Lenguajes de Programación, 2022-1
Nota de clase 13: Semántica Formal

Karla Ramírez Pulido Manuel Soto Romero

15 de enero de 2022
Facultad de Ciencias UNAM

Como se mencionó en la Nota de Clase 1, por semántica nos referimos al significado que se le da a lo escrito. Por ejemplo, la expresión $2 + 3 \times 5$ se entiende como *multiplicar 3 con 5 y sumar* 2, lo cual da como resultado 17. En pocas palabras, es la interpretación (evaluación) de una expresión con la suposición de que hubo un análisis sintáctico previo que nos indica que las expresiones pertenecen al lenguaje correspondiente. Ejemplos de semántica que seguramente la persona lectora ha trabajado a lo largo de la licenciatura son:

- Interpretación de fórmulas proposicionales.
- Interpretación de fórmulas de la Lógica de Primer Orden.
- Definición de Intérpretes para Lenguajes de Programación.

Es natural preguntarse entonces ¿cómo indicamos la semántica de un programa? En general, podría hacerse mediante alguna de las siguientes formas:

- En lenguaje natural, explicando lo que debe hacerse para evaluar cada expresión:

  Para interpretar una expresión aritmética, sustituimos las variables por su valor correspondiente y aplicamos los operadores respectivos.

- Implementando un intérprete:

```
;; interp: ASA Env → ASA-Value
(define (interp expr env)
   (match expr
      [(id i) (lookup i env)]
      [(add i d) (numV (+ (numV-n (interp i env))
                          (numV-n (interp d env))))]
      ...))
```

  Escribir un intérprete para un lenguaje, nos obliga a entender su semántica, sin embargo, dependemos de la definición particular o del comportamiento esperado por él o los diseñadores.

Usar lenguaje natural quizá sea la forma más fácil de especificar la semántica de un lenguaje, sin embargo, no está exenta de ambigüedades y depende de la subjetividad de cada hablante. Por ejemplo ¿qué significa tener un valor? ¿cómo aplicamos los operadores?. Por otro lado, al escribir un intérprete, nos acoplamos en gran medida con el lenguaje de programación anfitrión y por ende a los detalles técnicos de la implementación, lo cual presenta un

problema pues la manera de desarrollar el interprete puede variar de lenguaje a lenguaje (aún siguiendo las mismas reglas), debido al estilo de programación o propósito con el cual fue definido el mismo.

Evidentemente necesitamos una mejor técnica para describir el comportamiento de un lenguaje que además sea sencillo de trasladar a un lenguaje de programación. Dicho en otras palabras, necesitamos un lenguaje universal. Este lenguaje universal y formal es la matemática.

Normalmente consideramos dos niveles de semántica:

- *Semántica dinámica:* Determina el valor o evaluación de un programa.
- *Semántica estática:* Determina cuando un programa está bien definido en términos sintácticos.

13.0.1. Semántica Dinámica

Existen tres estilos básicos para la definir la semántica dinámica de un lenguaje de programación.

Semántica Operacional. Define el comportamiento de un programa en términos de sus operaciones. Indica *cómo* evaluar las expresiones de un lenguaje. Ejemplificaremos y profundizaremos en este tipo de semántica secciones más adelante.

Semántica Denotativa. Asocia las expresiones con objetos matemáticos tales como números, conjuntos, funciones, etc. Por ejemplo, podemos usar esta semántica para *traducir* el significado de los números romanos, los símbolos $[\![\ ]\!]$ se pueden leer como *significa* o *denota*[1]*:*

- $[\![I]\!] = 1$
- $[\![II]\!] = [\![I]\!] + [\![I]\!] = 1 + 1 = 2$
- $[\![IV]\!] = [\![V]\!] - [\![I]\!] = 5 - 1 = 4$

En este caso el significado está dado en términos de conjuntos de números positivos.

Semántica Axiomática. Es un enfoque basado en la lógica para probar que un programa es correcto. Tiene fuertes relaciones con la Lógica de Hoare, esta lógica utiliza ternas para verificar un programa, dichas ternas tienen como objetivo probar que dada una precondición, al ejecutar un programa en específico se cumple una postcondición. Por ejemplo, la terna:

$$\{i := 1\}\,(\texttt{i:=i+1})\,\{i := 2\}$$

indica que si iniciamos con el estado $i := 1$ (precondición), al ejecutar la expresión `i:=i+1` (programa) se cumple que el estado cambia a $i := 2$ (postcondición), que en este caso es verdadero.

