Lenguajes de Programación, 2022-1
Nota de clase 05: Evaluación Perezosa

Karla Ramírez Pulido Manuel Soto Romero

5 de noviembre de 2021
Facultad de Ciencias UNAM

Hasta ahora la forma en que hemos evaluado expresiones es mediante la función **`interp`** del intérprete correspondiente. Sin embargo (para cada lenguaje) la forma de evaluación depende de lo que decida el diseñador del lenguaje. En esta nota revisaremos el concepto de régimen de evaluación y discutiremos las principales diferencias entre éstos, haciendo especial énfasis en la definición y usos de la evaluación perezosa.

5.1. Estrategias de evaluación

Como hemos visto a lo largo de estas notas, el modelo básico de computación del paradigma funcional es la aplicación de funciones. Por ejemplo, supongamos que se tiene la función del Código 1, escrita en Racket.

```
1 ;; inc: number → number
2 (define (inc n)
3     (+ n 1))
```

Código 1: Función `inc`

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 inmediatamente el argumento hasta llegar a la evaluación del cuerpo de la función:

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

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 presente en 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 Racket o los lenguajes 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** en el Código 2.

```
1 ;; mult: (pairof number) → number
2 (define (mult v)
3    (* (car v) (cdr v)))
```

Código 2: Función mult

¿De qué forma podemos evaluar **(mult (cons (+ 1 2) (+ 2 3)))**?

Forma 1

```
> (mult (cons (+ 1 2) (+ 2 3)))
  (mult (cons 3 (+ 2 3)))
  (mult (cons 3 5))
  (* (car (cons 3 5)) (cdr (cons 3 5)))
  (* 3 (cdr (cons 3 5)))
  (* 3 5)
  15
```

Forma 2

```
> (mult (cons (+ 1 2) (+ 2 3)))
  (* (car (cons (+ 1 2) (+ 2 3)))
     (cdr (cons (+ 1 2) (+ 2 3))))
  (* (+ 1 2)
     (cdr (cons (+ 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 estudiaremos más adelante en otra nota.

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

Por ejemplo, dada la definición de factorial del Código 3.

```
1 ;; fact: number → number
2 (define (fact n)
3    (if (zero? n)
4        1
5        (* n (fact (sub1 n)))))
```

Código 3: Función `fact`

Podemos comprobar que el ejecutar **`(fact 100000)`** ocasiona lo que se conoce como *desborde de memoria*. Más tarde analizaremos estos casos con más detalle. Sin embargo, si definimos la siguiente expresión usando nuestra definición de factorial:

```
(let ([x (fact 100000)])
   1)
```

dado que la función factorial ni siquiera es usada, la evaluación perezosa ignorará dicha asignación (como es perezosa la evaluación de la asignación se realiza, pero no así la aplicación de función) 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, como veremos más adelante en esta nota.

5.3. Puntos estrictos

Al usar evaluación perezosa, debemos tener presente en qué momento es necesario evaluar expresiones, esto se conoce como *punto estricto*.

Definición 2. *(Punto estricto) A los puntos donde la implementación de un lenguaje de programación perezoso, fuerza la reducción de una expresión a un valor (si existe) se les llama puntos estrictos.*

Por ejemplo, supongamos el lenguaje **`CFWAE`** para condicionales.

```
<expr> ::= <id>
         | <num>
         | {+ <expr> <expr>}
         | {- <expr> <expr>}
         | {if0 <expr> <expr> <expr>}
         | {with {<id> <expr>} <expr>}
         | {fun {<id>} <expr>}
         | {<expr> <expr>}
```

El nuevo constructor, `if0` evalúa la primera expresión, si ésta se reduce a cero, devuelve el resultado de la segunda expresión (*then*), en caso contrario, evalúa la tercera expresión (*else*).

Si definiéramos un intérprete para este lenguaje usando evaluación perezosa, ¿cómo se evaluaría la siguiente expresión?

```
{if0 {- 10 10}
     1
     2}
```

La expresión no puede evaluarse hasta conocer el valor de la condición, por lo tanto, aquí se presenta un punto estricto: la condición del `if0`.

Otra expresión de interés son las operaciones aritméticas, por ejemplo:

```
{+ {- 2 3} {+ 3 4}}
```

La suma no puede ser efectuada hasta que todos sus argumentos sean reducidos (en este caso 2). Por lo tanto en los operandos de las de las operaciones aritméticas `+` y `-` encontramos otro punto estricto.

Finalmente, analicemos la siguiente expresión:

```
{with {f {fun {x} {+ x 2}}}
   {f {+ 2 3}}}
```

¿Qué necesito conocer para evaluar la expresión?

Erróneamente, se podría llegar a pensar que el argumento de la aplicación de función es un punto estricto, sin embargo, ésta es justo la definición de evaluación perezosa. El argumento debe ser evaluado hasta ser necesitado y en este caso ni siquiera sabemos dónde será usado, por lo tanto no es un punto estricto.

De esta forma, podemos observar que el punto estricto es el lado izquierdo en la aplicación de función. Si no evaluamos `f`, no podemos conocer su definición y por lo tanto no podemos sustituir el argumento.

En resumen, en **`CFWAE`** tenemos tres puntos estrictos:

- Condicional del `if0`.
- Operandos de las operaciones aritméticas.

- Expresión izquierda en una aplicación de función.

Veamos un ejemplo de evaluación, usaremos alcance estático por comodidad.

