Capítulo 1

Introducción a la programación con restricciones

Hacemos un pequeño resumen de los conceptos más destacados de programación lógica para motivar después su generalización con restricciones.

1.1 Repaso de programación lógica

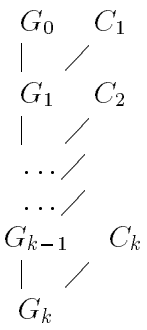
Comenzamos recordando los conceptos de cláusula de Horn y de cálculo de resolución SLD, así como la propiedad de ser correcto y completo restringido a cláusulas de Horn. Mostramos ejemplos de derivaciones y refutaciones en resolución SLD, y definimos los espacios de búsqueda de refutaciones SLD, distinguiendo entre cálculo SLD y los procedimientos resultantes de combinar el cálculo con diferentes estrategias de búsqueda.

Una **cláusula** es **de Horn** si al considerarla como conjunto de literales tiene a lo sumo un literal positivo; o equivalentemente, si al escribirla en notación de Kowalski toma una de estas cuatro formas:

1. $P \leftarrow$: **Hecho**
2. $P \leftarrow Q_1, \ldots, Q_n$: **Regla**
3. $\emptyset \leftarrow R_1, \ldots, R_m$: **Objetivo**
4. $\emptyset \leftarrow \emptyset$: **Exito**

Llamaremos cláusulas **definidas** a los hechos y las reglas. Nótese que un conjunto π de cláusulas definidas siempre admite como modelo la base de Herbrand B_π. Si $G \equiv \emptyset \leftarrow R_1, \ldots, R_m$ es un objetivo entonces $\pi \cup \{ G \}$ puede ser insatisfactible. El problema de refutar $\pi \cup \{ G \}$ equivale al problema de demostrar que la sentencia **existencial** $(R_1 \wedge \ldots \wedge R_m)^{\exists}$ es consecuencia lógica de π; de ahí la terminología "Objetivo" y "èxito" (u objetivo alcanzado, cuando se completa la refutación derivando $\emptyset \leftarrow \emptyset$). El nombre "regla" viene de entender la cláusula $P \leftarrow Q_1, \ldots, Q_n$ como una regla para reducir P a $(Q_1 \wedge \ldots \wedge Q_n)$; los "hechos" son las reglas incondicionales.

Una **función de selección** φ asocia a cada objetivo $G :\leftarrow R_1, \ldots, R_m$ un índice $\varphi(G) = i \in \{1, \ldots, m\}$ y a R_i le llamamos **átomo seleccionado** por φ. Sean un conjunto de cláusulas definidas π, un objetivo G y una función de selección φ. Una **derivación SLD** basada en π, con función de selección φ y objetivo inicial G es un árbol de la forma:



siendo $G_0 = G$ y para cada $1 \leq i \leq k$, C_i es una variante de una regla de π sin variables comunes con $G_1, \ldots, G_{i-1}$, ni con $C_1, \ldots, C_{i-1}$, y G_i resulta como resolvente general de G_{i-1} y C_i mediante unificación de la cabeza de C_i con el átomo de G_{i-1} seleccionado por φ.

Si $\varphi(G_{i-1}) = j$, la regla C_i se aplica al objetivo G_{i-1}, reduciéndolo al nuevo objetivo G_i:

$$
\begin{array}{rl}
G_{i-1} & = \leftarrow\ R_1, \ldots, R_{j-1},\ \mathbf{R_j},\ R_{j+1}, \ldots, R_m \\
 & \quad | \\
 & \quad |\mathbf{P}\ \leftarrow\ Q_1, \ldots, Q_n\ =\ C_i \\
 & \quad | \qquad\qquad\qquad\quad \diagup \\
 & \quad |\ \sigma_i = u.m.g.\{R_j,\ P\} \\
 & \quad | \qquad\quad \diagup \\
 & \quad | \qquad \diagup \\
G_i & = \leftarrow\ (R_1, \ldots, R_{j-1},\ Q_1, \ldots, Q_n,\ R_{j+1}, \ldots, R_m)\sigma_i
\end{array}
$$

Llamamos **respuesta** obtenida a partir de G con ayuda del programa π, mediante resolución SLD con la función de selección φ, a la sustitución $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ aplicada a la conjunción del cuerpo del objetivo inicial. Si la respuesta contiene variables, éstas se interpretan como cuantificadas universalmente. En este caso la respuesta obtenida es: $[(R_1 \wedge \ \ldots \ \wedge R_m)\sigma]^{\forall}$. Una misma respuesta puede ser calculada empleando diferentes funciones de selección.

Consideremos un conjunto π de cláusulas definidas y un objetivo G. Si fijamos una función de selección φ, en general, habrá varias (variantes de) cláusulas de π cuya cabeza sea unificable con el átomo de G seleccionado por φ, y lo mismo sucederá para cada uno de los nuevos objetivos resultantes de aplicar estas reglas. Así, resulta un árbol cuyos nodos son objetivos y cuyas ramas son todas las posibles derivaciones SLD basadas en π con objetivo inicial G y función de selección φ. A este árbol se le llama **espacio de búsqueda** SLD para π, G y φ. (Notación: $SLD(\pi, G, \varphi)$). Estos árboles no deben confundirse con los árboles lineales que representan derivaciones SLD.

Teorema de corrección y completitud para la resolución SLD

Sea π un conjunto de cláusulas definidas y $G = \leftarrow R_1, \ldots, R_m$ un objetivo:

(a) Toda respuesta $(R_1 \wedge \ldots \wedge R_m)\sigma$ calculada mediante SLD-refutación basada en π, es correcta; es decir, cualquiera que sea la función de selección empleada: $\pi \models [(R_1 \wedge \ldots \wedge R_m)\sigma]^{\forall}$.

(b) Si $\pi \cup \{ G \}$ es insatisfactible, entonces hay una SLD-refutación de $\pi \cup \{ G \}$, cualquiera que sea la función de selección empleada.

Lema: **Si** tenemos un paso de resolución SLD con $\varphi(G_{i-1}) = j$,

$$\begin{array}{rl} G_{i-1} & = \leftarrow R_1, \ldots, R_{j-1}, \mathbf{R_j}, R_{j+1}, \ldots, R_m \\ & \quad | \\ & \quad | \qquad\qquad P \leftarrow Q_1, \ldots, Q_n \; = \; C_i \\ & \quad | \qquad\qquad\quad \diagup \\ & \quad |\sigma_i \; = \; u.m.g.\{ R_j, \; P \} \\ & \quad | \qquad\qquad\quad \diagup \\ G_i & = \leftarrow (R_1, \ldots, R_{j-1}, \mathbf{Q_1}, \ldots, \mathbf{Q_n}, R_{j+1}, \ldots, R_m)\sigma_i \end{array}$$

entonces todo modelo de la cláusula C_i es modelo de la implicación:
$\bigwedge(R_1, \ldots, R_{j-1}, Q_1, \ldots, Q_n, R_{j+1}, \ldots, R_m)\sigma_i \longrightarrow \bigwedge(R_1, \ldots, R_{j-1}, R_j, R_{j+1}, \ldots, R_m)\sigma_i$

Demostración de la parte (a): **Sea** $I \models \pi$ **entonces** $I \models C_i$, para toda $C_i \in \pi$.

Supongamos que la derivación que genera la respuesta es la siguiente:

$$\begin{array}{rl} & G_0 \quad C_1 \\ \sigma_1 & | \quad \diagup \\ & G_1 \quad C_2 \\ \sigma_2 & | \quad \diagup \\ & \ldots \diagup \\ & \ldots \diagup \\ \sigma_{k-1} & | \quad \diagup \\ & G_{k-1} \quad C_k \\ \sigma_k & | \quad \diagup \\ & G_k \end{array}$$

Utilizando en cada paso de la derivación el lema anterior enunciado en la forma: **Si** $I \models C_i$ **entonces** $I \models (\bigwedge G_i \rightarrow \bigwedge G_{i-1}\sigma_i)$ y, por tanto, para toda sustitución θ, se cumple $I \models (\bigwedge G_i \rightarrow \bigwedge G_{i-1}\sigma_i)\theta$.

Aplicamos el lema para $i = k$ y obtenemos que $I \models (\bigwedge \emptyset \rightarrow \bigwedge G_{k-1}\sigma_k)$, es decir, $I \models \bigwedge G_{k-1}\sigma_k$.

Aplicándolo de nuevo para $i = k - 1$ y particularizando el resultado para $\theta = \sigma_k$, obtenemos que $I \models (\bigwedge G_{k-1} \rightarrow \bigwedge G_{k-2}\sigma_{k-1})\sigma_k$, luego $I \models \bigwedge G_{k-2}\sigma_{k-1}\sigma_k$.

Reiterando el proceso hasta la raiz de la derivación, para i = 0, obtenemos finalmente que $I \models \bigwedge G_0\sigma$, es decir, $\pi \models [(R_1 \wedge \ldots \wedge R_m)\sigma]^{\forall}$. ■

En la parte (b), se utilizan una serie de definiciones y propiedades:

Definición: Sea $T: A \to A$ un operador definido sobre el retículo completo $(A, \ll)$:

1. T es un operador **monotono** sii para todo I, J se cumple: Si $I \ll J$ entonces $T(I) \ll T(J)$.

2. T es un operador **finitario** sii para toda sucesión infinita $\{I_j | j \in \mathbb{N}\}$ se cumple: $T(\bigcup I_j) \ll \bigcup T(I_j)$.

3. T es un operador **continuo** sii es finitario y monotono sii para toda sucesión infinita $\{I_j | j \in \mathbb{N}\}$ se cumple $T(\bigcup I_j) = \bigcup T(I_j)$.

4. Las potencias de un operador se definen como: $T^0(I) = I$, $T^{n+1}(I) = T(T^n(I))$ y $T^\omega(I) = \bigcup T^n(I)$.

5. Un **operador continuo** admite un **menor punto fijo** que coincide con T^ω. □

Definición: Para cualquier programa π y para toda interpretación de Herbrand $I \subseteq B_\pi$ definimos:

1. **Operador de consecuencia inmediata** $T_\pi : \mathcal{P}(B_\pi) \longrightarrow \mathcal{P}(B_\pi)$
   $A \in T_\pi(I) \Leftrightarrow_{def}$ Existe una sustitución μ y existe una cláusula $C \in \pi$ con $C \equiv (P \leftarrow (Q_1, \ldots, Q_n))$ tal que: $A \equiv P\mu$ y $I \models \bigwedge(Q_1, \ldots, Q_n)\mu$.

2. **Mínimo modelo de Herbrand de** π es $M_\pi = T_\pi^\omega(\emptyset)$, es decir, el modelo de Herbrand que parte de la mínima información, pues $I = \emptyset \subseteq B_\pi$.

3. **El conjunto de los átomos básicos refutables a partir de** π es:
   $SS(\pi) = \{\text{A básicos} — \pi \cup \{\leftarrow A\} \text{ tiene una SLD-refutación}\}$. □

Propiedades de T_π:
(a) T_π es finitario y monotono, luego T_π es un operador continuo.
(b) $I \models \pi \Leftrightarrow T_\pi(I) \subseteq I$.
(c) Existe M_π y coincide con $SS(\pi)$.
Demostración:
(a) Finitario: Dada la sucesión infinita $\{I_j | j \in \mathbb{N}\}$ tenemos que demostrar que:
$T_\pi(\bigcup I_j) \subseteq \bigcup T_\pi(I_j)$.
Sea $A \in T_\pi(\bigcup I_j)$ entonces existen μ y $P \leftarrow (Q_1, \ldots, Q_n) \equiv C \in \pi$ tal que $A \equiv P\mu$ y además $(\bigcup I_j) \models \bigwedge(Q_1, \ldots, Q_n)\mu$, es decir, deberá existir un m con $I_m \models \bigwedge(Q_1, \ldots, Q_n)\mu$, por lo cual $A \in T_\pi(I_m)$ y por tanto $A \in \bigcup T(I_j)$.
(a) Monotono: Si $I \subseteq J$ entonces $T_\pi(I) \subseteq T_\pi(J)$. En efecto, si $A \in T_\pi(I)$ entonces existen μ y $P \leftarrow (Q_1, \ldots, Q_n) \equiv C \in \pi$ con $A \equiv P\mu$ y $I \models \bigwedge(Q_1, \ldots, Q_n)\mu$, como $I \subseteq J$ también $J \models \bigwedge(Q_1, \ldots, Q_n)\mu$, y por tanto $A \in T_\pi(J)$ como queriamos demostrar.
Por tanto, T_π es un operador continuo y existe siempre **menor punto fijo** que coincide con T_π^ω.

(b) $\Rightarrow$) Si $A \in T_\pi(I) \Leftrightarrow_{def}$ Existe una μ y una $P \leftarrow (Q_1, \ldots, Q_n) \equiv C \in \pi$ tal que $A \equiv P\mu$ y además $I \models \bigwedge(Q_1, \ldots, Q_n)\mu$. Como I es modelo de π, tenemos que $I \models A \equiv P\mu$ y por tanto $A \in I$.
(b) $\Leftarrow$) Sea $(P \leftarrow (Q_1, \ldots, Q_n)) \equiv C \in \pi$ una cláusula cualquiera, si $I \models \bigwedge(Q_1, \ldots, Q_n)$,

entonces elegimos una sustitución μ tal que $P\mu$ sea una instancia básica de P, luego también $I \models \bigwedge(Q_1, \ldots, Q_n)\mu$ y por tanto $P\mu \in T_\pi(I) \subseteq I$, así pues, $I \models P\mu$. Así pues, $I \models \pi$, por ser modelo de todas las instancias básicas de sus cláusulas.
Por **(a) + (b)**, si $I \models \pi$ entonces $T_\pi^\omega(I) \subseteq I \models \pi$, luego $T_\pi^\omega(I)$ está contenido en los I modelos de π.

(c) Existe el mínimo punto fijo del operador T_π definible a partir de la mínima información $I = \emptyset \subseteq B_\pi$ y denotado como M_π, que es el menor modelo de π, por estar contenido en todo modelo del programa.
$SS(\pi) \subseteq M_\pi$: Si $A \in SS(\pi)$ entonces $\pi \cup \{\leftarrow A\}$ tiene una SLD-refutación, luego por el teorema de corrección (apartado (a)) sabemos que $\pi \models A$, y por tanto, $M_\pi \models A$, es decir, $A \in M_\pi$.
$M_\pi \subseteq SS(\pi)$: Si $A \in M_\pi$ entonces existe un $k > 0$ tal que $A \in T_\pi^k(\emptyset)$. Se demuestra por inducción sobre $k > 1$, que $A \in SS(\pi)$.
Luego: $M_\pi = SS(\pi)$. ■

Ejemplo: Sea el programa π = {`arco(a, b)` $\leftarrow$; `camino(x, x)` $\leftarrow$;
`camino(x, y)` $\leftarrow$ `arco(x, z)`, `camino(z, y)`.} Las potencias de T_π son: $T_\pi^0(\emptyset) = \emptyset$;
$T_\pi^1(\emptyset)$ = {`arco(a, b)`, `camino(a, a)`, `camino(b, b)`};
$T_\pi^2(\emptyset)$ = {`arco(a, b)`, `camino(a, a)`, `camino(b, b)`, `camino(a, b)`}
y a partir de T_π^2, todas las potencias coinciden y el mínimo modelo de Herbrand es:
{`arco(a, b)`, `camino(a, a)`, `camino(b, b)`, `camino(a, b)`}.
En este caso, se comprueba fácilmente que $SS(\pi)$ coincide con M_π.

Demostración de la parte (b): Sea $G = \leftarrow R_1, \ldots, R_m$ con **Insat**$(\pi \cup \{G\})$ entonces $M_\pi \not\models G$. Como $G \equiv \forall(\neg R_1 \vee \ldots \vee \neg R_m)$, debe existir una sustitución μ tal que $\{R_1\mu, \ldots, R_m\mu\} \in M_\pi$, por el lema anterior $\{R_1\mu, \ldots, R_m\mu\} \in SS(\pi)$, es decir, existe una SLD refutación de $\pi \cup \{\leftarrow R_i\mu\}$, para $i \in \{1, \ldots, m\}$, y por tanto, existe una SLD refutación de $\pi \cup \{\leftarrow (R_1, \ldots, R_m)\mu\}$, que puede elevarse a una SLD refutación de $\pi \cup \{\leftarrow (R_1, \ldots, R_m)\}$ como deseabamos comprobar. ■

Teorema de independencia con respecto a la función de selección Dos espacios de búsqueda SLD que solo se diferencien en la función de selección $SLD(\pi, G, \varphi_1)$ y $SLD(\pi, G, \varphi_2)$ contienen siempre los mismos EXITOS, que proporcionan - salvo variantes - respuestas idénticas, pero ambos espacios pueden diferenciarse en el número de FALLOS, la longitud de las derivaciones SLD representadas por sus ramas, y la presencia o no de algunas derivaciones infinitas. ■

Definición: D es un **dominio de terminación** para un predicado P sii
$SLD(\pi, \leftarrow P(t_1, \ldots, t_n), \varphi)$ es finito para todo $P(t_1, \ldots, t_n) \in D$.

Ejemplo: Si la especificación de un programa para la suma con c y s como constructoras es: E(Suma) = {Suma($s^m(c)$, $s^n(c)$, $s^k(c)$) : $m, n \in \mathbb{N}$ y $k = m + n$}.

$$\mathrm{E}_k(\mathrm{Suma}) = \{\mathrm{Suma}(s^m(c),\ s^n(c),\ s^k(c)) \ :\ m, n \in \mathbb{N} \text{ y } n < k\}.$$

Un posible programa que lo define será:

`suma(X, c, X)`$\leftarrow$ `suma(X, s(Y), s(Z))`$\leftarrow$`suma(X, Y, Z).`

Algunos dominios de terminación son:
D1 = {Suma($s^m(c)$, $s^n(c)$, Z) : $m, n \in \mathbb{N}$} D2 = {Suma(X, $s^n(c)$, $s^k(c)$) : $n < k$}

D3 = {Suma(X, Y, $s^k(c)$) : $k \in \mathbb{N}$} D4 = {Suma($s^k(c)$, Y, $s^k(c)$) : $k \in \mathbb{N}$}

1.2 Limitaciones de la programación lógica

Normalmente se afirma como una propiedad esencial de la programación lógica la ecuación:

LOGICA + CONTROL = QUE + COMO

La idea expresa que la lógica se responsabiliza de la corrección del programa, estableciendo únicamente relaciones sintácticas entre los objetos a manipular y combinar para obtener el resultado esperado, mientras que el control se responsabiliza de la eficiencia ocupándose de las estratégias utilizadas en la combinación y manipulación que se efectue sobre los objetos.

La metodología ideal en programación lógica es:

- QUE: Lógica de primer orden.
- COMO: Resolución SLD + Búsqueda en profundidad de izqda a drcha.

Limitaciones de la programación lógica:

- Los objetos manipulados son estructuras sin interpretación semántica, es decir, términos formados con los símbolos del programa.
- La igualdad entre objetos es la identidad sintáctica, basada en constructoras libres, sin consideración semántica alguna.
- El control está basado en una regla simple y uniforme, que genera de forma natural problemas de divergencia o búsqueda infinita.

Propuesta de la programación lógica con restricciones:

- Utilización explícita de las propiedades semánticas de los objetos del dominio concreto en el que trabajamos.
- La igualdad entre objeto se transforma de sintáctica en semántica.
- Se incorporan técnicas de consistencia para superar las limitaciones del control.
  - Prpagación local de valores.
  - Computación guiada por el flujo de datos.
  - Comprobación prematura de condiciones para determinar la consistencia.

Con esta propuesta se superarán algunas limitaciones de la programación lógica:

- Se sustituye el mecanismo sintáctico de la unificación por el mecanismo semántico de la satisfactibilidad de un conjunto finito de restricciones.
- El esquema $CLP(X)$ sustituye cada X por $\mathcal{R}$, $\mathcal{Z}$, $\mathcal{Q}$, etc.
- Aplicación de las técnicas de consistencia para obtener rápidez en el cómputo con variables lógicas.
- El árbol de búsqueda se poda integrando la evaluación de restricciones y la búsqueda indeterminista para lograr un proceso de cómputo determinista.

Ejemplo: Nos interesa mejorar la **declaratividad** de la programación lógica. Como deseamos trabajar en $\mathcal{N}$ con el programa `suma(X,Y,Z) :- X + Y =:= Z`. Para mantener el sentido declarativo del programa, su comportamiento debería ser:

- ?– suma(2,3,5) Comprueba si la relación es cierta
- ?–suma(2,3,Z) Calcula la suma de 2 y 3
- ?– suma(2,Y,5) Halla un Y que sumado con 2 de 5
- ?– suma(X,3,5) Halla un X que sumado con 3 de 5
- ?– suma(X,Y,5) Descompone 5 como suma de X e Y

Para que el programa sea declarativo tendríamos que escribir varias cláusulas preguntando si las variables están o no instanciadas y luego usar la primitiva **is**, además el método usado generaría primero los valores de los números y luego comprobaría que la relación se verifica, es decir, el esquema de programación utilizado sería **Genera y Comprueba**.

Ejemplo: Ahora deseamos mejorar la eficiencia de la programación lógica. Sea el programa formado por los hechos: `v(3). v(2). v(1)`, y las reglas `mayor(X, Y):- X>Y.` `triple(X, Y, Z):- v(X), v(Y), v(Z), mayor(X, Y), mayor(Y, Z).` Para el objetivo `?-triple(A, B, C)` se genera el siguiente cómputo:

```
?-triple(A, B, C)
| A := X, B := Y, C := Z
?- v(X), v(Y), v(Z), mayor(X, Y), mayor(Y, Z).
| A := 3, B := Y, C := Z
?-v(Y), v(Z), mayor(3, Y), mayor(Y, Z).
| A := 3, B := 2, C := Z
?- v(Z), mayor(3, 2), mayor(2, Z).
| A := 3, B := 2, C := 1
?- mayor(3, 2), mayor(2, 1).
| A := 3, B := 2, C := 1
?- mayor(2, 1).
| A := 3, B := 2, C := 1
?- EXITO
```

PROLOG puede verse como un lenguaje de programación lógica con restricciones, puesto que si modificamos el orden de los hechos, el orden de los predicados del programa o del objetivo se producen reducciones de tiempo y memoria, por ejemplo, el programa:

```
tripleinv(X, Y, Z) :- mayor(X, Y), mayor(Y, Z), v(X), v(Y), v(Z).
```

que expresa lo mismo que el programa `triple` desde el punto de vista lógico, podría ser tratado **invernando** sus variables hasta conseguir valores **buenos** con los que avanzar sin realizar búsquedas ni retrocesos evitables.

Ejemplo: Una de las aplicaciones más frecuentes de PLR es la resolución de problemas combinatorios, dónde lo que se busca es la posibilidad de **existencia de soluciones** y no la forma más eficiente de calcularlas. Este es uno de los campos de experimentación más prometedores, puesto que la explosión combinatoria que se genera al resolver problemas reales con un gran número de variables se trata de manera muy elegante y eficiente con este tipo de lenguajes de programación.

Deseamos resolver dos problemas combinatorios, de características similares, las variables que aparecen representan cifras que han de tomar valores distintos entre sí.

$$\mathbf{A\ B\ C} + \mathbf{A\ D} = \mathbf{B\ C\ A}$$
$$\mathbf{S\ E\ N\ D} + \mathbf{M\ O\ R\ E} = \mathbf{M\ O\ N\ E\ Y}$$

Sus soluciones son `[A, B, C, D] = [5, 6, 1, 4]` y `[S, E, N, D, M, O, R, Y] = [9, 5, 6, 7, 1, 0, 8, 2]`, y para obtenerlas programamos siguiendo la metodología:

1. Definición del dominio de las variables.
   ```
   dominio([ ]).
   dominio([F | T]) :- F ∈ [0..9] □ dominio(T).
   ```

2. Restricciones que han de verificar dichas variables.
   ```
   rest([A, B, C, D]) :- A ≠ 0, B ≠ 0, S ∈ [0..1] , R ∈ [0..1],
                         C + D = A + 10*R, R + B + A = C + 10*S, S + A = B,
                         diflist([A, B, C, D]).
   restrict([S, E, N, D, M, O, R, Y]) :- S ≠ 0, M ≠ 0,
                                         1000*S + 100*E + 10*N + D
                                         + 1000*M + 100*O + 10*R + E
                                         = 10000*M + 1000*O + 100*N + 10*E + Y,
                                         diflist([S, E, N, D, M, O, R, Y]).
   ```
   siendo
   ```
   diflist([ ]).
   diflist([X | T]) :- no-miembro(X, T), diflist(T).
   ```

3. Generación de valores concretos.
   ```
   generaval([ ]).
   generaval([X|T]) :- miembro(X, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
                       generaval(T).
   ```

Por tanto, los programas buscados son:

1. `cripto([A, B, C, D]):- dominio([A, B, C, D]), rest([A, B, C, D]), generaval([A, B, C, D]).`

La evaluación se realiza de la siguiente manera: `A` $\neq$ `B` y `B = A + S` implica `S = 1`. Con ello, las ecuaciones que quedan son:

- `B = A + 1`, lo que implica que `A` $\neq$ `9`
- `R + A + 1 + A = C + 10`, lo que implica que `C = 2*A + R -9`
- `2*A + R - 9 + D = A + 10*R`, lo que implica que `D = 9*R + 9 - A`

Si `R = 1` **entonces** las ecuaciones resultantes son `B = A + 1, C = 2*A - 8, D = 18 - A`, que junto con las condiciones `A` $\neq$ `9, A` $\in$ `[0..9]` implican contradicción.
Si `R = 0` **entonces** las ecuaciones son `B = A + 1, C = 2*A - 9, D = 9 - A` y las soluciones:

- **Si** `A` $\in$ $\{$`1, 2, 3, 4`$\}$ **entonces** `C` < 0
- **Primera solución:** `A = 5, B = 6, C = 1, D = 4`
- **No hay solución:** `A = 6, B = 7, C = 3 = D`
- **Segunda solución:** `A = 7, B = 8, C = 5, D = 2`
- **Tercera solución:** `A = 8, B = 9, C = 7, D = 1`

2. `send([S, E, N, D, M, O, R, Y]):- dominio([S, E, N, D, M, O, R, Y]), restrict([S, E, N, D, M, O, R, Y]), generaval([S, E, N, D, M, O, R, Y]).`

La evaluación se realiza de la siguiente manera:
`S` $\neq 0$, `M` $\neq 0$ elimina dos valores y resulta la ecuación

(**Izqda**) `1000*S + 91*E + 10*R + D = 9000*M + 900*O + 90*N + Y` (**Drcha**)

Por lo tanto, **Izqda** < 18000 implica `M` $\neq 0$ y `M` ≤ 1 luego `M = 1`.

Además, **Izqda** ≥ 9000 junto con `91*E + 10*R + D` $\leq$ `819 + 90 + 9 = 918` implica que `S = 9` valor máximo.

Como `91*E + 10*R + D` $<$ `918` entonces `900*O + 90*N + Y` $<$ `918`, luego `O = 0`

Ya hemos generado `M = 1, S = 9, O = 0`, luego la ecuación

(izqda) `91*E + 10*R + D = 90*N + Y` (drcha)

toma sus variables de `[2..8]`, por tanto, `N` $>$ `E`, `91*2 + 10*3 + 4 = 216` $\leq$ izqda y `90*3 + 4 = 274` $\leq$ drcha implica `N` $\geq$ `3` y `E` $\geq$ `3`

La condición `N` $>$ `E` nos obliga a generar los siguientes valores:

- `N = 4, E = 3, R = 8` o bien `N = 4, E = 2, R = 8` (**Inconsistente**)
- `N = 5, E = 4, R = 8` entonces `Y + 6 = D` (**Inconsistente**)
- Si `N = 6, E = 5, R = 8`, entonces la (**solucion**) es `Y = 2, D = 7`

Observamos que cuando comienza la generación de los valores para las variables que aún quedan por asignar, el espacio de búsqueda se ha acotado, y en lugar de 10^8 posibilidades (valores en el intervalo `[0..9]`, para 8 variables) que teníamos al comienzo del cómputo, sólo nos quedan $7^3 * 4^2$ (para las variables `N, E, R` hay 3 posibles valores en el intervalo `[2..8]` y para `Y, D` sólo quedan al final los 2 restantes). Está claro que, fijado un valor su propagación sirve para detectar inconsistencias y pasar al valor siguiente.

1.3 Estructuras y programas con restricciones

Sean π un programa de Horn escrito con un vocabulario `DELTA` y $\mathcal{X} = (\texttt{VOC}, \mathcal{A})$ con `VOC` un vocabulario fijo y $\mathcal{A}$ una `VOC`-estructura, tal que `VOC` $\cap$ `DELTA` $= \emptyset$.

$$\mathcal{A} = (\mathbf{D}_{\mathcal{A}}, \{f^{\mathcal{A}}\}, \{r^{\mathcal{A}}\}) \quad \text{con} \quad =_{\mathcal{A}} \equiv \{(\mathrm{d}, \mathrm{d}) \text{ — } \mathrm{d} \in \mathrm{D}_{\mathcal{A}}\} \text{ incluida.}$$

Para poder manejar un lenguaje lógico con restricciones necesitamos conocer:

1. Predicados especiales, **restricciones**, que tienen como argumentos términos sobre el dominio $\mathrm{D}_{\mathcal{A}}$, acompañados de una teoría para manejarlos, y que se clasifican en:
   - Restricciones atómicas: `r(t1,...,tn)` $\in$ `Atomos(VOC)`
   - Sistemas de restricciones: `s = r1,...,rm`
2. Los predicados usuales, cuyos argumentos son términos de Herbrand y términos del dominio $\mathrm{D}_{\mathcal{A}}$.
3. Un programa de Horn sobre $\mathcal{X} = (\texttt{VOC}, \mathcal{A})$ está formado por:
   - Reglas con restricciones: `p` $\leftarrow$ `s` $\square$ `q1,...,qk`
   - `s = r1,...,rm`, con `rj` $\in$ `Atomos(VOC)`, `m` ≥ 0.
   - `p, qi` $\in$ `Atomos(DELTA)`, `k` ≥ 0, `p` y `qi` pueden contener términos de `VOC`.
4. Objetivos con restricciones: $\leftarrow$ `s` $\square$ `p1,...,pl`
   - `s = r1,...,rm`, con `rj` $\in$ `Atomos(VOC)`, `m` ≥ 0.
   - `pi` $\in$ `Atomos(DELTA)`, `l` ≥ 0, `pi` pueden tener términos de `VOC`.

Resolutores de Restricciones

Para poder manejar un almacen de restricciones que ha de ir acompañado de una teoría para manipular los argumentos de estas restricciones que son términos sobre el dominio $\mathrm{D}_{\mathcal{A}}$ necesitamos:

- Una definición explícita de sistema de restricciones en forma resuelta, que tiene que ser satisfactible en la teoría.

- Un algoritmo resolutor de restricciones que verifique
  - O bien pone a `s` en forma resuelta (`s` $\Longrightarrow$ `t`)
  - O bien detecta (`s` $\Longrightarrow$ `irresoluble`)

Características deseables para un resolutor:

- Debe ser incremental
  - El cómputo generado al añadir una nueva restriccón debe ser lo menor posible.
  - Le eliminación de una restricción, que puede producir una "vuelta atrás al punto de bifurcación", debe suponer el mímino esfuerzo.
- Debe ser eficiente.
- Debe ser correcto, es decir, si el resolutor de restricciones encuentra una inconsistencia en el almacen de restricciones, entonces el conjunto de restricciones almacenadas debe ser insatisfactible.
- Debería ser completo, es decir, si el conjunto de restricciones almacenas es insatisfactible entonces el resolutor debe detectarlo.

Ventajas de Programación Lógica con Restricciones

- Mayor expresividad en el tratamiento de problemas combinatorios sin tener que recurrir a impurezas operacionales, como (!).
- Al depender la SLD-resolución del tipo de objetos que maneja, nos permite un diseño mas uniforme y mayor efectividad.
- Gran aumento de la eficiencia, debido a la utilización de algoritmos especificos en la resolución de restricciones (simplex).

Inconvenientes de Programación Lógica con Restricciones

- Encontrar técnicas especificas para el tratamiento de los objetos sobre los que trabajemos.
- Estudiar las heurísticas conocidas y usar la más adecuada para la solución eficiente del problema que debemos resolver.
- Los resolutores completos son normalmente ineficientes, y siendo realista, en la mayor parte de las aplicaciones es más importante la eficiencia que la completitud.
- En muchos casos se hace necesario añadir restricciones redundantes que ayuden a los resolutores incompletos a mejorar su deteccion de inconsistencias.

1.4 Semántica declarativa y operacional de CLP(X)

Programación lógica con programas de Horn sobre el lenguaje de restricciones $\mathcal{X}$. Consideramos el esquema **CLP** fijo y el lenguaje $\mathcal{X}$ como un parámetro intercambiable.

- **CLP** es un procedimiento uniforme para diseñar extensiones potentes de la programación lógica.
- **CLP** garantiza que las propiedades semánticas fundamentales de la programación lógica sean heredadas por cada instancia **CLP**($\mathcal{X}$).
  1. Existencia de modelo de Herbrand mínimo, caracterizable como mínimo punto fijo definible.
  2. Correccion y Completitud de la resolucion SLD (acoplada al resolutor).
  3. Caracterización declarativa del fallo finito (bajo ciertas hipotesis sobre $\mathcal{X}$).

