# Pauta Control 3 – Lenguajes de Programación

Departamento de Ciencias de la Computación

Universidad de Chile

Profesor: Éric Tanter

15 de Junio 2009

2 horas

1. **(1.5 pts)** Benancio acaba de implementar en Scheme la función `(lista-pares n)`, que retorna la lista de los primeros `n` números pares. El código es el siguiente:

```
(define lista-pares
  (letrec
      ((lista '())
       (pares (lambda (n)
                (if (eq? n 0) lista
                    (begin
                      (set! lista (cons (* n 2) lista))
                      (pares (- n 1)))))))
   pares))
```

   *a*) **(0.5 pts)**¿Es correcta esa implementación? Ilustre. (Aplique `lista-pares` una vez, y una segunda vez.)

   No es correcta. Ya que al momento de la 2da ejecución, la variable `lista` sigue presente en la clausura de la función `pares`, por tanto todas las operaciones de {`cons .. lista`} aumentarán los valores en la lista.

```
(test (lista-pares 3) '(2 4 6))
(test (lista-pares 3) '(2 4 6)) ;; Bad
(test (lista-pares 1) '(2 2 4 6 2 4 6))
```

   *b*) **(1 pto)**¿Es necesario hacer uso de mutación para escribir `lista-pares`? Reescribala de forma funcional pura.

   No es necesario usar mutación, basta con pasar la lista acumulada como parámetro auxiliar a la función recursiva. Ver archivo: `lista-pares.ss`

   *c*) Por lo general, los lenguajes con evaluación perezosa no soportan mutación explícita. ¿Por qué?

   Los lenguajes Lazy evaluan las expresiones solo cuando las necesitan. 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/despues de la mutación), es practicamente imposible razonar sobre las evaluaciones y resultados que dara el intérprete. Además el usar mutación en el lenguaje implica que se pierde la propiedad de *transparencia referencial*.

2. **(1 pto)** Describa la evaluación del siguiente programa en el lenguaje VBCFAE detallando en cada paso qué pasa con el Environment y con el Store. ¿Cuál es el resultado obtenido?

   Sea claro al describir la evolución Environment/Store. Para ello **DEBE** usar diagramas.

```
{with {{x {box 100}}
       {y 20}}
  {with {{f {fun {z} {+ z {+ {unbox x} y}}}}}
    {seqn
      {set-box! x 200}
      {set! y 30}
      {with {{y 5}}
        {f 1}}}}}
```

   **Esbozo de solución.**

```
env                store
x -> 102            101 -> (numV 100)
                    102 -> (boxV 101)
y -> 103            103 -> 20
f -> 104            104 -> (closureV ...)
                           ;; quedan en la clausura (x -> 102, y -> 103)
                           ;; [0,3 pts]
                    101 -> (numV 200) ;; {set-box! x 200} [0,2 pts]
                    103 -> (numV 30)     ;; {set! y 30}   [0,2 pts]
y -> 106            106 -> (numV 5)      ;; [0,2 pts]
z -> 107            107 -> (numV 1);; {f 1}
                                ;;      => 1 + 200 + 30 = 231 [0,1 pts]
```

3. **(2 pts)** TinyScheme esta definido según la siguiente gramática:

```
<expr> ::= <id>
        |  <num>
        |  {lambda {<id>} <expr>}
        |  {<expr> <expr>}
        |  {let {<id> <expr>} <expr>}
        |  {+ <expr> <expr>}
```

   *a*) Tipos y tipos y tipos y tipos y...

   1) **(0,1 pts)**Exprese, en este lenguaje, un programa que nunca termina.

```
{{lambda {x} {x x}}
 {lambda {x} {x x}}}
```

   2) **(0,6 pts)**Modifique levemente la sintaxis de TinyScheme de modo de incluir anotaciones de tipos. Escriba los juicios de tipos que definen un sistema de tipo completo para TinyScheme. (a partir de ahora, TinyScheme se refiere al lenguaje tipeado.)

   **(0,1 pts)Sintaxis modificada:**

```
<expr> ::= <id>
        |  <num>
        |  {lambda {<type> <id>} <type> <expr>}
        |  {<expr> <expr>}
```

```
        |  {let {<type> <id> <expr>} <expr>}
        |  {+ <expr> <expr>}
<type> -> <num>
        |  <type> -> <type>
```

**(0.5 pts, -0.1 pts por cada faltante) Juicio de tipos**
**Jucio de tipos para números**

$$\Gamma \vdash Z : num$$

**Juicio de variables**

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

**Juicio de operaciones Aritméticas (Suma)**

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

**Juicio de funciones**

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

**Juicio de aplicación de funciones**

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

**Juicio del let**

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma[id \leftarrow t_1] \vdash e_2 : t_2}{\Gamma \vdash \{let\ \ \{t_1\ \ id\ e_1\}\ \ e_2\} : t_2}$$

3) **(0,2 pts)**Es posible asignar un tipo a su programa que nunca termina? Si lo es, escribalo. Si no, que puede decir de cualquier programa TinyScheme? Comente.

No es posible!. TinyScheme no tiene el mismo poder que una máquina de Turing ya que necesitariamos definir recursión y una regla de tipos para ella. Si usaramos la recursión en base al operador $Y$ entramos al problema de que este operador no es válido en tipos (no se puede obtener un tipo de él) [1].

