Lenguajes de Programación 2016-1
Tarea 2

Ricardo Daniel Dondiego Pacheco

Miguel Ángel Mendoza Ponce
Serrato Solano Victor Manuel

October 10, 2015

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:

FAE cumple con que tiene la condicinal "if", tambien tiene expresiones aritmeticas, funciones, aplicaciones de funciones y que podemos tener recursion general en nuestras funciones en funcion de un combinador "Y". Con esto podemos decir que es turing completo.

Para nuestra condicional "if" podemos utilizar dos casos:
TRUE: regresaremos esto en caso de que nuestro predicao regrese verdadero, entonces ejecutaremos lo que corresponda a este caso.
FALSE: que regresara si no se cumplio el predicado anterior y y entonces ejecutaremos lo correspondiente a este caso, que podemos definirlo o no.

Para hacer recursion en nuestra aplicacion de funcion utilizamos el combianador Y, que hasta que no se cumpla el caso base, se aplicara.

en genera el combinador "Y" funciona asi:
(Y f) = f(Y f) = f(f(Y f)) = ...
de forma que al usar nuestra condicional "if" junto con el combinador "Y" podemos definar recursion general.

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 es Gloton, pues al hacer un programa con recursión que requiea muchas llamadas llenara el stack de memoria pues si fuera perezoso pues no terminaria su stack en nuestro programa.

Primero tratamos de simular una evaluacon perezosa donde java lo maneja como short-circuit evaluation (evaluacion de cortocircuito) que si tenemos en una operacion booleana en este caso "&&" (AND) para evitar caer en un error de memoria para hacer falsa nuestra expresion.
En la segunda parte corremos la funcion sin evaluacion de cortocircuito y se termina el stack de memoria por ser evaluacion glotona.

El archivo se adjunta como ProblemaII.java

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: Se anexan dos archivos: En ejercicio3-original.rkt se usa el intérprete original,en ejercicio3-doug.rkt se utiliza el intérprete de doug.

Ambos utilizan la misma expresión pero el intérprete original devuelve :

```
> (exprV (num 10) (mtSub))
```

y el intérprete de doug devuelve:

```
> (numV 10)
```

Cuando los intérpretes llegan a lookup, el original devuelve el valor que le llega, en el de doug con stric t lo aplica hasta que no encuentra una exprV y ahí nos devuelve el resultado ya mencionado.

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: En contrapositiva usando el hecho de que java es glotón y usando asignación de valores a variables vemos que de hecho no importa ya que el ambiente de una función en java está bien definido por un orden lineal, y no cambia el sentido de el programa, donde de ser perezoso su ambiente estaría definido por el ultimo valor alcanzable hacia arriba en el código por lo que ésto podría cambiar por completo el sentido de un programa, razón por la cual la evaluación perezosa no permite asignación de valores a variables.