[1] En realidad la lectura de estos símbolos depende del contexto para el cuál se usa. Más adelante los usaremos para indicar el tipo de una expresión.

13.0.2. Semántica Estática

Dependiendo del lenguaje, un compilador o intérprete puede verificar ciertas propiedades semánticas en tiempo de compilación (estáticamente), por ejemplo:

- La existencia de presencias libres de variables.
- La correctud de los tipos de un programa.

Esta verificación es necesaria para poder definir la semántica de las instrucciones de manera simple y eficaz. Antes de definir el significado preciso de un programa, es necesario eliminar los programas sin sentido, por ejemplo, si el lenguaje tiene anotaciones de tipos explícitas, entonces su semántica dinámica estará definida si el programa en cuestión está bien formado con respecto a la semántica estática.

Esta semántica además determina qué estructuras de la sintaxis abstracta están bien formadas de acuerdo a ciertos criterios sensibles al contexto como la resolución de alcance al requerir que una variable sea declarada antes de usarse. Por lo general, la semántica estática consta de dos fases, aunque esto varía dependiendo del traductor.

1. La resolución del alcance de las variables.
2. La verificación de la correctud estática de un programa mediante la interacción con el sistema de tipos.

Profundizaremos en este tema más adelante en otra nota.

En estas notas estudiaremos únicamente la semántica operacional de un lenguaje. Si se desea profundizar en el estudio de los otros estilos, se recomienda llevar el curso optativo de Semántica y Verificación.

13.1. Semántica operacional para FAE

Partiendo de estas definiciones, queremos modelar la forma en que se evalúan las expresiones de nuestro lenguaje FAE estudiado con anterioridad. La manera en que hicimos este análisis fue discuitiendo los detalles y decisiones de diseño del mismo. En esta sección formalizaremos el comportamiento (semántica dinámica) del mismo usando el estilo de la semántica operacional.

Como decíamos, se requiere de una representación de los resultados de interpretación en un lenguaje matemático para que se comprenda de mejor manera y libre de ambigüedades el funcionamiento de éstos, para ello usaremos la semántica operacional que consiste en describir cómo se evalúan las expresiones del lenguaje. La representación usual en este estilo de semántica se da mediante reglas de inferencia.

Estudiaremos la definición de estas reglas contrastándolas con la implementación de nuestro intérprete.

13.1.1. Notación para las reglas

La evaluación de nuestras expresiones hasta este punto ha sido especificada por medio de la función `interp` de nuestro lenguaje. Recordemos la firma de esta función:

```
;; interp: FAE Env → FAE-Value
```

De esta forma, necesitamos una notación que modele tres cosas:

1. La expresión a evaluar.
2. El ambiente de evaluación.
3. El resultado después de evaluar.

La siguiente notación se adecúa a nuestras necesidades:

$$e, \mathscr{E} \Rightarrow v$$

Donde e es la expresión a evaluar, $\mathscr{E}$ es el ambiente de evaluación y v es el resultado después de evaluar. El símbolo $\Rightarrow$ se lee como "se reduce a" o "se evalúa a". Su lectura completa sería:

La expresión e bajo el ambiente de evaluación $\mathscr{E}$ se reduce a v.

Las expresiones a evaluar se suelen denotar mediante su sintaxis concreta o abstracta, en este caso usaremos la sintaxis concreta por comodidad. Los ambientes se denotan delimitando las parejas identificador-valor entre corchetes y mediante el símbolo de conjunto vacío cuando éste no tenga asignaciones:

- El ambiente `mtSub`: $\varnothing$
- El ambiente `(aSub 'a (numV 2) (aSub 'b (numV 3) (mtSub)))`: $[a \leftarrow 2, b \leftarrow 3]$. Notemos que no es necesario colocar la notación del ambiente vacío cuando haya al menos un elemento en el ambiente.

Finalmente, recordemos que el valor v, es el equivalente a nuestro `FAE-Value`:

```
(define-type FAE-Value
   [numV      (n number?)]
   [closureV (param symbol?) (body FAE?) (env Env?)])
```

De esta forma, denotaremos los valores numéricos (numerales) mediante $\widehat{n}$ y a las cerraduras mediante la tripleta $\langle param, body, \mathscr{E} \rangle$.

