Universidad Tecnológica de la Mixteca

Análisis de Sistemas de Programación Lógica por Restricciones

Tesis Profesional

Que para obtener el Titulo de

Ingeniero en Computación

Presenta

Wendy Yaneth García Martínez.

Acatlima, Huajuapan de León, Oaxaca Enero ‘98

Tesis presentada el 9 de Enero de 1998
ante los siguientes sinodales:

Dr. Anatoli Koulinitch
Dra. Virginia Berrón Lara
M. en I.A. Angélica Muñoz Meléndez

Asesor:

Dr. Anatoli Koulinitch

Dedicatorias

A mis padres, que son mi ejemplo a seguir ... por brindarme su apoyo y confianza en todo momento, me han alentado a seguir adelante pese a las adversidades y en los cuales he encontrado a mis más sinceros amigos.

A Iliana y Lorena que más qué mis hermanas, son mis amigas y a las cuales admiro por ese valor para sacar adelante a su familia.

A mi abuelito Arturo, por demostrar fortaleza y temple ante la vida y ser ejemplo para todos los que le amamos.

A mis amigas Karina y Lily, para que nunca se den por vencidas y luchen siempre por sus sueños e ideales.

A tantas y tantas personas que a lo largo de mi vida me han brindado su amistad y cariño incondicional y de las cuales siempre tendré un recuerdo muy especial.

Wendy Yaneth

Agradecimientos

A todas aquellas personas que confiaron en mi, me impulsaron a seguir adelante y han estado pendientes de mi avance profesional.

A toda mi familia en general por su apoyo moral y económico.

Agradezco sinceramente a:

Anatoli Koulinitch,
Esteban Guerrero,
Pascal Van Hentenryck,
Roland Yap, y
Antonio Fenández Levi

por su apoyo en la búsqueda de mi información, así como su ayuda para esclarecer mis dudas.

A todas aquellas neuronas que se extinguieron en el cumplimiento de su deber ...

Wendy Yaneth García Martínez.

Dedicatoria Especial

A mi papá, donde quiera que este…

Porque más que mi papá fue mi amigo y mi confidente, me demostró el verdadero sentido de la vida, siempre estubo ahí para ayudarme, fuera cual fuera el problema y del cual siempre me sentiré muy orgullosa …

Hoy ya no me acompañas en cuerpo, pero tu espirítu me guiará por siempre en cada una de mis metas, en cada uno de mis sueños, en cada uno de mis logros…

Te amo Pa´

tu hija
Wendy Yaneth

Indice

Introducción

Durante las últimas décadas, el paradigma de la **Programación Lógica** ha alcanzado su grado de madurez. La representación más conocida de esta clase de lenguajes de programación es **Prolog**, originado de ideas de Colmerauer en Marsella y Kowalski en Edim-burgo. “La programación en **Prolog** difiere de la programación convencional al hacer uso de la lógica para declarar el estado de un problema y de la deducción para resolverlo, **Prolog** se ha usado exitosamente en muchas áreas diferentes de aplicación.”[LS90]

La **Programación Lógica** provee una manera elegante de separar las partes **lógicas** y de **control** de un programa (en el sentido de la proposición de Kowalski [RK79]). En un caso ideal, la lógica de predicados de primer orden se usa para representar el problema (¿qué es lo que tengo que hacer?) y el perfil del mecanismo de cálculo se usa para resolver el problema (¿cómo tiene que encontrarse la solución?).
De esta forma, la **Programación Lógica** tiene la propiedad de ser **semántica, operacional** y **declarativa**. Las semánticas operacional y declarativa son tanto simples y elegantes, como equivalentes. Sin embargo, hay que pagar un precio por esta dualidad. Los únicos objetos manipulados por la **Programación Lógica** son estructuras no interpretadas, y la igualdad se da únicamente entre objetos sintácticamente idénticos. Esto forza a un razonamiento a bajo nivel, ya que cada objeto semántico tiene que codificarse explícitamente dentro de un término en el programa.

La regla de cálculo en la **Programación Lógica** es el origen de otro problema; el procedimiento de **búsqueda a profundidad** resulta de la aplicación del paradigma **genera/examina**, con todos sus problemas bien conocidos en la búsqueda a grandes espacios.

La **Programación Lógica por Restricciones** (o **CLP** - **Constraint Logic Programming**) intenta cubrir las fallas de la Programación Lógica integrando a un lenguaje como **Prolog** con mecanismos que solucionen restricciones. Curiosamente estas fallas se pueden disipar usando **restricciones**. **CLP** por lo tanto, es la unión natural de dos paradigmas declarativos: la **solución de restricciones** y la **programación lógica**. Esta combinación ayuda a hacer programas **CLP** expresivos y flexibles, y en algunos casos, más eficientes que otros programas.

