Programación Orientada a Objetos

Departamento de Computación, FCEyN, UBA

2018

Objetos - La metáfora

- Todo programa es una simulación. Cada entidad del sistema que se está simulando se representa en el programa a través de una entidad u objeto.
  - Asociar los objetos físicos o conceptuales de un dominio del mundo real en objetos del dominio del programa
  - los objetos en el programa tienen las características y capacidades del mundo real que nos interese modelar.
- Todas las componentes de un sistema son objetos

Modelo de Computación - La metáfora

- El modelo de computación consiste en el envío de mensajes: Un sistema está formado por *objetos* que comunican a través del intercambio de mensajes.
- Mensajes es una solicitud para que un objeto lleve a cabo una de sus operaciones
- El receptor, es decir, el objeto que recibe el mensaje, determina cómo llevar a cabo la operación.

Ejemplo: `"unCirculo radio"`/`unCirculo.radio`

- `unCirculo` es el objeto receptor
- `radio` es el mensaje que se le envía.

Objetos

- El conjunto de mensajes a los que un objeto responde se denomina interfaz o protocolo
- La forma en que un objeto lleva a cabo una operación está descripta por un método (describe la implementación de las operaciones)
- La forma en que un objeto lleva a cabo una operación puede depender de un estado interno
  - El estado se representa a través de un conjunto de colaboradores internos (también llamados atributos o variables de instancia)

unRectangulo

- interfaz: `area`
- atributos: `alto` y `ancho`
- método: `area = function (){return alto*ancho}`

Objetos

La única manera de interactuar con un objeto es a través del envío de mensajes

- la implementación de un objeto no puede depender de los detalles de implementación de otros objetos
- principio básico heredado de los Tipos Abstractos de Datos, antecesores de los Objetos:

Principio de ocultamiento de la información

- El estado de un objeto es privado y solamente puede ser consultado o modificado por sus métodos.
- No todos los lenguajes imponen esta restricción.

Detrás del modelo de cómputo: Method dispatch

- La interacción entre objetos se lleva a cabo a través de envío de mensajes
- Al recibir un mensaje se activa el método correspondiente
- Para poder procesar este mensaje es necesario hallar la declaración del método que se pretende ejecutar
- El proceso de establecer la asociación entre el mensaje y el método a ejecutar se llama method dispatch
- Si el method dispatch se hace en tiempo de
  - compilación (i.e. el método a ejecutar se puede determinar a partir del código fuente): se habla de method dispatch estático
  - ejecución: se habla de method dispatch dinámico

Corrientes

¿Quién es responsable de conocer los métodos de los objetos?

2 Alternativas conocidas:

- Clasificación
- Prototipado

Clasificación

Clases

- Modelan conceptos abstractos del dominio del problema a resolver
- Se utilizan principios de diseño para decidir cuando crearlas
- Definen el comportamiento y la forma de un conjunto de objetos (sus instancias)
- Todo objeto es instancia de alguna clase

Ejemplo

clase Point

Variables de instancia 'xCoord' e 'yCoord'

Métodos

```
x
    ^xCoord
y
    ^yCoord

dist: aPoint
"Answer the distance between aPoint and the receiver."
    | dx dy |
    dx := aPoint x - xCoord.
    dy := aPoint y - yCoord.
    ^ (dx * dx + (dy * dy)) sqrt
```

Componentes de una clase

Componentes de una clase

- un nombre
- definición de variables de instancia
- métodos de instancia
- por cada método se especifica
  - su nombre
  - parámetros formales
  - cuerpo

Ejemplo

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Vars. de instancia `'left right'`

Métodos de instancia

```
sum
    ^ left sum + right sum
```

clase Leaf

Métodos de clase

```
new: anInteger
    "Creates a leaf"
    ...
```

Vars. de instancia `'value'`

Métodos de instancia

```
sum
    ^value
```

Ejemplos

```
1) Leaf new: 5
2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum
```

Self

Pseudo variable que, durante la evaluación de un método, referencia al receptor del mensaje que activó dicha evaluación.

