Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

•••

Estrategias de evaluación: perezosa y glotona
Parte II

Karla Ramírez Pulido

Estrategias de evaluación

- **Perezosa** (del inglés *lazy evaluation*)

  Haskell

- **Glotona** (del inglés *eager evaluation*)

  Scheme, C, C++, Racket, Java, Python, Ruby,

  Julia, LISP, Algol, Pascal, FORTRAN, ...

Nuestro intérprete hasta el momento presenta un régimen de **evaluación glotón.**

Aplicación de función

(f x)

Si “x” es una expresión, entonces ésta se evalúa y pasamos el resultado de su evaluación a “f”.

```
[app (fun-expr arg-expr)
  (local ([ define fun-val (interp fun-expr env)]
          [ define arg-val (interp arg-expr env)] )
    … )]
```

Supongamos:
f=(lambda (x) (+ x x))

Si hacemos la llamada a “f” con el argumento (+ 2 2)
i.e. (f x)

Entonces f = (lambda (x) (+ x x))
x = (+ 2 2)

```
[app (fun-expr arg-expr)
  (local ([ define fun-val (interp fun-expr env)]
          [ define arg-val (interp arg-expr env)] )
    … )]
```

___

((lambda (x) (+ x x))
(+ 2 2))

La expresión anterior es de la forma: (f x)
i.e. es una APP de FUNCIÓN

```
[app (fun-expr arg-expr)
  (local ([ define fun-val (interp fun-expr env)]
          [ define arg-val (interp arg-expr env)] )
    … )]
```

=>

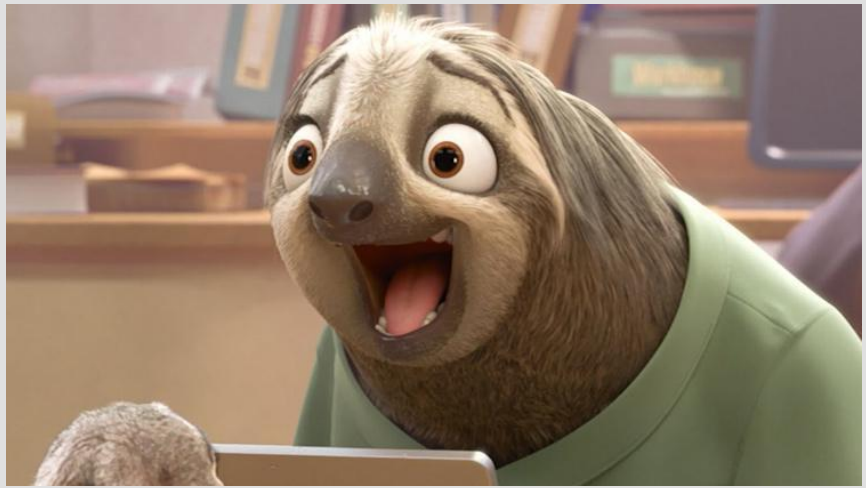
= ((lambda (x) (+ x x)) (+ 2 2))

= ((closureV x (+ x x) cl-env) 4)

fun-val= (closureV x (+ x x) cl-env)

arg-val= 4

Nos falta implementar en el intérprete: **evaluación perezosa**



Aplicación de función

(f x)

Si “x” es una expresión, entonces ésta se evalúa y pasamos el resultado de su evaluación a “f”.

```
[app (fun-expr arg-expr)
  (local ([define fun-val (strict (interp fun-expr env))]
          [define arg-val  (exprV arg-expr env)]
          … )]
```

Punto estricto

Concepto aplicado únicamente al régimen de evaluación perezoso

Aplicación de función:

(función argumento)

(f x)

¿Qué elemento de la aplicación se puede esperar a ser evaluado y cuál no?

R:= El argumento puede esperar

pero el paso del parámetro no.