Los lenguajes **CLP** reducen dramáticamente el tiempo de ejecución mientras logran una eficiencia similar a los lenguajes procedimentales. Los programas resultantes son más cortos y declarativos, por lo tanto más fáciles de mantener, modificar o extender. La riqueza en sus aplicaciones muestra la flexibilidad de **CLP** para adaptarse a diferentes áreas.

El objetivo de esta tesis es hacer un análisis de algunos sistemas de programación lógica que satisfacen restricciones (**CHIP**[PVH-88], **Prolog III**[PIII-93] y **CLP®**[JJ-92]); así como mostrar la superioridad expresiva de **CLP®** con respecto a estos lenguajes y su receptibilidad a la implementación práctica, a fin de aumentar el interés por las aplicaciones en esta área.

El trabajo consta de 4 capítulos:

En el capítulo 1 se introduce el esquema de los lenguajes **CLP**, su semántica lógica y operacional, y las consecuencias de su implementación.

El capítulo 2 muestra cómo se clasifican los lenguajes **CLP** y se destacan las principales características de los sistemas a analizar como son **CHIP, Prolog III** y **CLP®**, mostrando las ventajas y desventajas entre cada uno de ellos.

El capítulo 3 apunta a las capacidades del sistema y lenguaje **CLP®**, su modelo operacional y sus características generales.

Finalmente, en el capítulo 4 se muestran las aplicaciones prácticas del sistema **CLP®**, al presentar una aplicación en el campo del Diseño Financiero y una aplicación en el campo de los Circuitos Eléctricos, terminando el trabajo de tesis con las conclusiones obtenidas.

1
Programación Lógica por Restricciones (o CLP)

1.1 **Esquema de los Lenguajes CLP**

El aspecto clave de **CLP** es propagar restricciones a la programación lógica clásica (o **LP** - Logic Programming), y permitir el uso implícito de algoritmos especializados en el sistema de la programación lógica. Una de las principales desventajas de **LP**, es la necesidad de expresar cada objeto como una estructura no interpretada (términos del universo de Herbrand); mientras que el programador desea usar conceptos que se acerquen lo más posible a su dominio de cálculo (conjuntos, booleanos, enteros, reales).

El esquema de la **Programación Lógica por Restricciones** (o **CLP**) define una familia de lenguajes de la Programación Lógica, donde cada lenguaje es una instancia obtenida al especificar una estructura de cálculo. El lenguaje se caracteriza así, por una estructura algebraica (el dominio de cálculo, funciones y las relaciones sobre este dominio). “Todos estos lenguajes se basan sólidamente en una sola estructura de semántica formal la cual se extiende, de manera natural, a la estructura de la Programación Lógica” [JL87][JM92].
Puede decirse que la Programación Lógica por Restricciones implica la incorporación de restricciones y métodos que **solucionan** restricciones en un lenguaje basado en la lógica. Esta caracterización sugiere la posibilidad de muchos lenguajes basados en diferentes restricciones y diferentes lógicas, en donde **CLP** hereda (de la programación lógica), la implementación de las cláusulas de Horn.

1.1.1 Estructura General

Funciones especiales y predicados simbólicos se introducen dentro de la Programación Lógica, cuya interpretación sobre el dominio seleccio-nado es fijo, estos forman los **símbolos interpretados**. Las relaciones sobre el dominio de cálculo se llaman **restricciones**, estas se formulan involucrando las funciones especiales y los predicados simbólicos.

La unificación es la única operación para modificar datos disponibles en la Programación Lógica. Esta compara sintácticamente dos términos y produce un

unificador general en caso de ser unificables. Existen dos etapas en el proceso denotado como unificación: decidir si una relación **t = s** tiene solución; y, en caso de que la solución exista, encontrar su forma explícita. El primer aspecto es el más importante, el cual puede tomarse como un algoritmo decisivo mientras una **restricción** o conjunto de restricciones tiene solución o no.

Si el dominio original de la Programación Lógica es capaz de incluir no solamente al universo de Herbrand sino también a otros dominios algebraicos, su mecanismo tendría que aumentarse con un **resolvedor de restricciones** apropiado, el cual decidirá sobre la satisfactibilidad de los conjuntos de restricciones en un dominio específico y si la solución existe, encontrará también algunas formas ortodoxas.

