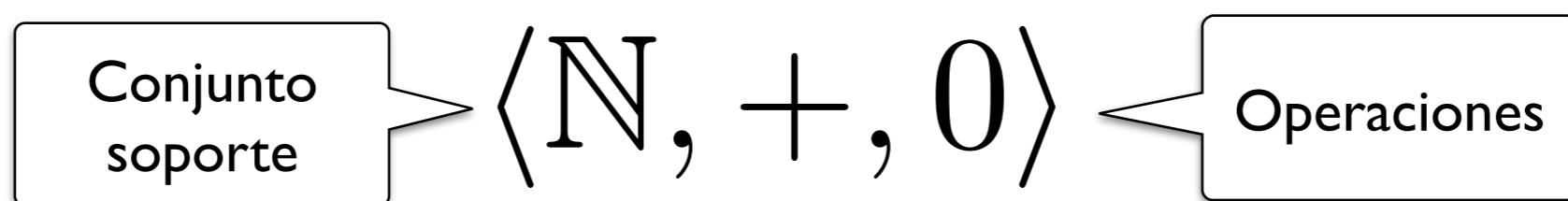


# Teoría de TADs

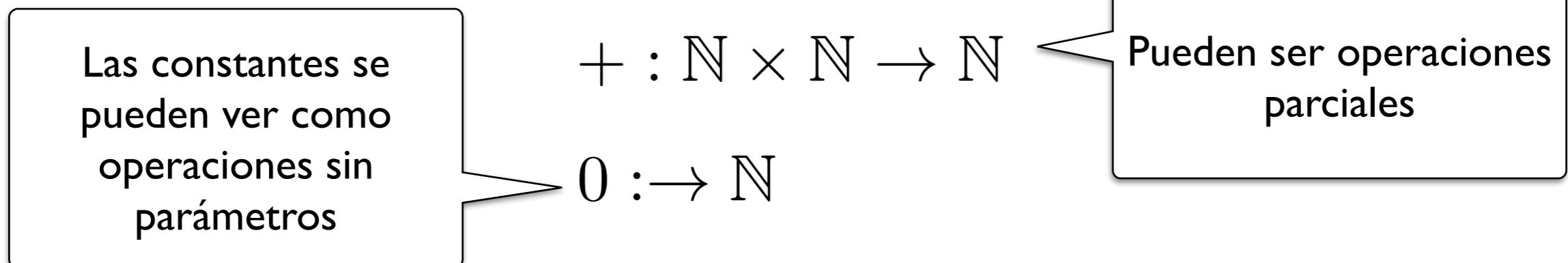
Pablo Castro  
Estructura de Datos y Alg.  
UNRC

# Tipos y Algebras

Un álgebra es un conjunto soporte más operaciones:



Las operaciones son funciones:



# Tipos y Algebras

Los tipos en un lenguaje de programación son algebras:

$$\langle \mathbb{Z}, +, -, *, 0, 1 \rangle$$

Enteros

$$\langle Bool, \vee, \wedge, \neg, true, false \rangle$$

Booleanos

$$\langle [A], :, [] \rangle$$

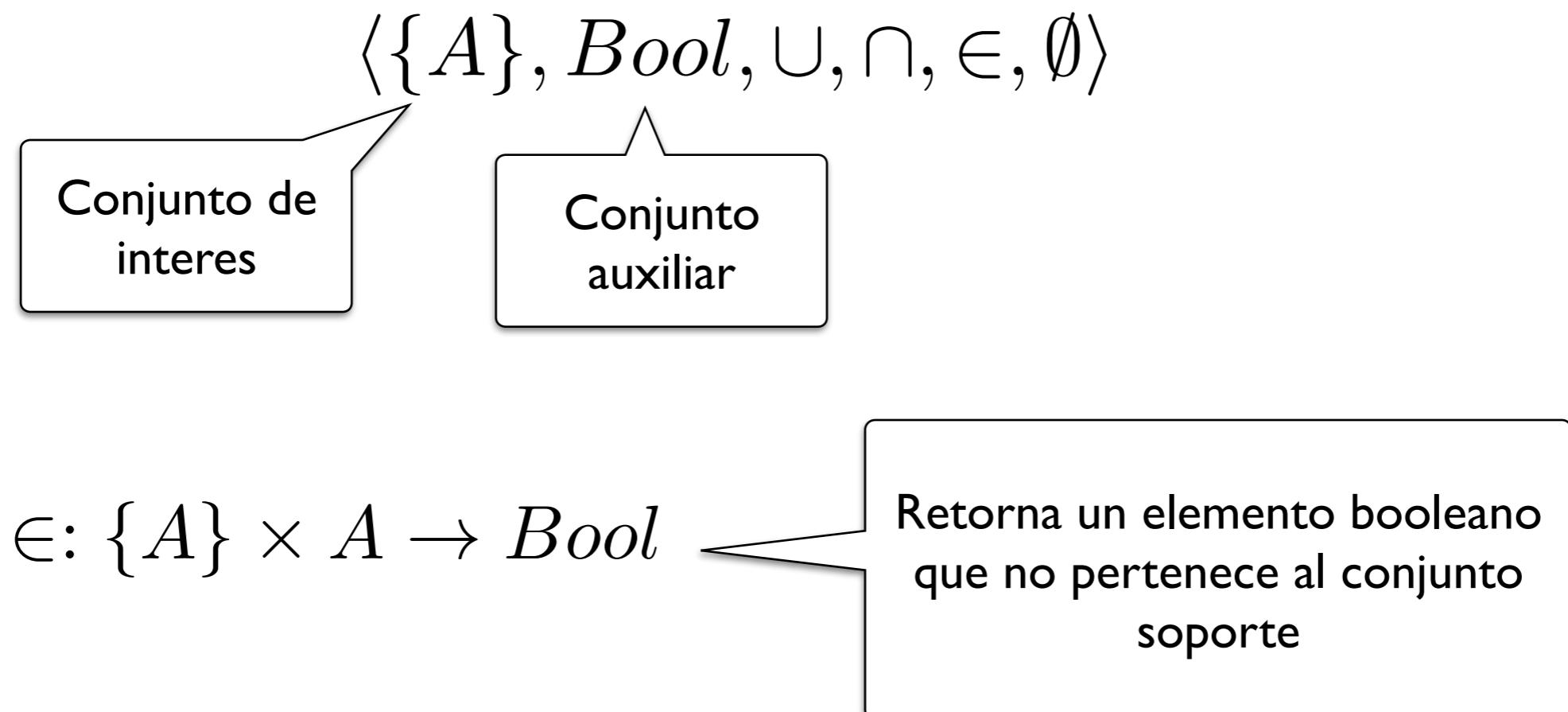
Listas polimórficas de tipo A

$$\langle \{A\}, \cup, \cap, \setminus, \emptyset \rangle$$

Conjuntos polimórficos de tipo A

# Algebras Heterogéneas

Muchas veces utilizaremos algebras que tienen varios conjuntos soportes



# Clases de Operaciones

**Observadoras:** Aquellas operaciones que devuelven un elemento de un tipo auxiliar

$$\in : \{A\} \times A \rightarrow \textit{Bool}$$

**Modificadoras:** Aquellas operaciones que toman al menos un elemento del tipo de interés, y devuelven algo del tipo de interés

$$\cup : \{A\} \times \{A\} \rightarrow \{A\}$$

**Generadoras:** Aquellas que no toman un elemento del tipo de interés, y devuelven algo del tipo de interés

$$\emptyset : \rightarrow \{A\}$$

# Descripción de TADs

Un TAD es una descripción abstracta de un tipo. Se especifican con axiomas:

Interface de la pila

$new : \rightarrow Stack$

$push : Stack \times El \rightarrow Stack$

$pop : Stack \rightarrow Stack$

$top : Stack \rightarrow El$

$empty : Stack \rightarrow Bool$

$pop(new()) = new()$   
 $pop(push(s, e)) = s$

$top(new()) = error$   
 $top(push(s, e)) = e$   
 $empty(new()) = true$   
 $empty(push(s, e)) = false$

Axiomas que definen sus propiedades

# Algebra de Términos

Podemos construir un álgebra utilizando los términos provistos por la interface:

```
new()  
pop(push(new(), 1))  
push((push(new(), 1), 2)  
push((push(new(), 0), 0)  
top(push(new(), 0))  
0
```

