Capítulo 1

ALGORITMOS DE CÁLCULO DE BASES

En esta sección se incluyen los pseudocódigos de los algoritmos implementados para el cálculo de bases directas, por el grupo de investigación GIMAC, en colaboración con la Dra. K. Bertet de la Universidad de La Rochelle. Uno de los objetivos de este proyecto es implementar estos algoritmos en Java siguiendo las estructuras de datos que están implementadas en el repositorio `https://github.com/kbertet/java-lattices` ya que es probable que en un futuro dichos algoritmos se incluyan en esta librería.

La estructura principal usada es la clase *ImplicationalSystem* que representa un conjunto o sistema de implicaciones e implementa las operaciones clásicas sobre éstos: sitema propio, base canónica, base canónica directa, sistema unario, etc.

ImplicationalSystem está compuesto por un conjunto de elementos (atributo `set` en la Figura 1.1) y un conjunto de implicaciones representadas mediante la clase *Rule*. Ésta a su vez está compuesta por dos conjuntos de atributos que representan la premisa y la conclusión de la implicación e implementa métodos para la obtención de cada una de las partes así como para la adición y eliminación de elementos de éstas. En la Figura 1.1 se muestra el diagrama de clases con los métodos y relaciones más relevantes:

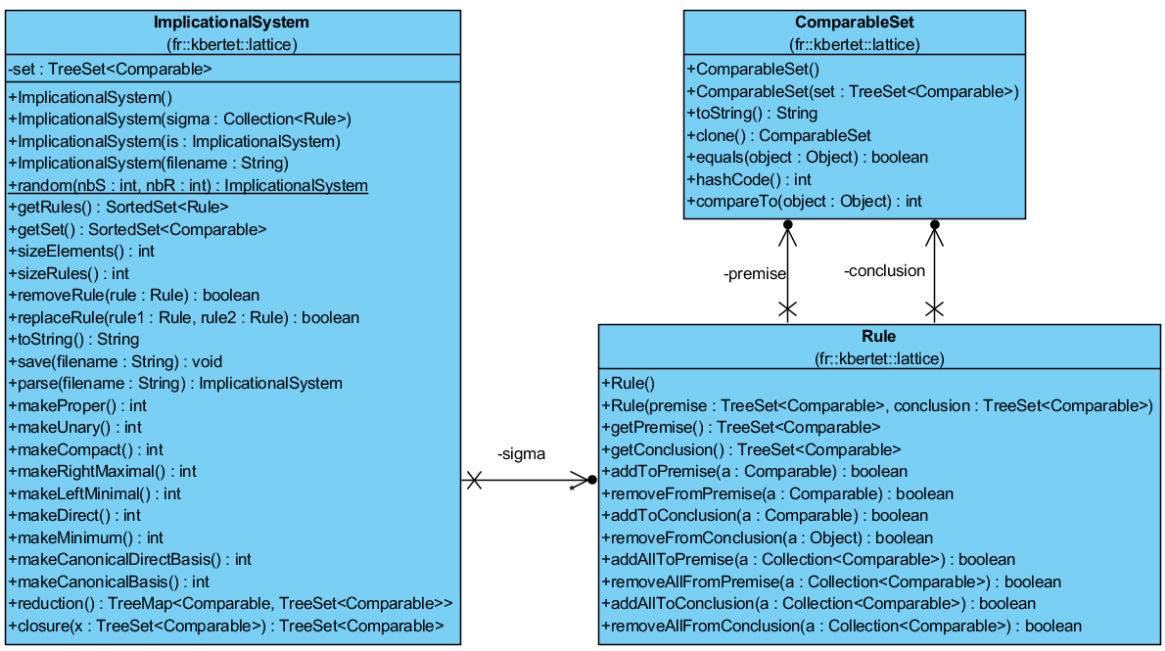

Figura 1.1: Clases librería java-lattice

Un ejemplo del uso de estos tipos, se puede ver en el cálculo de la cardinalidad de un sistema implicacional:

```java
/**
 * Number of system implications.
 * @param system Implicational System.
 * @return Implications number. {@code null} when the syste is null.
 */
public static final Integer getCardinality(ImplicationalSystem system) {
    Integer cardinality = null;
    if (system != null) {
        cardinality = system.sizeRules();
    }
    return cardinality;
}
```

