Semántica formal de un lenguaje de programación

3 – Arreglos de enteros

Maximiliano Cristiá
Universidad Nacional de Rosario
Argentina

2019

Este documento describe el modelo matemático que formaliza la semántica de un lenguaje de programación muy simple. Es una extensión del lenguaje y modelo presentado en el vídeo `https://youtu.be/oiXaJ0ClUyE`.

La extensión consiste en agregar arreglos de enteros.

La gramática del lenguaje de programación

En la Figura 1 presentamos la gramática del lenguaje de programación en BNF. Sea $\mathbb{Z}$ el conjunto de los números enteros, $\mathbb{N}_1$ el conjunto de los números naturales sin el cero, Ord el conjunto de las variables ordinarias (no arreglos), Arr el conjunto de las variables que denotan arreglos y $Var = Ord \cup Arr$ con $Ord \cap Arr = \emptyset$. Los arreglos son de la forma $var[e]$ donde var es una variable del lenguaje y e es una $ExprE$. Los arreglos deben declararse al inicio del programa. Una declaración de arreglo es de la forma $var[n]$ donde var es una variable y n es una constante en $\mathbb{N}_1$.

La semántica formal del lenguaje de programación

La complejidad semántica al incluir arreglos en el lenguaje es que el modelo de memoria se torna más complejo. Hasta el momento una memoria era una función del tipo $Var \to \mathbb{Z}$. Como los arreglos ocupan varias posiciones de memoria pero hay solo una variable que los identifica ese modelo de memoria no es suficiente. Una memoria que contiene arreglos la podemos modelar con la siguiente función:

$$Var \times \mathbb{N}_1 \to \mathbb{Z} \tag{1}$$

donde el segundo parámetro indica una componente en el caso de que la variable sea un arreglo. En este modelo de memoria las variables ordinarias tienen una componente, mientras que los arreglos tienen tantas como la cantidad indicada

$$ExprE ::= \mathbb{Z} \mid Ord \mid Arr[ExprE] \mid ExprE + ExprE$$

$$ExprB ::= \mathsf{true} \mid \mathsf{false} \mid ExprE == ExprE \mid ExprE <= ExprE \mid \sim ExprB$$

$$Sentencia ::= Ord = ExprE \mid Arr[ExprE] = ExprE \mid Estructura$$

$$\begin{aligned} Estructura ::= \\ &\mathsf{if}\ ExprB\ \mathsf{then}\ Programa\ \mathsf{fi} \\ &\mid \mathsf{while}\ ExprB\ \mathsf{do}\ Programa\ \mathsf{done} \end{aligned}$$

$$Declaracion ::= \mathsf{declare}\ Arr[\mathbb{N}_1] \mid Declaracion\ ;\ Declaracion$$

$$Programa ::= DeclProg \mid Prog$$

$$DeclProg ::= Declaracion \mid Declaracion\ ;\ Prog$$

$$Prog ::= Sentencia \mid Prog\ ;\ Sentencia$$

Figure 1: Gramática del lenguaje de programación

en su declaración. Entonces si m es una memoria y $v \in Ord$, $m(v, 1)$ es el valor de v en m; y si $a \in arr$ e $i \in \mathbb{N}_1$ entonces $m(a, i)$ es el valor de la componente i de a en m, o sea el valor de $a[i]$ en m.

Si bien este modelo es bastante expresivo no permite modelar un comportamiento muy importante relacionado con los arreglos. Digamos que declaramos $a[20]$ y luego en algún lugar del programa tenemos $a[31]$. Claramente no existe la componente 31 del arreglo a por lo que $a[31]$ no debería tener un valor. Por otro lado si m es una memoria, el tipo (1) dice que $m(a, 31)$ debería tener un valor, pues la memoria es una *función total*. Este problema se soluciona definiendo una memoria con arreglos como una *función parcial*:

$$Var \times \mathbb{N}_1 \nrightarrow \mathbb{Z}$$

Es decir que esta función no necesariamente está definida para todo par (v, n). Entonces $a[31]$ tiene un valor en m sí y solo sí el par $(a, 31)$ pertenece al dominio de m; formalmente, sí y solo sí $(a, 31) \in \operatorname{dom} m$.