Definición: Dado un programa π:

1. **Base de Herbrand** para π es

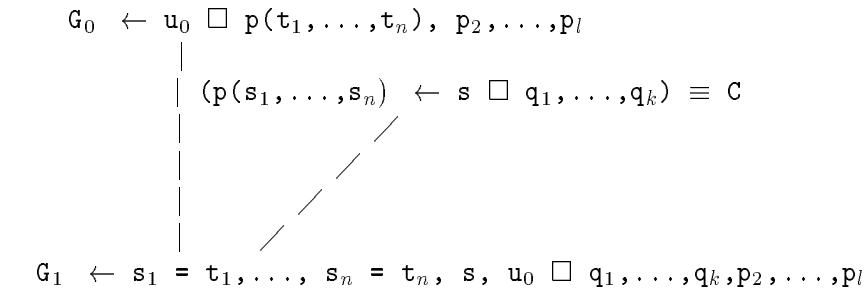
$$\mathtt{B}_\pi = \{\mathtt{p(d1,\ldots,dn)} \mid \mathtt{p/n} \in \mathtt{DELTA},\ \mathtt{di} \in \mathtt{D}_\mathcal{A} \cup \mathtt{H}_{\mathtt{DELTA}}\}$$

2. **Interpretación** de Herbrand para π es todo subconjunto $\mathtt{I} \subseteq \mathtt{B}_\pi$.
3. **Modelo** de Herbrand I de una cláusula de π: $\mathtt{I} \models (\mathtt{p} \leftarrow \mathtt{s} \square \mathtt{q1},\ldots,\mathtt{qk}) \iff$ $\mathtt{p.t.}\alpha \in \mathtt{Val}_\mathcal{A}\mathtt{:(Var} \to \mathtt{D}_\mathcal{A}\mathtt{):((q1}^\mathcal{A}(\alpha),\ \ldots,\mathtt{qk}^\mathcal{A}(\alpha) \in \mathtt{I}$ y $\mathcal{A} \models \mathtt{s}(\alpha)) \Rightarrow \mathtt{p}^\mathcal{A}(\alpha) \in \mathtt{I)}$
4. **Modelo** de Herbrand I de un programa π: $\mathtt{I} \models \pi \iff \mathtt{I} \models (\mathtt{p} \leftarrow \mathtt{s} \square \mathtt{q1},\ldots,\mathtt{qk})$, para toda $(\mathtt{p} \leftarrow \mathtt{s} \square \mathtt{q1},\ldots,\mathtt{qk}) \in \pi$.
5. **Operador de consecuencia inmediata** para π, $\mathtt{T}_\pi$, se define para todo $\mathtt{I} \subseteq \mathtt{B}_\pi$:
$\mathtt{T}_\pi\mathtt{(I)} = \{\mathtt{p}^\mathcal{A}(\alpha) \mid \mathtt{ex.(p} \leftarrow \mathtt{s} \square \mathtt{q1},\ldots,\mathtt{qk}) \in \pi,\ \mathtt{ex.}\alpha \in \mathtt{Val}_\mathcal{A}$
$\mathtt{con\ q1}^\mathcal{A}(\alpha),\ldots,\mathtt{qk}^\mathcal{A}(\alpha) \in \mathtt{I}$ y $\mathcal{A} \models \mathtt{s}(\alpha)\}$

Teorema: (Existencia y caracterización del modelo de Herbrand mínimo)

- $\mathtt{T}_\pi$ es un operador monotono y continuo.
- Su menor punto fijo es $\mathtt{M}_\pi = \bigcup\ (\mathtt{T}_\pi^n)(\emptyset)$
- $\mathtt{M}_\pi$ es el menor modelo de Herbrand de π.

SLD($\mathcal{X}$) indica SLD-resolución con restricciones sobre $\mathcal{X}$. (Supuesto $\mathtt{G}_0$ y C sin variables comunes).



El nuevo objetivo obtenido en un paso es: $\mathtt{G}_1 \leftarrow \mathtt{u}_1 \square \mathtt{q}_1, \ldots, \mathtt{q}_k, \mathtt{p}_2, \ldots, \mathtt{p}_l$, si denotamos a la nueva restricción obtenida como

$$\mathtt{u}_1 ::= \mathtt{s}_1 = \mathtt{t}_1, \ldots, \mathtt{s}_n = \mathtt{t}_n, \mathtt{s}, \mathtt{u}_0,$$

Denotamos como $\mathtt{G}_0 \Longrightarrow_{SLD} \mathtt{G}_1$, $(\mathtt{G}_0 \Longrightarrow^*_{SLD} \mathtt{G})$ uno (varios) pasos de SLD($\mathcal{X}$)-resolución.

Definición: Dado un programa π definimos:

1. P.t. $\mathtt{I} \subseteq \mathtt{B}_\pi$ y $\alpha \in \mathtt{Val}_\mathcal{A}$: α es una **solucion de** $\mathtt{G} \leftarrow \mathtt{s} \square \mathtt{p1}, \ldots, \mathtt{pl}$ **en** $\mathtt{I} \Longleftrightarrow \mathcal{A} \models \mathtt{s}(\alpha)$ y $\mathtt{p1}^\mathcal{A}(\alpha), \ldots, \mathtt{pl}^\mathcal{A}(\alpha) \in \mathtt{I}$.

2. $\mathrm{SOL}^I(G) = \{\alpha \in \mathtt{Val}_\mathcal{A} \text{ — } \alpha \text{ es una solucion de } \mathtt{G} \text{ en } \mathtt{I}\}$ es el conjunto de las soluciones de $\mathtt{G}$ fijado $\mathtt{I}$.

Teorema de corrección de SLD($\mathcal{X}$)
Si $\mathtt{G}_0 \Longrightarrow_{SLD} \mathtt{G}_1$ **entonces** $\mathrm{SOL}^I(G_1) \subseteq \mathrm{SOL}^I(G_0)$, para todo $\mathtt{I} \models \pi$.

Corolario: Si $\mathtt{I} \models \pi$, y $\mathtt{G}_0 \Longrightarrow^*_{SLD} \mathtt{G}$ **entonces** $\mathrm{SOL}^I(G) \subseteq \mathrm{SOL}^I(G_0)$.

Teorema de completitud de SLD($\mathcal{X}$)
Si $\alpha \in \mathrm{SOL}^{M_\pi}(\mathtt{G}_0)$ **entonces** existe $\mathtt{s}$ tal que $\mathtt{G}_0 \Longrightarrow^*_{SLD} (\leftarrow \mathtt{s} \square)$. y $\alpha \in \mathrm{SOL}^{M_\pi}(\mathtt{s})$
Además, podemos exigir que $\mathtt{s}$ esté en forma resuelta.
Idea de la demostración: Como sabemos que $\mathtt{M}_\pi = \bigcup \mathtt{M}^I_\pi$, siendo $\mathtt{M}^I_\pi = \bigcup \mathtt{T}^I_\pi(\emptyset)$. La demostración se hace por inducción sobre el nivel $\mathtt{I}$ tal que: $\alpha \in \mathrm{SOL}^{M^I_\pi}(\mathtt{G}_0)$.

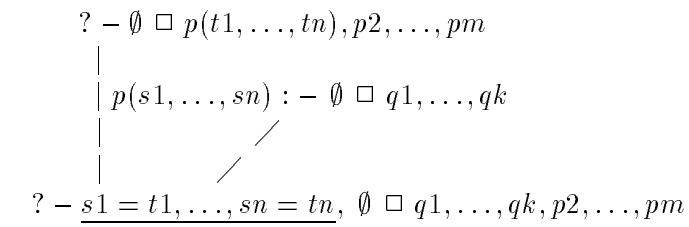
1.5 Algunas instancias interesantes del esquema CLP(X)

1.5.1 PROLOG como Programación Lógica con Restricciones

1. En PROLOG, suponemos que la unificación actúa de igualdad semántica, es decir, VOC $\supseteq \{=\}$ y tomamos como dominio de la estructura a Tf conjunto de los árboles finitos.

2. Todo término PROLOG, pasa a ser un árbol finito.

3. La resolución SLD actúa entre objetivos y reglas de la siguiente manera:



Si $\underline{\mathrm{s1} = \mathrm{t1}, \ldots, \mathrm{sn} = \mathrm{tn}}$ se puede resolver a forma normal, es decir, unifica, entonces continuamos y si no unifica entonces probamos con otra regla del programa.

PROLOG se generaliza ampliando por diferentes vías:

1. El dominio de la estructura, por ejemplo:
   - El dominio T de los árboles finitos e infinitos.
   - Dominios numéricos: $\mathcal{Z}, \mathcal{Q}, \mathcal{R}$.
   - Dominios finitos: $Bool = \{0,1\}$, Intervalos.
2. Restricciones no ecuacionales, por ejemplo:
   - s $\neq$ t Desigualdades
   - s $\leq$ t, s $\geq$ t, s $<$ t, s $>$ t Inecuaciones
   - r(t1,...,tn) Restricciones propias de cada dominio concreto.
3. Restricciones dentro de las reglas y objetivos:

$$
\begin{array}{c}
? - r \ \square \ p(t1, \ldots, tn), p2, \ldots, pm \\
\Big| \\
\Big| \ p(s1, \ldots, sn) : - \ s \ \square \ q1, \ldots, qk \\
\Big| \qquad\qquad \diagup \\
\Big| \qquad\quad \diagup \\
? - \underline{s1 = t1, \ldots, sn = tn, s, r} \ \square \ q1, \ldots, qk, p2, \ldots, pm
\end{array}
$$

Si s1 = t1,..., sn = tn, s, r se puede resolver a forma normal entonces continuamos y si se detecta inconsistencia entonces probamos con otra regla del programa.

1.5.2 PROLOG-II: Primer intento de PLR

1. `VOC` $\supseteq \{=, \neq\}$ y $D_{\mathcal{A}}$ = `Arboles infinitos`.
2. Los sistemas de restricciones son de la forma: `s = E` $\cup$ `I` siendo

$$\texttt{E} = \{\texttt{t1} = \texttt{s1}, \ldots, \texttt{tn} = \texttt{sn}\} \qquad \texttt{I} = \{\texttt{u1} \neq \texttt{v1}, \ldots, \texttt{um} \neq \texttt{vm}\}$$

3. Un conjunto {`Y1 = Y2, Y2 = Y3,...,Yk = Y1`} con las variables `Yj` diferentes entre sí se llama circuito de variables.
4. El sistema `s = E` $\cup$ `J` está en forma resuelta sii
   - `E` ={`X1 = t1,...,Xn = tn`} con `Xi` todas diferentes y no contiene circuitos de variables.
   - `J` ={..., <`Y1,...,Ym`> $\neq$ <`s1,...,sm`>, ...} representa cada "desigualdad" `Y1` $\neq$ `s1` $\vee \ldots \vee$ `Ym` $\neq$ `sm` y `m` > 1.
   - Las variables `Yj` y `Xi` son distintas.
   - La teoría obliga a comprobar que está en forma resuelta el sistema de ecuaciones `E` $\cup$ {`Y1 = s1,..., Ym = sm`} y permite hacer dicha comprobación por separado.

Primero estudiamos el **resolutor de ecuaciones**

$$\texttt{s1} = \texttt{E1} \cup \texttt{I} \Longrightarrow \begin{cases} \texttt{E2} \cup \texttt{I} & \text{con E2 en forma resuelta} \\ \texttt{Irresoluble} & \text{en otro caso} \end{cases}$$

Reglas de Transformación

1. (Absorción)
   $\texttt{s} \cup \{\texttt{X = X}\} \Longrightarrow \texttt{s} \qquad \texttt{s} \cup \{\texttt{c = c}\} \Longrightarrow \texttt{s}$

2. (Eliminación de variables) Si $\texttt{X} \neq \texttt{Y}$ y X tiene otras apariciones en s entonces:
   $\texttt{s} \cup \{\texttt{X = Y}\} \Longrightarrow \texttt{s[X} \leftarrow \texttt{Y]}$

3. (Confrontación) Si t1, t2 no son variables y |t1|<|t2|, entonces:
   $\texttt{s} \cup \{\texttt{X = t1, X = t2}\} \Longrightarrow \texttt{s} \cup \{\texttt{X = t1, t1 = t2}\}$

4. (Descomposición) $\texttt{s} \cup \{\texttt{f(t1,...,tn) = f(s1,...,sn)}\} \Longrightarrow$
   $\texttt{s} \cup \{\texttt{t1 = s1,...,tn = sn}\}$

5. (Fallo) $\texttt{s} \cup \{\texttt{f(t1,...,tn) = g(s1,...,sm)}\} \Longrightarrow$ Fallo, si $\texttt{f} \neq \texttt{g}$; $\texttt{n} \neq \texttt{m}$

Ejemplo: Calcula la forma resuelta de {X = Y, X = s(s(X)), Y = s(s(s(Y)))}

- (Elimiminación de variables){X = Y, X = s(s(X)), Y = s(s(s(Y)))} $\Longrightarrow$
- (Confrontación){X = Y, Y = s(s(Y)), Y = s(s(s(Y)))} $\Longrightarrow$
- (Descomposición 2-veces){X = Y, Y = s(s(Y)), s(s(Y)) = s(s(s(Y)))} $\Longrightarrow$
- (Confrontación){X = Y, Y = s(s(Y)), Y = s(Y) } $\Longrightarrow$
- (Descomposición){X = Y, s(Y) = s(s(Y)), Y = s(Y) } $\Longrightarrow$
- {X = Y, Y = s(Y)} **en forma resuelta y generando un árbol infinito**

Ahora, juntamos la parte de las desigualdades para obtener un **resolutor general**

$$\texttt{s} = \texttt{E} \cup \texttt{I} \Longrightarrow \begin{cases} \texttt{E} \cup \texttt{J} & \text{con E} \cup \text{J en forma resuelta} \\ \texttt{Irresoluble} & \text{en otro caso} \end{cases}$$

FASE I: Ecuaciones

- Si $\texttt{E} \Longrightarrow$ Irresoluble entonces $\texttt{s} \Longrightarrow$ Irresoluble
- Si $\texttt{s} \Longrightarrow$ (E1 $\cup$ I) con E1 en forma resuelta, entonces aplicar la fase II a (E1 $\cup$ I).

FASE II: Inecuaciones J := { }; **PARA CADA** $(\texttt{s} \neq \texttt{t}) \in \texttt{I}$ **HACER**

- **SI** $\texttt{E} \cup \{\texttt{s = t}\} \Longrightarrow$ Irresoluble **ENTONCES CONTINUAR**
  (SOL(E) $\subseteq$ SOL(s $\neq$ t) y $\texttt{s} \neq \texttt{t}$ puede eliminarse)
- **SI** $\texttt{E} \cup \{\texttt{s = t}\} \Longrightarrow \texttt{E'} \cup \{\texttt{Y1 = s1,...,Ym = sm}\}$ **ENTONCES**

- **SI** $m = 0$ **ENTONCES** $s \Longrightarrow$ Irresoluble
- **E. O. C.** J := J $\cup$ {<Y1,...,Ym> $\neq$ <s1,...,sm>}, E := E'

Si la iteración no se interrumpe y termina, entonces s = (E $\cup$ J) está en forma resuelta.

Observación: El algoritmo sólo es correcto si t representa a un árbol infinito. Si D = {a,b} es finito entonces la restricción s = {X $\neq$ a, X $\neq$ b} no tiene solución.

Ejemplo: Calcular la forma resuelta de s = {f(U, V) = f(s(X), W), g(V, W) = g(W, v)} $\cup$ {U $\neq$ 0, h(X, Y, Z) $\neq$ h(Y, Z, X)}.

- Fase I: s $\Longrightarrow$ {U = s(X), V = W} $\cup$ {U $\neq$ 0, h(X, Y, Z) $\neq$ h(Y, Z, X)}
- Fase II:
  - J = { }; {U = s(X), V = W, U = 0} $\Longrightarrow$ Eliminamos la desigualdad U $\neq$ 0 por ser insatisfactible con la ecuación U = s(X)
  - {U = s(X), V = W, h(X, Y, Z) = h(Y, Z, X)} $\Longrightarrow$ {U = s(X), V = W, X = Z, Y = Z} hace que J = {<X, Y> $\neq$ <Z, Z>}

Conclusión: s $\Longrightarrow$ {U = s(X), V = W, <X, Y> $\neq$ <Z, Z> }

1.5.3 El lenguaje CHIP

Arboles, aritmética lineal sobre racionales y Booleanos. Para las restricciones Booleanas tomamos B = ({0, 1}, $\odot$, $\sharp$), Anillo Booleano con $\odot$ la conjunción y $\sharp$ el O-exclusívo, y su teoría verifica los axiomas:

(X $\odot$ Y) $\odot$ Z	=	X$\odot$ (Y$\odot$ Z)	ASOCIATIVA	(X $\sharp$ Y) $\sharp$ Z	=	X $\sharp$ (Y $\sharp$ Z)
X $\odot$ Y	=	Y $\odot$ X	CONMUTATIVA	X $\sharp$ Y	=	Y $\sharp$ X
1 $\odot$ X	=	X	EL. NEUTRO	X $\sharp$ 0	=	X
X $\odot$ X	=	X	IDEMPOTENTE	X $\sharp$ X	=	0
X $\odot$ 0	=	0	X $\odot$ (Y $\sharp$ Z)	= (X $\odot$ Y)	$\sharp$	(X $\odot$ Z)

Se pueden demostrar los siguientes resultados con ayuda de la teoría anterior, que si tenemos la ecuación a $\odot$ X $\sharp$ b = 0 con a,b $\notin$ VAR entonces

- a $\odot$ X $\sharp$ b = 0 es consistente sii $\tilde{a} \odot b = 0$
- Si a $\odot$ X $\sharp$ b = 0 es consistente entonces el u. m. g. de X es X := b $\sharp$ $\tilde{a}$ $\odot$ Y
- Por eso, el algoritmo de unificación para restricciones Booleanas
  PROCEDURE unify (VAR X, Y: term); solve(X $\sharp$ Y).
  comprueba que $\tilde{a} \odot b = 0$, con $\tilde{a} =$ a $\sharp$ 1

```
FUNCTION solve (VAR T: term):  boolean;
VAR T1,T2:  term; X, U: variable;
BEGIN
IF T = 0  THEN solve := TRUE;
          ELSE IF T = X ⊙ T1 ♯ T2 AND SOLVE((T1 ♯ 1) ⊙ T2)
                THEN BEGIN  X:= (T1 ♯ 1) ⊙ U ♯ T2; solve := TRUE END
                ELSE solve := FALSE
                ENDIF
END.
```

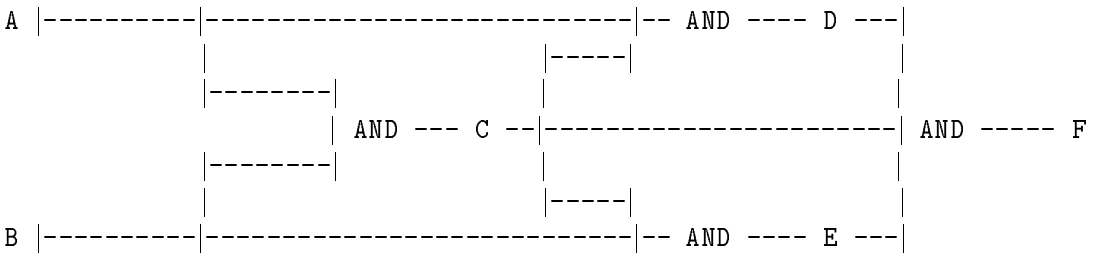
Ejemplo 1: Encontrar los valores de `A, B` que verifican `B ♯ A = 0`

1. Utilizamos `solve(B ♯ A)`
2. Descomponemos `(B ♯ A = X ⊙ T1 ♯ T2 = A ⊙ 1 ♯ B)`
3. Hay **Indeterminismo** pues también podemos descomponer `(B ♯ A = B ⊙ 1 ♯ A)`
4. En este caso `solve((1 ♯ 1) ⊙ B) = solve(0 ⊙ B) = solve(0) = TRUE`
5. Devolviendo `A := (1 ♯ 1) ⊙ U ♯ B = 0 ⊙ U ♯ B = 0 ♯ B = B`

Ejemplo 2: Encontrar los valores de `A, B, C` que verifican `A ⊙ B ⊙ C = 0`

1. Usamos `A ⊙ 0 = 0` y sólo ejecutamos `solve(B ⊙ C)`
2. Descomponemos `B ⊙ C = X ⊙ T1 ♯ T2 = C ⊙ B ♯ 0`
3. Hay **Indeterminismo** pues también podemos descomponer `B ⊙ C = B ⊙ C ♯ 0`
4. `solve((B ♯ 1)⊙ 0) = solve(0) = TRUE`
5. Devolviendo `C := (B ♯ 1) ⊙ U ♯ 0 = (B ♯ 1) ⊙ U`

Ejemplo 3: Dado el circuito lógico.



El predicado `Circuito` recoge las ecuaciones con `A, B` como entrada y `F` como salida

```
Circuito(A, B, F) :- C = 1 ♯ A⊙B, D = 1 ♯ A⊙C, E = 1 ♯ B⊙C, F = 1 ♯ D⊙E.
```

Sustituimos todas las ecuaciones para lograr que el predicado sólo dependa de A, B y F:
C = 1 ♯ A⊙B;
D = 1 ♯ A⊙C = 1 ♯ A⊙(1 ♯ A⊙B) = 1 ♯ A ♯ A⊙A⊙B = 1♯A♯A⊙B = 1 ♯ A⊙(1 ♯ B);
E = 1 ♯ B⊙C = 1 ♯ B⊙(1 ♯ A⊙B) = 1 ♯ B ♯ B⊙A⊙B = 1♯B♯A⊙B = 1 ♯ B⊙(1 ♯ A);
F = 1 ♯ D⊙E = 1 ♯ (1 ♯ A⊙(1♯B))⊙(1 ♯ B⊙(1♯A)) = 1 ♯ (1♯A♯A⊙B)⊙(1♯B♯B⊙A)
= 1 ♯ 1 ♯ A ♯ A⊙B ♯ B ♯ B⊙A ♯ B⊙A⊙B ♯ B⊙A ♯ B⊙A⊙A ♯ B⊙A⊙A⊙B = A ♯ B.

```
Circuito(A, B, F):- C = 1 ♯ A⊙B, D = 1 ♯ A⊙(1♯B), E = 1 ♯ B⊙(1♯A), F = A♯B.
```

Ejemplos con cuatro modos de uso diferentes. Obsérvese que el orden de los predicados en el cuepo de la cláusula no es determinante y se comienza evaluando la información recibida en modo in.

1. (out,out,in) ?-Circuito(A, B, 1).

- F = 1 = (A ♯ B) $\Longrightarrow$ A⊙B = 1⊙A⊙B = (A ♯ B)⊙A⊙B = 0
- A = A ♯ 0 = A ♯ (B ♯ B) = (A ♯ B) ♯ B = 1 ♯ B
- C = 1 ♯ A⊙B = 1 ♯ 0 = 1
- D = 1 ♯ A⊙(1 ♯ B) = 1 ♯ A ♯ A⊙B = 1 ♯ A♯0 = 1 ♯ A = 1♯1 ♯ B = 0 ♯ B = B
- E = 1 ♯ B⊙(1 ♯ A) = 1 ♯ B ♯ B⊙A = 1 ♯ B ♯ 0 = 1 ♯ B.

Respuesta A = 1 ♯ B, pero comprobando la consistencia del álmacen de restricciones.

2. (out,out,in) ?-Circuito(A, B, 0).

- F = 0 = (A ♯ B) $\Longrightarrow$ A = A ♯ 0 = A ♯ A ♯ B = B
- C = 1 ♯ A⊙B = 1 ♯ B⊙B = 1 ♯ B
- D = 1 ♯ A⊙(1 ♯ B) = 1 ♯ B⊙(1 ♯ B) = 1 ♯ B ♯ B⊙B = 1 ♯ B ♯ B = 1 ♯ 0 = 1
- E = 1 ♯ B⊙(1 ♯ A) = 1 ♯ B⊙(1 ♯ B) = 1 ♯ B ♯ B⊙B = 1 ♯ B ♯ B = = 1♯0 = 1.

Respuesta A = B, pero comprobando la consistencia del álmacen de restricciones.

3. (in,out,out) ?-Circuito(1, B, F).

- A = 1 $\Longrightarrow$ A⊙B = 1⊙B = B
- C = 1 ♯ A⊙B = 1 ♯ 1⊙B =1 ♯ B
- D = 1 ♯ A⊙(1 ♯ B) = 1 ♯ 1⊙(1 ♯ B) = 1 ♯ 1 ♯ 1⊙B = 0 ♯ B = B
- E = 1 ♯ B⊙(1 ♯ A) = 1 ♯ B⊙(1 ♯ 1) = 1 ♯ B⊙0 = 1 ♯ 0 =1
- F = A ♯ B = 1 ♯ B = B

Respuesta F = 1 ♯ B, pero comprobando la consistencia del álmacen de restricciones.

4. (in,out,out) ?-Circuito(0, B, F).

   - A = 0 $\Longrightarrow$ A$\odot$B = 0$\odot$B = 0
   - C = 1 $\sharp$ A$\odot$B = 1 $\sharp$ 0$\odot$B = 1 $\sharp$ 0 = 1
   - D = 1 $\sharp$ A$\odot$(1 $\sharp$ B) = 1 $\sharp$ 0$\odot$(1 $\sharp$ B) = 1 $\sharp$ 0 = 1
   - E = 1 $\sharp$ B$\odot$(1 $\sharp$ A) = 1 $\sharp$ B$\odot$(1 $\sharp$ 0) = 1 $\sharp$ B$\odot$1 = 1 $\sharp$ B
   - F = A $\sharp$ B = 0 $\sharp$ B = B

Respuesta F = B, pero comprobando la consistencia del álmacen de restricciones.

1.5.4 PROLOG-III

Arboles, aritmética lineal sobre racionales y Booleanos. Las restricciones lineales sobre números racionales verifican:

1. Las inecuaciones se manipulan usando el método del Simplex.
2. Transforma inecuaciones en ecuaciones con variables holgura (no negativas).
3. Sólo si todas las variables que aparecen en una desigualdad están instanciadas, ésta se comprueba.
4. Convierte inecuaciones estrictas en ecuaciones más desigualdades.
5. Maneja variables de dos tipos:
   - Las variables del problema, también llamadas básicas
   - Las que genera el proceso, también llamadas no básicas.
6. El tipo de una variable puede ser modificado a lo largo del cómputo.
7. Todas las restricciones trabajan sobre racionales, pero pueden usarse para naturales.

Problemas que han de resolverse para manejar adecuadamente estas restricciones.

- Detección de inconsistencias entre restricciones.
- Detección de redundancia de restricciones.
- Utilización de redundancia para mejorar la eficiencia de resolutores incompletos.
- Proyección de respuestas o elimación de variables no básicas.

Ecuaciones: Se ejecutan así:
R1: 2X + 3Y = 6 despejamos X := 3 - 3/2Y
R2: 3X + 2Y = 6 sustituyendo obtenemos Y:= 6/5; X:= 6/5
R3: X + Y = M sustituyendo obtenemos X:=6/5; Y:=6/5; M:=12/5.

Inecuaciones: Se transforman usando variables holgura `Si`.
`R1:` $2X + 3Y \leq 6$ se transforma en $2X + 3Y + S1 = 6$
`R2:` $3X + 2Y \leq 6$ se transforma en $3X + 2Y + S2 = 6$
`R3:` $X + Y \leq 3$ se transforma en $X + Y + S3 = 3$
`R4:` $X + Y \geq M$ se transforma en $X + Y - S4 = M$

De `R1` obtenemos `X := 3 - 3/2Y - 1/2S1`
Sustituyendo en `R2`: `Y:= 6/5 - 3/5S1 + 2/5S2;  X:= 6/5 + 2/5S1 - 3/5S2`
Sustituyendo en `R3`: `Y:= 6/5 - 3/5S1 + 2/5S2;  X:= 6/5 + 2/5S1 - 3/5S2;`
`S3:= 3/5 + 1/5S1 + 1/5S2.`
`R3` **es redundante con respecto a** `R1` **y** `R2`, pues `S1, S2` ≥ 0 implica `S3` ≥ 0.
Sustituyendo en `R4`: `Y:= 6/5 - 3/5S1 + 2/5S2; X:= 6/5 + 2/5S1 - 3/5S2;`
`S4:= (12/5 - M) - 1/5S1 - 1/5S2.`
`S4` ≥ 0 implica `M` $\leq$ `12/5`, pues en otro caso `R4` sería insatisfactible con respecto a `R1` y `R2`.

Cambio del tipo de las variables:
Insatisfactibilidad detectada usando variables no negativas.
`R1:` $X + Y \leq 2$ se transforma en $X + Y + S1 = 2$
`R2:` $X - Y + Z \leq 5$ se transforma en $X - Y + Z + S2 = 5$
`R3:` $X \geq 0$ se transforma en `Comprueba(X)`
`R4:` $-X + Y \geq 6$ se transforma en $-X + Y - S3 = 6$

De `R1` obtenemos `X := 2 - Y - S1;`
Sustituyendo en `R2` obtenemos `X := 2 - Y - S1;  Z := 3 + 2Y + S1 - S2`
`R3` transforma el tipo de la variable `X` en no básica, escribiendose por ello `Z` y `Y` en función de `X`: `Y := 2 - X - S1;  Z := 7 - 2X - S1 - S2`
Sustituyendo en `R4` obtenemos `-2X - S1 - S3 = 4`, es decir, `S1 = -4 - 2X - S3`
$X \geq 0$ y $S3 \geq 0$ implica $S1 < 0$, hecho incompatible con el tipo de la variable `S1`.

Desigualdades:
La satisfactibilidad de las desigualdades sólo puede decidirse si todas sus variables están instanciadas, mientras tanto siempre pueden ser satisfechas sobre los racionales.
`R1:` $2X + Y + 3Z + 2W = 10$ despejamos `Y := 10 - 2X - 3Z - 2W`
`R2:` $5X + 5Y + 8Z + W \neq K$ sustituyendo `Y` obtenemos $5X + 7Z + 9W \neq 50 - K$
`R3:` $X + 3Y + 2Z + W = 20$ sustituyendo `Y` obtenemos `X := 2 - 7/5 Z - W;`
`Y := 6 - 1/5 Z;` $4W \neq 40 - K$
`R4:` $3X + 4Y + 5Z + 5W = 36$ sustituyendo `X` e `Y` obtenemos `X := -1 - 7/5 Z;`
`Y := 6 - 1/5 Z; W := 3.` `R2` se transforma en $12 \neq 40 - K$, es decir, $K \neq 28$.

Inecuaciones estrictas:
Una inecuación estricta se transforma en una ecuación más una desigualdad.
`R1:` $3X + 2Y \leq 18$ se transforma en $3X + 2Y + S1 = 18$
`R2:` $-2X + Y \leq 2$ se transforma en $-2X + Y + S2 = 2$
`R3:` $X + Y < K$ se transforma en $X + Y - S3 = K$ $X + Y \neq K$

De `R1` obtenemos `X := 6 - 2/3 Y - 1/3 S1`
Sustituyendo en `R2` tenemos `X := 2 - 1/7S1 + 2/7S2; Y := 6 - 2/7S1 - 3/7S2`
Sustituyendo en `R3` tenemos `X := 2 - 1/7S1 + 2/7S2;  Y := 6 - 2/7S1 - 3/7S2;`

S2:= (57 - 7K) - 3S1 - 7S3

La clase de variables de las Si hace que 57 - 7K $\geq$ 0, es decir, K $\leq$ 8
Si K = 8 entonces S1 = S2 = S3 = 0, lo que implica que X + Y = 8 $\neq$ K = 8 que es inconsistente.

Para resolver las restricciones se sustituyen X, Y, S3 en la expresión X + Y $\neq$ K obteniendo K + S3 $\neq$ K, que representa el hecho de que S3 debe ser no nula.
Así pues, la solución a mostrar es X := 2 - 1/7S1 + 2/7S2; Y := 6 - 2/7S1 - 3/7S2; S2 := (57 - 7K) - 3S1 - 7S3; K $<$ 8; S3 $>$ 0

Este ejemplo pone de manifiesto que en la solución pueden aparecer variables que no han sido declaradas por el usuario, y que de no ser mostradas significarían inconsistencia en las soluciones, pues su valor es fundamental para la existencia de solución.

Si mostramos sólo X := 2 - 1/7S1 + 2/7S2; Y := 6 - 2/7S1 - 3/7S2; K $<$ 8 la solución sería incorrecta, pero mostrando la solución matemáticamente correcta aparecen variables desconocidas para el usuario que son parte esencial de la solución por tomar valores en un dominio restringido, pues son variables holgura (no negativas).

Números Naturales
Las restricciones sobre números naturales convierten las ecuaciones lineales en ecuaciones diofánticas y su solución consiste en encontrar la base diofántica. Esto significa que la forma normal de las infinitas soluciones es una base diofántica que genera cualquier solución como combinación lineal de los elemntos de la base.

Ecuaciones sobre números naturales :
R1: 2X + Y $\geq$ 3 se transforma en Y $\geq$ 3 - 2X
R2: X - 2Y $\neq$ 1 se transforma en 2Y $\neq$ X - 1

La base de soluciones es: (X, Y) = {(2, 0), (2,1)} + {(1, 0), (2, 1)}
$\bigcup$ {(2, 2), (1, 2), (0, 3)} + {(2, 1), (1, 1), (0, 1)}

1.5.5 CLP($\mathcal{R}$)

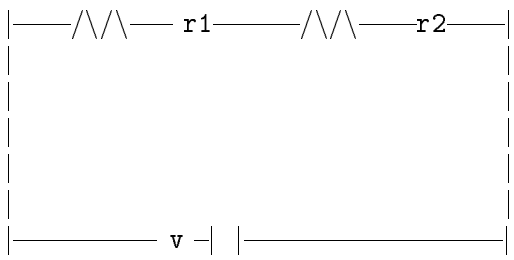
Manejo de números reales, teniendo en cuanta que

- VOC $\ni$ Constructoras, +, -, *, /,..., Constantes Reales.
- VOC $\ni$ {=, $<$, $<$, $>$, $>$} con su significado habitual.