Observación 13.1. La notación que planteamos en esta sección es usualmente empleada para definir la semántica dinámica de un lenguaje, sin embargo puede tener variaciones en cuanto a los símbolos, estilos y tipo de semántica. Por ejemplo en el caso de la semántica estática (juicios de tipo) que describiremos más adelante en otra nota, se emplea la notación $\Gamma \vdash e : \tau$ que se lee como *Bajo el contexto de tipos Γ se deduce que el tipo de e es τ.*

13.1.2. Reglas de evaluación

Partiendo de la notación anterior, veamos cómo describir el comportamiento de nuestro intérprete mediante estos formalismos. La manera en la que lo haremos será contrastando nuestra implementación en RACKET caso por caso.

```
;; interp: FAE Env → FAE-Value
(define (interp expr env)
   (type-case FAE expr
      [id (i)
         (lookup i env)]
      [num (n)
         (numV n)]
      [(add lhs rhs)
         (numV (+ (numV-n (interp lhs env)) (numV-n (interp rhs env))))]
      [(sub lhs rhs)
         (numV (- (numV-n (interp lhs env)) (numV-n (interp rhs env))))]
      [(fun p b)
         (closureV p b env)]
      [(app fun-expr arg)
         (let ([fun-val (interp fun-expr)])
            (interp (closureV-body fun-val)
                    (aSub (closureV-param fun-val)
                          (interp arg)
                          (closureV-env fun-val))))]))
```

Identificadores

Recordemos que el ambiente sirve para recordar tanto las variables usadas por la función como el valor que tienen éstas en las expresiones, permitiendo así que en cualquier momento realicemos la búsqueda de cualquier identificador *i* justo como lo hacíamos con la función `lookup`.

Implementación	*Regla*	*Ejemplo*
`[id (i)` `  (lookup i env)]`	$\frac{}{i, \mathscr{E} \Rightarrow \mathscr{E}(i)}$	$\frac{}{foo, [foo \leftarrow \widehat{2}] \Rightarrow \widehat{2}}$

El ejemplo de la derecha se muestra un ambiente formado por el identificador *foo* y su valor *2*. El lado izquierdo de la reducción representa la búsqueda del identificador en el ambiente correspondiente.

Números

Cuando tenemos una expresión numérica como 1, esta debe reducirse a un valor numérico usando una representación adecuada para éstos en un lenguaje de programación, representamos esto como $\widehat{1}$. Generalizamos esto usando reglas de inferencia de la siguiente forma:

Implementación	*Regla*	*Ejemplo*
`[num (n)` `  (numV n)]`	$\dfrac{}{n, \mathscr{E} \Rightarrow \widehat{n}}$	$\dfrac{}{1, \varnothing \Rightarrow \widehat{1}}$

Donde $\widehat{n}$ representa al numeral de n que como mencionamos es una representación adecuada en un lenguaje de programación. Por ejemplo nuestro lenguaje regresa valores de tipo `numV`.

Operaciones aritméticas

Siguiendo este orden de ideas, ¿qué pasa con las operaciones como la suma o resta? Es natural pensar que la forma de hacer la reducción de la suma sería:

Regla	*Ejemplo*
$\dfrac{}{\{+\ x\ y\}, \mathscr{E} \Rightarrow \widehat{x + y}}$	$\dfrac{}{\{+\ 2\ 3\}, \varnothing \Rightarrow \widehat{5}}$

Sin embargo, al analizar la implementación:

```
[(add lhs rhs)
         (numV (+ (numV-n (interp lhs env)) (numV-n (interp rhs env))))]
```

se puede notar que la operación de suma, necesita que ambos lados de la misma sean números cosa que no sucede a menos que hayamos reducido estos de forma recursiva, como vimos en su momento cuando interpretamos expresiones aritméticas en AE. Por lo que podemos reescribir la regla de la siguiente manera:

Regla	*Ejemplo*
$\dfrac{x, \mathscr{E} \Rightarrow \widehat{x_v} \quad y, \mathscr{E} \Rightarrow \widehat{y_v}}{\{+\ x\ y\}, \mathscr{E} \Rightarrow \widehat{x_v + y_v}}$	$\dfrac{2, \varnothing \Rightarrow \widehat{2} \quad 3, \varnothing \Rightarrow \widehat{3}}{\{+\ 2\ 3\}, \varnothing \Rightarrow \widehat{2 + 3}}$

Recordemos que este tipo de reglas de inferencias que se vieron en el curso de Lógica Computacional, incluye en la parte de arriba al antecedente que representa a todas las condiciones que se deben cumplir y a las expresiones en la parte de abajo se les llama consecuentes ya que es el resultado o conclusiones que se cumplirán.

