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

Semántica Dinámica

Manuel Soto Romero*

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota estudiaremos la semántica dinámica de los programas como fundamento para razonar con precisión sobre su ejecución. Comenzaremos fijando intuición con ejemplos simples y, enseguida, presentaremos tres enfoques de especificación: la semántica denotativa (significado como función matemática), la semántica axiomática o Lógica de Hoare (pre y postcondiciones), y el enfoque que usaremos de forma central, la semántica operacional. Formalizaremos la ejecución mediante sistemas de transición y trabajaremos los dos estilos operacionales: paso grande (semántica natural) y paso pequeño (semántica estructural), con reglas de inferencia y derivaciones sobre MiniLisp como lenguaje de estudio. Mostraremos la equivalencia entre ambos estilos (cerradura reflexivo-transitiva de $\to$ frente a $\Rightarrow$) y discutiremos propiedades importantes del lenguaje (determinismo y terminación) para entender por qué estas garantías son valiosas al diseñar, verificar y optimizar programas.

1. Introducción

Para explicar esta técnica, comencemos con un ejemplo sencillo para ganar intuición. Imagina que estás escribiendo un programa sencillo que calcula la suma de dos números y almacena el resultado en una variable. El programa podría verse así:

```
a = 5
b = 3
c = a + b
```

La semántica dinámica se refiere a lo que realmente sucede cuando este programa se ejecuta línea por línea. Es como si estuvieras siguiendo una receta de cocina y observando cada paso que toma el chef. Por ejemplo:

1. Primera línea `a = 5`:
   - **Acción:** El programa encuentra la instrucción que dice `a = 5`.
   - **Efecto:** El programa crea una variable llamada `a` y le asigna el valor 5.
   - **Estado actual:** Ahora tenemos `a` con valor 5 en nuestra memoria.
2. Segunda línea `b = 3`:

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

- **Acción:** El programa encuentra la siguiente instrucción `b = 3`.
- **Efecto:** El programa crea otra variable llamada `b` y le asigna el valor 3.
- **Estado actual:** Ahora tenemos `a = 5` y `b = 3` en nuestra memoria.

3. Tercera línea `c = a + b`:
   - **Acción:** El programa encuentra la instrucción `c = a + b`.
   - **Efecto:** El programa suma los valores de `a` y `b` (que son 5 y 3 respectivamente) y obtiene 8. Luego, crea una variable llamada `c` y le asigna este valor.
   - **Estado actual:** Ahora tenemos `a = 5`, `b = 3`, y `c = 8` en nuestra memoria.

La semántica dinámica es como seguir el flujo de la ejecución del programa y ver cómo cada línea de código cambia el estado del programa (la memoria y los valores de las variables) paso a paso. Nos ayuda a entender exactamente qué hace el programa cuando se ejecuta.

Es una manera de describir el comportamiento del programa en términos de cómo las instrucciones afectan el estado de un sistema en tiempo de ejecución. En este caso, hemos visto cómo el estado del programa cambia al asignar valores a variables y realizar operaciones con esas variables.

Entender la semántica dinámica es importante porque:

- Nos permite prever cómo se comportará un programa cuando se ejecute.
- Facilita la detección y corrección de errores.
- Ayuda en la optimización del código.
- Es fundamental para el diseño de nuevos lenguajes de programación y herramientas de desarrollo.

2. Formas de Especificar la Semántica Dinámica

Existen distintas técnicas que permiten especificar formalmente a la semántica dinámica. En esta sección revisaremos las tres técnicas más populares:

Semántica denotativa

Es un enfoque formal para definir el significado de los programas en términos matemáticos, asociando cada construcción del lenguaje con una función matemática que describe su comportamiento. En lugar de describir cómo se ejecutan los programas paso a paso, la semántica denotativa mapea cada elemento del lenguaje a su denotación o valor abstracto en algún dominio matemático. Este enfoque permite especificar la semántica dinámica de manera abstracta y general, proporcionando una manera de entender y razonar sobre el comportamiento de los programas a través de funciones matemáticas que capturan su efecto global sin necesidad de detallar cada paso de la ejecución.

Ejemplo 1

Supongamos que tenemos un pequeño programa que suma dos números y devuelve el resultado:

```
def suma(a,b):
    return a + b
```

La semántica denotativa busca capturar el significado de este programa en términos de una función matemática. En lugar de describir cómo se ejecuta el programa línea por línea, simplemente decimos qué resultado produce dada una entrada específica.

