Lenguajes de Programación, 2017-2

Práctica 5: Evaluación perezosa

Manuel Soto Romero

Fecha de inicio: 31 de marzo de 2017

Fecha de término: 21 de abril de 2017

1. Objetivos

- Programar una biblioteca para trabajar con arreglos perezosos, la implementación de éstos se hará usando una estructura llamada *Stream* que permite definir posibles listas infinitas.
- Implementar una versión completa del intérprete para el lenguaje diseñado en la Práctica 4 del curso. La implementación deberá soportar condicionales, identificadores, expresiones aritméticas, expresiones booleanas y funciones de primera clase, usando evaluación perezosa para interpretar las expresiones[1].

2. Antecedentes

2.1 Streams

Un stream es una posible lista infinita con valores que siguen un patrón específico[1]. Por ejemplo, para definir la lista `'(1 3 7 11 13 ...)` de los números impares se debe seguir un patrón $2n+1$.

Para implementar esta estructura, y la cual genere los valores de acuerdo a dicho patrón, se requiere de un comportamiento perezoso: postergar la evaluación del siguiente elemento hasta que sea requerido.

Para postergar la evaluación, podemos hacer uso de *thunks*, definidos como funciones sin parámetros, los cuales se ven de la siguiente forma:

`(lambda () <cuerpo>)`[2]

Usaremos estas funciones para postergar la evaluación, pues el cuerpo no se evalúa hasta que se intente aplicar la función. En el cuerpo de la función es donde se define el patrón para generar los valores. Por ejemplo, una forma de definir a los números naturales mediante *streams* es:

```
(define (genera-naturales n)
   (scons n (thunk (genera-naturales (+ n 1)))))
```

[1] La práctica se entrega siguiendo los lineamientos especificados en la página del curso `http://lenguajesfc.com/lineamientos.html` y por equipos de tres integrantes.

[2] Racket provee una forma resumida de esto: `(thunk <cuerpo>)`

Esta función genera un *stream* (lista infinita) con los números naturales iniciando en cero. Podemos notar que la creación de un *stream* es muy parecida a la creación de listas, sin embargo, la definición recursiva no está dada por otra lista en sí, sino por un *thunk* cuyo cuerpo es la llamada recursiva a la función con el patrón que permite generar el siguiente elemento del *stream*.

Esta estructura sirve como base para implementar otras estructuras de datos con comportamiento perezoso como los arreglos que se implementarán en la primera parte de esta práctica.

2.2 Intérpretes perezosos

Para implementar un intérprete con comportamiento perezoso, se tiene que considerar que los identificadores no deben almacenarse con su valor en el ambiente sino que guardaremos una expresión sin evaluar.

2.2.1 Cerraduras de expresiones

Para guardar las expresiones sin evaluar, usaremos *cerraduras de expresiones*, algo muy similar a lo que hacíamos con las cerraduras de funciones.

Las cerraduras de expresiones almacenan:

- La expresión propiamente dicha.
- El ambiente dónde fue definida.

2.2.2 Puntos estrictos

Al trabajar con el comportamiento perezoso, tenemos que forzar la evaluación en algún momento, por ejemplo si tenemos:

```
{+ {+ 4 5} {+ 4 5}}
```

Debemos poder evaluar esa expresión, pues el resultado de la suma no debe postergarse, debemos forzar su evaluación.

A los puntos donde la implementación de un lenguaje perezoso fuerza la reducción de una expresión a un valor (si existe) se les llama puntos estrictos del lenguaje[2].

En esta versión del intérpre, los puntos estrictos son:

- Operaciones n-arias.
- Condicionales de `if` y `cond`.
- La primera expresión de la aplicación de funciones.

3. Archivos requeridos

Anexo a este archivo en formato PDF se encuentran los siguientes archivos, necesarios para desarrollar la práctica:

- Un arhivo `streams.rkt` que contiene la biblioteca para trabajar con *streams*.
- Un archivo `arreglos.rkt` que contiene la biblioteca para trabajar con arreglos perezosos usando *streams*.
- Un archivo `grammars.rkt` que contiene los TDA necesarios para implementar el intérprete.
- Un archivo `parser.rkt` que contiene las funciones necesarias para convertir sintaxis concreta en la sintaxis abstracta correspondiente.
- Un archivo `interp.rkt` que contiene los procedimientos necesarios para evaluar las expresiones del lenguaje.
- Un archivo `practica5.rkt` que se encarga de ejecutar el intérprete final, usando todas las funciones anteriores.