Figura 1.2: Cálculo de la cardinalidad de un sistema implicacional

O en el cálculo de su tamaño:

```java
/**
 * Sum of the number of attributes of right and left side for all implications.
 * @param system Implicational system.
 * @return The sum of the number of attributes of right and left side for all
 * implications.
 * <br/> {@code null} when the system is null.
 */
public static final Integer getSize(ImplicationalSystem system) {
    Integer size = null;

    if (system != null) {
        size = 0;
        for (Rule implication : system.getRules()) {
            size += implication.getPremise().size() + implication.getConclusion().size();
        }
    }
    return size;
}
```

Figura 1.3: Cálculo del tamaño de un sistema implicacional

1.1. CLA

A continuación se muestra el algoritmo CLA presentado en el artículo [**?**], definiéndose primeramente las funciones *Direct-Reduced* y *RD-Simplify* que después son usadas en el algoritmo. Para abreviar, se hará referencia al sistema implicacional con las siglas en inglés IS (Implicational System).

La función *Direct-Reduced(*Σ*)* calcula el IS directo-reducido de Σ:

Direct-Reduced(Σ)

---

input: Un sistema implicacional Σ en S
output: El IS directo-reducido Σ_{dr} en S
begin
  foreach $A \to B \in \Sigma_{dr}$ y $C \to D \in \Sigma_{dr}$ **do**
    if $B \cap c \neq \emptyset \neq D \backslash (A \cup B)$ **then** add $AC - B \to D - (AB)$ to Σ_{dr};
  return Σ_{dr}

La función *RD-Simplify(*Σ*)* calcula el IS directo-reducido-simplificado a partir de Σ reducido:

RD-Simplify(Σ)

---

input: Un sistema implicacional directo-reducido Σ en S
output: El IS directo-reducido-simplificado Σ_{drs} en S equivalente a Σ
begin
  $\Sigma_{drs} := \emptyset$
  foreach $A \to B \in \Sigma$ **do**
    foreach $C \to D \in \Sigma$ **do**
      if $C = A$ **then** $B := B \cup D$;
      if $C \nsubseteq A$ **then** $B := B \backslash D$;
    if $B \neq \emptyset$ **then** add $A \to B$ to Σdrs;
  return Σ_{drs}

La función *doSimp(*Σ*)* calcula el IS directo-óptimo equivalente a Σ usando las dos funciones anteriores:

doSimp(Σ)

input: Un sistema implicacional Σ en S

output: El IS directo-óptimo Σ_{do} en S

begin

$\Sigma_r := \{A \to B - A | A \to B \in \Sigma, B \not\subseteq A\}$

Σ_{dr} := Direct-Reduced(Σ_r)

Σ_{do} := RD-Simplify(Σ_{dr})

return Σ_{do}

1.2. Direct Optimal Basis

A continuación se muestra el algortimo Direct Optimal Basis presentado en el artículo [**?**]. La diferencia con el algoritmo CLA, es que se hace una simplificación izquierda y derecha antes de aplicar la regla Strong Simplification.

