Lenguajes de Programación
Tarea II

Andrea Itzel González Vargas
Karla Esquivel Guzmán
Carlos Gerardo Acosta Hernández

Entrega: 10 de Octubre de 2015
Facultad de Ciencias UNAM

Problema I

FAE es un lenguaje Turing-Completo. Decimos que un lenguaje es Turing-Completo en el paradigma funcional si es posible hacer recursión general con él. Este concepto, viene acompañado del condicional ïf", para que pueda cumplirse.

- **Condicional**

  En el *cálculo lambda* es posible representar tipos de datos como booleanos y números naturales. El lenguaje FAE ya nos permite directamente el uso de números y dado que el lenguaje anfitrion de FAE es *Racket*, es posible dar una representación de los booleanos gracias a las funciones anónimas y a partir de dichas definiciones construir el condicional *if-then-else*. En implementaciones que extienden de FAE, ya contamos directamente con el condicional *if0*.

  En cálculo lambda puro, las definiciones de los booleanos son de la siguiente forma:

  $$\text{true} =_{def} \ \lambda x \lambda y.x$$
  $$\text{false} =_{def} \ \lambda x \lambda y.y$$

Mientras que el condicional podemos construirlo como

$$\text{if-then-else} =_{def} \ \lambda v.\lambda t \lambda f.vtf$$

Entonces para la rama del *then* tendríamos

$$\text{if-then-else true e1 e2} \rightarrow^* \ e1$$

Y para el *else*

$$\text{if-then-else false e1 e2} \rightarrow^* \ e2$$

- **Recursión**

  Gracias al combinador Y, que es una función que nos permite resolver la recursión dada por una función. No podemos hacer recursión directamente con FAE, sin embargo, el combinador Y nos permite definir la recursión general sobre funciones de FAE.
  Es mostrada una implementación en el *Shriram* como sigue:

```
(lambda (p)
  ((lambda (f)
     (f f))
   (lambda (f)
     (p (f f)))))
```

  Siendo que su propiedad trabaja de forma que $(Y\ f) = f(Y\ f) = f(f((Y\ f))) = ...$ Y es posible de resolver gracias al condicional, decimos que es tenemos recursión general en el lenguaje FAE. Por lo tanto es Turing-completo.

Problema II

Comenzemos diciendo que *Java* es un lenguaje de programación con evaluación glotona. Para probarlo, nos valdremos del lanzamiento de excepciones de *Java* en tiempo de ejecución. En el código fuente que se anexa a

la carpeta de esta tarea, viene escrito un método llamado *proffeager* en la clase Prueba. Dicho método recibe un entero y devuelve otro; en el cuerpo del método se declara una asignación de variable a una expresión aritmética que de antemano sabemos que al resolverse devuelve la excepción *"java.lang.ArithmeticException: / by zero"*.

Ahora supongamos que *Java* tiene evaluación perezosa. Esto es lo que debe ocurrir bajo la aplicación del método:
Primero ocurre la asignación de la variable x con la expresión aritmética.
Luego, si el método recibe un 0, se devolverá el resultado de sumar 2 con 3.
Si el método recibe un 1, se evaluará el valor de x y se imprimirá en pantalla.
En cualquier otro caso se imprimirá un 42.
En el cuerpo del método *main* llamamos primero la función con parámetro 0, luego con 1 y finalmente con 33, pues esperamos que imprima un 5, luego lance una excepción que capturaremos y finalmente imprima un 42. Sin embargo lo que ocurre es muy distinto. Al ejecutarse, el programa lanza una excepción por cada llamada a función, cuando se suponía sólo se lanzaría una para el caso particular de la entrada 1. Bajo nuestros supuestos, hemos llegado a una contradicción.

No es difícil asociar dichos errores con la asignación de la variable. Sin embargo, nos aseguramos de imprimir la ruta al culpable de lanzar la excepción en su captura, y efectivamente se trata de la asignación en la línea 5. Esto quiere decir que lo primero que hizo *Java* fue evaluar la expresión aritmética en la asignación, en lugar de asignar la operación aritmética directamente, como haría un lenguaje perezoso. Cabe señalar que para la asignación el tipo primitivo *int* no se induce un punto estricto (si Java fuera perezoso), pues bien podría tratarse de un método que devolviera la evaluación de una expresión aritmética errónea, pero igualmente, el método sería llamado en la asignación (aunque estuviera definido el resultado del método en el dominio int) y no hasta el uso de la variable.

