Lenguajes del Programación
Unidad 2: Especificación Formal de los Lenguajes de Programación

Semántica Estática

Manuel Soto Romero*

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota vamos a adentrarnos en el concepto de semántica estática, es decir, esa parte del significado de los programas que podemos analizar sin necesidad de ejecutarlos. Exploraremos cómo el lenguaje mismo nos ofrece herramientas para detectar errores antes de correr el código, a través de mecanismos como la verificación de tipos, el alcance de las variables y las reglas de visibilidad. Además, presentaremos un sistema de tipos sencillo aplicado a nuestra versión de MiniLisp, extendiendo su sintaxis con valores booleanos y la operación de negación. A partir de esto, veremos cómo se formulan reglas de inferencia para decidir si una expresión está bien tipificada, y con ello comprenderemos mejor la importancia de la semántica estática para mejorar la calidad, la seguridad y la confiabilidad de los programas que escribimos.

1. Introducción

En el mundo de los lenguajes de programación, la semántica estática se refiere a la parte del significado de los programas que podemos analizar sin necesidad de ejecutar el programa. Es como si al leer un libro, pudieramos entender qué va a pasar en la historia sin necesidad de leer todo el libro, sólo con observar ciertas partes o reglas del texto.

Por ejemplo, consideremos la declaración de una variable en C:

```
int edad = 30;
```

Aquí, `int` indica que la variable `edad` debe ser un número entero. La semántica de este programa nos dice que siempre que veamos `edad`, se refiere a un número entero. No necesitamos ejecutar el programa para saber esto; podemos deducirlo sólo con observar la declaración.

La semántica estática se ocupa, de cosas como:

Verificación de tipos Asegurarse de que no estamos intentando realizar operaciones no permitidas, como sumar un número y una cadena de texto.

Alcance de las variables Determinar dónde están disponibles ciertas variables en nuestro código.

*Un agradecimiento a Juan Mario Sosa Romo por sus aportes durante la revisión de esta nota como parte de su servicio social.

Reglas de visibilidad Decidir qué partes del código pueden acceder a qué datos.

Este nivel de semántica es importante porque ayuda a prevenir errores en el código antes de que el programa se ejecute, es decir, ayuda a corregir nuestros programas antes de que lo ejecutemos, asegurando que las partes del código tengan sentido en el contexto del lenguaje de programación que estemos usando.

Proporciona una forma de analizar y asegurar la validez de nuestros programas en términos de tipos y estructura sin necesidad de ejecutarlo. Esto no sólo optimiza el proceso de desarrollo, sino que también mejora la calidad y seguridad del software que escribimos.

2. Un sistema de tipos para MiniLisp

Dado que nuestra versión de MiniLisp es aún muy *joven* para realizar un análisis sofisticado de la semántica estática, mostraremos el proceso que se sigue para realizar una verificación de tipos sencilla y profundizaremos en estos temas más adelante en otra nota.

Un sistema de tipos en la teoría de lenguajes de programación es un sistema formal que asigna un tipo a las construcciones sintácticas de un lenguaje de programación, como variables, expresiones o funciones, con el objetivo de reducir la posibilidad de errores en el comportamiento del programa. El sistema de tipos utiliza reglas formales para determinar si una expresión o un programa es válido bajo las restricciones de tipos especificadas.

Definición 1: Sistema de Tipos

Un ***sistema de tipos*** *puede ser definido como una tripléta* $(T, \Gamma, \vdash)$*, donde:*

- T *es un conjunto de* ***tipos.***
- Γ *es un* ***contexto de tipificado****, que es un mapeo de identificadores a tipos.*
- $\vdash$ *es una* ***relación de derivación de tipos****, que es una relación entre un contexto* Γ*, una expresión* e *y un tipo* τ*. Esto se denota como* $\Gamma \vdash e : \tau$*, que se lee como "bajo el contexto* Γ*, la expresión* e *tiene el tipo* τ*.*

Extendiendo MiniLisp

Sintaxis concreta

Añadiremos expresiones booleanas a nuestro lenguaje para que tengamos al menos dos tipos. La gramática correspondiente se muestra a continuación, sólo incluiremos la operación de negación para estas expresiones.

```
<S> := <Expr>

<Expr> ::= <Int>
       | <Bool>
       | (+ <Expr> <Expr>)
       | (- <Expr> <Expr>)
       | (not <Expr>)

<Int> := <N>
```

```
        | -<M>

<D> := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<N> := 0 | <D>{<N>}

<M> := <D>{<N>}

<Bool> := #f | #t
```

Sintaxis abstracta

Para las expresiones booleanas, usaremos la etiqueta *Boolean*:

- *Boolean*(b) es un ASA si $b \in \mathbb{B}$.

$$\frac{b \in \mathbb{B}}{\mathit{Boolean}(b)\ \mathsf{ASA}}$$

Para el operador `not` usaremos la etiqueta *Not*:

- *Not*(e) es un ASA si tanto e es un ASA también. Es decir, es un árbol con un subárbol.