input: Un sistema implicacional Σ en S
output: El IS directo-óptimo Σ_{do} en S
1 **begin**
2 /* Fase 1: Generación de Σ_r por reducción de Σ*/
3 $\Sigma_r = \emptyset$
4 **foreach** $A \rightarrow_\Sigma B$ **do**
5 **if** $B \nsubseteq A$ **then** add $A \rightarrow B - A$ to Σ_r;
6 /* Fase 2: Generación de Σ_{sr} por simplificación de Σ_r*/
7 $\Sigma_{sr} = \Sigma_r$
8 **repeat**
9 **foreach** $A \rightarrow B \in \Sigma_{sr}$ **do**
10 **foreach** $C \rightarrow D \in \Sigma_{sr}$ **do**
11 **if** $A \subseteq C$ **then**
12 **if** $C \subseteq A \cup B$ **then**
13 replace $A \rightarrow B$ and $C \rightarrow D$ by
$A \rightarrow BD$ in Σ_{sr};
14 **else if** $D \subseteq B$ **then**
15 remove $C \rightarrow D$ from Σ_{sr}
16 **else** replace $C \rightarrow D$ by
$C - B \rightarrow D - B$ inΣ_{sr};
17 **until** Σ_{sr} es un punto fijo;
18 /*Fase 3: Generación de Σ_{dsr} por Strong Simplification de Σ_{sr}*/
19 $\Sigma_{dsr} = \Sigma_{sr}$
20 **foreach** $A \rightarrow B \in \Sigma_{dsr}$ and $C \rightarrow D \in \Sigma_{dsr}$ **do**
21 **if** $B \cap C \neq \emptyset \neq D \backslash (A \cup B)$ **then**
22 add $AC - B \rightarrow D - (AB)$ to Σ_{dsr}
23 /*Fase 4: Generación de Σ_{do} por optimización de Σ_{dsr}*/
24 $\Sigma_{do} = \emptyset$
25 **foreach** $A \rightarrow B \in \Sigma_{dsr}$ **do**
26 **foreach** $C \rightarrow D \in \Sigma_{dsr}$ DO
27 **if** $C = A$ **then** $B = B \cup D$;
28 **if** $C \nsubseteq A$ **then** $B = B \backslash D$;
29 **if** $B \neq \emptyset$ **then** add $A \rightarrow B$ to Σ_{do};
30 **return** Σ_{do}

1.3. Implementaciones

En esta sección se describe la implementación de los algoritmos anteriores en Java y un ejemplo de uso. Ambos algoritmos implementan la interfaz *es.uma.pfc.is.algorithms.Algorithm* para que puedan ser ejecutadas por IS Bench. En la siguiente figura, se muestra el diagrama que muestra la jerarquía de clases implementada.