- no puede ser modificada por medio de una asignación.
- se liga automáticamente al receptor cuando comienza la evaluación del método.

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Var. de instancia `'left right'`

Métodos de instancia

```
l
    ^left
r
    ^right
sum
    ^ (self l) sum + (self r) sum
```

Jerarquía de clases

- Es común que nuevas clases aparezcan como resultado de la extensión de otras existentes incluyendo
  - adición o cambio del comportamiento de uno o varios métodos
  - adición de nuevas variables de instancia o clase
- Una clase puede heredar de o extender una clase pre-existente (la superclase)
- La transitividad de la herencia da origen a las nociones de ancestros y descendientes

Ejemplo

```
Object subclass: #Point
instanceVariableNames: 'x y'
```

Métodos de clase

```
x: p1 y: p2
    ^self new setX: p1 setY: p2
```

Métodos de instancia

```
x
    ^x
y
    ^y

setX: xValue setY:yValue
   x :=  xValue.
   y :=  yValue.
```

```
Point subclass: #ColorPoint
instanceVariableNames: 'color'
```

Métodos de clase

```
x: p1 y: p2 color: aColor
    |instance|
    instance := self x: p1 y: p2.
    instance color: aColor.
    ^instance
```

Métodos de instancia

```
color: aColor
    color := aColor

color
    ^color
```

uso

```
ColorPoint x: 10 y: 20 color: red.
```

Herencia

- Hay dos tipos de herencia
  - Simple: una clase tiene una única clase padre (salvo la clase raíz `object`)
  - Múltiple: una clase puede tener más de una clase padre
- La gran mayoría de los lenguajes OO utilizan herencia simple
- La herencia múltiple complica el proceso de method dispatch

Inconveniente con herencia múltiple

- Supongamos que
  - clases A y B son incomparables y C es subclase de A y B
  - A y B definen (o heredan) dos métodos diferentes para m
  - se envía el mensaje m a una instancia C
- ¿Qué método debe ejecutarse?
- Dos soluciones posibles:
  - Establecer un orden de búsqueda sobre las superclases de una clase
  - Si se heredan dos métodos diferentes para el mismo mensaje debe ser redefinidos en la clase nueva

Method Dispatch Estático

- Method dispatch dinámico es uno de los pilares de la POO (junto con la noción de clase y de herencia)
- Por cuestiones de eficiencia (o diseño, como el caso de C++) muchos lenguajes también cuentan con method dispatch estático
- Sin embargo, hay algunas situaciones donde method dispatch estático es requerido, más allá de cuestiones de eficiencia
- Un ejemplo es el `super`

Method Dispatch Estático

Supongamos que queremos extender la clase `point` del siguiente modo:

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY: yValue setColor: aColor
  x:= xValue.
  y:= yValue.
  color:= aColor.
```

¡`setX: setY: setColor:` duplica código innecesariamente!

Method Dispatch Estático

- Esto es un ejemplo claro de mala práctica de programación en presencia de herencia
- Deberíamos recurrir al código ya existente del método `initialize` de `point` para que se encargue de la inicialización de `x` e `y`

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY:yValue setColor: aColor
   self setX: xValue setY: yValue.
   color:= aColor.
```

Method Dispatch Estático

- ¿La siguiente variante funciona?

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.

