Examen – Lenguajes de Programación (CC41A)

Departamento de Ciencias de la Computación – Universidad de Chile

Profesor: Éric Tanter

21 de Noviembre 2007

(2 horas)

sin apuntes

1. **(1.5pt)** Considere la función `fold` que reduce una lista de elementos basado en:
   - una función de reducción `f`
   - un valor inicial
   - una lista de elementos de cualquier tipo

   Por ejemplo:

   ```
   (fold + 0 '(1 2 3)) ---> 6
   (fold + 2 '(1 2 3)) ---> 8
   ```

   a) ¿Cual tiene que ser la firma de la función de reducción? De un ejemplo de uso con una función `f` *anónima* definida por el programador.

   b) Sabiendo que el primer parametro de la función de reducción tiene que ser el elemento tomado de la lista a reducir, defina `fold` en Scheme.

   c) Use `fold` para definir `reverse`, la función que toma una lista y retorna la lista inversa:

   ```
   (reverse '(1 2 3)) ---> (3 2 1)
   ```

2. **(1.5pt)**

   a) ¿Por qué en un lenguaje con substitución directa (= no diferida) no es necesario introducir una noción de clausura para preservar el scope estático?

   b) ¿Por qué en un lenguaje con funciones de primer orden no es necesario introducir ambientes recursivos para soportar definiciones de funciones recursivas?

   c) Explique por qué tener scope estatico es importante para hacer uso de funciones de funciones genéricas como `fold` o `map`. Ilustre con un ejemplo que no sería posible realizar en un lenguaje que tiene solo funciones de primera clase con scope dinámico.

3. **(1.5pt)**

   a) ¿Que significa que un lenguaje tenga un régimen de evaluación peresozo? ¿Cómo (con que prueba) Ud. puede determinar cual es el régimen de evaluación de un lenguaje? Cite dos lenguajes de esta familia.

   b) ¿Que ventaja tiene un lenguaje peresozo al momento de componer programas/funciones? Ilustre.

   c) ¿Como se optimiza la evaluación peresoza en Haskell? ¿por qué esa optimización es válida (= no interfiere con la semántica de un programa)?

4. **(1.5pt)** Considere el siguiente programa:

```
{with {f {fun {x : num} : (num -> num)
            {fun {y : num} : num
              {+ x y}}}}
  {+ 3
     {f 5}}}
```

   a) Explique las anotaciones de tipo del programa.

   b) Escriba y explique los juicios de tipo de las siguientes expresiones: `with`, `fun` (definición), y `app`.

   c) Demuestre, desarrollando los juicios de tipos, que el programa anterior **no** es válido.

Pauta

1. **(1.5pt)**

   a) ¿Cual tiene que ser la firma de la función de reducción? De un ejemplo de uso con una función `f` *anónima* definida por el programador.

   - **(0.25 pt)** Función de reducción para fold

     ```
     f : a -> b -> a
     ```

   - **(0.25 pt)** Ejemplo de uso

     ```
     (fold (lambda (x y) (* x y)) 1 '(1 2 3)) -> 6
     ```

   b) Sabiendo que el primer parametro de la función de reducción tiene que ser el elemento tomado de la lista a reducir, defina `fold` en Scheme.

   - **(0.5 pt)** 0.1 patrón, 0.4 resto

     ```
     (define (fold f acum l)
             (cond
                  ((empty? l) acum)
                  (else (f (car l) (fold f acum (cdr l))))))
     ```

   c) Use `fold` para definir `reverse`, la función que toma una lista y retorna la lista inversa:

   ```
   (reverse '(1 2 3)) ---> (3 2 1)
   ```

   - **(0.5 pt)**

     ```
     (define (reverse l) (fold cons '() l))
     ```

   - Definición alternativa **(máx 0.4 pt)**

     ```
     (define (--reverse l)
        (foldl (lambda (x y) (append y (cons x '()))) '() l))
     ```

   - Sin usar fold **(0 pt)**

2. **(1.5pt)**

   a) ¿Por qué en un lenguaje con substitución directa (= no diferida) no es necesario introducir una noción de clausura para preservar el scope estático?

   - **(0.5 pt)** La substitucion se hace sobre la definición de la función, por lo tanto, es seguro que los identificadores substituidos tienen los valores correspondientes al momento de la definición de la función.

   b) ¿Por qué en un lenguaje con funciones de primer orden no es necesario introducir ambientes recursivos para soportar definiciones de funciones recursivas?

   - **(0.5 pt)** En un lenguaje de primer orden, las funciones viven en un espacio distinto a donde viven nuestro programa. Esto permite que el programa conoce a priori todas las definiciones de funciones, por tanto al ejecutar una función cualquiera dentro del

programa, conoce a priori la lista completa de funciones disponibiles. Luego puedo crear funciones que se llaman a sí mismo (ie. recursivas) sin problemas.

c) Explique por qué tener scope estatico es importante para hacer uso de funciones de funciones genéricas como `fold` o `map`. Ilustre con un ejemplo que no sería posible realizar en un lenguaje que tiene solo funciones de primera clase con scope dinámico.