La teoría para manejar las restricciones sobre números reales verifica:

1. Las restricciones lineales se manejan como restricciones sobre números racionales.
2. Las restricciones no lineales, como son indecidibles, sólo se comprueban si tienen todas sus variables instancias.
3. Aunque las restricciones trabajan sobre reales, pueden usarse sobre los naturales.

Ejemplo 1 : Dado el circuito eléctrico:



Principios físicos básicos:

- Ley de OHM: `ohm(V, I, R):- V = I * R` □.
- Principio de KIRCHOFF: `kirchoff(Xs):-` □ `suma(Xs, 0).`
  siendo: `suma([ ],0).`
  `suma([X | Xs], N):- N = X + M` □ `suma(Xs, M).`

Datos disponibles:

- Resistencias: `hayres(10).` `hayres(14).` `hayres(27).` `hayres(60).`
- Baterias: `haybat(10).` `haybat(20).`

Objetivo a resolver:
?- `14.5<V2, V2<16.25` □ `hayres(R1), hayres(R2), haybat(V), ohm(V1,I1,R1), ohm(V2,I2,R2), kirchoff([I1, -I2]), kirchoff([-V,V1,V2]).`

Se eligen los primeros valores para `R1, R2` y `V`, con lo que el objetivo es de la forma:
`14.5 < V2 < 16.25, (V1 / r1 ) - (V2 / r2) = 0, V1 + V2 = v`
como la restricción es satisfactible hay tres respuestas para las tuplas `(r1, r2, v)`
`(10, 27, 20), (14, 60, 20), (27, 100, 20)`

Ejemplo 2 : Multiplicación y división de complejos.

`c-mul(c(R1,I1),c(R2,I2),c(R3,I3)):- R3 = R1*R2 - I1*I2,I3 = R1*I2 + R2*I1`□

Veamos como utilizar este objetivo con distintos modos de uso.

1. `?-` □ `c-mul(c(1,1), c(2,2), Z).` Cálculo directo de `Z`.
2. `?-` □ `c-mul(c(1,1), Y, c(0,4)).`
   Resuelve el sistema de ecuaciones lineales: `1*R2 - 1*I2 = 0  1*I2 + R2*1 = 4`
   Generando una respuesta única `R2 = 2, I2 = 2`
3. `?-` □ `c-mul(X, c(2, 2), c(0, 4)).`
   Resuelve el sistema de ecuaciones lineales: `R1*2 - I1*2 = 0  R1*2 + I1*2 = 4`
   Generando una respuesta única `R1 = 1, I1 = 1`
4. `?-` □ `c-mul(c(X, Y), c(X, Y), c(-3, 4)).`
   Genera el sistema de ecuaciones no lineales: `X*X - Y*Y = -3  2*X*Y = 4` que debe ser invernado hasta que se pueda solucionar.

5. `?- c-mul(c(X, Y), c(X, Y), c(-3, 4)), aux(Y, Z).` siendo
   `aux(Y, Z):- Y = 2*Z, Z = Y - 1` □ genera el sistema:
   `X*X - Y*Y = -3  2*X*Y = 4  Y = 2*Z  Z = Y - 1`
   De las dos últimas ecuaciones se obtiene `Y = 2`, generando el nuevo sistema
   `X*X = 1  X = 1` y comprobando que `1*1 = 1`.

Invernación de Restricciones.

- El resolutor de CLP($\mathcal{R}$) no puede decidir directamente la satisfactibilidad de algunas restricciones.
- Las restricciones lineales se resuelven directamente usando una variante del método del Simplex.
- Algunas restricciones no lineales se invernan hasta convertirse en lineales, por vinculación de sus variables, siendo este mecanismo no accesible al usuario.

Capítulo 2

Programación con restricciones sobre dominios finitos

Hay un concepto de restricción que no está basado en fijar un dominio y definir las restricciones sobre ese dominio como un caso particular de predicados, sino que está fundamentado en las técnicas de consistencia, donde la principal preocupación es la rapidez del cómputo con variables lógicas en función de la información inferida del álmacen de variables. Este tipo de restricciones se utilizaron en lenguajes gráficos desde el año 1981, y como variante de este tipo de evaluación apareció la llamada **Propagacion Local**.

Ejemplo 1: Transforma Centigrados en Fahrenheit o viceversa usando `F = 1.8 C + 32` y con ayuda del grafo: Si `C = 10` entonces `F = 50`. Si `F = -40` entonces `C = -40`.

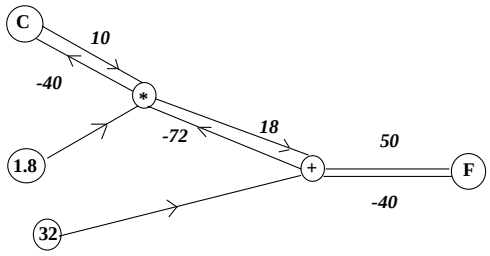


Figura 2.1: Ejemplo 1

Ejemplo 2: La propagación local no resuelve el problema si el grafo tiene ciclos. Así, el sistema `A + T = B; B + T = C` se recoge en el grafo con ciclos:

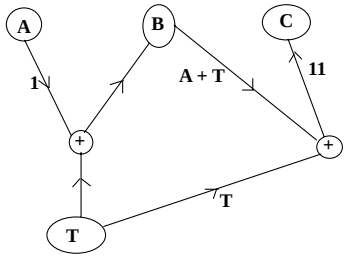


Figura 2.2: Ejemplo 2

Si `A = 1` y `C = 11` tenemos `1 + T = B` y `B + T = 11`, pero `A = 1`, `C = 11` y `1 +2*T = 11` produce la solución `T = 5`, que no es encontrada por este método.

2.1 Ejemplos de programas en dominios finitos

Ejemplo 3: Queremos planificar 6 tareas en una jornada laboral de 5 horas, cumpliendo:

- Todas las tareas son de 1 hora de duración.
- `T2` y `T3` no pueden realirse simultaneamente.
- Cada tarea situada a la derecha de otra debe realizarse después de ésta.

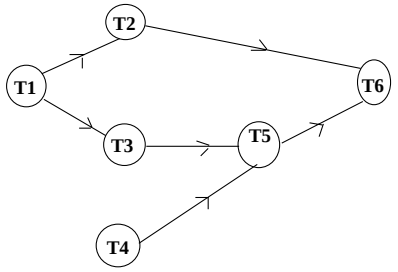


Figura 2.3: Diagrama de las tareas

Para resolver este problema empleamos la metodología usada ya antes:

1. Dominio de las variables: {`1, 2, 3, 4, 5`}.
2. Restricciones a verificar: `distintos(T2, T3), antes(T1, T2), antes(T1, T3), antes(T2, T6), antes(T3, T5), antes(T4, T5), antes(T5, T6)`.
   Las técnicas de consistencia propagan la información entre las variables con ayuda de las restricciones.
   - Sabemos que `T1, T2` $\in$ {`1, 2, 3, 4, 5`} si usamos `antes(T1, T2)` eliminamos del dominio de `T1` el valor `5` y del dominio de `T2` el valor `1`. Así pues, `T1` $\in$ {`1, 2, 3, 4`} y `T2` $\in$ {`2, 3, 4, 5`}.
   - La **Arco-consistencia** o propagación de la consistencia entre los arcos produce la siguiente reducción de dominios: `T1` $\in$ {`1, 2`}, `T2` $\in$ {`2, 3, 4`} `T3` $\in$ {`2, 3`} `T4` $\in$ {`1, 2, 3`} `T5` $\in$ {`3, 4`} `T6` $\in$ {`4, 5`}
3. Generación de valores concretos: Hemos de usar un mecanismo de enumeración de los valores de las variables sobre los dominios ya restringidos.
   - Si `T1 = 2` entonces `T2` no puede valer `2`, pues ambos valores han de ser distintos. Para detectar estas y otras inconsistencias suele etiquetarse cada variables con el primero de los valores permitidos y propagarse la elección. De este modo, si la elección es correcta, obtenemos la solución y en otro caso tenemos un método efectivo de enumeración de los valores de las variables que nos permite pasar al siguiente valor permitido.
   - Si damos a `T1` el valor `2`, entonces `T2` $\in$ {`3,4`} y `T3` $\in$ {`3`}.
   - Además `distintos(T2, T3)` restringe `T2` $\in$ {`4`} y propagando esta información obtenemos `T1` $\in$ {`2`} `T2` $\in$ {`4`} `T3` $\in$ {`3`} `T4` $\in$ {`1, 2, 3`} `T5` $\in$ {`4`} `T6` $\in$ {`5`}

Ejemplo 4: El problema de las ecuaciones diofánticas sobre los números naturales.

- R1: `2X + Y` $\geq$ `3` se tranforma en `Y` $\geq$ `3 - 2X`
- R2: `X - 2Y` $\neq$ `1` se tranforma en `X` $\neq$ `2Y + 1`

Basándonos en la propagación local, modelizamos las restricciones en términos de redes de manera que los nodos sean las variables y los arcos las restricciones entre ellas. Así:

```
X ::  [0 ..  maxint] ⟹ min(X) = 0
      |
      | Y ≥ 3 - 2X ⟺ Y ≥ 3
      |
Y ::  [0, 1, 2, 3, 4,..., maxint] ⟹ min(Y) = 3
      |
      | 2Y + 1 ≠ X ⟺ 7 ≠ 0
      |
X ::  [1 ..  maxint]
```

Primera Solución: `X = 0, Y ::  [3 ..  maxint]`

Si deseamos otra solución:

```
X ::  [1 ..  maxint] ⟹ min(X) = 1
      |
      | Y ≥ 3 - 2X ⟺ Y ≥ 1
      |
Y ::  [0, 1, 2, 3, 4,..., maxint] ⟹ min(Y) = 1
      |
      | 2Y + 1 ≠ X ⟺ 3 ≠ 1
      |
X ::  [2 ..  maxint]
```

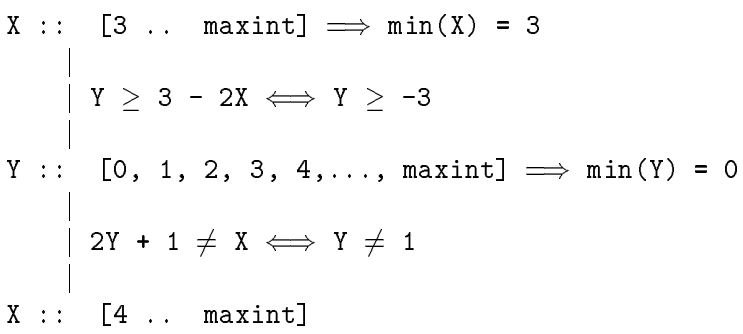
Segunda Solución: `X = 1, Y ::  [1 ..  maxint]`

Si queremos otra solución:

```
X ::  [2 ..  maxint] ⟹ min(X) = 2
      |
      | Y ≥ 3 - 2X ⟺ Y ≥ -1
      |
Y ::  [0, 1, 2, 3, 4,..., maxint] ⟹ min(Y) = 0
      |
      | 2Y + 1 ≠ X ⟺ 2Y ≠ 1
      |
X ::  [3 ..  maxint]
```

Tercera Solución: `X = 2, Y ::  [0 ..  maxint]`

Si queremos otra solución más:



Cuarta Solución: `X = 3, Y :: [0] ∪ [2 .. maxint]`

Así seguiriamos enumerando soluciones, mientras nos fuesen solicitadas. Queda claro en este ejemplo que los arcos tienen que manejar la poda de dominios de manera que, en ambas direcciones del arco, la consistencia esté garantizada.

Ejemplo 5: Disyunciones de restricciones

En un problema de planificación sustituimos la realización de una tarea `antes que` otra por:

1. La primera tarea comienza en el tiempo `T1` y tiene una duración de `D1`.
2. La segunda tarea comienza en el tiempo `T2` y tiene una duración de `D2`.
3. No pueden efectuarse simultáneamente por usar los mismos recursos.
4. El orden de ejecución influye en la solución obtenida si resolvemos el problema con predicados de la forma:
   - `disjuntas(T1, D1, T2, D2):- T1 + D1 < T2.`
   - `disjuntas(T1, D1, T2, D2):- T2 + D2 < T1.`
5. Luego, hemos de **deducir la máxima cantidad de información del almacén** de restricciones y **retrasarlos puntos de elección** lo más posible.

Un ejemplo clásico de metapredicado global que maneja disjunción de restricciones **retrasando los puntos de eleccióne**s es **Cardinalidad**, `card(L, U,[R1,R2,...,Rk])`: El predicado vale **Cierto** sii del conjunto de las `k` restricciones, al menos `L` y a lo sumo `U` se satisfacen. Operacionalmente, el predicado verifica:

- Si `L < 0` y `U > k` entonces vale **Cierto**.
- Si `L > k` entonces vale **Falso**.
- Si el resolutor comprueba que una restricción no se cumple, entonces la elimina de la lista.
- Si el resolutor comprueba que una restricción se cumple, entonces también la elimina de la lista, pero además decrementa en 1 el valor de `L` y `U`.

La solución del problema de planificación usando este metapredicado sería:
`card(1, 2, [T1 + D1 < T2, T2 + D2 < T1])`, es decir, `L = 1, U = 2, k = 2`:

- Si una restricción es cierta entonces `L = 0, U = 1 = k` y el predicado vale **Cierto**.
- Si ambas son ciertas, entonces `L = -1, U = 0 = k` y el predicado vale **Cierto**.
- Si ambas son falsas, entonces `L = 1, U = 2, k = 0` y el predicado vale **Falso**.

La disyunción constructiva se usa para deducir **la máxima información común a varias restricciones**. Por ejemplo, el predicado:

- `constructivo(X, Y) :- X > Y + 20.`
- `constructivo(X, Y) :- Y > X + 20.`

Para el objetivo `?- X::[5..35], Y::[10..40], constructivo(X, Y).` obtenemos:

1. Alternativa: `min(Y) = 10, X>30` e `max(X) = 35, Y<15`.
2. Alternativa: `min(X) = 5, X<20` e `max(Y) = 40, Y>25`.
3. El sistema de restricciones debe permitir la manipulación de intervalos siguiente:
   Si `(X<20 Or X>30) And X::[5..35]` **entonces** `X::[5..19]` $\cup$ `[31..35]`
   Si `(Y<15 Or Y>25) And Y::[10..40]` **entonces** `Y::[10..14]` $\cup$ `[26..40]`.

Ejemplo 6: El operador implicación `R` $\Rightarrow$ `R'` debe actuar de modo que en cuanto del almacén de restricciones se deduzca la restricción `R` entonces se realice `R'`.
El predicado `and(X, Y, Z)` se puede definir usando $\Rightarrow$ así:

```
and(X, Y, Z):- (X = 0 ⇒ Z = 0), (Y = 0 ⇒ Z = 0),
               (Z = 1 ⇒ (X = 1, Y = 1)),
               (X = 1 ⇒ Y = Z), (Y = 1 ⇒ X = Z).
```

En las tres primeras implicaciones hay propagación local y en las otras dos hay propagación simbólica. Para explicar la ejecución de algunos objetivos escribimos primero el objetivo y separado por un ; el almacén de restricciones.

1. {`and(X, Y, Z)`; $\mathbf{(X = 0)}$} produce {$\mathbf{(X = 0, Z = 0)}$}
2. {`and(X, Y, Z)`; $\emptyset$} no modifica el almacén ni evalua.
3. {`(X = 0 ⇒ Z = 0), (T = 0 ⇒ X = 0)`; $\mathbf{(T = 0)}$} inverna al primer objetivo y evalúa el segundo {`(X = 0 ⇒ Z = 0)`; $\mathbf{(X = 0, T = 0)}$} ahora evalúa el resto generando {$\mathbf{(X = 0, T = 0, Z = 0)}$}

Ejemplo 7: Necesitamos un sumador con entradas `X` e `Y`, resultado `S` y variables auxiliares `Cin, Cout`:

```
sumador(X,Y,Cin,S,Cout):- and(X,Y,C1), xor(X,Y,S1),
                          and(Cin,S1,C2), xor(Cin,S1,S), or(C1,C2,Cout).
```

El objetivo `?- sumador(X, Y, 1, S, 0)` se evalúa así:

- Los valores de entrada generan el almacén: $\{(\mathbf{Cin = 1, Cout = 0})\}$.
- $\{$`or(C1,C2,Cout)`$; (\mathbf{Cout = 0})\} \Longrightarrow \{(\mathbf{C1 = 0, C2 = 0})\}$
- $\{$`and(Cin, S1, C2)`$; (\mathbf{Cin = 1, C2 = 0})\} \Longrightarrow \{(\mathbf{S1 = 0})\}$
- $\{$`xor(X, Y, S1)`$; (\mathbf{S1 = 0})\} \Longrightarrow \{(\mathbf{X = Y})\}$
- $\{$`xor(Cin,S1,S)`$; (\mathbf{Cin = 1, S1 = 0})\} \Longrightarrow \{(\mathbf{S = 1})\}$
- $\{$`and(X, Y, C1)`$; (\mathbf{X = Y, C1 = 0})\} \Longrightarrow \{(\mathbf{X = 0, Y = 0})\}$
- El resultado del almacén es: `X = 0, Y = 0, S = 1`.

Ejemplo 8: Muchos problemas de restricciones usan una función de coste que se quiere minimizar. Tenemos 3 máquinas que pueden realizar 5 tareas cada una, los costes de esas tareas son distintos según la máquina que las realice. Para minimizar los costes enumeramos las posibilidades usando el predicado `elemento(N, Lista, Valor)` cuyo significado es `Valor` es el `N`-ésimo elemento de la `Lista`.

```
?-[M1,M2,M3]::[1..5], distintas(M1,M2,M3), elemento(M1,[3,2,6,8,9],C1),
                elemento(M2,[4,6,2,3,2],C2), elemento(M3,[6,3,2,5,2],C3),
                                        C1 + C2 + C3 = CTOTAL, CTOTAL ≤ 9.
```

El objetivo significa que tenemos 3 máquinas, que han de realizar las 5 tareas, que no pueden realizarlas simultaneamente y que el coste en tiempo de todas las tareas no puede superar las 9 horas. La solución del objetivo nos proporciona los siguientes valores:

- `M1` sólo puede realizar `T1` o `T2`, pues `T3, T4` y `T5` consumen demasiado tiempo.
- `M2` puede realizar cualquier tarea excepto `T2` que consume demasiado tiempo.
- `M3` puede realizar cualquier tarea excepto `T1` que consume demasiado tiempo.
- El coste total está entre `6 (p.e. [T2,T3,T5])` y `9 (p.e. [T2,T3,T4])`.

Propagación local frente a Resolutor para Dominios Finitos

1. Hay ocasiones en que el dominio de trabajo sólo posee dos valores (Booleanos) y entonces debe trabajarse con resolutores que tengan en cuanta la teoría del dominio.
2. El uso constante de restricciones de la forma `distintos(X, Y)`, o bién en su forma más compleja `diferentes(X,Y,...,Z)` hace que todos los resolutores deban implementar eficientemente este tipo de predicados, para lo cual han de tener un buen mecanismo de propagación de la información sobre las variables y su compartición, esto implica que el resolutor ha de invernar las variables hasta que la instanciación de algunas permite la poda del dominio de las demás.

2.2 Estrategias de búsqueda en problemas de satisfacción de restricciones

Llamamos **problema de satisfacción de restricciones** (CSP: Constraint Satisfaction Problem) a un triple formado por:

- Un conjunto de variables `V = {X1,...,Xn}`.
- Para cada variables de `V` un conjunto de posibles valores `Di`, que llamaremos **dominio** de `Xi`.
- Un conjunto de restricciones, normalmente binarias, `Cij(Xi,Xj)` que determinan los valores que las variables pueden tomar simultáneamente.
- Decimos que los valores `vi`, `vj` de las variables `Xi` y `Xj` son **consistentes** sii `Cij(vi,vj)` es cierta para $i \neq j$.

Una **solución de un problema de satisfacción de restricciones** es toda asignación de valores de cada dominio permitido a las variables, verificando las restricciones.

Las estratégias de búsqueda de soluciones en problemas de satisfacción de restricciones tratan de encontrar las tuplas de valores `(v1,...,vn)` de las variables `X1,...,Xn` que satisfacen cierta propiedad `Pn(v1,...,vn)` expresada mediante las restricciones básicas o primitivas `Cij`.

Una restricción entre varias variables determina el subconjunto del producto cartesiano de sus dominios formado por las combinaciones de sus valores consistentes entre sí.

Ejemplo 9: Si tenemos dos variables `X` e `Y` con `Dom(X) = {1, 2, 3}` y `Dom(Y) = {1, 2}` entonces cualquier subconjunto de `{(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)}` satisfará una posible restricción entre `X` e `Y`.

- La restricción `X = Y` representa al subconjunto `{(1,1), (2,2)}`.
- `distintos(X, Y)` al subconjunto `{(1,2), (2,1), (3,1), (3,2)}`.

Aunque la mayor parte de las restricciones verifican ecuaciones o inecuaciones lineales, no siempre tiene que ser así, y en ocasiones entenderlas simplemente como subconjuntos de productos cartesianos ayuda a la intuición. Además, la aridad de una restricción nos puede ayudar a comprender mejor el problema de búsqueda de soluciones:

- **Restricción unitaria**: Tiene una sola variable afectada.
  La restricción puede usarse para excluir un valor concreto del dominio de definición, por ejemplo, $X \neq 1$.
- **Restricción binaria**: Tiene dos variables afectadas.
  Una restricción binaria entre variables de dominios de tamaño `m` y `n`, puede ser representada mediante una matriz de tamaño `m x n` con valores:
  - `1` si la restricción se satisface para ese par de valores.
  - `0` si la restricción no se satisface para ese par de valores.

  Aunque esta no es una forma práctica de definir restricciones, es bueno recordar que una matriz aleatoria es una posible representación de un restricción binaria.

La mayor parte de los algoritmos para resolver problemas de satisfacción de restricciones buscan sistemáticamente la primera solución mediante la posible asignación de valores a variables y la encuentran, si ésta existe, pero tardan mucho en comprobar que el problema no tiene ninguna solución. Por eso, vamos a presentar algunos ejemplos de **algoritmos para la construcción parcial de la solución asignando valores a las variables en cada paso**.

Para formalizar lo que hacen estos algoritmos necesitamos tener en cuanta que todo procedimiento de **backtracking** define propiedades intermedias $P_i(v1, \ldots, v_i)$ que son expresadas mediante restricciones primitivas Cij y tales que:

para todo $0 \leq k < n$ se verifica que $P_{k+1}(v1, \ldots, v_k, v_{k+1}) \longrightarrow P_k(v1, \ldots, v_k)$

Algoritmo de genera y comprueba

- Se asigna un valor del dominio permitido a cada una de las variables de la tupla que se va a evaluar, y después se comprueba que forman una solución, es decir, si verifican todas las restricciones. Si no las verifica entonces se desecha esa tupla y genera la siguiente.
- El proceso se repite hasta encontrar una solución, o hasta que se hayan generado y comprobado todos los casos posibles.
- **Este algoritmo evalúa cada vez las restricciones entre todas las variables.**

Algoritmo de backtracking simple

- Se asigna un valor del dominio permitido a la siguiente variable a evaluar, y se comprueba si forma parte de la solución parcial, es decir, si este valor junto con la solución parcial ya construída verifica las restricciones. $P_k(v1, \ldots, v_k)$ es cierta para todo $k < n$ **sii**

  1. $v_i \in D_i$, para todo $1 \leq i \leq k$.
  2. Cij(vi,vj) es cierta para $1 \leq i < j \leq k$.

  Si no verifica las restricciones entonces se desecha ese valor y se toma el siguiente valor del dominio permitido para la variable.
- Si se intentan todos los valores de la variable y ninguno forma parte de la solución, entonces se pasa a eliminar el valor de la variable anterior evaluada y se toma el siguiente valor para esa variable.
- El proceso se repite hasta encontrar una solución, o hasta que se hayan probado por este método todos los casos posibles.
- **Este algoritmo sólo computa las restricciones entre la nueva variable a evaluar y las variables computadas previamente.**

Algoritmo de backtracking con chequeo previo

- Se asigna un valor de su dominio a la nueva variable a evaluar, y se comprueba no sólo si forma parte de la solución parcial, es decir, si este valor junto con los valores de la solución parcial ya construída verifica las restricciones, sino que además se mira si algún valor de una futura asignación tiene conflicto con este valor y si es así se elimina temporalmente del dominio de esa futura variable.
  P_k(v1, ..., v_k) es cierta para todo k $<$ n **sii**

  1. $v_i \in D_i$, para todo 1 $\leq$ i $\leq$ k.
  2. Cij(vi,vj) es cierta para 1 $\leq$ i $<$ j $\leq$ k.
  3. Para todo l con k $<$ l $\leq$ n, existe un $v_l \in D_l$ tal que C_{1l}(v1,vl), ..., Ckl(vk, ..., vl) son ciertas.

- La ventaja de este método es que si el dominio de una futura variable llega a ser vacío, esto significa que la solución parcial es inconsistente, con lo cual se prueba con otro valor de la variable activa o se vuelve atrás y el dominio de las futuras variables es restaurado.

- Con el backtracking simple no se habría detectado la inconsistencia hasta que no se hubieran evaluado todas las futuras variables.

- **En el algoritmo de backtracking con chequeo previo las ramas de los árboles que van a dar inconsistencia son podadas con anterioridad.**

Algoritmo de backtracking con chequeo previo generalizado

P_k(v1, ..., v_k) es cierta para todo k $<$ n **sii**

1. Las tres condiciones anteriores del backtracking con chequeo previo.
2. Para todo l con k $<$ l $\leq$ n, existe un $v_l \in D_l$ tal que es posible encontrar los valores $v_{k+1} \in D_{k+1}$, ..., $v_{l-1} \in D_{l-1}$, $v_{l+1} \in D_{l+1}$, ..., $v_n \in D_n$ verificando C_{k+1l}(v_{k+1},v_l), ...,C_{l-1l}(v_{l-1},v_l), C_{ll+1}(v_l,v_{l+1}), ..., Cln(vl, ..., vn) son ciertas.

Ejemplo 10: El problema de las cuatro reinas, es decir, como colocar en un tablero de tamaño 4x4, cuatro reinas sin que se ataquen. Tenemos que dar valores a una lista de cuatro elementos [R1,R2,R3,R4], con dom(Ri) = [1,2,3,4] y verificando que no hay más de una reina por fila, columna y dos diagonales.

1. Un algoritmo clásico de **genera y comprueba** para este problema podría ser: Generar cada permutación de los cuatro valores y luego mirar si se atacan o no los valores de las distintas filas.

```
%%%El problema de las cuatro reinas sin implementar ningun backtraking
:-use_module(library(clpfd)).

%%%Borrar un elemento de una lista
borrar(X,[X|Xs],Xs).
borrar(X,[Y|Ys],[Y|Rs]):- borrar(X,Ys,Rs).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Permutar una lista de elementos
permutacion([],[]).
permutacion([X|Xs],Ls):- borrar(X,Ls,Rs),permutacion(Xs,Rs).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Comprobar que una reina no ataca a ninguna de la lista dada, estando
%%%dicha reina en la fila del subtablero indicada por N
no_ataca(X,[],N).
no_ataca(X,[Y|Ys],N):- X #\= Y - N,X #\= Y + N,N1 is N+1,no_ataca(X,Ys,N1).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Regla que inicializa la anterior: comprueba si una reina ataca o no a una
%%%lista de reinas inicializando la N a 1.
noataca(X,Xs):- no_ataca(X,Xs,1).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Comprueba si una lista de posiciones para las reinas en el tablero es
%%%segura, es decir, comprueba que no se atacan unas a otras.
seguro([]).
seguro([X|Xs]):- noataca(X,Xs),seguro(Xs).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Programa principal: va generando distintas posibilidades de colocación de
%%%las reinas en el tablero, y comprobando si son una solucion del problema

cuatro_reinas([R1,R2,R3,R4],Type):- permutacion([R1,R2,R3,R4],[1,2,3,4]),
                    seguro([R1,R2,R3 R4]),labeling([Type],[R1,R2,R3,R4]),
                                statistics(runtime,[_,T]), print_time(T).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Presentación de los datos estadísticos en milisegundos
print_time(T):- print_time('%%% Timing',T).
print_time(What,T):- Secds is T/1000,Hour is T//3600000,S0 is T-Hour*3600000,
                     Min is S0//60000,S1 is R0-Min*60000,Sec is S1/1000,
    format( "~|~w~t~15+ ~|~`0t~d~2+:~|~`0t~d~2+:~|~`0t~3f~6+ ~t~3f~10+~n",
                                            [What,Hour,Min,Sec,Secds]).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- cuatro_reinas(L,ff).              || ?- cuatro_reinas(L,ffc).
%%% Timing      00:00:23.400  23.400 || %%% Timing      00:01:26.966  86.966
L = [2,4,1,3] ? ;                    || L = [2,4,1,3] ? ;
%%% Timing      00:00:03.284   3.284 || %%% Timing      00:00:03.634   3.634
L = [3,1,4,2] ? ;                    || L = [3,1,4,2] ? ;
no                                   || no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- cuatro_reinas(L,max).             || ?- cuatro_reinas(L,min).
%%% Timing      00:01:15.366  75.366 || %%% Timing      00:00:15.517  15.517
L = [2,4,1,3] ? ;                    || L = [2,4,1,3] ? ;
%%% Timing      00:00:04.000   4.000 || %%% Timing      00:00:02.683   2.683
L = [3,1,4,2] ? ;                    || L = [3,1,4,2] ? ;
no                                   || no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
?- cuatro_reinas(L,all).
%%% Timing      00:00:16.350  16.350
L = [2,4,1,3] ? ;
%%% Timing      00:00:03.500   3.500
L = [3,1,4,2] ? ;
no
```

2. Un algoritmo de backtracking standard **sin chequeo previo** para este problema podría ser: Cada vez que va a colocar una nueva reina, comprueba antes que no ataca a ninguna de las ya colocadas hasta el momento.

```
%%%Se encarga de ir asignando un posible valor a cada variable-reina de los
%%%que quedan disponibles, y comprueba que dicha asignacion es valida para
%%%la solucion, es decir, que la nueva reina no ataca a las ya colocadas
reinas([],Colocadas,[]).
reinas([X|Xs],Colocadas,Valores):-
borrar(X,Valores,NuevosValores), %% damos un posible valor a la reina
noataca(X,Colocadas), %%comprobamos que la asignacion no ataca a las otras
reinas(Xs,[X|Colocadas],NuevosValores). %%tenemos una nueva reina colocada
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%Programa principal: llama a la funcion reinas inicializando la lista
%%%auxiliar donde se iran almacenando las reinas ya colocadas, y los
%%%posibles valores para cada variable-reina
cuatro_reinas([R1,R2,R3,R4],Type):-reinas([R1,R2,R3,R4],[],[1,2,3,4]),
                                   labeling([Type],[R1,R2,R3,R4]),
                             statistics(runtime, [_,T]),print_time(T).
%%%%%%%%%%Los predicados que faltan son como en el programa anterior%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- cuatro_reinas(L,ff).             ||?- cuatro_reinas(L,ffc).
%%% Timing    00:01:46.450  106.450 ||%%% Timing     00:00:35.917   35.917
L = [2,4,1,3] ? ;                   ||L = [2,4,1,3] ? ;
%%% Timing    00:00:03.767    3.767 ||%%% Timing     00:00:02.666    2.666
L = [3,1,4,2] ? ;                   ||L = [3,1,4,2] ? ;
no                                  || no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
| ?- cuatro_reinas(L,max).          ||?- cuatro_reinas(L,min).
%%% Timing    00:00:35.334   35.334 ||%%% Timing     00:00:19.184   19.184
L = [2,4,1,3] ? ;                   ||L = [2,4,1,3] ? ;
%%% Timing    00:00:02.766    2.766 ||%%% Timing     00:00:06.700    6.700
L = [3,1,4,2] ? ;                   ||L = [3,1,4,2] ? ;
no                                  ||no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
| ?- cuatro_reinas(L,all).
%%% Timing    00:00:23.333  23.333
L = [2,4,1,3] ? ;
%%% Timing    00:00:03.283   3.283
L = [3,1,4,2] ? ;
no
```

Este programa comienza dando valor a la primera reina: `R1 = 1`.

- Da a la segunda reina el valor 1, `R2 = 1`, no satisface las restricciones.
- Da a la segunda reina el valor 2, `R2 = 2`, no satisface las restricciones.
- Da a la segunda reina el valor 3, `R2 = 3`, si satisface las restricciones.
  - Da a la tercera reina el valor 1, `R3 = 1`, no satisface las restricciones.
  - Da a la tercera reina el valor 2, `R3 = 2`, no satisface las restricciones.
  - Da a la tercera reina el valor 3, `R3 = 3`, no satisface las restricciones.
  - Da a la tercera reina el valor 4, `R3 = 4`, no satisface las restricciones.
- Da a la segunda reina el valor 4, `R2 = 4`, si satisface las restricciones.
  - Da a la tercera reina el valor 1, `R3 = 1`, no satisface las restricciones.
  - Da a la tercera reina el valor 2, `R3 = 2`, si satisface las restricciones.
    - Da a la cuarta reina el valor 1, `R4 = 1`, no satisface las restricciones.
    - Da a la cuarta reina el valor 2, `R4 = 2`, no satisface las restricciones.
    - Da a la cuarta reina el valor 3, `R4 = 3`, no satisface las restricciones.
    - Da a la cuarta reina el valor 4, `R4 = 4`, no satisface las restricciones.
  - Da a la tercera reina el valor 3, `R3 = 3`, no satisface las restricciones.
  - Da a la tercera reina el valor 4, `R3 = 4`, no satisface las restricciones.