Point subclass: #BluePoint
Métodos de instancia
setX: xValue setY: yValue
   self setX: xValue setY: yValue.
   color:= 'azul'.
```

Method Dispatch Estático

Super

- Pseudovariable que referencia al objeto que recibe el mensaje
- Cambia el proceso de activación al momento del envío de un mensaje.
- Una expresión de la forma `"super msg"` que aparece en el cuerpo de un método `m` provoca que el method lookup se haga desde el padre de la clase anfitriona de `m`

Código corregido

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  x:= xValue.
  y:= yValue.
```

```
Point subclass: #BluePoint
Métodos de instancia
setX: xValue setY: yValue
   super setX: xValue setY: yValue.
   color:= 'azul'.
```

Variante de super en algunos lenguajes

```
super[A] n(...)
```

- Similar super ya visto
- Salvo que la búsqueda comienza desde la clase A
- A debe ser una superclase de la clase anfitriona de m, el método que contiene la expresión super

Lenguajes basados en Objetos

- Caracterizados por la ausencia de clases
- Constructores para la creación de objetos particulares

```
let celda = {
    contenido : 0,
    get : function () {return this.contenido;},
    set : function (n) {this.contenido = n;},
}
```

- Procedimientos para generar objetos

```
Celda = function () {
    this.contenido = 0;
    this.get = function ()
            {return this.contenido;};
    this.set = function (n)
            {this.contenido = n;};
}

otracelda = new Celda ();
```

Prototipado

- Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos)
- Otras instancias se generan por clonación (copia shallow)

```
celdaClonada = Object.create(celda)
```

- los clones se pueden cambiar

```
celdaClonada.set = function (n){
        this.contenido = this.contenido + n;
        }
```

- Herencia a través de prototipos (mas en la práctica de mañana)

Cálculo de Objetos no tipado (ς cálculo) [Abadi&Cardelli,98]

Ingredientes

- Objetos como única estructura computacional.
- Los objetos son una colección de atributos nombrados (registros).
- Todos los atributos son métodos.
- Cada método tiene una única variable ligada que representa a `self` (`this`) y un cuerpo que produce un resultado.
- Los objetos proveen dos operaciones:
  - envío de mensaje (invocación de un método)
  - redefinición de un método

Sintaxis

$$
\begin{array}{lcll}
a, b & ::= & x & \textit{variable} \\
 & | & [l_i = \varsigma(x_i) b_i{}^{i \in 1..n}] & \textit{objeto} \\
 & | & a.l & \textit{selección / envío de mensaje} \\
 & | & a.l \Leftarrow \varsigma(x) b & \textit{redefinición de método}
\end{array}
$$

Example

$$o \stackrel{\text{def}}{=} [l_1 = \varsigma(x_1)[],\ l_2 = \varsigma(x_2) x_2.l_1]$$

- o tiene dos métodos:
  - l_1 retorna un objeto vacío $[]$.
  - l_2 es un método que envía el mensaje l_1 a `self` (representado por el parámetro x_2).

Atributos vs métodos

- el cálculo ς no incluye explícitamente atributos (campos/fields).
- los atributos se representan como métodos que no utilizan al parámetro `self`. Por ejemplo, l_1 en

$$o \stackrel{\text{def}}{=} [l_1 = \varsigma(x_1)[],\ l_2 = \varsigma(x_2)x_2.l_1]$$

- De esta manera,
  - el envío de un mensaje representa también a la selección de un atributo
  - la redefinición de un método representa también a la asignación de un atributo

Notación

- Atributo: $[\ldots, l = b, \ldots]$ es una abreviatura de $[\ldots, l = \varsigma(x)b, \ldots]$ cuando x no se usa en b.
- Asignación de atributo: $o.l := b$ denota $o.l \Leftarrow \varsigma(x)b$ cuando x no se usa en b

Example (Ejemplo)

Escribimos

$$o \stackrel{\text{def}}{=} [l_1 = [],\ l_2 = \varsigma(x_2)x_2.l_1]$$

en lugar de

$$o \stackrel{\text{def}}{=} [l_1 = \varsigma(x_1)[],\ l_2 = \varsigma(x_2)x_2.l_1]$$

Sintaxis - Variables libres

ς es un ligador para el parámetro `self` x_i en el cuerpo b_i de la expresión $\varsigma(x_i)b_i$

