int
inc n = n + 1
```

Esta función puede aplicarse al argumento `2 * 3` de la siguiente manera:

```
> inc (2 * 3)
  inc 6
  6 + 1
  7
```

Otra forma de evaluar la misma expresión es, no evaluar el argumento hasta llegar a la evaluación del cuerpo de la función:

```
> inc (2 * 3)
  inc (2 * 3)
  (2 * 3) + 1
  6 + 1
  7
```

*Adaptación de las *Notas de clase de Lenguajes de Programación* de Karla Ramírez, Manuel Soto, et. al.

Como vemos, sin importar la forma en que se van aplicando las funciones, el resultado no debe cambiar en un principio. Esta es una propiedad de los lenguajes funcionales puros que proviene del principio de transparencia referencial. De forma tal que:

Dos formas de evaluar la misma expresión siempre producirán el mismo resultado final.

Por supuesto, esta propiedad no aplica con lenguajes impuros como los imperativos. Por ejemplo, supongamos la siguiente evaluación imperativa.

```
n := 0
n + (n := 1)
```

Veamos las formas de evaluarla:

```
> n + (n := 1)
  0 + (n := 1)
  0 + 1
  1
```

```
> n + (n := 1)
  n + 1
  1 + 1
  2
```

En las evaluaciones anteriores podemos apreciar la impureza de este tipo de lenguajes. Veamos otro ejemplo. definimos ahora la función `mult`:

```
mult :: (Int,Int) → Int
mult v = fst v * snd v
```

¿De qué forma podemos evaluar `mult ((1+2),(2+3))`?

Forma 1

```
> mult ((1+2),(2+3))
  mult (3,(2+3))
  mult (3,5)
  fst (3,5) * snd (3,5)
  3 * 5
  15
```

Forma 2

```
> mult ((1+2),(2+3))
  fst ((1+2),(2+3)) * snd ((1+2),(2+3))
  (1+2) * (2+3)
  3 * (2+3)
  3 * 5
  15
```

La primera forma hace uso de una estrategia de evaluación llamada *evaluación glotona* o *ansiosa*. En este tipo de evaluación, los argumentos pasados a una función siempre deben ser reducidos a un *valor*.

La segunda forma es llamada *evaluación perezosa*. En este tipo de evaluación los argumentos de una función no son evaluados hasta que es estrictamente necesario. Se dice entonces que son pasados por *nombre*.

Existen otras formas de paso de parámetros, que se estudian a profundidad en los cursos de Lenguajes de Programación.

10.2. Beneficios de la evaluación perezosa

La mayoría de lenguajes de programación hacen uso de evaluación glotona, debido principalmente a que la evaluación perezosa consume más espacio en memoria al guardar las expresiones sin evaluar.

Sin embargo, trae varios beneficios consigo. En general, la evaluación perezosa se define como:

Definición 10.1. *(Evaluación Perezosa)* La evaluación perezosa es una estrategia de evaluación dónde los parámetros en la llamada a una función no son evaluados hasta que sean requeridos en el cuerpo de la misma.

Por ejemplo, dada la definición de factorial:

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

Podemos comprobar que el ejecutar `fact 100000` ocasiona lo que se conoce como *desborde de memoria*. Sin embargo, si definimos la siguiente expresión usando nuestra definición de factorial:

```
let x = fact 100000 in
   1
```

dado que la función factorial ni siquiera es usada, la evaluación perezosa ignora dicha asignación regresando un 1 como resultado.

Sin embargo, si usamos evaluación glotona, el 1 ni siquiera llega a evaluarse.

Esto es útil pues permite ignorar cálculos que no son usados en un programa. Otra aplicación útil es la definición de estructuras de datos infinitas.

10.3. Estructuras de datos infinitas

Gracias a la evaluación perezosa, es posible definir de cierta forma, estructuras de datos infinitas. En realidad, sólo podemos simularlo pues no tenemos como tal el concepto de infinidad presente en computación.

Por ejemplo, consideremos la siguiente definición que usa evaluación perezosa:

```
unos :: [Int]
unos = 1:unos
```

Esta función sin parámetros genera una lista conformada por un uno como cabeza y la llamada a la función recursiva como resto.

Esta definición, en pocas palabras genera una lista de unos. Es fácil notar que al evaluar la función, ésta no terminará la recursión pues no tiene definido un caso base. Sin embargo, con la evaluación perezosa podemos operar con esta definición como si se tratara de una lista infinita de unos.

Veamos la evaluación de la expresión `head unos` usando ambos regímenes de evaluación.