El algoritmo **sin chequeo previo**, vuelve a dar valor a la primera reina: `R1 = 2`.

- Da a la segunda reina el valor 1, `R2 = 1`, no satisface las restricciones.
- Da a la segunda reina el valor 2, `R2 = 2`, no satisface las restricciones.
- Da a la segunda reina el valor 3, `R2 = 3`, no satisface las restricciones.
- Da a la segunda reina el valor 4, `R2 = 4`, si satisface las restricciones.
  - Da a la tercera reina el valor 1, `R3 = 1`, si satisface las restricciones.
    - Da a la cuarta reina el valor 1, `R4 = 1`, no satisface las restricciones.
    - Da a la cuarta reina el valor 2, `R4 = 2`, no satisface las restricciones.
    - Da a la cuarta reina el valor 3, `R4 = 3`, si satisface las restricciones.

La primera solución generada tras 18 fallos es `[2,4,1,3]`.

3. Un algoritmo de backtracking **con chequeo previo** para este problema podría ser: Tomamos un valor, lo eliminamos de la lista de valores permitidos y si no ataca a ninguno de los valores siguientes entonces lo damos por colocado.

```
%%%El problema de las cuatro reinas con forward checking: incrementa la
%%%poda al elegir la posicion de cada nueva reina, no solo comprobando
%%%si dicha posicion amenaza a las reinas ya colocadas, sino que reduce
%%%el dominio de las siguientes reinas dependiendo de su posicion.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
compatible(V1,V2,N):- V1 #\= V2 + N, V1 #\= V2 - N, V1 #\= V2.
%%%%%%%%%%%%%%%%%%
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).
%%%%%%%%%%%%%%%%%%
borra_valor(X,[],N,[]).
borra_valor(X,[V|Resto],N,[V|NResto]):- compatible(X,V,N),!,
                                        borra_valor(X,Resto,N,NResto).
borra_valor(X,[V|Resto],N,NResto):- borra_valor(X,Resto,N,NResto).
%%%%%%%%%%%%%%%%%%
forward(X1,Resto,NResto):- forward(X1,Resto,1,NResto).
forward(X,[],N,[]).
forward(X,[[Var,Dom]|Resto],N,[[Var,[F|T]]|NResto]):-
        borra_valor(X,Dom,N,[F|T]),N1 is N+1,forward(X,Resto,N1,NResto).
%%%%%%%%%%%%%%%%%%
reinas_aux([]).
reinas_aux([[X1,D]|Resto]):- member(X1,D), forward(X1,Resto,NResto),
                             reinas_aux(NResto).
%%%%%%%%%%%%%%%%%%

cuatro_reinas([R1,R2,R3,R4],Type):- L = [1,2,3,4],
                             reinas_aux([[R1,L],[R2,L],[R3,L],[R4,L]]),
                             labeling([Type],[R1,R2,R3,R4]),
                             statistics(runtime, [_,T]), print_time(T).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- cuatro_reinas(L,ff).            ||?- cuatro_reinas(L,ffc).
%%% Timing    00:00:31.300  31.300 ||%%% Timing    00:00:16.933   16.933
L = [2,4,1,3] ? ;                  ||L = [2,4,1,3] ? ;
%%% Timing    00:00:04.817   4.817 ||%%% Timing    00:00:02.867    2.867
L = [3,1,4,2] ? ;                  ||L = [3,1,4,2] ? ;
no                                 ||no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?- cuatro_reinas(L,max).           ||?- cuatro_reinas(L,min).
%%% Timing    00:00:13.550  13.550 ||%%% Timing    00:00:13.067  13.067
L = [2,4,1,3] ? ;                  ||L = [2,4,1,3] ? ;
%%% Timing    00:00:02.833   2.833 ||%%% Timing    00:00:03.467   3.467
L = [3,1,4,2] ? ;                  ||L = [3,1,4,2] ? ;
no                                 ||no
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 ?- cuatro_reinas(L,all).
%%% Timing    00:00:10.733  10.733
L = [2,4,1,3] ? ;
%%% Timing    00:00:02.533   2.533
L = [3,1,4,2] ? ;
no
```

El algoritmo **con chequeo previo**, da valor a la primera reina `R1 = 1`, lo cual implica que `dom(R2) = [3,4], dom(R3) = [2,4], dom(R4) = [2,3]`.

- Da a la segunda reina el valor 3, `R2 = 3`, luego `dom(R3) =` $\emptyset$`, dom(R4) = [2]`.

- Da a la segunda reina el valor 4, `R2 = 4`, luego `dom(R3) = [2], dom(R4) = [3]`.
  Da a la tercera reina el valor 1, `R3 = 2`, luego `dom(R4) =` $\emptyset$.

El algoritmo **con chequeo previo**, revalora a la primera reina `R1 = 2`, lo cual implica que `dom(R2) = [4], dom(R3) = [1,3], dom(R4) = [1,3,4]`.

- Da a la segunda reina el valor 4, `R2 = 4`, luego `dom(R3) = [1], dom(R4) = [1,3]`.

- Da a la tercera reina el valor 1, `R3 = 1`, luego `dom(R4) = [3]`.
  Da a la cuarta reina el valor 3, `R4 = 3`, luego la primera solución generada tras 2 fallos es `[2,4,1,3]`.

Si todas las restricciones son binarias, entonces solamente necesitan ser comprobadas las restricciones entre la variable activa y las variables por evaluar, pues con las variables anteriores no es necesario hacer verificaciones.

En el ejemplo de las reinas, las comprobaciones podan rápidamente los dominios, pero el trabajo de comprobación es más o menos el mismo con chequeo previo que sin él. Vamos a estudiar un ejemplo en el que el chequeo previo evita mucha evaluacuón.

Ejemplo 11: Sean `X1, X2,..., Xn` variables tales que `dom(X1, X2,..., Xn) = [1,2]` y queremos que verifiquen la restricción `diferentes(X1, X2,...,Xn)`.

Backtracking simple asigna a `X1 = 1`, después `X2 = 2` y continua intentando con `X3,...,Xn`, repitiendo la vuelta a valores anteriores, hasta conseguir cubrir todos las hojas del árbol, todo él de fallos.

Backtracking con chequeo previo asigna a `X1 = 1`, entonces `dom(X2,..., Xn) = [2]`, ahora asigna a `X2 = 2`, entonces `dom(X3,..., Xn) =` $\emptyset$ y no hace más comprobaciones.

Limitaciones de la programación de estratégias de búsqueda en programación lógica

- Incrementa el esfuerzo del programador.

- La formulación del problema se hace menos natural.

- La mejora del algoritmo sólo se nota para valores grandes. En el problema de las reinas a partir de 12 reinas.

- Los predicados primitivos del lenguaje están basados en la poda `a posteriori` del espacio de búsqueda.

2.3 Planteamiento general de un problemas de satisfacción de restricciones en dominios finitos

Un **problema de satisfacción de restricciones** es un problema compuesto de un número finito de **variables**, cada una de ellas asociada a un dominio **finito** y un conjunto de restricciones que permiten restringir los valores que pueden tomar las variables simultáneamente. Denotaremos como `(Z, D,C)` a un problema de satisfacción de restricciones, donde `Z` es el conjunto finito de variables, `D` es una función que hace corresponder a cada variable un conjunto finito de objetos y `C` es un conjunto de restricciones sobre un subconjunto de `Z`.

Una **etiqueta** es un par `(X,a)` formado por una variable `X` y un valor `a` que representa la asignación de un valor perteneciente al dominio de la variable a dicha variable. Una **etiqueta compuesta** es la asignación simultánea de valores a un conjunto de variables, que puede ser vacío. Una **restricción** $\mathbf{C}_S$ sobre un conjunto de variables `S` es un conjunto de etiquetas compuestas para dichas variables. Una etiqueta compuesta satisface una restricción si pertenece a ella.

Una **tupla solución** es una etiqueta compuesta correspondiente a todas las variables del problema y que está en todas las restricciones. Un problema de satisfacción de restricciones es **satisfactible** si existe una tupla solución para él.

Decimos que dos problemas de satisfacción de restricciones son **equivalentes** si tienen identicos conjuntos de variables e identicos conjuntos de tuplas de soluciones. Dados dos problemas de satisfacción de restricciones `P = (Z, D,C)` y `P' = (Z', D',C')` decimos que `P` se **reduce** a `P'` sii

- `P` y `P'` son equivalentes.
- Todos los dominios de variable de `D'` están en un subconjunto de su dominio en `D`.
- `C'` es igual o más restrictiva que `C`.

Para resolver un problemas de satisfacción de restricciones han de realizarse básicamente tres tareas:

1. Reducción del problema hasta conseguir uno equivalente y mínimo.(Consistencia).
2. Búsqueda o enumeración de las tuplas solución.
3. Optimizaciones de lo anterior combinando ambas técnicas.

2.3.1 Reducción de problemas: Consistencia

Si representamos un problemas de satisfacción de restricciones mediante un grafo, en el que los **nodos** representen a las variables y las **arcos** a las restricciones binarias, y consideramos a las restricciones unitarias como un caso particular de las binarias, podemos obtener tres tipos de consistencia en un grafo:

Consistencia sobre un **Nodo** (1-Consistencia):
Los valores del dominio de cada variable satisfacen las restricciones unitarias sobre ella.

Ejemplo 12: Restricción unitaria $V \neq 3$, siendo Dom(V) = [1, 2, 3, 4, 5].

V :: [1, 2, 3, 4, 5]
|
| V ≠ 3
|
V :: [1, 2, ~~3~~, 4, 5]

Consistencia sobre una **Arco** (2-Consistencia):
Un arco (X, Y) de un problema de satisfacción de restricciones es consistente sii para todo valor a del dominio de X que satisfaga las restricciones de X, existe un valor del dominio de Y que es compatible con la etiqueta (X, a).

Ejemplo 13: Restricción $X < Y - 2$, siendo Dom(X, Y) = [1, 2, 3, 4, 5].

X	X + 2	Y		X	X + 2	Y
1	3	1		1	3	~~1~~
2	4	2	X + 2 < Y obliga a podar	2	4	~~2~~
3	5	3		~~3~~	~~5~~	~~3~~
4	6	4		~~4~~	~~6~~	4
5	7	5		~~5~~	~~7~~	5

Los dominios de X, Y se podan quedando únicamnte los valores consistentes con el arco $X < Y - 2$ que son: Dom(X) = [1,2] y Dom(Y) = [4,5].

Ejemplo 14: Tenemos las actividades A, B, C, D junto con su duración y precedencia. Esto nos permite generar de manera natural restricciones asociadas a las tareas :

Tarea	Duración	Precedencia	Restricción
A	3	B	inicio(A) + Duración(A) ≤ inicio(B)
A	3	C	inicio(A) + Duración(A) ≤ inicio(C)
B	2	D	inicio(B) + Duración(B) ≤ inicio(D)
C	4	D	inicio(C) + Duración(C) ≤ inicio(D)
D	2		inicio(D) + Duración(D) ≤ finobra

Usamos variables que representan el principio y final del proyecto y el comienzo de cada actividad. Para calcular el extremo superior del dominio inicial sumamos todas las duraciones $3 + 2 + 4 + 2 = 11$ y expresamos el problema con restricciones binarias:

Variable	Dominio inicial	Restricción
inicio	[0]	inicio = 0
iA	[0..11]	iA + 3 ≤ iB
iA	[0..11]	iA + 3 ≤ iC
iB	[0..11]	iB + 2 ≤ iD
iC	[0..11]	iC + 4 ≤ iD
iD	[0..11]	iD + 2 ≤ final
final	[0..11]	

Primero, se realiza el cálculo del máximo intervalo para `final` podando los extremos inferiores de los dominios de las otras variables:

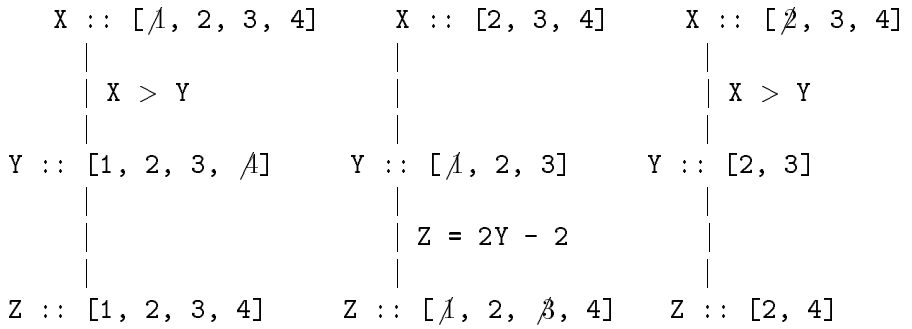
`iA+3≤iB::[3..11], iA+3≤iC::[3..11], iB+2≤iD::[5..11], iC+4≤iD::[7..11]`

Las dos últimas afirmaciones implican `iD::[7..11]`, luego `iD+2≤final::[9..11]`.

Obtenido el intervalo `final::[9..11]` se realiza el cálculo de los otros intervalos podando los extremos superiores: `iD+2≤final` implica `iD::[7..9]`. Pero debe cumplirse que `iB+2≤iD::[7..9]`, `iC+4≤iD::[7..9]`, lo cual nos acota los intervalos `iB::[3..7]`, `iC::[3..5]`. Por último, `iA+3≤iB::[3..7]`, `iA+3≤iC::[3..5]` implica `iA::[0..2]`.

Con esto hemos llevado a cabo la reducción del problema usando la Arco-consistencia, pero si buscamos alguna solución concreta entonces hemos de obtener una tupla solución. Así, por ejemplo, si deseamos minimizar el tiempo de la obra, entonces tomamos el mínimo valor de `final = 7` y los minimos valores de las otras variables `iA = 0, iB = 3, iC = 3, iD = 7` y comprobamos que las inecuaciones se verifican.

Ejemplo 15: La conjunción de las restricciones binarias `X > Y, Z = 2Y - 2`, siendo `Dom(X, Y, Z) = [1,2,3,4]`.



`X > Y` elimina el extremo inferior de `X::[2,3,4]` y superior de `Y::[1,2,3]`. Por tanto, `(2Y - 2)::[0,2,4]` y `Z = 2Y - 2` implica `[1,2,3,4] ∩ [0,2,4] = [2,4]`. Esto elimina los valores `1` y `3]` del dominio de `Z` y el valor `1` del dominio de `Y`. De nuevo `X > Y` elimina el extremo inferior de `X`.

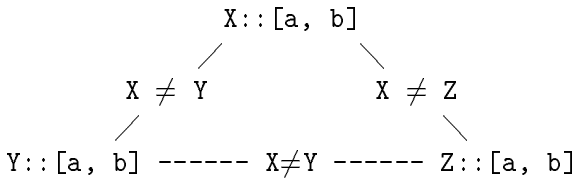
En los ejemplos anteriores con la nodo-consistencia y la arco-consistencia hemos tenido bastante para eliminar valores de los dominios que no forman parte de la solución, pero este método, en general, es incompleto. Por ello, para la detección de la satisfactibilidad de restricciones es necesario introducir el concepto de consistencia sobre un **Camino**:

Un camino `(X, Y, Z)` es un camino consistente sii para cada par de valores `a` ∈ `dom(X)`, `c` ∈ `dom(Z)`, exista un `b` ∈ `dom(Y)` tal que `(a,b)` pertenece a la restricción que une `X` con `Y` y `(b,c)` pertenece a la restricción que une `Y` con `Z`. Si tal `b` no existe entonces se quitan simultáneamente `a` con `c`.

Ejemplo 16: Si `Dom(X) = [1, 2]`, `Dom(Y) = [4, 5]` y `Dom(Z, W) = [6..10]`, evaluar la conjunción `X < Y - 2, Y < Z, 5*X ≤ Z, 5*X ≤ W, W + Z < 20`.

Con esos dominios iniciales las arco-consistencias se verifican, sin embargo, para que el camino (X, W, Z) sea consistente, tomemos el valor X = 2, entonces $X < Y - 2$ hace Y = 5. Además, $5*X \leq Z$, $5*X \leq W$ implican que Z = 10 y W = 10, y por supuesto se verifica que $Y < Z$. Falta sólo comprobar $W + Z < 20$, pero $20 < 20$ no se cumple, así pues 2 se elimina del dominio de X. Este es el motivo por el cual la reducción sólo se cuida de la eliminación de soluciones inconsistentes, pero siempre que se realiza una enumeración de valores de las variables para formar un tupla solución hay que volver a verificar las restricciones, pues sólo se han eliminado las inconsistencias seguras, pero hay otras inconsistencias, asociadas sobre todo a los valores de los extremos de los intervalos, que pueden invalidar a algunos de los valores del dominio.

Ejemplo 17: Para Dom(X, Y, Z) = [a, b], evaluar $X \neq Y$, $Y \neq Z$, $X \neq Z$.



IMPOSIBLE

Un problemas de satisfacción de restricciones es **arco-consistente** sii todo arco de su grafo es consistente.

Un problemas de satisfacción de restricciones es **camino-consistente** sii todo camino de su grafo es consistente.

Un problemas de satisfacción de restricciones P = (Z, D,C) es **k-satisfactible** sii para todo subconjunto de k variables de Z, existe un conjunto de etiquetas para ellas que satisface las restricciones del problema.

Un problemas de satisfacción de restricciones sobre N variables es **satisfactible** sii es k-satisfactible para todo k entre 1 y N.

Satisfactibilidad implica consistencia.

2.3.2 Algoritmos generales para alcanzar la nodo y arco consistencia

El algoritmo para alcanzar la nodo-consistencia (NC) sería:

```
procedure NC(Z, D, C)
begin
for cada X de Z do
   for cada a de D_X do
      if not satisface((X, a), C_X) then D_X := D_X - {a};
return(Z, D, C);
end.
```

El algoritmo ingenuo para alcanzar la arco-consistencia (AC) sería:

```
procedure AC-1(Z, D, C)
begin
NC(Z, D, C)
Q:= {(X, Y) | C_{X,Y} ∈ C}  /* C_{X,Y} coincide con C_{Y,X} */
repeat
cambio:= false;
   for cada (X, Y) ∈ Q do
      cambio:= revisar-dom((X,Y), (Z, D, C)); /* D_X puede reducirse */
until not cambio;
return(Z, D, C);
end.

procedure revisar-dom((X,Y), (Z, D, C))
begin
borrado := false;
for cada a de D_X do
   if not existe b ∈ D_Y que satisfaga ((X, a), (Y, b), C_{X,Y}) then
      begin
      D_X := D_X - {a}; borrado:= true
      end;
return borrado;
end.
```

El algoritmo mejorado para alcanzar la arco-consistencia (AC) sería:

```
procedure AC-3(Z, D, C)
begin
NC(Z, D, C)
Q:= {(X, Y) | C_{X,Y} ∈ C}
while Q ≠ { } do
begin
for cada (X, Y) ∈ Q
   Q:= Q - {(X, Y)};
   if revisar-dom((X,Y),(Z,D,C)) then Q:= Q ∪ {(Z,X)|C_{Z,X}∈C ∧ Z≠X ∧ Z≠Y};
/* D_X puede reducirse */
end;
return(Z, D, C);
end.
```

2.3.3 Búsqueda o enumeración de las tuplas solución

Una vez reducidos los dominios de las variables usando los algoritmos de consistencia, todavía nos queda generar adecuadamente las tuplas solución. Para ello se utilizan los algoritmos de mirar hacia adelante (Lookahead), que siguen básicamente el siguiente esquema:

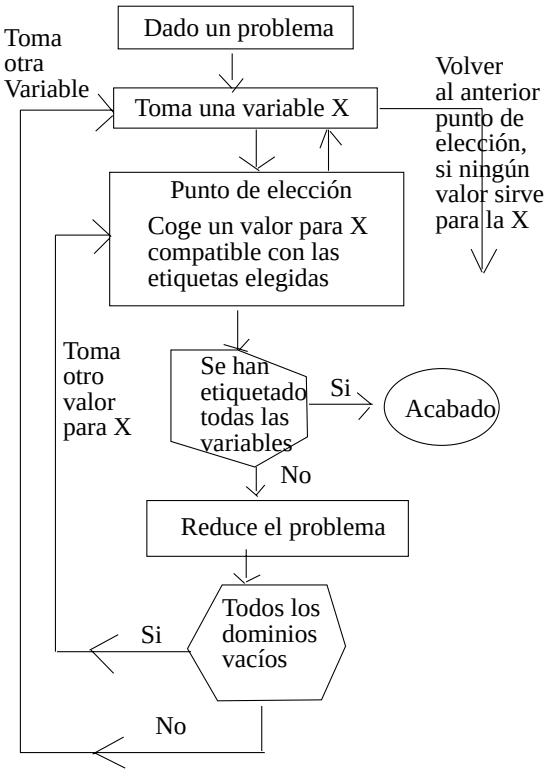


Figura 2.4: Lookahead

Algoritmo concreto para el chequeo previo (Forward Checking):

```
procedure Forward Checking-1(Z, D, C)
begin  FC(Z, { }, D, C) end.

procedure FC(Sinetiq, Eti-comp, D, C)
begin
if  Sinetiq = { } then  return Eti-comp
else  begin  tomar una variable X de Sinetiq
   repeat tomar un valor a de D_X; borrar a de D_X;
   if (Eti-comp + (X, a)) no viola ninguna restricción
   then
      begin  D' := actualiza(Sinetiq - {X}, D, C, (X, a))
      if no hay dominios vacíos en D' then  begin
        resultado := FC(Sinetiq - {X}, Eti-comp + (X, a), D', C);
        if resultado ≠ nil  then return resultado; end
     end
   until (D_X = { });
   return nil;
   end /* else*/
end.
```

```
procedure actualiza(W, D, C, Etiqueta)
begin
D' := D;
for cada variable Y de W do
  for cada valor a de D'_Y do
    if (Y, a) es compatible con Etiqueta respecto a C then D'_Y := D'_Y - {a};
return D';
end.
```

Algoritmo concreto para mirar totalmente hacia adelante (Full Look ahaed):

```
procedure AC-Lookahead-1(Z, D, C)
begin  AC-L(Z, { }, D, C) end.
procedure AC-L(Sinetiq, Eti-comp, D, C)
begin
if  Sinetiq = { } then return Eti-comp
else begin tomar una variable X de Sinetiq
   repeat tomar un valor a de D_X; borrar a de D_X;
   if (Eti-comp + (X, a)) no viola ninguna restricción
   then
      begin D' := actualiza(Sinetiq - {X}, D, C, (X, a))
      D'' := AC-L(Sinetiq - {X},Eti-comp + (X,a), D', C)
      if no hay dominios vacíos en D'' then begin
        resultado := AC-L(Sinetiq - {X}, Eti-comp + (X, a), D', C);
        if resultado ≠ nil then return resultado; end
     end
   until (D_X = { });
   return nil;
   end /* else*/
end.
```

2.4 Ordenes de búsqueda en problemas de satisfacción de restricciones

El esquema básico tiene la siguiente forma:

1. Ordenación de variables
   - Ordenación por mímima anchura.
   - Ordenación por mímima anchura de banda.
2. Heurísticas de búsqueda.
   - Menor dominio.
   - Dominio más restringido.
3. Ordenación de valores.
4. Ordenación de restricciones.

2.4.1 Ordenación de variables

El algoritmo de búsqueda en un árbol que satisface ciertas restricciones, requiere que las variables que van a ser consideradas estén ordenadas, puesto que hemos de tener claro en todo momento como realizar la vuelta atrás. El orden puede ser:

- **Estático**: Ha de ser especificado al comienzo.
- **Dinámico**: La próxima variable a considerar depende del estado de la búsqueda. Esto no siempre es posible decidirlo.
  - **En el backtracking simple**: No tengo bastante información durante la búsqueda para hacer una elección diferente de la ordenación inicial.
  - **En el backtracking con chequeo previo**: El estado activo lleva información de los dominios de las variables y ésto puede usarse para elegir la próxima variable a evaluar.

2.4.2 Heurísticas de búsqueda

Una heurística de ordenación de variables se puede basar en el principio de "fail-first", en el sentido de que primero probemos valores que tengan un dominio pequeño, pues lo más seguro es que alguno de esos valores, que son pocos, deba formar parte de la solución. En el backtracking con chequeo previo, este principio se implementa de manera natural. Si cualquier valor tiene la misma probabilidad para formar parte de la solución, probemos primero con los menores dominios, pues si la solución parcial conduce a una rama muerta cuanto antes lo descubramos mejor. Por otra parte, si la solución parcial puede ser expandida a una solución completa, entonces cada variable ha de estar instanciada.

- Si tomo primero la variable con mayor dominio entonces, teóricamente, tendré que hacer más pruebas. Si tomo una variable con menor dominio y el problema tiene solución entonces un valor de ese dominio será parte de la solución parcial y lo encontraré antes que en un dominio mayor.
- Cuando la solución parcial permita una solución completa, nos interesa generar la solución directamente sin fomentar backtracking, y si no conduce a la solución nos interesa saberlo lo antes posible, por eso elegimos el dominio menor.
- Con esta heurística podrá reducirse la media de la profundidad de las ramas de los árboles provocando fallos tempranamente.
- Por lo tanto, aún no habiendo solución, o si deseamos generar todas las soluciones, el tamaño del árbol que hemos de explorar es menor que si usamos la ordenación estática.
- Se puede mejorar esta heuríatica si, para las variables con dominios de igual tamaño, elegimos primero las que aparecen en más restricciones, pues fuerzan más la solución del problema.
- Sin embargo, cuando los problemas reales tengan una componente que fuerce a dar una elección de variables con otros criterios, por ejemplo problemas de planificación en los que deben reducirse prioritariamente algunos de los costes, entonces se tendrá que manejar otra heurística.

2.4.3 Ordenación de los valores

Ya hemos estudiado que el algoritmo de búsqueda requiere que las variables estén ordenadas, pero nos queda el problema de cómo seleccionar los valores a asignar a dichas variables. Si no hay conflictos lo más sencillo es considerar el orden "natural" de los dominios y asignar desde el menor valor hasta el mayor.

Las distintas ordenaciones de los valores a asignar a las variables cambian las ramas de árbol. Esto podría ser bueno si sólo queremos una solución, pero si queremos todas las soluciones o debemos comprobar que no hay solución, entonces el orden de las ramas nos es indiferente.

2.4.4 Ordenación de restricciones

Normalmente, al programar con este tipo de lenguajes se suelen escribir primero las restricciones más importantes, sin embargo de lo visto anteriormente, podemos deducir que si expresamos nuestro problema sólo con restricciones binarias entonces la estrategía de chequeo previo, junto con arco-consistencia nos permite obtener resultados óptimos.

Quedan algunos tópicos por comentar:

- **Simetría**: En muchos problemas reales las soluciones suelen pertenecer a grupos de soluciones equivalentes. Esto significa que es conveniente podar, ordenando las tareas a realizar, el árbol de búsqueda y no repitiendo soluciones equivalentes.

- **Solución óptima**: Hay un tipo particular de problemas de satisfacción de restricciones, que son aquellos que añaden a las tres componentes normales del problema `P = (Z, D,C)` una cuarta componente `f` que es una función de óptimización, de forma que no sólo buscamos una tupla que satisfaga el problema si no que además debe hacerlo con **coste** máximo o mínimo, según indique `f`. Normalmente, para lograr la mejor, se repite el proceso de evaluar el coste de una solución y si el de la siguiente es mejor se toma ésta como posible óptima hasta encontrar otra que pudiere ser mejor.

Con todo lo visto hasta ahora, hay problemas que debido a su explosión combinatoria se hacen mejor con restricciones y otros que si se conoce una solución óptima por investigación operativa es mejor resolverlos con esa heuríatica concreta.

2.5 CLP(FD): Búsqueda + Arco Consistencia

En `CLP(FD)` hay una serie de relaciones primitivas entre variables de dominio finito $\{=, \neq, >, \geq, <, \leq\}$ que se evaluan de la siguiente manera:

1. $\neq$ se implementa con chequeo previo, pues se verifica la consistencia de la restricción cuando alguna de las variables de la desigualdad se instancia.

2. `=` se implementa con mirar totalmente hacia adelante, pues se verifica la consistencia de la restricción cuando se produce cualquier tipo de modificación de los dominios de las variables involucradas.

3. {>, $\geq$, <, $\leq$} se implementan con mirar parcialmente hacia adelante, pues se verifica la consistencia de la restricción cuando se modifican los valores mínimo o máximo del dominio de alguna de las variables involucradas.

Ejemplo 18: Queremos resolver el puzzler critográfico siguiente:

$$\text{DONALD} + \text{GERALD} = \text{ROBERT}$$

Para resolver este problema empleamos la metodología habitual:

1. Dominio de las variables: `Dom(O,N,A,L,E,B,T) = [0..9]`, `Dom(D,G,R) = [1..9]`.

2. Restricciones a verificar: `distintos(D,O,N,A,L,G,E,R,B,T)` y además:
```
      100000*D + 10000*O + 1000*N + 100*A + 10*L + D
    + 100000*G + 10000*E + 1000*R + 100*A + 10*L + D
   #= 100000*R + 10000*O + 1000*B + 100*E + 10*R + T.
```

3. Generación de valores concretos: Hemos usado chequeo previo combinado con arco-consistencia y el rendimiento del algoritmo lo hemos medido por el númro de veces que detecta fallo y hace vuelta atrás y tiene 8018 fallos.

Si resolvemos el problema añadiendo 5 nuevas variables para el acarreo:

1. Dominio de las variables: `Dom(O,N,A,L,E,B,T) = [0..9]`, `Dom(D,G,R) = [1..9]`, `Dom(C1,C2,C3,C4,C5) = [0..1]`.

2. Restricciones a verificar: `distintos(D,O,N,A,L,G,E,R,B,T)` y además:
```
2*D = 10*C1 + T, 2*L + C1 = 10*C2 + R, 2*A + C2 = 10*C3 + E,
N + R + C3 = 10*C4 + B, E + C4 = 10*C5, D + G + C5 = R.
```

3. Generación de valores concretos: Con chequeo previo más arco-consistencia y sin ordenación de valores salen 212 fallos, pero si tomamos primero las variables con dominios menores entonces la solución aparece trás 14 fallos.

2.6 Restricciones de Intervalo

Antes de concretar nuestra elección de restricciones utilizables sobre intervalos finitos de enteros, estudiamos brevemente la **Aritmética de intervalos finitos de enteros**. Su estudio dentro del campo de la informática se retomó para resolver problemas de redondeo numérico con números en coma flotante.

1. Operadores: `A = [a..a']  B = [b..b']`

2. Operaciones:

   - `A + B = [(a + b) ..  (a' + b')]`
   - `A - B = [(a - b') ..  (a' - b)]`
   - `A * B = [min(a*b, a*b',b*a', a'*b')..max(a*b, a*b',b*a', a'*b')]`
   - `A / B = [a..a'] * [1/b'..1/b]` SI 0 $\in$`[b..b']` NO ESTANDAR

Esto da lugar a unas propiedades un tanto pecualiares.
`A - A` $\neq$ `[0..0]` `A / A` $\neq$ `[1..1]` `A - B = C` $\neq$ `A = C + B`

Ejemplo 19: Si `A = B = [1..3]`, `C = [-2..2]`, tenemos: `[1..3] - [1..3] = [-2..2]`
`[1..3] / [1..3] = [1/3..3]` `[1..3]`$\neq$`[-2..2] + [1..3] = [-1..5]`

El teorema fundamental de esta aritmética es: La evaluación de una expresión o de una función siguiendo las operaciones de la aritmética de intervalos establece una clausura (mayor o menor) que contiene al resultado correcto, pero no tiene completitud.

Ejemplo 20: Dadas dos representaciones de la misma función:
`F1(X) =` $2 * X^3 - 3 * X^2 + 1$ `F2(X) =` $X^2 * (2 * X - 3) + 1$
Si `X` $\in$ `[1..2]` entonces obtenemos: `F1(X)`$\in$`[-9..14]` `F2(X)`$\in$`[-3..5]`
Aunque el intervalo correcto es `[0..5]` que se corresponde con `[F1(1)..F2(2)]`.

No debemos confundir las restricciones que vamos a utilizar sobre intervalos, que tienen un caracter claramente relacional y representan a un conjunto de valores, con la aritmética de intervalos finitos de enteros que tiene un claro caracter funcional y está basada en el álgebra de intervalos.

Uno de los motivos fundamentales para trabajar con restricciones sobre dominios finitos ha sido evitar que PROLOG produjera incorrecciones en las operaciones en punto flotante. Así por ejemplo, dos valores distintos de una misma variable dan el mismo resultado.

- `X = 0.66, Y = 0.45, Z = 0.30, Z IS X*Y. (TRUE)`
- `X = 0.67, Y = 0.45, Z = 0.30, Z IS X*Y. (TRUE)`
- ¿ `X = 0.66 o X = 0.67 ?, Z` $\in$ `[0.2970, 0.3015]`

Por lo tanto, si deseamos mantener la naturaleza inversible de los predicados de PROLOG, hemos de mantener como equivalentes: `Z = X / Y` $\equiv$ `Z * Y = X`.

Para manejar dominio finitos se utilizan el concepto de pertenencia a un intervalo y la técnica se fundamenta en implementar de forma óptima una restricción primitiva `X::I` y reducir todas las demás restricciones sobre intervalos a ésta, si es posible. Esta técnica reduce mucho el tiempo de ejecución. El significado declarativo de `X::I` es `X` $\in$ `I` reduciendo así el manejo de:

- Unión, intersección y complementario de intervalos.
- Intervalos entre constantes.
- Máximos y mínimos de variables.

