Lenguajes de Programación, 2020-I[*]
Nota de clase 5, Introducción a la semántica de lenguajes de programación, el caso del lenguaje EAB[**]

Favio E. Miranda Perea Lourdes Del Carmen González Huesca
Facultad de Ciencias UNAM

27 de agosto de 2019

En esta nota damos una breve introducción a la semántica de los lenguajes de programación y desarrollamos una semántica estática y una semántica operacional para el lenguaje EAB de expresiones aritméticas y booleanas.

1. Introducción

La semántica de un lenguaje de programación consiste en dar significado a diversas características e instrucciones del lenguaje, por ejemplo, el comportamiento en tiempo de ejecución. Si bien la semántica de un lenguaje se especifica con frecuencia de manera informal (en manuales del lenguaje con ejemplos de código) hay buenas razones para desarrollar la semántica de manera formal. Los formalismos semánticos de lenguajes de programación se utilizan por ejemplo en:

- Implementación: especificación del comportamiento del lenguaje de manera independiente a una máquina en particular. Correctud y optimización de análisis de programas.
- Verificación: son base de los métodos de razonamiento acerca de propiedades y especificaciones de programas.
- Diseño de lenguajes: permiten descubrir ambigüedades y detalles imprevistos en constructores de lenguajes de programación. Las herramientas matemáticas utilizadas para semántica pueden sugerir nuevos estilos de programación, el ejemplo más prominente es la influencia del cálculo lambda (desarrollado en 1934) en los lenguajes de programación funcional.

Normalmente se consideran dos niveles de semántica:

- Semántica estática: determina cuando un programa está bien definido mediante criterios sintácticos.
- Semántica dinámica: determina el valor o evaluación de un programa.

A continuación describimos ambas clases de semántica.

[*]Agradecemos la colaboración de Susana Hahn Martín-Lunas en la revisión 2018-I

[**]Estas notas se basan en el libro *Types and Programming Languages* de Benjamin C. Pierce y en diversas versiones del libro de Robert Harper *Practical Foundations of Programming Languages.*

2. Semántica Dinámica

Existen tres estilos básicos para definir la semántica dinámica de un lenguaje de programación, a continuación los explicamos brevemente.

2.1. Semántica Denotativa

En este estilo los significados de modelan mediante objetos matemáticos, como números o funciones, que representan al efecto o resultado de ejecutar un constructor de programa. De esta manera el objeto de interés es el resultado y no el proceso de cómputo. Dar una semántica denotativa para un lenguaje consiste en encontrar una colección de dominios semánticos para después definir una función de interpretación que envia términos a elementos de dichos dominios. La busqueda de dominios apropiados para modelar diversas características del lenguaje ha dado lugar a una vasta y elegante área de investigación llamada teoría de dominios. Este estilo de semántica tiene un fuerte contenido matemático de tal forma que en sus inicios se llamó semántica matemática.

2.2. Semántica Axiomática

En este estilo las propiedades específicas de la ejecución de un programa se expresan por medio de aserciones de manera que ciertos aspectos de la ejecución se ignoran. Estas propiedades se definen mediante axiomas y reglas de una lógica que modela propiedades de programas (lógica de Hoare-Floyd). En tal caso el significado de un programa es simplemente lo que se puede probar acerca de él. La belleza de los métodos axiomaticos radica en que se enfocan en el proceso de razonamiento acerca de programas. La semántica axiomática ha producido ideas muy poderosas en ciencias de la computación tales como el concepto de *invariante*.

2.3. Semántica Operacional

En este estilo el significado de una instrucción de programa se especifica mediante el cómputo que induce en una máquina. En particular resulta de interés como se produce el resultado o efecto de un cómputo. El comportamiento del programa se modela definiendo una *máquina abstracta* en el sentido de que sus estados o código de máquina son entidades abstracta como las mismas expresiones del lenguaje. Para lenguajes simples un estado es simplemente una expresión y el comportamiento de la máquina se define mediante una relación o sistema de transición que devuelve el siguiente estado, el cual se obtiene al desarrollar una simplificación o evaluación o bien declarando que la máquina se ha detenido. El significado de la expresión e es entonces el estado final alcanzado por la máquina al iniciar su funcionamiento tomando a e como estado inicial. Esto se conoce como semántica estructural o de paso pequeño. En contraste existen las llamadas semánticas naturales o de paso grande que evaluán el término devolviendo el resultado final en un solo paso.

Con frecuencia es útil dar dos o más semánticas operacionales para un mismo lenguaje, algunas más abstractas con estados similares a las expresiones que usa el programador y otras más cercanas a las estructuras manipuladas por el intérprete o compilador en cuestión. Probar que ambas semánticas son equivalentes al ejecutar un mismo programa corresponde a probar la correctud de una implementación del lenguaje.

En nuestro curso nos dedicaremos exclusivamente a las semánticas operacionales.

2.4. Sistemas de Transición

Los sistemas de transición se utilizan para describir el comportamiento de la ejecución de programas definiendo un dispositivo de cómputo abstracto sobre un conjunto S de estados que se relacionan mediante una relación de transición entre estados, denotada $\rightarrow$. Los juicios de transición describen como cambia el estado de la máquina durante la ejecución.

