Teoría de TADs

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

Tipos y Algebras

Un algebra es un conjunto soporte más operaciones:

Conjunto soporte

$$\langle \mathbb{N}, +, 0 \rangle$$

Operaciones

Las operaciones son funciones:

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

Pueden ser operaciones parciales

$$0 :\to \mathbb{N}$$

Las constantes se pueden ver como operaciones sin parámetros

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 polimorficas de tipo A

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

Conjuntos polimorficos de tipo A

Algebras Heterogéneas

Muchas veces utilizaremos algebras que tienen varios conjuntos soportes

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

Conjunto de interes

Conjunto auxiliar

$$\in: \{A\} \times A \to Bool$$

Retorna un elemento booleano que no pertenece al conjunto soporte

Clases de Operaciones

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

$$\emptyset :\to \{A\}$$

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

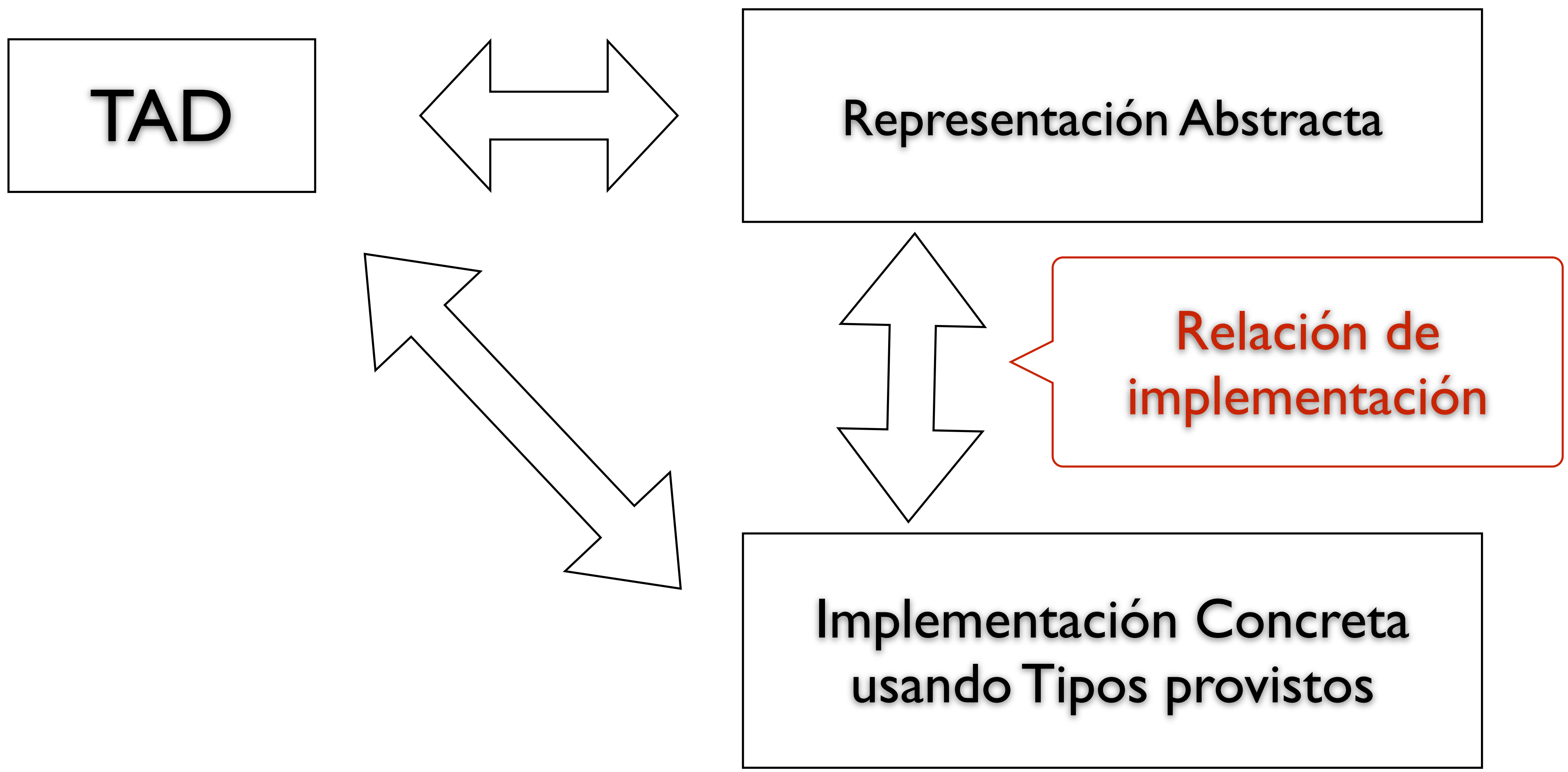
$$\in: \{A\} \times A \to Bool$$

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

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

Implementación de TADS

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



Para demostrar que implementamos el TAD correctamente, debemos probar los items descritos a continuación

