Lenguajes de Programación, 2019-1*
Nota de clase 16: Paradigma Orientado a Objetos III**
Java Peso Pluma***

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

23 de noviembre de 2018

1. Introducción

El objetivo final del curso es modelar ciertos aspectos del lenguaje de programación Java en un sistema de tipos donde los conceptos fundamentales como clases y objetos sean primitivos. No existe un modelo perfecto puesto que deben abstraerse ciertos detalles, de manera que los modelos no son buenos o malos en general sino con respecto a un conjunto específico de propiedades que se desean estudiar en éllos. Diversos tipos de modelos son:

- Código fuente vs. código en bytes.
- incluyentes y grandes vs. específicos y pequeños.
- sistemas de tipos vs. características de ejecución
- extensiones.

En nuestro caso el próposito es modelar características fundamentales de la programación orientada a objetos y sus sistemas de tipos, nada más, para eso utilizaremos un modelo llamado *Java Peso Pluma*.[1] Este formalismo excluye diversas características de Java como:

- Clases internas, carga de clases, reflexión.
- Concurrencia, hilos.
- Excepciones, ciclos.
- Interfases, sobrecarga.
- Asignación (referencias)

*Esta nota fue editada del texto original para cumplir los propósitos del curso 2018-1
**Estas notas se basan en el libro de Pierce y en material de Eduardo Bonelli y Gerwin Klein
***Featherweight Java
[1] *Featherweight Java*, abreviado como FJ o en español JPP

Entonces ¿Qué nos queda?

- Clase y objetos
- Métodos e invocación a los mismos.
- Atributos y acceso a los mismos.
- Herencia.
- Recursión abierta mediante `this`
- Subtipado (*Casting*).

2. Formalizando a JPP

Antes de definir formalmente la sintaxis de JPP nos detenemos un momento a reflexionar acerca de lo que tenemos hasta ahora y lo que vamos a necesitar.

- Sintaxis: en comparación con el cálculo lambda tenemos
  - Variables: sí, por ejemplo atributos, argumentos para los métodos y variables locales,
  - Abstracciones lambda: sí, en la forma de estructuras de ligado como declaraciones de clase, de atributo, de métodos y en el manejo de variables locales.
  - Aplicaciones: sí, en la forma de invocaciones a métodos y denotación de atributos y métodos.
- Sistema de tipos: ¿qué clase de garantías necesitamos?
  - El objetivo principal debe ser que los objetos entiendan la invocación de métodos.
  - Subtipos: el sistema de tipos y subtipos es nominal, A es subtipo de B si A hereda de B. Esto sucede de manera explícita, la definición de A declara que A hereda de B, en contraste con los sistemas estructurales donde la relación de subtipos se especifica mediante reglas.
- Sintaxis adicional: necesitamos nuevos mecanismos sintácticos como:
  - Declaración de clases incluyendo nombre.
  - Declaración de métodos incluyendo nombre, argumentos y cuerpo.
  - Declaración de constructores incluyendo nombres de variables, llamada a superclase y asignación de atributos.
  - Devolución de expresiones.
  - Invocación de métodos
  - Búsqueda de atributos
  - Generación de objetos
- Valores: con respecto a la evaluación ¿ cuales serán los valores? Respuesta: los objetos.

Algunos ejemplos de la sintaxis descrita arriba y que definiremos formalmente a continuación, son las siguientes definiciones de clases en JPP:

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

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

class Par extends Object {
  Object fst;
  Object snd;

  Par(Object x, Object y) {
    super(); this.fst = x; this.snd =y; }

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

En estos ejemplos observamos algunas convenciones importantes:

- La superclase se incluye siempre, aún cuando es `Object`.
- El constructor se llama igual que la clase, se declara siempre después de los atributos aún cuando sea trivial (i.e. cuando no hay atributos).
- La variable `super` se llama siempre desde el constructor, aún cuando no se pase argumento alguno.
- Siempre se menciona el objeto receptor en una invocación a método o acceso a atributo, aún cuando sea `this`.
- Los métodos siempre consisten de una única expresión `return`
- Los constructores:
  - toman el mismo número y tipo de parámetros que los atributos de la clase.
  - asignan parametros de constructor a atributos locales.
  - llaman al constructor `super` para asignar los atributos restantes.
  - No hacen nada más

Escribimos ahora algunos ejemplo de definición de clases ya vistos en la nota anterior, en la sintaxis de Java Peso Pluma. Comenzamos con los árboles binarios con información en las hojas:

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

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

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


class Leaf extends Object {
 Nat value;

 Leaf(Nat v) { super(); this.value = v }
 Nat sum() { return this.value }
}
```

El siguiente ejemplo implementa un contador simple.

