Lenguajes de Programación 2022-1
Nota de clase 15: Featherweight Java
Orientación a Objetos

Javier Enríquez Mendoza

15 de enero de 2022

El objetivo de esta nota de clase es modelar algunos aspectos del paradigma de programación orientado a objetos, mediante el estudio de un lenguaje de estudio llamado Featherweight Java que es un subconjunto del lenguaje Java.

Como pudimos observar con el lenguaje TinyC , la formalización de lenguajes de programación imperativos resulta una tarea más compleja de lo que es en el estilo funcional, esto se debe principalmente a lo separados que se encuentran los lenguajes imperativos de sistemas matemáticos en contraste con los lenguajes funcionales que tienen una estrecha relación con la teoría que los fundamenta.

Para lograr estudiar los conceptos de Java no existe un modelo perfecto, en su mayoría los modelos teóricos de lenguajes orientados a objetos son malos para abstraer el comportamiento completo del lenguaje. Sin embargo, estos modelos son útiles si lo que se busca es abstraer un conjunto especifico de propiedades del lenguaje.

En nuestro caso lo que buscamos es modelar las características principales del paradigma orientado a objetos así como su sistema de tipos. Para este propósito Featherweight Java resulta bastante útil. Las principales características que modela Featherweight Java son:

- Clases.
- Objetos.
- Métodos.
- Atributos y acceso a ellos.
- Herencia.
- Polimorfismo de subtipado.
- Recursión abierta.

Sin embargo se dejan fuera del modelo otros conceptos interesantes como lo son:

- Concurrencia.
- Interfaces.

- Manejo de accesos.
- Referencias.
- Clases internas.
- Excepciones.

Los cuales se pueden estudiar con otros modelos del paradigma orientado a objetos, que abstraen específicamente estas características.

1 Sintaxis

En esta sección se presenta la sintaxis concreta de Featherweight Java . Así como algunas convenciones sintácticas importantes que se deben tomar en cuenta al escribir un programa en el lenguaje.

Definición 1.1 (Sintaxis de Featherweight Java). Se presenta la sintaxis del lenguaje separado en categorías y usando las siguientes metavariables:

- Nombres de variables: `x,y,z`
- Nombres de atributos: `f,g`
- Nombres de clases: `C,A,B`
- Nombres de métodos: `m`

Expresiones

$$\begin{array}{rcll} \texttt{e} & ::= & \texttt{x} & \text{Variables} \\ & | & \texttt{e.f} & \text{Acceso a atributo} \\ & | & \texttt{e.m}(\vec{\texttt{e}}) & \text{Invocación de método} \\ & | & \texttt{new C}(\vec{\texttt{e}}) & \text{Instancia de objeto} \\ & | & (\texttt{C})\ \texttt{e} & \text{Casting} \end{array}$$

Valores

$$\texttt{v} ::= \texttt{new C}(\vec{\texttt{v}}) \quad \text{Instancia de objeto}$$

Métodos y Clases

$$\begin{array}{rcll} \texttt{K} & ::= & \texttt{C}\,(\vec{\texttt{C}}\,\vec{\texttt{x}})\ \{\ \texttt{super}(\vec{\texttt{x}}); \texttt{this.}\vec{\texttt{f}} = \vec{\texttt{x}}\ \} & \text{Constructores} \\ \texttt{M} & ::= & \texttt{C m}\,(\vec{\texttt{C}}\,\vec{\texttt{x}})\ \{\ \texttt{return e};\ \} & \text{Métodos} \\ \texttt{L} & ::= & \texttt{class C extends B}\ \{\ \vec{\texttt{A}}\,\vec{\texttt{f}}\,;\ \texttt{K}\,\vec{\texttt{M}}\ \} & \text{Clases} \end{array}$$

En la definición anterior se usa constantemente la notación $\vec{\texttt{t}}$ que no habíamos empleado anteriormente. Ésta recibe el nombre de notación vectorial y es una forma de simplificar la notación multiparamétrica que usamos en notas anteriores. Un vector denota la sucesión de expresiones

$$\vec{\texttt{t}} =_{def} \texttt{t}_1, \texttt{t}_2, \ldots, \texttt{t}_\texttt{n}$$