Funciones

En el caso de las funciones se necesita representar a las cerraduras (*closures*), las cuales se forman por sus parámetros $\{p\}$, su cuerpo b y el ambiente donde éstas fueron definidas $\mathscr{E}$ lo cual se representa por medio de las tripletas antes mencionadas de la siguiente manera:

Implementación	*Regla*	*Ejemplo*
`[(fun p b)` `  (closureV p b env)]`	$\overline{\{fun\ \{p\}\ b\}, \mathscr{E} \Rightarrow \langle p, b, \mathscr{E}\rangle}$	$\overline{\{fun\ \{x\}\ x\}, \varnothing \Rightarrow \langle x, b, \varnothing\rangle}$

En el ejemplo se muestra un caso en el que la función inicia con un ambiente particular, en este caso el ambiente vacío ($\varnothing$). Sin embargo, podemos mantener el símbolo $\mathscr{E}$ cuando no nos estemos refiriendo a un ambiente en particular.

Aplicaciones de función

Las aplicaciones de función, que se componen de una función y su argumento, deben extender el ambiente como hicimos en la función `interp`.

```
[(app fun-expr arg)
         (let ([fun-val (interp fun-expr)])
            (interp (closureV-body fun-val)
                    (aSub (closureV-param fun-val)
                          (interp arg)
                          (closureV-env fun-val))))]
```

Es decir, evaluamos la función, obteniendo una cerradura y a partir de esa cerradura, evaluamos el cuerpo de ésta en el ambiente formado por su parámetro formal y la evaluación del argumento (en caso de que sea glotón).

Regla	*Ejemplo*
$\dfrac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}'\rangle \quad a, \mathscr{E} \Rightarrow a_v \quad b, \mathscr{E}'[i \leftarrow a_v] \Rightarrow b_v}{\{f\ a\}, \mathscr{E} \Rightarrow b_v}$	$\dfrac{\{fun\ \{x\}\ x\}, \varnothing \Rightarrow \langle x, x, \varnothing\rangle \quad 2, \varnothing \Rightarrow 2 \quad x, [x \leftarrow 2] \Rightarrow 2}{\{\{fun\ \{x\}\ x\}\ 2\}, \varnothing \Rightarrow 2}$

Nótese que al reducir la función en el ambiente $\mathscr{E}$, se obtiene la tripleta $\langle i, b, \mathscr{E}\rangle$ donde el ambiente cambia ($\mathscr{E}'$), ¿por qué?, sucede porque una función se evalúa en su ambiente local, por lo que se necesita extender el ambiente $\mathscr{E}$ con lo que se encuentre en el ambiente local y a este ambiente se le llama $\mathscr{E}'$. Éste nuevo ambiente $\mathscr{E}'$ requiere agregar el parámetro como variable y su argumento como el valor, lo cual se representa con $\mathscr{E}'[i \leftarrow a_v]$[2].

13.1.3. Conclusiones

Llamamos a este tipo de semántica *semántica operacional de paso grande* o natural. Se le llama de esta forma porque las reducciones que representamos con $\Rightarrow$ reducen expresiones a su forma normal (es decir, ya no pueden ser reducidas). En contraste, una *semántica de paso pequeño o estructural* realiza una reducción a la vez, como si fuéramos ejecutando el programa paso a paso, sin embargo, su revisión queda fuera de los alcances de este curso. Adicionalmente decimos que es operacional porque únicamente describimos cómo se aplican las operaciones en el lenguaje, en este caso usando el lenguaje de las matemáticas. De acuerdo con [2], este tipo de semántica describe el significado de ejecutar un programa mediante la evaluación y reducción de expresiones a otras irreducibles de manera elegante, matemática y mecánica.

A continuación mostramos ejemplos de uso de las reglas definidas para FAE.

[2]Nótese que este juicio, representa la evaluación de la expresión usando alcance estático, ya que si el cuerpo de la función se evaluara con el ambiente $\mathscr{E}$,usaría alcance dinámico . Por lo que un simple cambio representa un cambio en la semántica de la expresión.

13.2. Ejemplos

Ejemplo 13.1. Evaluación de la expresión *x* en donde el ambiente incluye el identificador *x*.

$$\frac{}{x, [x \leftarrow 8] \Rightarrow 8}$$

□

Ejemplo 13.2. Evaluación de la expresión *x* en donde el ambiente no incluye el identificador *x*.

$$\frac{}{x, [y \leftarrow 8] \Rightarrow \texttt{error}}$$

□

Ejemplo 13.3. Evaluación de la expresión *1729* con el ambiente vacío.

$$\frac{}{1729, \varnothing \Rightarrow 1729}$$

□

Ejemplo 13.4. Evaluación de la expresión {+ 5 {+ 1 1}} con el ambiente vacío.