———

Otro punto estricto

Condicional de la categoría sintáctica IF

IF

Estructura del if es:

if (cond-expr then-expr else-expr)

... body-if ...

cond-expr es un PUNTO ESTRICTO

then-expr y else-expr pueden esperar

Función: strict

```
(define (strict e )
  (type-case CFAL-Value e
    [exprV (expr env)
      (local  ([define the-value (strict (interp expr env))])
                (begin (printf ”Forcing exprV to ~a “ the-value) the-value))]
              [else e]))
```

Introducir algunos constructores:

<CFAE/L> ::= <num>

| {+ <CFAE/L> <CFAE/L>}

| <id>

| {fun {<id>} <CFAE/L>}

| {<CFAE/L> <CFAE/L>

Extendemos el lenguaje con los siguientes valores:

```
(define-type CFAE/L-Value
    [numV (n number?)]
    [closureV (param symbol?) (body CFAE/L?) (env Env?)]
    [exprV (expr CFAE/L?) (env Env?)])
```

Más ejemplos usando evaluación perezosa

Ejemplo de uso no.1

Supongamos que declaramos al inicio de un programa:

$$x = \{2 + \{2 + \{ 2 + \{ 2 + \{ 2 + 3 \}\}\}\}\}$$

Posteriormente en el mismo programa usamos esa x **al menos** unas 10 veces

{ ...

$$\{ x * \{ x * \{ x * \{ x * \{ x * \{ x * \{ x * \{ x * \{ x * x \}\}\}\}\}\}\}\}\}$$

... }

Usando régimen de evaluación perezoso ¿cuántas veces se realizará la evaluación de la misma x?

Ejemplos de uso

Supongamos que declaramos al inicio de un programa:

x = {2 + {2 + { 2 + { 2 + { 2 + 3 }}}}}

Posteriormente en el mismo programa usamos esa x **al menos** unas 10 veces

{ ...

{ x * { x * { x * { x * { x * { x * { x * { x * { x * x }}}}}}}}}

... }

Usando régimen de evaluación glotón ¿cuántas veces se realizará la evaluación de la misma x?

Ejemplo de uso no.2

Supongamos que declaramos una variable en un programa como:

x = { 2 + { 2 + { 2 + { 2 + { 2 + **{2 / 0}** }}}}}

Escenario A:

Evaluación perezosa, y

En el resto del programa NUNCA usamos “x”

Alumn@s: ¿Se evalúa la expresión asignada a “x”?

Ejemplo de uso no.2

Supongamos que declaramos una variable en un programa como:

x = { 2 + { 2 + { 2 + { 2 + { 2 + **{2 / 0}** }}}}}

Escenario B:

Evaluación glotona, y

En el resto del programa NUNCA usamos x

Alumn@s: ¿Se evalúa la expresión asignada a “x”?

¿Qué pasa cuando queremos generar LISTAS INFINITAS?

Programa en SHELL

Abrir sus respectivos intérpretes de comandos ¿están en algún Unix, cierto?

Ejemplo simple de generación infinita de streams: programa **yes el cual genera un stream infinito de y's**

Por cierto, usaremos C-z o C-c

> yes

¿Qué pasa?

Programa: YES

Ven algo así como:

> y

y

y

y

y

y

OK, OK, suficiente.

C-z ó C-C

¿Ya está todo bien ahora?

¿Qué pasó?
Sí sí, generamos una lista infinita de y’s

¿Qué hacer para ir generando estas listas pero poco a poco?

Programa: YES

> yes | head -5

y

y

y

y

y

Los programas en Unix pueden consumir streams infinitos y, se pueden componer con “|”.

La aplicación de Unix: **yes** llama **head** con el argumento de **5** (sintaxis “-5”) (por omisión es 10)

Otro ejemplo de streams infinitos

La aplicación **wc** que cuenta el número de **caracteres (-c)**, **palabras (-w) y líneas (-l)** en una entrada de streams.

c = character
w = word
l = lines

```
>   yes | head -5 | wc -l
```

```
5
```

Evaluación perezosa

Es una estrategia de evaluación la cual retrasa la evaluación de una expresión hasta que su valor es requerido.

Consecuencias:

- Los elementos son generados bajo demanda, así algunas estructuras no son generadas hasta que se requieren.
- Las expresiones pueden expresar generación de estructuras (listas) infinitas.
- Diseño del lenguaje.

Pros vs Cons

Evaluación Glotona

En promedio:

- Mejor uso de espacio
- Mejor desempeño (tiempo)

Evaluación Perezosa

- **En promedio:**
- Peor uso de espacio
- Peor desempeño (tiempo)
-

¿En qué casos sí resulta ser más eficiente la Evaluación Perezosa?

x = { + 1 { + 2 { + 3 { + 4 { + 5 { + 6 { + 7 ....}}}}}}}

Nunca se use esa variable en el programa

¡Perfecto!

Nunca se necesitó la x

Nadie hizo nada :)

Más eficiente en desempeño.