También se pueden presentar dos vectores juntos sin comas o puntos y coma separándolos, se entiende que ambos vectores tienen la misma longitud y en tal caso esta combinación denota a la sucesión obtenida tomando un elemento de cada vector a la vez, es decir

$$\vec{\texttt{C}}\,\vec{\texttt{x}} =_{def} \texttt{C}_1\,\texttt{x}_1, \texttt{C}_2\,\texttt{x}_2, \ldots, \texttt{C}_\texttt{n}\,\texttt{x}_\texttt{n}$$

En el caso de la definición de el constructor el vector empleado significa:

$$\texttt{this.}\vec{\texttt{f}} = \vec{\texttt{x}} =_{def} \texttt{this.f}_1 = \texttt{x}_1; \texttt{this.f}_2 = \texttt{x}_2; \ldots; \texttt{this.f}_n = \texttt{x}_n$$

de igual forma se entiende que ambos vectores tienen la misma longitud.

De la definición de la sintaxis podemos abstraer algunas convenciones importantes que se deben seguir en los programas en Featherweight Java y que no necesariamente se siguen en los programas en Java. A continuación las listamos:

- En la definición de una clase se incluye siempre la superclase, aún cuando ésta es `Object`.
- Los constructores siempre se declaran explícitamente aún cuando estos son triviales o cuando el comportamiento es el esperado por Java y tienen el mismo nombre que la clase.
- Dentro del constructor siempre se llama al constructor de la superclase mediante la instrucción `super`.
- Siempre se menciona el objeto al invocar un método o acceder a un atributo aún cuando éste es `this`.
- Los métodos consisten únicamente de una expresión `return`.
- Todos los constructores funcionan de forma genérica, es decir, reciben el mismo número de parámetros como atributos tenga la clase, hacen la asignación de los mismos y no pueden hacer nada más.
- Se puede definir un único constructor por clase.
- En la definición de las clases los atributos solo son declarados, es decir, no podemos asignar valores por defecto. Solo toman valor en el constructor.

Veamos a continuación algunos ejemplos de programas escritos en Featherweight Java .

Ejemplo 1.1 (La clase `Pair`)**.** Para ejemplificar la sintaxis se define un programa para manejo de pares.

```
class A extends Object { A () { super(); } }

class B extends Object { B () { super(); } }

class Pair extends Object {
    Object fst;
    Object snd;

    Pair (Object fst, Object snd){
        super();
        this.fst = fst;
        this.snd = snd;
    }

    Pair setfst (Object newfst){
        return new Pair(newfst, this.snd);
    }
```

```
}
```

Con las definiciones anteriores la expresión:

```
new Pair(new A (), new B ()).setfst(new B ())
```

se evaluaría intuitivamente a la expresión:

```
new Pair(new B (), new B ())
```

Ejemplo 1.2 (Árboles binarios en Featherweight Java). Se define un programa para manejo de árboles binarios con información solo en las hojas.

```
class Node extends Object {
    Object left;
    Object right;

    Node (Object left, Object right) {
        super();
        this.left = left;
        this.right = right;
    }

    Nat sum () {
        return (this.left.sum() + this.right.sum());
    }
}

class Leaf extends Object {
    Nat value;

    Leaf (Nat value) {
        super();
        this.value = value;
    }

    Nat sum() {
        return this.value;
    }
}
```

Para este ejemplo se supone definido el tipo `Nat` que modela los números naturales y la operación de suma para los elementos de este tipo.

1.1 Tablas de Clases

Un programa en JPP consta de una expresión e junto con una tabla de clases T la cual generalmente no se menciona explícitamente.

Definición 1.2 (Tabla de Clases)**.** Una tabla de clases es una función finita que asigna clases a nombres de clase, en otras palabras, T es una sucesión finita de declaraciones de clase de la forma

$$T(\mathtt{C}) = \mathtt{class\ C\ extends\ B\ \{\ } \vec{\mathtt{A}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\}$$

Veamos ahora como acceder a la información almacenada en esta tabla.