```
class Counter extends Object {
  Nat count;

  Counter(Nat n){ super(); this.count = n}

  Void inc() { return this.count = this.count + 1 }
  Void get() { return this.count }
}
```

2.1. Sistemas de tipos nominales

Del ejemplo anterior se observa la desaparición de las acostumbradas declaraciones de tipo. Esto se debe a que el sistema de tipos de JAVA es nominal y no estructural como en todos los sistema anteriores del curso. Esta dicotomia en sistemas de tipos puede describirse a grandes rasgos como sigue:

- Sistema de tipos estructural:
  - Lo importante acerca de un tipo es su estructura.
  - Los nombres son simples abreviaturas (azucar sintáctica).
  - El subtipado también es estructural.
  - Son más simples y elegantes
  - Fáciles de extender (todo el curso ha tratado de extensiones)
  - Al considerar tipos recursivos se pierde la simplicidad y elegancia en parte.
- Sistemas de tipos nominales:
  - Los tipos siempre se nombran.
  - El verificador de tipos manipula la mayor parte del tiempo los puros nombres y no las estructuras, esto causa que la verificación sea fácil y eficiente.

- El programador declara explícitamente la relación de subtipos, para ser verificado posteriormente por el compilador.
- Los nombres de tipos son útiles en tiempo de ejecución, por ejemplo para casting, test de tipos, reflexión, etc.

A continuación presentamos la sintaxis formal de JPP, de importancia es notar que la omisión de efectos imperativos de asignación causa un efecto lateral ventajoso: las únicas maneras en las que dos objetos difieren son mediante las clases de donde provienen o mediante los parámetros que se le pasan a sus constructores en el momento de creación. Esta información está disponible en el operador de generación de objetos `new` de manera que podemos identificar el objeto creado con la expresión `new`, esto indica que los valores del sistema deben ser objetos creados mediante `new`.

3. Sintaxis de JPP

- Nombres: se usan las siguientes metavariables para nombres:
  - Nombres de variables: `x,y,z,...`
  - Nombres de atributos: `f,g`
  - Nombres de clases: `C`, `A`, `B`, `D`, `E`
  - Nombres de métodos: `m`

  Suponemos que estas categorías de nombres son ajenas dos a dos. Esto es conveniente para el formalismo aunque no siempre adecuado en la práctica.

- Expresiones

$$
\begin{array}{lll}
\texttt{e} ::= & & \\
 & \texttt{x} \mid & -\textit{variable} \\
 & \texttt{e.f} \mid & -\textit{acceso a atributo} \\
 & \texttt{e.m}(\tilde{\texttt{e}}\ ) \mid & -\textit{invocacion a metodo} \\
 & \texttt{new C}(\vec{\texttt{e}}\ ) \mid & -\textit{creacion de objeto} \\
 & (\texttt{C})\ \texttt{e} \mid & -\textit{cast}
\end{array}
$$

- Valores

$$v ::= \texttt{new C}(\vec{\texttt{v}}\ ) \quad -\textit{creacion de objeto}$$

- Métodos y clases.

$$\texttt{K} \;::=\; \texttt{C}\,(\vec{\texttt{C}}\;\vec{\texttt{x}}\,)\;\{\texttt{super}(\vec{\texttt{x}}\,)\;;\texttt{this.}\vec{\texttt{f}} = \vec{\texttt{x}}\;;\} \quad -\textit{declaracion de constructor}$$

$$\texttt{M} \;::=\; \texttt{C m}\,(\vec{\texttt{C}}\;\vec{\texttt{x}}\,)\;\{\texttt{return e};\,\} \quad -\textit{declaracion de metodos}$$

$$\texttt{CL} \;::=\; \texttt{class C extends C}\;\{\vec{\texttt{C}}\;\vec{\texttt{f}}\;;\;\texttt{K}\;\vec{\texttt{M}}\,\} \quad -\textit{declaracion de clases}$$

- Notación vectorial
  Resulta adecuado aclarar el mecanismo de notación vectorial. Un vector $\vec{t}$ denota a una sucesión de expresiones $t_1, t_2, \ldots, t_n$ separadas por comas, en algunos casos por punto y coma $t_1; t_2; \ldots; t_n$ o inclusive sin signos de puntuación para separar $t_1\ t_2\ \ldots\ t_n$. Cuando figuran dos vectores seguidos, como en el caso $\vec{\texttt{C}}\;\vec{f}$ sin coma o punto 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

  $$\vec{\texttt{C}}\;\vec{f} =_{def} \texttt{C}_1\ f_1,\ \texttt{C}_2\ f_2, \ldots,\ \texttt{C}_n\ f_n$$

  En el caso de la definición de constructor $\texttt{this.}\vec{\texttt{f}} = \vec{\texttt{x}}$ significa:

  $$\texttt{this.}f_1 = x_1;\ \texttt{this.}f_2 = x_2; \ldots;\ \texttt{this.}f_n = x_n$$

Analicemos con detalle la sintaxis del siguiente programa, similar a un ejemplo anterior pero esta vez respetando la separación de nombres:

```
class Par extends Object {
  Object fst;
  Object snd;

  Par(Object x, Object y) {
    super(); this.fst = x; this.snd =y; }

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

  Par setsnd(Object newsnd) {
    return new Par(this.fst,newsnd); }
}
```

- Nombres de clase (C): `Par, Object`
- Nombres de atributo (f): `fst,snd`
- Nombres de método (m): `setfst, setsnd`
- Nombres de variable: `x,y,newfst,newsnd`

- Sucesión de atributos ($\vec{\texttt{C}}\ \vec{f}$): `Object fst; Object snd`
- Declaración de constructor (`K`):

```
Par(Object x, Object y) {
  super(); this.fst = x; this.snd =y; }
```

  - Nombre de clase (`C`): `Par`
  - Sucesión de parámetros del constructor ($\vec{\texttt{C}}\ \vec{x}$): `Object x, Object y`
  - Llamada a los atributos de la superclase (`super`($\vec{f}$): `super`(), la sucesión $\vec{f}$ es vacía pues la superclase es `Object` que no tiene atributos.
  - Atributos locales: (`this.`$\vec{f} = \vec{x}$): `this.fst` $= x$; `this.snd` $= y$

- Declaración de métodos ($\vec{\texttt{M}}$):

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

Par setsnd(Object newsnd) {
    return new Par(this.fst,newsnd); }
```

  - Nombres de clase (`C`): `Par,Object`
  - Nombres de método (`m`): `setfst, setsnd`
  - Sucesión de parámetros de método ($\vec{\texttt{C}}\ \vec{x}$): `Object newfst` y `Object newsnd`
  - Expresiones en el cuerpo de `return` ($\vec{e}$):

    `new Par(newfst, this.snd)` y `new Par(this.fst, newsnd)`

3.1. Tabla 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. Una tabla de clases T 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

$$T(\texttt{C}) = \texttt{class C extends D } \{\vec{\texttt{C}}\ \vec{f}\ ;\ \texttt{K}\ \vec{\texttt{M}}\ \}$$

Los siguientes juicios extraen información relevante de la tabla de clases:

- Búsqueda de atributos

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

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

- Búsqueda del tipo de un método.

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

  Aquí $\vec{\mathtt{B}} \to \mathtt{B}$ debe entenderse como $\mathtt{B}_1 \to \mathtt{B}_2 \ldots \to \mathtt{B}_n \to \mathtt{B}$. Obsérvese que esto es una simple convención ya que no hay tipos función en el sistema, los únicos tipos son las clases.

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

- Búsqueda del cuerpo de un método

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

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

4. Semántica Estática

Además de los juicios auxiliares anteriores necesitaremos los siguientes:

$\mathtt{T} \leq \mathtt{S}$	subtipado
$\Gamma \vdash e : \mathtt{C}$	tipado de expresiones
$\mathtt{M}\ \mathsf{ok\ in}\ \mathtt{C}$	método bien formado
$\mathtt{C}\ \mathsf{ok}$	clase bien formada
$T\ \mathsf{ok}$	tabla de clases bien formada

Se observa que los tipos son precisamente los nombres de clases. Analicemos con detalle cada juicio:

4.1. Subtipado

El subtipado debe ser reflexivo y transitivo como antes:

$$\frac{}{\mathtt{C} \leq \mathtt{C}}\ (\text{S-REFL}) \qquad \frac{\mathtt{C} \leq \mathtt{D} \quad \mathtt{D} \leq \mathtt{E}}{\mathtt{C} \leq \mathtt{E}}\ (\text{S-TRANS})$$

Las reglas primitivas de subtipado se dan explícitamente en la tabla de clases:

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

Es decir, si la clase `C` extiende a la clase `D` entonces `C` es subtipo (subclase) de `D`

4.2. Tipado

Veamos como tipar cada forma de expresión:

- Variables: igual que en los otros sistemas, la única forma de tipar una variable es declarando su tipo en el contexto:

$$\frac{}{\Gamma, x : \mathtt{C} \vdash x : \mathtt{C}} \text{ (T-VAR)}$$

  donde x es un nombre de variable, el cual podría ser `this`.

- Acceso a atributos:

$$\frac{\Gamma \vdash e : \mathtt{C} \quad \mathsf{fields}(\mathtt{C}) = \vec{\mathtt{C}}\ \vec{f}}{\Gamma \vdash e.f_i : \mathtt{C}_i} \text{ (T-FLD)}$$

  $\mathsf{fields}(\mathtt{C})$ se encarga de atravesar la jerarquía de clases (en la tabla de clases) para encontrar el atributo apropiado.

- Invocación de métodos:

$$\frac{\Gamma \vdash e_0 : \mathtt{C}_0 \quad \Gamma \vdash \vec{e} : \vec{\mathtt{C}} \quad \mathsf{mtype}(\mathtt{m}, \mathtt{C}_0) = \vec{\mathtt{D}} \to \mathtt{C} \quad \vec{\mathtt{C}} \leq \vec{\mathtt{D}}}{\Gamma \vdash e_0.\mathtt{m}(\vec{e}\,) : \mathtt{C}} \text{ (T-INV)}$$

  Aquí se observa la interacción de la semántica estática con la tabla de clases, así como la inclusión de la regla de subsunción de subtipos esto se conoce como subtipado algorítmico, y es justamente como JAVA lo hace.

- Creación de objetos

$$\frac{\Gamma \vdash \vec{e} : \vec{\mathtt{C}} \quad \mathsf{fields}(\mathtt{C}) = \vec{\mathtt{D}}\ \vec{f} \quad \vec{\mathtt{C}} \leq \vec{\mathtt{D}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\vec{e}) : \mathtt{C}} \text{ (T-NEW)}$$

  Se observa de nuevo la inclusión del subtipado. Además se deben pasar valores a todos los parámetros del constructor.

4.2.1. Reglas de *Casting*

- *Casting* hacia arriba

$$\frac{\Gamma \vdash e : \mathtt{D} \quad \mathtt{D} \leq \mathtt{C}}{\Gamma \vdash (\mathtt{C})\, e : \mathtt{C}} \text{ (T-UCAST)}$$

- *Casting* hacia abajo

$$\frac{\Gamma \vdash e : \mathtt{D} \quad \mathtt{C} \leq \mathtt{D} \quad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\, e : \mathtt{C}} \text{ (T-DCAST)}$$

  Se consideran dos reglas pues es así como JAVA implementa la operación de *casting*.

- *Casting* estúpido[2].

$$\frac{\Gamma \vdash e : \mathtt{D} \quad \mathtt{C} \not\leq \mathtt{D} \quad \mathtt{D} \not\leq \mathtt{C} \quad \textit{enviar advertencia estúpida}}{\Gamma \vdash (\mathtt{C})\, e : \mathtt{C}} \text{ (T-SCAST)}$$

  Esta última regla es un tecnicismo necesario para poder probar la preservación de tipos con una semántica operacional estructural como lo haremos aquí.

Al estar presentes las tres reglas anteriores resulta que cualquier casting es permisible. De manera que podríamos usar sólo una regla:

$$\frac{\Gamma \vdash e : \mathtt{D}}{\Gamma \vdash (\mathtt{C})\, e : \mathtt{C}}$$

Sin embargo las reglas de casting hacia abajo y hacia arriba corresponden directamente a la implementación de JAVA.

4.3. Formación de clases

Una clase bien formada tiene cero o más atributos, un constructor que inicializa los atributos de la superclase y de la subclase y cero o más métodos.

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

Obsérvese que para tomar en cuenta la redefinición de métodos, no hay verificación del tipado de estos para aceptar una clase como bien formada, el tipado será relativo a la subclase.