Con este nuevo modelo de memoria debemos modificar las definiciones de las funciones *eval*, *evalb* y *exec*. Empezamos por *eval*. La primera cuestión a tener en cuenta es que es posible que no se pueda evaluar una expresión entera. Por ejemplo si declaramos $a[20]$, la expresión $a[31]+5$ no se puede evaluar porque no hay un valor definido para $a[31]$. El comportamiento habitual de un programa cuando no se puede evaluar una expresión es terminar abruptamente. Vamos a representar este comportamiento con el símbolo $\perp$. Entonces ahora *eval* puede retornar un $\mathbb{Z}$ o $\perp$:

$$eval : (Var \times \mathbb{N}_1 \nrightarrow \mathbb{Z}) \times ExprE \to \mathbb{Z} \cup \{\perp\}$$

Empezamos por definir la evaluación de expresiones de la forma $a[i]$, o sea de arreglos (tenemos que saber evaluar expresiones de esta forma para, por ejemplo, cuando queremos establecer el significado de una asignación de la forma $x = a[i+1]$).

$$m[\![a[i]]\!] = \begin{cases} \bot & \text{si } m[\![i]\!] = \bot \\ \bot & \text{si } m[\![i]\!] \neq \bot \wedge (a, m[\![i]\!]) \notin \operatorname{dom} m \\ m(a, m[\![i]\!]) & \text{si } m[\![i]\!] \neq \bot \wedge (a, m[\![i]\!]) \in \operatorname{dom} m \end{cases}$$

Es decir, la definición dice que para evaluar $a[i]$ primero hay que evaluar la *ExprE* que funge como índice del arreglo, o sea i. Pero como i es una expresión entera su evaluación podría fallar en cuyo caso la evaluación de $a[i]$ falla también. Un caso típico donde la evaluación de i falla es el siguiente: digamos que declaramos $b[10]$ y $a[20]$, entonces la evaluación de $a[b[12]]$ fallará porque $b[12]$ no está definido. La segunda regla dice que aunque la evaluación de i no falle podría dar un número que no corresponda a una componente del arreglo según su declaración, y en ese caso la evaluación debe fallar. Un caso típico de esta situación es el siguiente: si declaramos $a[20]$ y $m[\![i]\!] = 20$, entonces la evaluación de $a[i+2]$ en m fallará porque $(a, 22) \notin \operatorname{dom} m$, aunque la evaluación en m de la expresión $i+2$ no falle. Finalmente, la última regla dice que el valor de $a[i]$ en m es $m(a, m[\![i]\!])$; o sea que primero evaluamos el índice y luego vemos qué valor hay en esa componente.

Si intentamos sumar dos expresiones enteras una de las cuales falla, la suma debe fallar.

$$m[\![e_1 + e_2]\!] = \begin{cases} \bot & \text{si } m[\![e_1]\!] = \bot \\ \bot & \text{si } m[\![e_2]\!] = \bot \\ m[\![e_1]\!] + m[\![e_2]\!] & \text{en cualquier otro caso} \end{cases}$$

Las dos últimas reglas son las que ya teníamos solo que la evaluación de variables ordinarias toma un segundo parámetro que siempre vale 1.

$m[\![e]\!] = e$, si $e \in \mathbb{Z}$

$m[\![e]\!] = m(e, 1)$, si $e \in \mathit{Ord}$

En la Figura 2 redefinimos la función *evalb*. Como en muchos casos para evaluar expresiones booleanas primero hay que evaluar expresiones enteras y, como ya vimos, esto puede no ser posible, la evaluación de expresiones booleanas también puede fallar por lo que *evalb* puede retornar $\bot$.