Usaremos la notación clásica de semántica denotativa $[\![\cdot]\!]$ para representar el significado de las construcciones del lenguaje.

Definición del programa:

- `suma(a, b)` toma dos números, `a` y `b`, y devuelve su suma.

Interpretación matemática

- Usamos $[\![\texttt{suma(a,b)}]\!]$ para denotar la semántica del programa `suma`.
- En este caso, la función denotativa puede ser representada como: $[\![\texttt{suma(a,b)}]\!] = \texttt{a} + \texttt{b}$

Si llamamos a la función `suma` con los valores 3 y 5, el programa devuelve 8. En términos de semántica denotativa esto se representa como:

$$[\![\texttt{suma(3,5)}]\!] = 3 + 5 = 8$$

Este enfoque es valioso porque:

- Permite especificar el comportamiento de los programas de manera precisa y concisa.
- Facilita la verificación formal y el razonamiento sobre los programas.
- Es útil para la optimización y transformación del código, ya que las funciones matemáticas pueden ser manipuladas y analizadas de manera rigurosa.

En esencia, la semántica denotativa proporciona una manera clara y abstracta de entender qué hace un programa, centrándose en el qué en lugar del cómo, y la notación $[\![\cdot]\!]$ formaliza esta relación. Su estudio se profundiza en el curso de Semántica y Verificación.

Semántica Axiomática (Lógica Hoare)

La semántica axiomática es un enfoque para definir el significado de los programas basado en la lógica formal y las condiciones que deben cumplirse antes y después de la ejecución de los constructores del lenguaje. Utiliza afirmaciones para especificar las propiedades del estado del programa en ciertos puntos clave, particularmente antes y después de la ejecución de un bloque de código. Este método se centra en verificar la corrección de programas mediante el uso de precondiciones y postcondiciones, formuladas como tripletas de Hoare $\{P\}C\{Q\}$, donde P es la precondición, C es el comando, y Q la postcondición. La semántica axiomática es una forma de especificar la semántica dinámica porque proporciona una manera formal y lógica de razonar sobre el comportamiento del programa y las garantías que debe cumplir durante su ejecución, asegurando que las afirmaciones sobre los estados se mantengan válidas.

Ejemplo 2

Supongamos que tenemos un fragmento de código que incrementa una variable `x`:

```
x = x + 1
```

Queremos razonar sobre este código usando las tripletas de Hoare para asegurarnos de que se cumple cierta propiedad antes y después de la ejecución del fragmento de código.

Por ejemplo, supongamos que queremos asegurarnos de que `x` sea mayor que 1 después de incrementar su valor. La tripleta de Hoare podría ser:

$$\{x > 0\}x = x + 1\{x > 1\}$$

- **Precondición** $x > 0$: Antes de ejecutar el comando, sabemos que `x` es mayor que `0`.
- **Comando** `x = x + 1`: Esta es la asignación que incrementa el valor de `x` en 1.
- **Postcondición** $x > 1$: Después de ejecutar el comando, queremos que `x` se mayor que 1.

En este caso, es claro que la terna de Hoare es verdadera, pues al sumarle 1 a nuestra variable, que es mayor que 0, siempre será mayor que 1. Sin embargo, al ser un sistema lógico existen otra reglas de análisis más sofisticadas para analizar expresiones o comandos más complicados como estructuras de repetición y decisión e incluso extensiones como la llamada Lógica de Separación.

El enfoque de la semántica axiomática es importante por varias razones:

- Verificación de la corrección de programas.
- Detección de errores
- Especificación precisa
- Optimización del código
- Modulado y reutilización
- Fundamento para herramientas automatizadas

Este enfoque es fundamental para garantizar la corrección, fiabilidad y eficiencia del software, proporcionando una base solida para el desarrollo y mantenimiento de sistemas complejos y críticos. Al igual que el enfoque denotativo, su estudio se profundiza en el curso de Semántica y Verificación así como en los cursos de Métodos Formales y Lógica Computacional II.

Semántica Operacional

La semántica operacional es un enfoque para definir el significado de los programas mediante la especificación de reglas de transición que describen cómo se ejecutan las construcciones del lenguaje en un entorno operativo. Estas reglas describen cómo los programas pasan de un estado a otro, capturando el comportamiento dinámico de la ejecución del programa. Es una forma de especificar la semántica dinámica porque se centra en el proceso de ejecución del programa, proporcionando una descripción detallada de cómo las acciones del programa afectan su estado en tiempo de ejecución.

