Tarea 0 Evaluador

Teoría de la Computación
Universidad ORT Uruguay

Marzo 2022

El objetivo de esta tarea es codificar[1] en Haskell un *lenguaje* consistente de fórmulas que combinan las habituales operaciones aritméticas sobre valores *enteros*. Las fórmulas permitirán usar *variables* cuyos valores están registrados en una *memoria* y que se pueden modificar, a través de asignaciones a la C.

Memoria: Una memoria es una *tabla* variable-valor. Estas tablas serán codificadas mediante listas de pares ordenados. Por ejemplo, $M = [(x, 1), (z, 45)]$ es la memoria que tiene dos variables x y z con los valores 1 y 45 respectivamente.

Existen dos operaciones fundamentales sobre las memorias[2]: lectura y escritura. Para *leer una variable* x en una memoria M usaremos en este documento la notación Mx. Para sobreescribir o *actualizar* una memoria pondremos $M \prec +(x, n)$, con el efecto de que, luego de realizada esta operación, el valor de x en la memoria será n. Siguiendo con el ejemplo precedente, sea $M' = M \prec +(x, 3)$; luego $M'x = 3$. Igualmente, si $M'' = M \prec +(y, 14)$, entonces $M''y = 14$.

Expresiones: Las expresiones o fórmulas podrán ser *variables*, números (*literales*), *asignaciones*, o aplicaciones de alguna de las *operaciones aritméticas* comunes definidas para enteros. En particular, usaremos el símbolo := para la asignacón. Ejemplos de expresiones son, por lo tanto: $x := x+1$, $x := (a+3)*m$ y $z := (x := 1)$.

A continuación presentaremos una definición inductiva de las expresiones[3], en una notación que se explicará:

$$e ::= x \mid l \mid x := e \mid e \diamond e.$$

Aquí las letras x, l y e representan *tipos* de objetos sintácticos, usualmente llamados *categorías sintácticas* por ser usadas para establecer la *sintaxis*, es decir, las reglas de escritura del lenguaje. Específicamente, x, l y e representan cualquier *variable*, *literal entero* y *expresión*, respectivamente. Una variable será un *string*, un literal entero será una secuencia de dígitos decimales eventualmente

[1] Otro término técnico utilizado es *embeber*. En inglés se usan *to encode* y *to embed*.
[2] Dado que son tablas...
[3] Es decir, indicando cómo ellas se *construyen*.

con signo, y las expresiones e son definidas por la regla de arriba, de la siguiente manera:

El símbolo ::= se lee *"es una de las siguientes opciones"*, donde las opciones se separan una de otra por la barra vertical (*pipe*). Por lo tanto, en la regla estamos diciendo que una expresión es: o bien una variable, o bien un literal numérico, o bien una asignación, compuesta a su vez por una variable y una expresión, o bien la combinación de dos expresiones por un operador $\diamond$. Convenimos entonces que $\diamond \in \{+, -, *, \div, \%\}$.

Las expresiones son, entonces, o bien *atómicas* o bien *compuestas*, en el sentido de si contienen o no otras expresiones. Las atómicas son las variables y los literales. Las compuestas deben pensarse como árboles cuya raíz es el operador (sea de asignación u operador aritmético) y sus componentes del tipo que aparece en la opción correspondiente de la regla[4].

Evaluación: El propósito de esta tarea es codificar en Haskell expresiones como las definidas en el párrafo anterior y programar sus *reglas de evaluación*. Para especificar cómo las expresiones deben evaluarse, consideraremos la relación $\Downarrow$ (relación de evaluación) dada por las siguientes reglas, que se explican abajo:

$$\texttt{var}\ \frac{}{M, x \Downarrow M, Mx}\ x \text{ inicializada en } M \qquad \texttt{lit}\ \frac{}{M, l \Downarrow M, l}$$

$$\texttt{ass}\ \frac{M, e \Downarrow M', n}{M, x := e \Downarrow M' \prec + (x, n), n} \qquad \texttt{bin}\ \frac{M, e_1 \Downarrow M', m \qquad M', e_2 \Downarrow M'', n}{M, e_1 \diamond e_2 \Downarrow M'', m \triangle n}$$

con $\triangle \in \{(+), (-), (*)(\div), (\%)\}$[5] representando las operaciones matemáticas.

Léase por ejemplo la regla `var` como: dada una memoria M y una variable x, el resultado de evaluar x bajo M es la misma memoria (sin modificar) y el valor de x en M, es decir Mx.

Más generalmente entonces, la función de evaluación $\Downarrow$ es una relación binaria entre parejas memoria-expresión, por un lado, y memoria-valor entero, por el otro. Esto quiere decir que relaciona un par "memoria-expresión" (que puede pensarse como la *entrada*) con un par "memoria-entero". Dado que una expresión puede ser/contener una asignación, la memoria posiblemente sea modificada, luego una nueva debe retornarse, en general. Esto se conoce como *side effect* (efecto lateral) y es lo que sucede en lenguajes de pregramación como C, C++ y Java. Por ejemplo, dada la memoria $M = [(y, 2)]$, el resultado de evaluar $x := y + 3$ es el par $([(y, 2), (x, 5)], 5)$. Lo expresado en la oración anterior de manera alternativa se puede poner $([(y, 2)], x := y + 3) \Downarrow [(y, 2)(x, 5)], 5$.

Por otro lado, $\Downarrow$ es una *función parcial*, es decir, para cada par memoria expresión de entrada existe a lo sumo un par memoria-entero de salida. Existen muchos casos en los que el comportamiento no está definido o debería fallar: por ejemplo, si se trata de recuperar el valor de una variable que no está en

[4] Como se explicará luego en el curso, esto significa que estamos dando una sintaxis *abstracta* del lenguaje.

[5] Al colocarlos entre paréntesis, estos signos pasan a denotar las funciones sobre enteros correspondientes, es decir: suma, resta, producto, cociente y resto o módulo.

la memoria o si se divide por cero. En estos casos se debería interrumpir la ejecución del programa de evaluación usando la función especial del preludio `error`.

La regla `lit` es trivial: un número n evalúa en si mismo sin alterar la memoria[6].

Continuando con las reglas, la siguiente es `ass`. Ésta dice que para conocer el valor de una asignación $x := e$ bajo una memoria M, primero debemos evaluar e, lo que nos devolverá otra memoria M' (porque e podría contener otras asignaciones) y un valor n. Luego, si actualizamos M' con $x = n$ tendremos la memoria final; el valor final retornado seguirá siendo n[7].

Ejercicio 1 Analice la cuarta y última regla: ¿Qué puede decir acerca del orden de evaluación? ¿Importa?

Ejercicio 2 Codifique en Haskell el tipo de la memoria y programe las funciones `read`(Mx) y `write`$(M \prec + (x, r))$.

Ejercicio 3 Codifique en Haskell el tipo de las expresiones y programe la función de evaluación `eval`.

Casos de ensayo para el evaluador. Se especifica en cada caso la expresión e y la memoria m en que ella se evaluará, en el formato (m, e). El resultado correspondiente se puede determinar usando las reglas dadas arriba.

- $([], 13)$.
- $([], x)$.
- $([], (x := 100) + (x * x))$.
- $([], x := (y := 1))$.
- $([], (x := 0) + ((y := 1) \div x))$.
- $([(\texttt{x}, 1)], (x := 100) + (y := (x + x)))$.

[6]Aquí condundimos, a propósito con el fin de simplificar la explicación, literal con valor entero.

[7]Igual que en C...