¿Qué hubiera pasado en un lenguaje con evaluación glotona?

Sí se hicieron TODAS las evaluaciones de las expresiones para que x tuviera un valor.

¿Más eficiente en manejo de tiempo?

A. SI **B. NO**

Pros vs Cons

Evaluación Glotona

En promedio:

- Mejor uso de espacio
- Mejor desempeño (tiempo)

En el peor de los casos:

Para algunos programas resulta ser **menos eficiente** cuando se tiene expresiones declaradas pero no utilizadas.

Evaluación Perezosa

En promedio:

- Peor uso de espacio
- Peor desempeño (tiempo)

En el mejor de los casos:

Para algunos programas resulta ser **más eficiente** cuando se tiene expresiones declaradas pero no utilizadas.

Pero ¿el no realizar algunas operaciones (ev. perezosa) siempre resulta ser algo “bueno” en el diseño de lenguajes?

Ejemplo:

```
y = 0;

x = { 5 / { 3 * { 2 * { 1 * y }}}}
```

¿Qué ocurrirá en los lenguajes con estrategia de eva. perezosa?

No se detectará este error hasta que se use a “x” y esto puede o no ocurrir… si se llama o no, puede estar dentro de una rutina de EXCEPTIONs ¿?

¿Qué ocurrirá en los lenguajes con estrategia de eva. glotona?

Se detecta desde un inicio algún posible error de semántica.

¿Qué pasa cuando evaluamos?

Evaluación Glotona

x = {1 + 2}

x = 3

Evaluación Perezosa

x = {1 + 2}

Hasta que se requiera evaluar...

Transparencia Referencial

La transparencia referencial se traduce comúnmente como la capacidad de "reemplazar iguales por iguales".

Por ejemplo, siempre podemos reemplazar 1 + 2 con 3.

Ahora pensemos en esa definición: ¿cuándo no se puede reemplazar algo con alguna otra cosa, y no sea igual a la cosa original?

La transparencia referencial describe una relación. Ésta relaciona pares de términos exactamente cuándo pueden considerarse equivalentes en todos los contextos.

Por lo tanto, en la mayoría de los lenguajes:

- 1 + 2 es referencialmente transparente a 3 (asumiendo que no hay desbordamiento).
- √4 es referencialmente transparente a 2 (asumiendo que la función de raíz cuadrada devuelve solo la raíz positiva).

Haskell: relación es inherentemente más fuerte en lenguajes sin mutación.

Ahora podemos hacer la siguiente pregunta: ¿cómo debe de ser la relación de transparencia referencial para un programa en un lenguaje dado?

Supongamos:

x = {1 + 2} es decir x = 3

Pero en algunos lenguajes pueden no solo mutar el valor de la “x” pueden cambiar su tipo.

x podría ser asignada por algún programador como bool, i.e. mutamos su valor y tipo, podría ocasionar algún error en tiempo de ejecución (tiempo de ejecución)

Los lenguajes donde NO se permite la mutación explícita: Haskell (composición de funciones)

Tarea Opcional 1

Medio punto extra sobre el **examen 2**: haz una lista de al menos 5 características de cada uno de las estrategias de evaluación en lenguajes de programación, puedes describir sus características, ventajas y desventajas, justifica cada punto. Usa ejemplos para desarrollar cada uno de los puntos.

Entrega: 17 de marzo de 2023 hasta las 12 de la noche, por correo a la profesora.
Tarea individual.

Formato de entrega: archivo PDF, debe de contener tu nombre completo.

Tarea Opcional 2

Medio punto extra sobre el examen 2: leer el Prefacio del libro “Elogio de la Pereza” del *Dr. José de Jesús Galaviz Casas y hacer una* **reflexión de la lectura.**
¿Crees que los computólogos en general trabajamos bajo la regla de “entregar hasta el último momento” y hacer generalizaciones de soluciones a problemas que se nos encomiendan, para ya no volver a hacer dicha tarea?

Entrega: 17 de marzo de 2023 hasta las 12 de la noche.
Tarea individual y la entrega es opcional. Enviar al correo de la profesora.
Formato de entrega: archivo PDF, debe de contener tu nombre completo.

http://repositorio.fciencias.unam.mx:8080/xmlui/bitstream/handle/11154/177712/2003%20Galaviz%2C%20J.%20Libro%20-%20Elogio%20Pereza.pdf?sequence=1

¿DUDAS?

Gracias