

Examen – Lenguajes de Programación (CC41A)

Departamento de Ciencias de la Computación – Universidad de Chile

Profesor: Éric Tanter

14 de Julio 2008

duración: 2 horas / 1.5pt por pregunta

sin apuntes

1. Preguntas

1. *a)* Defina en Scheme la función `compose` que toma como parámetro dos funciones unarias f y g y retorna una función que corresponde a la composición $f \circ g$. Eg:

```
((compose square add1) 7) --> 64
```

- b)* Usando `compose`, defina la función `repeated` que toma una función f y un número $n > 0$, y retorna una función que corresponde a $f \circ f \circ \dots \circ f$ (f aplicada n veces). Eg:

```
((repeated add1 10) 5) --> 15
```

2. *a)* Los lenguajes con evaluación perezosa no incluyen expresiones de mutación (como asignaciones). Explique porque es así, usando un ejemplo pequeño que ilustre su argumento.

- b)* Un compañero le pide a usted que implemente la función `if0` en Scheme que recibe como tres parámetros: `e1`, `e2`, `e3`; tal que evalúa `e2` si `e1` es 0, `e3` sino. La restricción que le imponen es que sea usada de la siguiente manera.

```
(if0 (- 4 5)
      (write "zero")
      (write "not zero"))
-> "not zero"
```

¿Qué le responde usted a su compañero? ¿Cambia su respuesta si le pide que sea implementada en Haskell?

3. Estamos usando un lenguaje que usa un GC tipo copy-collector, usando dos espacios y sin generaciones. Al ejecutar el programa, detectamos tiempos de detención durante el GC que son demasiado grandes para la aplicación. Un ingeniero propone agrandar el espacio de memoria dinámica con que se ejecuta el programa pero otro opina que eso hará que el GC demore más aún. Discuta esa afirmación.

Si el lenguaje provee una opción de cambiar el GC por un Reference Count, sería una buena idea usarla?

4. Considere el siguiente lenguaje (similar al FAE visto en clases):

```
<expr> ::= <id>
         | <num>
         | (+ <expr> <expr>)
         | (lambda (<id>) <body>)
         | (<expr> <expr>)
```

- Exprese, en este lenguaje, un programa que nunca termina.
- Especifique los juicios de tipos para las distintas expresiones de este lenguaje. No es necesario asumir que los programas llevan anotaciones de tipo (ie. puede suponer que tiene un inferenciador de tipos).
- ¿Es posible asignar un tipo a su programa que nunca termina? En caso en que no, esto significa que tiene un lenguaje (FAE+tipos) que asegura que todos los programas validos en tipos siempre terminan. ¿No es eso contradictorio con el hecho de que los sistemas de tipos son sujetos al Halting Problem? Comente.

2. Pauta

```
1. ;; (0.1 pts contrato)
   ;; compose:: (b -> c)x(a -> b) -> (a -> c)
   ;; (0.6 pts impl.)
   (define compose
     (lambda (f g)
       (lambda (x)
         (f (g x)))))

   ((compose (lambda (x) (* x x)) (lambda (x) (+ 1 x))) 7)

   ;; (0.1 pts contrato)
   ;;repeated :: (a -> a)x Number -> (a -> a)
   ;; (0.7 pts impl.)
   (define repeated
     (lambda (f n)
       (if (= n 1) f
           (compose f (repeated f (- n 1)))))

   ((repeated (lambda (x) (+ 1 x)) 10) 5)
```

2. a) (0.5 ptos) Los lenguajes Lazy evalúan las expresiones solo cuando las necesitan, entonces en el caso de que una sub-expresión tenga un cambio de valor a una variable mutable, dado que mutación implica una noción de tiempo (antes/después de la mutación), es prácticamente

imposible razones sobre las evaluaciones y resultados que dara el interprete. Además el usar mutación en el lenguaje implica que se pierde la propiedad de *transparencia referencial*, esto hace que no se pueda usar caching.

```
{define a {box 0}}
...
{{lambda {x}
  {seqn {set-box a {+ 1 {unbox a}}}
        {unbox a}}}
} {set-box a {+ 1 {unbox a}}}}
```

(0.4 ptos ejemplo, más su respectiva explicación). En este ejemplo se ve que la función no necesita el argumento `x` por tanto el interprete lazy no lo va evaluar, aunque en el parámetro actual este una expresión que cambie un valor mutable. Entonces un lenguaje lazy retorna 1, en un eager retorna 2, por tanto es difícil razones sobre los resultados que puede dar una expresión en la evaluación.

b)(0.3 ptos en Scheme y 0.3 ptos en Haskell) El problema que surge con la condicional `If0` es su forma de evaluar, ya que en un lenguaje Eager como Scheme se va evaluar los argumentos que corresponde a las opciones, entonces mostraría tanto `zero` como `not zero`. En el caso de un lenguaje Lazy no hay ningún problema, ya que los lenguajes Lazy solo evalúa el argumento que corresponda a la opción seleccionada.