Este es el enfoque que usaremos a lo largo de las notas siguientes de forma tal que necesitamos profundizar en su formalización y uso.

3. Especificación Formal mediante un Enfoque Operacional

Existen dos formas de especificar la semántica dinámica usando el enfoque operacional: (1) Semántica Estructural o de paso pequeño o (2) Semántica Natural o de paso grande. En la semántica estructural, cada transición entre estados del programa se descompone en pasos discretos y mínimos, lo que permite un análisis detallado del comportamiento del programa en cada etapa de la ejecución. Por otro lado en la semántica natural, las transiciones entre estados se realizan en pasos más amplios, donde una única reducción puede representar una secuencia de pasos más compleja en la ejecución del programa. La semántica de paso pequeño proporciona una visión más detallada y precisa del proceso de ejecución, mientras que la semántica de paso grande puede ser más compacta y general, simplificando el análisis en algunos casos pero perdiendo detalles importantes en otros.

Estudiaremos ambos estilos en esta sección. Como puedes observar algo que comparten ambos estilos de semántica operacional es el uso de transiciones, por lo que necesitamos antes que nada formalizar esta noción.

Sistemas de Transición

Definición 1: Sistema de Transición

*Un **sistema de transición** se define como una tupla (E, δ, I), donde:*

- *E es un conjunto finito de estados.*
- *$\delta \subseteq E \times E$ es una relación de transición entre estados, que describe cómo un estado puede evolucionar a otro.*
- *I es un conjunto de estados iniciales, que especifica los estados desde los cuales puede comenzar la ejecución del sistema.*

Un sistema de transición modela el comportamiento dinámico de un sistema, proporcionando una representación formal de cómo evoluciona a lo largo del tiempo a partir de ciertos estados iniciales.

Observación 1

Un par de observaciones sobre la Definición 1:

- Un sistema de transición no incluye explícitamente los estados finales porque está diseñando para modelar la evolución dinámica del sistema en términos de transiciones y estados, sin enfocarse en un estado final específico. La inclusión de estados finales no es necesaria para capturar la dinámica del sistema, ya que la relación de transición δ describe todas las posibles transiciones entre estados, independientemente de si algún estado se considera final o no. En muchos casos, los sistemas pueden evolucionar continuamente a través de una secuencia infinita de estados sin llegar a un estado final en particular, por lo que no tiene sentido definir explícitamente un conjunto de estados finales en el sistema de transición. En cambio, la atención se centra en modelar las transiciones entre estados y los estados iniciales desde los cuales puede comenzar la ejecución del sistema.
- Puede haber varios estados iniciales en un sistema de transición para modelar situaciones en las que la ejecución del sistema puede comenzar desde diferentes puntos de partida. Esto es común en sistemas distribuidos, concurrentes o no deterministas, donde el comportamiento

inicial del sistema puede depender de condiciones iniciales diversas o contextos variables. Al permitir varios estados iniciales, se ofrece flexibilidad para representar diferentes escenarios de inicio de la ejecución del sistema. Esto es fundamental para capturar la complejidad de sistemas del mundo real, donde la ejecución puede iniciar desde múltiples condiciones iniciales posibles.

Un sistema de transición puede modelar diversos sistemas dinámicos, por ejemplo:

Semáforo

- **Estados:** Los estados podrían representar los diferentes colores del semáforo (rojo, amarillo, verde), así como el tiempo transcurrido en cada estado.
- **Estados iniciales:** El estado inicial podría ser el semáforo en rojo, indicando que el tráfico debe detenerse.
- **Relación de transición:** La relación describiría cómo el semáforo cambia de un estado a otro con el paso del tiempo o en respuesta a señales externas, como presionar un botón de cruce peatonal.

Control de temperatura en un horno

- **Estados:** Los estados podrían representar diferentes temperaturas del horno, así como el tiempo transcurrido en cada temperatura.
- **Estados iniciales:** El estado inicial podría ser una temperatura ambiente.
- **Relación de transición:** La relación de transición describiría cómo cambia la temperatura del horno en respuesta a las acciones del termostato o a la temperatura y cierre de la puerta del horno.