Tenemos un número infinito de términos

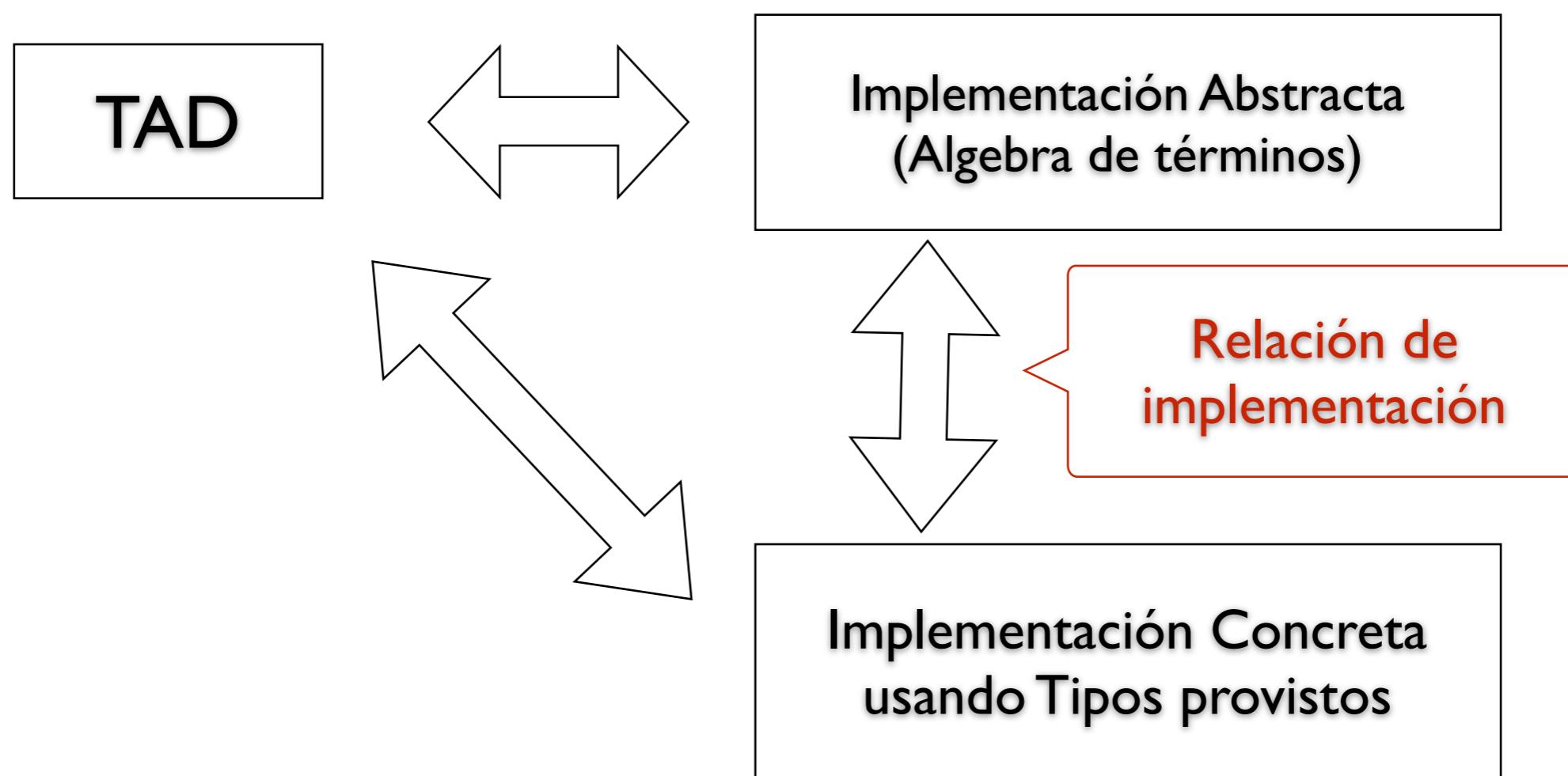
Podemos agrupar aquellos términos que son equivalentes por los axiomas

```
push(new(), 1)  
push(new(), 0)
```

El álgebra de términos denotada:  $T(X)/\equiv$   
y satisface los axiomas

# Implementación de TADS

Muchas veces los tipos que necesitamos no están provistos en los lenguajes de programación.



# Implementación de TADS

- Llamaremos al tipo que queremos implementar Algebra Abstracta
- Llamaremos a su implementación Algebra Concreta

La relación entre ambas álgebras viene dada por una función de abstracción:

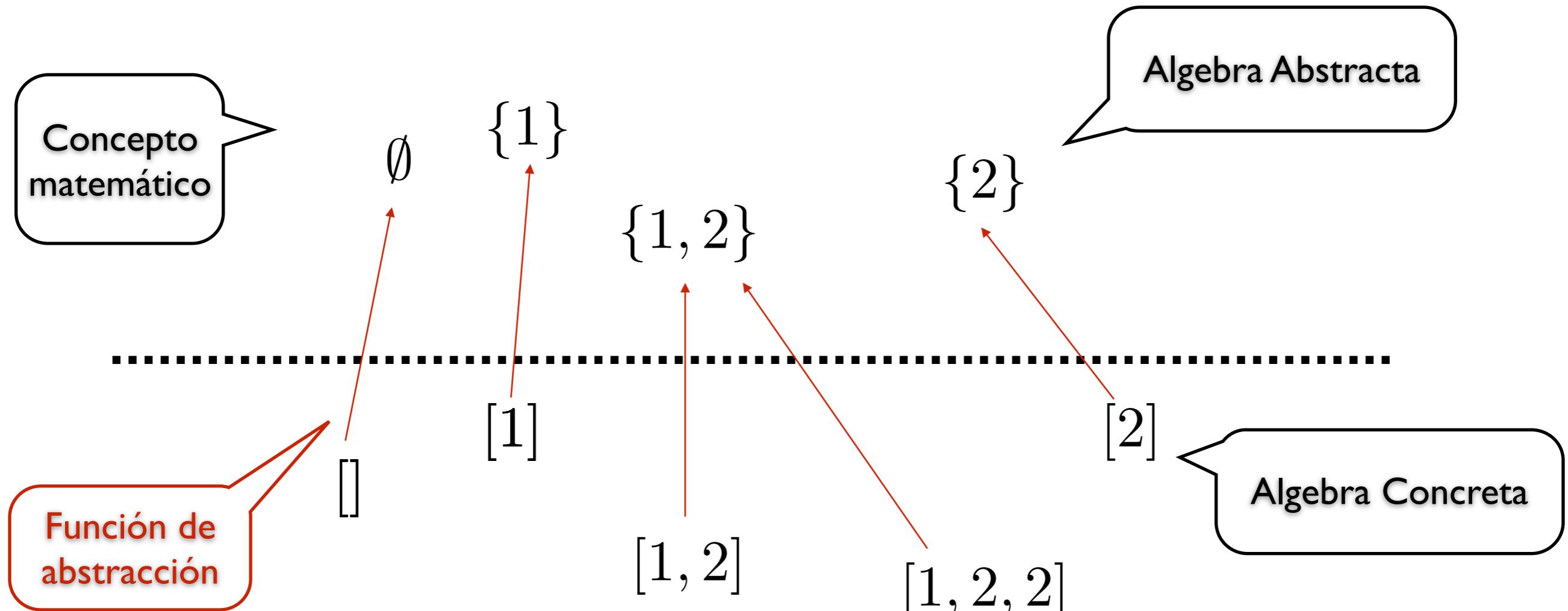
$$[-] : C \rightarrow A$$

Algebra concreta

Algebra abstracta

# Ejemplo

El tipo de conjuntos en general no viene provisto por el lenguaje de programación.



Se implementan conjuntos con listas.

# Función de Abstracción

Relaciona un tipo con su implementación:

$$[-] : C \rightarrow A$$

Debe cumplir:

- Es suryectiva.
- Cada operación del álgebra abstracta tiene una correspondiente en el álgebra concreta  
(y cumplir ciertos requisitos...)

# Operaciones Generadoras

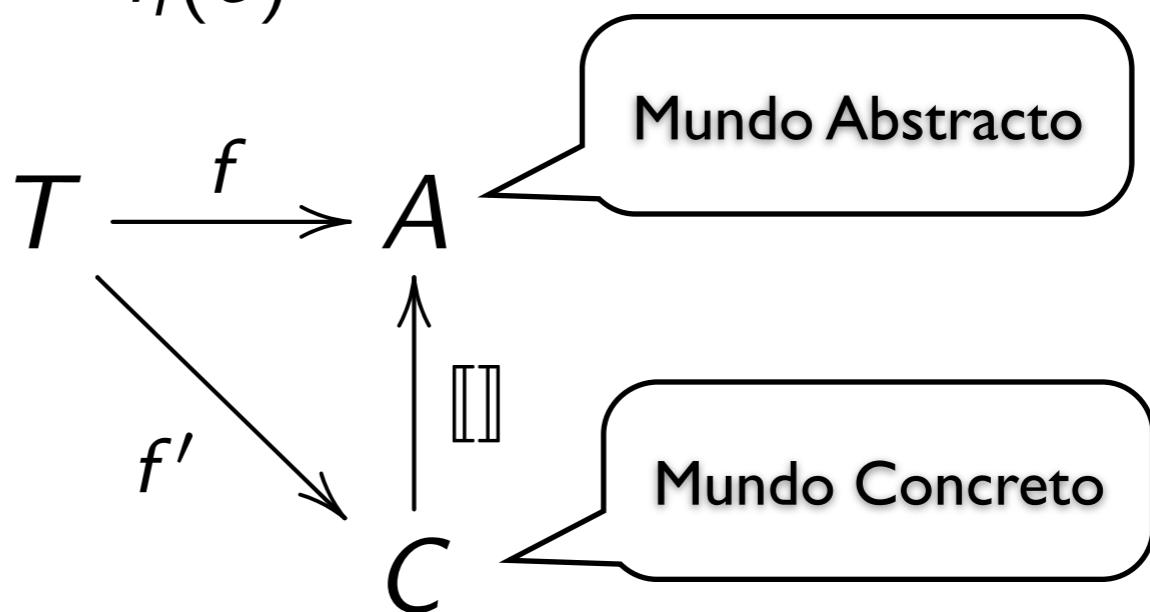
Para cada operación generadora del álgebra abstracta:

$$f : T \rightarrow A$$

Debe haber una operación generadora del álgebra concreta, tal que:

$$[f'_i(c)] = f_i(c)$$

En diagramas:



# Operaciones Modificadoras

Para cada operación modificadora abstracta:

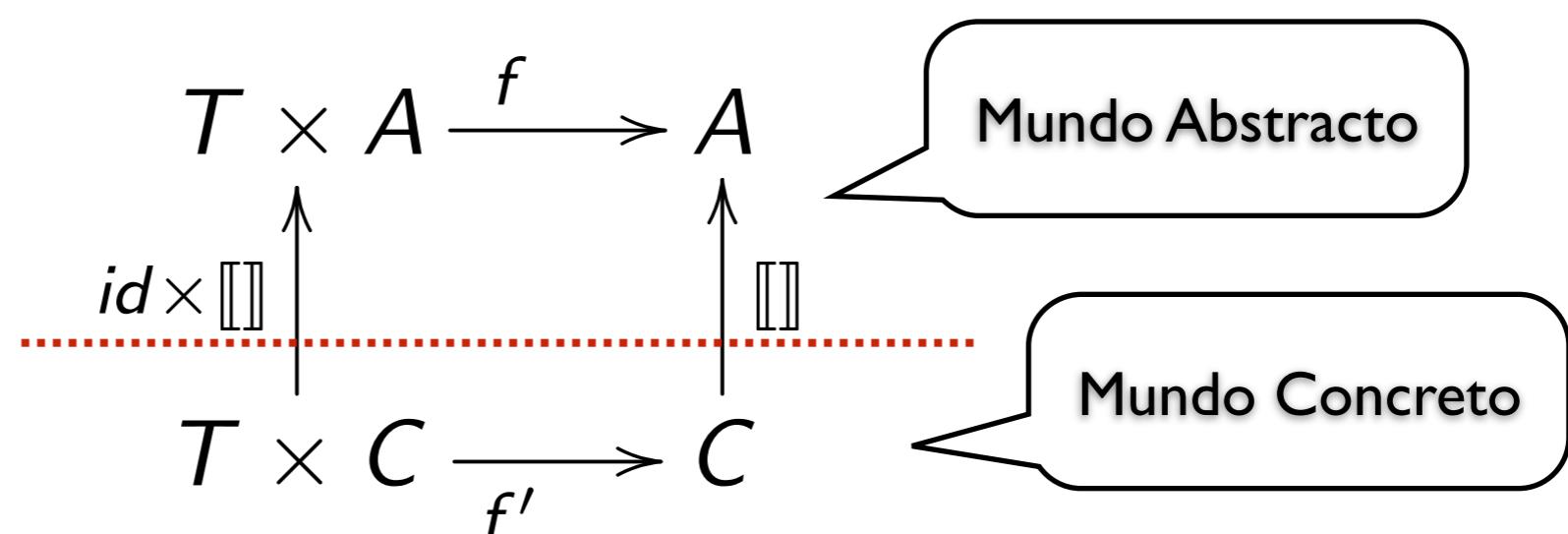
$$f : T \times A \rightarrow A$$

Tiene que haber una modificadora concreta:

$$f' : T \times C \rightarrow C$$

Tal que:  $\llbracket f'(x, c) \rrbracket = f(x, \llbracket c \rrbracket)$

O en diagramas:



# Operaciones Observadoras

Para cada operación observadora abstracta:

$$f : S \times A \rightarrow T$$

Debe haber una operación observadora concreta:

$$f' : S \times C \rightarrow T$$

Tal que:  $f'(x, c) = f(x, [c])$

En diagramas:

$$\begin{array}{ccc} S \times A & \xrightarrow{f} & T \\ id \times [] \uparrow & & \nearrow f' \\ S \times C & & \end{array}$$

# Ejemplo

Supongamos el álgebra de los conjuntos:

$$\langle \{A\}, \cup, \cap, \in, \emptyset \rangle$$

Podemos implementarla con listas:

Conjuntos  
representados por  
listas

$$\langle [A], \overline{\cup}, \overline{\cap}, \overline{\in}, \overline{\emptyset} \rangle$$

Implementación de  
las operaciones

Definida  
recursivamente

Función de abstracción:

$$[\![[]]\!] = \emptyset$$

$$[\![x : xs]\!] = \{x\} \cup [\![xs]\!]$$

# Ejemplo (cont.)

Debemos dar la implementación de cada operación, por ejemplo:

$$[] \overline{\cup} ys = ys$$

Repite elementos!

$$(x : xs) \overline{\cup} ys = x : (xs \overline{\cup} ys)$$

Deberíamos demostrar:

$$[xs \overline{\cup} ys] = [xs] \cup [ys]$$

$$[xs \overline{\cap} ys] = [xs] \cap [ys]$$

Es decir, las operaciones comutan

$$x \overline{\in} xs = x \in [xs]$$

# Otro Ejemplo

Consideremos el álgebra de los enteros

$$\langle \text{Int}, 0, +, -(unaria) \rangle$$

Podemos implementarla con los naturales:

$$\langle (\text{Nat}, \text{Nat}), \bar{0}, \oplus, \ominus, \rangle$$

La función de abstracción es:

$$[(m, n)] = m - n$$

# Ejemplo (cont.)

Las operaciones podemos definirlas como:

$$(m, n) \oplus (m', n') = (m + m', n + n')$$

$$\ominus(n, m) = (m, n)$$

$$\bar{0} = (0, 0)$$

Demostremos la corrección de la suma:

$$\begin{aligned}\llbracket (n, m) \oplus (p, q) \rrbracket \\ = [\text{def. } \oplus]\end{aligned}$$

$$\begin{aligned}\llbracket (n + p, m + q) \rrbracket \\ = [\text{def. } \llbracket \rrbracket]\end{aligned}$$

$$\begin{aligned}n + p - (m + q) \\ = [\text{Arit.}]\end{aligned}$$

$$\begin{aligned}n - m + p - q \\ = [\text{def. } \llbracket \rrbracket]\end{aligned}$$

$$\llbracket (n, m) \rrbracket + \llbracket (p, q) \rrbracket$$

# Ejemplo III

Supongamos que queremos implementar los booleanos:

$$\langle \text{Bool}, \text{true}, \text{false}, \wedge, \vee, \neg \rangle$$

Los implementaremos con los naturales:

$$\langle \text{Nat}, \overline{\text{true}}, \overline{\text{false}}, \overline{\wedge}, \overline{\vee}, \overline{\neg} \rangle$$

En donde:

$$[\![n]\!] = (n \neq 0)$$

Cualquier numero distinto de 0 representa true, y 0 representa false

# Ejemplo (III)

Definamos las operaciones:

$$\overline{\text{true}} = 8$$

$$\overline{\text{false}} = 0$$

$$p \overline{\wedge} q = p * q$$

$$p \overline{\vee} q = p + q$$

$$\overline{\neg} p = \text{if } p = 0 \rightarrow 1$$

$$\quad \square p \neq 0 \rightarrow 0$$

Ejercicio: Demostrar su corrección.

# Propiedades Formales

Dada una función de abstracción:

$$[\![\quad]\!] : C \rightarrow A$$

Suryectiva

Hay una relación de equivalencia asociada:

$$x \equiv_{[\![\quad]\!]} y \equiv [\![x]\!] = [\![y]\!]$$

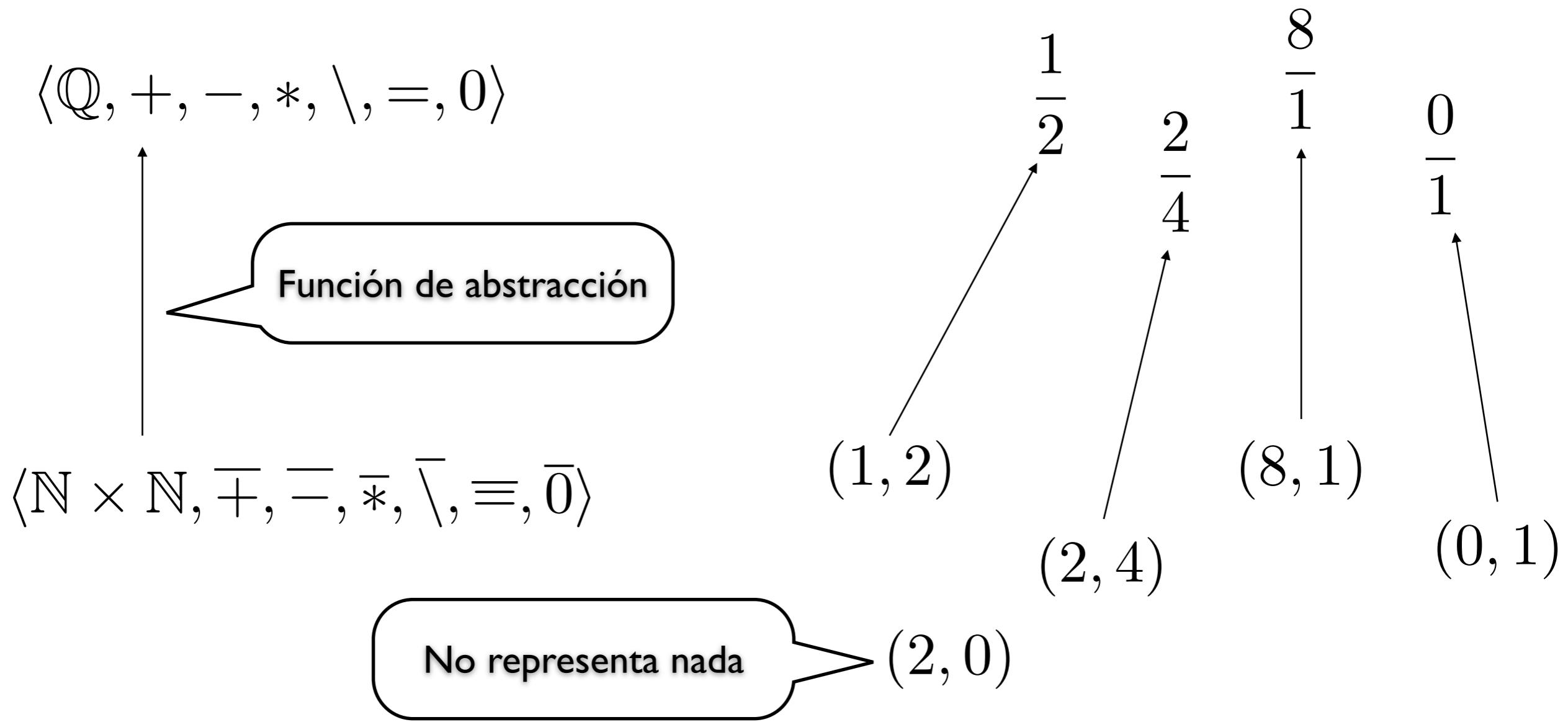
Y una álgebra modulo esta equivalencia:

$$C \setminus \equiv_{[\![\quad]\!]}$$

Que se comporta igual que el álgebra abstracta.

# Invariantes de Representación

Consideremos el siguiente ejemplo:



# Invariantes de Representación

Un invariante de representación es un predicado sobre el álgebra concreta que permite caracterizar aquellos elementos que representan elementos del álgebra abstracta

Es decir, es una función:

$$inv : C \rightarrow \text{Bool}$$

# Invariantes de Representación

Los invariantes de representación deben cumplir ciertos requisitos:

- Para cada operación generadora:  $g : T \rightarrow C$   
 $\forall t \in T : inv(g(t))$   
se generan elementos validos
- Para cada modificadora:  $m : T \times C \rightarrow C$   
 $\forall c \in C, t \in T : inv(c) \Rightarrow inv(m(t, c))$   
se retornan elementos validos
- Además:  
 $\forall a \in A : \exists c \in C : inv(c) \wedge [c] = a$

# Ejemplo

Retomando el ejemplo de los racionales:

$$[(n, d)] = \frac{n}{d}$$

El invariante es:

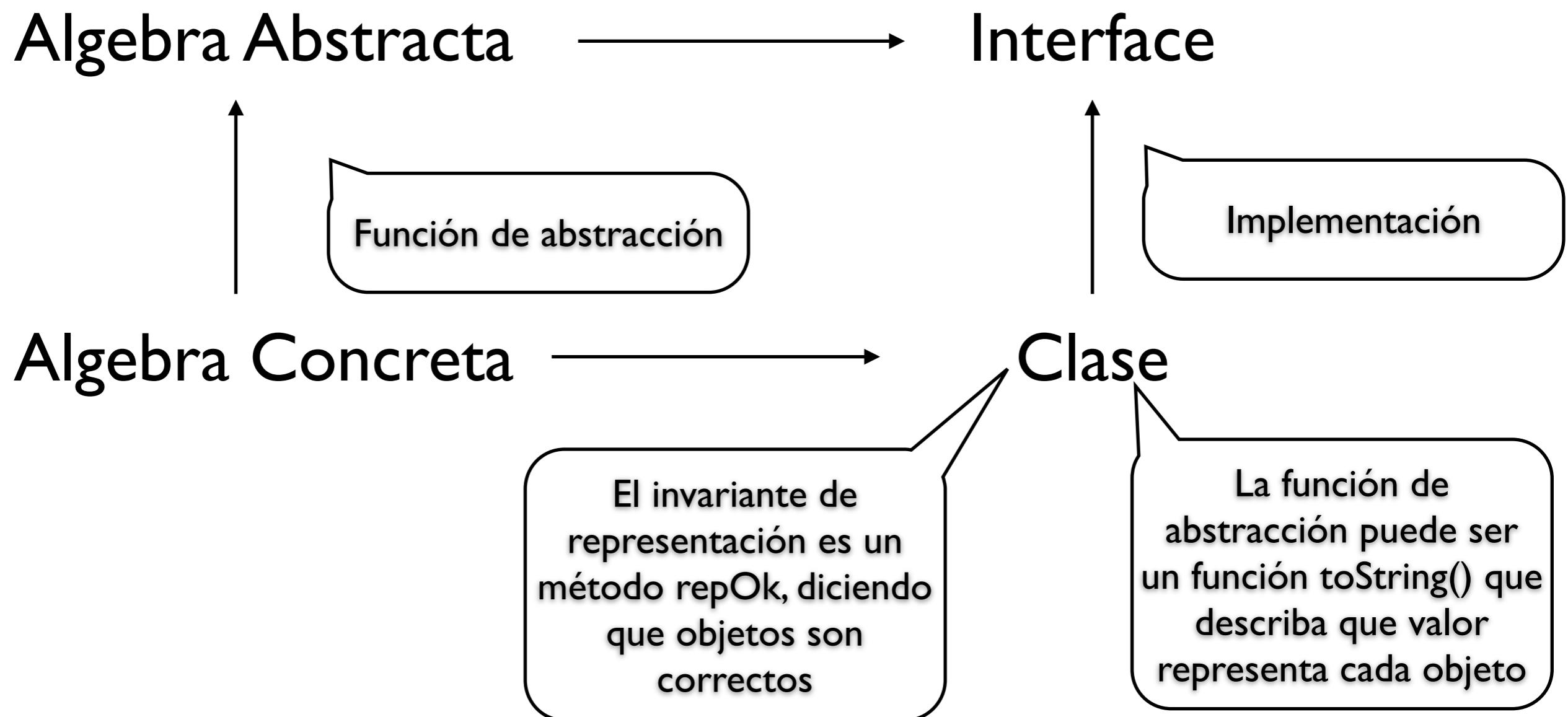
$$inv(n, d) = d \neq 0$$

Ejercicio: definir las operaciones.

