Lenguajes de Programación, 2021-2
Nota de clase 11: *Memoización*

Karla Ramírez Pulido Manuel Soto Romero Javier Enríquez Mendoza

14 de enero de 2021
Facultad de Ciencias UNAM

11.1. Introducción

En la Nota de Clase 10, se expusieron distintos tipos de paso de parámetros y en particular se estudiaron dos técnicas de paso de parámetros relacionadas con la evaluación perezosa: (1) paso por nombre y (2) paso por necesidad. Éste último evita la generación de cálculos repetitivos, una vez que una función es evaluada con un parámetro, este resultado es guardado y evita calcularse cada que es llamado (tal y como lo hace el paso de parámetros por nombre).

En esta nota se revisan algunos ejemplos de funciones que generan llamadas repetitivas y se presenta a la *Memoización* como una técnica de optimización que da solución a este problema.

11.2. Llamadas repetitivas

En el Código 1 se muestra la definición de la función Fibonacci, esta función es un ejemplo de la geneneración de llamadas repetitivas debido a que cada llamada, genera cálculos que otras llamadas ya habían resuelto y se vuelve por lo tanto ineficiente.

```
1 ;; fibo: number → number
2 (define (fibo n)
3    (if (< n 2)
4        1
5        (+ (fibo (sub1 n)) (fibo (- n 2)))))
```

Código 1: Función fibo

En la Figura 1, se muestra el árbol de llamadas para la función Fibonacci con llamada `(fibo 5)`. Cada rama muestra las llamadas pendientes del caso recursivo hasta llegar a un caso base.

Puede observarse que varias ramas del árbol incluyen exactamente la misma llamada a función (mismo parámetro) y por ende se crean registros de activación idénticos durante misma ejecución. Esto ocasiona, entre otras cosas, que el tiempo de ejecución de la función Fibonacci incremente conforme el tamaño de su entrada lo hace.

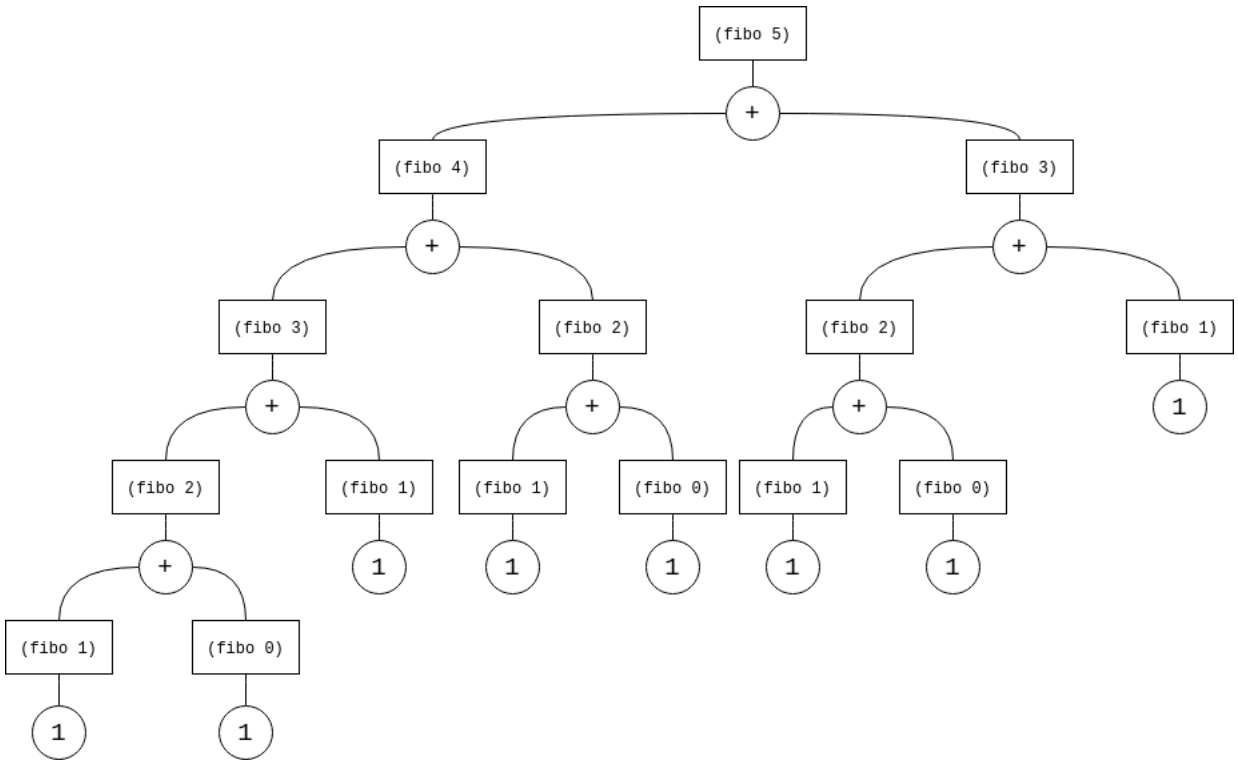

Figura 1: Árbol de llamadas de Fibonacci

Este tipo de problema se asocia con la Evaluación Perezosa ya que describe de qué tan perezoso o no debería ser un lenguaje de programación con respecto a la evaluación de funciones.