$$\frac{\dfrac{5, \varnothing \Rightarrow 5 \qquad \dfrac{1, \varnothing \Rightarrow 1 \qquad 1, \varnothing \Rightarrow 1}{\{+\ 1\ 1\}, \varnothing \Rightarrow 2}}{\{+\ 5\ \{+\ 1\ 1\}\}, \varnothing \Rightarrow 7}$$

□

Ejemplo 13.5. Evaluación de la expresión {fun {y} {+ y 2}} con el ambiente vacío.

$$\frac{}{\{fun\ \{y\}\ \{+\ y\ 2\}\}, \varnothing \Rightarrow \langle y, \{+\ y\ 2\}, \varnothing \rangle}$$

□

Ejemplo 13.6. Evaluación de la expresión {{fun {y} {+ y 2}} 3} con el ambiente vacío.

$$\frac{\{fun\ \{y\}\ \{+\ y\ 2\}\}, \varnothing \Rightarrow \langle y, \{+\ y\ 2\}, \varnothing \rangle \qquad 3, \varnothing \Rightarrow 3 \qquad \dfrac{y, [y \leftarrow 3] \Rightarrow 3 \qquad 2, [y \leftarrow 3] \Rightarrow 2}{\{+\ y\ 2\}, [y \leftarrow 3] \Rightarrow 5}}{\{\{fun\ \{y\}\ \{+\ y\ 2\}\}\ 3\}, \varnothing \Rightarrow 5}$$

□

Ejemplo 13.7. Evaluación de la expresión {{fun {z} {+ w z}} w} con el ambiente $[w \leftarrow 7]$.

$$\dfrac{\dfrac{\{fun\ \{z\}\ \{+\ w\ z\}\}, [w \leftarrow 7] \Rightarrow \langle z, \{+\ w\ z\}, [w \leftarrow 7]\rangle \qquad w, [w \leftarrow 7] \Rightarrow 7 \qquad \dfrac{w, [w \leftarrow 7, z \leftarrow 7] \Rightarrow 7 \qquad z, [w \leftarrow 7, z \leftarrow 7] \Rightarrow 7}{\{+\ w\ z\}, [w \leftarrow 7, z \leftarrow 7] \Rightarrow 14}}{\{\{fun\ \{z\}\ \{+\ w\ z\}\}\ w\}, [w \leftarrow 7] \Rightarrow 14}$$

□

Ejemplo 13.8. Regla de evaluación para la primitiva `with` de FAE:

$$\dfrac{v, \mathscr{E} \Rightarrow v_v \qquad b, \mathscr{E}[i \leftarrow v_v] \Rightarrow b_v}{\{with\ \{i\ v\}\ b\}, \mathscr{E} \Rightarrow b_v}$$

□

Ejemplo 13.9. Evaluación de la siguiente expresión con el ambiente vacío.

```
{with {f {fun {x} {- x x}}}
     {f 2}}
```

$$\dfrac{\dfrac{\{fun\ \{x\}\ \{-\ x\ x\}\}, \varnothing \Rightarrow \langle x, \{-\ x\ x\}, \varnothing\rangle \qquad \dfrac{f, [f \leftarrow \langle x, \{-\ x\ x\}, \varnothing\rangle] \Rightarrow \langle x, \{-\ x\ x\}, \varnothing\rangle \qquad 2, [f \leftarrow \langle x, \{-\ x\ x\}, \varnothing\rangle] \Rightarrow 2 \qquad \dfrac{x, [x \leftarrow 2] \Rightarrow 2 \qquad x, [x \leftarrow 2] \Rightarrow 2}{\{-\ x\ x\}, [x \leftarrow 2] \Rightarrow 0}}{\{f\ 2\}, [f \leftarrow \langle x, \{-\ x\ x\}, \varnothing\rangle] \Rightarrow 0}}{\{with\ \{f\ \{fun\ \{x\}\ \{-\ x\ x\}\}\}\ \{f\ 2\}\}, \varnothing \Rightarrow 0}$$

□

Se deja como ejercicio al lector la definición de las reglas de evaluación usando semántica operacional de paso grande para el lenguaje CFWAE.

Referencias

[1] Shriram Krishnamirthi, *Programming Languages Application and Interpretation*, Primera Edición, Brown University, 2007.

[2] Hanne Riis Nielson, Flemming Nielson, *Semantics with Applications: An Appetizer,* Primera Edición, Springer, 2007.

[3] Manuel Soto, *Notas de Clase de Semántica y Verificación*, Facultad de Ciencias UNAM, Revisión 2022-1.