Búsqueda de Atributos

$$\frac{}{\mathsf{fields}(\mathtt{Object}) = \varnothing}$$

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ B\ \{\ } \vec{\mathtt{C}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\} \qquad \mathsf{fields}(\mathtt{B}) = \vec{\mathtt{B}}\,\vec{\mathtt{g}}}{\mathsf{fields}(\mathtt{C}) = \vec{\mathtt{B}}\,\vec{\mathtt{g}}, \vec{\mathtt{C}}\,\vec{\mathtt{f}}}$$

Búsqueda del tipo de un método

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ } \vec{\mathtt{C}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\} \qquad \mathtt{B\ m}\,(\vec{\mathtt{B}}, \vec{\mathtt{x}})\ \mathtt{\{\ return\ e;\ \}}\ \text{figura en}\ \vec{\mathtt{M}}}{\mathsf{mtype}(\mathtt{m}, \mathtt{C}) = \vec{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ } \vec{\mathtt{C}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\} \qquad \mathtt{m}\ \text{no figura en}\ \vec{\mathtt{M}}}{\mathsf{mtype}(\mathtt{m}, \mathtt{C}) = \mathsf{mtype}(\mathtt{m}, \mathtt{D})}$$

En estas reglas la notación vectorial $\vec{\mathtt{B}} \to \mathtt{B}$ se entiende como $\mathtt{B}_1 \to \mathtt{B}_2 \to \cdots \to \mathtt{B}$.

Búsqueda del cuerpo de un método

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ } \vec{\mathtt{C}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\} \qquad \mathtt{B\ m}\,(\vec{\mathtt{B}}, \vec{\mathtt{x}})\ \mathtt{\{\ return\ e;\ \}}\ \text{figura en}\ \vec{\mathtt{M}}}{\mathsf{mbody}(\mathtt{m}, \mathtt{C}) = (\vec{\mathtt{x}}, \mathtt{e})}$$

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ } \vec{\mathtt{C}}\,\vec{\mathtt{f}}\mathtt{;\ K}\,\vec{\mathtt{M}}\mathtt{\ }\} \qquad \mathtt{m}\ \text{no figura en}\ \vec{\mathtt{M}}}{\mathsf{mbody}(\mathtt{m}, \mathtt{C}) = \mathsf{mbody}(\mathtt{m}, \mathtt{D})}$$

2 Semántica Dinámica

En esta sección se presenta la semántica dinámica de Featherweight Java mediante una semántica operacional estructural.

Definición 2.1 (Semántica Operacional de Featherweight Java)**.** Se define la semántica como un sistema de transición con función de transición $\longrightarrow_{\mathsf{FJ}}$ que depende de la tabla de clases T de un programa, esta tabla se omite en las reglas del sistema, sin embargo siempre está presente. Recordemos que los únicos valores del lenguaje están definidos por la instrucción `new`.

Selección de Atributos Como los parámetros del constructor corresponden exactamente con

los atributos definidos en las clases, podemos obtener directamente el valor de cada uno a partir de la instancia del objeto. Modelado en la siguiente regla:

$$\frac{\mathsf{fields}(\mathtt{C}) = \vec{\mathtt{C}}\,\vec{\mathtt{f}}}{(\mathtt{new}\,\mathtt{C}\,(\vec{\mathtt{v}})).\mathtt{f_i} \longrightarrow_{\mathtt{FJ}} \mathtt{v_i}}$$

Invocación de métodos

$$\frac{\mathsf{mbody}(\mathtt{m}, \mathtt{C}) = (\vec{\mathtt{x}}, \mathtt{e})}{(\mathtt{new}\,\mathtt{C}\,(\vec{\mathtt{v}})).\mathtt{m}(\vec{\mathtt{w}}) \longrightarrow_{\mathtt{FJ}} e[\vec{\mathtt{x}}, \mathtt{this} := \vec{\mathtt{w}}, (\mathtt{new}\,\mathtt{C}\,(\vec{\mathtt{v}}))]}$$

En la regla anterior para la evaluación de una invocación de un método, se evalúa el cuerpo del método en donde se sustituyen todos los parámetros de éste por los argumentos de la llamada y se sustituye `this` por la instancia misma que hace la invocación.