"La Programación Lógica por Restricciones es así una extensión de la Programación Lógica, donde:

- no hay restricción al universo de Herbrand,
- no hay restricción a la unificación,
- no hay restricción a las ecuaciones,

dando como resultado un lenguaje de programación basado en la lógica, la semántica operacional, la semántica declarativa y las relaciones entre estas semánticas; limitado por el dominio de cálculo y las restricciones"[LM93].
El esquema resultante define las clases de lenguajes CLP(**X**) obtenidos al instanciar el parámetro **X**. Este parámetro se interpreta como la tupla (∑**,D,L,T**) donde ∑ es una asignatura, **D** es una estructura de ∑, **L** es una clase de fórmulas en función de ∑ y **T** es una teoría en ∑ de primer orden. ∑ determina el predicado predefinido y la función simbólica así como sus aridades, **D** es la estructura sobre la cual se ejecuta el cálculo, **L** es la clase de restricciones que pueden tener sentido y **T** es una axiomatización a algunas propiedades de **D** [JM87].

1.1.2 Dominios restrictivos

Para cualquier asignatura ∑, sea **D** una estructura de ∑ (el dominio de cálculo) y **L** una clase de fórmulas de ∑ (las restricciones). Llamando al par (**D,L**) un **dominio restrictivo**[1] . Para simplificar la exposición, se asume que:

- los términos y restricciones en **L** se dan a partir de un lenguaje de primer orden[2].
- el predicado simbólico binario "**=**" está contenido en ∑ y se interpreta como una identidad en **D**[3]**.**

[1] En un ligero abuso de notación algunas veces se denotará al dominio restrictivo únicamente por D
[2] Sin esta suposición, algunos de los resultados no son aplicables dado que no puede haber una teoría **T** de primer orden apropiada.
[3] Esta suposición se vuelve innecesaria cuando los términos tienen un unificador más general en D, por otro lado el "**=**" se necesita para expresar el paso de parámetros.

- hay restricciones en **L** las cuales son respectivamente, ciertas o falsas en **D**.
- las clases de restricciones en **L** se aproximan bajo la variable renombrada, la conjunción y el cuantificador existencial.

Se denota al conjunto más pequeño de restricciones, el cual satisface estas suposiciones y contiene todas las restricciones primitivas, por $\mathbf{L}_{\Sigma}$. **L** puede ser estrictamente mayor que $\mathbf{L}_{\Sigma}$ dado que los cuantificadores universales o disyunciones se permiten en **L** , sin embargo se tomará $\mathbf{L} = \mathbf{L}_{\Sigma}$. En ocasiones se considerará una extensión de Σ y **L** por Σ^* y **L*** respectivamente, tal que haya una constante en Σ^* para cada elemento de **D**.

Ejemplo

Permitamos que Σ contenga al predicado simbólico "=", la función simbólica "**.**", una constante λ y un sin número de constantes. Sea D el conjunto de cadenas finitas de las constantes. El símbolo "**.**" se interpreta en **D** como una concatenación de cadenas y λ se interpreta como una cadena vacía. **L** es el conjunto de restricciones generadas por las ecuaciones entre términos. Entonces **WE=(D, L)** es el dominio restringido de las ecuaciones sobre las cadenas. Una restricción por ejemplo es **x. a = b. x**[4] [AC90].

El dominio restrictivo para valores booleanos y funciones se usa en **BNR-Prolog**[OB93], **CAL**[AS88], **CHIP**[PVHD88] y **Prolog III**[AC90]. **CAL** y **CHIP** emplean un dominio restringido más general, el cual incluye valores booleanos simbólicos. Los dominios finitos de **CHIP** son mejor vistos teniendo a los enteros como la estructura fundamental en el lenguaje restrictivo.

Otros dominios restrictivos de interés incluyen restricciones pseudo-booleanas [AB93], las cuales son intermediarias de las restricciones enteras y booleanas; dominios consistentes de conjuntos regulares de cadenas [CW89]; dominios de conjuntos finitos [DR93], dominios CLP(Fun(D)) el cual emplea una función variable [TH93], entre otros.

Estos dominios restrictivos soportan (quizás de una forma débil) exámenes y operaciones sobre restricciones, los cuales tienen mayor importancia en los lenguajes CLP.

- primeramente se requiere un examen de **consistencia** y **satisfactibilidad**: $\mathbf{D} \models \tilde{\exists} c$ **.**

[4] **x**, **a** y **b**; son elementos de cadenas finitas en D.