4. Desarrollo de la práctica

Parte I: Arreglos perezosos

Completar el cuerpo de las funciones faltantes del archivo `arreglos.rkt` como sigue[3]:

1. (1 pt.) La función `(crea tipo dim)` debe crear un arreglo del tipo y dimensión especificados. Se debe crear un *stream* cuyas entradas sean `(nulo)`, no se debe regresar un *stream* vacío. Ejemplo:

   ```
   > (crea integer? 4)
   (arreglo integer? 4 (scons (nul) (thunk ...)))
   ```

2. (1 pt.) La función `(agrega arr ind elm)` debe agregar en la posición `ind` del arreglo `arr` el elemento `elm`. No se debe incrementar el tamaño del *stream* asociado, simplemente modificar el valor de dicha posición. Ejemplo:

   ```
   > (agrega (crea arr integer? 4) 0 1729)
   (arreglo integer? 4 (scons 1729 (thunk ...)))
   ```

3. (1 pt.) La función `(obten arr ind)` debe regresar el elemento en la posición `ind` del arreglo `arr`. Ejemplo:

   ```
   > (obten (agrega (crea arr integer? 4) 0 1729) 0)
   1729
   ```

[3]Usar las funciones de la biblioteca streams.rkt implementadas en laboratorio

4. (1 pt.) La función `(imprime arr)` que imprime el arreglo `arr` en la pantalla. La llamada a esta función no debe regresar una cadena, sino el resultado de una llamada a la funcion `display`. Los arreglos deben mostrarse con el formato: `[e1|e2|e3|...|en]`, si un elemento es nulo, dejar la entrada vacía. Ejemplo:

   ```
   > (imprime (agrega (crea arr integer? 4) 0 1729))
   [1729| | | ]
   ```

Parte II: Intérprete perezoso

Completar los siguientes ejercicios para lograr la correcta ejecución del archivo `practica5.rkt` que implementa un intérprete para el siguiente lenguaje descrito en notación EBNF[4]:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | {<op> <expr>+}
         | {if <expr> <expr> <expr>}
         | {cond {<expr> <expr>}+ {else <expr>}}
         | {with {{<id> <expr>}+} <expr>}
         | {with* {{<id> <expr>}+} <expr>}
         | {fun {<id>*} <expr>}
         | {<expr> <expr>*}

<id> := a | .. | z | A | ... | Z | aa | ab | ... | aaa | ...
        (Cualquier combinación de caracteres alfanuméricos
              con al menos uno alfabético)

<num> ::= ... | -2 | - 1 | 0 | 1 | 2 | ...

<bool> ::= true | false

<op> ::= + | - | * | / | % | min | max | pow
       | neg | and | or | < | > | <= | >= | = | !=
```

Ejercicios

1. (1 pt.) Completar el cuerpo de la función `(parse sexp)` contenida en el archivo `parser.rkt` que recibe una expresión en sintaxis concreta y posteriormente construir el árbol de sintaxis abstracta del lenguaje `CFWBAE/L`.

2. (2 pts.) Completar el cuerpo de la función `(desugar sexps)` contenida en el archivo `parser.rkt` que recibe una expresión dentro del lenguaje `CFWBAE/L` y eliminar el azúcar sintáctica, es decir, regresar el árbol de sintaxis abstracta dentro del lenguaje `CFBAE/L`.

   `CFBAE/L` es una versión *desendulzada* de `CFWBAE/L` que no cuenta con constructores para `with`, `with*` ni `cond`. Para eliminar el azúcar sintáctica de este tipo de expresiones, considerar:

[4]Del inglés Extended Backus–Naur Form

- `with` puede ser expresado como una aplicación de función endulzada.
- `with*` puede ser expresado como una cadena de expresiones `with` anidadas.
- `cond` puese ser expresado como una cadena de expresiones `if` anidadas. Por ejemplo:

```
{cond
   {{< 10 2} true}
   {{> 9 4} false}
   {else true}}
```

es una versión endulzada de

```
{if {< 10 2}
    true
    {if {> 9 4}
        false
        true}}
```

3. (3 pts.) Completar el cuerpo de la función `(interp expr env)` contenida en el archivo `interp.rkt` que recibe un árbol de sintaxis abstracta `CFBAE/L` y un ambiente de sustitución y a su vez regresa la interpretación del árbol como un valor de tipo `CFBAE/L-Value`[5].

   El funcionamiento correcto del intérprete, dependerá (1) tanto del cómo se introducen las variables al ambiente al aplicar una función, (2) de la implementación de una función `lookup` que recupere el valor de los identificadores, (3) como de la implementación de una función `strict` que aplique los puntos estrictos correspondientes.

Referencias

[1] Cornel University, *Notas del curso de Estructuras de Datos y Programación Funcional* en la Universidad de Cornell, Primavera-2011.

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

[3] Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[5] El ambiente usando una estructura de lista con comportamiento de pila está definido en el archivo `grammars.rkt`.