Definición 1 *Un sistema de transición se especifica mediante los siguientes juicios:*

- s estado*, el cual afirma que* s *es un estado del sistema de transición.*
- s final*, el cual requiere que* s estado *y afirma que* s *es un estado final.*
- s inicial*, el cual requiere que* s estado *y afirma que* s *es un estado inicial.*
- $s \rightarrow s'$ *con* s estado, s' estado*, el cual afirma que* s *transita hacia* s'*.*

Además se requiere que si s final *entonces no exista* s' *con* $s \rightarrow s'$*. Es decir, no hay transiciones desde estados finales.*

Definición 2 *Un estado* s *está bloqueado si no existe otro estado* s' *tal que* $s \rightarrow s'$*. Esto lo denotamos con* $s \not\rightarrow$*.*

Obsérvese que, de acuerdo a su definición, todo estado final está bloqueado mas no al reves.

Definición 3 *Si* $s \rightarrow s'$ *entonces decimos que* s *es un redex, siendo* s' *un reducto de* s*.*

Definición 4 *Una secuencia de transición es una secuencia de estados posiblemente infinita*

$$s_0, \ldots, s_n, \ldots$$

tal que se cumple las siguientes condiciones

- s_0 inicial
- $s_i \rightarrow s_{i+1}$*, para cada* $0 \leq i \leq n$*.*

Una secuencia de transición es maximal si existe un n *tal que* $s_n \not\rightarrow$*. Una secuencia es completa si es maximal y además* s_n final*. Obsérvese que toda secuencia completa es maximal mas no al reves.*

Dada una relación de transición $\rightarrow$, se definen inductivamente las siguientes relaciones derivadas de importancia:

- La cerradura reflexiva y transitiva de $\rightarrow$, denotada $\rightarrow^\star$:

$$\frac{}{s \rightarrow^\star s}\ (crt1) \qquad \frac{s \rightarrow s' \quad s' \rightarrow^\star s''}{s \rightarrow^\star s''}\ (crt2)$$

  Intuitivamente se cumple $s \rightarrow^\star s'$ si y sólo si es posible llegar de s a s' mediante un número finito de pasos, tal vez ninguno, de la relación $\rightarrow$.

- La cerradura transitiva de $\rightarrow$, denotada $\rightarrow^+$:

$$\frac{s \rightarrow s'}{s \rightarrow^+ s'}\ (ct1) \qquad \frac{s \rightarrow s' \quad s' \rightarrow^+ s''}{s \rightarrow^+ s''}\ (ct2)$$

  Intuitivamente se cumple $s \rightarrow^+ s'$ si y sólo si es posible llegar de s a s' mediante un número finito de pasos, al menos uno, de la relación $\rightarrow$. Se deja como ejercicio mostrar que la segunda regla puede reemplazarse por:

$$\frac{s \rightarrow^+ s' \quad s' \rightarrow^+ s''}{s \rightarrow^+ s''}\ (ct2b)$$

- La iteración a n-pasos de $\rightarrow$, denotada $\rightarrow^n$ con n Nat:

$$\frac{}{s \rightarrow^0 s}\ (it0) \qquad \frac{s \rightarrow s' \quad s' \rightarrow^n s''}{s \rightarrow^{n+1} s''}\ (itn)$$

  Es facil ver que $\rightarrow = \rightarrow^1$

Dado que estas relaciones se definen recursivamente podemos usar inducción estructural para razonar acerca de ellas. Por ejemplo, el principio correspondiente a la relación $\rightarrow^\star$ es el siguiente: para mostrar que un juicio binario P sobre estados es válido para cualesquiera dos estados tales que $s \rightarrow^\star s'$ basta mostrar que P es cerrado bajo las reglas para la relación $\rightarrow^\star$, es decir, debemos probar que

- Base de la inducción: para todo estado s , se cumple $(s, s)\, P$
- Paso inductivo: Suponer $s \rightarrow s'$ y $(s', s'')\, P$ (Hipótesis de inducción) y probar que se cumple $(s, s'')\, P$.

La siguiente proposición se deja como ejercicio:

Proposición 1 *Para cualesquiera dos estados s, s' se cumple que $s \rightarrow^\star s'$ si y sólo si $s \rightarrow^k s'$ para algún k tal que k* Nat*.*

A continuación definimos un lenguaje para expresiones aritméticas y booleanas y estudiamos detalladamente su semántica formal tanto estática como dinámica utilizando las herramientas matemáticas antes expuestas.

3. EAB: Un lenguaje para expresiones aritméticas y booleanas

Se presenta una extensión del lenguaje para expresiones aritméticas EA con variables y expresiones booleanas, denotado EAB. Como operadores aritméticos agregamos el sucesor y el predecesor y como expresiones booleanas definimos las constantes true, false, el condicional if y un operador iszero que verifica si un número es cero.

La sintaxis concreta es:

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid \\
& & e + e \mid e * e \mid \mathsf{suc}\, e \mid \mathsf{pred}\, e \mid \\
& & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{iszero}\, e \mid \texttt{let}\ x = e\ \texttt{in}\ e\ \texttt{end}
\end{array}
$$

Si bien se observa que la gramática anterior es ambigua, entendemos que se puede transformar a una gramática no ambigua equivalente. El proceso de conversión para eliminar la ambigüedad no nos concierne aquí, puesto que trabajaremos siempre con la sintaxis abstracta que nunca es ambigua pues es el resultado del proceso de análisis sintáctico a partir de la gramática no ambigua para la sintaxis concreta.

