χ

Modelo funcional de computación
Semántica operacional
Marzo de 2022

El concepto de semántica operacional. Además de escribirla correctamente, uno debe saber darle sentido a cada expresión que lo tenga. Sentido, significado y, técnicamente hablando, *semántica* son (al menos aproximadamente) equivalentes. En nuestro caso podemos decir genéricamente que *una expresión es algo que puede ser evaluado* —ése es el sentido general que venimos dando a las expresiones desde un comienzo. Por lo tanto, para entender una expresión es preciso saber cómo evaluarla. Y, consecuentemente, una manera de establecer la semántica del lenguaje, es decir el significado de sus expresiones, pasa por indicar cómo evaluar dichas expresiones. Esta forma específica de proporcionar sentido o explicar un lenguaje se denomina *semántica operacional*. Existen otras variedades de teorías semánticas que quedan fuera del alcance de este curso. Una ventaja de la semántica operacional es que proporciona una especificación lo suficientemente precisa como para servir de base a la implementación de compiladores o intérpretes del lenguaje en cuestión.

Valores. Como ya discutimos en la introducción, nuestro real punto de partida son los *valores*. Un resumen de cómo funciona el lenguaje es el siguiente: Los constructores de valores c internos al lenguaje son los formadores más básicos de valores; sobre éstos están destinadas a operar las abstracciones funcionales (expresiones lambda) por medio de las expresiones `case`. La clase inicial de valores se forma entonces aplicando cualquier constructor de valores c a valores ya formados previamente. Esta definición es recursiva, puesto que decimos que un valor se forma aplicando un constructor a otros valores *previamente* existentes. Más precisamente, se trata de una definición inductiva, cuyo caso base está dado por cada constructor de valores tomado por sí mismo, sin argumentos.

Por ejemplo, en el lenguaje de fundamentos se podían introducir constructores en el lenguaje de la siguiente manera:

`data` *Nota* `where` $\{$ *Do*; *DoS*; *Re*; *ReS*; *Mi*; *Fa*; *FaS*; *Sol*; *SolS*; *La*; *LaS*; *Si* $\}$.

$$\texttt{data}\ \mathit{Natural}\ \texttt{where}\ \{\ \begin{array}{lcl} \mathit{Zero} & : & \mathit{Natural}; \\ \mathit{Suc} & : & \mathit{Natural} \to \mathit{Natural}\} \end{array}$$

Aquí los constructores introducidos, denotados anteriormente en su conjunto por la metavariable c, son: *Suc*, *Zero*, *Do*, *DoS*, etc. La definiciones de tipo anteriores además de introducir los constructores también nos especifican la firma de los constructores, y que éstos generan únicamente valores del tipo *Nota* o *Natural* respectivamente. Lo que significa por ejemplo que *Do* es un valor de tipo *Nota*, que *Suc Zero* es otro valor pero de tipo *Natural*, mientras que *Do DoS*, *Suc DoS*, *Suc Suc* no son valores válidos. Como en χ no tenemos noción de tipo, las aplicaciones de los constructores son en cambio irrestrictas, admitiendo cualquier número de argumentos. En particular los valores anteriormente

inválidos son aceptados como valores posibles en χ.

También consideraremos valores a las abstracciones funcionales. Es decir, cada expresión $\lambda\overline{x}.e$ que expresa una transformación será un valor. En esto seguimos la norma de los lenguajes funcionales como Haskell: uno podría pretender que $\lambda\overline{x}.e$ fuera un valor sólo si el cuerpo e estuviera totalmente evaluado, pero se obtiene una teoría más sencilla y no menos útil si evitamos evaluar los cuerpos de las abstracciones. Sobre este detalle volveremos más adelante.

Las consideraciones anteriores nos dan la siguiente definición de los valores:
$v ::= \lambda\overline{x}.e \mid c\ \overline{v}$
Ésta *parece* notación de sintaxis concreta, pero cumplimos con indicar expresamente que vamos a interpretarla como sintaxis abstracta.

<u>*La relación de evaluación*</u>. Procedemos ahora a dar las *reglas* de evaluación de cada expresión. El método para estipular las reglas de evaluación es definir una relación binaria entre expresiones y valores, que llamaremos $\Downarrow$. O sea, $e \Downarrow v$ significa que la expresión e *evalúa* a v o que *tiene* valor v. La definición será dada a través de reglas de inferencia, de la forma

$$\textit{nombre regla}\ \frac{P_1\ P_2\ \ldots\ P_n}{C,}\ \textit{condiciones}$$

donde las afirmaciones P_i son las *premisas*, C es la *conclusión* y las condiciones son predicados que se deben cumplir para que la regla pueda aplicarse. El caso $n = 0$ corresponde a lo que normalmente es llamado un *axioma*.