- segundo, la **implicación** (o **vinculación**) de una restricción por otra : **D** $\models \infty \rightarrow$ **c_i** . Generalmente podemos preguntar si una disyunción de restricciones se implica por una restricción **D** $\models \infty \rightarrow$ **$V^n_{i=1}$ c_i**
- la **proyección** de una restricción **c_0** dentro de las variables $\tilde{x}$ para obtener una restricción **c_1** tal que **D** $\models c_1 \leftrightarrow \exists_{-\tilde{x}} c_0$. Es siempre posible tomar **c_1** para que sea $\exists_{-\tilde{x}} c_0$, pero se espera calcular los **c_1** más simples con menos cuantificadores. En general, no es posible eliminar todos los usos del cuantificador existencial.
- la **detección** de que, dada una restricción **c_1** exista un solo valor que la variable **x** pueda tomar tal que sea consistente con **c**. Esto es, **D** $\models c(x,\tilde{z}) \wedge c(y,\tilde{w}) \rightarrow x = y$ o bien **D** $\models \exists z \forall x, \tilde{y} c(x,\tilde{y}) \rightarrow x = z$ **.** Decimos que **x** está determinada (o aterrizada) por **c**.

La primera operación es la más importante, mientras que las otras pueden o no usarse en algunos lenguajes **CLP**. Algunas implemen-taciones de estas operaciones (en particular el examen de satisfacti-bilidad) están incompletos. En algunos casos se ha argumentado que si un algoritmo está incompleto con respecto al dominio restrictivo deseado, este puede ser completo con respecto a otro dominio restrictivo (construido artificialmente) [AC93][AC-93][BM93].

Algunas propiedades de los dominios restrictivos son:

Definición 1

La semántica algebraica del esquema **CLP** se basa en varios tipos de (estructuras algebraicas) **X**. Decimos que **X** es una **solución compacta** si:

1. cada elemento en **X** es la única solución de un conjunto finito o infinito de restricciones.
   $\forall$d $\in$ X, d = $\cap$**C_i** donde **C_i** son las restricciones.
2. cada elemento en el complemento del espacio solución de una restricción **C,** corresponde al espacio solución de cada una de las restricciones **C_i** ;
   $\forall C_i \tilde{C} = \cup C_i$ por razones de la negación, requerimos que la teoría **T** la cual corresponde a **X_T**, sea una **satisfacción completa** (satisfaction complete), esto es:

   T $\models \neg$**C** siempre que no **T** $\models \exists$**C,** decimos que la teoría **T** corresponde a la estructura **X** si :
   X $\models$ **T**, **X** modela **T** y **X** $\models \tilde{\exists} C$ implica que **T** $\models \exists$**C** $\forall$**C**

Ejemplo 1.1

Sea $\mathfrak{R}^{+}_{\mathbf{Lin}}$ el dominio restrictivo obtenido de $\mathfrak{R}_{\mathbf{Lin}}$**= (D',L')**[5] al agregar la restricción primitiva $\mathbf{x \neq \pi}$**.** La negación de esta restricción ($\mathbf{x = \pi}$) no se puede representar como una disyunción de restricciones en $\mathfrak{R}^{+}_{\mathbf{Lin}}$. Así $\mathfrak{R}^{+}_{\mathbf{Lin}}$ no es una solución compacta.

Definición 2

Para una asignatura Σ, sea (**D,L**) un dominio restrictivo con una asignatura Σ y una teoría **T**. Decimos que **D** y **T** corresponden a **L** si:

- **D** es un modelo de **T**, y
- para cada restricción $\mathbf{c \in L}$, $\mathbf{D} \models \tilde{\exists} c$ si $\mathbf{T} \models \tilde{\exists} c$.

Decimos que **T** es una **satisfacción completa** con respecto a **L** si para cada restricción $\mathbf{c \in L}$, $\mathbf{T} \models \tilde{\exists} c$ ó $\mathbf{T} \models \neg \tilde{\exists} c$. "La satisfacción completa es una debilidad en la noción de una teoría completa"[SB92].

La noción de independencia de las restricciones negadas juegan un papel importante en la **Programación Lógica por Restricciones**, esto es; la independencia de los estados de desigualdades: "si una conjunción de restricciones negativas y positivas son inconsistentes, entonces una de las restricciones negativas es inconsistente con las restricciones positivas" [AC84].

Definición 3

Un dominio restrictivo (**D,L**) tiene la propiedad de **independencia de las restricciones negadas** si, para todas las restricciones $\mathbf{c, c_1, \ldots, c_n \in L}$,

$\mathbf{D} \models \tilde{\exists} c \wedge \neg c_1 \wedge \ldots \wedge \neg c_n$ si $\mathbf{D} \models \tilde{\exists} c \wedge \mathbf{c_i}$ para $\mathbf{i = 1, \ldots, n}$.