La sintaxis abstracta se obtiene de manera similar al caso del lenguaje de expresiones aritméticas.

$$
\begin{array}{rcl}
t & ::= & x \mid \texttt{num}[n] \mid \texttt{bool}[\mathsf{true}] \mid \texttt{bool}[\mathsf{false}] \mid \\
& & \texttt{suma}(t_1, t_2) \mid \texttt{prod}(t_1, t_2) \mid \texttt{suc}(t) \mid \texttt{pred}(t) \mid \\
& & \texttt{if}(t_1, t_2, t_3) \mid \texttt{iszero}(t) \mid \texttt{let}(t_1, x.t_2)
\end{array}
$$

donde en la expresión $\texttt{num}[n]$ se sobreentiende que n Nat.

4. Semántica Dinámica para el lenguaje EAB

La semántica dinámica del lenguaje EAB se dará mediante un sistema de transición particular, usando el estilo llamado semántica operacional estructural o de paso pequeño.

- Estados $S = \{t \mid t\ \mathsf{asa}\}$. Es decir, los estados son árboles bien formados de la sintaxis abstracta[1].

$$\frac{t\ \mathsf{asa}}{t\ \mathsf{estado}}\ (edo)$$

- Estados iniciales $I = \{t \mid t\ \mathsf{asa},\ FV(t) = \varnothing\}$. Es decir, los estados iniciales son expresiones cerradas, es decir, sin variables libres.

$$\frac{t\ \mathsf{asa}\ \ FV(t) = \varnothing}{t\ \mathsf{inicial}}\ (ein)$$

- Estados finales

  Para poder modelar de manera fiel el proceso de evaluación definimos una categoría de *valores* los cuales son un subconjunto de expresiones que ya se han terminado de evaluar y no pueden reducirse más. De esta manera los valores representan a los posibles resultados finales de un proceso de evaluación. Su definición es la esperada, dada mediante un juicio v valor definido como:

[1] Recordemos que siempre trabajamos con árboles de ligado abstracto ala pero los llamamos asa

$$\frac{}{\texttt{bool}[\mathsf{true}]\ \mathsf{valor}}\ (vtrue) \qquad \frac{}{\texttt{bool}[\mathsf{false}]\ \mathsf{valor}}\ (vfalse) \qquad \frac{}{\texttt{num}[n]\ \mathsf{valor}}\ (vnum)$$

Durante todo el curso usaremos la metavariable v para designar valores. Definimos entonces a los estados finales como los valores

$$F = \{\texttt{num}[n] \mid n\ \mathsf{Nat}\} \cup \{\texttt{bool}[\mathsf{true}], \texttt{bool}[\mathsf{false}]\}.$$

En forma de juicio:

$$\frac{}{\texttt{num}[n]\ \mathsf{final}}\ (fnum) \qquad \frac{}{\texttt{bool}[\mathsf{true}]\ \mathsf{final}}\ (ftrue) \qquad \frac{}{\texttt{bool}[\mathsf{false}]\ \mathsf{final}}\ (ffalse)$$

- Los juicios de transición para expresiones aritméticas son:

$$\frac{}{\texttt{suma}(\texttt{num}[n], \texttt{num}[m]) \to \texttt{num}[n+m]}\ (esumaf) \qquad \frac{}{\texttt{prod}(\texttt{num}[n], \texttt{num}[m]) \to \texttt{num}[n*m]}\ (eprodf)$$

$$\frac{t_1 \to t_1'}{\texttt{suma}(t_1, t_2) \to \texttt{suma}(t_1', t_2)}\ (esumai) \qquad \frac{t_1 \to t_1'}{\texttt{prod}(t_1, t_2) \to \texttt{prod}(t_1', t_2)}\ (eprodi)$$

$$\frac{t_2 \to t_2'}{\texttt{suma}(\texttt{num}[n], t_2) \to \texttt{suma}(\texttt{num}[n], t_2')}\ (esumad) \qquad \frac{t_2 \to t_2'}{\texttt{prod}(\texttt{num}[n], t_2) \to \texttt{prod}(\texttt{num}[n], t_2')}\ (eprodd)$$

$$\frac{}{\texttt{suc}(\texttt{num}[n]) \to \texttt{num}[n+1]}\ (esucn) \qquad \frac{}{\texttt{pred}(\texttt{num}[0]) \to \texttt{num}[0]}\ (epred0)$$

$$\frac{}{\texttt{pred}(\texttt{num}[n+1]) \to \texttt{num}[n]}\ (epreds)$$

$$\frac{t \to t'}{\texttt{suc}(t) \to \texttt{suc}\, t'}\ (esuc) \qquad \frac{t \to t'}{\texttt{pred}(t) \to \texttt{pred}\, t'}\ (epred)$$

Obsérvese que la semántica del predecesor en cero no es la más adecuada, lo que deberíamos reportar en este caso es un error mediante una excepción. Esto puede modelarse introduciendo una constante de error pero en general aún no tenemos herramientas suficientes para hablar de excepciones.