```
{with {a {+ 2 3}}
   {with {b {+ 4 5}}
      {with {c {+ 6 7}}
         {if0 a b c}}}}
```

1. Ambiente inicial

Se construye el ambiente con las sustituciones (asignaciones) encontradas. Observar que al usar evaluación perezosa los argumentos no son evaluados.

c	{+ 6 7}
b	{+ 4 5}
a	{+ 2 3}

2. Se evalúa el cuerpo del with más anidado, es decir, {if0 a b c}

El cuerpo del `with` *es una expresión condicional* `{if0 a b c}`*, por lo tanto tenemos un puntos estricto. Para continuar con la evaluación, debemos reducir primero la condición, dado que es un identificador lo buscamos en el ambiente.*

Al ser la primera búsqueda, se inicia desde la región donde fue definido el condicional, es decir, el tope de la pila.

	c	{+ 6 7}
	b	{+ 4 5}
⇒	a	{+ 2 3}

3. Se obtiene una suma, {+ 2 3}

Tenemos otro punto estricto. Para continuar con la ejecución del programa debemos evaluar el lado izquierdo y derecho de la operación, dado que son números, no hay mucho que hacer, por lo tanto obtenemos 5.

4. Evaluación de la condición

Dado que $5 \neq 0$*, evaluamos la rama* `else`*. Como seguimos tratando de evaluar el condicional, la búsqueda se mantiene en el tope de la pila, por lo tanto se busca y encuentra* `c`*.*

⇒	c	{+ 6 7}
	b	{+ 4 5}
	a	{+ 2 3}

5. Se obtiene una suma, {+ 6 7} y se obtiene el resultado final

Tenemos otro punto estricto, para continuar con la evaluación debemos evaluar el lado izquierdo y derecho de la operación. Dado que son números no hay mucho que hacer, por lo tanto obtenemos 13 como resultado final.

Se deja como ejercicio al lector la evaluación de

```
{with {a {+ 2 2}}
   {with {b {+ a a}}
      {with {a {+ 3 3}}
         b}}}
```

Usando:

1. Alcance estático y evaluación glotona
2. Alcance estático y evaluación perezosa
3. Alcance dinámico y evaluación glotona
4. Alcance dinámico y evaluación perezosa

5.4. 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 (no finito) presente en computación.

Por ejemplo, consideremos la definición del Código 4 escrita en el lenguaje de programación Haskell que usa evaluación perezosa.

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

Código 4: Función `unos`

Esta función (sin parámetros) genera una lista conformada por un uno como cabeza y la llamada a la función recursiva como cola. El operador `:` es equivalente a **`cons`** en Racket.

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 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", es decir, sólo los argumentos que en el momento se van necesitando (evaluaciones parciales). 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]
```

Podemos generar una lista de números primos infinitos, con el apoyo de listas por comprensión. Una lista por comprensión es una forma de expresar listas como se hace en Teoría de Conjuntos. Por ejemplo, puedo generar el conjunto de los números pares del 2 al 10 de la siguiente manera:

$$P = \{2x : x \in \{1, 2, 3, 4, 5\}\}$$

Esto puede expresarse en HASKELL de la siguiente forma:

```
[2*x | x <- [1,2,3,4,5]]
```

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 en el Código 5.

```
1 criba :: [Int] -> [Int]
2 criba (p:xs) = p:(criba [x | x <- xs, mod x p /= 0])
3
4 primos :: [Int]
5 primos = criba [2..]
```

Código 5: Función Generación de números primos

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]
```

5.5. Implementación de listas infinitas

En lenguajes con evaluación glotona, como Racket, es posible simular listas infinitas mediante el uso de *thunks*. Un *thunk* es, en pocas palabras, una función que no recibe parámetros. De esta forma, se puede postergar la evaluación de ciertas expresiones encapsulándolas en *thunks* que no serán evaluadas hasta que la función no sea aplicada.

5.5.1. Listas pseudo-infinitas

Veamos la implementación de listas pseudo-infinitas. Para ello, modificaremos ligeramente la definición usual de listas. Este tipo de estructura es usualmente llamado *stream*, por lo tanto añadiremos una s al inicio del nombre de los constructores.