11.3. Método de optimización

Una forma de optimizar una función para que no genere llamadas repetidas es mediante la técnica de *Memoización*[1] que consiste en cargar una tabla con los resultados ya conocidos para cada una de las posibles entradas. De esta forma al evaluar una función, se busca el resultado para el parámetro de entrada y en caso de encontrarlo, simplemente se devuelve; en caso contrario se calcula y se ingresa el registro correspondiente en la tabla. Esta forma de programar se conoce como *Programación Dinámica*.

Podría usarse cualquier estructura de datos para almacenar los registros, sin embargo, lo usual es hacer uso de Tablas de Dispersión pues la búsqueda de parámetros (llave) resulta ser bastante eficiente.

Ejemplo 11.1. El Código 2 muestra la modificación a la función Fibonacci *memoizada*. Se agrega un parámetro nuevo que es la tabla de dispersión correspondiente. Para manipular tablas de dispersión en Racket se usan las siguientes funciones:

- **`make-hash`** Crea una nueva tabla de dispersión.
- **`hash-ref`** Obtiene el valor asociado a una llave. En caso de no encontrarlo regresa el valor por defecto especificado.

[1] Del inglés *memoization*.

- **`hash-set!`** Agrega un nuevo registro a la tabla.

Al igual que al usar la técnica de recursión de cola, debe modificarse la llamada principal para que haga uso de la versión optimizada. En este caso se inicializa la tabla con el valor ya conocido de los casos base.

```
1  ;; fibo: number → number
2  (define (fibo n)
3     (fibo-memo n (make-hash (list (cons 0 1) (cons 1 1)))))
4
5  ;; fibo-memo: number hash-table → number
6  (define (fibo-memo n tbl)
7     (let ([busqueda (hash-ref! tbl n 'vacia)])
8        (cond
9           [(equal? busqueda 'vacia)
10             (define nuevo
11                (+ (fibo-memo (- n 1) tbl) (fibo-memo (- n 2) tbl)))
12             (hash-set! tbl n nuevo)
13             nuevo
14          [else busqueda])))
```

Código 2: Memoización de Fibonacci

Del Código 2:

- Las líneas 2 a 3 muestran la modificación de la función original.
- La línea 7 muestra la búsqueda del valor recibido, en caso de no encontrarlo (línea 9), las líneas 10 a 13, calculan el valor para el parámetro, lo asignan a la tabla y lo devuelven. En caso de encontrarlo (línea 14) simplemente se devuelve el valor. □

Ejemplo 11.2. Supongamos el siguiente problema:

Un niño quiere subir saltando una escalera. Con cada salto que da, puede subir 1, 2 o 3 escalones. Por ejemplo, si la escalera tiene tres escalones, la puede subir de cuatro formas distintas: 1, 1, 1; 1, 2; 2, 1 o 3.

La siguiente función **`(numero-formas e)`** recibe número de escalones de una escalera e indica el número de formas que puede subir saltando el niño.

```
1  ;; numero-formas: number → number
2  (define (numero-formas e)
3     (cond
4        [(= e 1) 1]
5        [(= e 2) 2]
6        [(= e 3) 4]
7        [else (+ (numero-formas (- e 1))
8                 (numero-formas (- e 2))
9                 (numero-formas (- e 3)))]))
```

Código 3: Función `numero-formas`

Versión memoizada

```
;; numero-formas: number → number
(define (numero-formas e)
   (formas-memo e (make-hash (list (cons 1 1) (cons 2 2) (cons 3 5)))))

;; formas-memo: number hash-table → number
(define (formas-memo e tbl)
   (let ([busqueda (hash-ref! tbl e 'vacia)])
      (cond
         [(equal? busqueda 'vacia)
            (define nuevo
               (+ (formas-memo (- e 1)
                  (formas-memo (- e 2))
                  (formas-memo (- e 3))))
            (hash-set! tbl e nuevo)
            nuevo]
         [else busqueda])))
```

Código 4: Función `numero-formas` memoizada

□

Ejemplo 11.3. *Memoización* de **map** para listas con elementos repetidos.

Versión original

```
;;map: procedure (listof any) → (listof any)
(define (map f ls)
   (if (empty? ls)
       empty
       (cons (f (car ls)) (map f (cdr ls)))))
```

Código 5: Función `map`

Versión memoizada

```
;;map: procedure (listof any) → (listof any)
(define (map f ls)
   (map-memo f ls (make-hash empty)))

;;map-memo: procedure (listof any) hash-table → (listof any)
(define (map-memo f ls tbl)
   (cond
      [(empty? ls) empty]
      [else
         (define a (car ls))
         (define busqueda (hash-ref tbl a 'vacia))
         (if (equal? busqueda 'vacia)
             (begin
               (define nuevo (f a))
```

```
15                (hash-set! tbl a nuevo)
16                (cons nuevo (map f (cdr ls) tbl))
17              (cons busqueda (map f (cdr ls) tbl)))]))
```

Código 6: Función `map` memoizada

□

En los ejemplos anteriores, ambas funciones fueron optimizadas mediante *memoización* y de hecho cualquier función puede ser *memoizada*, con el riesgo de que no mejore su desempeño.