- Los juicios de transición para expresiones booleanas son:

$$\frac{}{\texttt{if}(\texttt{bool}[\mathsf{true}], t_2, t_3) \to t_2}\ (eiftrue) \qquad \frac{}{\texttt{if}(\texttt{bool}[\mathsf{false}], t_2, t_3) \to t_3}\ (eiffalse)$$

$$\frac{t_1 \to t_1'}{\texttt{if}(t_1, t_2, t_3) \to \texttt{if}(t_1', t_2, t_3)}\ (eif)$$

$$\frac{}{\mathsf{iszero}(\texttt{num}[0]) \to \texttt{bool}[\mathsf{true}]}\ (eisz0) \qquad \frac{}{\mathsf{iszero}(\texttt{num}[n+1]) \to \texttt{bool}[\mathsf{false}]}\ (eiszs)$$

$$\frac{t_1 \to t_1'}{\mathsf{iszero}(t_1) \to \mathsf{iszero}\, t_1'}\ (eisz)$$

- Los juicios de transición para expresiones `let` son:

$$\frac{v \text{ valor}}{\texttt{let}(v, x.e_2) \to e_2[x := v]}\ (eletf)$$

$$\frac{t_1 \to t_1'}{\texttt{let}(t_1, x.t_2) \to \texttt{let}(t_1', x.t_2)}\ (eleti)$$

Algunas observaciones son necesarias:

- Las variables son estados bloqueados, pero no son finales. En particular las variables libres no pueden evaluarse.
- Las reglas sin premisas que involucren a la relación de transición se conocen como instrucciones puesto que corresponden a pasos primitivos de ejecución, mientras que las reglas restantes definen transiciones condicionales que determinan el orden en el que se ejecutan las instrucciones.
- A veces utilizaremos la sintaxis concreta para facilitar el manejo de las transiciones en el papel o pizarrón, sin embargo debemos recordar que cualquier discusión se refiere a la sintaxis abstracta. Por ejemplo las reglas de evaluación de las expresiones aritméticas suma y producto, así como de las expresiones let se escribirían como sigue:

$$\frac{n + m = p}{n + m \to p} \qquad \frac{n * m = p}{n * m \to p}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_1 \to e_1'}{e_1 * e_2 \to e_1' * e_2}$$

$$\frac{e_2 \to e_2'}{n + e_2 \to n + e_2'} \qquad \frac{e_2 \to e_2'}{n * e_2 \to n * e_2'}$$

$$\frac{v \text{ valor}}{\texttt{let } x = v \texttt{ in } e_2 \texttt{ end} \to e_2[x := v]}$$

$$\frac{e_1 \to e_1'}{\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \to \texttt{let } x = e_1' \texttt{ in } e_2 \texttt{ end}}$$

4.1. Estrategias de Evaluación

Si observamos con detalle las reglas de transición observamos que los operadores básicos como sumas y productos se evaluan ejecutando la operación respectiva mientras que las expresiones `let` se evaluan sustituyendo el valor de la variable ligada en el cuerpo. En los tres casos se requiere que los argumentos principales de la expresión, ambos en el caso de suma y producto y el ligado de variable e_1 en una expresión `let`, sean valores. Esto modela una estrategia particular de evaluación, la llamada por valor[2] correspondiente a la evaluación ansiosa o glotona[3]. Las instrucciones o reglas

[2]En inglés *call-by-value*
[3]En inglés *Eager evaluation*

finales sólo se aplican cuando los argumentos principales se han evaluado completamente. En nuestro caso se eligió el orden de evaluación de izquierda a derecha, es decir, primero se evalua el primer argumento, luego el segundo, etcétera. Una vez que los argumentos se evaluan la instrucción se ejecuta. La evaluación puede tomar múltiples pasos, pero las transiciones se definieron de manera que en cada una de ellas se ejecuta un solo paso de evaluación.

Ejemplo 4.1 Consideremos la siguiente secuencia de evaluación para la expresión

$$\texttt{let } x = 1 + 2 \texttt{ in } \mathsf{iszero}\,((x + 5) * (x + 2)) \texttt{ end}$$

donde usamos sintaxis concreta y escribimos en el extremo derecho de cada renglón la combinación de reglas usadas para la siguiente transición.

$$\begin{array}{lcl}
\texttt{let } x = 1 + 2 \texttt{ in } \mathsf{iszero}\,((x + 5) * (x + 2)) \texttt{ end} & \to & (eleti) \\
\texttt{let } x = 3 \texttt{ in } \mathsf{iszero}\,((x + 5) * (x + 2)) \texttt{ end} & \to & (eletf) \\
\big(\mathsf{iszero}\,((x + 5) * (x + 2))\big)[x := 3] & = & \\
\mathsf{iszero}\,((3 + 5) * (3 + 2)) & \to & (eisz), (esumaf) \\
\mathsf{iszero}\,(8 * (3 + 2)) & \to & (eisz), (eprodd), (esumaf) \\
\mathsf{iszero}\,(8 * 5) & \to & (eisz), (eprodf) \\
\mathsf{iszero}\ 40 & \to & (eiszs) \\
\mathsf{false} & &
\end{array}$$

Más adelante discutiremos otras estrategias de evaluación.

4.2. Propiedades de la semántica dinámica

La relación de transición $\to$ del lenguaje EAB cumple las siguientes propiedades.

Proposición 2 *Si v valor entonces $v \not\to$, es decir v está bloqueado.*

Demostración. Inducción sobre el juicio v valor.

Proposición 3 (Determinismo de la relación $\to$) *Si $e \to e_1$ y $e \to e_2$ entonces $e_1 = e_2$.*

Demostración. Inducción sobre la relación $e \to e_1$.