A esta clase de restricciones se las llama **indexical**. Los indexicales `X::I` se usan para jugar dos clases de roles:

- **Propagación de indexicales**: Utilizado para resolver restricciones.
  Cuando propagamos un indexical `X::I`, evaluamos `I` en el almacén actual `S` y le añadimos la nueva restricción `X::S(I)`, donde `S(I)` denota el valor de `I` en el almacén actual `S`.

- **Comprobación de indexicales**: Utilizado para comprobar restricciones. Cuando comprobamos un indexical `X::I`, comprobamos si el dominio de `X` en el almacén actual `S` está contenido en `S(I)`, si es así, la restricción correspondiente al indexical se da por cierta.

Ejemplo 21: Para calcular `Z = X + Y`, la compilación producirá:

```
X in [min(Z) - max(Y) ..  max(Z) - min(Y)]
Y in [min(Z) - max(X) ..  max(Z) - min(X)]
Z in [min(X) + min(Y) ..  max(X) + max(Y)]
```

Además, no es lo mismo manejar restricciones **monotonas**, como `B = 3*A` que transforman `A::[5..20]` y `B::[10..30]` en los intervalos `A::[5..10]` y `B::[15..30]`, que trabajar también con restricciones **no monotonas**, como `B = A*A` que transforman `A::[-5..5]` y `B::[-16..16]` en los intervalos `A::[-4..4]` y `B::[0..16]`.

Uno de los primeros problemas que hemos de resolver para manejar estas restricciones es la extensión de la unificación de términos a la primitiva `X::I`, que liga una variable cualquiera `X` con el intervalo de sus posibles valores `I`. Dado un término `X::I`, la unificación con otro término `Y::J` se lleva a cabo de la siguiente manera:

- Si `Y` es una variable no ligada entonces `Y::I`.
- Si `Y::J` entonces `X::I`$\cap$`J` y `Y::I`$\cap$`J`.
- Si `Y::J` no siendo `J` intervalo entonces FALLO
- Si `Y` tiene un valor constante k, entonces `Y::[k,k]`

Al contrario que en la programación lógica clásica, aquí está permitida la reunificación, mediante la que podemos disminuir los intervalos. Así, por ejemplo,

Ligaduras iniciales	`X::[0,2]`	`Y::[1,3]`	`Z::[4,6]`
`X + Y`			`[1, 5]`
`Z - X`		`[2, 6]`	
`Z - Y`	`[1, 5]`		
Ligaduras finales	`X:[1, 2]`	`Y:[2,3]`	`Z:[4,5]`

Se puede tratar de mejorar la representación de los intervalos, suponiendo que los podemos tomar como abiertos, semiabiertos o cerrados por los extremos. Así, `<`X_u`, `Y_v`>` representa a cualquier posible intervalo, siendo `u` y `v` corchetes o paréntesis.

```
[X, Y]     [X, Y)     (X, Y]     (X, Y)
```

El cálculo de qué corchetes corresponden se hace siguiendo la tabla:

+	)	[	]	(
)	)		)	
[		[		(
]	)		]	
(		(		(

siguiendo la regla: X_u `+` Y_v `=` $(X + Y)_{u+v}$

Ejemplo 22:

	$< 3.1_{]}, 6.8_{)} >$
+	$< 2.0_{]}, 3.0_{]} >$
	$< 5.1_{]}, 9.8_{)} >$

Se utiliza para el manejo de estos intervalos, tanto backtracking como heurísticas de división del trabajo.

- $-\infty$ indica el menor valor posible y $+\infty$ el mayor.
- Se da un trato especial a los puntos `X, Y` adyacentes en el intervalo, ya que `[X, X]` y `(X,Y)` no son divisibles, lo cual nos hace genarrar automáticamente los intervalos:
  - `[X, Y]` genera los 3 intervalos `[X, X], (X, Y), [Y, Y]`.
  - `(X, Y]` genera los 2 intervalos `(X, Y), [Y, Y]`.
  - `[X, Y)` genera los 2 intervalos `[X, X], (X, Y)`.

Ejemplo 23: El intervalo `[0.1234, 0.1235]` genera

`[0.1234, 0.1234], (0.1234, 0.1235), [0.1235, 0.1235]`.

Este es un buen método para garantizar la corrección y terminación con tres posibles resultados: FALLO SOLUCION SOLUCION INCOMPLETA O POTENCIAL

El modelos de caja transparente de los indexicales para dominios finitos tiene varias ventajas frente a los modelos opacos:

- Flexibilidad: ¿Mejor Formulación = Mejor Rendimiento?
- Rendimiento ≡ Eficiencia + Optimización del número de variables + Menor tamaño y mayor legibilidad del código + ...

Para el manejo de indexicales suponemos como primitiva `X::RANGO`, siendo `RANGO` un intervalo de números enteros denotado por `[a..b]` y además:

- Funciones o términos indexicales: `min(Y), max(Y), val(Y), dom(Y)`, etc,...
- Operadores entre enteros: `+ , - , *, /`, etc,...

Generamos además propagación del siguiente modo:

- `X::[1..3], Y::dom(X)` $\Longrightarrow$ `Y::[1..3]`
- `X::[1..3], Y::[dom(X)+1..max(X)]` $\Longrightarrow$ `Y::[2..3]`
- `X::[1..10], Z::[val(X)..20]` $\Longrightarrow$ **Retraso**
- `X::[1..1], Z::[val(X)..20]` $\Longrightarrow$ `Z::[1..20]`

Usamos el estrechamiento para manejar estos intervalos:

- Intersección de intervalos: `X::[a..b], X::[c..d]` $\Longrightarrow$ `X::[max(a,c)..min(b,d)]`
- Consistencia: Comprobación de dominios vacíos: `X::[9..0]` implica inconsistencia
- `X::[n..n]` $\Longleftrightarrow$ `X = n`

Los indexical se pueden utilizar para simular:

- Relaciones binarias: `X<Y` $\Longleftrightarrow$ `X::[0..max(Y)-1], Y::[min(X)+1..inf].`
- Relaciones que representan funciones:
```
plus(X, Y, Z)⇔ X::[min(Z)-max(Y)..max(Z)-min(Y)],
               Y::[min(Z) - max(X)..max(Z) - min(X)],
               Z::[min(X) + min(Y)..max(X) + max(Y)].
```
- Indexical en el dominio (1 = TRUE, 0 = FALSE)
```
AND(X, Y, Z) ⟺ Z::[min(X)*min(Y)..max(X)*max(Y)],
               X::[min(Z)..max(Z)*max(Y)+1-min(Y)],
               Y::[min(Z)..max(Z)*max(X)+1-min(X)].
```

Ejemplo 24:

- `X::[1..10], Y::[1..10], X<Y` $\Longrightarrow$ `X::[1..9], Y::[2..10]`
- `X::[1..10], Z::[4..9], plus(X, Y, Z)` $\Longrightarrow$ `Y::[-6..8]`
- `AND(1, 1, 1)  AND([0,1], 0, 0)  AND([0,1], 1, [0,1]).`

En dominios finitos los indexicals tienen muchas ventajas, como son **expresividad, transparencia y eficiencia** y algunas desventajas, como son:

- Linealidad de las restricciones: Aunque algunas restricciones no lineales pueden ser simuladas, en general ésto no es posible:
```
X^2 = Z ⟺ Z::min(X)*min(X)..max(X)*max(X),
           X::Red_exc(sqrt(min(Z)))..Red_defec(sqrt(max(Z))).
```
- Sólo para dominios discretos y finitos
- ¿ Terminación - Completitud en dominios continuos ?

Como el lenguaje `CLP(X)` $\equiv$ `PLR(X)` tiene las siguientes componentes:

	PROLOG
+	Declaraciones de dominios
+	Declaraciones de operadores
+	Mecanismos de unificación extendidos
	PLR(X)

necesitamos que el metainterprete de `CLP(X)`$\equiv$`PLR(X)` modifique al de PROLOG para que admita a partir de una lista de los objetivos a procesar:

- Un conjunto actual de restricciones.
- Un conjunto temporal de restricciones obtenido actualizando el conjunto anterior con las nuevas restricciones que aparecen en las reglas aplicadas.
- Con estas modificaciones el metainterprete es:
  - `metainterprete([ ], C, C).`
  - ```
    metainterprete([OBJ | ROBJ], RESTRA, RESTRN):-
                                metainterprete(OBJ, RESTRA, RESTRP),
                                metainterprete(ROBJ, RESTRP, RESTRN).
    ```
  - ```
    metainterprete(OBJ, RESTRA, RESTRN):- clause(OBJ, CUERPO, RESTREGLA),
                                mezcla-restric(RESTRA, RESTREGLA, RESTRP),
                                metainterprete(CUERPO, RESTRP, RESTRN).
    ```
- El predicado más importante y complejo es el determina la vuelta al punto de elección y mantiene el almacén de las variables de forma incremental:
  `mezcla-restric(RESTRAC, RESTREGLA, RESTRTP).`

Capítulo 3

Programación con restricciones sobre los números reales

Presentamos una definición precisa de la estructura $\mathcal{R}$, las restricciones primitivas, los programas en CLP($\mathcal{R}$) y el modelo operacional subyacente.

La estructura $\mathcal{R}$ está formada por dos clases de objetos:

- Los números reales con las operaciones $\{+, -, *, /\}$ forman el conjunto de los términos reales: `t` $\in$ `TR::= r`$\in \mathcal{R}$ | `X REAL | t1 + t2 | t1 - t2 | t1 * t2`
- Los términos no interpretados forman el universo de Herbrand:
  `th` $\in$ `UH::= X VAR | const | f(th1,...,thN)`
- `t` $\in$ `TERMINOS::= f(t1,...,tM), para ti` $\in$ `UH` $\bigcup$ `TR = T`.

Las restricciones en CLP($\mathcal{R}$) son:

- Una restricción primitiva R es satisfactible o resoluble sii existe una sustitución de sus variables por números reales o por términos de Herbrand de forma que aplicada a $\mathcal{R}$ dá cierto.
- Restricciones aritméticas primitivas: `t1`$\geq$`t2, t1`$\leq$`t2, t1 = t2`.
- Restricciones no-aritméticas primitivas: `th1 = th2` (Unificación)

Un CLP($\mathcal{R}$) programa es un conjunto finito de reglas, siendo:

- `A::= p(t1,...,tK)`, para `ti`$\in$ `T`, con `p`$\notin\{\geq, \leq, \neq\}$ Atomos
- `A0:- A1, A2,...,Ak.` donde `Ai`(i$\geq$1) son átomos o restricciones primitivas: Regla

Un CLP($\mathcal{R}$) objetivo es `?- B1,...,Br.` donde los `Bj` son átomos o restricciones primitivas. Las restricciones primitivas del objetivo se clasifican en dos grupos, las que se resuelven y las que se invernan.

Ejemplo: Un programa que calcula el N-ésimo número de Fibonacci podría ser:

- `fib(0, 1).  fib(1, 1).`
- `fib(N, X1 + X2) :- N > 1, fib(N - 1, X1), fib(N - 2, X2).`
- Para calcular un número `A` con `fib(A)` entre 80 y 90 lanzo el objetivo
  `?- 80 <= B, B <= 90, fib(A,B).` y obtengo la solución: `A = 10, B = 89.`

La estratégia de selección de objetivos es inherente al modelo operacional de CLP($\mathcal{R}$) y tiene que especificar claramente cuando una restricción debe dormirse o despertarse y cuál debe ser el siguiente átomo a resolver.

En un objetivo `G` pueden distinguirse tres partes:

- `A1, A2,..., AN` con N$\geq$0, los átomos por resolver.
- `S1, S2,..., SM` con M$\geq$0, las restricciones ya resueltas.
- `D1, D2,..., DK` con K$\geq$0, las restricciones dormidas.

Dado el objetivo `G: ?- {A1, A2,..., AN, S1, S2,..., SM, D1, D2,..., DK}`, diremos que hemos generado un nuevo objetivo `G'` en una etapa sii

ó bién `G' ≡ {A1,...,AN},{S1,...,SM,Di},{D1,.-.,DK}`

- contiene la misma sucesión de átomos `A1, A2,..., AN`
- añade una restricción dormida `Di` al conjunto de las resueltas `S1,...,SM`
- elimina la restricción `Di` del conjunto de las dormidas `D1,...,DK`

ó bién `G' ≡ {A1,...,Ai-1,B1,...,Bs,Ai+1,...,AN},{S1,...,SM},`
`{D1,...,DK,Ai≡B,D11,...,D12}` usa `B:- {B1,...,Bs},{D11,...,D12}`

- los átomos `A1,...,Ai-1,B1,...,Bs,Ai+1,...,AN` se obtienen al colocar los átomos del cuerpo de la cláusula usada.
- las restricciones resueltas `S1,...,SM`, no varian
- las restricciones dormidas son `D1,...,DK, Ai≡B, D11,...,D12` obtenidas añadiendo todas las restricciones del cuerpo de la cláusula junto con la unificación del átomo de la cabeza.

Las restricciones del objetivo inicial están dormidas. Además, para que la estratégia de resolución coincida con PROLOG en programas lógicos puros, se utiliza resolución con el primer átomo o la primera restricción por la izquierda, salvo que la restricción sea no lineal, en cuyo caso se duerme. También la elección de las reglas se hace como en PROLOG.

Si a partir de `G` obtenemos una derivación infinita, entonces el programa cicla.
Si a partir de `G` obtenemos una derivación finita, entonces

- Si `GK` sólo contiene restricciones resueltas, entonces **EXITO**
- Si `GK` contiene restricciones resueltas y dormidas, entonces **EXITO CONDICIONAL**
- En otro caso, **FALLO FINITO**
- En los casos con éxito, las restricciones obtenidas se llaman Restricciones Respuesta

Ejemplo: Este ejemplo pone de manifiesto como las restricciones no lineales se resuelven sólo si sus variables toman valores explícitos obtenidos al resolver las restricciones lineales en las que aparecen. Dado el programa `ohm(V,I,R) :- V = I * R.` La ejecución del objetivo `?- ohm(V1,I,R1), ohm(V2,I,R2), V = V1 + V2, R1 = 15, R2 = 5.` es:

Se llama al predicado `ohm` obteniendo: `V1 = V', I = I', R1 = R', V' = I' * R', ohm(V2,I,R2), V = V1 + V2, R1 = 15, R2 = 5.`

Las tres primeras restriciones se resuelven y la cuarta se duerme: `V1 = V', I = I', R1 = R', V' = I' * R', ohm(V2,I,R2), V = V1 + V2, R1 = 15, R2 = 5.`

Se llama de nuevo al predicado `ohm` obteniendo: `V1 = V', I = I', R1 = R', V' = I'*R', V2 = V'', I = I'', R2 = R'', V'' = I''*R'', V = V1+V2, R1=15, R2=5.`

Se resuelven tres y se duerme la cuarta: `V1 = V', I = I', R1 = R', V' = I'*R', V2 = V'', I = I'', R2 = R'', V'' = I''*R'', V = V1+V2, R1 = 15, R2 = 5.`

Al resolver `V = V1 + V2` y unificar `R1`, la restricción `V' = I'*R'` se convierte en lineal y se resuelve `V1 = V', I = I', R1 = R', V' = I'*R', V2 = V'', I = I'', R2 = R'', V'' = I''*R'', V = V1+V2, R1 = 15, R2 = 5.`

Se unifica `R2` con `5` y la restricción `V'' = I''*R''` se convierte en lineal y se resuelve, `V1 = V', I = I', R1 = R', V' = I'*R', V2 = V'', I = I'', R2 = R'', V'' = I''*R'', V = V1+V2, R1 = 15, R2 = 5.`

Por último, es necesario una operación estética para mostrar una solución simplificada `V' = V1, V'' = V2, I = I'= I'', R' = R1, R'' = R2, V1 = 0.75*V, I = 0.05*V, R1 = 15, V2 = 0.25*V, R2 = 5.`

En este objetivo la única variable no restringuida es `V`, y la solución observable es: `V1 = 0.75*V, I = 0.05*V, R1 = 15, V2 = 0.25*V, R2 = 5`

Al admitir restricciones no lineales que son indecidibles, nos aparecen dos problemas:

- Proseguir indefinidamente las derivaciones aunque haya una derivación con éxito. Incompletitud relativa como en PROLOG.
- Obtener una colección de restricciones respuesta que pueda ser irresoluble. Esto es debido a que las respuestas representan un subconjunto del espacio de soluciones y tendría que garantizarse que el conjunto no es vacío.

3.1 Problemas aritméticos y de planificación en CLP($\mathcal{R}$)

Las restricciones primitivas están pensadas para expresar propiedades locales del problema a resolver, mientras que las propiedades globales de un problema deben ser expresadas mediante las reglas del programa.

La forma más simple de representar propiedades globales es por composición jerarquizada: `p(...):- restricciones primitivas, p1(...), p2(...),...,pk(...)` con `pi(...)` definiciones de partes independientes del sistema.

Ejemplo: Si usamos el predicado anterior, que describe una propiedad local, podemos escribir un programa más general de la siguiente forma:

```
circuito-par(V,I,R1,R2) :  - I1 + I2 = I, ohm(V,I1,R1),ohm(V,I2,R2).
  circuito-sec(V,I,R1,R2) :  - V1 + V2 = V, ohm(V1,I,R1),ohm(V2,I,R2).
```

Jerárquicamente podemos construír predicados cada vez más complejos:

```
par-serie(V,I,R1,R2,R3,R4):- V1+V2 = V, circuito-par(V1,I,R1,R2),
                                          circuito-par(V2,I,R3,R4).
```

La propia naturaleza de este lenguaje permite obtener de forma sencilla y compacta restricciones como salida del programa:

- Descripción de objetos sin forma sintáctica explícita:
  `(X*X = 2 , X`$\geq$`0)` representa a `sqrt(2)`.
- Representamos finitamenye un número infinito de objetos:
  `(0` $\leq$`X, X`$\leq$ `1)` representa a `[0..1]`.

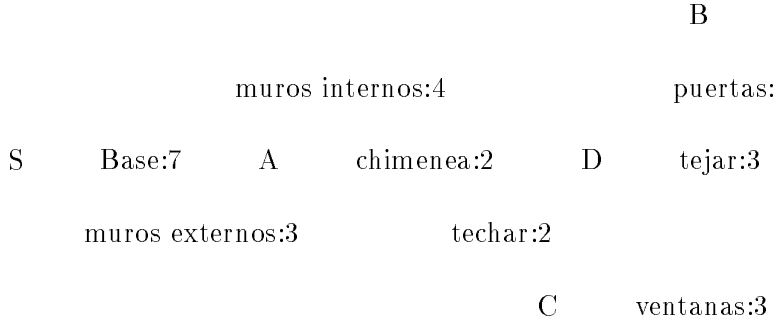
Ejemplo: Un ejemplo más sofisticado que expresa la relación entre el principal de una hipotéca, el interés compuesto mensual, el tiempo en meses y el balance sería:

- `hipoteca(P, T, I, B, MP):- T>0, T`$\leq$`1, B + MP = P*(1 + I).`
- `hipoteca(P, T, I, B, MP):- T>1, hipoteca(P*(1 + I)-MP, T-1, I, B, MP).`

Posibles objetivos a ejecutar:

- `G1:- hipoteca(120000,120,0.01,0,MP).` `MP = 1721.651381`
- `G2:- hipoteca(P,120,0.01,0,1721.651381).` `P = 120000`
- `G3:- hipoteca(P,120,0.01,0,MP).` `P = 69.700522*MP`
- `G4:- hipoteca(P,120,0.01,B,MP).` `P = 0.302995*B + 69.700522*MP`
- ```
  G5:- hipoteca(999, 3, INT, 0, 400).
  hipoteca(999*(1 + INT) - 400, 2, INT, 0, 400)
  hipoteca((599 + 999*INT)*(1 + INT) - 400, 1, INT, 0, 400)
  0 + 400 = ((599 + 999 * INT) * (1 + INT) - 400) * (1 + INT)
  La respuesta es:  400 = (-400 + (599 + 999 * INT)*(1 + INT))*(1 + INT)
  ```
  ```

Ejemplo: El siguiente gráfico muestra el orden en que deben llevarse a cabo las tareas de construcción de un chalet.



La familia de restricciones que genera es: $T_S \geq 0, T_A \geq T_S + 7, T_B \geq T_A + 4$, $T_C \geq T_A + 3, T_D \geq T_A + 2, T_D \geq T_C + 2, T_E \geq T_C + 3, T_E \geq T_B + 2, T_E \geq T_D + 3$

Si comenzamos suponiendo que la obra comienza para $T_S = 0$, entonces una posible solución sería: $\{T_S = 0, T_A = 7, T_B = 11, T_C = 10, T_D = 12, T_E = 15\}$, de lo que se deduce que la obra no se puede realizar en menos de 15 días, puesto que hemos tomado valores mínimos.

La metodología comprueba y genera se aplica en CLP para resolver problemas combinatorios y de satisfacción de restricciones. Las restricciones guián el espacio de búsqueda:

- Cuando se cumple por generar activamente de los valores solución.
- Cuando fallan por podar el árbol de búsqueda.

Un programa de comprueba y genera típico tiene la forma siguiente:

```
p(...):- restricciones primitivas,
                    p1(...), p2(...),...,pk(...),
                              generadores-de-valores.
```

siendo `pi(...)` predicados. Los `generadores-de-valores` se utilizan para instanciar las variables sobre un dominio particular mejorando así esta técnica.

Un resolutor parcial sólo falla si está totalmente seguro, es decir, semidecide:

Efectivamenteinsat(Almacén de restricciones + nueva restricción)
ENTONCES Fallo
EN OTRO CASO Cierto (aunque pudiese ser insatisfactible)

Por eso, siempre hay que lograr un equilibrio entre completitud y eficiencia: ¿ Qúe es mejor podar el espacio de búsqueda, tardando mucho o usar heurísticas sencillas, aunque el resolutor sea parcial, pero rápido ?

El sistema que vamos a estudiar tiene las siguientes componentes:

- Motor de inferencia (Programación lógica + Vuelta átras).
- Interface (Resuelve restricciones fáciles).
- Resolutor de ecuaciones lineales (Método de Gausss).
- Resolutor de inecuaciones (Método del Simplex modificado)
- Manipulador de restricciones no lineales(Duerme/Despierta),
- Módulo de salida (Maquillador de respuestas)

3.2 Las restricciones con reales: Método del simplex

De los dos métodos de manejo de ecuaciones e inecuaciones que tiene el sistema que vamos a estudiar, no comentaremos nada sobre Gauss por ser demasiado conocido, sin embargo, si nos dedicaremos a estudiar la variante de dos fases del método del simplex. Para ello, necesitamos una serie de definiciones auxiliares:

Un problema de programación lineal es un problema de la forma:

- **maximiza, minimiza** $z = c^t x$ con $c, x \in (n \times 1)$
- **sujeto a** $Ax(\leq, =, \geq)b$ con $x \geq 0$, $A \in (n \times m)$, $b \in (m \times 1)$

Un problema de programación lineal está en forma **canónica** sii

- La función objetivo es de la forma maximizar.
- Todas las restricciones son **inecuaciones del tipo** ($\leq$).
- Todas las variables son no negativas.

Un problema de programación lineal está en forma **standard** sii

- La función objetivo es de la forma maximizar.
- Todas las restricciones son **ecuaciones**.
- Todas las variables son no negativas.

Ejemplo:
minimiza `Z = 3*X1 - X2 + 2*X3`
sujeto a

```
   X1 + 2*X2        ≤ 4
2*X1          + X3 = -3
   X1 + 3*X2  + X3 ≥ 8
X1, X3 ≥ 0, X2 sin restringir
```

Transformación en canónica-standard:
maximiza `EXP = -3*X1 + A - B + 2*X3`
sujeto a

```
   X1 + 2*A - 2*B      ≤ 4
2*X1            + X3 = -3
- X1 - 3*A + 3*B - X3 ≤ 8
```

`X1, X3, A, B` ≥ 0 todas las variables no negativas.

maximiza `EXP = -3*X1 + A - B + 2*X3`
sujeto a

```
   X1 + 2*A - 2*B      + S1       = 4
2*X1            + X3             = -3
- X1 - 3*A + 3*B - X3       - S2 = - 8
```

`X1, X3, A, B, S1, S2` ≥ 0 `S1, S2` se llaman **variables holgura**.
Estas manipulaciones añaden 4 variables nuevas.

3.2.1 Definiciones y resultados previos

El sistema de ecuaciones $Ax = b$ se resuelve tomando una submatriz B de A de orden m y no singular, y comprobando que $x = B^{-1}b \geq 0$.

- B es **una base factible** y x es una **solución básica factible**.
- El número máximo de soluciones factibles es $\binom{n}{m}$.
- Se llama **región factible** F al conjunto de todas las soluciones factibles $F = \{x \mid Ax = b \ y \ x \geq 0\}$
- Si $F = \emptyset$ el problema asociado se denomina **infactible**.

Ejemplo gráfico:
maximiza `Z = 2*X1 + X2`
sujeto a

```
5*X1 + 2*X2 ≤ 10:  R1
3*X1 + 5*X2 ≤ 15:  R2
X1, X2 ≥ 0:R3
```

El punto de corte de `r1:5*X1 + 2*X2 = 10` y `r2:3*X1 + 5*X2 = 15` es `B:(20/19,45/19)`, ademas `C:(2,0)`$\in$`r1` y `A:(0, 3)`$\in$`r2`.
Por último, `O:(0, 0)`$\in$ `R1`$\cap$`R2`$\cap$`R3`.

Punto	(X1, X2)	Z
O	(0, 0)	0
A	(0, 3)	3
B	(20/19,45/19)	85/19
C	(2,0)	4

La solución está en el extremo del conjunto F.

Ejemplo:
maximiza `Z = 2*X1 + 4*X2 + X3`
sujeto a

$$\begin{aligned} &\texttt{2*X1 + 2*X2} \leq 6 \\ &\texttt{X1 + 4*X2 - X3} \leq 12 \\ &\texttt{X1, X2 , X3} \geq 0 \end{aligned}$$

En forma standard:
maximiza `Z = 2*X1 + 4*X2 + X3 + 0*X4 + 0*X5`
sujeto a

$$\begin{aligned} &\texttt{2*X1 + 2*X2 + X4} = 6 \\ &\texttt{X1 + 4*X2 - X3 + X5} = 12 \\ &\texttt{X1, X2 , X3 , X4, X5} \geq 0 \end{aligned}$$

En forma matricial:

$$A = \begin{pmatrix} 2 & 2 & 0 & 1 & 0 \\ 1 & 4 & -1 & 0 & 1 \end{pmatrix} \qquad b = \begin{pmatrix} 6 \\ 12 \end{pmatrix}$$

$$c = \begin{pmatrix} 2 & 4 & 1 & 0 & 0 \end{pmatrix} \qquad x = \begin{pmatrix} X1 & X2 & X3 & X4 & X5 \end{pmatrix}$$

El problema a resolver es $\begin{cases} \textbf{maximiza} & \mathtt{z = c^t x} & \textit{con} & \mathtt{c, x} \in (\mathtt{1} \times \mathtt{5}) \\ \textbf{sujeto a} & \mathtt{Ax^t = b} & \textit{con} & \mathtt{x} \geq \mathtt{0}, \mathtt{A} \in (\mathtt{2} \times \mathtt{5}), \mathtt{b} \in (\mathtt{2} \times \mathtt{1}) \end{cases}$

Una base B obtenida a partir de la primera y segunda columna de A y su solución asociada resolviendo el sistema de ecuaciones son:

$$B = \begin{pmatrix} 2 & 2 \\ 1 & 4 \end{pmatrix} \qquad x_B = B^{-1}b = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

Luego: $x_{B1} = x_1 = 0; x_{B2} = x_2 = 3$ es una solución **básica factible degenerada**

Una base B obtenida a partir de la primera y tercera columna de A y su solución asociada resolviendo el sistema de ecuaciones son:

$$B = \begin{pmatrix} 2 & 0 \\ 1 & -1 \end{pmatrix} \qquad x_B = B^{-1}b = \begin{pmatrix} 3 \\ -9 \end{pmatrix}$$

Luego: $x_{B1} = x_1 = 3; x_{B2} = x_3 = -9 \leq 0$ es una solución **básica infactible**

Una base B obtenida a partir de la cuarta y quinta columna de A y su solución asociada resolviendo el sistema de ecuaciones son:

$$B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad x_B = B^{-1}b = \begin{pmatrix} 6 \\ 12 \end{pmatrix}$$

Luego: $x_{B1} = x_4 = 6; x_{B2} = x_5 = 12$ es una solución **básica formada únicamente por variables holgura**.

Si $A = (a_1, \ldots, a_n)$ entonces cada a_i es un vector $(m \times 1)$ y si A tiene m vectores linealmente independientes, con ellos se forma $B = (b_1, \ldots, b_m)$. Si a_j es uno de los vectores que no pertenece a la base se cumple que $a_j = By_j \Leftrightarrow y_j = B^{-1}a_j$ y $z_j = (c_B)^t y_j$ es un número que nos va a guiar en la aplicación del algoritmo del simplex. Si B = Id entonces $y_j = a_j$ p.t. j

Para la base $B = \begin{pmatrix} 2 & 2 \\ 1 & 4 \end{pmatrix}$ calculamos $y_1 = B^{-1}a_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $y_2 = B^{-1}a_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

$$y_3 = B^{-1}a_3 = \begin{pmatrix} 1/3 \\ -1/3 \end{pmatrix} \quad y_4 = B^{-1}a_4 = \begin{pmatrix} 2/3 \\ -1/6 \end{pmatrix} \quad y_5 = B^{-1}a_5 = \begin{pmatrix} -1/3 \\ 1/3 \end{pmatrix}$$

Como $(c_B)^t = (2 \quad 4)$, las distintas soluciones $z_j = (c_B)^t y_j$ son $z_1 = 2; z_2 = 4; z_3 = -2/3; z_4 = 2/3; z_5 = 2/3$. Y $z_2 = 4$ es una solución **factible óptima** pues verifica $z = c^t X \geq c^t X'$, para toda $X' \in F$ y a z se le denomina **valor óptimo**.

Teorema: El conjunto de soluciones factibles de un problema lineal standard es un conjunto convexo y cerrado. Si A es una matriz con $rango(A) = m$ y b es un vector $(m \times 1)$ entonces el conjunto de soluciones $F = \{x \mid Ax = b \ y \ x \geq 0\}$ es un poliedro.
X es una solución factible $\Leftrightarrow$ X es un punto extremo de F.

Un problema lineal standard puede tener:

- Solución óptima acotada (**Unica**)
- Solución óptima múltiple (**Varias**)
- Solución óptima no acotada (**Inalcanzable**)
- Irresoluble.

3.2.2 El simplex modificado

El esquema del método del simplex es el siguiente:

PASO 0: Búsqueda de una solución básica factible (Punto Extremo).

REPETIR

- *PASO 1*: SI existe una solución básica factible adyacente que mejora el objetivo
  - ENTONCES PASO 2
  - SINO Solución óptima
- *PASO 2*: Determina esa solución mejor y vuelve al PASO 1

HASTA QUE (Solución acotada infactible)

Para realizar el paso 1 necesitamos poder determinar

- La optimalidad o mejora de la solución básica factible que ya hemos calculado.
- Determinar que variable es candidata a **dejar** la base.
  Dada la solución factible $x_B = B^{-1}b$, **SI** el vector columna a_j que no está en la base tiene un $y_{i,j} \geq 0$ para algún i **ENTONCES** puede entrar en la base en lugar de un b_k que verifique $x_{B_k}/y_{k,j} = \min\{x_{B_i}/y_{i,j} \text{ con } y_{i,j} \geq 0\}$.
- Determinar que variable es candidata a **entrar** en la base y la regla de **parada**.
  Dada la solución factible $x_B = B^{-1}b$ con valor $z = c_B{}^t x$, la actual solución es **óptima** si $z_j - c_j \geq 0$, para toda a_j de A.
- La solución óptima es única si $z_j - c_j > 0$ para toda a_j de A.
- La variable que entra en la base es aquella que verifica que $(z_j - c_j)$ es el más negativo de todos los valores para los a_j de A que no están en la base. Si hay varios con esta propiedad se toma uno cualquiera de ellos.

Ejemplo:
maximiza `Z = 2*X1 + 3*X2`
sujeto a

```
  X1 + X2 ≤ 9
3*X1 + X2 ≤ 12
X1, X2 ≥ 0
```

En forma standard:
maximiza `Z = 2*X1 + 3*X2 + 0*X3 + 0*X4`
sujeto a