Solo los pares cuyo segundo elemento es diferente a 0, representan racionales

# Implementación

En Java tenemos la siguiente correspondencia:



# Un Ejemplo

## Consideremos los racionales:

```
/**  
 * Una interface para racionales, para ilustrar las noción de función de abstracción  
 * @author Pablo Castro  
 */  
  
public interface Racional{  
  
    /**  
     * Operación para suma racionales  
     */  
    public void suma(Racional r);  
  
    /**  
     * Operación para multiplicación de racionales  
     */  
    public void mult(Racional r);  
  
    /**  
     * operación para resta de racionales  
     */  
    public void neg();  
  
    /**  
     * Operación para dividir racionales  
     */  
    public void div(Racional r);  
}
```

Podemos agregar todas las operaciones que queramos

# Una Implementación

```
public class RacionalPar implements Racional{  
    private int num; // el numerador del racional  
    private int den; // el denominador  
  
    /**  
     * Un constructor basico para racionales  
     * @pre d != 0  
     */  
    public RacionalPar(int n, int d){  
        this.num = n;  
        this.den = d;  
    }  
  
    /**  
     * Observadora, retorna el numerador  
     */  
    public int getNum(){  
        return num;  
    }  
  
    /**  
     * Observadora retorna el denominador  
     */  
    public int getDen(){  
        return den;  
    }  
}
```

Implementamos racionales con pares

Métodos observadores

# Una Implementación (I)

```
/**  
 * Suma un racional al actual  
 * @param r el racional para sumar  
 * @pre r debe ser RacionalPar y r.num !=0  
 * @return la suma  
 */  
public void suma(Racional r){  
    if (r instanceof RacionalPar){  
        RacionalPar param = (RacionalPar) r;  
        this.num = (this.num * param.getDen()) + (param.getNum() * this.den);  
        this.den = this.den * param.getDen();  
    }  
    else  
        throw new RacionalExpcion();  
}
```

Implementamos cada una de las operaciones

Hay que tener cuidado con los tipos que se devuelven y de los parámetros!

El chequeo de tipos en tiempo de ejecución se puede evitar usando genericidad

# Una Implementación (II)

```
/**  
 * La funcion de abstraccion  
 * @return el numero que representa el objeto actual  
 */  
public String toString(){  
    return String.valueOf((float) this.num / (float) this.den);  
}  
  
/**  
 * El invariante de representacion  
 * @return true cuando el segundo componente del par es diferente a 0  
 */  
public boolean repOk(){  
    return (den != 0);  
}  
}// end of class
```

toString es una forma de describir la función de abstracción

repOk es el invariante de clase

# Implementación con Genericidad

```
/**  
 * Una interface simple de los racionales usando  
 * genericidad  
 */  
  
public interface Racional<T extends Racional>{  
  
    /**  
     * Operacion para suma racionales  
     */  
    public void suma(T r);  
  
    /**  
     * Operacion para multiplicacion de racionales  
     */  
    public void mult(T r);  
  
    /**  
     * operacion para resta de racionales  
     */  
    public void neg();  
  
    /**  
     * Operacion para dividir racionales  
     */  
    public void div(T r);  
  
}
```

Usamos genericidad para evitar hacer castings

# Implementación con Genericidad II

```
/**  
 * Una implementacion de los racionales con genericidad  
 */  
  
public class RacionalPar implements Racional<RacionalPar>{  
    private int num; // el numerador  
    private int den; // el denominador  
  
    // Implementar el resto de los metodos  
    /**  
     * Multiplica un racional al actual  
     * @param r el racional para multiplicar  
     * @pre r debe ser RacionalPar y r.num !=0  
     * @return la multiplicacion  
     */  
    public void mult(RacionalPar r){  
        this.num = this.num * r.getNum();  
        this.den = this.den * r.getDen();  
    }  
    /**  
     * La funcion de abstraccion  
     * @return el numero que representa el objeto actual  
     */  
    public String toString(){  
        return String.valueOf((float) this.num / (float) this.den);  
    }  
    /**  
     * El invariante de representacion  
     * @return true cuando el segundo componente del par es diferente a 0  
     */  
    public boolean repOk(){  
        return (den != 0);  
    }  
}
```

No necesitamos hacer castings