Corolario 1 (Determinismo de la relación $\to^\star$) *Si $e \to^\star e_1$ y $e \to^\star e_2$ con $e_1 \not\to$, $e_2 \not\to$ entonces $e_1 = e_2$.*

4.3. Evaluación de expresiones

La evaluación de una expresión puede definirse mediante la semántica operacional como sigue:

Definición 5 *La función de evaluación $eval : EAB \to EAB$ se define como sigue:*

$$eval(e) = e_f \text{ si y sólo si } e \to^\star e_f \ \text{ y } \ e_f \not\to$$

El corolario 1 garantiza que *eval* es una función. De manera más específica este corolario garantiza que *eval* es una función parcial, es decir si dada una expresión e la expresión $eval(e)$ existe entonces es única. Sin embargo no sabemos que para toda expresión e la expresión $eval(e)$ exista pues podría darse el caso de que el proceso de reducción de la semántica operacional sea infinito. Este no es el caso para EAB como lo garantiza la siguiente

Proposición 4 (Terminación) *Para cada expresión e existe una expresión e_f tal que*

- $e \rightarrow^\star e_f$
- e_f *está bloqueada.*

Demostración. La demostración es complicada debido a la presencia de expresiones let y se omite. El lector debe convencerse de que cualquier intento de inducción estructural fallará. ⊣

Obsérvese que aunque *eval* sí es una función no podemos garantizar que la evaluación de una expresión devuelva como resultado un valor, debido a que existen estados (expresiones) bloqueados, por ejemplo

$$x,\ 2 + x,\ \mathsf{iszero}\,\mathsf{true},\ \mathsf{if}\ 3 + 5\ \mathsf{then}\ 4\ \mathsf{else}\ x$$

en estos casos la función de evaluación devuelve una expresión bloqueada que no es un valor. Esto corresponde a un error en tiempo de ejecución.

Para eliminar esta clase de errores y garantizar que cada vez que el proceso de evaluación se lleve a cabo se devuelva un valor, requerimos de la llamada semántica estática.

4.4. Implementación

La semántica dinámica debe implementarse mediante las siguientes funciones:

- `eval1 : EAB ->EAB` tal que `eval1 e = e'` syss $e \rightarrow e'$
- `evals: EAB ->EAB` tal que `evals e = e'` syss $e \rightarrow^\star e'$ y e' está bloqueado. Por ejemplo la ejecución de `evals ((2*6)+true)` debe devolver `12+true`
- `eval :EAB ->EAB` tal que `eval e = e'` syss $e \rightarrow^\star e'$ y e' es un valor. La diferencia con `evals` es que deben manejarse los errores de ejecución. Por ejemplo la ejecución de `eval(2+true)` debe devolver un error informativo acerca de que la suma sólo funciona con números.

5. Semántica Estática

Dependiendo del lenguaje, un compilador o intérprete puede verificar ciertas propiedades semánticas en tiempo de compilación (es decir 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 una manera más simple y eficaz. Antes de definir el significado preciso de un programa es necesario eliminar los programas sin sentido, por ejemplo, si un lenguaje es fuertemente tipado entonces su semántica dinámica sólo estará definida si el programa en cuestión está bien formado respecto a la semántica estática.

La semántica *estática* determina qué estructuras de la sintaxis abstracta (asa o ala) están bien formadas de acuerdo a ciertos criterios sensibles al contexto como la resolución del alcance, al requerir que cada variable sea declarada antes de usarse (recordemos que este requerimiento no puede modelarse con una gramática libre de contexto).

Por lo general la semántica estática consiste de dos fases:

- La resolución del alcance de variables, el cual en nuestro caso está codificado directamente en la sintaxis abstracta.
- La verificación de correctud estática de un programa mediante la interacción con el sistema de tipos.

5.1. Sistemas de Tipos

Un sistema de tipos es una colección de reglas de inferencia que imponen ciertas restricciones en la formación de programas. Las frases del lenguaje se clasifican mediante tipos que dictan cómo pueden usarse en combinación con otras frases. Intuitivamente el tipo de una frase predice la forma de su valor, por ejemplo, la suma de dos expresiones numéricas debe ser numérica. Por otra parte si intentamos sumar una expresión numérica con una expresión booleana digamos $\mathsf{true} + 7$ dicha expresión debe ser prohibida ya que genera un error de tipos[4].

De manera intuitiva podemos decir que un tipo es un descripción abstracta de una colección de valores particulares. De esta forma podría pensarse ingenuamente que un tipo es un conjunto, situación correcta en algunos casos como los enteros, flotantes pero dicha interpretación deja de ser cierta en casos complejos como los tipos de funciones o los tipos polimórficos. En realidad los tipos se interpretan de manera correcta como órdenes parciales en el estudio de semánticas denotativas.

El uso de sistemas de tipos en el diseño de un lenguaje de programación tiene las siguientes ventajas:

- Permite descubrir errores de programación tempranamente.
- Ofrece seguridad, un programa correctamente tipado no puede funcionar mal.
- Soporta abstracción, importante para la descripción de interfaces.
- Los tipos documentan un programa de manera mas simple y manejable que los comentarios.
- Los lenguajes tipados pueden implementarse de manera más clara y eficiente.

Una propiedad fundamental de los sistemas de tipos es la llamada *seguridad o correctud del sistema*[5]: los programas correctamente tipados no pueden funcionar mal.

[4] Type-checking error

[5] *Type safety* o *type soundness*

Un programa es erróneo si su evaluación se bloquea, es decir termina en una expresión que no es un valor simple y no puede seguir evaluándose.

La seguridad del sistema de tipos relaciona a la semántica estática con la semántica dinámica y se prueba generalmente en dos partes

- *Progreso*: Un programa correctamente tipado no se bloquea.
- *Preservación*: Si un programa correctamente tipado se ejecuta entonces el programa resultante está correctamente tipado. De hecho en muchos casos se preserva el mismo tipo.