4.4. Formación de métodos

La buena formación de métodos debe considerar el proceso de redefinición (overriding)

$$\frac{\begin{array}{c}T(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ldots\} \\ \mathsf{mtype}(\mathtt{m}, \mathtt{D}) = \vec{\mathtt{C}} \to \mathtt{C}_0 \\ \vec{x} : \vec{\mathtt{C}},\ \mathtt{this} : \mathtt{C} \vdash e_0 : \mathtt{C}_0' \\ \mathtt{C}_0' \leq \mathtt{C}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\,(\vec{\mathtt{C}}\ \vec{x}\ )\ \{\mathtt{return}\ e_0;\ \}\ \mathsf{ok\ in\ C}}$$

En las premisas se está considerando la posibilidad de que el método `m` se haya redefinido cuidando que se respete el tipo dado en la superclase, tal vez mediante subtipado. Es decir, el cuerpo del método en la subclase (en este caso e_0) debe devolver un subtipo (en este caso $\mathtt{C}_0'$) 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.
De importancia es observar que si $\mathtt{D} = \mathsf{Object}$ 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 $\mathtt{C}_0$ y $\vec{\mathtt{C}}$.

[2]Este es el nombre oficial de la regla, *stupid casting*

4.5. Formación de tablas de clases

Una tabla está bien formada si todas las clases declaradas en ella están bien formadas.

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

5. Semántica dinámica

Discutimos ahora la semántica dinámica de JPP mediante una semántica operacional estructural $e \to_T e'$ que depende de una tabla de clases T, cuya mención generalmente se omite.

Recordemos que los valores del sistema son objetos de la forma $\mathtt{new\ C}(\vec{v})$.

- Selección de atributos: puesto que convenimos en que los parámetros del constructor corresponden exactamente a los atributos de la clase, una instancia de la clase contiene toda la información requerida para determinar los valores de los atributos de esta instancia. De forma que una instancia es esencialmente una colección etiquetada de atributos. Cada instancia está etiquetada con su clase, lo cual sirve como guía para el despacho de métodos.

$$\frac{\mathsf{fields}(\mathtt{C}) = \vec{\mathtt{C}}\ \vec{f}}{(\mathtt{new\ C}(\vec{v}\ )).f_i \to v_i}\ (\text{E-ATR})$$

  La selección de atributos recupera el valor correspondiente de la superclase o de la subclase. Esto puede hacerse explícito cambiando la regla anterior aunque no es necesario.

- Invocación de métodos: si la instancia de clase $\mathtt{new\ C}(\vec{v})$ y los argumentos $\vec{w}$ pasados a un método son valores, entonces se puede evaluar la invocación como sigue:

$$\frac{\mathsf{mbody}(\mathtt{m}, \mathtt{C}) = (\vec{x}, e)}{\mathtt{new\ C}(\vec{v}\ ).\mathtt{m}(\overrightarrow{w}\ ) \to e\left[\vec{x}, \mathtt{this} := \overrightarrow{w}, \mathtt{new\ C}(\vec{v}\ )\right]}\ (\text{E-INEW})$$

  de manera que el envio de mensajes reeemplaza a la variable `this` por la instancia misma y a los parámetros del método por los valores pasados.

- *Casting*: aquí se verifica que la instancia sea una subclase de la clase destino en cuyo caso se devuelve la instancia misma, es decir, simplemente se elimina la anotación de casting.

$$\frac{\mathtt{C} \leq \mathtt{D}}{(\mathtt{D})\ (\mathtt{new\ C}(\vec{v}\ )) \to \mathtt{new\ C}(\vec{v})}\ (\text{E-CNEW})$$

  Obsérvese que esta es una regla de evaluación condicional que causa dificultades en la implementación del sistema pues delega la verificación de subclase al tiempo de ejecución.