```
  X1 + X2 + X3       = 9
3*X1 + X2       + X4 = 12
X1, X2 , X3, X4 ≥ 0
```

Base inicial con variables de holgura: $B = Id = B^{-1}$, por tanto $x_B = B^{-1}b = b$, luego $X3 = 9; X4 = 12$. El par $(X3, X4)$ es un solución **básica factible**, y su valor calculado es $Z = 0$.

Calculemos si es posible mejorar la solución para a_1 y a_2.

$$y_1 = B^{-1}a_1 = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \qquad y_2 = B^{-1}a_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Sabemos que nuestra solución actual es $z_1 = 0; z_2 = 0$, además $c_1 = 2, c_2 = 3$, por lo tanto, $z_1 - c_1 = -2; z_2 - c_2 = -3$ **Variable de entrada** $X2$.

Como hay un candidata a entrar, escojamos que variable ha de salir, es decir, calculemos $x_{B_k}/y_{k,2} = \min\{x_{B_1}/y_{1,2},\ x_{B_2}/y_{2,2}\} = \min\{9/1,\ 12/1\} = 9$. Así pues, la **Variable de salida** es $X3 = x_{B_1}$.

Nueva base inicial con variables $X2$ y $X4$:

$$B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \quad B^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \quad x_B = B^{-1}b = \begin{pmatrix} 9 \\ 3 \end{pmatrix}$$

El par $(X2 = 9, X4 = 3)$ es un solución **factible**, y su valor calculado es $Z = 27$.
Calculemos si es posible mejorar la solución para a_1 y a_3.

$$y_1 = B^{-1}a_1 = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \qquad y_3 = B^{-1}a_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Sabemos que la solución actual es $z_1 = 3; z_2 = 3$, además $c_1 = 2, c_3 = 0$, luego, $z_1 - c_1 = 1; z_3 - c_3 = 3$, como ninguno es negativo, estamos ante un caso de no hay **posible mejora**. **Solución óptima** X = (0, 9, 0, 3), con valor calculado $Z = 27$.

Si dibujamos Z = 2*X1 + 3*X2, en los puntos A = (0, 9), el valor es Z = 27, en B = (1'5, 7'5) obtenemos Z = 25'5 y en C = (4,0) sale Z = 8.

Para poder detener el algoritmo necesitamos detectar condiciones de infactibilidad, no acotación o soluciones óptimas alternativas.

Infactibilidad

- Toda restricción de la forma $\sum_{j=1}^{n} a_{p,j} * x_j \geq b_p$ se puede transformar en $\sum_{j=1}^{n} a_{p,j} * x_j - t_p + r_p = b_p$, tomando t_p (variable **holgura**) y r_p(variable **artificial**)
- Como todos los x_j han de ser no negativos, la variable artificial se añade para poder asegurar que para $x_j = 0$ hay un valor de t_p y r_p que sea mayor o igual que cero y que permite tomar estos valores como inicio del proceso del simplex.
- Una solución factible $x_B = B^{-1}b$ con $z_j - c_j \geq 0$ para toda a_j de A es **infactible** si alguna de las variables de x_B es **artificial** y **positiva**.

Ejemplo: Si partimos del sistema:
$2 * X1 + X2 \leq 8 \Longleftrightarrow 2 * X1 + X2 + S1 = 8$
$4 * X1 + 3 * X2 \geq 14 \Longleftrightarrow 4 * X1 + 3 * X2 - T1 = 14$
si $X1 = X2 = 0$ entonces $S1 = 8; T1 = -14 \leq 0$, que **no** es una solución básica factible.

Pero, si transformamos el sistema de manera que añadimos la variable $R1$:
$2 * X1 + X2 + S1 = 8$
$4 * X1 + 3 * X2 - T1 + R1 = 14$
para $X1 = X2 = 0$ obtenemos $S1 = 8; T1 = 0; R1 = 14$ que puede ser admitida como solución básica factible para comenzar el algoritmo del simplex, aunque una vez acabado $F = \emptyset$.

No acotado Sea una solución factible $x_B = B^{-1}b$ **si** existe una a_j no básica con $z_j - c_j < 0$ y $y_{i,j} \leq 0$ **entonces** el problema es **no acotado**.

Ejemplo:
maximiza `Z = X1 + X2`
sujeto a

```
  X1 - X2 + X3      = 2
5*X1 -2*X2     + X4 = 16
```

`X1, X2 , X3, X4` ≥ 0, `X3, X4` variables holgura

Tomando $B = Ide$, obtenemos que $X3 = 2$ y $X4 = 16$, puesto que:

$$x_B = B^{-1}b = \begin{pmatrix} 2 \\ 16 \end{pmatrix} \qquad y_1 = B^{-1}a_1 = \begin{pmatrix} 1 \\ 5 \end{pmatrix} \qquad y_2 = B^{-1}a_2 = \begin{pmatrix} -1 \\ -2 \end{pmatrix}$$

Por tanto, $z_1 - c_1 = -1; z_2 - c_2 = -1$; $X2$ tiene $(z_2 - c_2 < 0)$ y todos los $y_2 \leq 0$.

La región asociada al problema es:
$X1 - X2 \leq 2$
$5 * X1 - 2 * X2 \leq 16$
Punto común (4,2), en esta región $X1 + X2$ crece de forma no acotada.

Soluciones óptimas alternativas: Si $z_j - c_j = 0$ para alguna variable de fuera de la base **entonces** al cambiar dicha variable la solución no gana en aproximación.

Dada una solución factible **óptima** $x_B = B^{-1}b$ **si** existe una a_j no básica con $z_j - c_j = 0$ **entonces** el problema tiene solución óptimas **alternativas**.

Ejemplo:
maximiza `Z = 6*X1 + 10*X2`
sujeto a

```
5*X1 + 2*X2 ≤ 10
3*X1 + 5*X2 ≤ 15
```

`X1, X2` ≥ 0

Se transforma en:
maximiza `Z = 6*X1 + 10*X2 + 0*X3 + 0*X4`
sujeto a

```
5*X1 + 2*X2 + X3      = 10
3*X1 + 5*X2     + X4 = 15
```

`X1, X2 , X3, X4` ≥ 0, `X3, X4` variables holgura

Los puntos $A = (0,3); B = (20/19, 45/19); C = (2,0)$, determinan los siguientes valores de Z, $Z_A = 6*0+10*3 = 30$, $Z_B = 6*20/19+10*45/19 = 30$ y $Z_C = 6*2+10*0 = 12$. Está claso que en este caso existen al menos otras dos soluciones óptimas.

Base inicial con variables holgura y $B = Ide$, $x_B = B^{-1}b$, luego $X1 = 0; X2 = 0; X3 = 10; X4 = 15; Z = 0$. El par $(X3, X4)$ es la solución básica factible y Z es el valor calculado.

Calculamos si es posible una mejora para a_1 y a_2.

$$y_1 = B^{-1}a_1 = \begin{pmatrix} 5 \\ 3 \end{pmatrix} \qquad y_2 = B^{-1}a_2 = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

Como $z_1 = 0; z_2 = 0; z_1 - c_1 = -6; z_2 - c_2 = -10$, la **variable de entrada** es $X2$.

Calculamos $x_{B_k}/y_{k,2} = \min\{x_{B_1}/y_{1,2}, x_{B_2}/y_{2,2}\} = \min\{10/2, 15/5\} = 3$, así pues, la **variable de salida** es $X4 = x_{B_2}$.

Sea B la nueva base con variables $X3$ y $X2$, entonces $x_B = B^{-1}b$, luego $X1 = 0; X2 = 3; X3 = 4; X4 = 0; Z = 30$ y el par $(X3, X2)$ es una solución factible y Z es el valor calculado.

Calculamos si es posible una mejora para a_1 y a_4:

$$y_1 = B^{-1}a_1 = \begin{pmatrix} 19/5 \\ 3/5 \end{pmatrix} \qquad y_4 = B^{-1}a_4 = \begin{pmatrix} -2/5 \\ 1/5 \end{pmatrix}$$

Como $z_1 = 4; z_4 = 2; z_1 - c_1 = -2; z_4 - c_4 = 2$ la **variable de entrada** es $X2$.

Calculamos $x_{B_k}/y_{k,1} = \min\{x_{B_1}/y_{1,1}, x_{B_2}/y_{1,2}\} = \min\{4 * 5/19, 3 * 5/3\} = 20/19$, así pues, la **variable de salida** es $X3 = x_{B_1}$.

Sea B la nueva base con variables $X1$ y $X2$, entonces $x_B = B^{-1}b$, luego $X1 = 20/19; X2 = 45/19; X3 = 0; X4 = 0; Z = 30$ y el par $(X1, X2)$ es una solución factible y Z es el valor calculado.

Calculamos si es posible una mejora para a_3 y a_4:

$$y_3 = B^{-1}a_3 = \begin{pmatrix} 5/19 \\ -3/19 \end{pmatrix} \qquad y_4 = B^{-1}a_4 = \begin{pmatrix} -2/19 \\ 5/19 \end{pmatrix}$$

Como $z_3 = 0; z_4 = 38/19; z_3 - c_3 = 0 \geq 0; z_4 - c_4 = 38/19 \geq 0$ no hay **variable de entrada**, pues si tomasemos de entrada a $X3$, tendríamos: $x_{B_k}/y_{k,3} = \min\{x_{B_1}/y_{1,3}, x_{B_2}/y_{1,3}\} = \min\{(20/19)/(5/19)\} = 20/5$, ya que el negativo no se considera y con ello estaríamos en el paso anterior.

3.2.3 Simplex con tabla

El primer paso de este algoritmo consiste en la construcción de una tabla inicial partiendo de los datos del problema, por ejemplo:

maximiza `Z = 3*X1 + 2*X2`
sujeto a
`2*X1 + 3*X2` $\leq$ `12`
`2*X1 + X2` $\leq$ `15`
`X1, X2` ≥ 0

La base inicial está formada por las variables holgura $X3$ y $X4$ con coeficientes $c_3 = c_4 = 0$ y matriz asociada $B = Ide = B^{-1}$ se calculan los $y_j = B^{-1}a_j$:

$$Y = \begin{pmatrix} 2 & 3 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix} \qquad x_B = B^{-1}b = \begin{pmatrix} 12 \\ 8 \end{pmatrix}$$

Además $z_j - c_j = (c_B) - c_j = (0 \quad 0)y_j - c_j = -c_j$ y el valor calculado asociado es $z = (c_B) * x_B^t = (0 \quad 0) * (12 \quad 8)^t = 0$

	c_j	$c_1 = 3$	$c_2 = 2$	$c_3 = 0$	$c_4 = 0$	
c_B	V. Básicas	x_1	x_2	x_3	x_4	x_B
c_{B_1}	$x_{B_1} = x_3$	$y_{1,1} = 2$	$y_{1,2} = 3$	$y_{1,3} = 1$	$y_{1,4} = 0$	$x_{B_1} = 12$
c_{B_2}	$x_{B_2} = x_4$	$y_{2,1} = 2$	$y_{2,2} = 1$	$y_{2,3} = 0$	$y_{2,4} = 1$	$x_{B_2} = 8$
		$z_1 - c_1 = -3$	$z_2 - c_2 = -2$	$z_3 - c_3 = 0$	$z_4 - c_4 = 0$	$z = 0$

Descripción del algoritmo del simplex modoficado:

- **PASO 0:** Construcción de la tabla inicial.
- **PASO 1: Si** existe j con $z_j - c_j < 0$ *posible mejora.*
  - **entonces PASO 3**
  - **sino** {para todo j con $z_j - c_j \geq 0$} **PASO 2**
- **PASO 2: Si** no hay variables artificiales básicas positivas
  - **entonces** *solución óptima*
  - **sino** *infactible*
- **PASO 3: Si** para algún j con $z_j - c_j < 0$ su vector asociado $y_j \leq 0$
  - **entonces** *problema no acotado*
  - **sino PASO 4**
- **PASO 4:** Calcular $\min\{z_j - c_j \mid z_j - c_j \leq 0\} =$ m (* si hay varios tomo uno*)
  Sea x_k la **variable de entrada** con $z_k - c_k = m$, k columna **pivote**
  Calcular $\min\{x_{B_i}/y_{i,j} \mid y_{i,j} > 0\} =$ m', sea x_{B_r} y x_r la **variable de salida** asociada a m', r fila **pivote**
  Construcción de una nueva tabla con **elemento pivote** $y_{r,k}$ y volver al **PASO 1**

Construcción de la nueva tabla

- Sustituir $x_{B,r}$ por x_k y $c_{B,r}$ por c_k.
- La fila r de la nueva tabla se obtiene dividiendo la fila r de la antigua tabla por $y_{r,k}$ (pivote).
- La columna k de la nueva tabla es toda de ceros, salvo la posición (r, k) en la que hay 1.

- $e_{m,n}$ elemento (m,n) de la tabla antigua.
- $factor(i,j) := (e_{r,j} * e_{j,k})/y_{r,k}, (i \neq r)$.
- $y_{i,j} := y_{i,j} - factor(i,j)$.
- $z_j - c_j := (z_j - c_j) - factor(i,j)$.
- $z := z - factor(i,j)$.
- $x_{B_i} := x_{B_i} - factor(i,j)$

Siguiendo con nuestro ejemplo:

- **PASO 1:** $z_1 - c_1 = -3$; $z_2 - c_2 = -2$ **implica**
- **PASO 3:**
$$y_1 = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \qquad y_2 = \begin{pmatrix} 3 \\ 1 \end{pmatrix} \textbf{ implica}$$
- **PASO 4:**
$\min\{z_1 - c_j 1, z_2 - c_2\}$ = -3, luego $k = 1, X1$ variable de **entrada**
$\min\{x_{B_1}/y_{1,1}; x_{B_2}/y_{2,1}\} = \min\{\ 12/2, 8/2\ \} = 4$, luego $r = 2, X4$ variable de **salida**.
Así pues $y_{2,1} = 2$ es el elemento pivote.

Así pues, la nueva (fila 2):= (fila 2)/pivote = (fila 2)/2 y la columna es k = 1. Además, $y_{1,2} = 3 - ((1*2)/2) = 2$; $y_{1,3} = 1 - ((0*2)/2 = 1$; $y_{1,4} = 0 - ((1*2)/2) = -1$ y los nuevo valores para $x_{B_1} = 12 - ((8*2)/2) = 4$ y $z = 0 - ((8*(-3))/2) = 12$, las diferencias son pues $z_2 - c_2 = -2 - ((1*(-3))/2) = -1$; $z_3 - c_3 = 0 - ((0*(-3))/2) = 0$; $z_4 - c_4 = 0 - ((1*(-3))/2) = 3/2$.

La nueva tabla es:

	c_j	3	2	0	0	
c_B	V. Básicas	x_1	x_2	x_3	x_4	x_B
0	x_3	0	2	1	-1	4
3	x_1	2/2	1/2	0/2	1/2	8/2
		0	$-1/2$	0	3/2	12

Para la nueva tabla volver al **Paso 1** significa:

- **PASO 1:** $z_2 - c_2 = -1/2$ implica
- **PASO 3:**
$$y_2 = \begin{pmatrix} 2 \\ 1/2 \end{pmatrix} \textbf{ implica}$$
- **PASO 4:**
$\min\{z_2 - c_2\}$ = -1/2 , luego $k = 2, X2$ variable de **entrada**
$\min\{x_{B_1}/y_{1,2}; x_{B_2}/y_{2,2}\} = \min\{4/2, 4/(1/2)\} = 2$, luego $r = 1, X3$ variable de **salida**. Así pues $y_{1,2} = 2$ es el elemento pivote.

Así pues, la nueva (fila 1):= (fila 1)/pivote = (fila 1)/2 y la columna es k = 2. Además, $y_{2,1} = 1 - ((0 * 1/2)/2) = 1$; $y_{2,3} = 0 - ((1 * (1/2))/2 = -1/4$; $y_{2,4} = 1/2 - (((-1) * (1/2))/2) = 3/4$ y los nuevo valores para $x_{B_2} = 4 - ((4 * (1/2))/2) = 3$ y $z = 12 - ((4 * (-1/2))/2) = 13$, las diferencias son pues $z_1 - c_1 = 0 - ((0 * (-1/2))/2) = 0$; $z_3 - c_3 = 0 - ((1 * (-1/2))/2) = 1/4$; $z_4 - c_4 = (3/2) - (((-1) * (-1/2))/2) = 5/4$.

La nueva tabla es:

	c_j	3	2	0	0	
c_B	V. Básicas	x_1	x_2	x_3	x_4	x_B
2	x_2	0/2	2/2	1/2	−1/2	4/2
3	x_1	1	0	−1/4	3/4	3
		0	0	1/4	5/4	13

Volver al **PASO 1** significa para la nueva tabla que todos los $z_j - c_j > 0$, lo cual implica **PASO 2**, pero como no hay variables artificiales en la base, la solución $X1 = 3; X2 = 2; X3 = X4 = 0$ es **óptima** con **valor óptimo** $Z = 13$.

Ideas intuitivas sobre el método de las dos fases que se utiliza como fundamento teórico del resolutor de inecuaciones:

- **FASE I:** Se intenta determinar la solución básica factible a partir de la minimización de la **función objetivo artificial**.
  Esta función se obtiene como suma de todas las variables artificiales consideradas.
  - **Si** Función objetivo artificial = 0
  - **entonces** tenemos una solución inicial para el problema original y pasamos a la **FASE II**.
  - **sino** problema infactible.

- **FASE II:**
  - **Si** la **FASE I** devuelve una base sin variables artificiales
  - **entonces** aplicamos el método del simplex al problema original utilizando como solución básica factible la obtenida al final de la **FASE I**.
  - **sino** aplicamos el método del simplex al problema original con las modificaciones siguientes:
    * Función objetivo modificada con las variables artificiales que queden en la base afectadas de coeficientes nulos.
    * Eliminación de la columna de variables artificiales que no pertenezcan a la base final de la **FASE I**.

Algoritmo del simplex en dos fases:

1. **PASO 0:** Poner el problema en forma stardard de maximización con variables artificiales.

2. **FASE I:**

   - **PASO 1** Construir una nueva función objetivo Z^0 cambiando los coeficientes de la función objetivo del paso 0, poniendo 0 a las variables no artificiales y (-1) a las artificiales.
   - **PASO 2** Aplicar el método del simplex al problema del paso 1. El proceso termina cuando *o bién* $Z^0 = 0$ *o bién* todos los $z_j - c_j \geq 0$.
     Si no hay variables artificiales en la base con valor positivo
     entonces PASO 3
     sino problema infactible.

3. **FASE II:**

   - **PASO 3** Consideramos la función Z y si hay variables artificiales en la base del paso 2, ponerles coeficientes 0 y eliminar todas las columnas que contengan variables artificiles que no esten en la base del tabla de la fase I.
   - **PASO 4** Construir la tabla inicial de la fase II partiendo de la tabla del paso 3, cambiando c_B y c_j, y recalculando $z_j - c_j$ y Z.
   - **PASO 5 Si** la función objetivo del paso 3 no tiene variables artificiales
     entonces método del simplex
     sino paso 6
   - **PASO 6**Aplicar el simplex con modificaciones:
     Si X_k es la variable de entreda
     entonces considerar las variables artificiales $y_{i,k}$ de la base y si alguna $y_{i,k} < 0$, la tomamos como variable de salida
     sino utilizar la regla de salida del método del simplex.

Obsérvese que, la modificación de la regla de salida se hace para que en la fase II las variables artificiales de la base no tomen valores positivos.

Ejemplo 1: Entramos es la fase II sin variables artificiales en la base.
minimiza `Z = X1 + X2 + 4*X3`
sujeto a

```
X1 + 2*X2 - X3 ≥ 20
3*X1       + X3 = 14
X1, X2, X3 ≥ 0
```

PASO 0: maximiza `Z' = - X1 - X2 - 4*X3 + 0*X4 - X5 - X6`
sujeto a la condición (C)

```
X1 + 2*X2 - X3 - X4 + X5      = 20
3*X1       + X3          + X6 = 14
Xi ≥ 0, X4 variable holgura, X5, X6 variables artificiales
```

FASE I:
PASO 1: maximiza Z^0 `= 0*X1 + 0*X2 + 0*X3 + 0*X4 - X5 - X6`

PASO 2: Construimos la tabla del simplex para este objetivo y las condiciones (C):

	c_j	0	0	0	0	-1	-1	
c_B	V. Básicas	x_1	x_2	x_3	x_4	x_5	x_6	x_B
-1	x_5	1	2	-1	-1	1	0	20
-1	x_6	3	0	1	0	0	1	14
		-4	-2	0	1	0	0	34

Como $\min\{z_j - c_j\}$ = -4, tenemos que $k = 1$ y $X1$ es de **entrada**. Además, $\min\{x_{B_1}/y_{1,1};\ x_{B_2}/y_{1,2}\} = 14/3$, implica que $r = 2$ y $X6$ es de **salida**. Por lo tanto, pivote $y_{2,1} = 3$. La nueva tabla es:

V. Básicas	x_1	x_2	x_3	x_4	x_5	x_6	x_B
x_5	0	2	$-4/3$	-1	1	$-1/3$	$46/3$
x_1	1	0	$1/3$	0	0	$1/3$	$14/3$
	0	-2	$4/3$	1	0	$4/3$	$-46/3$

Como $\min\{z_j - c_j\}$ = -2, tenemos que $k = 2$ y $X2$ es de **entrada**. Además, $\min\{x_{B_1}/y_{2,1}\} = (46/3)/2$, implica que $r = 1$ y $X5$ es de **salida**. Por lo tanto, pivote $y_{1,2} = 2$. La nueva tabla es:

V. Básicas	x_1	x_2	x_3	x_4	x_5	x_6	x_B
x_2	0	1	$-2/3$	$-/21$	$1/2$	$-1/6$	$23/3$
x_1	1	0	$1/3$	0	0	$1/3$	$14/3$
	0	0	0	0	1	1	0

Como $Z^0 = 0$, se termina el proceso y no hay variables artificiales en la base.

FASE II:
PASO 3: Nueva función objetivo: **maximiza** `Z' = - X1 - X2 - 4*X3 + 0*X4`

PASO 4: Creación de la tabla inicial, sin variables artificiales, cambiando c_B y c_j, y recalculando $z_j - c_j$ y Z:

	c_j	-1	-1	-4	0	
	V. Básicas	x_1	x_2	x_3	x_4	x_B
-1	x_2	0	1	$-2/3$	$-/21$	$23/3$
-1	x_1	1	0	$1/3$	0	$14/3$
		0	0	$13/3$	$1/2$	$-37/3$

PASO 5: Todos los $z_j - c_j \geq 0$, luego la solución es óptima
$X1 := 14/3; X2 := 23/3; X3 = X4 = 0; Z = -Z' = 37/3$.

Ejemplo 2: No entramos es la fase II.
maximiza `Z = X1 + 3*X2`
sujeto a

`X1 + X2` $\leq$ `2`
`-X1 + X2` $\leq$ `5`
`2*X1+ X2 = 6`
`X1, X2` ≥ 0

PASO 0: maximiza `Z = X1 + 3*X2`
sujeto a la condición (C)