5.2. Semántica Estática

La semántica estática del lenguaje EAB se define mediante un juicio ternario entre expresiones t, tipos T y contextos de declaraciones de variables tipadas Γ. Este juicio captura a la relación "*la expresión t tiene tipo T bajo el contexto Γ*" y se denota

$$\Gamma \vdash t : \mathsf{T}$$

en donde

- $\Gamma = \{x_1 : \mathsf{T}_1, \ldots, x_n : \mathsf{T}_n\}$ es un conjunto de declaraciones de variables llamado contexto de tipado donde si $i \neq j$ entonces $x_i \neq x_j$. Es decir, todas las variables son distintas. En este sentido Γ puede considerarse como una función finita tal que $\Gamma(x_i) = \mathsf{T}_i$.
- t es una expresión de la sintaxis abstracta.
- T es un tipo.

5.2.1. El sistema de tipos para EAB

En nuestro lenguaje sólo tenemos dos valores posibles: números o booleanos, por lo que sólo necesitamos dos tipos.

$$\mathsf{T} ::= \mathsf{Nat} \mid \mathsf{Bool}$$

La relación de tipado $\Gamma \vdash t : \mathsf{T}$ se define inductivamente como sigue:

- Tipado de variables
$$\frac{}{\Gamma, x : \mathsf{T} \vdash x : \mathsf{T}}\ (tvar)$$
- Tipado de valores numéricos:
$$\frac{}{\Gamma \vdash \mathtt{num}[n] : \mathsf{Nat}}\ (tnum)$$
- Tipado de valores booleanos:
$$\frac{}{\Gamma \vdash \mathtt{bool}[\mathsf{true}] : \mathsf{Bool}}\ (ttrue) \qquad \frac{}{\Gamma \vdash \mathtt{bool}[\mathsf{false}] : \mathsf{Bool}}\ (tfalse)$$

- Tipado de expresiones aritméticas:

$$\frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{suma}(t_1, t_2) : \mathsf{Nat}}\ (tsum) \qquad \frac{\Gamma \vdash t_1 : \mathsf{Nat} \quad \Gamma \vdash t_2 : \mathsf{Nat}}{\Gamma \vdash \mathtt{prod}(t_1, t_2) : \mathsf{Nat}}\ (tprod)$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathtt{suc}\, t : \mathsf{Nat}}\ (tsuc) \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathtt{pred}\, t : \mathsf{Nat}}\ (tpred)$$

- Tipado de expresiones booleanas:

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \quad \Gamma \vdash t_2 : \mathsf{T} \quad \Gamma \vdash t_3 : \mathsf{T}}{\Gamma \vdash \mathtt{if}(t_1, t_2, t_3) : \mathsf{T}}\ (tif) \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathtt{iszero}\, t : \mathsf{Bool}}\ (tisz)$$

- Tipado de expresiones `let`:

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \quad \Gamma, x : \mathsf{T} \vdash e_2 : \mathsf{S}}{\Gamma \vdash \mathtt{let}(e_1, x.e_2) : \mathsf{S}}\ (tlet)$$

En esta última regla el contexto $\Gamma, x : \mathsf{T}$ que figura en la segunda premisa es en realidad el contexto $\Gamma \cup \{x : \mathsf{T}\}$, por lo que de acuerdo a la definición, la declaración $x : \mathsf{T}$ no figura en Γ. Obsérvese entonces que la variable x ya no figura en el contexto Γ que tipa a la expresión `let` por lo que las variables ligadas por el let no existen fuera de su alcance, es decir, son variables locales al let.

Las reglas recien presentadas están organizadas de acuerdo a un principio de introducción y eliminación de tipos. Los constructores del lenguaje pueden clasificarse como *formas de introducción* cuando son el medio para crear expresioes de un tipo en particular o como *formas de eliminación* cuando proporcionan un medio para computar con expresiones de cierto tipo para así obtener valores del mismo u otro tipo. En nuestro caso los constructores de introducción son $\mathtt{num}[n]$ y $\mathtt{bool}[b]$ mientras que los constructores restantes son formas de eliminación, a excepción de las variables que no se clasifican en ninguno de estos rubros.

Como ya mencionamos antes, es común usar la notación de la sintaxis concreta por simplicidad, situación de la que nos serviremos constantemente, escribiendo por ejemplo:

$$\Gamma \vdash \mathsf{true} : \mathsf{Bool} \quad \Gamma \vdash 0 : \mathsf{Nat} \quad \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{T}$$

Ejemplo 5.1 Consideremos la expresión `let` del ejemplo 4.1. Derivamos a continuación el tipado

$$\vdash \mathtt{let}\ x = 1 + 2\ \mathtt{in}\ \mathsf{iszero}\,((x + 5) * (x + 2))\ \mathtt{end} : \mathsf{Bool}$$

que muestra que dicha expresión es de tipo booleano en el contexto vacío. Enumeramos cada paso de la derivación secuencialmente para hacer referencia a éllos en pasos posteriores, justificando con las reglas utilizadas en el extremo derecho del renglón.