Casting

$$\frac{\mathtt{C} \leq \mathtt{D}}{(\mathtt{D})\ (\mathtt{new}\,\mathtt{C}\,(\vec{\mathtt{v}})) \longrightarrow_{\mathtt{FJ}} \mathtt{new}\,\mathtt{C}\,(\vec{\mathtt{v}})}$$

Reglas de congruencia Como es usual la forma de evaluar es de derecha a izquierda lo que se modela con las siguientes reglas burocráticas:

$$\frac{\mathtt{e} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'}{\mathtt{e.f} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'.\mathtt{f}}$$

$$\frac{\mathtt{e} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'}{\mathtt{e.m}(\vec{\mathtt{e}}) \longrightarrow_{\mathtt{FJ}} \mathtt{e}'.\mathtt{m}(\vec{\mathtt{e}})}$$

$$\frac{\mathtt{e_i} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'_\mathtt{i}}{\mathtt{e.m}(\ldots, \mathtt{e_i}, \ldots) \longrightarrow_{\mathtt{FJ}} \mathtt{e.m}(\ldots, \mathtt{e}'_\mathtt{i}, \ldots)}$$

$$\frac{\mathtt{e_i} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'_\mathtt{i}}{\mathtt{new}\ \mathtt{C}(\ldots, \mathtt{e_i}, \ldots) \longrightarrow_{\mathtt{FJ}} \mathtt{new}\ \mathtt{C}(\ldots, \mathtt{e}'_\mathtt{i}, \ldots)}$$

$$\frac{\mathtt{e} \longrightarrow_{\mathtt{FJ}} \mathtt{e}'}{(\mathtt{C})\ \mathtt{e} \longrightarrow_{\mathtt{FJ}} (\mathtt{C})\ \mathtt{e}'}$$

3 Semántica Estática

Para la definición de la semántica estática se define un sistema de tipos para el lenguaje, en este caso los tipos son los nombres de las clases y no se tienen tipos primitivos, es decir, todos los tipos que se necesiten para un programa deben ser definidos dentro del mismo, a excepción claro de `Object`.

Definición 3.1 (Semántica Estática de Featherweight Java)**.** Para la definición de su sistema de tipos deben definirse las reglas de tipado de las expresiones, las reglas de subtipado y *casting* así como reglas que modelen cuando un método y una clase están bien formadas. A continuación se presentan todas estas definiciones.

Reglas de Subtipado

$$\frac{T(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{\ \vec{C}\,\vec{f}\,;\ K\,\vec{M}\ \}}}{\mathtt{C} \leq \mathtt{D}}$$

Es decir, si la clase C extiende a la clase D entonces C es subtipo de D.
Adicional a esta regla el subtipado debe cumplir las propiedades de transitividad y reflexividad presentadas en la nota de clase anterior.

Reglas de Tipado

$$\frac{}{\Gamma, x : \mathtt{C} \vdash x : \mathtt{C}} \quad \text{Variables}$$

$$\frac{\Gamma \vdash e : \mathtt{C} \qquad \mathsf{fields}(\mathtt{C}) = \vec{C}\,\vec{f}}{\Gamma \vdash e.\mathtt{f_i} : \mathtt{C_i}} \quad \text{Acceso a atributos}$$

$$\frac{\Gamma \vdash e_0 : \mathtt{C}_0 \qquad \Gamma \vdash \vec{e} : \vec{C} \qquad \mathsf{mtype}(\mathtt{m}, \mathtt{C}_0) = \vec{D} \to \mathtt{C} \qquad \vec{C} \leq \vec{D}}{\Gamma \vdash e_0.\mathtt{m}(\tilde{\mathtt{e}}) : \mathtt{C}} \quad \text{Invocación de métodos}$$

$$\frac{\Gamma \vdash \vec{e} : \vec{C} \qquad \mathsf{fields}(\mathtt{C}) = \vec{D}\,\vec{f} \qquad \vec{C} \leq \vec{D}}{\Gamma \vdash \mathtt{new\ C}(\vec{e}) : \mathtt{C}} \quad \text{Creación de objetos}$$