```
X1 + X2 + X3           = 2
-X1 + X2      + X4      = 5
2*X1 + X2          + X5 = 6
```

`Xi` ≥ 0`, X3, X4 variables holgura, variable artificial`

FASE I:
PASO 1: maximiza Z^0 `= 0*X1 + 0*X2 + 0*X3 + 0*X4 - X5`

PASO 2: Construimos la tabla del simplex para este objetivo y las condiciones (C):

	c_j	0	0	0	0	-1	
c_B	V. Básicas	x_1	x_2	x_3	x_4	x_5	x_B
0	x_3	1	1	1	0	0	2
0	x_4	-1	1	0	1	0	5
-1	x_5	2	1	0	0	1	6
		-2	-1	0	0	0	-6

Como $\min\{z_j - c_j\}$ = -2, tenemos que $k = 1$ y $X1$ es de **entrada**. Además, $\min\{x_{B_1}/y_{1,1};\ x_{B_3}/y_{1,3}\} = 2$, implica que $r = 1$ y $X3$ es de **salida**. Por lo tanto, pivote $y_{1,1} = 1$. La nueva tabla es:

V. Básicas	x_1	x_2	x_3	x_4	x_5	x_B
x_1	1	1	1	0	0	2
x_4	0	2	1	1	0	7
x_5	0	-1	-2	0	1	2
	0	1	2	0	0	-2

Como todas las $z_j - c_j$ son positivas o cero, no admite mejora la solución. Pero además, $Z^0 = -2 \neq 0$, por lo que el problema es **infactible**.

Desde el punto de vista gráfico, el problema consiste en encontrar la intersección de la recta $2 * X1 + X2 = 6$ con la región $(X1 + X2 \leq 2) \cap (-X1 + X2 \leq 5) = \emptyset$ sobre los ejes positivos $(X1, X2 \geq 0)$.

3.3 El sistema de módulos CLP($\mathcal{R}$)

El sistema CLP(R) se ha construido mediante un sistema de módulos con la intención de que las restricciones más sencillas sean resueltas con la máxima prioridad y las más sofisticadas necesiten usar el sietema completo.

3.3.1 El motor de inferencia y la interfaz

El motor de inferencia manipula restricciones sencillas sin hacer uso del resolutor, mantiene las ligaduras implicitas y controla la creación de puntos de elección tanto de tipo normal, como de tipo resolutor.

I. **Unificación**: Denominaremos *R-variables* a las que aparecen en la colección de restricciones manejadas en el resolutor, y *variables* al resto.

T2\T1	Var	R-var	Num	Term	R-term
Var	Unifica				
R-var	T1→T2	Resolver			
Num	T1→ T2	Resolver	¿T1 = T2?		
Term	T1→T2	Error	Error	¿ T1 = T2 ?	
R-term	Interface	Interface	Interface	Error	Interface

Tabla de la unificación en CLP(R)

La unificación entre una variable y un número real se generaliza de la siguiente forma:

- En el motor de inferencia y en la interface (**explícita**)
- En el resolutor de ecuaciones y en el de inecuaciones (**implícita**)

Además, para tener rápido acceso de esta información desde estos módulos es necesario mantener una **estructura de datos global** en la que se recojan estas **ligaduras**. Sin embargo, no es necesario realizar un tipo de ligadura especial entre variables y términos aritméticos, por dos motivos:

1. Las ligaduras cíclicas entre variables que acaban tomando valores numéricos no son necesariamente inconsistentes.
2. Dado el programa

   - cicla(0).
   - cicla(I) :- I > 0, p, cicla(I-1).

   Ejecutamos el objetivo ?- cicla(n), del siguiente modo:
   Además de realizar N-veces el predicado p, tenemos que ligar la variable I1 de la primera iteración al número n, la siguiente variable I2 al n-1 y así sucesivamente, teniendo que comprobar cada vez si la Ij correspondiente es o no 0, con un costo proporcional a n en el mejor de los casosMientras que si dejamos que las ecuaciones las manipule el resolutor, el coste es lineal en n.

II. Sincronización de la vuelta trás del motor de inferencia con el resolutor de restricciones

Sistema con dos clases de puntos de elección:

- Punto de elección del motor de inferencia, que se crea cuando hay **varias alternativas** para la elección de regla de entrada. (Punto de elección **PROLOG**)
- Punto de elección del resolutor, restaura las restricciones que habían sido creadas (**mantenidas**) por puntos de elección usuales.

Como no hay una correspondencia biunívoca entre ambas clases de puntos de elección, y además un punto de elección del resolutor se mantiene invariante durante varios puntos de elección usuales, conviene:

- Reducir los puntos de elección del resolutor debido a su tamaño (relativamente grande) y al gasto considerable que significa generarlos y eliminarlos.
- Generar un punto de elección del resolutor sólo si se encuentran nuevas restricciones que añadir desde el último punto de elección del resolutor generado.

El motor de inferencia mantiene un **contador**, que cuenta el número de puntos de elección usuales creados sin crear su correspondiente punto de elección del resolutor. Para crear un punto de elección del resolutor se realiza el siguiente algoritmo:

- Crear el punto de elección del resolutor.
- Salvar el contador actual en el punto de elección del resolutor generado.
- contador :=0.

Cuando el motor de inferencia genera vuelta atrás, se determina además si hay que dar vuelta atrás al resolutor, aplicando el siguiente algoritmo:
Si contador $>$ 0 **entonces** contador:= contador - 1;
sino vuelta atrás desde el punto de elección del resolutor y restauración del valor del contador guardado en ese punto de elección eliminado
...**continua** la vuelta atrás del motor de inferencia ...

III. **El interface**

Este módulo es llamado por el motor de inferencia cuando una restricción contiene un término aritmético.
Dato: Restricción con expresiones aritméticas.

1. Evaluación de las expresiones aritméticas.
2. **Si** tras 1. la restricción se hace básica **entonces** evaluación de la misma
   sino Si existe una única variable `X` y la restricción resultante es una ecuación
   entonces ligadura implícita **sino** preprocesa la restricción y llama al resolutor.

Ejemplo: Comportamiento del interface.
`Y = 5 + 7, Z = 2*3, X = Y + Z, 2*Z + 3*Y` $\leq$ `7*X` genera
`Y = 12, Z = 6, X = 18, 2*6 + 3*12` $\leq$`7*18`, es decir, `Y = 12, Z = 6, X = 18.`

Redución de expresión básica: `Y = 2 + 3, Z = 2*Y, 2*Z - 3* Y - 6*X + 7 = 0`
genera `Y = 5, Z = 10, 2*10 - 3*5 - 6*X + 7 = 0`, es decir, `Y = 5, Z = 10, 12 - 6*X = 0`. Luego, la redución a ligadura implícita genera: `Y = 5, Z = 10, X = 2`

`Y = 1 + 2*3, Z = -2*Y, 9*X` $\leq$ `4*Y + 3*Z + 4` genera
`Y = 7, Z = -14 , 9*X` $\leq$ `4*7 + 3*(-14) + 4`, es decir, `Y = 7, Z = -14,`
`9*X`$\leq 18 \longleftrightarrow X \leq 2 \longleftrightarrow X' = 2 - X \geq 0$
Luego, la redución a la forma restringida `Y = 7, Z = -14, X' + X = 2, X'` ≥ 0

`X = Y + Z, W = Z - X, 3*X + 2*Y + Z - 3*W + 5 = 6*X + 9*Z` manda las dos primeras restricciones al resolutor sin resolverlas y transforma la tercera en
`3*(Y + Z) + 2*Y + Z - 3*(Z - X) + 5 = 6*(Y + Z) + 9*Z`
Luego el resolutor recibe: `X = Y + Z, W = Z - X, 2*Y - 11*Z + 5 = 0.`

Preprocesar una restricción ya simplificada significa reducirla a una de estas cuatro formas canónicas:

- Inecuación de la forma: Variable > 0.
- Inecuación de la forma: Variable ≥ 0.
- Ecuación con $c_i, d \in \mathbb{N}$, de la forma (basta con 4 variables):
  $c_1 * X1 + c_2 * X2 + c_3 * X3 + c_4 * X4 = D$.
- Ecuación de la forma: Variable = c * Var1 * Var2, que se corresponde con ecuaciones no lineales.

3.3.2 Resolutor de ecuaciones

Este módulo mantiene una colección consistente de ecuaciones lineales. Dada una nueva restricción lineal comprueba su consistencia con las ya almacanadas e intenta resolver el sistema obtenido usando una implementación de matriz-difusa sobre una variante del método de eliminación de Gauss. Pero, este módulo no posee un resolutor normal de ecuaciones debido a que trabaja en un contexto de inecuaciones lineales y a que debe preveer posibles vueltas atrás.

Hay tres formas de invocar este módulo:

1. Via **interface** con ecuaciones lineales.
2. Via **resolutor de inecuaciones** con ecuaciones implícitas.
3. Via **manipulador de restricciones no lineales** al despertar una ecuación.

En la implementación tiene prioridad la rápida respuesta a la consistencia, es decir, el resolutor de restricciones tiene que devolver cierto sii las restricciones consistentes ya almacenadas en el resolutor junto con la nueva restricción de entrada siguen siendo consistentes.

I. La matriz de las ecuaciones (sólo R-variables)

La estructura de datos del resolutor está compuesta por una matriz cuyas filas almacenan la representación de una variable como combinación lineal de variables paramétricas. El conjunto de variables que manda el interface se parte en dos conjuntos disjuntos de variables: las **paramétricas** y las **no paramétricas**. Las ecuaciones almacenadas en el resolutor son de la forma `Xi = bi + ci1 * Ti1 + ci2 * Ti2 + ... + cin * Tin` donde $n \geq 0$, `Xi` es la variable no paramétrica o **sujeto** de la ecuación, `cij, j`$\in\{$`1,...,n`$\}$ son los coeficientes reales y `Tij, j`$\in\{$`1,...,n`$\}$ son variables paramétricas.

II. ¿ Cómo se añade una restricción lineal ?

La forma resuelta paramétrica inicial es $FRP_0 = \emptyset$

Hemos construido hasta FRP_{j-1} y recibimos la ecuación lineal $E_j (j \geq 1)$, hemos de determinar si $(FRP_{j-1} \cap E_j)$ es **consistente** y en caso afirmativo construir FRP_j. Para ello, sea R_j el resultado de sustituir cada una de las variables no paramétricas X que aparecen en FRP_{j-1} en la ecuación E_j.

1. **Si** R_j es de la forma $0 = 0$ **entonces** $FRP_j := FRP_{j-1}$ y devuelve **cierto**.

2. **Si** R_j es de la forma $C = 0$, para $C \neq 0$ **entonces** no hay FRP_j y devuelve **falso**.

3. **Si** R_j contiene variables no paramétricas nuevas en el resolutor `X1, X2,...,XM` $(M \geq 1)$ **entonces** escogemos una de ellas como sujeto de la ecuación R_j y las demas variables pasarán a ser paramétricas y $FRP_j := FRP_{j-1} \cup$ {`la nueva ecuacion`}

   - **Si** $M = 1$ **entonces** `la nueva ecuacion` puede despertar restricciones no lineal dormidas
   - **sino** devolvemos **cierto**

4. **Si** R_j contiene sólo variables paramétricas **entonces** elegimos a T como sujeto de la ecuación R_j siguiendo las siguientes prioridades:

   - **Si** no está en el resolutor de inecuaciones **entonces** se elige primero
   - **sino si** alguna variable está entre las básicas del resolutor de inecuaciones
   - **entonces** se elige primero **sino** se toma alguna

   Sea R' la ecuación resultante que nos permite la eliminación paramétrica, es decir, FRP_j se obtiene sustituyendo en FRP_{j-1} cada aparición de T por la ecuación que la define y añadiendo la ecuación a la matriz de ecuaciones.
   Si debido a la elección de T hemos de exportar la ecuación R' al resolutor de inecuaciones **entonces** devolvemos como valor cierto el que nos devuelva el resolutor de inecuaciones y **sino** devolvemos **cierto**.

Ejemplo 1: $FRP_0 := \emptyset$; `X + 3*Y = 7` (aplicamos (3)) `X = -3*Y + 7` (`X` sujeto)

FRP_1:= {`X:= (-3*Y + 7)`}
`2*X + 6*Y = 14` (construcción de R_1), `2*(-3*Y +7) + 6*Y = 14`, es decir, $0 = 0$

$FRP_2 := FRP_1$
`X + 9*Y + Z = 12` (construcción de R_2), `(-3*Y + 7) + 9*Y + Z = 12`, o bien,
`6*Y + Z = 5` (aplicamos (3)) `Z = -6*Y + 5` (`Y` ya era paramétrica)

FRP_3:= `X:= (-3*Y + 7), Z:= (-6*Y + 5)`}
`3*X + 2*Z = 16` (construcción de R_3), `3*(-3*Y +7) + 2*(-6*Y + 5) = 16`, es decir,
`Y = 5/7` (aplicamos (4)) `R' : Y = 5/7`, pues no hay otra elección de variable posible.

FRP_4:= {`X:= 4'85, Y:= 0'71, Z:= 0'71`}

Hemos supuesto que no habia ninguna variable afectada ni por el resolutor de inecuaciones, ni por el manipulador de restricciones dormidas.

Ejemplo 2: $FRP_0 := \emptyset$;

`X = Y + Z` tiene una variable no paramétrica `X` y dos variables paramátricas `Y, Z`.

FRP_1:= {`X:= (Y + Z)`}
`W = Z - X` (construcción de R_1); `W = Z - (Y + Z) = - Y`, (`W` nueva variable no paramétrica)

FRP_2:= {`X:= (Y + Z), W:= (- Y)`}
`2*Y - 11*Z = -5`, ya está en forma R_2, pues `Y, Z` son paramétricas.

Si suponemos que ninguna variable está afectada ni por el resolutor de inecuaciones, ni por el manipulador de restricciones dormidas, tomaríamos la de *nombre más joven*, sin embargo en este ejemplo, sería más adecuado despejar `Z` en función de `Y` y dejar `Y` como único parámetro. FRP_3:= {`X:= (5 + 13*Y)/11, W:= -Y, Z:= (5 + 2*Y)/11`}

III. ¿ Cuándo se despiertan las restricciones dormidas ?

Tando en las etapas (3) como (4) del algoritmo anterior se pueden obtener valores concretos para ciertas variables y este hecho es necesario comunicarselo al manipulador de restricciones no lineales. Dicha comunicación debe realizarse tras haber manipulado totalmente la ecuación que causó la valoración de las variables, debido a dos motivos:

- El resolutor de ecuaciones utiliza estructuras de datos **globales**.
- El resolutor de ecuaciones y el manipulador de restricciones no lineales **son mutuamente recursivos**

En muchos casos, una respuesta negativa del resolutor de ecuaciones puede estar producida por el manipulador de restricciones no lineales.

IV. De sobre las estructuras de datos y su relación con la vuelta atrás

- La forma resuelta paramétrica se representa como una lista de listas enlazadas.
- Cada variable paramétrica debe llevar la lista de sus apariciones en la matriz global, lo cual dá lugar a una tabla de referencias cruzadas.
- Necesitamos una pila de seguimiento que nos permita restaurar el estado anterior de las formas paramátricas cuando realizamos la vuelta atrás.

Sustituciones paramétricas: Cuando una variable paramétrica pasa a ser sujeto de una ecuación, hemos de acceder con ayuda de su tabla de referencias cruzadas a las posiciones donde aparezca y realizar la correspondiente sustitución. Esto supone un costo proporcional a la longitud de su tabla de referencias cruzadas.

La elección de la variable paramétrica en (4), hace que la longitud de la matriz global no aumente de no ser necesario, ya que la ecuación que define a la nueva variable sujeto tiene que ser añadida a la matriz global tras realizar todas las sustituciones.

Para la vuelta atrás de los puntos de elección en el resolutor se ha optado por la copia entera de la matriz global debido a dos causas:

- El tamaño no demasiado grande de las ecuaciones.
- El gran costo en tiempo de la reconstrucción de las tablas de referencias cruzadas anteriores al punto de elección.

3.3.3 Resolutor de inecuaciones

Es una adaptación del método del simplex en dos fases con las modificaciones siguientes:

- Manipula variables no restringidas o variables holgura.
- Manipula ecuaciones estrictas.
- Admite como entrada tanto ecuaciones como inecuaciones.
- Detecta ecuaciones implícitas.

Las entradas que recibe este módulo son: Inecuaciones lineales del interface y ecuaciones lineales del resolutor de ecuaciones.

El trabajo del resolutor de inecuaciones es el siguiente: Dada una restricción de entrada,

- **Si** es inconsistente con las restricciones almacenadas
- **entonces** devolver falso
- **sino** mezcla las restricciones de la matriz global, determina su conjunto de ecuaciones implícitas y manda las ecuaciones al resolutor de ecuaciones.

I. La matriz de inecuaciones

Todas las restricciones se representan en una matriz global de forma que cada fila (f) de la matriz contiene una ecuación de alguna de estas dos formas:

$$\texttt{X + Y - Z} \leq \texttt{3} \longleftrightarrow \texttt{X + Y - Z + S1 = 3}$$
$$\texttt{X + Y - Z} > \texttt{4} \longleftrightarrow \texttt{X + Y - Z - S2 = 4 + eps}$$

siendo `eps` es un valor simbólico que denota que proviene de una inecuación estricta. Las variables de la matriz son de dos tipos, no restringidas (`X`$\in \mathbb{R}$) y holgura (`S`$\in \mathbb{R}^+$)

	X	Y	Z	S1	S2	Const	Simb
Fila1	1	1	-1	1	0	3	0
Fila2	1	0	0	0	-1	4	1

Las primitivas para acceder a la información son: `Coef(X,f):= Coeficiente de X de la fila f; valor(f):= (C, D) = (termino independiente, valor simbólico); const(f):= C; eps(f):= D.`

Definimos un orden entre las filas: `(C,D)` $<$ `(0,0) sii (C` $<$ `0) o (C = 0, D`$<$ `0).`

Decimos que una matriz de n ecuaciones está en forma resuelta sii existen n columnas distintas (columnas básicas) tales que:

- (R1) Cada columna **Const** contiene únicamente numeros > 0.
- (R2) Cada columna **Simb** contiene únicamente numeros > 0, si su correspondiente **Const** es 0.
- (R3) Cada **columna básica** de variable `X` tiene exáctamente **una entrada no nula**.
- (R4) Cada **columna básica** de variable `S` tiene exáctamente **una entrada** no nula, es decir, **positiva**.
- (R5) Una fila (ecuación) que contiene **una S básica** sólo contiene **S-variables**.

La matriz del simplex se ha aumentado con:

- Variables no restringidas, holgura y valores simbólicos que denotan infinitésimos.
- Las variables no restringidas básicas tienen coeficientes positivos o negativos.
- Se prioriza para ser básicas a las variables no restringidas frente a las holgura.

II. ¿ Cómo se añade una nueva restricción ?

Los datos de entrada son de tres clases:

1. Una inecuación lineal estricta o no estricta.
2. Una ecuación lineal cuyas variables están en el resolutor de inecuaciones como **no básicas**.
3. Una ecuación lineal cuyas variables están en el resolutor de inecuaciones y al menos `X` es **básica**.

En el primer caso, hacemos **una copia de la restricción** para usarla después, si es necesario, en el algoritmo de las ecuaciones implícitas. **Sustituimos** en el dato de entrada todas las **variables no paramétricas** por sus correspondientes ecuaciones, quedando sólo variables paramétricas en el dato de entrada. En los casos 1. y 3. **sustituimos todas las variables básicas** que aparecen en el dato de entrada, salvo la variable `X` del caso 3. Tras este preprocesado, pasamos a estudiar cada caso por separado:

PRIMER CASO: Una inecuación lineal estricta o no estricta

Pasamos de `C1*Y1 + ...+ CN*YN` $\geq$ `C` a `C1*Y1 + ...+ CN*YN - S = C`
Pasamos de `C1*Y1 + ...+ CN*YN` $>$ `C` a `C1*Y1 + ...+ CN*YN - S = C + 1*EPS`
donde todas las `Yi` son variables **no básicas**, debido al preprocesado.
Si $C < 0$ *normalizamos* la ecuación multiplicando por (-1), y la ecuación normalizada la añadimos como última fila de la matriz.

(1.1.) **Si** la ecuación contiene una variable nueva no restringida
entonces la tomamos como básica en la última fila de la matriz y devolvemos **cierto**.

(1.2.) **Si** la ecuación sólo contiene S-variables, si **Y** es nueva con coeficiente positivo
entonces la tomamos como básica en la última fila de la matriz y devolvemos **cierto**
sino llamamos al algoritmo principal.

SEGUNDO CASO: Una ecuación lineal con variables no básicas

Sea `X = C1*X1 + C2*X2 + ....+ CN*XN + C` la ecuación lineal con todas sus variables no básicas, reemplazamos todas las apariciones de `X` en la matriz por esta ecuación. **Si** debido a esta sustitución algunas de las ecuaciones (f) de la matriz toman `valor(f)`$< (0,0)$ **entonces** las multiplicamos (-1) y devolvemos **cierto**. Obsérvese que la matriz así obtenida está en forma resuelta pues se verifican las condiciones `(R1) - (R5)`.

TERCER CASO: Una ecuación lineal con alguna variable básica

Escribimos la ecuación de entrada usando una variable básica `X` como sujeto, que por ser básica aparece sólo en una fila y s.p.g. suponemos que es la última. Por lo tanto, usamos la ecuación `X = C1*X1 + C2*X2 +...+ CN*XN + C` para sustituir `X` por su valor en la última fila, obteniendo una nueva ecuación (f). Si `valor(f)`$< (0,0)$ normalizamos la ecuación multiplicando por (-1) y estudiamos los tres casos posibles:
Si f tiene una variable nueva no restringida
entonces la tomamos como básica y devolvemos **cierto sino**

- **Si** `valor(f)`$> (0,0)$ **entonces** llamamos al algoritmo principal
- **sino Si** `valor(f)` $= (0,0)$ **entonces** la ecuación es de forma `expresión = 0 + 0*EPS`, que tiene solución trivial `Yi` $= 0$, p.t. `Yi` de la `expresión`.
  Para determinar si f produce ecuaciones implícitas, mandamos f2:= f*(-1) y f al algoritmo de ecuaciones implícitas **Si** este resuelve **entonces** devolvemos **cierto**.

Obsérvese que el tamaño de la matriz sólo crece cuando la entrada es una inecuación, en los otros casos manipula ecuaciones ya existentes, pero no aumenta de tamaño.

ALGORITMO PRINCIPAL

ENTRADA: Matriz con $N \geq 0$ filas (ecuaciones) cumpliendo que la **submatriz** formada por las $N-1$ primeras ecuaciones está en **forma resuelta** y **no** contiene **ecuaciones implícitas**, es decir, que sus restricciones no implican `X = 0` para ninguna `X` de la submatriz. La **N-esima ecuación** contiene *variables no básicas* para la submatriz.

La **salida** del algoritmo principal es de tres tipos:

- **Irresoluble** La nueva restricción es inconsistente con las ya almacenadas.
- **Solución sin ecuaciones implícitas** La nueva restricción es consistente con las ya almacenadas y la nueva matriz en forma resuelta no implica ninguna ecuación.
- **Solución con ecuaciones implícitas** La nueva restricción es consistente con las ya almacenadas y la nueva matriz en forma resuelta implica algunas ecuaciones.

Como nuestra aplicación del método del simplex no tiene función objetivo, las reglas de variable de entrada y salida se simplifican considerablemente.

Descripción de los tres procesos usados:

P1: Selección de variable básica `X` de entrada (columna)
Si en la fila N hay variables no restringidas **entonces** elegimos la más reciente **sino** **Si** todas son S-variables **entonces** elegimos la más joven de coeficiente positivo **sino** devolvemos **falso** (inconsistencia).

P2: Selección de variable básica `X` de salida (fila)
Si la variable `X` no es una S-variable **entonces** f es la fila N-ésima **sino** `f:=` f_S, para `S` la variable más joven de `U:=` $\{f_S|$ `S básica y valor-unit(X,`f_S`) mínimo`$\}$ siendo `valor-unit(X,f):= (Const(f)/Coef(X,f); Eps(f)/Coef(X,f))`

P3: Procedimiento que garantiza que la única fila de la matriz que contiene a `X` sea `f`.
pivotización(X,f): Consiste en restar de cada fila `f'` que contenga a `X` un múltiplo apropiado de la fila `f`.

Algoritmo para encontrar una variable básica para la última fila de la matriz.

REPETIR
P1: Selección de variable básica `X` de entrada (columna)
Si **P1** no encuentra ninguna `X` **entonces**

- **Si** `valor(fila` N`) = (0,0)` **entonces** solución ecuaciones implícitas
- **sino** **irresoluble**

sino **P2**: Selección de variable básica `X` de salida (fila) **pivotización(X,f)**
HASTA QUE f sea la última fila.

Como la solución no genera ecuaciones implícitas, resulta sencillo comprobar que este algoritmo termina siempre. Coste del algoritmo

- Para que f sea la última fila se necesita un coste exponencial. Pero en la mayoría de los casos es 0 o en todo caso un número pequeño.
- La búsqueda de la fila f conocida la columna `X` depende del número de filas que contengan `X`, pero en ellas sólo consultamos `Coef, Const` y `Eps`.
- La pivotización tiene un coste proporcional al número de filas con `X` y al número de entradas no nulas en cada una de ellas.

Algoritmo de ecuaciones implícitas: Este algoritmo genera todas las ecuaciones implícitas de la matriz entrada y devuelve una matriz equivalente sin ecuaciones implícitas. Obsérvese que aunque podemos necesitar invocar al algoritmo principal, su uso dentro de este algoritmo no puede implicar inconsistencia, ya que nuestra matriz de entrada es consistente. El proceso se invoca con el tablero en forma resuelta desde el algoritmo principal o si la ecuación de entrada causa `valor(f) = (0,0)`, por lo tanto, la última fila está en forma eliminable, es decir, es de la forma `C1*S1 + C2*S2 + ....+ CN*SN = 0`, p.t. `i`$\in$`{1,...,n}, Ci` $<$ `0` y `Si` son S-variables.

Definiciones:

- Una variable `X` es **nula** si la matriz implica `X = 0`.
- Todas las variables de una ecuación **eliminable** son nulas.
- `ecuación-implicita(s):= ecuación asociada a la variable nula s.`
  Se obtiene convirtiendo en ecuación a la inecuación de entrada en la que se generó `s` y que se guardó al comienzo del proceso para usarla aquí.
- `filas-relevantes(S)` es la submatriz formada por todas las filas f que verifican: `valor(f) = (0,0)`, es decir, son ecuaciones homogéneas.
  Todas las variables básicas de f son S-variables.

Descripción del algoritmo

Sea `S:=` {`s` | `S-variable nula de la última fila`}
Eliminamos la última fila de la matriz y además `EI:=` $\emptyset$
REPETIR

- Elegimos `s` $\in$ `S; EI:= EI` $\cup$ {`ecuación-implicita(s)`}; `S:= S` $\setminus$ { `s` }
- **Si** existe `f` $\in$ `filas-relevantes(S)` | `Coef(f,s)` $<$ `0` **entonces**
  - Tomamos `s` como sujeto de f: `s:= ECUACION` aplicamos esta sustitución a todas las filas y eliminamos `s` de f; `algoritmo principal(filas-relevantes(S))`
  - **Si** solución con ecuaciones implícitas **entonces** eliminamos f de la matriz y `S:= S` $\cup$ {`todas las variables de f`}

HASTA QUE `S` $= \emptyset$ Hasta aquí manipulamos la parte homogénea de la matriz.

Para cada `Y` $\in$ {`variables básicas de la submatriz homogénea`}
Sustituir `Y` en la submatriz no homogénea
Salida `EI`

Estructuras de datos utilizadas y su relación con la vuelta atrás

Una matriz dispersa se representa eficientemente con dos conjuntos de listas enlazadas, una para la representación de la matriz global y otra de referencias cruzadas de las variables de las filas de la matriz.

Identificadores de variables: Para las variables del resolutor de ecuaciones e inecuaciones usamos el mismo número, pero para las s-variables usamos números nuevos.

Encontrar una variable básica de entrada: Al almacenar cada ecuación colocamos las variables no restringidas primero y por edades y detrás las s-variables también por edades.

Encontrar una variable básica de salida: La matriz de referencias cruzadas nos permite acceder a la lista de todas las ecuaciones en las que aparezca la variable de entrada.

Pivotar: Sea `L` la lista de ecuaciones con `X`, para cada ecuación `f'` $\in$ `L`, hemos de realizar `suma-fila(f,f',c)`, para cada variable `Y` $\in$ `f'`, `Y` $\neq$ `X`, con la correspondiente modificación de las listas de referencias cruzadas.

Vuelta atrás: Como el número de variables en cada ecuación es pequeño, se ha optado por copiar enteras las filas que han sido modificadas después del último punto de elección del resolutor.

Resolutor de inecuaciones

La entrada es una matriz `T` y una restricción `C`

Si `C` es una inecuación **entonces** sustituir las variables básicas de `C`, transformarla con `S, Eps` y añadir esta nueva `C` en la última fila de `T`.
Si son aplicables **(1.1) o (1.2) entonces** se aplican y devolvemos **cierto**
sino `resolver(T)`

Si `C` es de la forma `X = Exp` y todas sus variables son no básicas
entonces sustituimos `X` por `Exp` en la matriz y devolvemos **cierto**

Si `C` es de la forma `X = Exp` y tiene alguna variables básicas **entonces** sustituimos las variables básicas en `Exp`, llamamos `f:= única fila con X` y sustituimos `X en f usando C` y normalizamos
Si es aplicable **(1.1) entonces** se aplica y devolvemos **cierto**
sino
Si `f` es de la forma `Exp = 0` **entonces** `f2:= (-1)*f; T2 := T \f` $\cup$ `f2`
    Si `resolver(T)` dá **irresoluble entonces** devolvemos **irresoluble**
    sino `resolver(T2)`
sino `resolver(T)`

El único procedimiento que falta es `resolver(T)`
`algoritmo principal(T);`
Si irresoluble entonces devolvemos **irresoluble**
sino
Si la última fila de `T` es no eliminable **entonces Solución**
sino `ecuaciones-implicitas(T,EI)`
Para cada `E` $\in$ `EI` **hacer** `resolutor-ecuaciones(E)`
Si irresoluble entonces devolvemos **irresoluble**
Solución

Vemos por la descripción completa del algoritmo que las ecuaciones implicitas sólo se manejan si son necesarias. Las restricciones más sencillas y frecuentes son las que primero se manejan y se demuestra que todos los algoritmos paran y detectan inconsistencia si ésta se presenta.

Ejemplos de resolución de objetivos

`Ejemplo(1, [X1, X2], Z):- X1` $\leq$ `50, X2` $\leq$ `200, X1 + 0.2*X2`$\leq$`72,`
`150*X1+25*X2` $\leq$ `10000, Z = 250*X1 + 45*X2, maximize(Z).`

La ejecución se realiza de la siguiente forma: Las restricciones `X1` $\leq$ `50, X2` $\leq$ `200` que están ya en forma canónica se almacenan en la matriz como:

X1	X2	S1	S2	COEF	EPS
1	0	1	0	50	0
0	1	0	1	200	0

En `X1 + 0.2*X2` $\leq$ `72` sustituimos `X1` y `X2` por su valor y por no verificarse **(1.2)**, se genera una nueva fila `S1 + 0.2*S2 - S3 = 18 + 0*EPS`, quedando la matriz como:

X1	X2	S1	S2	S3	COEF	EPS
1	0	1	0	0	50	0
0	1	-5	0	5	110	0
0	0	1	0.2	-1	18	0

En `150*X1 + 25*X2` $\leq$ `10.000` sustituimos `X1` y `X2` por su valor y por no verificarse **(1.2)**, se genera la fila `25*S1 + 125*S3 - S4 = 250 + 0*EPS`, luego la matriz es:

X1	X2	S1	S2	S3	S4	COEF	EPS
1	0	1	0	0	0	50	0
0	1	-6	0	0	1/25	100	0
0	0	6	1	0	-1/25	100	0
0	0	25	0	125	-1	250	0

En la ecuación `Z` sustituimos `X1` y `X2` por su valor, obteniendo `Z = 270*S1 - 9*S4 + 17000`, así pues el máximo se alcanza para `S4 = 0`, luego los valores máximos son `X1 = 40, X2 = 160, Z = 17200`.

```
Ejemplo(2, [X, Y], Z):- 2*X + Y ≤16, X + 2*Y ≤11,
                X + 3*Y ≤15, Z = 30*X + 50*Y, maximize(Z).
```

La ejecución es: La restricción `2*X + Y ≤ 16` se almacena en la matriz como:

X	Y	S1	COEF	EPS
2	1	1	16	0

En `X + 2*Y ≤11` sustituimos `X` por su valor y por verificarse **(1.1)**, se genera la fila `3*Y - S1 + S2 = 6 + 0*EPS`, luego la matriz es:

X	Y	S1	S2	COEF	EPS
6	0	4	-1	42	0
0	3	-1	1	6	0

En `X + 3*Y ≤15` sustituimos `X` e `Y` por su valor y por verificarse **(1.2)**, se genera la fila `2*S1 -5*S2 + S3 = 12 + 0*EPS`, luego la matriz es:

X	Y	S1	S2	S3	COEF	EPS
6	0	4	-1	0	42	0
0	3	-1	1	0	6	0
0	0	2	-5	1	12	0

Si en `Z` sustituimos `X` e `Y` por su valor, tenemos `Z = 310 - 10/3*S1 - 35/3*S2`, así pues el máximo se alcanza para `S1 = S2 = 0`, como además `S3 = 12` se hace positiva, los valores que se obtienen como máximos son válidos con respecto a todas las condiciones `X = 7, Y = 2, Z = 310`.

```
Ejemplo(3, [X, Y, Z], Min):- X + 3*Y + 2*Z ≥ 5, 2*X + 2*Y + Z ≥ 2, 4*X -
2*Y + 3*Z ≥ -1,
         X ≥ 0, Y ≥ 0, Z ≥ 0, Min = 6*X + 5*Y + 2*Z, minimize(Min).
```

La ejecución es: La restricción `X + 3*Y + 2*Z ≥ 5` se almacena en la matriz como:

X	Y	Z	S1	COEF	EPS
1	3	2	-1	5	0

En `2*X + 2*Y + Z ≥ 2` sustituimos `X` por su valor y por verificarse **(1.1)**, se genera la fila `4*Y + 3*Z - 2*S1 + S2 = 8 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	COEF	EPS
-4	0	1	-2	3	4	0
0	4	3	-2	1	8	0

En `4*X - 2*Y + 3*Z ≥ -1` sustituimos `X` e `Y` por su valor y por verificarse **(1.1)**, se genera la fila `11*Z - 6*S1 + 7*S2 - S3 = 14 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	S3	COEF	EPS
-44	0	0	-16	26	1	30	0
0	44	0	-1	-10	3	46	0
0	0	11	-6	7	-1	14	0

En `X` $\geq$ `0` sustituimos `X` pero por no verificarse **(1.2)**, se genera la fila `- 16*S1 + 26*S2 + S3 - 44*S4 = 30 + 0*EPS` y la eliminación de `S3` en el resto de las filas de la matriz:

X	Y	Z	S1	S2	S3	S4	COEF	EPS
1	0	0	0	0	0	-1	0	0
0	-1	0	-1	2	0	-3	1	0
0	0	1	-2	3	0	-4	4	0
0	0	1	-16	26	1	-44	30	0

En `Y` $\geq$ `0` sustituimos `Y` pero por no verificarse **(1.2)**, se genera la fila `- S1 + 2*S2 - 3*S4 -S5 = 1 + 0*EPS` y la eliminación de `S2` en el resto de las filas de la matriz:

X	Y	Z	S1	S2	S3	S4	S5	COEF	EPS
1	0	0	0	0	0	-1	0	0	0
0	1	0	0	0	0	0	-1	0	0
0	0	2	-1	0	0	1	3	5	0
0	0	0	-3	0	1	-5	13	17	0
0	0	0	-1	2	0	-3	-1	1	0

En `Z` $\geq$ `0` sustituimos `Z` pero por verificarse **(1.2)**, se genera la fila `- S1 + S4 + 3*S5 + 2*S6 = 1 + 0*EPS` y la nueva matriz es:

X	Y	Z	S1	S2	S3	S4	S5	S6	COEF	EPS
1	0	0	0	0	0	-1	0	0	0	0
0	1	0	0	0	0	0	-1	0	0	0
0	0	2	-1	0	0	1	3	0	5	0
0	0	0	-3	0	1	-5	13	0	17	0
0	0	0	-1	2	0	-3	-1	0	1	0
0	0	0	-1	0	0	1	3	2	5	0

Si en `Min` sustituimos `X, Y` y `Z` por su valor, tenemos `Min = 5 + S1 + 5*S4 + 2*S5`, así pues el mínimo se alcanza para `S1 = S4 = S5 = 0`, de la matriz se obtienen los valores de las otras variables holgura y son positivos `S2 = 1/2, S3 = 17`, la solución es `X = 0, Y = 0, Z = 2.5, Min = 5`.

```
Ejemplo(4, [X, Y, Z], Min):- X ≤ 2, X + Y + 2*Z ≤ 4, 3*Y + 4*Z ≤ 6,
       X ≥ 0, Y ≥ 0, Z ≥ 0, Min = -X - 2*Y - 4*Z, minimize(Min).
```

La ejecución es: La restricción `X` $\leq$ `2` se almacena en la matriz como:

X	Y	Z	S1	COEF	EPS
1	0	0	1	2	0

En `X + Y + 2* Z` $\leq$ `4` sustituimos `X` por su valor y por verificarse **(1.1)**, se genera la fila `Y + 2*Z - S1 + S2 = 2 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	COEF	EPS
1	0	0	1	0	2	0
0	1	2	-1	1	2	0

En `3*Y + 4*Z` $\leq$ `6` sustituimos `Y` por su valor y por verificarse **(1.1)**, se genera la fila `2*Z - 3*S1 + 3*S2 - S3 = 0 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	S3	COEF	EPS
1	0	0	1	0	0	2	0
0	1	0	2	-2	1	2	0
0	0	2	-3	3	-1	0	0

En `X` $\geq$ `0` sustituimos `X` por su valor y por verificarse **(1.2)**, se genera la fila `S1 + S4 = 2 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	S3	S4	COEF	EPS
1	0	0	1	0	0	0	2	0
0	1	0	2	-2	1	0	2	0
0	0	2	-3	3	-1	0	0	0
0	0	0	1	0	0	1	2	0

En `Y` $\geq$ `0` sustituimos `Y` por su valor y por verificarse **(1.2)**, se genera la fila `2*S1 - 2*S2 + S3 + S5 = 2 + 0*EPS`, luego la matriz es:

X	Y	Z	S1	S2	S3	S4	S5	COEF	EPS
1	0	0	1	0	0	0	0	2	0
0	1	0	2	-2	1	0	0	2	0
0	0	2	-3	3	-1	0	0	0	0
0	0	0	1	0	0	1	0	2	0
0	0	0	2	-2	1	0	1	2	0

En `Z` $\geq$ `0` sustituimos `Z` por su valor y por no verificarse **(1.2)**, se genera la fila `3*S1 - 3*S2 + S3 - 2*S6 = 0 + 0*EPS` y se elimina `S3` en el resto de la matriz, luego:

X	Y	Z	S1	S2	S3	S4	S5	S6	COEF	EPS
1	0	0	1	0	0	0	0	0	2	0
0	1	0	-1	1	0	0	0	2	2	0
0	0	1	0	0	0	0	0	-1	0	0
0	0	0	1	0	0	1	0	0	2	0
0	0	0	-1	1	0	0	1	2	2	0
0	0	0	3	-3	1	0	0	-2	0	0

Si en `Min` sustituimos `X, Y` y `Z` por su valor, tenemos `Min = -6 - S1 +2*S2`, así pues el mínimo se alcanza para `S2 = 0`. Como la última fila de la matriz la matriz es eliminable y `S3` no aparece en `Min`, la nueva matriz es:

X	Y	Z	S1	S4	S5	COEF	EPS
1	0	0	1	0	0	2	0
0	1	0	2	0	0	2	0
0	0	1	-1.5	0	0	0	0
0	0	0	1	1	0	2	0
0	0	0	2	0	1	2	0

Como tampoco interviene S5 en la expresión de Min, hacemos su valor 0 y así podemos obtener valores para las variables holgura restantes S1 = 1, S4 = 1, comprobando que son positivas y la solución es X = 1, Y = 0, Z = 1.5, Min = -7

```
Ejemplo(5, [X1, X2, X3, X4, X5, X6, X7], Min):-
     X1 + X3 - X4 + X5 + 2*X6 + X7 ≤6, X2 + X4 - 2*X5 + X6 - 2*X7 ≤4,
     X3 - X4 + 2*X6 + X7 ≤1, 0 ≤ X1, X1 ≤ 6, 0 ≤ X2, X2 ≤ 6, 0 ≤ X3,
     0 ≤ X4, X4 ≤ 4, 0 ≤ X5, X5 ≤ 2, 0 ≤ X6, X6 ≤10, 0 ≤ X7,
    Min = 3*X1 - 4*X2 - 2*X3 + 2*X4 + 14*X5 - 11*X6 + 5*X7, minimize(Min).
```

La ejecución es: La restricción X1 + X3 - X4 + X5 + 2*X6 + X7 ≤ 6 se almacena en la matriz como:

X1	X2	X3	X4	X5	X6	X7	S1	COEF
1	0	1	-1	1	2	1	1	6

X2 + X4 - 2*X5 + X6 - 2*X7 ≤ 4 por verificarse **(1.1)**, se genera la fila X2 + X4 - 2*X5 + X6 - 2*X7 + S2 = 4 + 0*EPS, luego la matriz es:

X1	X2	X3	X4	X5	X6	X7	S1	S2	COEF
1	0	1	-1	1	2	1	1	0	6
0	1	0	1	-2	1	-2	0	1	4

X3 - X4 + 2*X6 + X7 ≤ 1 por verificarse **(1.1)**, se genera la fila X3 - X4 + 2*X6 + X7 + S3 = 1 + 0*EPS y se modifica la primera fila de la matriz:

X1	X2	X3	X4	X5	X6	X7	S1	S2	S3	COEF
1	0	0	0	1	0	0	1	0	-1	5
0	1	0	1	-2	1	-2	0	1	0	4
0	0	1	-1	0	2	1	0	0	1	1

En cada par de restricciones 0≤ Xi ≤ N, sustitutimos el valor de Xi en la matriz e introducidos las nuevas variables holgura que pasan a ser o bién variables básicas o generan modificaciones en la matriz. Tras esas manipulaciones calcularíamos el valor de Min = 3*X1 - 4*X2 - 2*X3 + 2*X4 + 14*X5 - 11*X6 + 5*X7. Como el mínimo viene dado por los menores valores de las variables de coeficiente positivo, para facilitar la exposición supongo que X1 = X4 = X5 = X7 = 0, con lo cual la matriz obtenida es:

X2	X3	X6	S1	S2	S3	COEF
0	0	0	1	0	-1	5
1	0	1	0	1	0	4
0	1	2	0	0	1	1

En `0 ≤ X2 ≤ 6` sustituimos `X2` por su valor y por verificarse **(1.2)** para ambos casos, se generan las filas `X6 + S2 + S4 = 4 + 0*EPS; -X6 - S2 + S5 = 2 + 0*EPS`, luego la matriz es:

X2	X3	X6	S1	S2	S3	S4	S5	COEF
0	0	0	1	0	-1	0	0	5
1	0	1	0	1	0	0	0	4
0	1	2	0	0	1	0	0	1
0	0	1	0	1	0	1	0	4
0	0	-1	0	-1	0	0	1	2

En `X3 ≥ 0` sustituimos `X3` por su valor y por verificarse **(1.2)**, se genera `2*X6 + S3 + S6 = 1 + 0*EPS`, además, en `0 ≤ X6 ≤ 10` en el primer caso se verifica **(1.1)** y sustituimos `X6` por su valor y por verificarse **(1.2)** en el segundo caso, se generan `X6 - S7 = 0 + 0*EPS; S7 + S8 = 10 + 0*EPS`, luego la matriz es:

X2	X3	X6	S1	S2	S3	S4	S5	S6	S7	S8	COEF
0	0	0	1	0	-1	0	0	0	0	0	5
1	0	0	0	1	0	0	0	0	1	0	4
0	1	0	0	0	1	0	0	0	2	0	1
0	0	0	0	1	0	1	0	0	1	0	4
0	0	0	0	-1	0	0	1	0	-1	0	2
0	0	0	0	0	1	0	0	1	2	0	1
0	0	1	0	0	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	0	0	1	1	10

Si en `Min` sustituimos `X2, X3` y `X6` por su valor, tenemos `Min = - 18 + 4*S2 + 2*S3 - 3*S7`, así pues el mínimo se alcanza para `S2 = S3 = 0` y `S7` máximo, luego la nueva matriz es:

X2	X3	X6	S1	S4	S5	S6	S7	S8	COEF
0	0	0	1	0	0	0	0	0	5
1	0	0	0	0	0	0	1	0	4
0	1	0	0	0	0	0	2	0	1
0	0	0	0	1	0	0	1	0	4
0	0	0	0	0	1	0	-1	0	2
0	0	0	0	0	0	1	2	0	1
0	0	1	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	1	1	10

Las ecuaciones de las variables holgura son `S1 = 5, S2 = 0, S3 = 0, S4 = 4 - S7, S5 = 2 + S7, S6 = 1 - 2*S7, S8 = 10 - S7`, es decir, todas están en función de `S7` y han de ser no negativas, así pues el valor más grande de `S7` viene dado por la ecuación más desfavorable `S6 = 0 = 1 - 2*S7`, es decir, `S7 = 0.5`. Por lo tanto, comprobamos que las variables holgura son todas correctas: `S1 = 5, S2 = 0, S3 = 0, S4 = 3.5, S5 = 2.5, S6 = 0, S7 = 0.5, S8 = 9.5` y nuestra solución es: `X1 = X4 = X5 = X7 = 0, X2 = 3.5, X3 = 0, X6 = 0.5, Min = -19.5`