```
1 ;; Predicado para definir estructuras genéricas
2 ;; any?: any → boolean
3 (define (any? a) #t)
4
5 ;; Definición de listas pseudo-infinitas.
6 (define-type Stream
7    [sempty]
8    [scons (head any?) (tail procedure?)])
```

Código 6: Definición de streams

De esta forma, podemos definir la función `unos`, definida anteriormente en HASKELL.

```
1  ;; Función que genera una lista infinita de unos.
2  ;; unos: Stream
3  (define (unos)
4     (scons 1 (thunk (unos))))
```

Código 7: Lista infinita de unos

Observación 1. La función `thunk` de RACKET es, como dijimos anteriormente, una función sin parámetros. En este caso es equivalente a `(λ () (unos))`.

Para acceder a los elementos del *stream*, es necesario definir algunas funciones como **`tail`** y **`take`**. Para generar cada elemento de la lista, es necesario aplicar el *thunk* que genere el siguiente elemento de la lista. El Código 8 muestra la definición de estas funciones.

```
1  ;; Función que obtiene el resto de un stream.
2  ;; stail: Stream → Stream
3  (define (stail s)
4     ((scons-tail s)))
5
6  ;; Función que obtiene los primeros n elementos de un stream en forma
7  ;; de lista.
8  ;; stake: number Stream → list
9  (define (stake n l)
10    (if (zero? n)
11        empty
12        (cons (scons-head l) (stake (sub1 n) (stail l)))))
```

Código 8: Funciones auxiliares para streams

Por ejemplo, para obtener, 10 elementos de la lista de unos, usamos la función **`stake`** como sigue:

```
> (stake 10 (unos))
'(1 1 1 1 1 1 1 1 1 1)
```

Se deja como ejercicio al lector la definición de la lista **`primos`** usando esta representación de listas infinitas.

5.6. Ejercicios

5.6.1. Sintaxis de Condicionales

1. Dada la sintaxis concreta del lenguaje CFWAE y dada su sintaxis abstracta:

   - Muestra una derivación de las siguientes expresiones o justifica por qué éstas no son parte de la gramática.
   - En caso de ser posible obtén su correspondiente sintaxis abstracta.

   (a) `{if0 {* a 2} 17 29}`
   (b) `{fun {n} {if0 n {+ n 2} {- n 2}}}`
   (c) `{with {f {fun {n} {if0 n n {+ {f {- n 1}}}}} {f 10}}`

5.6.2. Puntos Estrictos

1. Tomando como base tu lenguaje de programación favorito, lista los posibles puntos estrictos del mismo. Considera puntos estrictos nuevos, no únicamente las condiciones en estructuras de control, por ejemplo.

2. Algunos lenguajes de programación no hacen uso de puntos estrictos para continuar con la ejecución en lenguajes perezosos (por ejemplo la versión perezosa de RACKET). Este tipo de lenguajes otorgan primitivas al programador para que pueda realizar la evaluación cuando lo requiera. ¿Cuáles son las ventajas y desventajas de esta forma de evaluación?

5.6.3. Alcance y Evaluación

1. Evalúa las siguientes expresiones usando cada una de las siguientes combinaciones de alcance y evaluación:

   - Alcance dinámico y evaluación glotona.
   - Alcance dinámico y evaluación perezosa.
   - Alcance estático y evaluación glotona.
   - Alcance estático y evaluación perezosa.

   Es necesario mostrar el ambiente de evaluación usando representación de pila en cada caso.

   (a)
```
{with {x {+ 10 10}}
   {with {y {- x 5}}
      {with {f {fun {z} {+ x {+ y z}}}}
         {with {x {+ 5 5}}
            {with {g {fun {w} {+ x {+ y w}}}}
               {with {x 2}
                  {with {y 1}
                     {with {z - 10}
                        {with {w - 5}
                           {f 15}}}}}}}}}}
```

   (b)
```
{with {x 1}
   {with {x {+ 1 1}}
      {with {foo {fun {y} {+ y {+ x {+ x y}}}}}
         {with {x {+ 1 2}}
            {with {x {+ 2 2}}
               {foo 5}}}}}}
```

   (c)
```
{with {x 3}
   {with {y 2}
      {with {x 1}
         {with {foo1 {fun {y} {+ y {- y x}}}}
            {with {x 3}
               {with {y 0}
                  {with {x -1}
                     {with {foo2 {fun {x} {+ x {- x y}}}}
```

```
{with {x - 3}
   {with {y 0}
      {with {x 2}
         {foo1 {+ 2 2}}}}}}}}}}}}}
```

5.6.4. Listas Infinitas

1. Usando HASKELL o RACKET define una lista infinita que nos permita obtener la tabla de multiplicar del 9. Después obtén la suma de los primeros 1000 múltiplos de 9.

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] Dexter Kozen, *Data Structures and Functional Programming Lecture Notes*, Cornell University, Revisión 2011.