Definición

$$
\begin{array}{ll}
\mathsf{fv}(\varsigma(x)b) & = \mathsf{fv}(b) \setminus \{x\} \\
\mathsf{fv}(x) & = \{x\} \\
\mathsf{fv}([l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]) & = \bigcup^{i\in 1..n} \mathsf{fv}(\varsigma(x_i)b_i) \\
\mathsf{fv}(a.l) & = \mathsf{fv}(a) \\
\mathsf{fv}(a.l \Leftarrow \varsigma(x)b) & = \mathsf{fv}(a.l) \cup \mathsf{fv}(\varsigma(x)b)
\end{array}
$$

Un término a es cerrado si $\mathsf{fv}(a) = \varnothing$.

Sustitución

$$
\begin{aligned}
x\{x \leftarrow c\} &= c \\
y\{x \leftarrow c\} &= y && \textit{if } x \neq y \\
([l_i = \varsigma(x_i)b_i^{i\in 1..n}])\{x \leftarrow c\} &= [l_i = (\varsigma(x_i)b_i)\{x \leftarrow c\}^{i\in 1..n}] \\
(a.l)\{x \leftarrow c\} &= (a\{x \leftarrow c\}).l \\
(a.l \Leftarrow \varsigma(x)b)\{x \leftarrow c\} &= (a\{x \leftarrow c\}).l \Leftarrow ((\varsigma(x)b)\{x \leftarrow c\}) \\
(\varsigma(y)b)\{x \leftarrow c\} &= \varsigma(y')(b\{y \leftarrow y'\}\{x \leftarrow c\}) \\
& && \textit{if } y' \notin \mathsf{fv}(\varsigma(y)b) \cup \mathsf{fv}(c) \cup \{x\}
\end{aligned}
$$

Equivalencia de términos ($\equiv$)

- Los términos $\varsigma(x)b$ y $\varsigma(y)(b\{x \leftarrow y\})$ con $y \notin \mathsf{fv}(b)$ se consideran equivalentes (α-conversión).
- Dos objetos que difieren en el orden de sus componentes son considerados equivalentes.
- Por ejemplo,

$$o_1 \stackrel{\text{def}}{=} [l_1 = \varsigma(x_1)[],\ l_2 = \varsigma(x_2)x_2.l_1]$$
$$o_2 \stackrel{\text{def}}{=} [l_2 = \varsigma(x_3)x_3.l_1,\ l_1 = \varsigma(x_1)[]]$$

son equivalentes ($o_1 \equiv o_2$).

Semántica operacional

Valores

$$v ::= [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}]$$

Reducción *big-step* $\longrightarrow$

$$\frac{}{v \longrightarrow v}\ [\textsc{Obj}]$$

$$\frac{a \longrightarrow v' \quad v' \equiv [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}] \quad b_j\{x_j \leftarrow v'\} \longrightarrow v \quad j \in 1..n}{a.l_j \longrightarrow v}\ [\textsc{Sel}]$$

$$\frac{a \longrightarrow [l_i = \varsigma(x_i)b_i{}^{i\in 1..n}] \qquad j \in 1..n}{a.l_j \Leftarrow \varsigma(x)b \longrightarrow [l_j = \varsigma(x)b,\ l_i = \varsigma(x_i)b_i{}^{i\in 1..n-\{j\}}]}\ [\textsc{Upd}]$$