Sistemas de Transición y Lenguajes de Programación

Para modelar un lenguaje de programación como sistema de transición, los componentes se definen de la siguiente manera:

- **Estados:** Los estados representan los diferentes estados del programa mediante su ejecución. Cada estado podría incluir información sobre la configuración actual del programa, como el estado de las variables, la posición del contador del programa, etc. Por ejemplo, un estado podría representar el programa en un punto específico de su ejecución, con ciertos valores asignados a las variables y la posición actual del programa.
- **Estados iniciales:** Los estados iniciales son los puntos de partida desde los cuales puede comenzar la ejecución del programa. Estos estados representan el estado del programa al inicio de su ejecución. En un lenguaje de programación, los estados iniciales podrían ser diferentes configuraciones iniciales del programa antes de comenzar su ejecución. Por ejemplo, un estado inicial podría representar el programa justo antes de la ejecución principal, con todas las variables inicializadas a ciertos valores.
- **Estados finales:** En un lenguaje de programación, puede ser conveniente especificar estados finales en el sistema de transición en ciertos casos. Dado que la mayoría de lenguajes que modelaremos lo necesita, especificaremos siempre este componente y será definido como un conjunto de estados que representan los puntos de término o finalización de la ejecución del programa. Estos estados podrían incluir:

1. El estado final después de la ejecución exitosa de un programa.
2. Estados finales después de lanzar una excepción o detectar un error.
3. Estados finales que indican la finalización de una función o un método específico.
4. Estados finales que representan el final de un ciclo de ejecución.

En este caso los estados finales no necesariamente son subconjunto de E como veremos en otra nota.

- **Relación de transición:** La relación de transición describe cómo el programa evoluciona de un estado a otro durante su ejecución. Esta función modela las posibles transiciones entre los estados del programa en función de las instrucciones que se ejecutan. En un lenguaje de programación, usualmente se manejan funciones de transición que podrían definir reglas para cada tipo de instrucción del lenguaje, especificando cómo cambia el estado del programa después de ejecutar una instrucción particular. Por ejemplo, para una instrucción de asignación, la función de transición podría describir cómo se actualiza el valor de una variable en el estado del programa después de la asignación.

Ejemplo 3: Un sistema de transición para MiniLisp

Estados: $E = \{a \mid a \ \mathsf{ASA}\}$, es decir los estados del sistema son las expresiones bien formadas del lenguaje en sintaxis abstracta. Aquí el punto de conexión entre sintaxis y semántica.

Estados iniciales: $I = \{a \mid a \ \mathsf{ASA}\}$, en este caso $E = I$, pues podemos partir de cualquier expresión bien formada.

Estados finales: $F = \{Num(\mathtt{n}) \mid \mathtt{n} \in \mathbb{Z}\}$, para el caso de MiniLisp, en esta primera versión, los estados finales siempre serán números pues únicamente contamos con expresiones aritméticas y además $F \subseteq E$.

Relación de transición: La relación de transición se define de acuerdo al enfoque de la semántica operacional (natural o estructural) a partir de los estados previamente definidos. Las siguientes subsecciones definen ambos enfoques.

Semántica Natural

Como dijimos, la semántica natural consiste en describir cómo reducir las expresiones, se dice que es de paso grande pues los resultados son irreducibles, es decir, relaciona programas con sus estados finales. Para el caso de nuestro sistema de transición, relaciona un estado e con nuestro único estado final f, esto se denota como:

$$e \Rightarrow f$$

que se lee como *La expresión e se reduce a la expresión f.*

Al igual que con las reglas de sintaxis, usaremos reglas de inferencia para representar esta relación. Se definirá una regla por cada una de las construcciones del lenguaje, es decir, cada uno de los tipos de ASA que definimos en la nota pasada. Añadiremos una explicación de cada una de éstas.

Números

Los ASA que representan números se reducen a sí mismos. Este comportamiento es usual en muchos lenguajes de programación, tales como Python, Racket o Haskell.

$$\frac{}{Num(n) \Rightarrow Num(n)}$$

Expresiones aritméticas

Los ASA que representan sumas se reducen a aplicar la suma de la evaluación del lado izquierdo y derecho de las mismas.

$$\frac{i \Rightarrow Num(n_1) \qquad d \Rightarrow Num(n_2)}{Add(i, d) \Rightarrow Num(n_1 + n_2)}$$

La conclusión es cierta si i_f y d_f son las reducciones del lado izquierdo y derecho de la suma respectivamente.

El caso de la resta es análogo:

$$\frac{i \Rightarrow Num(n_1) \qquad d \Rightarrow Num(n_2)}{Sub(i, d) \Rightarrow Num(n_1 - n_2)}$$

Ejemplo 4

Derivación de algunas expresiones usando las reglas de semántica natural. Mostraremos el proceso integrando todos los conceptos estudiados hasta el momento.

