Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación

Tarea 4: Estrategias de Evaluación

Johann Ramón Gordillo Guzmán
418046090

José Jhovan Gallardo Valdéz
310192815

Tarea presentada como parte del curso de **Lenguajes de Programación** impartido por la profesora **M.I. Karla Ramírez Pulido**.

31 de Marzo del 2020

Link al código fuente: https://github.com/JohannGordillo/

1. Preguntas

1. Evalúa la siguiente expresión usando el tipo de alcance y régimen de evaluación que se indica. Es necesario incluir el ambiente final en cada caso.

$$\{with\ \{a\ \{+\ 2\ 2\ \}\}\ \{with\ \{b\ \{+\ a\ a\ \}\}\ \{with\ \{foo\ \{fun\ \{x\}\ \{-\ x\ b\ \}\}\}\ \{with\ \{a\ \{-\ 2\ 2\ \}\}\ \{with\ \{b\ \{-\ a\ a\ \}\}\ \{foo\ -3\}\}\}\}\}$$

   a) Alcance estático y evaluación glotona.

   Respuesta.

   Primero, creamos el ambiente en forma de lista:

   Env 0 = ()
   Env 1 = ((a 4))
   Env 2 = ((b 8) (a 4))
   Env 3 = ((foo {fun {x} {- x b}}) (b 8) (a 4))
   Env 4 = ((a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))
   Env 5 = ((b 0) (a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))

   Env = ((b 0) (a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))

   Tenemos que evaluar {foo -3}.

   foo es {fun {x} {- x b}}, por lo que al sustituir x = -3, tenemos (- -3 b).

   Como estamos usando alcance estático, tenemos que b es 8, por lo que el resultado final será (- -3 8), que es -11.

   $\therefore$ El resultado final es -11.

   b) Alcance dinámico y evaluación glotona.

   Respuesta.

   Primero, creamos el ambiente en forma de lista:

   Env 0 = ()
   Env 1 = ((a 4))
   Env 2 = ((b 8) (a 4))
   Env 3 = ((foo {fun {x} {- x b}}) (b 8) (a 4))
   Env 4 = ((a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))
   Env 5 = ((b 0) (a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))

Env = ((b 0) (a 0) (foo {fun {x} {- x b}}) (b 8) (a 4))

Tenemos que evaluar {foo -3}.

foo es {fun {x} {- x b}}, por lo que al sustituir x = -3, tenemos (- -3 b).

Como estamos usando alcance dinámico, tenemos que b es 0, por lo que el resultado final será (- -3 0), que es -3.

∴ El resultado final es -3.

c) Alcance estático y evaluación perezosa.

Primero, creamos el ambiente en forma de lista:

Env 0 = ()
Env 1 = ((a (+ 2 2)))
Env 2 = ((b (+ a a)) (a (+ 2 2)))
Env 3 = ((foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))
Env 4 = ((a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))
Env 5 = ((b (- a a)) (a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))

Env = ((b (- a a)) (a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))

Tenemos que evaluar {foo -3}.

foo es {fun {x} {- x b}}, que al sustituir x = -3 se obtiene (- (-3) b)

Como estamos usando alcance estático, b es (+ a a).

Luego, al estar usando alcance estático, a es (+ 2 2).

Finalmente, b es (+ (+ 2 2) (+ 2 2)).

Ahora únicamente resta evaluar (- (-3) (+ (+ 2 2) (+ 2 2))), que se reduce a (- (-3) (+ 4 4)), luego a (- (-3) 8) y de esto último se obtiene -3 - 8 = -11, que es el resultado final.

∴ El resultado final es -11.

d) Alcance dinámico y evaluación perezosa.

Respuesta.

Primero, creamos el ambiente en forma de lista:

Env 0 = ()
Env 1 = ((a (+ 2 2)))
Env 2 = ((b (+ a a)) (a (+ 2 2)))
Env 3 = ((foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))

Env 4 = ((a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))
Env 5 = ((b (- a a)) (a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))

Env = ((b (- a a)) (a (- 2 2)) (foo {fun {x} {- x b}}) (b (+ a a)) (a (+ 2 2)))

Tenemos que evaluar {foo -3}.

foo es {fun {x} {- x b}}, que al sustituir x = -3 se obtiene (- (-3) b)

Como estamos usando alcance dinámico, b es (- a a).

Luego, al estar usando alcance dinámico, a es (- 2 2).