Ejemplos $[a = [], b = \varsigma(x)x.a].b \longrightarrow \ldots$

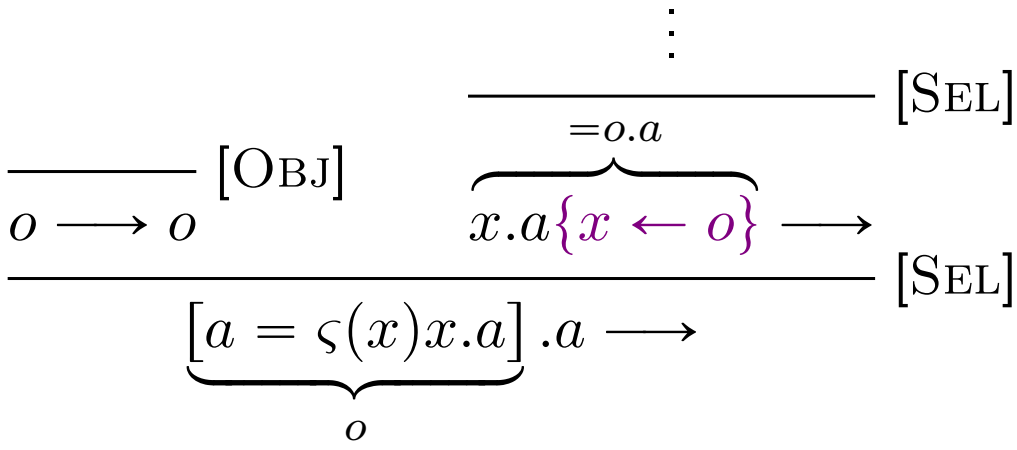
$$\dfrac{\dfrac{o \longrightarrow o \qquad \overbrace{[]\{x \leftarrow o\}}^{=[]} \longrightarrow []}{\overbrace{(x.a)\{x \leftarrow o\}}^{=o.a} \longrightarrow []}\,[\textsc{Sel}]}{\underbrace{[a = [], b = \varsigma(x)x.a]}_{o}.b \longrightarrow []}\,[\textsc{Sel}]$$

$$\dfrac{}{o \longrightarrow o}\,[\textsc{Obj}]$$

Ejemplos $([a = [], b = \varsigma(x)x.a].b \Leftarrow \varsigma(y)[]).b \longrightarrow \ldots$

$$\dfrac{\dfrac{\dfrac{}{o' \longrightarrow o'}\,[\textsc{Obj}]}{\underbrace{o}_{o \longrightarrow [a = [], b = []]}}\,[\textsc{Upd}] \qquad \dfrac{}{\underbrace{[]\{x \leftarrow o\}}_{=[]} \longrightarrow []}\,[\textsc{Obj}]}{\underbrace{(\underbrace{[a = [], b = \varsigma(x)x.a]}_{o'}.b \Leftarrow \varsigma(y)[])}_{o}.b \longrightarrow []}\,[\textsc{Sel}]$$

Ejemplos $([a = \varsigma(x)x.a].a \longrightarrow \ldots$



La evaluación de esta expresión se indefine (análogo a `fix` $\lambda x : \sigma.x$).

Ejemplo: Los naturales

- Asumir que existen los objetos `true` y `false` que corresponden a los valores booleanos (Ver práctica)
- Luego

$$\texttt{zero} \stackrel{\text{def}}{=} [\, iszero = \texttt{true},\ pred = \varsigma(x)x,\ succ = \varsigma(x)(x.iszero := \texttt{false}).pred := x]$$

$$\texttt{uno} \stackrel{\text{def}}{=} \texttt{zero}.succ$$

$$\texttt{uno} \longrightarrow \underbrace{[iszero = \texttt{false}, pred = \texttt{zero}, succ = \ldots]}_{\texttt{uno}'}$$

$$\texttt{dos} \stackrel{\text{def}}{=} \texttt{zero}.succ.succ$$

$$\texttt{dos} \longrightarrow [iszero = \texttt{false}, pred = \texttt{uno}', succ = \ldots]$$

$$\ldots$$

Las funciones

- Es posible codificar los términos del cálculo λ (no tipado).

$$M ::= MN \mid \lambda x.M \mid x$$

- Idea:
  - Representar a una función como un objeto $[arg = \ldots,\ val = \ldots]$.
  - Al aplicar una función, primero se asigna el valor del argumento al atributo arg y luego se envía el mensaje val que evalúa el cuerpo de la función.
  - (fv) se traduce en $(\mathsf{o}_f.arg := \mathsf{o}_v).val$