Reglas de casting

$$\frac{\Gamma \vdash e : \mathtt{D} \qquad \mathtt{D} \leq \mathtt{C}}{\Gamma \vdash (\mathtt{C})\ e : \mathtt{C}} \quad \textit{Upcasting}$$

$$\frac{\Gamma \vdash e : \mathtt{D} \qquad \mathtt{C} \leq \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\ e : \mathtt{C}} \quad \textit{Downcasting}$$

$$\frac{\Gamma \vdash e : \mathtt{D} \qquad \mathtt{C} \not\leq \mathtt{D} \qquad \mathtt{D} \not\leq \mathtt{C} \qquad \mathsf{stupid\ warning}}{\Gamma \vdash (\mathtt{C})\ e : \mathtt{C}} \quad \textit{Casting} \text{ estúpido}$$

Esta última regla es un tecnicismo necesario para poder probar la preservación de tipos con la semántica operacional estructural.

Formación de Clases Para definir las reglas que modelan cuándo una clase está bien formada, se definen los nuevos juicios $\mathtt{M\ ok\ in\ C}$ para indicar que el método M está bien formado en la clase C, $\mathtt{C\ ok}$ que indica que la clase Cestá bien formada y por último el juicio $T\ \mathtt{ok}$ para decir que la tabla de clases T está bien formada.

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}\,(\vec{D}\,\vec{y}, \vec{C}\,\vec{x})\ \{\ \mathtt{super}(\vec{y}); \mathtt{this.}\vec{f} = \vec{x}\ \} \\ \mathsf{fields}(\mathtt{D}) = \vec{D}\,\vec{g} \\ \vec{M}\ \mathsf{ok\ in\ C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\ \vec{C}\,\vec{f}\,;\ \mathtt{K}\,\vec{M}\ \}\ \mathtt{ok}}$$

Formación de métodos

$$\frac{\begin{array}{c}T(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\,\{\ldots\} \\ \mathsf{mtype}(\mathtt{m}, \mathtt{D}) = \vec{C} \to \mathtt{C}_0 \\ \vec{x} : \vec{C}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e} : \mathtt{C}_0' \\ \mathtt{C}_0' \leq \mathtt{C}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\,(\vec{C}\,\vec{x})\{\ \mathtt{return\ e};\ \}\ \mathtt{ok\ in\ C}}$$

En las premisas se está considerando la posibilidad de que el método `m` se haya redefinido (*overriding*) verificando que se respete el tipo dado en la superclase. Es decir, el cuerpo del método en la subclase debe devolver un subtipo del tipo del resultado del mismo método en la superclase. También se observa que los tipos de los parámetros del método deben ser exactamente los mismos que los de la superclase.
Es importante observar que si $\texttt{D} = \texttt{Object}$ entonces el método no se está redefiniendo puesto que `Object`no tiene métodos. En tal caso la premisa acerca de mtype debe ignorarse, siendo aceptadas cualesquiera $\texttt{C}_0$ y $\vec{\texttt{C}}$.

Formación de Tablas Decimos que una tabla de clases está bien formada si todas las clases declaradas en ella están bien formadas, modelado con la regla:

$$\frac{\forall \texttt{C} \in \mathsf{dom}(T), T(\texttt{C})\ \texttt{ok}}{T\ \texttt{ok}}$$

4 Seguridad de Featherweight Java

Al igual que con los sistemas anteriores estudiaremos la seguridad del lenguaje Featherweight Java con las propiedades de preservación de tipos y progreso de la función de transición, en este caso estas propiedades deben considerar la herencia así como el uso de *castings* inválidos.