Implementación de TADS

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

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

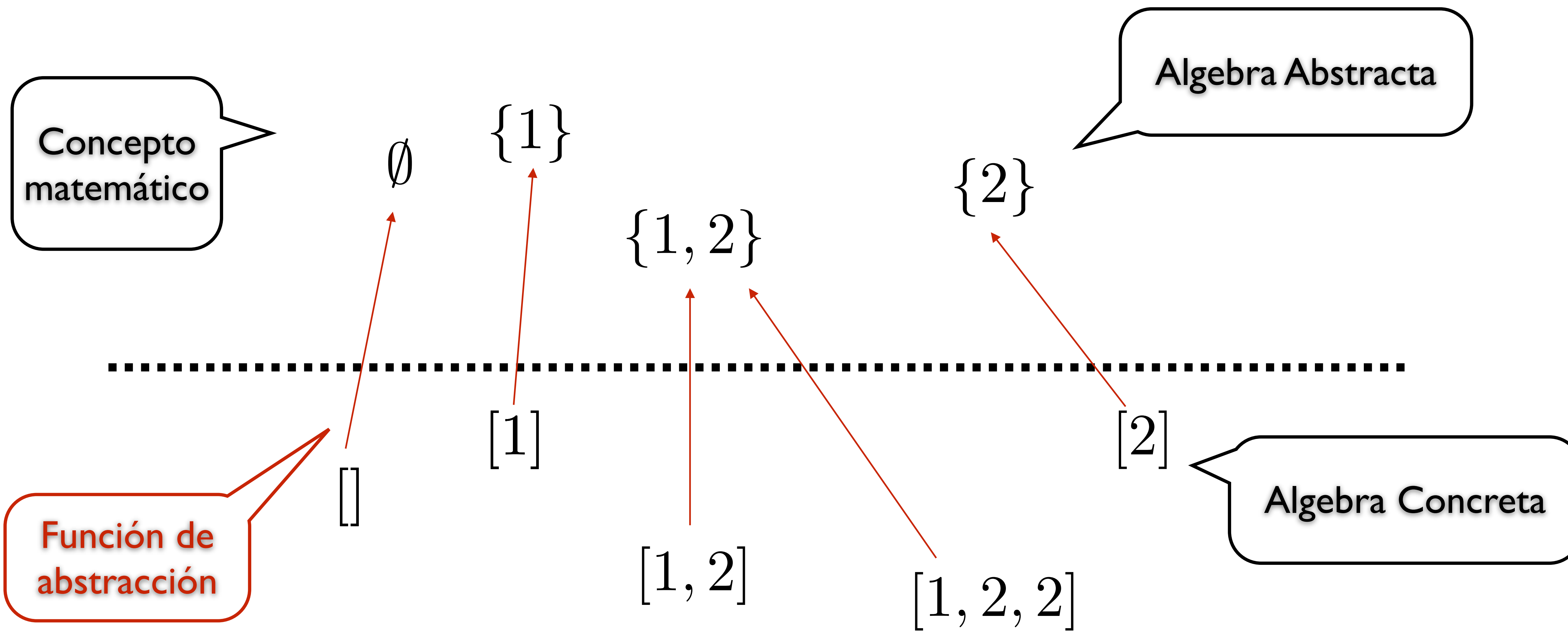
$$[\![-]\!] : C \rightarrow A$$

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:

$$[\![-]\!] : C \to A$$

Debe cumplir:

- Es suryectiva: $\forall a \in A, \exists c \in C \,|\, [\![c]\!] = a$
- Cada operación del algebra abstracta
  tiene una correspondiente en el algebra concreta

(y cumplir los requisitos a continuación)

Función de Abstracción: Ejemplo

Supongamos el algebra 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:

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

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

Operaciones Generadoras

Para cada operación generadora del algebra abstracta:

$$f : T \to A$$

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

$$\llbracket f_i'(c) \rrbracket = f_i(c)$$

En diagramas:

$$\begin{tikzcd}
T \arrow[r, "f"] \arrow[dr, "f'"'] & A \\
& C \arrow[u, "\llbracket\rrbracket"']
\end{tikzcd}$$

Mundo Abstracto

Mundo Concreto

Operaciones Modificadoras

Para cada operación modificadora abstracta:

$$f : T \times A \to A$$

Tiene que haber una modificadora concreta:

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

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

O en diagramas:

$$\begin{tikzcd} T \times A \arrow[r, "f"] & A \\ T \times C \arrow[u, "id \times [\![]\!]"] \arrow[r, "f'"'] & C \arrow[u, "[\![]\!]"'] \end{tikzcd}$$

Mundo Abstracto

Mundo Concreto

Operaciones Observadoras

Para cada operación observadora abstracta:

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

Debe haber una operación observadora concreta:

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

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

En diagramas:

$$\begin{tikzcd} S \times A \arrow[r, "f"] & T \\ S \times C \arrow[u, "id \times [\![]\!]"] \arrow[ur, "f'"'] & \end{tikzcd}$$

Ejemplo

Supongamos que queremos implementar los booleanos:

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

Los implementaremos con los naturales:

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

En donde:

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

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

Ejemplo

Definamos las operaciones:

$$
\begin{aligned}
\overline{true} &= 8 \\
\overline{false} &= 0 \\
p \overline{\wedge} q &= p * q \\
p \overline{\vee} q &= p + q \\
\overline{\neg} p &= if\ \ p = 0 \rightarrow 1 \\
&\qquad\ \ \square\ p \neq 0 \rightarrow 0
\end{aligned}
$$

Ejercicio: Demostrar su corrección.

Ejemplo

Supongamos el algebra de los conjuntos:

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

Podemos implementarla con listas:

Conjuntos representados por listas

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

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:

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

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

Repite elementos!

Deberíamos demostrar:

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

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

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

Es decir, las operaciones conmutan

Otro Ejemplo

Consideremos el algebra de los enteros

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

Podemos implementarla con los naturales:

$$\langle (\mathit{Nat}, \mathit{Nat}), \overline{0}, \oplus, \ominus, \rangle$$

La función de abstracción es:

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

Ejemplo (cont.)

Las operaciones podemos definirlas como:

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

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

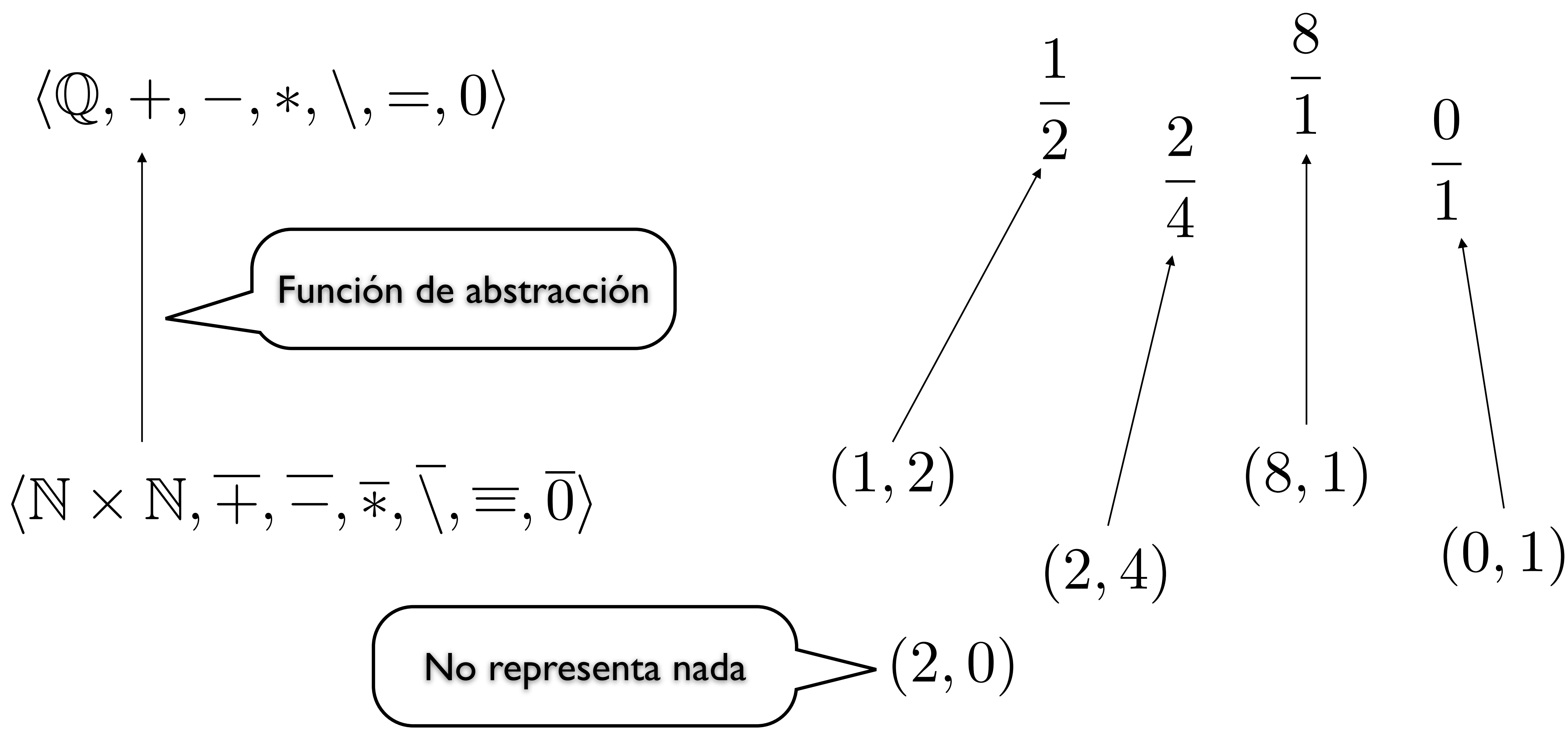
$$\overline{0} = (0, 0)$$

Demostremos la corrección de la suma:

$$\begin{array}{c}
[\![(n, m) \oplus (p, q)]\!] \\
= [\text{def.}\oplus] \\
[\![(n + p, m + q)]\!] \\
= [\text{def.}[\![]\!]] \\
n + p - (m + q) \\
= [\text{Arit.}] \\
n - m + p - q \\
= [\text{def.}[\![]\!]] \\
[\![(n, m)]\!] + [\![(p, q)]\!]
\end{array}$$

Invariantes de Representación

Consideremos el siguiente ejemplo:



Invariantes de Representación

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

Es decir, es una función:

$$inv : C \to Bool$$

Invariantes de Representación

Los invariantes de representación deben cumplir ciertos requisitos:

- Suryectividad:

$$\forall a \in A : \exists c \in C : inv(c) \wedge [\![c]\!] = a$$

- Para cada operación generadora: $g : T \to C$

$$\forall t \in T : inv(g(t))$$

se generan elementos validos

- Para cada modificadora: $m : T \times C \to C$

$$\forall c \in C, t \in T : inv(c) \Rightarrow inv(m(t, c))$$

Para aquellos parámetros que m esté definida

se retornan elementos validos

Ejemplo

Retomando el ejemplo de los racionales:

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

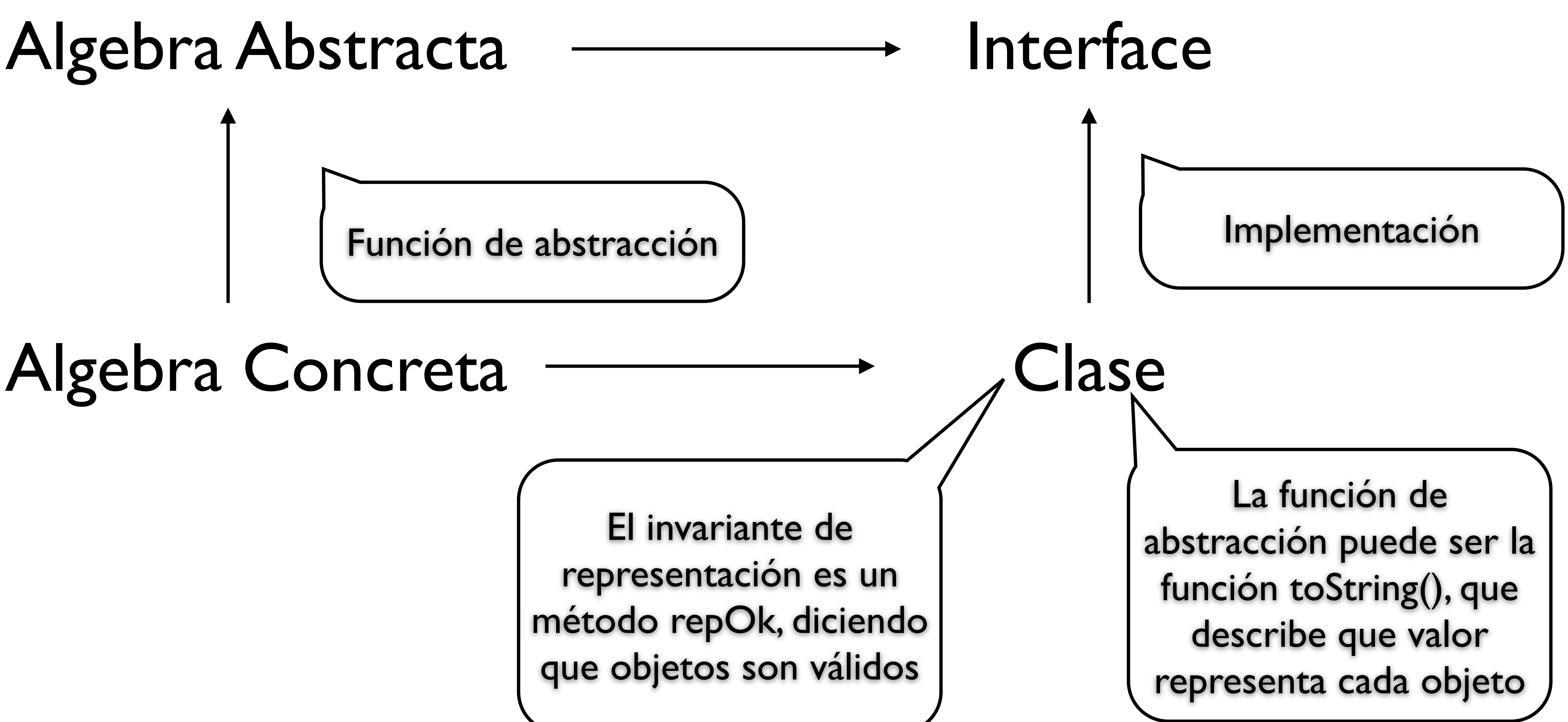
El invariante es:

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

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

Ejercicio: Demostrar la corrección de las operaciones, y que cumplen con inv

En Java tenemos la siguiente correspondencia:



Un Ejemplo

Consideremos los racionales:

```
public interface Racional {
    /**
    * Retorna el numerador
    */
    public int getNum()
    /**
    * Retorna el denominador
    */
    public int getDen()
    /**
    *   Operacion para suma racionales
    */
    public void suma(Racional r);
    /**
    *   Operacion para multiplicacion de racionales
    */
    public void mult(Racional r);
    /**
    *   operacion para resta de racionales
    */
    public void neg();
    /**
    *   Operacion para dividir racionales
    */
    public void div(Racional r);
}
```

Una Implementación

```java
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

Implementamos cada una de las operaciones