Finalmente redefinimos la función *exec*. Tal como hicimos con *eval* y *evalb*, debemos considerar el caso en el cual el programa ejecuta una evaluación no definida (o sea una evaluación que falla). En este caso decimos que el programa retorna $\bot$ en lugar de retornar una memoria, lo que se interpreta como una terminación abrupta. La terminación abrupta significa que no se puede saber el valor de las variables del programa.

$$\mathit{exec} : (\mathit{Var} \times \mathbb{N}_1 \nrightarrow \mathbb{Z}) \times \mathit{Programa} \to (\mathit{Var} \times \mathbb{N}_1 \nrightarrow \mathbb{Z}) \cup \{\bot\}$$

$evalb : (Var \times \mathbb{N}_1 \nrightarrow \mathbb{Z}) \times ExprB \to \mathbb{B} \cup \{\bot\}$

$evalb(m, b) = b$, si $b \in \mathbb{B}$

$$evalb(m, e_1 == e_2) = \begin{cases} \bot & \text{si } m[\![e_1]\!] = \bot \text{ o } m[\![e_2]\!] = \bot \\ \mathsf{true} & \text{si } m[\![e_1]\!] = m[\![e_2]\!] \\ \mathsf{false} & \text{si } m[\![e_1]\!] \neq m[\![e_2]\!] \end{cases}$$

$$evalb(m, e_1 <= e_2) = \begin{cases} \bot & \text{si } m[\![e_1]\!] = \bot \text{ o } m[\![e_2]\!] = \bot \\ \mathsf{true} & \text{si } m[\![e_1]\!] \leq m[\![e_2]\!] \\ \mathsf{false} & \text{si } m[\![e_1]\!] > m[\![e_2]\!] \end{cases}$$

$$evalb(m, \sim e) = \begin{cases} \mathsf{false} & \text{si } e = \mathsf{true} \\ \mathsf{true} & \text{si } e = \mathsf{false} \\ \bot & \text{si } evalb(e, m) = \bot \\ evalb(m, \sim evalb(e, m)) & \text{en cualquier otro caso} \end{cases}$$

Figure 2: Definición de la función *evalb*

- Para las declaraciones de arreglos vamos a dar tres semánticas diferentes progresivamente más complejas pero a la vez más expresivas.

  En la primera asumimos que en la memoria inicial desde la que se ejecuta el programa ningún arreglo ha sido declarado. En consecuencia aumentamos el dominio de m según la declaración de cada arreglo e inicializamos las componentes a cero. Recordar que c es una constante en $\mathbb{N}_1$.

  $$m[\![\mathsf{declare}\ a[c]]\!] = m \cup ((\{a\} \times [1, c]) \times \{0\}) \tag{2}$$

  En la segunda semántica la memoria inicial podría contener la declaración de algún arreglo pero la eliminamos y la reemplazamos por la definición que usamos en el primer modelo semántico.

  $$m[\![\mathsf{declare}\ a[c]]\!] = ((\{a\} \lhd \operatorname{dom} m) \lhd m) \cup ((\{a\} \times [1, c]) \times \{0\}) \tag{3}$$

  La fórmula se explica de la siguiente forma. El dominio de m es una relación binaria. Entonces $\{a\} \lhd \operatorname{dom} m$ es la relación binaria incluida en m tal que la primera componente de todos sus pares ordenados es a. Es decir, $\{a\} \lhd \operatorname{dom} m$ denota el conjunto de pares de la forma (a, i) que están en el dominio de m. Finalmente $(\{a\} \lhd \operatorname{dom} m) \lhd m$ elimina de m todos esos pares. Por lo tanto, en $(\{a\} \lhd \operatorname{dom} m) \lhd m$ no existe ninguna componente de a.

  El tercer modelo semántico es el más complejo pero permite iniciar el programa desde una memoria donde se han declarado una o más componentes

de a. Esto es útil para conectar el programa con el entorno. Es decir, el entorno puede establecer los valores iniciales del arreglo a para luego ejecutar el programa. De todas formas, para mantener la consistencia con la declaración del arreglo, la fórmula semántica que utilizaremos garantiza que solo se usen las primeras c componentes que existan en m y que si hay menos de c se completen con ceros[1].