$$\frac{e\ \mathsf{ASA}}{\mathit{Not}(e)\ \mathsf{ASA}}$$

Semántica dinámica

Tomaremos la funcionalidad clásica del operador `not` usada en Lisp, es decir*, todo lo que no es explícitamente* `#t` *(verdadero), es* `#f` *(falso)*. Usaremos semántica natural para especificar su comportamiento.

$$\frac{}{\mathit{Boolean}(b) \Rightarrow \mathit{Boolean}(b)}$$

$$\frac{e \Rightarrow \mathit{Boolean}(False)}{\mathit{Not}(e) \Rightarrow \mathit{Boolean}(True)}$$

$$\frac{e \Rightarrow \mathit{Num}(n)}{\mathit{Not}(e) \Rightarrow \mathit{Boolean}(False)}$$

$$\frac{e \Rightarrow \mathit{Boolean}(True)}{\mathit{Not}(e) \Rightarrow \mathit{Boolean}(False)}$$

Veamos ahora cómo verificar si una expresión está bien tipificada.

Reglas de inferencia

Los sistemas de tipos utilizan reglas de inferencia para determinar si una expresión está bien tipificada. Estas reglas definen cómo los tipos de las subexpresiones de una expresión compleja determinan el tipo de toda la expresión.

- **Regla de tipificado para números**

$$\frac{}{\Gamma, Num(n) : \mathsf{number}}$$

  Esta regla establece que bajo un contexto de tipos cualquiera Γ, el tipo de un número siempre será number (no debe verificarse ninguna restricción/propiedad adicional).

- **Regla de tipificado para booleanos**

$$\frac{}{\Gamma, Boolean(b) : \mathsf{boolean}}$$

  Esta regla establece que bajo un contexto de tipos cualquiera Γ, el tipo de un booleano siempre será boolean (no debe verificarse ninguna restricción/propiedad adicional).

- **Regla de tipificado para expresiones aritméticas**

$$\frac{\Gamma, i : \mathsf{number} \qquad \Gamma, d : \mathsf{number}}{\Gamma, Add(i, d) : \mathsf{number}}$$

  Esta regla establece que bajo un contexto de tipos cualquiera Γ, el tipo de una suma será number siempre y cuando tanto el lado izquierdo como el derecho de la suma sean números también. Dicho de otra forma, sólo se permite realizar sumas entre números. Observemos que ninguna de las especificaciones anteriores (sintaxis concreta, sintaxis abstracta o semántica dinámica) capturan esta noción.

$$\frac{\Gamma, i : \mathsf{number} \qquad \Gamma, d : \mathsf{number}}{\Gamma, Sub(i, d) : \mathsf{number}}$$

  Esta regla establece que bajo un contexto de tipos cualquiera Γ, el tipo de una resta será number siempre y cuando tanto el lado izquierdo como el derecho de la resta sean números también.

- **Regla de tipificado para expresiones booleanas**

$$\frac{}{\Gamma, Not(e) : \mathsf{boolean}}$$

  Esta regla establece que bajo un contexto de tipos cualquiera Γ, el tipo de la operación de negación siempre será boolean (no debe verificarse ninguna restricción/propiedad adicional). El hecho de que el operador funcione con cualquier tipo de dato justifica este comportamiento.

Observación 1

Notemos que las reglas anteriores realmente no están haciendo uso del contexto de tipo, motivo por el cuál podríamos quitarlo de la especificación. Estudiaremos estos más adelante, sin embargo, con el fin de contextualizar su uso, éste se integrará con el uso de las llamadas variables de tipo.

Exploraremos algunas propiedades de los sistemas de tipos y modelaremos otro tipo de propiedades que permite capturar la semántica estática más adelante, cuando nuestro lenguaje sea lo suficientemente maduro para incluir este tipo de características.

3. Conclusión

En conclusión, lo revisado en esta nota nos permite reconocer que la semántica estática constituye un primer filtro esencial para garantizar que nuestros programas tengan sentido antes de ser ejecutados. A través del sistema de tipos y de las reglas de inferencia, observamos cómo es posible formalizar restricciones que nos protegen de errores comunes y nos obligan a pensar de manera más rigurosa sobre el diseño de un lenguaje. El ejercicio de extender MiniLisp con expresiones booleanas y el operador `not` nos mostró, de forma práctica, cómo estas ideas se aplican incluso en lenguajes muy básicos. Más allá de la técnica, lo importante es comprender que la semántica estática no sólo mejora la eficiencia del desarrollo, sino que también constituye una base teórica sólida para la construcción de lenguajes de programación más expresivos, seguros y confiables.

Referencias

[1] K.-M. Choe y K.-D. Lee, *Foundations of Programming Languages*. Springer, 2017.
[2] M. Gabbrielli y S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2023.
[3] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. Brown University, 2007.
[4] H. R. Nielson y F. Nielson, *Semantics with Applications: An Appetizer*. Springer, 2007.
[5] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.
[6] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.