Lenguajes de Programación 2016-1
Tarea 2

Ricardo Garcia Garcia Juan Carlos López López Luis Rodrigo Rojo Morales

10 de octubre de 2015
Facultad de Ciencias UNAM

Problema I

En teoría y laboratorio hemos visto el lenguaje FAE, que es un lenguaje con expresiones aritméticas, funciones y aplicaciones de funciones. ¿Es FAE un lenguaje Turing-Completo?. Debes proveer una respuesta breve e inambigua, seguida de una justificación más extensa de tu respuesta. Hint: Investiguen sobre el combinador Y

Respuesta: Un lenguaje de paradigma funcional necesita tener condicionales y recursion para que sea un lenguaje turing completo. Con FAE podemos hacer condicionales haciendo un if0 y con una funcion que verifique si es cero para usarla en el iterprete, y para la recursión, pues el convinador Y es una función de alto orden que toma como paramero una funcion que no es recursiva y devuelve una función que lo es como estamos trabajando en racket este tiene funciones anonimas entonces puedes hacer por ejemplo: (define factorial (lambda (f) (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))) que no usas factorial dentro de factorial pero hace la recursion con la funcion anonima entonces como el lenguaje anfitrion de FAE es racket podemos implementar funciones de FAE de esta manera Basicamente es Extender la gramatica FAE a una RCFAE (recursion-condicionales-expresiones aritméticas) como lo vimos en clase.

Problema II

¿Java es glotón o perezoso? Escribe un programa para determinar la respuesta a esta pregunta. El mismo programa, ejecutado en cada uno de los dos regímenes, debe producir resultados distintos. Puedes usar todas las características de Java que gustes, pero debes mantener el programa relativamente corto: **penalizaremos cualquier programa que consideremos excesivamente largo o confuso** (Hint: es posible resolver este problema con un programa de unas cuantas docenas de lineas).

Respuesta: Java pertenece al regímen **glotón** Analizando el código al realizar la ejecucion el resultado es la excepcion division entre cero. A la hora de ejecutar el codigo en su equivalente version de Haskell tenemos que devuelve 5.0 Esto se debe a la forma en como esta evaluando, en el llamado a la funcion **comp** donde el parametro es 5/0 el resultado es 5, ya que se espera hasta el ultimo momento para evaluar la expresion.

Problema III

En nuestro intérprete perezoso, identificamos 3 puntos en el lenguaje donde necesitamos forzar la evaluación de las expresiones closures (invocando a la función `strict`): la posición de la función de una aplicación, la expresión de prueba de una condicional, y las primitivas aritméticas. Doug Oord, un estudiante algo sedentario, sugiere que podemos reducir la cantidad de código reemplazando todas las invocaciones de `strict` por una sola. En el interprete visto en el capitulo 8 del libro de Shriram elimino todas las instancias de `strict` y reemplazo

```
[id (v) (lookup v env)]
```

por

```
[id (v) (strict (lookup v env))]
```

El razonamiento de Doug es que el único momento en que el interprete regresa una expresión closure es cuando busca un identificador en el ambiente. Si forzamos esta evaluación, podemos estar seguros de que en ninguna otra parte del interprete tendremos un closure de expresiones, y eliminando las otras invocaciones de `strict` no causaremos ningún daño. Doug evita razonar en la otra dirección, es decir si esto resultara o no en un interprete mas glotón de lo necesario.

Escribe un programa que produzca diferentes resultados sobre el interprete original y el de Doug. Escribe el resultado bajo cada interprete e identifica claramente cual interprete producirá cada resultado. Asume un lenguaje interpretado con características aritméticas, funciones de primera clase, with, if0 y rec (aunque algunas no se encuentren en nuestro interprete perezoso). Hint: Compara este comportamiento contra el interprete perezoso que vimos en clase y no contra el comportamiento de Haskell.

Si no puedes encontrar un programa como el que se pide, defiende tus razones de por que no puede existir, luego considera el mismo lenguaje con cons, first y rest añadidos.

Respuesta: En el archivo ej3.rkt usa interprete original:

```
> (rinterp (cparse '{{fun {x} x}1}))
(exprV (num 1) (mtSub))
```

El archivo ej3doug.rkt usa el interprete de doug

```
> (rinterp (cparse '{{fun {x} x}1}))
(numV 1)
```

Evaluan la misma expresion pero los resultados son diferentes, esto pasa porque cuando evalua cuando llega la parte de hacer el lookup, el interprete original hace lookup y ya devuelve lo que le de y el de doug con lo que de lookup hace strict y lo que va a hacer strict es seguir haciendo strict hasta que obtenga algo que no es exprV y en este caso en algun momento strict va a dar (numV 1) y eso es lo que va a devolver strict.

Problema IV

Ningún lenguaje perezoso en la historia ha tenido operaciones de estado (tales como la mutación de valores en cajas o asignación de valores a variables) ¿Por que no?

La mejor respuesta a esta pregunta incluiría dos cosas: un pequeño programa (que asume la evaluación perezosa) el cual usara estado y una breve explicación de cual es el problema que ilustra la ejecución de dicho programa. Por favor usa la noción original (sin cache) de perezosos sin cambio alguno. Si presentas un ejemplo lo suficientemente ilustrativo (el cual no necesita ser muy largo), tu explicación sera muy pequeña.

Respuesta: Para responder este ejercicio usamos una función en Haskell (primos(n)), porque Haskell usa evaluación perezosa, esta funcion nos regresa una lista de primos hasta n, para hacer esto usamos una lista infinita con elementos que cumplen una condicion (ser primo), gracias a la evaluación perezosa podemos definirlo con una lista infinita y evaluar hasta n y tambien no hace una asignacion de una lista infinita a una variable. En comparacion java, que usa evaluación glotona, no podriamos hacer una lista infinita porque tendria que hacer asignacion a una variable de la lista infinita y se acabaria la memoria.