UNIVERSIDAD DE MÁLAGA

E.T.S.I. INFORMÁTICA

Lenguajes de Programación

4.º del Grado en Ingeniería Informática (Computación)

1 de febrero de 2021

APELLIDOS, NOMBRE: ______________________________

- Para la evaluación del examen se valorarán la **corrección** y la **claridad** de las soluciones.
- Solo se modifican y entregan los ficheros `AexpSOS.hs`, `NaturalSemantics.hs` y `StructuralSemantics.hs`.
- Las definiciones semánticas y la implementación Haskell deben hacerse en las **secciones** señaladas mediante **comentarios** en el fichero correspondiente. No modifiques el resto del código suministrado.
- En las implementaciones Haskell debes utilizar una ecuación por cada axioma y regla de la semántica.
- La semántica de las nuevas construcciones debe definirse de manera primitiva, sin azúcar sintáctico.

Problema 1. (*1.0 + 1.0 + 0.5 ptos.*) Una ventaja de la semántica estructural operacional es que nos permite definir con precisión el **orden de evaluación** de las expresiones aritméticas. Por ejemplo, podemos especificar que las expresiones se evalúan **de izquierda a derecha**. Un operador binario reduce paso a paso primero su operando izquierdo hasta su valor o forma normal; después reduce su operando derecho y, finalmente, cuando ambos operandos son valores, se aplica el operador.

Por ejemplo, suponiendo que $x = 1$, $y = 2$ y $z = 4$, la expresión $(x + y) * (7 - z)$ se evalúa de la siguiente manera:

```
(x + y) * (9 - z) =>
(1 + y) * (9 - z) =>
(1 + 2) * (9 - z) =>
3 * (9 - z)       =>
3 * (9 - 4)       =>
3 * 5             =>
15
```

Consideraremos el lenguaje de expresiones aritméticas **Aexp** extendido con el operador de división entera a_1/a_2, que devuelve el cociente entero de dividir a_1 entre a_2. Para especificar la semántica utilizaremos dos tipos de configuraciones Γ:

- $\langle a, s\rangle$ donde $a \in$ **Aexp**, $s \in$ **State** denota una configuración intermedia, y
- z donde $z \in \mathbb{Z}$ denota una configuración final.

Supondremos definidas las funciones semánticas $\mathcal{N}$, que dado un literal entero devuelve su valor, así como su inversa, $\mathcal{N}^{-1}$, que dado un entero devuelve su literal numérico.

a. Define en el fichero `AexpSOS.hs` la semántica estructural operacional de **Aexp**. Solo es necesario que definas las reglas para los literales enteros, variables, adición y división.

b. Implementa en el fichero `AexpSOS.hs` la semántica definida en el apartado anterior. Tienes que implementar la semántica para todo **Aexp**, incluyendo producto y sustracción. Utiliza el tipo `AexpConfig` para representar las configuraciones Γ.

c. Implementa en el fichero `AexpSOS.hs` la función `aExpDerivSeq :: Aexp ->State ->AexpDerivSeq` que dadas una expresión aritmética y un estado devuelve la secuencia de derivación correspondiente.

Problema 2. (*1.75 + 1.75 ptos.*) Tomaremos como base una versión simplificada —sin bucles— del lenguaje WHILE. El fichero `While21.hs` contiene los tipos algebraicos para representar la sintaxis abstracta, así como las funciones semánticas `aVal` y `bVal` para evaluar expresiones aritméticas (**Aexp**) y Booleanas (**Bexp**), respectivamente. El fichero `NaturalSemantics.hs` contiene la definición de la semántica natural de WHILE. El fichero `StructuralSemantics.hs` contiene la definición de la semántica estructural operacional de WHILE.

