LTP 2B - L2 - Examen primer parcial - 4 de Noviembre de 2020

Alumno:

1 Cuestiones (40%)

1.1. (10%) Describe en qué consiste la *reflexión* en los lenguajes de programación. Enumera sus pros y sus contras. Da al menos dos ejemplos de lenguajes con reflexión.

1.2. (10%) Indica qué tipos de áreas de la memoria se distinguen en función de su uso en la ejecución de un programa y cómo se utiliza cada una de ellas.

1.3. (10%) Describe las etapas del proceso de compilación de un programa indicando brevemente qué problema resuelve cada una.

1.4. (10%) ¿Cómo se demuestra la corrección de un programa empleando el cálculo de la precondición más débil? Da un ejemplo de terna de Hoare que, independientemente del programa considerado, *siempre* permita demostrar su corrección.

2 Problemas (60%)

2.1. (20%) Las siguientes clases de Java describen clases que pueden utilizarse para representar la semántica operacional de paso pequeño del lenguaje SIMP. (...)

1. Señala las líneas donde se utiliza *sobrecarga* y *coerción* en el código. Justifica tus afirmaciones.

   Solución.

   - Respecto al uso de la *sobrecarga*, en las líneas 35, i.e.,

     ```
     35 public int eval(State s) { return s.vvalues[x]; }
     ```

     así como en la 76, i.e.,

     ```
     76 public Boolean eval(State s)
     ```

     se definen dos versiones distintas del método `eval`, con tipos de retorno distintos: `int` y `Boolean`, respectivamente.
   - En cuanto a la *coerción*, se utiliza implícitamente en la línea 50, i.e.,

     ```
     50 for (i=0; i<n; ++i) add = add + aexps[i].eval(s);)
     ```

     para poder sumar la variable `add`(de tipo `long`) y el valor devuelto por `aexps[i].eval(s)` (de tipo `int`). En la línea 51, i.e,

     ```
     51 return (int) (add / n);
     ```

     se realiza una coerción *explícita* para transformar el cociente entre `add` y `n` en un entero.

2. Señala las líneas donde se utiliza *genericidad* en el código. Justifica tus afirmaciones.

   Solución. En la línea 10, i.e.,

   ```
   10 abstract class Expression<V> {
   ```

   se inicia la definición de la clase genérica `Expression<V>` que contiene un método genérico `eval` en la línea 11, i.e.,

   ```
   11 abstract public V eval(State s);
   ```

   La clase `Expression<V>` se particulariza en dos clases, `ArithExpression`, cuando `V` se asocia a la clase `Integer`, línea 21, i.e.,

   ```
   21 extends Expression<Integer> {
   ```

   y `BoolExpression`, cuando `V` se asocia a la clase `Boolean`, línea 61, i.e.,

   ```
   61 extends Expression<Boolean> {
   ```

   que proporcionan implementaciones (sobrecargadas) del método `eval`.

3. ¿En las líneas 2 y 3, podría reemplazarse `public` por `protected`? ¿Y por `private`? Justifica cada respuesta.

   Solución.

   - En la línea 2 de la definición de la clase `State`, i.e.,

     ```
     2 public int vvalues[];
     ```

     puede reemplazarse `public` por `protected` porque este modificador permite que los identificadores sean visibles dentro del mismo *package* de Java. Sin embargo, no puede reemplazarse por `private` porque entonces la variable `vvalues` no sería visible en la línea 35, i.e.,

     ```
     35 public int eval(State s) { return s.vvalues[x]; }
     ```

     que forma parte del código de la clase `VariableExp`.

   - En la línea 3, i.e.,

     ```
     3 public int nvars;
     ```

     puede reemplazarse `public` tanto por `protected` como por `private` porque `nvars` solo es accedida en la línea 6, i.e.,

     ```
     6 nvars = size; vvalues = new int[size];
     ```

     que se encuentra dentro del código de la clase `State` donde se declara el atributo.

4. Las clases `ArithmeticExp` y `BooleanExp` no declaran atributos ni métodos, pero a pesar de todo son abstractas. ¿Podría eliminarse el modificador `abstract`? Justifica tu respuesta.