Codificación de lambda cálculo $[\![\_]\!] : M \to a$

$$
\begin{array}{lcl}
[\![x]\!] & \stackrel{\text{def}}{=} & x \\
[\![MN]\!] & \stackrel{\text{def}}{=} & ([\![M]\!].arg := [\![N]\!]).val \\
[\![\lambda x.M]\!] & \stackrel{\text{def}}{=} & [\ val = \varsigma(y)[\![M]\!]\{x \leftarrow y.arg\}, \\
 & & \quad arg = \varsigma(y)y.arg \quad\quad]
\end{array}
$$

Example $((\lambda x.x)y)$

$$
\begin{array}{lcl}
[\![\lambda x.x]\!] & \stackrel{\text{def}}{=} & [val = \varsigma(y)[\![x]\!]\{x \leftarrow y.arg\}, arg = \varsigma(y)y.arg] \\
 & = & [val = \varsigma(y)x\{x \leftarrow y.arg\}, arg = \varsigma(y)y.arg] \\
 & = & [val = \varsigma(y)y.arg, arg = \varsigma(y)y.arg]
\end{array}
$$

$$
\begin{array}{lcl}
[\![(\lambda x.x)\ y]\!] & \stackrel{\text{def}}{=} & ([\![\lambda x.x]\!].arg := [\![y]\!]).val \\
 & = & ([val = \varsigma(x)x.arg, arg = \varsigma(x)x.arg].arg := y).val \\
 & \longrightarrow & y
\end{array}
$$

Qué sucede al evaluar $[\![\lambda x.x]\!].val$?

Métodos con parámetros

Un método que espera un parámetro es un método cuya definición es (un objeto que codifica a) una función.

$$\varsigma(x)[\![\lambda x.M]\!]$$

Notación

- $\lambda(x)M$ en lugar de $[\![\lambda x.M]\!]$
- $M(N)$ en lugar de $[\![MN]\!]$

Ejemplo: Punto en el plano

- Un punto en el plano que puede ser desplazado y se encuentra inicialmente en el origen de coordenadas.

$$\texttt{origen} \stackrel{\text{def}}{=} [\ x = 0,\ y = 0,\ mv\_x = \varsigma(p)\lambda(d_x)p.x := p.x + d_x,\ mv\_y = \varsigma(p)\lambda(d_y)p.y := p.y + d_y\ ]$$

- Luego,

$$\texttt{unidad} \stackrel{\text{def}}{=} \texttt{origen}.mv\_x(1).mv\_y(1) \longrightarrow [x = 1, y = 1, mv\_x = \ldots, mv\_y = \ldots]$$

(Stateless) Traits

- Un trait es una colección de métodos.
- require un conjunto de métodos que parametrizan el comportamiento provisto.
- (Stateless) *Traits* no especifican variables de estado ni acceden al estado.

$$\texttt{CompT} \stackrel{\text{def}}{=} [\ eq\ = \varsigma(t)\lambda(x)\lambda(y)\ if(x.comp(y)) == 0\ then\ \texttt{true}\ else\ \texttt{false},\ leq = \varsigma(t)\lambda(x)\lambda(y)\ \texttt{if}(x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}\ ]$$

(Stateless) Traits

- Los vamos a representar como una colección de pre-metodos:
  - pre-metodo: $\varsigma(\mathtt{t})\lambda(y)b$ con $t \notin \mathsf{fv}(\lambda(y)b)$ (no usan el parámetro `self` `t`).
  - Recordar que en este caso podemos omitir $\varsigma(\mathtt{t})$ y escribir $\lambda(y)b$.
  - Luego, $\mathtt{t} = [l_i = \lambda(y_i)b_i{}^{i\in 1..n}]$ es un trait.
- A partir de un trait $\mathtt{t} = [l_i = \lambda(y_i)b_i{}^{i\in 1..n}]$ podemos definir un constructor de objetos (cuando `t` es completo).