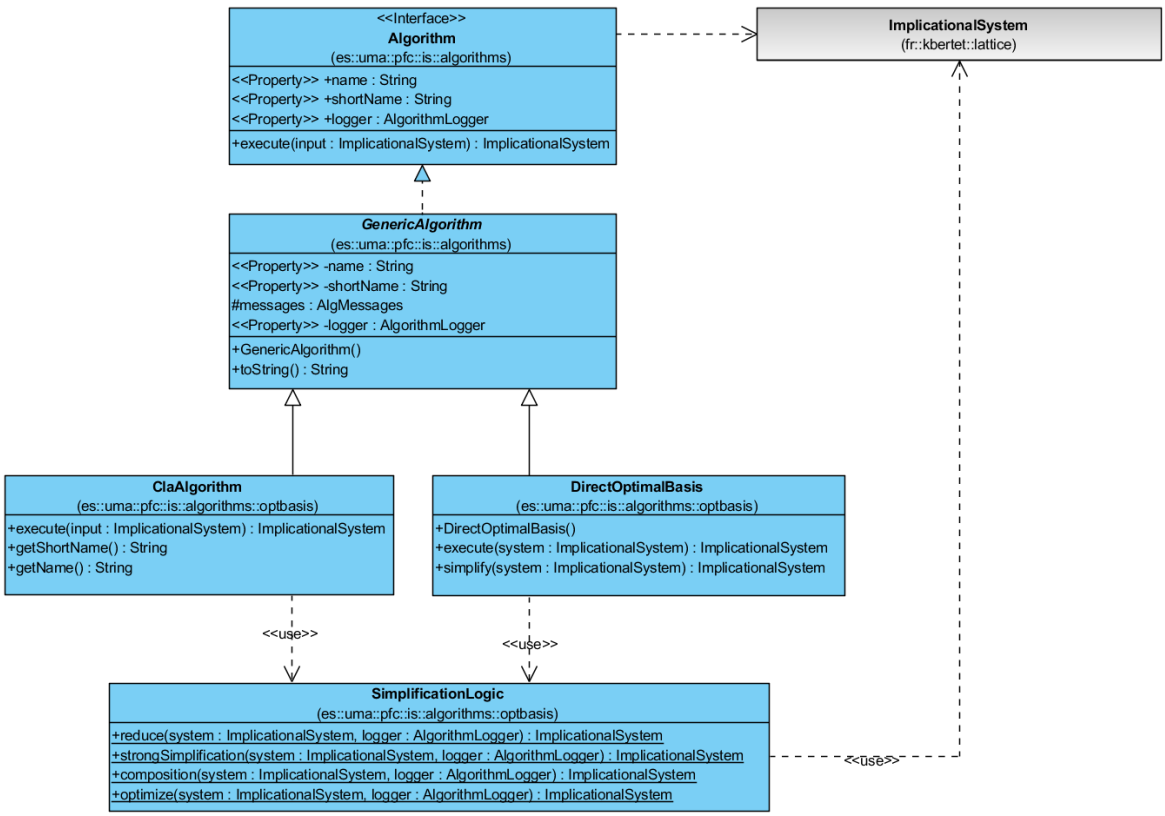

Figura 1.4: Implementación de Algoritmos

Los algoritmos tienen un parámetro de entrada de tipo *ImplicationalSystem*, que es el sistema implicacional inicial y devuelve un valor del mismo tipo con el sistema implicacional final. Ambos hacen uso de la clase *es.uma.pfc.is.algorithms.optbasis.SimplificationLogic* donde se implementan las reglas de equivalencia de la Lógica de Simplificación (Simplification Logic) mencionada en el artículo [**?**].

CLA

Clase: es.uma.pfc.is.algorithms.optbasis.ClaAlgorithm

```
public class ClaAlgorithm extends GenericAlgorithm {
     @Override
    public ImplicationalSystem execute(ImplicationalSystem input) {
        ImplicationalSystem directOptimalBasis = new ImplicationalSystem(input);
        directOptimalBasis = SimplificationLogic.reduce(
```

```
            directOptimalBasis, getLogger());
        directOptimalBasis = SimplificationLogic.strongSimplification(
            directOptimalBasis, getLogger());
        directOptimalBasis = SimplificationLogic.composition(
            directOptimalBasis, getLogger());
        directOptimalBasis = SimplificationLogic.optimize(
            directOptimalBasis, getLogger());
        return directOptimalBasis;
    }
    @Override
    public String getShortName() {
        return "cla";
    }
    @Override
    public String getName() {
        return "CLA";
    }
}
```

Direct Optimal Basis

Clase: es.uma.pfc.is.algorithms.optbasis.DirectOptimalBasis

```
public class DirectOptimalBasis extends GenericAlgorithm {
    /**
     * Executes the Direct Optimal Basis algorithm.
     * @param system Input system.
     * @return Direct optimal basis.
     */
    @Override
    public ImplicationalSystem execute(ImplicationalSystem system) {
        ImplicationalSystem directOptimalBasis = new ImplicationalSystem(system);
        // Stage 1 : Generation of sigma-r by reduction of sigma
        directOptimalBasis = SimplificationLogic.reduce(
            directOptimalBasis, getLogger());
        // Stage 2: Generation of sigma-sr by simplification
        // (left+right) + composition of sigma-r
        directOptimalBasis = simplify(directOptimalBasis);
        // Stage 3: Generation of sigma-dsr by completion of sigma-sr
        directOptimalBasis = SimplificationLogic.strongSimplification(
            directOptimalBasis, getLogger());
        // Stage 4: Composition of sigma-dsr
```

```
        SimplificationLogic.composition(directOptimalBasis, getLogger());
        // Stage 5: Generation of sigma-do by optimization of sigma-dsr
        directOptimalBasis = SimplificationLogic.optimize(
             directOptimalBasis, getLogger());
        return directOptimalBasis;
    }
    @Override
public String getShortName() {
return "do";
}
@Override
public String getName() {
return "Direct Optimal Basis";
}
  }
```