*b*) **(0,2 pts)**¿TinyScheme cumple con las propiedades de *Type Safety* y *Type Soundness*? ¿y Scheme? Justifique.

Cumple con Type Soundness, por ejemplo la aplicación de una función con tipo $num \rightarrow num$ y argumento $num$ al momento de ser evaluada generará un valor que debe ser del tipo $num$. En si debería aplicarse una demostración formal, que garantice que la aproximación estática proporcionada por el sistema de tipos sea coherente con las evaluaciones (ejecución) de las expresiones en TinyScheme. Esta demostración es extensa para incluirla en la pauta, pero tanto la semántica como los juicios de tipos de TinyScheme son estandar y en lenguages similiar esta propiedad ya ha sido demostrada.

Cumple con Type safety. Dado que el lenguaje interpretador (Scheme) tiene chequeo de tipos dinámico, por ejemplo las primitivas de aplicación de suma siempre evaluan expresiones del tipo entero (por la primitiva + de Scheme). Por tanto toda evaluación siempre se aplica a expresiones con tipos válidos.

Scheme es safety, dado que verifica en tiempo de ejecución que cada expresión sea del tipo adecuado en su evaluación. Scheme al tener un sistema de tipos dinámico el concepto de soundness es inaplicable, dado que no hay un chequeo estático de tipos.

[1] Para mayor información ver 26.2 y 26.3 del PLAI

*c*) De tipos y subtipos...
Extendamos TinyScheme con estructuras de datos arbitrarias (como `define-type`), soportando además una noción de subtipos.

1) **(0,3 pts)** Introduzca una sintaxis, con anotaciones de tipos, para `define-type`, ilustrando con la definicion de un tipo `Point` (atributos numericos `x` e `y`), y un subtipo `ColorPoint`.

```
<expr> ...
       |  {def-type <id> {<type> <id>}* }
       |  {def-type <id> extends <id> {<type> <id>}* }
<type> ...
       |  <id>
```

Toda expresión que omite la definición de su supertipo, será automaticamente transformado (vía azucar sintáctico) a `.. extends Top ..` donde Top es el tipo primitivo base de los otros tipos (similar a Object en Java). Ejemplo del `Point` y `ColorPoint` se escribiría:

```
{def-type Point {{num x} {num y}}}
...
{def-type ColorPoint extends Point {num color}}
```

2) **(0,3 pts)** Describa la relación de subtipo entre funciones (juicio), e ilustre con un ejemplo.

$$\frac{\Gamma \vdash\ A' <: A \qquad \Gamma \vdash\ B <: B'}{\Gamma \vdash\ A \to B <: A' \to B'}$$

La relación de subtipos para funciones dice que $A \to B$ es subtipo de $A' \to B'$, si $A'$ es subtipo de $A$ (contravarianza) y $B$ es subtipo de $B'$ (covarianza). Un ejemplo en TinyScheme es: $Point \to num <: ColorPoint \to num$.

3) **(0,3 pts)** Si agregamos mutación y tipos referencia (como {`ref A`}), y tenemos `A <: B`, ¿qué relación podemos tener entre {`ref A`} y {`ref B`} ? Explique.
Supongamo que $ref\ A <: ref\ B$, entonces: Hay dos casos importantes que analizar, uno es la lectura y el otro la escritura. En el primero, si el contexto de la lectura espera obtener un valor con tipo $B$ y si la referencia contiene un valor con tipo $A$ (esto es lo mismo decir que la referencia tiene tipo $ref\ A$), entonces $A <: B$ debe cumplirse tal que no viole el contexto de la lectura. En la escritura, si usamos la misma referencia con tipo $ref\ A$ y el nuevo valor para la escritura tiene tipo $B$, entonces se debe cumplir $B <: A$ para poder hacer una escritura apropiada.
Por lo dicho anteriormente la unica relación existente entre $ref\ A$ y $ref\ B$ es:

$$\frac{\Gamma \vdash\ A <: B \qquad \Gamma \vdash\ B <: A}{\Gamma \vdash\ ref\ A <: ref\ B}$$

Lo que implica: $A$ y $B$ son tipos equivalente $A \equiv B$ o invariantes.

4. **(1,5pts)**En clases, vimos que es posible dar una representación procedural de tipos de datos abstractos (como ambientes, colas, etc.). Es decir, es posible implementar esos tipos de datos como *funciones* (o conjuntos de funciones), y así proveer toda la interfaz del tipo de datos considerado sin usar `define-type`.

Tambien hemos visto como agregar estado a funciones, gracias al scope estático. Además, si usamos mutación, podemos permitir que una función tenga estado mutable. Recuerde que definiciones de funciones pueden estar anidadas.

Consideremos el tipo de dato abstracto *dict*, tal que representa una lista de pares de `keys` y `value`, con las operaciones de inserción, búsqueda y eliminación en el diccionario. Defina, usando la *representación procedural* de un *dict*, las funciones `create-dict`, `search-dict`, `insert-dict` y `delete-dict` tal que:

```
(define mydict (create-dict))
..
(insert-dict mydict 5 "Cinco")
(insert-dict mydict 3 "Tres")
(insert-dict mydict 7 "Siete")

(insert-dict mydict 5 "Five") --> Error!!

(test (search-dict mydict 3) "Tres")

(delete-dict mydict 7)

(test (search-dict mydict 7) #f)
```

Implementación en el archivo: `dict.ss`.