```
Evaluación glotona

> head unos
  head (1:unos)
  head (1:1:unos)
  ...
```

```
Evaluación perezosa

> head unos
  head (1:unos)
  1
```

De esta forma, podemos apoyarnos de funciones que operan de forma similar a `head` para obtener elementos de a poco. Por ejemplo,

```
> take 3 unos
  take 3 (1:unos)
  1:(take 2 unos)
  1:(take 2 (1:unos))
  1:1:(take 1 unos)
  1:1:(take 1 (1:unos))
  1:1:1:(take 0 unos)
  1:1:1:[]
  [1,1,1]
```

Además de este tipo de funciones sencillas, podemos definir funciones más complejas. Por ejemplo:

Una forma de encontrar los números primos en un determinado rango es mediante la conocida *Criba de Eratóstenes*. El algoritmo consiste en ir tomando los números del rango de uno en uno y eliminar todos los números que sean múltiplos de éste. El algoritmo terminará cuando no se pueda eliminar ningún número más. Por ejemplo, para encontrar los números primos del 2 al 20, se tiene la siguiente lista:

```
[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

El primer paso consiste en eliminar todos aquellos números que sean múltiplos de dos (excepto en el que estemos posicionados), con lo cual quedaría la siguiente lista:

```
[2,3,5,7,9,11,13,15,17,19]
```

Ahora, se pasa al siguiente número en la lista, en este caso el tres, y se repite el procedimiento:

```
[2,3,5,7,11,13,17,19]
```

Para el siguiente paso, se deben eliminar los múltiplos de cinco, sin embargo no queda ningún múltiplo de este número en la lista con lo cual, termina el algoritmo y se concluye que los números primos del 2 al 20 son:

```
[2,3,5,7,11,13,17,19]
```

De esta forma podemos generar una función `criba` que implemente el algoritmo descrito anteriormente, usando una lista por comprensión y a partir de esta definición, dar una función `primos` que permita construir infinitos números primos. La definición de estas funciones se muestra a continuación:

```
criba :: [Int] → [Int]
criba (p:xs) = p:(criba [x | x <- xs, mod x p /= 0])

primos :: [Int]
criba [2..]
```

De esta forma podemos obtener números primos con el apoyo de `take`.

```
> take 3 primos
  take 3 (criba [2..])
  take 3 (2:(criba [x | x <- [3..], mod x 2 /= 0]))
  2:(take 2 (3:(criba [x | x <- [5..], mod x 3 /= 0])))
  2:3:(take 1 (5:(criba [x | x <- [7..], mod x 5 /= 0])))
  2:3:5:(take 0 (7:(criba [x | x <- [11..], mod x 7 /= 0])))
  2:3:5:[]
  [2,3,5]
```

10.4. Evaluación Estrictra

Hemos dicho que HASKELL es un lenguaje de programación perezoso, sin embargo, la especificación del lenguaje, establece que en realidad es un lenguaje de programación *no estricto* que no es lo mismo que *perezoso*.

El término *no estricto* es un concepto semántico que especifica que la reducción (evaluación) de una expresión se realiza comenzando con el operador de mayor precedencia, por ejemplo, si tenemos la expresión `(a + (b * c))` entonces primero se reduce el `+` y luego la expresión `(b * c)`. Los lenguajes estrictos funcionan al revés, es decir, primero reducen los operandos del operador principal. Esto es importante para la semántica de un lenguaje, pues si se tiene una expresión que se evalúa a $\perp$ (es decir, un error o ciclo infinito) en un lenguaje estricto, éste se propagará dando como resultado $\perp$. Sin embargo, en un lenguaje *no estricto* algunas de las subexpresiones son eliminadas por reducciones externas, por lo que no se evalúan y entonces puede que nunca se encuentre $\perp$.