- Sintaxis concreta: `1729`
- Sintaxis abstracta: $Num(\texttt{1729})$
- Evaluación:

$$\frac{}{Num(\texttt{1729}) \Rightarrow Num(\texttt{1729})}$$

- Sintaxis concreta: `(+ 18 35)`
- Sintaxis abstracta: $Add(Num(\texttt{18}), Num(\texttt{35}))$
- Evaluación:

$$\frac{Num(\texttt{35}) \Rightarrow Num(\texttt{35}) \qquad Num(\texttt{18}) \Rightarrow Num(\texttt{18})}{Add(Num(\texttt{18}), Num(\texttt{35})) \Rightarrow Num(\texttt{53})}$$

- Sintaxis concreta: `(+ (- 4 0) (- 5 5))`
- Sintaxis abstracta: $Add(Sub(Num(\texttt{4}), Num(\texttt{0})), Sub(Num(\texttt{5}), Num(\texttt{5})))$
- Evaluación:

$$\frac{\dfrac{Num(\texttt{4}) \Rightarrow Num(\texttt{4}) \qquad Num(\texttt{0}) \Rightarrow Num(\texttt{0})}{Sub(Num(\texttt{4}), Num(\texttt{0})) \Rightarrow Num(\texttt{4})} \qquad \dfrac{Num(\texttt{5}) \Rightarrow Num(\texttt{5}) \qquad Num(\texttt{5}) \Rightarrow Num(\texttt{5})}{Sub(Num(\texttt{5}), Num(\texttt{5})) \Rightarrow Num(\texttt{0})}}{Add(Sub(Num(\texttt{4}), Num(\texttt{0})), Sub(Num(\texttt{5}), Num(\texttt{5}))) \Rightarrow Num(\texttt{4})}$$

Semántica Estructural

Se le conoce también como semántica de paso pequeño o de transición y como dijimos. describe paso a paso la ejecución mostrando los cómputos que genera en cada paso individualmente. Para modelar esta noción, definimos la siguiente relación:

$$e_1 \to e_2$$

que a diferencia de la semántica natural, no relaciona necesariamente expresiones con su valor final, si no que relaciona expresiones con expresiones (pudiendo ser estados finales). Se lee como *en un paso* e_1 *se reduce a* e_2.

La definición de reglas en este tipo de semántica, es más complicada pues se tienen que considerar pasos intermedios que en ocasiones pasan desapercibidos al ejecutar o analizar un programa. Veamos las reglas:

Números

El comportamiento de los números el mismo que el de la semántica de paso grande, pues no hay subexpresiones que se puedan continuar reduciendo.

$$\frac{}{Num(n) \to Num(n)}$$

Expresiones aritméticas

Las sumas se dividen en tres reglas para los siguientes casos respectivamente.

- **Caso 1:** Ninguno de los argumentos de la sima ha sido reducido a un valor. El paso siguiente consiste en reducir el lado izquierdo. Por ejemplo, si tenemos `(+ (+ 3 4) (+ 5 2))` un paso de evaluación corresponde a evaluar `(+ 3 4)`.

$$\frac{i \to i'}{Add(i, d) \to Add(i', d)}$$

- **Caso 2:** El argumento izquierdo de la suma ha sido reducido a un número pero el derecho no. El paso siguiente es proceder con el lado derecho. Por ejemplo, si tenemos `(+ 7 (+ 5 2))`, el lado izquierdo ya es un número, por lo que procedemos con el lado derecho `(+ 5 2)`.

$$\frac{d \to d'}{Add(Num(n_1), d) \to Add(Num(n_1), d')}$$

- **Caso 3:** Los dos argumentos de la suma han sido reducidos a un número. El paso siguiente es realizar la suma tal cual. Por ejemplo si tenemos `(+ 7 7)`,ambos argumentos ya son un número, por lo que procedemos a sumar los mismos obteniendo 14.

$$\frac{}{Add(Num(n_1), Num(n_2)) \to Num(n_1 + n_2)}$$

El caso de la resta es análogo:

$$\frac{i \to i'}{Sub(i, d) \to Sub(i', d)}$$

$$\frac{d \to d'}{Sub(Num(n_1), d) \to Sub(Num(n_1), d')}$$

$$\frac{}{Sub(Num(n_1), Num(n_2)) \to Num(n_1 - n_2)}$$

Ejemplo 5

Derivación de algunas expresiones usando las reglas de semántica natural. Mostraremos el proceso integrando todos los conceptos estudiados hasta el momento.