1. $x : \mathsf{Nat} \vdash x : \mathsf{Nat}$ $(tvar)$
2. $\vdash 1 : \mathsf{Nat}$ $(tnum)$
3. $\vdash 2 : \mathsf{Nat}$ $(tnum)$
4. $\vdash 1 + 2 : \mathsf{Nat}$ $(tsum)\ 2, 3$
5. $x : \mathsf{Nat} \vdash 5 : \mathsf{Nat}$ $(tnum)$
6. $x : \mathsf{Nat} \vdash x + 5 : \mathsf{Nat}$ $(tsum)\ 1, 5$
7. $x : \mathsf{Nat} \vdash 2 : \mathsf{Nat}$ $(tnum)$
8. $x : \mathsf{Nat} \vdash x + 2 : \mathsf{Nat}$ $(tsum)\ 1, 7$
9. $x : \mathsf{Nat} \vdash (x + 5) * (x + 2) : \mathsf{Nat}$ $(tprod)\ 6, 8$
10. $x : \mathsf{Nat} \vdash \mathsf{iszero}\big((x + 5) * (x + 2)\big) : \mathsf{Bool}$ $(tisz)\ 9$
11. $\vdash \texttt{let}\ x = 1 + 2\ \texttt{in}\ \mathsf{iszero}\big((x + 5) * (x + 2)\big)\ \texttt{end} : \mathsf{Bool}$ $(tlet)\ 4, 10$

5.3. Implementación

La relación de tipado debe implementarse como sigue:

- Definir un tipo `Tipo` para los tipos de EAB
- Definir un tipo `Ctx` para los contextos $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$ lo más simple es una lista de pares de variables y tipos.
- Definir la función de verificación de tipado `vt :: Ctx ->EAB ->Tipo ->Bool` tal que

$$\texttt{vt}\ \ \Gamma\ e\ T\ \ \texttt{= True}\ \text{si y sólo si}\ \Gamma \vdash e : T.$$

5.4. Propiedades de la semántica estática

A continuación enunciamos algunas propiedades importantes de la relación de tipado.

Las reglas de tipado son dirigidas por la sintaxis en el sentido de que hay exactamente una regla por cada forma de expresión. En consecuencia, es fácil dar condiciones necesarias para el tipado de una expresión que inviertan las condiciones suficientes expresadas por la regla de tipado correspondiente.

Lema 1 (Inversión) *Las reglas de inferencia de la semántica estática son invertibles, es decir se cumple lo siguiente:*

- *Si* $\Gamma \vdash x : \mathsf{T}$ *entonces la declaración* $x : \mathsf{T}$ *figura en* Γ.
- *Si* $\Gamma \vdash \texttt{num}[n] : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Nat}$
- *Si* $\Gamma \vdash \texttt{bool}[\mathsf{true}] : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Bool}$.
- *Si* $\Gamma \vdash \texttt{bool}[\mathsf{false}] : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Bool}$.
- *Si* $\Gamma \vdash \texttt{suma}(t_1, t_2) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Nat}$ *y* $\Gamma \vdash t_1 : \mathsf{Nat},\ \Gamma \vdash t_2 : \mathsf{Nat}$
- *Si* $\Gamma \vdash \texttt{prod}(t_1, t_2) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Nat}$ *y* $\Gamma \vdash t_1 : \mathsf{Nat},\ \Gamma \vdash t_2 : \mathsf{Nat}$

- *Si* $\Gamma \vdash \texttt{suc}(t) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Nat}$ *y* $\Gamma \vdash t : \mathsf{Nat}$.
- *Si* $\Gamma \vdash \texttt{pred}(t) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Nat}$ *y* $\Gamma \vdash t : \mathsf{Nat}$.
- *Si* $\Gamma \vdash \texttt{if}(t_1, t_2, t_3) : \mathsf{T}$ *entonces* $\Gamma \vdash t_1 : \mathsf{Bool}, \Gamma \vdash t_2 : \mathsf{T}$ *y* $\Gamma \vdash t_3 : \mathsf{T}$.
- *Si* $\Gamma \vdash \texttt{iszero}(t) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Bool}$ *y* $\Gamma \vdash t : \mathsf{Nat}$.
- *Si* $\Gamma \vdash \texttt{let}(t_1, x.t_2) : \mathsf{S}$ *entonces existe* T *tal que* $\Gamma \vdash t_1 : \mathsf{T}$ *y* $\Gamma, x : \mathsf{T} \vdash t_2 : \mathsf{S}$.

Demostración. Inmediato de las reglas de tipado ⊣

Este lema implica que el tipo de un término puede calcularse a partir del tipo de sus subtérminos y proporciona un algoritmo para tipar términos.

Proposición 5 (Unicidad del tipado) *Dados cualquier contexto* Γ *y expresión* t *existe a lo más un tipo* T *tal que* $\Gamma \vdash t : \mathsf{T}$. *Más aún, cada término tipable en un contexto tiene una única derivación de su tipo.*

Demostración. Inducción sobre t. ⊣

Proposición 6 (Formas Canónicas) *Sea* v *un valor.*

- *Si* $\Gamma \vdash v : \mathsf{Bool}$ *entonces* $v = \texttt{bool}[\mathsf{true}]$ *ó* $v = \texttt{bool}[\mathsf{false}]$.
- *Si* $\Gamma \vdash v : \mathsf{Nat}$ *entonces* $v = \texttt{num}[n]$.

Demostración. Por inspección de la forma sintáctica de v y usando el lema de inversión. ⊣

5.4.1. Propiedades estructurales

Las propiedades estructurales de los sistemas de tipos son aquellas que se refieren a la estructura de las derivaciones y por lo general se utilizan para simplificar u optimizar el proceso de tipado.

Lema 2 (Debilitamiento) *Si* $\Gamma \vdash t : \mathsf{T}$ *entonces* $\Gamma, x : \mathsf{S} \vdash t : \mathsf{T}$ *para cualquier variable* x *no declarada en* Γ *y cualquier tipo* S.