Ejemplo 11.4. Memoización de la función `factorial`.

Versión original

```
1 ;; factorial: number → number
2 (define (factorial n)
3    (if (zero? n)
4        1
5        (* n (factorial (sub 1 n)))))
```

Código 7: Función `factorial`

Versión memoizada

```
1  ;; factorial: number → number
2  (define (factorial n)
3     (fact-memo n (make-hash (list (cons 0 1)))))
4
5  ;; factorial-memo: number hash-table → number
6  (define (factorial-memo n tbl)
7     (let ([busqueda (hash-ref! tbl n 'vacia)])
8        (cond
9           [(equal? busqueda 'vacia)
10             (define nuevo (factorial-memo (sub1 n) tbl))
11             (hash-set! tbl n nuevo)
12             nuevo]
13          [else busqueda])))
```

Código 8: Función `factorial` memoizada

□

La función factorial del ejemplo 11.4 fue memoizada, sin embargo, al tener complejidad lineal ($O(n)$), no aprovecha el reuso de resultados previos ya que sólo utiliza los argumentos una única vez.

Otra forma de ver a la memoización es como una especie de *aprendizaje* que van adquiriendo las funciones conforme las vamos ejecutando con ciertos parámetros. Sin embargo, para este caso debemos hacer que la *memoria* de la función se encuentre de forma externa a ella lo cual puede ocasionar efectos secundarios.

Ejemplo 11.5. El predicado es-raro? decide si un número es raro. Es decir, si cumple con ser de la forma: $n = \underbrace{n_o^l + n_1^l + ...}_{l \text{ elementos}}$ con l el número de dígitos del número. Por lo cual, es necesario, primero calcular la longitud del número, después descomponerlo en dígitos y elevar cada uno de éstos a la longitud obtenida previamente.

Versión original

```
;; es-raro?: number → boolean
(define (es-raro? n)
    (let* ([long (longitud n)]
           [suma (suma n long)])
        (equal? n suma)))

;; longitud: number → number
(define (longitud n)
    (if (< n 10)
        1
        (+ 1 (longitud (quotient n 10)))))

;; suma: number number → number
(define (suma n l)
    (if (< n 10)
        (expt n l)
        (+ (expt (modulo n 10) l) (suma (quotient n 10) l))))
```

Código 9: Predicado es-raro?

Versión memoizada

```
;; Memoria de la función es-raro?
(define tbl (make-hash empty))

;; es-raro?: number → boolean
(define (es-raro? n)
    (let ([busqueda (hash-ref! tbl n 'vacia)])
        (cond
            [(equal? busqueda 'vacia)
                (define nuevo
                    (let* ([long (longitud n)]
                           [suma (suma n long)])
                        (equal? n suma)))
                (hash-set! tbl n nuevo)
                nuevo]
            [else busqueda])))
```

Código 10: Predicado es-raro memoizado

De esta forma, aunque la función no carga de forma explícita el resultado de cada llamada, se tiene almacenada como una variable global, con lo cual la memoria de la función no depende de la ejecución y evita hacer cálculos llamada tras llamada pues con el uso que se le vaya dando a la función, aprenderá de cierta manera, el resultado de cada posible entrada.

□

11.4. Implementación

Para hacer que nuestro lenguaje aplique este tipo de paso de parámetros es necesario que al evaluar una aplicación de función se apliquen de manera similar, los pasos de los ejemplos anteriores:

1. Los resultados de cada llamada a función con su respectivo parámetro deben guardarse en una tabla de dispersión (o alguna estructura equivalente). Para que cada función cuente con esta información, puede añadirse un parámetro a las cerraduras de función, como sigue:

   ```
   [closureV (param symbol?) (body BRCFAE?) (rest-env Env?) (tbl hash?)]
   ```

2. Al evaluar una aplicación de función, se debe buscar en la tabla de dispersión asociada a la cerradura, si la función fue aplicada con el mismo parámetro con anterioridad, en cuyo caso debe regresarse dicho valor.

3. Si el valor no es encontrado en la tabla, debe realizarse la interpretación correspondiente y modificar la cerradura para que su tabla de dispersión, almacene el resultado encontrado. Finalmente se debe regresar este nuevo valor calculado.

Referencias

[1] Dexter Kozen, *Data Structures and Functional Programming,* Notas de Clase, Revisión 2011. [En línea: `https://www.cs.cornell.edu/courses/cs3110/2011sp/`]

[2] Lourdes del C. González Huesca, *Programación dinámica puramente funcional: el caso de la memoización*. Tesis de Maestría, UNAM, 2010.