$$\begin{aligned} & m[\![\mathsf{declare}\ a[c]]\!] = \\ & \quad (((\{a\} \lhd \operatorname{dom} m) \lhdd m) \\ & \quad \cup \{i \in [1,c] \bullet ((a,i), \textbf{if}\ (a,i) \in \operatorname{dom} m\ \textbf{then}\ m(a,i)\ \textbf{else}\ 0)\} \end{aligned} \quad (4)$$

Es decir, al encontrar una declaración de arreglo se modifica la memoria de manera tal que haya exactamente c componentes en el arreglo a en las cuales se preservan los valores que había en esas celdas o se inicializan a cero en caso de que no las haya. O sea que estamos diciendo que en este lenguaje de programación los arreglos se inicializan automáticamente a cero si el entorno no los ha inicializado o lo ha hecho parcialmente. En otras palabras, la semántica preserva las primeras c componentes de a (si existen), elimina las restantes (si existen) e inicializa a cero las que no existieran.

- Para la asignación a variables ordinarias, $x \in \mathit{Ord}$.

$$m[\![x = expr]\!] = \begin{cases} \bot & \text{si } m[\![expr]\!] = \bot \\ m \oplus ((x,1), m[\![expr]\!]) & \text{en cualquier otro caso} \end{cases}$$

  Es decir que la asignación a una variable ordinaria falla cuando falla la evaluación de la expresión a la derecha de la asignación. O sea que en cuanto el programa trata de ejecutar una asignación cuya expresión equivale a $\bot$, el programa termina abruptamente.

- Para la asignación a arreglos, $a \in \mathit{Arr}$.

$$m[\![a[i] = e]\!] = \begin{cases} \bot & \text{si } m[\![e]\!] = \bot \\ \bot & \text{si } m[\![a[i]]\!] = \bot \\ m \oplus ((a, m[\![i]\!]), m[\![e]\!]) & \text{en cualquier otro caso} \end{cases}$$

  Notar que el segundo caso abarca todas las formas en que una componente de un arreglo no existe.

- Para la estructura if.

$$m[\![\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{fi}]\!] = \begin{cases} \bot & \text{si } m[\![e]\!] = \bot \\ m[\![P]\!] & \text{si } m[\![e]\!] = \mathsf{true} \\ m & \text{si } m[\![e]\!] = \mathsf{false} \end{cases}$$

[1] El 'if' usando en la fórmula no es la estructura condicional del lenguaje de programación. Es una forma habitual de escribir una función cuyo resultado depende de una condición.

Claramente tenemos un nuevo caso para la ejecución de una estructura if puesto que la evaluación de su condición (e) puede fallar, en cuyo caso el programa se detiene abruptamente.

- Para la estructura while:

$$m[\![\mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done}]\!] = \begin{cases} \bot & \text{si } m[\![e]\!] = \bot \\ m[\![P\ ;\ \mathsf{while}\ e\ \mathsf{do}\ P\ \mathsf{done}]\!] & \text{si } m[\![e]\!] = \mathsf{true} \\ m & \text{si } m[\![e]\!] = \mathsf{false} \end{cases}$$

- Para la concatenación de sentencias:

$$m[\![P\ ;\ Q]\!] = \begin{cases} \bot & \text{si } m[\![P]\!] = \bot \\ m[\![P]\!][\![Q]\!] & \text{en cualquier otro caso} \end{cases}$$

Claramente un programa se detendrá abruptamente en cuanto una de sus sentencias falle.

Ejercicios

1. En el modelo semántico asumimos que en el dominio de la memoria inicial están todos los pares $(v, 1)$ donde $v \in Ord$. Extienda la gramática y la semántica de forma tal que se pueda iniciar la ejecución desde una memoria vacía.

2. Modifique la semántica de manera de no poner un valor fijo en el **else** de la fórmula (4).

3. Extienda la gramática y la semántica con una asignación que permita asignar todas las componentes de un arreglo al mismo tiempo.

4. Extienda la gramática y la semántica con los operadores div y mod de la aritmética entera.

5. Modifique la gramática y la semántica reemplazando $\mathbb{Z}$ por el intervalo $[\mathsf{minint}, \mathsf{maxint}]$. Considere una semántica donde la suma es modular y otra donde el desborde en una suma conduce a una falla.