```
/**
* Suma un racional al actual
*/
public void suma(Racional r){
        this.num = (this.getNum() * r.getDen()) + (r.getNum() * this.getDen());
        this.den = this.getDen() * r.getDen();
}
```

Una Implementación

toString es una forma de describir la función de abstracción

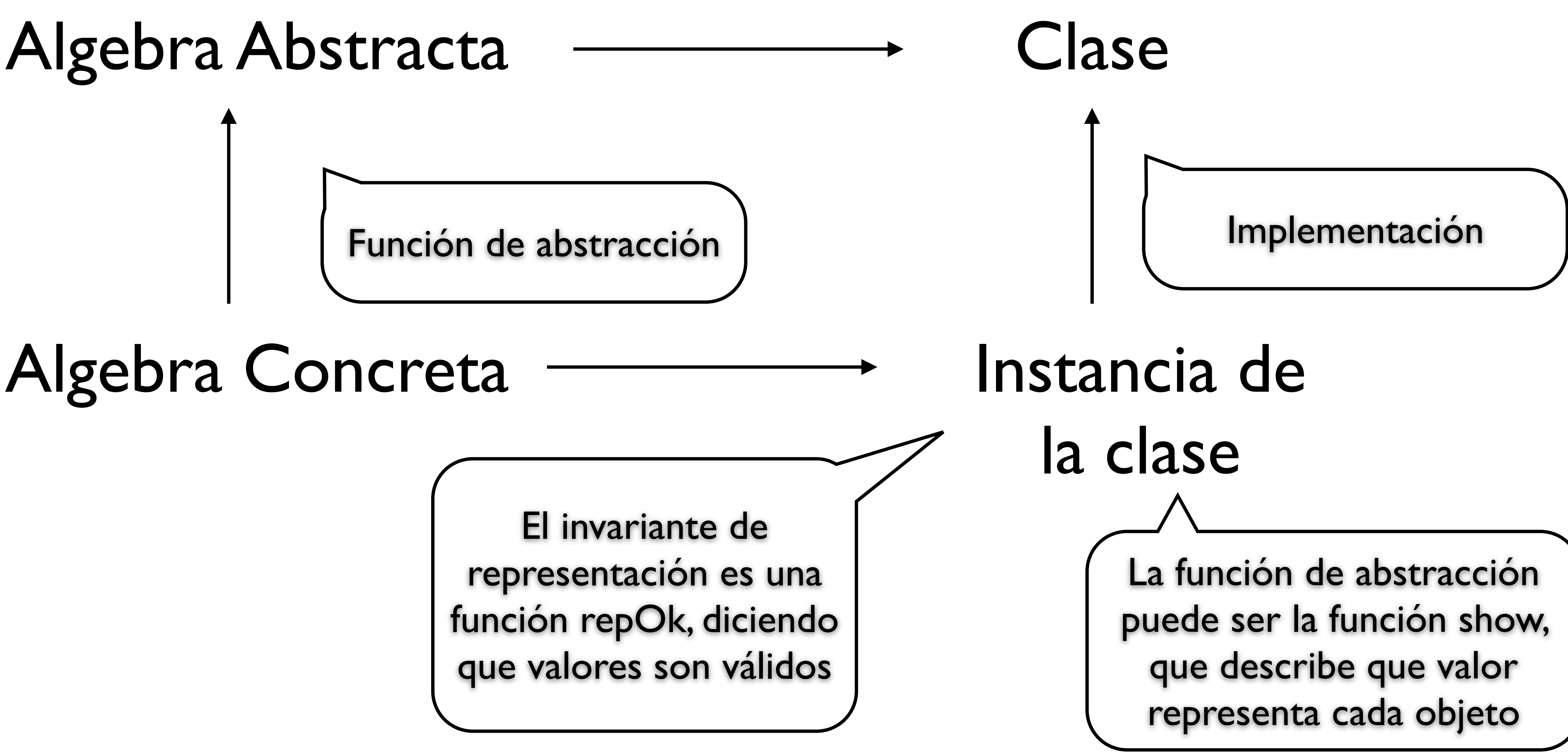
```
    /**
     * La funcion de abstraccion
     */
    public String toString() {
        return this.num + "/" + this.den;
    }

    /**
     * El invariante de representacion
     */
    public boolean repOk(){
        return (den != 0);
    }

}// end of class
```

repOk es el invariante de clase

En Haskell tenemos la siguiente correspondencia:



Implementación en Haskell

En Haskell podemos usar las clases de tipos para definir tipos abstractos, y dar implementaciones con instancias

Implementa la clase Fractional (Racionales) ya definida en Haskell

```
module FracPar where
import Data.Ratio

-- | definimos un tipo nuevo para representar los racionales
data FracPar = Integer :/ Integer deriving Show -- | escribir un mejor Show!

instance Num FracPar where  -- | Implementamos la clase Num, ya que los Frac son Num
    (p :/ q) + (n :/ m) = ((p*m)+(n*q)) :/ (q*m)
    (p :/ q) * (n :/ m) = (p*n) :/ (q*m)
    negate (p :/ q) = (- p) :/ q
    (p :/ q) - (n :/ m) = (p :/ q) + (negate (n :/ m))
    abs (p :/ q) = (abs p) :/ (abs q)
    fromInteger p = (p :/ 1)
    signum (p :/ q) = ((signum p) * (signum q)) :/ 1

instance Fractional FracPar where -- | Implementamos la division
    (p :/ q) / (n :/ m) = (p*m) :/ (q*n)
    fromRational r =  (numerator r) :/ (denominator r)
```

Actividades

- Leer el apunte "Tipos de datos". J. Blanco.

Bibliografía

- "Tipos de datos". J. Blanco.