- Sintaxis concreta: `1729`
- Sintaxis abstracta: $Num(\texttt{1729})$
- Evaluación:

  $Num(\texttt{1729})$
  $\to Num(\texttt{1729})$

- Sintaxis concreta: `(+ 18 35)`
- Sintaxis abstracta: $Add(Num(\texttt{18}), Num(\texttt{35}))$
- Evaluación:

  $Add(Num(\texttt{18}), Num(\texttt{35}))$
  $\to Num(\texttt{18} + \texttt{35}) = Num(\texttt{53})$

- Sintaxis concreta: `(+ (- 4 0) (- 5 5))`
- Sintaxis abstracta: $Add(Sub(Num(\texttt{4}), Num(\texttt{0})), Sub(Num(\texttt{5}), Num(\texttt{5})))$
- Evaluación:

  $Add(Sub(Num(\texttt{4}), Num(\texttt{0})), Sub(Num(\texttt{5}), Num(\texttt{5})))$
  $\to Add(Num(\texttt{4}), Sub(Num(\texttt{5}), Num(\texttt{5})))$
  $\to Add(Num(\texttt{4}), Num(\texttt{0}))$
  $\to Num(\texttt{4})$

4. Equivalencia entre la Semántica Natural y la Estructural

No importa qué estilo de semántica usemos para especificar las reglas de evaluación del lenguaje de programación, deben producir los mismos resultados finales, de forma tal que deben ser equivalente.

De forma intuitiva, lo que debe suceder es que si realizamos múltiples aplicaciones de la relación $\to$ de paso pequeño, en algún momento podemos llegar al mismo estado final que modela la relación de paso grande $\Rightarrow$. Definimos entonces la cerradura transitiva y reflexiva de $\to$:

Definición 2: Cerratura transitiva y reflexiva de $\to$

Se denota la cerradura transitiva y reflexiva de $\to$ por $\to^\star$ y se define con las siguientes dos reglas:

$$\frac{}{e \to^\star e}$$

$$\frac{e1 \to e_2 \qquad e2 \to^\star e_3}{e1 \to^\star e_3}$$

De forma intuitiva, la definición anterior modela que es posible llegar de una expresión a otra en un número finito de pasos de la relación de transición $\to$ (posiblemente cero).

De esta forma tenemos el siguiente teorema:

Teorema 1: Equivalencia entre la Semántica Natural y la Estructural

Para cualquier expresión e de MiniLisp y cualquier estado final f, se cumple:

$$e \to^\star f \text{ si y sólo si } e \Rightarrow f$$

es decir, la semántica natural es equivalente a la estructural.

5. Propiedades de MiniLisp

El hecho de haber formalizado nuestro lenguaje mediante estas relaciones de transición, nos permite demostrar propiedades importante sobre el lenguaje diseñado. Por ejemplo:

- Demostrar que el lenguaje es determinista, es decir que si parto de un mismo programa no obtendré dos evaluaciones distintas.
- Demostrar que todo programa del lenguaje termina.

Ambas propiedades se modelan en términos de nuestra relación de transición $\to$ a continuación:

Proposición 1: Determinismo de MiniLisp

Si $e \to e_1$ *y* $e \to e_2$ *entonces* $e_1 = e_2$.

Proposición 2: Terminación de MiniLisp

Para cada expresión e de MiniLisp existe un estado final f *tal que* $e \to^\star f$.

6. Conclusión

La semántica dinámica permite razonar con precisión sobre la ejecución de los programas, mostrando cómo cada instrucción transforma el estado del sistema. Al comparar los distintos enfoques (denotativo, axiomático y operacional) se observa que cada uno aporta herramientas valiosas para especificar, verificar y optimizar el comportamiento de un lenguaje. La formalización mediante (sistemas de transición) y el análisis en estilos de paso grande y paso pequeño ofrecen una visión complementaria entre resultados finales y procesos intermedios. Sin embargo, en este curso priorizaremos el uso de la semántica estructural, ya que

su carácter de paso pequeño nos permite observar con detalle el flujo de ejecución y construir un marco más sólido para demostrar propiedades como el determinismo y la terminación, esenciales para garantizar la confiabilidad de los lenguajes de programación.

Referencias

[1] M. Gabbrielli y S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2023.
[2] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. Brown University, 2007.
[3] K. Lee, *Foundations of Programming Languages*. Springer, 2017.
[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.