Proposición 4.1 (Preservación de tipos)**.** Si T es una tabla de clases bien formada, $\Gamma \vdash e : \texttt{C}$ y $e \longrightarrow_{\texttt{FJ}} e'$, entonces existe $\texttt{C}'$ tal que $\texttt{C}' \leq \texttt{C}$ y $\Gamma \vdash e'\texttt{C}'$

Demostración. Consultar 1 □

Respecto a la propiedad de progreso esta vez debemos observar que un un programa bien tipado podría bloquearse debido a un uso indebido del casting, pero en general podemos probar que la única forma en la qu eun programa bien tipado puede bloquearse es si llega a un punto en el que no puede computar un *downcasting*.

Proposición 4.2 (Progreso de la relación $\longrightarrow_{\texttt{FJ}}$)**.** Sea T una tabla de clases bien formada, si $\varnothing \vdash e : \texttt{C}$ entonces sucede una y solo una de las siguientes condiciones:

- e es un valor.
- e contiene una expresión de la forma $(\texttt{C})\ \texttt{new}\ \texttt{D}(\vec{\texttt{v}})$ en donde $\texttt{D} \leq \texttt{C}$, es decir, no se puede realizar el *downcast*.
- Existe e' tal que $e \longrightarrow_{\texttt{FJ}} e'$.

Demostración. Consultar 1 □

Teorema 4.3 (Seguridad de Featherweight Java)**.** Si T es una tabla de clases bien formada, $\varnothing \vdash e : \texttt{C}$ y $e \longrightarrow^{\star}_{\texttt{FJ}} e'$ con e' en forma normal, entonces se cumple una y solo una de las siguientes condiciones:

- e' es un valor v tal que $\varnothing \vdash v : \texttt{D}$ y $\texttt{D} \leq \texttt{C}$.
- e' contiene como subexpresión $(\texttt{C})\ \texttt{new}\ \texttt{D}(\vec{\texttt{v}})$ en donde $\texttt{D} \leq \texttt{C}$.

En otras palabras un programa bien tipado siempre termina o bien tiene un *downcasting* que no se puede resolver.

Demostración. Directo de las proposiciones 4.1 y 4.2 y la propiedad de transitividad de la relación de subtipado. □

5 Correspondencia con Java

Como se menciono al inicio de esta nota, Featherweight Java es un modelo de algunas de las características principales del lenguaje de programación Java, principalmente diseñada para el estudio de la seguridad de su sistema de tipos. Sin embargo otras de las características de la programación orientada a objetos así como mucho del poder de expresividad de Java quedan fuera de este modelo.

Por esta razón es importante hacer explicita la correspondencia deseada entre un programa en Featherweight Java y su versión en Java. Esta correspondencia se da con los tres lemas siguientes:

Lema 5.1. Cada programa sintácticamente correcto en Featherweight Java es también sintácticamente correcto en Java.

Lema 5.2. Un programa sintácticamente correcto es tipable en Featherweight Java si y sólo si es tipable en Java.

Lema 5.3. La ejecución de un programa bien tipado en Featherweight Java se comporta de la misma forma en Java.

Como consecuencia del lema 5.3 tenemos el siguiente corolario.

Corolario 5.4. La evaluación de un programa en Featherweight Java no termina si y sólo si compilarlo y ejecutarlo en Java causa no terminación.

Por supuesto los lemas anteriores no pueden demostrarse ya que no existe una formalización completa del lenguaje Java. Sin embargo es muy útil enunciar la correspondencia de manera precisa para dejar en claro el objetivo que estamos tratando de alcanzar.

Referencias

[1] Igarashi A., Pierce B., Wadler P., Featherweight Java: A Minimal Core Calculus for Java and GJ, ACM-TRANSACTION, Enero 23 2002. `https://www.cis.upenn.edu/~bcpierce/papers/fj-toplas.pdf`

[2] Miranda Perea F., González Huesca L., Nota de Clase del curso de Lenguajes de Programación, Facultad de Ciencias UNAM, Semestre 2021-1.