Por otro lado, el término *evaluación perezosa* se refiere al comportamiento operacional, es decir, la forma en como se ejecuta el código en una computadora real, en donde se indica que la evaluación de una expresión se va a realizar hasta que sea necesario el valor de ésta, además de evitar evaluaciones repetidas. La evaluación perezosa da lugar a una semántica no estricta, es por eso que los conceptos son tan similares, sin embargo la evaluación perezosa no es la única forma de implementar una semántica no estricta.

La evaluación perezosa puede ser una herramienta útil para mejorar el rendimiento de un programa, pero también puede ocasionar ciertos inconvenientes como un uso excesivo de memoria. Por ejemplo, veamos que sucede con la siguiente expresión.

```
foldl (+) 0 [1..1000000]
```

Recordemos que la función `foldl` es una función de plegado definida usando la técnica de recursión de cola, por lo que la evaluación de la expresión anterior luce de la siguiente manera:

```
foldl (+) 0 [1..1000000] = foldl (+) (0+1) [2..1000000]
                         = foldl (+) ((0+1) +2) [3..1000000]
                         = foldl (+) (((0+1) +2) +3) [4..1000000]
                         = ...
                         = foldl (+) ((((0+1) +2) +3) +...+1000000) []
                         = (((((0+1) +2) +3) +...) +1000000)
                         = 500000500000
```

¿Cuál es el problema con esta expresión? Debido a la evaluación perezosa, en cada llamada recursiva vamos guardando una expresión cuyo tamaño es linealmente proporcional a la cantidad de elementos de la lista de entrada, esta expresión se debe guardar en algún lugar de la memoria en caso de que se evalúe más tarde. Almacenar y evaluar expresiones tan grandes es costoso e innecesario si de todos modos se van a evaluar.

Haskell usa evaluación perezosa de forma predeterminada, pero también proporciona una mecanismo para evaluar de forma estricta los argumentos de una función. El operador `$!` es la versión estricta del operador de aplicación. Una expresión de la forma `f $! x` se comporta de la misma manera que la aplicación `f x`, excepto la evaluación de `x` se realiza antes de aplicar `f`.

Por ejemplo, sea `square` la función que calcula el cuadrado de un número definida de la siguiente manera.

```
square :: Int → Int
square n = n * n
```

La evaluación de la expresión `square (1+2)` se ve de la siguiente manera:

```
square $! (1+2)
-- Evaluamos (1+2)
= square $! 3
-- Aplicamos square
= square 3
= 3 * 3
= 9
```

Utilizando el operador de aplicación estricta podemos dar una versión estricta de `foldl` para hacerla más eficiente.

```
foldl' :: (b -> a -> a) -> a -> [a] -> b
foldl' _ e [] = e
foldl' f e (x:xs) = (foldl' $! (f e x)) xs
```

Con esta nueva definición la evaluación de la expresión `foldl' (+) 0 [1..1000000]` se ve así:

```
foldl' (+) 0 [1..1000000] = foldl' (+) (0+1) [2..1000000]
                          = foldl' (+) 1 [2..1000000]
                          = foldl' (+) 1 [2..1000000]
                          = foldl' (+) (1+2) [3..1000000]
                          = foldl' (+) 3 [3..1000000]
                          = ...
                          = foldl' (+) 499999500000 [1000000]
                          = foldl' (+) (499999500000 + 1000000) []
                          = foldl' (+) (499999500000 + 1000000) []
                          = foldl' (+) 500000500000 [] = 500000500000
```

A pesar de que la versión estricta de las funciones puede ser más eficiente en alguno casos, no olvidemos el comportamiento de una función estricta frente a $\perp$.

Referencias

[1] Graham Hutton, *Programming in Haskell*, Tercera edición, Cambridge University Press, 2008

[2] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation,* Segunda Edición, Brown University, 2007.

[3] Haskell Wiki. "Lazy vs. non-strict" Web. Consultada el 22 de noviembre de 2020. `https://wiki.haskell.org/Lazy_vs._non-strict`