Finalmente, b es (- (- 2 2) (- 2 2)).

Ahora únicamente resta evaluar (- (-3) (- (- 2 2) (- 2 2))), que se reduce a (- (-3) (+ 0 0)), luego a (- (-3) 0) y de esto último se obtiene -3 - 0 = -3, que es el resultado final.

$\therefore$ El resultado final es -3.

2. Usando las listas pseudo-infinitas de Haskell define una lista que genere la sucesión de Fibonacci y obtén los primeros elementos usando la función **take**.

Respuesta.

Aquí está el código:

```haskell
-- | ========================================
-- | >> Author:
-- |          Johann Gordillo
-- | >> Email:
-- |          jgordillo@ciencias.unam.mx
-- | >> Date:
-- |          03 / 31 / 2020
-- | ========================================
-- | Implementation of the Fibonacci's sequence
-- | using infinite lists in Haskell.
-- | ========================================

module Fibo where

-- | Returns an infinite list containing the Fibonacci sequence.
fibs :: [Integer]
fibs = 0 : 1 : [n | x <-[2..], let n = ((fibs !! (x-1)) + (fibs !! (x-2)))]

-- | Returns the first 'n' terms of the Fibonacci sequence.
-- |
-- | Example:
-- | >> takeNFibs 10
-- | [0,1,1,2,3,5,8,13,21,34]
takeNFibs :: Int -> [Integer]
takeNFibs n = take n fibs

```

Por ejemplo, veamos las siguientes ejecuciones:

```
*Fibo> takeNFibs 10
[0,1,1,2,3,5,8,13,21,34]
*Fibo>
*Fibo>
*Fibo> takeNFibs 20
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181]
*Fibo>
*Fibo>
*Fibo> takeNFibs 30
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811
,514229]
*Fibo>
*Fibo>
*Fibo> takeNFibs 40
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811
,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986]
*Fibo>
*Fibo>
*Fibo> takeNFibs 50
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,75025,121393,196418,317811
,514229,832040,1346269,2178309,3524578,5702887,9227465,14930352,24157817,39088169,63245986,102334155,165580141,267914296
,433494437,701408733,1134903170,1836311903,2971215073,4807526976,7778742049]
*Fibo>
*Fibo>
*Fibo>
```

Y podríamos seguir así sucesivamente con n's cada vez más grandes.

3. Usando la implementación de listas pseudo-infinitas en Racket, vista en clase, define la misma sucesión que en el Ejercicio 2 obtén los primeros elementos usando la función **stake**.

Respuesta.

Aquí está el código:

```
#lang plai

;; Predicado para definir estructuras genéricas
;; any?: any -> boolean
(define (any? a) #t)

;; Definición de listas pseudo-infinitas.
(define-type Stream
  [sempty]
  [scons (head any?) (tail procedure?)])

;; Función que obtiene el resto de un stream.
;; stail: stream -> stream
(define (stail s)
  ((scons-tail s )))

;; Función que obtiene los primeros n elementos de un stream en forma
;; de lista.
;; stake: number stream -> list
(define (stake n l)
  (if (zero? n)
      empty
      (cons (scons-head l) (stake (sub1 n) (stail l)))))

;; Función generadora de la sucesión de Fibonacci.
;; fibonacci-generator: stream
(define (fibonacci-generator n m)
  (scons n (thunk (fibonacci-generator m (+ n m)))))

;; Lista infinita de la sucesión de Fibonacci con primeros elementos a 0 y a 1.
;; fibonacci: stream
(define (fibonacci)
  (fibonacci-generator 0 1))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
> (stake 10 (fibonacci))
'(0 1 1 2 3 5 8 13 21 34)
>
```

En la parte inferior de la imagen se muestra una ejecución usando **stake** para obtener los primeros 10 elementos de la sucesión.

2. Bibliografía

- Ramírez, K. (2020).
  Notas del curso de Lenguajes de Programación.
  Facultad de Ciencias - UNAM
  Ciudad de México, México.

- Krishnamurthi, S. (2017).
  Programming Languages: Application and Interpretation.
  Estados Unidos de América.

- Sookocheff, K. (2018).
  Alpha Conversion.
  Recuperado el 23 de marzo del 2020 de:
  https://sookocheff.com/post/fp/alpha-conversion/

- Sookocheff, K. (2018).
  Beta Reduction.
  Recuperado el 23 de marzo del 2020 de:
  https://sookocheff.com/post/fp/beta-reduction/