Usaremos la siguiente sintaxis abstracta para las expresiones del lenguaje —en notación similar a la usada recién para valores :

$$e ::= x \mid c \mid \lambda\overline{x}.e \mid e\,\overline{e} \mid \texttt{case}\ e\ \texttt{of}\ \{\ \overline{b}\ \} \mid \texttt{rec}\ x.\ e$$

$$b ::= c\ \overline{x}\ e$$

El primer caso es el de una variable, pero no necesitamos considerarlo. En efecto, las variables aparecen solamente dentro de una abstracción funcional, y hemos decidido no efectuar evaluación allí. Ésta es la primera simplificación que se obtiene al tomar las abstracciones como valores.

El segundo caso es el de un constructor de valores, consistente en un identificador, digamos c. Claramente se trata del caso más simple de valor y se tiene entonces:

$$\text{cte}\ \frac{}{c \Downarrow c[\,]}$$

El tercer caso es el de la abstracción funcional, y hemos dicho que cada una de ellas es un valor, de modo que otra vez tenemos un axioma simple:

$$\text{abs}\ \frac{}{\lambda\overline{x}.e \Downarrow \lambda\overline{x}.e}$$

Para evaluar una aplicación $e\,\overline{e}$ uno debe comenzar por evaluar la función que está siendo aplicada, es decir e. Naturalmente hay dos resultados aceptables, si es que la evaluación tiene éxito: o una abstracción funcional (expresión lambda) o un valor formado por un constructor de valores aplicado a otros valores. Consideremos el primero de estos casos:

$$\text{ap-abs}\ \frac{e \Downarrow \lambda\overline{x}.e' \qquad \overline{e} \Downarrow \overline{v} \qquad e'[\overline{x} := \overline{v}] \Downarrow v}{e\,\overline{e} \Downarrow v}\ \#\overline{x} = \#\overline{e}$$

Si e da como resultado una abstracción funcional $\lambda\overline{x}.e'$ entonces estamos en condiciones de aplicar la conocida regla β, es decir, sustituiremos los parámetros $\overline{x}$ por los argumentos correspondientes. En esta versión del lenguaje, hemos escogido pasar esos argumentos *ya evaluados*, lo cual está explicitado en la seugnda premisa. Finalmente, en la tercera premisa, se toma el cuerpo e' y allí se sustituye la lista de parámetros $\overline{x}$ de la abstracción funcional por la lista de argumentos efectivos ya evaluados $\overline{v}$ en relación de uno a uno. Si la expresión resultante evalúa a v, éste será el valor de la aplicación original, tal como lo expresa la conclusión.

Es importante observar en este caso que la evaluación no es, como en Haskell, perezosa, sino "ansiosa" (inglés: *eager*): *todos* los argumentos $\overline{e}$ son evaluados previamente a su uso en la evaluación de la aplicación. Otro aspecto a ser analizado es la condición establecida para esta regla. La misma habla del *tamaño* de la lista de variables $\overline{x}$ y de la lista de argumentos $\overline{e}$: en la regla se establece que éstos deben ser iguales. De esta manera no incluiremos casos en los que se declaren más variables que argumentos ni más argumentos que variables de la abstracción. Cuando la condición para esta regla no se cumpla simplemente se devolverá un mensaje de error indicando que la expresión no puede ser reducida. Es importante notar que este error no es posible de identificar antes de evaluar la expresión, ya que necesitamos reducir e para saber si es una abstracción. Lo anterior pone este problema dentro de la clasificación de aquellos que se pueden identificar sólo en tiempo de ejecución (o de evaluación).

El otro caso de la aplicación es que la evaluación de la función e_1 resulte en un constructor aplicado a valores:

$$\text{ap-cte}\ \frac{e \Downarrow c\ \overline{v_1} \qquad \overline{e} \Downarrow \overline{v_2}}{e\,\overline{e} \Downarrow c(\overline{v_1}\ \texttt{++}\ \overline{v_2})}$$

En este caso los argumentos efectivos $\overline{e}$ se evalúan (nuevamente en versión ansiosa) y simplemente se agregan a los que ya traía el constructor c.