3.
 - **(0.8 ptos)** Agrandar el espacio de memoria dinamica total ayuda a evitar que ocurra un GC muy seguido por lo que efectivamente disminuimos el numero total de GCs en un tiempo dado de ejecucion. Por otro lado, un copy-collector, con dos espacios, demora un tiempo proporcional a la cantidad de objetos usados, por lo que podriamos considerar cada ejecucion de tiempo constante, asi que efectivamente ganamos en tiempo de ejecucion. Sin embargo, si el problema era que los tiempos de detención son muy grandes, eso seguirá exactamente igual: habrá menos cantidad de esos eventos, pero cada uno demorará lo mismo que antes.
 - **(0.7 ptos)** Si existe la posibilidad de usar un Reference Count, efectivamente eliminaríamos los tiempos de detencion y arreglamos este punto. Sin embargo, habría que confirmar si en el lenguaje no se están usando estructuras cíclicas, porque estas no serán nunca recolectadas.
4.
 - **;; (0.5 ptos ejemplo)**

```
( (lambda (x) (x x))
  (lambda (x) (x x)) )
```

- **(0.5 ptos, 0.1 pts cada juicio)**

Juicio de tipos para números, Son todos los valores constantes de números permitidos por scheme.

$$\Gamma \vdash Z : number$$

Juicio de variables, la particularidad de este caso es la búsqueda de su tipo en el repositorio de tipos definidos, para nuestro caso inicialmente toda variable tiene un tipo

polimórfico y según su contexto se restringue o no.

$$\Gamma \vdash i : \Gamma(i)$$

Jucio de operaciones Aritméticas, toda operación aritmética hecha en nuestro tipos primitivos tiene como resultado un valor determinista único constante que pertenece al conjunto de los enteros.

$$\frac{\Gamma \vdash i : number \quad \Gamma \vdash d : number}{\Gamma \vdash \{+ i d\} : number}$$

Jucio de funciones, las funciones como son bien conocidas están definidas como procesos que recibe un parámetro y retorna un cálculo, en otras palabras una entrada y una salida.

$$\frac{\Gamma[i : \leftarrow t_1] \vdash b : t_2}{\Gamma \vdash \{fun(i : t_1) : t_2 \ b\} : (t_2 \rightarrow t_1)}$$

Jucio de aplicación de funciones, el caso de las aplicaciones es un estudio donde tanto la función derivada de un juicio anterior y su argumento real se encuentra para obtener un resultado concreto y estricto.

$$\frac{\Gamma \vdash f : \{t_1 \rightarrow t_2\} \quad \Gamma \vdash a : t_1}{\Gamma \vdash \{f \ a\} : t_2}$$

■ **(0.5 ptos, por ambas respuestas)**

No es posible determinar el tipo de la expresión en el primer inciso, esto se debe a que cada vez que quiera asignar un tipo a la función (`lambda (x) (x x)`) se entrará a un ciclo infinito. La función anónima mencionada anteriormente debe tener un tipo $t_1 \rightarrow t_2$, entonces `x` tendría que ser del tipo t_1 y luego en el cuerpo de la función se ve que `x` se aplica como una función a `x`, dando el tipo de `x`, $t_1 \rightarrow t_2$ (como el argumento es el mismo `x` ya definido y la salida es la salida del `lambda`). En conclusión tenemos que el tipo de `x` es $t_1 = t_1 \rightarrow t_2$, esta es una recurrencia sin caso base.

No contradice el *Halting Problem*, por que el lenguaje FAE+tipos no es turing completo. Esto se demuestra por que FAE no introduce llamadas recursivas como tal, ej. `{rec ...}`, por tanto FAE no es un lenguaje tan potente como uno que implementa ciclos.