Para hacer más claro nuestro razonamiento, hemos decidido anexar a la carpeta un programa en *Haskell* con una estructura similar. Ya se ha dicho en clase que *Haskell* es un lenguaje con evaluación perezosa, por lo que se espera que para el primer caso de la función $profflazy$ definida en el código, la asignación de la variable ocurra sin evaluar la expresión a la que se enlaza y devuelva sin problemas la operación aritmética de sumar 2 y 3. Y para el segundo caso, debe lanzar un error al exigirse el valor de la variable x en el cuerpo del la expresión *let*, que se evaluará hasta ese momento. Y efectivamente ocurre.

Problema III

Ya hemos platicado en clase que una función genera su propio closure, con su parámetro como identificador, el cuerpo de la función como expresión del lenguaje asociada y el ambiente del que procede. Podemos fácilmente señalar en el código que nos proporcionaron, dicho comportamiento del intérprete en la línea 163. Considerando lo que dice nuestro querido Doug Oord, en el momento que afirma que sólo en la búsqueda del valor de un identificador en el ambiente es cuando se obtiene un closure, sospechamos que no es correcto su razonamiento.

A continuación el programa que escribimos con la idea de que produciría resultados distintos para el intérprete normal *(Shriram, capítulo 8)* y el intérprete de Doug.

```
{with {f {fun {x} x}} {f 4}}
```

La hipótesis era que, mientras el intérprete de Doug forzaría un punto estricto sobre el id $'x$ que dejamos como resultado de la función (identidad) en la interpretación del programa, pues suponemos que no se contempla el hecho de que la función induce un closure, el intérprete normal no forzaría una evaluación, sino que devolvería la expresión de valor asociada al id sin evaluar.

Los resultados fueron los siguientes:

- `> (rinterp (cparse '{with {f {fun {x} x}} {f 4}}))`
  `(exprV (num 4) (aSub 'f (exprV (fun 'x (id 'x)) (mtSub)) (mtSub)))`

  Para el intérprete normal. Y

- `> (rinterp (cparse '{with {f {fun {x} x}} {f 4}}))`
  `(numV 4)`

  Para el intérprete de Doug.

Es importante decir que el resultado es equivalente, sin embargo, podemos observar que el intérprete de Doug es "más glotón" que el normal y son resultados distintos. De acuerdo a esto, bajo la aplicación de strict (una interpretación más) sobre el resultado del primer intérprete, obtenemos el resultado del segundo. O séase que

```
> (strict (exprV (num 4) (aSub 'f (exprV (fun 'x (id 'x)) (mtSub)) (mtSub))))
(numV 4)
```

De tal suerte que se confirma nuestra hipótesis.

Problema IV

Los lenguajes perezosos no pueden permitirse las operaciones de estado porque comprometerían la integridad en un programa, entre una evaluación esperada y la que resulta en realidad. Puesto de esa manera, cambiaría el significado del programa.

Suponiendo que *Racket* fuera un lenguaje perezoso, escribimos un programa ejemplo (estado.rkt) para ilustrar el problema de utilizar una asignación de valores como ocurre gracias a la función *set*! que nos ofrece el lenguaje. En el cuerpo del programa se define la función recursiva fib (de fibonacci) y una función más *change-state* que dados dos valores asigna el primero al principio de la función a la variable x, luego se le asigna a la variable y la llamada a la función fib con la primera asignación de x, posteriormente se cambia el valor de x por el segundo parámetro de la función y finalmente se devuelve el valor de y. Dada nuestra forma de escribir y pensar el código (linealmente, como dicta la intuición), uno esperaría que se devolviera el $fibonacci$ del primer parámetro de la función. Sin embargo, como ocurrió la segunda asignación y tenemos en frente un supuesto lenguaje con evaluación perezosa, lamentablemente para el momento de la evaluación de y, ya habrá ocurrido la segunda asignación de x y en realidad obtendríamos el $fibonacci$ del segundo parámetro recibido.

Claramente, en este caso particular no es muy difícil evitar errores si tenemos un programa tan pequeño. Simplemente no hacemos la segunda asignación, en esta ocasión, futil. Por otro lado si tenemos un programa de mayores proporciones lo más probable que ocurra es que obtengamos salidas o resultados inesperados. Como usuarios de un lenguaje de programación con evaluación perezosa y operaciones de estado, estaríamos expuestos a un bajo control sobre nuestra implementación.