- Orden de evaluación: como es usual de izquierda a derecha determinado por las siguientes reglas burocráticas:

$$\frac{e \to e'}{e.f \to e'.f}\ (\text{E-BATR}) \qquad \frac{e \to e'}{e.\mathtt{m}(\vec{e}\ ) \to e'.\mathtt{m}(\vec{e}\ )}\ (\text{E-BINV})$$

$$\frac{e_i \to e_i'}{v.\mathtt{m}(\vec{v}, e_i, \vec{e}\,) \to \mathtt{m}(\vec{v}, e_i', \vec{e}\,)}\ (\text{E-BIARG}) \qquad \frac{e_i \to e_i'}{\mathtt{new\ C}(\vec{v}, e_i, \vec{e}\,) \to \mathtt{new\ C}(\vec{v}, e_i', \vec{e}\,)}\ (\text{E-BNARG})$$

$$\frac{e \to e'}{(\mathtt{C})\ e \to (\mathtt{C})\ e'}\ (\text{E-BCAST})$$

6. Seguridad de Java Peso Pluma

Es de esperar que los teoremas de preservación y progreso requieren de restricciones adicionales respecto a sus análogos en otros sistemas. Por ejemplo ahora debemos considerar la herencia así como el uso de un casting inválido.

Proposición 1 (Preservación de tipos) *Si T es una tabla de clases bien formada, $\Gamma \vdash e : \mathtt{C}$ y $e \to_T e'$ entonces existe $\mathtt{C}'$ tal que $\mathtt{C}' \leq \mathtt{C}$ y $\Gamma \vdash e' : \mathtt{C}'$.*

Demostración. Ejercicio. ⊣

La regla de casting estúpido es indispensable para probar la preservación de tipos, pues de otra manera tendríamos el siguiente ejemplo: considérese una clase trivial `D` y una instancia `new D()`. El paso de evaluación

$$(\mathtt{C})\ (\mathtt{Object})\ \mathtt{new\ D}() \to (\mathtt{C})\ \mathtt{new\ D}()$$

es válido puesto que $(\mathtt{Object})\ \mathtt{new\ D}() \to \mathtt{new\ D}()$ dado que $\mathtt{D} \leq \mathtt{Object}$, pero la expresión $(\mathtt{C})\ \mathtt{new\ D}()$ no estaría tipada correctamente de no existir la regla de casting estúpido.

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, por ejemplo la expresión

$$(\mathtt{C})(\ \mathtt{new\ Object}())$$

con $\mathtt{C} \neq \mathtt{Object}$ está bien tipada debido a la regla del casting estupido, pero queda bloqueada puesto que la regla (E-CNEW) exige verificar que $\mathtt{Object} \leq \mathtt{C}$ lo cual no se cumple. De hecho cuando se bloquea una expresión bien tipada esta es la única causa posible.

Proposición 2 (Progreso) *Sea T una tabla de clases bien formada. Si $\vdash e : \mathtt{C}$ entonces sucede una y sólo una de las siguientes condiciones:*

- *e es un valor.*
- *e contiene una expresión de la forma $(\mathtt{C})\ \mathtt{new\ D}(\vec{v}\,)$ donde $\mathtt{D} \not\leq \mathtt{C}$.*
- *Existe e' tal que $e \to_T e'$*

Demostración. Ejercicio ⊣

7. Correspondencia con JAVA

Para finalizar hacemos explícita la correspondencia deseada entre un programa en Java Peso Pluma y su versión en JAVA:

- Cada programa sintácticamente correcto en Java Peso Pluma también lo es en JAVA.
- Un programa sintácticamente correcto es tipable en Java Peso Pluma (sin usar la regla de casting estúpido) si y sólo si es tipable en JAVA.
- Un programa bien tipado en Java Peso Pluma se comporta igual que en JAVA, por ejemplo evaluar un programa en Java Peso Pluma causa no terminación si y sólo si compilarlo y ejecutarlo en JAVA causa no terminación.

Por supuesto que las afirmaciones anteriores no pueden probarse al no existir una formalización de JAVA. Sin embargo es muy útil enunciar la correspondencia de manera precisa para dejar en claro el objetivo que estamos tratando de alcanzar. Esto facilita en particular una forma rigurosa de juzgar contraejemplos.

8. Implementación de Números Naturales

Presentamos aquí una posible implementación de números naturales mediante objetos.

```
class Cero extends Nat {

 Cero (Object n) { super(n);}

 Nat pred(){ return this; }

 Nat suma (Nat n) { return n; }

 Nat multi (Nat n) { return this; }

}


class Nat extends Object {

  Object p;

  Nat (Object n) { super(); this.p = n }

  Nat suc() { return new Nat(this); }

  Nat pred() { return (Nat) this.p; }
```

```
  Nat suma(Nat n) { return this.pred().suma(n.suc()); }

  Nat multi(Nat n) { return n.suma(this.pred().multi(n)); }
}
```