```
Ejemplo(6, [X1, X2, X3, X4, X5, X6, X7], Min):-
     X1 + X3 - X4 + X5 + 2*X6 + X7 ≤ 6, X2 + X4 - 2*X5 + X6 - 2*X7 ≤ 4,
     X3 - X4 + 2*X6 + X7 ≤ 1, 0 ≤ X1, X1 ≤ 6, 0 ≤ X2, X2 ≤ 6,0 ≤ X3,
     0 ≤ X4, X4 ≤ 4, 0 ≤ X5, X5 ≤ 2, 0 ≤ X6, X6 ≤10, 0 ≤ X7,
    Min = -3*X1 + 4*X2 + 2*X3 - 2*X4 - 14*X5 + 11*X6 - 5*X7, minimize(Min).
```

Este ejemplo tiene las mismas restricciones que el anterior, pero la expresión a minimizar es la opuesta, ésto significa que la matriz trás la evaluación de las tres primeras restricciones es:

X1	X2	X3	X4	X5	X6	X7	S1	S2	S3	COEF
1	0	0	0	1	0	0	1	0	-1	5
0	1	0	1	-2	1	-2	0	1	0	4
0	0	1	-1	0	2	1	0	0	1	1

En cada par de restricciones `0≤ Xi ≤ N`, sustitutimos el valor de `Xi` en la matriz e introducidos las nuevas variables holgura que pasan a ser o bién variables básicas o generan modificaciones en la matriz. Tras esas manipulaciones calcularíamos el valor de `Min = -3*X1 + 4*X2 + 2*X3 - 2*X4 - 14*X5 + 11*X6 - 5*X7`. Como el mínimo viene dado por los menores valores de las variables de coeficiente positivo, para facilitar la exposición supongo que `X2 = X3 = X6 = 0`, con lo cual la matriz obtenida es:

X1	X4	X5	X7	S1	S2	S3	COEF
1	0	1	0	1	0	-1	5
0	1	-2	-2	0	1	0	4
0	-1	0	1	0	0	1	1

En `0 ≤ X1 ≤ 6` sustituimos `X1` por su valor y por verificarse **(1.2)** para ambos casos, se generan las filas `X5 + S1 - S3 + S4 = 5 + 0*EPS; -X5 - S1 + S3 + S5 = 1 + 0*EPS`, luego la matriz es:

X1	X4	X5	X7	S1	S2	S3	S4	S5	COEF
1	0	0	0	0	0	0	0	1	10
0	1	0	0	-2	-1	0	0	2	4
0	0	0	1	-2	-1	1	0	2	5
0	0	0	0	0	0	0	1	1	10
0	0	-1	0	-1	0	1	0	1	5

En `0 ≤ X4 ≤ 4` sustituimos `X4` por su valor y por verificarse **(1.2)** para ambos casos, se generan las filas `-2*S1 - S2 + 2*S5 + S6 = 4 + 0*EPS; 2*S1 + S2 - 2*S5 + S7 = 0 + 0*EPS`, luego la matriz es:

X1	X4	X5	X7	S1	S2	S3	S4	S5	S6	S7	COEF
1	0	0	0	0	0	0	0	1	0	0	10
0	1	0	0	-2	-1	0	0	2	0	0	4
0	0	0	1	-2	-1	1	0	2	0	0	5
0	0	0	0	0	0	0	1	1	0	0	10
0	0	-1	0	-1	0	1	0	1	0	0	5
0	0	0	0	-2	-1	0	0	2	1	0	4
0	0	0	0	2	1	0	0	-2	0	1	0

En `0 ≤ X5` sustituimos `X5` por su valor y por no verificarse **(1.2)**, se genera `- S8 + S7 = 0 + 0*EPS` y la eliminación de `S7` del resto de las filas de la matriz. Además, `X5 ≤ 2` verifica **(1.2)** generando `S8 + S9 = 2 + 0*EPS`. En `0 ≤ X7` sustituimos `X7` por su valor y por verificarse **(1.2)** se genera `S2 + S3 - 2*S8 - 2*S10 = 5 + 0*EPS`, haciendo básica a `S6`, lo que nos obliga a eliminarla de las otras filas de la matriz, luego la matriz es:

X1	X4	X5	X7	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	C
1	0	0	0	1	0	0	0	0	0	0	1	0	0	5
0	-1	0	0	0	1	2	0	0	0	0	-2	0	0	6
0	0	0	1	0	0	0	0	0	0	0	0	0	-1	0
0	0	0	0	1	0	-1	1	0	0	0	1	0	0	5
0	0	-1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	2	0	0	-1	0	-2	0	0	6
0	0	0	0	0	1	2	0	0	0	0	-1	0	0	10
0	0	0	0	-1	0	1	0	1	0	0	-1	0	0	5
0	0	0	0	0	0	0	0	0	0	1	-1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	2
0	0	0	0	0	1	1	0	0	0	0	-2	0	1	5

Si en `Min` sustituimos `X1, X4, X5` y `X7` por su valor, tenemos `Min = -3 + 3*S1 - 2*S2 - 4*S3 - 7*S8 - 5*S10`, así pues el mínimo se alcanza para `S1 = 0`. Las ecuaciones de las variables holgura `S7 = S8, S9 = 2 - S8, S6 = 4 - S8`, están en función de `S8` y han de ser no negativas, así pues el valor más grande de `S8` viene dado por la ecuación más desfavorable `S9 = 2 - S8`, es decir, `S8 = 2`. Como `S4 + S5 = 10` y estas variables holgura no están en `Min`, no nos interesamos por sus valores. Sabemos además que `S2 + 2*S3 = 10 + S8 = 12`, luego, `Min = -3 + 3*0 - 2*(S2 + 2*S3) - 7*2 - 5*S10 = -3 - 24 - 14 -5*S10`. Para calcular el mejor valor posible hemos de utilizando la ecución `S10 = 5 + 2*S8 - S2 - S3 = 5 + 4 -(12 - S3) = -3 + S3`, siendo `S2 + 2*S3 = 12`, si tomo `S2 = 0` con lo cual `S3 = 6` y `S10 = 3`. Por lo tanto, comprobamos que las variables holgura son todas correctas: `S1 = S2 = 0, S3 = 6, S4 + S5 = 10, S6 = 2, S7 = 2, S8 = 2, S9 = 0, S10 = 3` y nuestra solución es: `X1 = 3, X2 = X3 = X6 = 0, X4 = 2, X5 = 2, X7 = 3, Min = -56`

3.3.4 Manipulador de restricciones no lineales

Las funciones del módulo este son:

- Almacenar las restricciones no lineales que genera el interface y que son de la forma `X = C *X1 * X2`.
- Recibir todas las ecuaciones básicas de la forma `X = número` que le envía el resolutor de ecuaciones y despertar las restricciones que se vuelvan lineales y enviarlas al resolutor de ecuaciones.
- **Si** después de la manipulación exterior se produce la vuelta atrás, debe reconstruir la pila no lineal anterior al punto de elección.

Lo fundamental en este módulo son las estructuras de datos que maneja y su acoplamiento con la vuelta atrás.

I) La pila no lineal contiene dos clases de nodos:

- **El nodo no lineal** que contiene la representación de la eciación no lineal `X = C *X1 * X2`, junto con las referencias a los nodos hijos `X1` y `X2`.
- **El nodo despertador** que indica cual es la restricción no lineal almacenada en algún nodo anterior de la pila, que ha sido despertada como resultado de una ecuación del resolutor lineal indicando la variable responsable.

Por lo tanto, en cada instante, el conjunto de restricciones no lineales está determinado por los nodos de la pila no lineal que no poseen ningún nodo despertador asociado.

II) Tabla de referencias cruzadas despertadas

`ref-cruz(X)` indica los nodos de la pila no lineal en los que aparece `X`, es decir, la referencia a los nodos de la pila no lineal en los que `X` está en el lado derecho de la ecuación y que no tienen nodo despertador asociado.

III) Las tres operaciones principales son:

A) Añadir una nueva restricción no lineal
Añadir un nodo no lineal `N` con `X = C *X1 * X2`.
Añadir un nodo `R1` a `ref-cruz(X1)` y un nodo `R2` a `ref-cruz(X2)` con `N`.
Añadir las referencias `R1` y `R2` al nodo `N`.

B) Despertar una restricción almacenada
Suponemos que el resolutor de ecuaciones nos envia el dato `X1 = NUMERO`.
Si `ref-cruz(X1)` $= \emptyset$ **entonces** no hacemos nada **sino**
para cada `N` $\in$ `ref-cruz(X1)` **hacer**
    Añadir un nodo despertador `D` a la pila.
    Colocar una referencia a `N` en `D`.
    Eliminar de `N` las dos referencias `R1` y `R2`.
Esta última eliminación se hace para no crear otro nodo despertador si después aparece `X2= NUMERO`.

C) Eliminar un nodo de la pila no lineal debido a una vuelta atrás.
Dato de entrada: **nodo de la pila no lineal**

- **Si** `no-lineal(N)` **entonces** eliminar las `R1` y `R2` guardadas en `N` y el propio `N` de la cima de la pila
- **sino Si** `despertador(W)` **entonces** acceder al nodo no lineal `N` asociado a `W`, insertar nuevo nodo en `ref-cruz(Xi)` p.t. `Xi` $\in$ `N` y eliminar el nodo `W` de la cima de la pila
- **sino ERROR**

E) Módulo de salida

Este módulo sirve para mostrar la salida después de resolver el objetivo y por ello a las variables del objetivo denominadas distinguidas.

Por ejemplo, la restricción `X + Y + Z` $\geq$ `2, X - 5*Y - Z` $\geq$ `2` con variables distinguidas `X, Y` espera que la solución aparezca en la forma `X` $\geq$ `2 + 2*Y`.

No existe un algoritmo eficiente conocido que nos permita representar la proyección de la solución cumpliendo que sólo aparezcan variables distinguidas y que el número de restricciones que se presentan sea mínimo.

Clasificación de las variables

- **Primarias**: Las distinguidas y aritméticas que aparecen en términos no aritméticos.
- **Auxiliares**: Las que se relacionan en una ecuación no lineal bién con una variable primaria, bién con una variable auxiliar.
- **No mostrables**: El resto de las variables.

Dos observaciones importantes:

- No todas las variables auxiliares son mostrables.
- Este módulo no presenta problemas de vuelta atrás por si mismo, como puede pedirse otra solución del objetivo, tras mostrar la generada en este módulo, es necesario copiar el sistema entero y realizar las manipulaciones que describimos a continuación sobre esa copia.

I. Manipulación de ecuaciones lineales

Si `X` aparece en la ecuación `E` definimos `pivota(X,E)` para que realize las manipulaciones siguientes:

- Transforma `E` para que `X` sea variable sujeto .
- Aplica la sustitución obtenida al resto de las ecuaciones e inecuaciones lineales.
- Elimina `E` de la matriz de ecuaciones.

Algoritmo para la matriz de ecuaciones lineales

1. Se eliminan mediante `pivota(X,E)` aquellas ecuaciones cuyo sujeto sea no mostrable.
2. Mientras existan variables no mostrables `X` en alguna ecuación lineal `E` se realiza `pivota(X,E)`.
3. Mientras existan variables auxiliares `Y` en alguna ecuación lineal `E` se realiza `pivota(Y,E)`.

Ejemplo: Sean `X, Y, Z` variables distinguidas, en las siguientes restricciones:
`C1:X = f(a,M); C2:N = 2*T; C3:Y = 4*T; C4:Z = R + T; C5:M = N*R`
Entonces suponemos que: `Prim = {X, Y, Z, M}, Aux = {N, R}, NMos ={T}`

El algoritmo actua de la siguiente manera:
`pivota(T,C2):  T = N/2` sujeto y `NMos` $= \emptyset$, luego:
`C1:  X = f(a,M); C3:  Y = 2*N; C4:  Z = R + 0.5*N; C5:  M = N*R`
`pivota(R,C4):  R = Z - 0.5*N` sujeto y `Aux` $= \{$`N`$\}$, luego:
`C1:  X = f(a,M); C3:  Y = 2*N; C5:  M = N*(Z - 0.5*N)`
`pivota(N,C3):  N = Y/2` sujeto y `Aux` $= \emptyset$, luego:
`C1:  X = f(a,M); C5:  M = (0.5*Y)*(Z - 0.25*Y)`

II. Manipulación de inecuaciones lineales

`c` es redundante con respecto a una colección de restricciones `C` sii `C\{c}` $\vdash$ `C`
Este proceso combina dos aspectos fundamentales:

- La forma más sencilla de redundancia entre inecuaciones es la redundancia paralela, es decir, que dos inecuaciones definan hiperplanos paralelos.

- Eliminación de variables no mostrables.

Algoritmo de la matriz de inecuaciones lineales.

1. Eliminamos las inecuaciones que sólo contienen variables no mostrables.

2. Eliminamos todas las inecuaciones con redundancia paralela respecto de las restricciones almacenadas.

3. Repetimos el proceso anterior hasta que no queden variables eliminables.

Heurística para la elección de la variable pivote entre las variables eliminables. Para cada `Z` definimos: `N+(Z):= número de apariciones positivas de Z` y `N-(Z):= número de apariciones negativas de Z`. La variable pivote se elige de entre las variables eliminables con `[(N+ * N-) - (N+ + N-)]` mínimo. Si `Z` es una variable eliminable elegida así, el siguiente proceso a realizar es `Fourier(Z)`:
Si (`N+(Z) = 0`) **o bién** (`N-(Z) = 0`) **entonces** no tenemos en cuenta las inecuaciones que contienen a `Z`, pues no nos informan de su relación con variables primarias.
Si (`N+(Z) = 1`)**o bién** (`N-(Z) = 1`) **entonces** pivotamos `Z` usando la inecuación positiva o negativa y eliminamos la inecuación.
Si (`N+(Z)` $>$ `1`) **y** (`N-(Z)` $>$ `1`) **entonces** generamos (`N+(Z)*N-(Z)`) inecuaciones a lo más de la siguiente forma:

- Sumamos cada una de las inecuaciones positivas en `Z` con cada una de las inecuaciones negativas en `Z` multiplicadas por un número **adecuado** para que cada inecuación resultante no contenga a `Z`.

- Eliminamos todas las inecuaciones con `Z`.

Como el procedimiento `Fourier(Z)` puede generar inecuaciones redundantes, antes de almacenar una inecuación lineal producida detectamos si es **Kohler**-redundante.

Llamaremos **generador** de una nueva inecuación `I` creada en el proceso `Fourier(Z)` a cualquiera de las inecuaciones que en la iteración del proceso han contribuido a `I`.

Ejemplo:

- **Si** `I1` e `I2` han dado origen a `I4`.
- **Si** `I1` e `I3` han dado origen a `I5`.
- **Si** `I4` e `I5` han dado origen a `I6`.
- **entonces** `generadores(I6) = { I1,I2,I3 }`

Test de Kohler: Si en la n-ésima iteración del proceso `Fourier(Z)` una nueva inecuación tiene n+1 generadores **entonces** es redundante con respecto a la colección de inecuaciones creadas en esa iteración.

Este test no es completo, pero si bastante simple y reduce mucho el número de inecuaciones creadas.

Salida global generada por este módulo:

- Las variables distinguidas no aritméticas se escriben como siempre. Si alguna de ellas no se ha ligado, simplemente se ignora.
- Los términos se escriben como siempre, salvo que alguna de sus variables sea aritmética.
- Si la variable aritmética `X` no tiene un valor inmediato, entonces se escribe sustituyendo `X` por la ecuación que la define.

En el caso de nuestro anterior ejemplo tenemos que `C1: X = f(a,M); C5: M = (0.5*Y)*(Z - 0.25*Y)` se escribe como `X = f(a, (0.5*Y)*(Z - 0.25*Y))`.

En resumen:

- Todas las ecuaciones lineales se escriben.
- Todas las inecuaciones lineales resultantes del proceso `Fourier(Z)` se escriben.
- Todas las ecuaciones no lineales se escriben, aunque no pueda garantizarse su satisfactibilidad.

3.4 Ejemplos en Dominios Finitos, CLP($\mathcal{R}$) y Booleanos

Para terminar con esta presentación de programación lógica con restricciones, comenzamos resolviendo un mismo problema en los tres estilos de programación que hemos estudiado, observando posibles ventajas e inconvenientes. El problema que queremos resolver es

$$\mathbf{S\ E\ N\ D} + \mathbf{M\ O\ R\ E} = \mathbf{M\ O\ N\ E\ Y}$$

En Dominios Finitos la metodología es:

1. Dominio de las variables: `domain([S, E, N, D, M, O, R, Y], 0, 9)`

2. Restricciones que han de cumplir:
```
restrict([S, E, N, D, M, O, R, Y]):- S #> 0, M #> 0,
          all-different([S,E,N,D,M,O,R,Y]), sum([S,E,N,D,M,O,R,Y]).
```
siendo
```
sum([S,E,N,D,M,O,R,Y]) :- 1000*S + 100*E + 10*N + D
                          + 1000*M + 100*O + 10*R + E
                          #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

3. Generación de los valores: `labeling(Type, [S, E, N, D, M, O, R, Y])`

El programa es:
```
send([S,E,N,D,M,O,R,Y], Type):-
                                domain([S, E, N, D, M, O, R, Y], 0, 9),
                                restrict([S, E, N, D, M, O, R, Y]),
                                labeling(Type, [S, E, N, D, M, O, R, Y]).
```

La evaluación del objetivo `?- send([S, E, N, D, M, O, R, Y], [ ])` produce la salida en orden alfanumérico, exáctamente:

```
D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2
```

En CLP($\mathcal{R}$) la metodología es la misma, pero se explicita el acarreo:

1. Dominio de las variables:
```
dominio(S, E, N, D, M, O, R, Y):-
        S > 0, E ≥ 0, N ≥ 0, D ≥ 0, M > 0, O ≥ 0, R ≥ 0, Y ≥ 0,
        S ≤ 9, E ≤ 9, N ≤ 9, D ≤ 9, M ≤ 9, O ≤ 9, R ≤ 9, Y ≤ 9
```

2. Restricciones que han de verificarse:
```
restrict(S, E, N, D, M, O, R, Y) :-
        S #> 0, M #> 0, D + E = Y + 10*C1,C1 + N + R = E + 10*C2,
        C2 + E + O = N + 10*C3, C3 + S + M = O + 10*M,
        carry(C1, C2, C3), Dig(S), dig(E), dig(N), dig(D), dig(M),
         dig(O), dig(R), dig(Y), difflist([S, E, N, D, M, O, R, Y])
```
siendo
```
carry(1,1,1), carry(1,1,0), carry(1,0,1), carry(1,0,0),
carry(0,1,1), carry(0,1,0), carry(0,0,1), carry(0,0,0), dig(9), dig(8),
dig(7), dig(6), dig(5), dig(4), dig(3), dig(2), dig(1), dig(0)
difflist([ ]).
difflist([X|T]) :- no-miembro(X, T), difflist(T).
no-miembro(X, [Y | Z]):- X < Y, no-miembro(X, Z).
no-miembro(X, [Y | Z]):- X > Y, no-miembro(X, Z).
no-miembro(X, [ ]).
```

3. Generación de los valores: No hay.

El programa es:
```
send(S, E, N, D, M, O, R, Y):- dominio(S,E,N,D,M,O,R,Y),
                                restrict(S, E, N, D, M, O, R, Y).
```
El objetivo `?- send(S, E, N, D, M, O, R, Y)` produce la salida en orden de las variables: `S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2`

Sobre el dominio de los Booleanos se resuelve por columnas, los dígitos de cada letra están codificados con 4 bits, el orden del etiquetado influye mucho en la eficiencia.

La solución es: `S = [1,0,0,1], E = [0,1,0,1], N = [0,1,1,0], D = [0,1,1,1], M = [0,0,0,1], O = [0,0,0,0], R = [1,0,0,0], Y = [0,0,1,0]`

1. Dominio de las variables:
```
dominio(A):- A = [S, E, N, D, M, O, R, Y],
            dcb-digit(S), dcb-digit(E), dcb-digit(N), dcb-digit(D),
            dcb-digit(M), dcb-digit(O), dcb-digit(R), dcb-digit(Y),
                          diff0(S), diff0(M), all-dcb-digit-diff(A).
```
siendo `and0(1, 1) = 0` y `or1(0, 0) = 1`.
```
dcb-digit(D):- D = [B3, B2, B1, -], or(B2, B1, B21), and0(B3, B21).
```
Si $\mathbf{B3 = 1}$ entonces $\mathbf{B2 = 0}$ y $\mathbf{B1 = 0}$.
```
diff0([B3, B2, B1, B0]):- or(B3, B2, B32),or(B1, B0, B10),or1(B32, B10).
all-dcb-digit-diff([ ]).
all-dcb-digit-diff([X | L]):-diff-of(L, X), all-dcb-digit-diff(L).
diff-of([ ],-).
diff-of([Y |L], X):- dcb-digit-diff(X, Y), diff-of(L, X).
dcb-digit-diff([X3, X2, X1, X0], [Y3, Y2, Y1, Y0]):-
                    xor(X3, Y3, XY3), xor(X2, Y2, XY2), xor(X1, Y1, XY1),
                    xor(X0, Y0, XY0), diff0([XY3, XY2, XY1, XY0]).
```

2. Restricciones que han de verificarse:
```
restrict(A) :- LC = [C1, C2, C3, C4],
     Z = [0, 0, 0, 0], dcb-add(0, D, E, Y, C1), dcb-add(C1, N, R, E, C2),
     dcb-add(C2,E,O,N,C3), dcb-add(C3,S,M,O,C4), dcb-add(C4,Z,Z,M,0),!.
```
siendo
```
dcb-add(CI,[X3,X2,X1,X0],[Y3,Y2,Y1,Y0],[Z3,Z2,Z1,Z0],CO):-
   fadd(CI,X0,Y0,Z0,C1),fadd(C1,X1,Y1,I1,C2),fadd(C2,X2,Y2,I2,C3),
   fadd(C3,X3,Y3,I3,C4),or(I2,I1,I12),and(I3,I12,I123),or(C4,I123,Hex),
hadd(I1,Hex,Z1,D2),fadd(D2,I2,Hex,Z2,D3),hadd(D3,I3,Z3,D4),or(C4,D4,CO).
fadd(CI,X,Y,Z,CO):- hadd(X,Y,Z1,C1),hadd(CI,Z1,Z,C2),or(C1,C2,CO).
hadd(X,Y,Z,CO):- and(X,Y,CO),xor(X,Y,Z).
```

3. Generación de los valores: `array-labeling(A),labeling(LC).`

El programa es:
```
bsend(A):- domain(A), restrict(A),
                      array-labeling(A), labeling(LC).
```

El objetivo `?- send([S,E,N,D,M,O,R,Y], [ ])` produce la respuesta anterior.

Por último, algunos ejemplos interesantes, programados en: **Dominios Finitos**

Ejemplo 1 Producto de incognitas: Encontrar el valor de los dígitos que verifican el siguiente producto, sabiendo que cada dígito debe aparecer exáctamente dos veces.

		X1	X2	X3
		X4	X5	X6
		X7	X8	X9
	X10	X11	X12	
X13	X14	X15		
X16	X17	X18	X19	X20

Solución:

```
[X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15, X16, X17,
X18, X19, X20] = [1, 7, 9, 2, 2, 4, 7, 1, 6, 3, 5, 8, 3, 5, 8, 4, 0, 0, 9, 6]
```

```
Q:- get-labeling(LAB), mult(LAB,LD),write(LD).

mult(LAB,LD):- LD = [X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12,
  X13, X14, X15, X16, X17, X18, X19, X20], domain(LD,0,9), atmost(2,LD,0),
  atmost(2,LD,1), atmost(2,LD,2), atmost(2,LD,3), atmost(2,LD,4),
  atmost(2,LD,5),atmost(2,LD,6), atmost(2,LD,7), atmost(2,LD,8),
  atmost(2,LD,9), Y = 100*X1 + 10*X2 + X3, Z1 = 100*X7 + 10*X8 + X9,
  Z2 = 100*X10 + 10*X11 + X12, Z3 = 100*X13 + 10*X14 + X15,
  'XY=Z'(X6,Y,Z1), 'XY=Z'(X5,Y,Z2), 'XY=Z'(X4,Y,Z3), 100*X7 + 10*X8 + X9
   + 1000*X10 + 100*X11 + 10*X12 + 10000*X13 + 1000*X14 + 100*X15
  = 10000*X16 + 1000*X17 + 100*X18 + 10*X19 + X20, lab(LAB,LD).
```

Ejemplo 2: Puzzle alfabetico aleatorio. Los números del 1 al 26 han sido asignados aleatoriamente a las letras del alfebeto. Los números que aparecen a la derecha de las palabras son la suma total de los valores de las letras. Encontrar el valor de cada letra con ayuda de las siguientes ecuaciones.

BALLET	45	GLEE	66	POLKA	59	SONG	61
CELLO	43	JAZZ	58	QUARTET	50	SOPRANO	82
CONCERT	74	LYRE	47	SAXOPHONE	134	THEME	72
FLUTE	30	OBOE	53	SCALE	51	VIOLIN	100
FUGUE	50	OPERA	65	SOLO	37	WALTZ	34

Solución: [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z] =
[5,13,9,16,20,4,24,21,25,17,23,2,8,12,10,19,7,11,15,3,1,26,6,22,14,18]

```
Q:- get-labeling(LAB), alpha(LAB,LD),write(LD).
alpha(LAB,LD):- fd-vector-max(26), LD = [A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,
  P,Q,R,S,T,U,V,W,X,Y,Z], all-different(LD), domain(LD,1,16),
  B + A + L + L + E + T = 45, C + E + L + L + O = 43, S + O + L + O = 37,
  C + O + N + C + E + R + T = 74, F + L + U + T + E = 30,
  F + U + G + U + E = 50, G + L + E + E = 66, J + A + Z + Z = 58,
  L + Y + R + E = 47, O + B + O + E = 53, O + P + E + R + A = 65,
  P + O + L + K + A = 59, Q + U + A + R + T + E + T = 50, S+O+N+G=61,
  S + C + A + L + E = 51, S + A + X + O + P + H + O + N + E = 134,
  S + O + P + R + A + N + O = 82,T + H + E + M + E = 72,
  V + I + O + L + I + N = 100, W + A + L + T + Z = 34, lab(LAB,LD).

lab(normal, L):- labeling(L).
lab(ff, L):- labelingff(L).
get-labeling(LAB):- argc(C), get-labeling1(C, LAB).
get-labeling1(1, normal).
get-labeling1(2, Lab):- argv(1, Lab).
```

Booleanos

Ejemplo 1 Principio del palomar: Cómo se pueden colocar n palomas en m palomares, sabiendo que sólo hay solución si n $\leq$ m. Nuestra solución es una lista de la forma `[[p11,...,p1m],...,[pn1,...,pnm]]` tal que `pij = 1` si la paloma `i` está en el palomar `j`. Para `n = 2` y `m = 3` tenemos `[[0,0,1], [0,1,0]], [[0,0,1], [1,0,0]], [[0,1,0],[0,0,1]], [[0,1,0],[1,0,0]], [[1,0,0],[0,0,1]], [[1,0,0],[0,1,0]]`. Dicha solución hay que leerla como: La primera paloma está en el tercer palomar, la segunda en el segundo, etc.

```
Q:- read-integer(N,M), bpalomar(N,M,A), write(A).

bpalomar(N,M,A):- create-array(N,M,A), for-each-line(A, only1),
            for-each-column(A, atmost1), !, array-labeling(A).
```

Ejemplo 2 Colorear los enteros `1,2,...,n` con ayuda de 3 colores de forma que no exista ningún triple `(X, Y, Z)` monocolor siendo `X + Y = Z`. Sabemos que existe solución sii `n` $\leq$ `13`. La solución es una lista `[[ent11, ent12, ent13],..., [entn1, entn2, entn3]]`, donde `entij = 1` si el entero `i` tiene el color `j`.

Solución para `n = 4`: `[[0,0,1], [0,1,0], [0,0,1], [1,0,0]], [[0,0,1], [0,1,0], [0,1,0], [0,0,1]]` .....

La solución hay que leerla como: El número 1 color 1, el numero 2 color 2, el número 3 color 3, el número 4 color 1, siendo `(1,2,3)`, `(1,2,4)`, `(1,3,4)`, `(2,3,4)` triples no monocromáticos, etc.

La primera solución para `n = 13` es: `[[0,0,1], [0,1,0], [0,1,0], [0,0,1], [1,0,0], [1,0,0], [0,0,1], [1,0,0], [1,0,0], [0,0,1], [0,1,0], [0,1,0], [0,0,1]]`

```
Q:- read-integer(N), colorear(N,A), write(A).

colorear(N,A):- create-array(N,3,A), for-each-line(A, only1),
            pair-constraints(A,A), !, array-labeling(A).

pair-constraints([ ], -):- !.
pair-constraints([-], -):- !.
pair-constraints([[K1,K2,K3] |A2],[[I1,I2,I3] |A1]):- and0(I1,K1),
          and0(I2,K2), and0(I3,K3), tri-constraints(A2,A1,[I1,I2,I3]),
                    pair-constraints(A2,A1).

tri-constraints ([ ],-,-).
tri-constraints([[K1,K2,K3] |A2], [[J1,J2,J3] |A1],[I1,I2,I3]):-
          and0(I1,J1,K1), and0(I2,J2,K2), and0(I3,J3,K3),
                    tri-constraints(A2,A1,[I1,I2,I3]).
```

Reales y racionales

Ejemplo 1 Cálculo de la envolvente convexa de un conjunto de puntos.

```
envol(X):- conv-envol([P | RP], X).
conv-envol(Puntos, XS):- lin-comb(Puntos, Lambdas, Cero, Xs), cero(Cero),
                    politopo(Lambdas).
politopo(Xs):- sum-positiva(Xs, 1).

sum-positiva([ ], Z):- { Z = 0 }.
sum-positiva([X|XS], SX):- {X ≥ 0, SX = X + SUM}, sum-positiva(XS, SUM).

cero([ ]).
cero([Z|ZS]):- { Z = 0 }, cero(ZS).
lin-comb([ ], [ ], S, S).
lin-comb([PS|RP],[K|KS],S1,S3):- lin-comb-r(PS,K,S1,S2),
                                 lin-comb(RP,KS,S2,S3).
lin-comb-r([ ], -, [ ], [ ]).
lin-comb-r([P|PS], K, [S|SS], [KPS|SS1]):- { KPS = K*P + S },
                                 lin-comb-r(PS, K, SS, SS1).
```

3 puntos en dimensión 3 representados por `[[1,0,0], [0,1,0], [0,0,1]]` tienen la envoltura convexa: `?- conv-envol([[1,0,0],[0,1,0],[0,0,1]], [X1,X2,X3]).` dada por `X3 = 1 - X1 - X2, X1 ≤ 1 - X2, X2 ≥ 0, X1 ≥ 0`

En dos dimensiones y para los puntos: `[[1,1], [2,0], [3,0], [1,2], [2,2]]`

```
envol([X,Y]):- conv-envol([[1,1], [2,0], [3,0], [1,2], [2,2]], [X,Y]).
```

1. Ecuaciones definidas por el predicado para `(Ki ≥ 0)`

```
K1 + 2*K2 + 3*K3 + K4 + 2*K5 = X
K1 +          + 2*K4 + 2*K5 = Y
K1 + K2 + K3 + K4 + K5 = 1
```

2. Solución paramétrica:

```
K1 = Y - 2*K4 - 2*K5                 K1 = - Y + 2 - 2*K2 - 2*K3
K4 = Y - X - 2*K2 - 3*K3            K5 = X - 1 - 3*K2 - 4*K3
K1 + K2 + K3 + K4 + K5 = 1
```

3. Condiciones deducibles de `Ki ≥ 0`.

```
K1 = Y - 2*K4 - 2*K5 ≥ 0 ⟺ Y ≥ 0
K1 = - Y + 2 - 2*K2 - 2*K3 ≥ 0 ⟺ - (Y - 2) ≥ 0 ⟺ Y ≤ 2
K5 = X - 1 - 3*K2 - 4*K3 ≥ 0 ⟺ (X - 1) ≥ 0 ⟺ X ≥ 1
```

4. Utilizamos que las `ki` han de sumar 1.

```
X + Y = (2 * SUMA DE Ki) + K3 + K4 + 2*K5 ⟺ X + Y ≥2
2*X + Y = (6 * SUMA DE Ki) - 3*K1 - 2*K2 - 2*K4 ⟺ 2*X + Y ≤6
X ≤ 3 - 1/2*Y, Y ≤2, Y ≥0, X ≥1, X ≥ 2 - Y.
```