Demostración. Inducción sobre $\Gamma \vdash t : \mathsf{T}$ ⊣

Desde el punto de vista de la programación el lema anterior nos permite usar una expresión en cualquier contexto que ligue a sus variables libres. Si t está bien tipada bajo Γ entonces podemos importar este tipado a cualquier contexto que incluya a Γ. En otras palabras la introducción de más variables que las requeridas por una expresión t no invalida el tipado de la misma.

Lema 3 (Sustitución) *Si* $\Gamma, x : \mathsf{S} \vdash t : \mathsf{T}$ *y* $\Gamma \vdash s : \mathsf{S}$ *entonces* $\Gamma \vdash t[x := s] : \mathsf{T}$

Demostración. Inducción sobre $\Gamma, x : \mathsf{S} \vdash t : \mathsf{T}$ $\dashv$

Este lema expresa cierto concepto de modularidad si pensamos en las expresiones t y s como dos componentes de un sistema donde t es un cliente de la implementación s. El cliente declara una variable que especifica el tipo de la implementación y su tipado se verifica sólo con esta información. Por otra parte la implementación debe tiparse con el mismo tipo de dicha variable para satisfacer las necesidades del cliente. En tal caso, podemos ligar al cliente y a la implementación obteniendo el sistema $t[x := s]$, repitiendo este proceso se eliminan todas las variables declaradas y se obtiene un sistema (expresión cerrada) listo para ejecutarse (evaluarse).

6. Seguridad del lenguaje

Muchos de los lenguajes de programación contemporáneos son *seguros* (seguros por tipos[6], fuertemente tipados). Informalmente esto significa que cierta clase de errores no pueden surgir durante la ejecución. Por ejemplo, la seguridad de EAB afirma que la ejecución o evaluación nunca se encontrará en una situación en que un booleano se sume a un número o en que un condicional tenga a un número como guardia.

En general la seguridad por tipos expresa la coherencia entre la semántica estática y la semántica dinámica. La semántica estática predice que el valor de una expresión tendrá cierta forma de manera que la semántica dinámica de dicha expresión está bien definida. En consecuencia, la evaluación no puede bloquearse en un estado no final para el cual no haya transición posible, lo cual corresponde en una implementación a la ausencia de errores causados por una instrucción ilegal en tiempo de ejecución.

La seguridad consta de dos aspectos, el progreso de la evaluación y la preservación de tipos:

- Preservación: relaciona la evaluación y el tipado, si $\Gamma \vdash e : \mathsf{T}$ y $e \to e'$ entonces $\Gamma \vdash e' : \mathsf{T}$, es decir si un programa tipado correctamente toma un paso de evaluación entonces el resultado es un programa correctamente tipado (y generalmente con el mismo tipo).
- Progreso: relaciona el tipado de expresiones cerradas y los valores (estados finales), si $\vdash e : T$ entonces e es un valor o $e \to e'$ para algún e', es decir, en un programa correctamente tipado la evaluación progresa hasta terminar.

Proposición 7 (Preservación de tipos) *Si* $\Gamma \vdash e : T$ *y* $e \to e'$ *entonces* $\Gamma \vdash e' : T$.

Demostración. Inducción sobre $\Gamma \vdash e : T$. $\dashv$

Proposición 8 (Progreso) *Si* $\vdash e : \mathsf{T}$ *para algún tipo* T *entonces se cumple una y sólo una de las siguientes condiciones:*

- *e es un valor*
- *Existe una expresión e' tal que $e \to e'$.*

[6] *Type safe*

Demostración. Inducción sobre $\vdash e : T$. $\dashv$

Obsérvese que el progreso sólo se garantiza en programas cerrados, es decir, en programas que no tienen variables libres.

6.1. Implementación de la evaluación empleando la seguridad del lenguaje

Empleando el filtro de la semántica estática, implementar la función de evaluación `evalt` que es esencialmente una reimplementación de la función `eval` de la sección 4.4 de forma que `evalt` verifique el tipado de la expresión y sólo en caso positivo (servirse de la función `vt`) inicie el proceso de evaluación cuyo resultado será necesariamente un valor. En otro caso debe informarse de un error por existencia de variables libres o por tipado. Por ejemplo `evalt (x+2)` debe devolver un error informativo indicando que la expresión tiene variables libres y `evalt (iszero true)` debe devolver un error informativo acerca de que el tipo del argumento del operador `izero` debe ser `Nat`.

7. Terminación

La seguridad de un lenguaje no garantiza por sí sola que la evaluación de un programa correctamente tipado termine eventualmente, de hecho esta propiedad no se cumple en lenguajes más complicados como veremos más adelante, ni por supuesto en los lenguajes reales de programación. Ya observamos que la propiedad es válida de manera más general en EAB. Sin embargo la interacción de ambas semánticas $\rightarrow$ y $\vdash$ nos permite garantizar algo más.

Proposición 9 (Terminación de la evaluación) *Si e es un programa correctamente tipado sin variables libres, es decir si $\vdash e : \mathsf{T}$, entonces existe un valor v tal que $e \rightarrow^\star v$*

Demostración. Las pruebas de terminación por lo general requieren de una herramienta conocida como *relaciones lógicas y/o predicados de computabilidad* que están fuera del alcance de este curso. $\dashv$

La terminación de la evaluación garantiza que ningún programa correctamente tipado diverge, es decir no hay ciclos infinitos. Esta propiedad implica en particular que el lenguaje en cuestión carece de recursión general, es decir, no es Turing-completo.