   Solución. No sería posible, porque tanto `ArithmeticExp` como `BooleanExp` heredan el método abstracto `eval` de `Expression`. Como no proporcionan una implementación para éste, deben seguir siendo abstractas.

2.2. (10%) Considera la siguiente clase `Main`: (...)

- ¿Hay algún tipo de uso de la *sobrecarga*? ¿Y de la *coerción*? ¿Dónde?

  Solución.

  – Con relación a la *sobrecarga*, en las líneas 114 y 115, i.e.,

  ```
  114 System.out.print(str + ".  Hence,  (x <= y) is ");
  115 System.out.print(b.eval(s));
  ```

  se utiliza el método `print` para mostrar cadenas de caracteres (`String`) y booleanos (`Boolean`), respectivamente.

  – Con relación a la *coerción*, en la línea 112, i.e.,

  ```
  112 "x = " + s.vvalues[0] + " and y = " + s.vvalues[1];
  ```

  se utiliza coerción implícita de `s.vvalues[0]`, de tipo `int`, a `string` para poder utilizar el operador de concatenación de cadenas de caracteres.

- En la línea 106, ¿podría utilizarse el constructor de `ArithmeticExp` en lugar del de `VariableExp` para asignar el objeto a la variable `a1`, de tipo `ArithmeticExp`? ¿Podría la variable `leqBexp` de la línea 108 declararse de clase `LeqExp` en vez de `BooleanExp`? ¿Podría la variable `b` de la línea 109 declararse también de clase `LeqExp`?

  Solución.

  – En la línea 106, i.e.,

  ```
  106 ArithmeticExp a1 = new VariableExp(0);
  ```

  no se puede utilizar el constructor de la clase `ArithmeticExp` para obtener un objeto porque es una clase abstracta.

  – En la línea 108, i.e.,

  ```
  108 BooleanExp leqBexp = new LeqExp(a1,a2);
  ```

  la variable `leqBexp` puede declararse de clase `LeqExp` sin problemas, es decir, podemos escribir

  ```
  108' LeqExp leqBexp = new LeqExp(a1,a2);
  ```

  porque lo único que se hace con ella es utilizarla como parámetro en el constructor de `NotExp` en la línea 109, i.e.,

  ```
  109 BooleanExp b = new NotExp(leqBexp);
  ```

  cuyo tipo es `BooleanExp`, superclase de `LeqExp`.

  – En la línea 109 /ver arriba), la variable `b` *no* puede declararse de clase `LeqExp` puesto que el objeto construido es de la clase `NotExp`, que no es subclase de `LeqExp`.

- ¿En qué parte de la memoria se alojan los *objetos* referenciados por las variables `s` y `b` (líneas 102 y 109)? ¿Y las *variables* `s` y `b`?

  Solución. La línea 109 se muestra más arriba y la línea 102 es:

  ```
  102 State s = new State(2);
  ```

  Los objetos se alojan en el *heap*, mientras que las variables, como son locales al método `main`, se alojan en la pila.

2.3. (20%) El lenguaje de programación C permite instrucciones como x `<<=`d (donde x es una variable y d es una expresión aritmética) que asigna a la variable x el resultado de *desplazar* su valor n posiciones donde n es el resultado de evaluar la expresión d. Teniendo en cuenta que cada desplazamiento binario es equivalente a multiplicar por 2, se pide:

1. Define la semántica operacional de paso pequeño para esta nueva instrucción por *traducción* de la misma a las instrucciones ya existentes en SIMP.

2. Realizar los cambios necesarios en la semántica de paso pequeño y paso grande (*small-step*/*big-step*) de SIMP para dar semántica a la nueva instrucción de forma *directa*(no por traducción).

3. Mostrar la traza de ejecución completa para el siguiente programa:

```
n := 1;
n <<= n;
while (n>1) do n := n-1
```

2.4. (**10%**) Escoger una de las siguientes:

1. Modificar la definición de *pmd* para poder utilizar también la instrucción x `<<=` d definida en el apartado anterior.

2. ¿Es correcta la siguiente terna de Hoare? Justifica tu respuesta indicando los cálculos necesarios para obtenerla.

```
{ x > 0 }

  if x < y then z := x else z := y
  y := x+1;

{ 2z = x+y }
```