$$new \stackrel{\text{def}}{=} \lambda(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}]$$

$$\begin{array}{rcl} o & \stackrel{\text{def}}{=} & new\ \mathtt{t} \\ & \longrightarrow & [l_i = \varsigma(s)\mathtt{t}.l_i(s)^{i\in 1..n}] \\ & = & [l_i = \varsigma(y_i)b_i{}^{i\in 1..n}] \end{array}$$

(Stateless) Traits

$$\texttt{CompT} \stackrel{\text{def}}{=} [\ eq = \varsigma(t)\lambda(x)\lambda(y)\ if\,(x.comp(y)) == 0\ then\ \texttt{true}\ else\ \texttt{false},\ leq = \varsigma(t)\lambda(x)\lambda(y)\ if\,(x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}$$

$$new \stackrel{\text{def}}{=} \lambda(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}]$$

$$new\ \texttt{CompT} \longrightarrow [\ eq = \varsigma(x)\lambda(y)\ if\,(x.comp(y)) == 0\ then\ \texttt{true}\ else\ \texttt{false},\ leq = \varsigma(x)\lambda(y)\ if\,(x.comp(y)) < 0\ then\ \texttt{true}\ else\ \texttt{false}\ ]$$

- Este objeto es inutilizable (porque `CompT` no es completo).
- Se puede definir un procedimiento que incorpora un *trait* a un objeto.

Clases

- Una clase es un *trait* (completo) que además provee un método new.

$$\begin{array}{rcl} \mathsf{C} \stackrel{\text{def}}{=} & [\ new & = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}], \\ & l_i & = \lambda(s)b_i{}^{i\in 1..n}] \end{array}$$

- Luego,

$$\begin{array}{rcl} o & \stackrel{\text{def}}{=} & \mathsf{C}.new \\ & \longrightarrow & [l_i = \varsigma(s)\mathsf{C}.l_i(s)^{i\in 1..n}] \\ & \approx & [l_i = \varsigma(s)b_i{}^{i\in 1..n}] \end{array}$$

Clase Contador

$$
\begin{aligned}
Contador \stackrel{\text{def}}{=} [\ new = \varsigma(z)[\ & v = \varsigma(s)z.v(s), \\
& inc = \varsigma(s)z.inc(s), \\
& get = \varsigma(s)z.get(s)], \\
v = \lambda(s)0, & \\
inc = \lambda(s)s.v := s.v + 1, & \\
get = \lambda(s)s.v & \ ]
\end{aligned}
$$

Representando Herencia

- Sea la clase

$$\mathsf{c} \stackrel{\text{def}}{=} [\ new = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}],\ l_i = \lambda(s)b_i{}^{i\in 1..n}]$$

- Se desea definir c' como subclase de c que agrega los pre-metodos $\lambda(s)b_k{}^{k\in n+1..n+m}$

$$\mathsf{c}' \stackrel{\text{def}}{=} [\ new = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n+m}],\ l_j = \mathsf{c}.l_j{}^{j\in 1..n},\ l_k = \lambda(s)b_k{}^{k\in n+1..n+m}\ ]$$

Consideraciones adicionales

- El cálculo tiene un sabor funcional.
- Existe una versión imperativa, donde se mantienen un store con referencias a objetos.
  - `Clone`(a) crea un nuevo objeto que tiene las mismas etiquetas de a y cada componente comprate los métodos con las componentes de a.
  - Introduce herencia a nivel de objetos (más en la práctica)
- las primitivas fueron seleccionadas desde una perspectiva de tipado estático.
  - No se pueden agregar o eliminar dinámicamente métodos en un objeto.
  - Los métodos no pueden extraerse de los objetos.
  - Las distintas versiones vienen equipadas con un sistema de tipos.
  - Existe una diferencia significativa en el tipado de λ y σ (la contravarianza en el parámetro `self` falla).