Añade al lenguaje WHILE la sentencia condicional múltiple `case`. La sintaxis de la sentencia `case` es:

$$
\begin{array}{rcl}
S & ::= & \texttt{case}\ a\ \texttt{of}\ LC\ \texttt{end} \\
LC & ::= & LL : S\ LC \\
 & | & LL : S \\
 & | & \texttt{default} : S \\
LL & ::= & n, LL \\
 & | & n
\end{array}
$$

donde a es una expresión aritmética, LC una lista de casos, LL una lista de etiquetas y n un literal entero. Observa que un caso no es más que una sentencia S precedida por una lista no vacía de etiquetas. Las listas de etiquetas son secuencias de literales enteros separados por comas. Existe además una etiqueta especial, `default`, que solo puede aparecer como caso final.

Intuitivamente, la semántica del `case` se define del siguiente modo; para ejecutar la sentencia:

```
case a of
   n11, n12 ... n1i : S1
   ...
   mj1, mj2 ... mjk : Sj
   default : Sd            // opcional
end
```

se evalúa la expresión aritmética a, y se va comparando el resultado obtenido con las etiquetas enteras n_{ij} en el **orden de aparición** (de arriba a abajo y de izquierda a derecha):

- Si encontramos una etiqueta n_{ij} cuyo valor coincida con el de a, se ejecuta la sentencia S_i correspondiente y se ignora el resto de casos.
- Si ninguna etiqueta n_{ij} coincide con el valor de a y al final aparece un caso `default`, se ejecuta la sentencia S_d.
- Si ninguna etiqueta n_{ij} coincide con el valor de a y no aparece un caso `default`, se aborta la ejecución del programa.

Las etiquetas n_{ij} no tienen que ser consecutivas, pueden aparecer repetidas y no tienen que aparecer ordenadas.

a. Define e implementa en el fichero `NaturalSemantics.hs` la semántica natural de la sentencia `case`. Dado que la semántica natural no puede gestionar la terminación abrupta, la implementación debe elevar una excepción mediante la función `error`.

b. Define e implementa en el fichero `OperationalSemantics.hs` la semántica estructural operacional de la sentencia `case`. Recuerda que la semántica estructural sí puede gestionar la terminación abrupta.

Los ficheros `TestCase.w` y `TestNestedCase.w` contienen ejemplos de programas WHILE que utilizan la sentencia `case` para probar tu implementación.

Problema 3. (*1.0 + 1.0 ptos.*) Define la función $a\{x \mapsto y\}$ que dada una expresión $a \in \mathbf{Aexp}$ reemplaza todas las apariciones de la variable x por la variable y. Por ejemplo:

- $(x+1)\{x \mapsto y\} = y+1$
- $(x+1)\{z \mapsto y\} = x+1$
- $((x+y) - (3*y) + (z+x))\{x \mapsto y\} = (y+y) - (3*y) + (z+y)$

Demuestra que $|FV(a\{\ x \mapsto y\})| \leq |FV(a)|$, donde $FV(a)$ es el conjunto de variables libres de a y $|A|$ es el cardinal del conjunto A. Solo es necesario demostrar los casos base y uno de los casos inductivos.

Problema 4. (*2.0 ptos.*) Tres desarrolladores de JetPains están discutiendo una refactorización para su afamado IDE. La refactorización consiste en extraer código común delante de una sentencia condicional. En concreto, están discutiendo si la sentencia:

```
if b then (S; S1) else (S; S2)
```

se puede refactorizar en:

```
S; if b then S1 else S2
```

preservando la semántica del programa; es decir, que las sentencias son semánticamente equivalentes. Sin embargo, nuestros desarrolladores no se ponen de acuerdo:

- Hadi piensa que la refactorización **nunca** tiene sentido porque nunca preserva la semántica
- Irina piensa que la refactorización solo tiene sentido **en algunos casos** porque en algunos casos preserva la semántica
- Trisha piensa que la refactorización **siempre** tiene sentido porque siempre preserva la semántica

Ayuda a nuestros desarrolladores: escoge una opción, enuncia rigurosamente el resultado a demostrar y demuéstralo formalmente. Puedes suponer que las sentencias `S`, `S1` y `S2` siempre terminan.