- Respuesta 1 **(0.5 pt)** Con scope estatico se captura los valores del scope activo al momento de la definición:

  ```
  (define (accum y)
     (lambda (x) (+ x y))
  ```

  Esto no es factible con scope dinamico, ya que 'y'podria tener cualquier valor (e incluso ser indefinido) segun donde se aplica la lambda.

- Respuesta 2 **(0.5 pt)** Con scope estático estamos asegurando que las variables que son usadas en la definición de funciones tendrán sólo validez dentro de ese contexto. Considerando el sgte. ejemplo:

  ```
  (define (map f l)
       (cond
         ((empty? l) '())
         (cons (f (car l)) (map f (cdr l)))))
  (let (f 10)
  (map f '(1 2 3))
  -- Error trying to apply a number.
  ```

  En este caso, en un supuesto scope dinámico, deberíamos conocer el nombre de las variables que usa la definición de map para no pisar su valor.

3. **(1.5pt)**

a) ¿Que significa que un lenguaje tenga un régimen de evaluación peresozo? ¿Cómo (con que prueba) Ud. puede determinar cual es el régimen de evaluación de un lenguaje? Cite dos lenguajes de esta familia.

- **(0.1 pt)** El intérprete del lenguaje, no reduce expresiones (ie interpretándolas) hasta que es necesario, esto puede reducir (en algunos casos) un grupo de cálculos de expresiones que pudiesen ser no necesarias.
- **(0.2 pt)** Ejemplo

  ```
  ((lambda (x) 4) (car '())
  ```

  En un lenguaje lazy, esta expresión retornaría 4. Ya que nuestra función anónima, no hace utilización de su parámetro ('x), luego nunca interpreta el argumento (sólo lo guarda como expresión). En un lenguaje eager, este programa no ejecutaría dado dada la expresión (car '()), que será calculada, a pesar de que nunca será usada.

- **(0.2 pt)** Lenguajes: Haskell, Schell scripting

b) ¿Que ventaja tiene un lenguaje peresozo al momento de componer programas/funciones? Ilustre.

- **(0.4pt)** Básicamente, que el productor y el consumidor no tienen que coordinarse explicitamente sobre el numero de elementos que tiene que producir, sino que puede producir ínfinitos valores', ya que serán evaluados solos los que realmente necesita el consumidor.
- **(0.1pt)** Ej.

  ```
  take 10 ([20, 12, 8] ++ cycle [0])
  ```

c) ¿Como se optimiza la evaluación peresoza en Haskell? ¿por qué esa optimización es válida (= no interfiere con la semántica de un programa)?

- **(0.25 pt)** Utilizando caching, una vez reducida una expresión memorizamos este valor (cachéandolo) de manera de no tener que volver a calcularlo cada vez que se necesita.
- **(0.25 pt)** Esta optimización es válida ya que en Haskell no tenemos mutaciones, luego una expresión valdrá lo mismo dentro del programa. (ya que la componentes de la expresión no pueden cambiar una vez definidas)

4. **(1.5pt)** Considere el siguiente programa:

```
{with {f {fun {x : num} : (num -> num)
           {fun {y : num} : num
             {+ x y}}}}
 {+ 3
    {f 5}}}
```

a) Explique las anotaciones de tipo del programa.

- **(0.3 pt)**

  ```
  (fun (x : num) : (num -> num) ... )
  ```

  Esta función recibe como parámetro un número, y retorna una nueva función que toma como parámetro un numero y retorna otro.
- **(0.2 pt)**

  ```
  (fun (y : num) : num ... )
  ```

  Esta función recibe como parámetro un número y retorna otro número.

b) Escriba y explique los juicios de tipo de las siguientes expresiones: `with`, `fun` (definición), y `app`.

- **(0.15 pt)** fun

$$\frac{\Gamma\,[i \leftarrow \tau_1] \vdash b : \tau_2}{\Gamma \vdash \{fun\{i : \tau_1\} : \tau_2\ b\} : \{\tau_1 \rightarrow \tau_2\}}$$

- **(0.15 pt)** app

$$\frac{\Gamma \vdash f : (\tau_1 \to \tau_2) \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash \{f \;\; a\} : \tau_2}$$

- **(0.2 pt)** with

$$\frac{\Gamma \vdash var : \tau_a \qquad \Gamma \vdash body : \tau_b}{\Gamma \vdash \{with\{var : \tau_a\} \;\; body : \tau_b\} : \tau_b}$$

c) Demuestre, desarrollando los juicios de tipos, que el programa anterior **no** es válido.

- **(0.5pt)**

$$\frac{\varnothing \vdash f : (number \to (number \to number))}{\varnothing \vdash \{f \;\; 5\} : (number \to number)}$$

Luego, como:

$$\frac{\Gamma \vdash l : number \qquad \Gamma \vdash r : number}{\Gamma \vdash \{+ \;\; l \;\; r\} : number}$$

Pero con la información $\{\mathrm{f}\, 5\}$ no tiene tipo number, luego el programa no es correcto.