Lo siguiente a considerar serán las expresiones `case`. La expresión a evaluar será de la forma `case` e `of` $\overline{b}$ donde e es la *expresión principal* o *discriminante* del `case` y $\overline{b}$ es la lista de ramas. Debemos primeramente evaluar el discriminante, y el resultado debe ser de la forma $c\ \overline{v}$ para algún constructor de valores c. Más aún, este constructor debe encontrarse como cabeza de alguna rama dentro de la lista $\overline{b}$, acompañado por una lista de variables y una expresión f. El resultado final de evaluar el `case` se obtiene evaluando la expresión f habiendo

efectuado de ella la sustitución de las variables de la rama por los argumentos $\overline{v}$. En la regla aquí abajo usamos la notación $c \xrightarrow{\overline{b}} (\overline{x}, f)$ para afirmar que el constructor c se encuentra en la lista $\overline{b}$ asociado a la lista de variables $\overline{x}$ y a la expresión f:

$$\text{cse}\ \frac{e \Downarrow c\ \overline{v} \qquad c \xrightarrow{\overline{b}} (\overline{x}, f) \qquad f[\overline{x} := \overline{v}] \Downarrow v}{\texttt{case}\ e\ \texttt{of}\ \overline{b} \Downarrow v}\ \#\overline{x} = \#\overline{e}$$

Es importante notar que la condición dada para la aplicación de una abstracción con una lista de argumentos también aplica para esta regla.

Finalmente nos interesará reducir también la expresión que hace explícita la recursión, `rec` $x.\ e$. La regla que prometimos discutir mientras analizábamos la sintaxis es la siguiente:

$$\text{rec}\ \frac{e[x := (\texttt{rec}\ x.\ e)] \Downarrow v}{\texttt{rec}\ x.\ e \Downarrow v}$$

Es decir, se instancia el cuerpo de la definición recursiva en la propia definición recursiva.

Restan entonces dos construcciones auxiliares a realizar para completar la semántica operacional de χ. Son ellas:

1. La operación de sustitución y
2. la relación ternaria $\longrightarrow$ que busca un constructor en una lista de ramas y le asocia, en caso de encontrarlo, la expresión correspondiente

Vamos a dejar la sustitución como ejercicio para el lector, con este comentario fundamental: en un lenguaje donde existen operadores que ligan nombres (como en este caso, el constructor sintáctico λ) debe en general tenerse cuidado en evitar el fenómeno de *captura* de nombres (variables libres). Sin embargo, en nuestro lenguaje la captura no puede darse porque en una sustitución $e[x := e']$ nuestra expresión e' nunca contendrá variables libres. Esto se debe a que las expresiones a reducir nunca se encuentran dentro de un constructor λ y, por lo tanto, son siempre cerradas. Ésta es la segunda y fundamental simplificación que se consigue al no evaluar bajo el constructor de la abstracción funcional.

Pasemos entonces a la relación $\longrightarrow$. Se trata de otra versión de la búsqueda en una lista:

$$\frac{}{c \xrightarrow{(c\ ,\overline{x}\ ,\ f)\ :\ \_} (\overline{x}, f)}$$

$$\frac{c_1 \xrightarrow{bs} (\overline{x}, f)}{c_1 \xrightarrow{(c_2\ ,\_\ ,\_)\ :\ bs} (\overline{x}, f)}\ (c_1 \neq c_2)$$

Ejemplo. Recordamos del práctico anterior la macro not.
not $= \lambda b.$ `case` b `of` $\{$ *False* $\rightarrow$ *True*; *True* $\rightarrow$ *False*$\}$.

Mostramos a continuación el árbol de derivación de la evaluación para la expresión not *False*, separado en dos partes:

$$\text{cse}\ \frac{\text{cte}\ \dfrac{}{\mathit{False} \Downarrow \mathit{False}} \qquad \mathit{False} \overset{\mathit{False} \rightarrow \mathit{True};\ \mathit{True} \rightarrow \mathit{False}}{\longrightarrow} \mathit{True} \qquad \text{cte}\ \dfrac{}{\mathit{True} \Downarrow \mathit{True}}}{(*)\ \texttt{case}\ \mathit{False}\ \texttt{of}\ \{\ \mathit{False}\ \rightarrow\ \mathit{True};\ \mathit{True}\ \rightarrow\ \mathit{False}\} \Downarrow \mathit{True}}$$

$$\text{ap-abs}\ \frac{\text{abs}\ \dfrac{}{\lambda b.\ \texttt{case}\ b\ \ldots \Downarrow \lambda b.\ \texttt{case}\ \ldots} \qquad \text{cse}\ \dfrac{\vdots}{(*)\ (\texttt{case}\ \text{b}\ \ldots)\ [\text{b} := \mathit{False}] \Downarrow \mathit{True}}}{(\lambda b.\ \texttt{case}\ b\ \ldots)\ \mathit{False} \Downarrow \mathit{True}}$$