El hecho de que **L** se asuma a ser estrecho bajo la conjunción y el cuantificador existencial es una restricción importante en la anterior definición.

Ejemplo 3.1

---

[5] Dominio restringido en la aritmética lineal sobre números reales.

En un dominio restrictivo de Herbrand (**FT**) con solo dos funciones simbólicas, una constante "**a**" y una función unaria "**f**", se ve que las siguientes declaraciones son ciertas: **FT** $\models \exists x,y,z\ x = f(y) \wedge \neg y = a \wedge \neg y = f(z)$; **FT** $\models \exists x,y\ x = f(y) \wedge \neg y = a$; **FT** $\models \exists x,y,z\ x = f(y) \wedge \neg y = f(z)$.

Sin embargo, cuando se considera por completo al tipo de restricciones en **FT** tenemos los siguientes hechos: la declaración **FT** $\models \exists x,y\ x=f(y) \wedge \neg y=a \wedge \neg \exists z\ y=f(z)$ no es cierta, dado que se dice que cada árbol finito "**y**" es tanto una constante "**a**" o tiene la forma "**f(z)**" para algunos árboles finitos "**z**".

Por otro lado, tanto **FT** $\models \exists x,y\ x = f(y) \wedge \neg y = a$ como **FT** $\models \exists x,y,z\ x = f(y) \wedge \neg \exists z, y = f(z)$ es cierto. Así para estas funciones simbólicas, la independencia de restricciones negadas no se mantiene.

1.1.3 Semánticas Lógicas[6]

Existen dos semánticas lógicas comunes de los programas **CLP** sobre un dominio restringido (**D,L**). La primera interpreta una regla

$$p(\tilde{x}) \leftarrow b_1,\ldots,b_n$$

como la fórmula lógica $\forall \tilde{x},\tilde{y} p(\tilde{x}) \vee \neg b_1 \vee \ldots \vee \neg b_n$, donde $\tilde{x} \cup \tilde{y}$ es el conjunto de todas las variables libres de la regla. La colección de todas las fórmulas correspondientes a las reglas de **P** da una teoría también denotada por **P**.

La segunda semántica lógica asocia una fórmula lógica a cada predicado en **P.** Si el conjunto de todas las reglas de **P** con **p** en la cabeza es:

$$p(\tilde{x}) \leftarrow B_1$$
$$p(\tilde{x}) \leftarrow B_2$$
$$\ldots$$
$$p(\tilde{x}) \leftarrow B_n$$

entonces la fórmula asociada a **p** es:

$$\forall \tilde{x} p(\tilde{x}) \leftrightarrow$$

[6] Ver J. Jaffar, M.J. Maher. *Constraint Logic Programming: A survey.* pág. 15-16

$$\exists \tilde{y}_1 B_1$$
$$\vee \exists \tilde{y}_2 B_2$$
$$\ldots$$
$$\vee \exists \tilde{y}_n B_n$$

donde $\tilde{y}_i$ es el conjunto de variables en **B_i** excepto para las variables en $\tilde{x}$. Si **p** no ocurre en la cabeza de la regla **P** entonces la fórmula es $\forall \tilde{x} \neg p(\tilde{x})$.

Una **valoración** es un trazado de variables a **D**, y su extensión, la cual traza términos a **D** y fórmulas cercanas a las L*****. Si **X** es un conjunto de hechos entonces **$[X]_D$ = { v(a) | (a ← c) ∈ X, D ⊨ v(c)}.**

Una **interpretación *D,*** representa una interpretación de la fórmula con el mismo dominio **D** y la misma interpretación para los símbolos en $\sum$ como **D**. Esto puede representarse como el subconjunto **B_D** donde **B_D = { p($\tilde{d}$) | p ∈ P,** $\tilde{d}$ **∈ D^* }.** Un modelo **D** de una fórmula estrecha es una interpretación **D** la cual es un modelo de dicha fórmula.

Sea **T** una teoría de satisfacción completa para (**D,L**). La semántica lógica usual se basa en los modelos **D** de **P** y los modelos **P*,T**. El modelo mínimo **D** de una fórmula **Q** bajo el subconjunto ordenado se denota por **lm(Q,D)**, y el más grande se denota por **gm(Q,D)**. Una solución a una pregunta **G** es una valoración **v** tal que **v(G) ⊆ lm(P,D)**.

Las semánticas lógicas y algebraicas son las mismas con excepción de los aspectos operacionales de falla finita [JL87]. Falsamente, **CLP** no se caracteriza por la falla finita del intérprete, sino que se caracteriza únicamente por una **falla finita aterrizada**[7]. Esto se hace dada la existencia de derivaciones infinitas que dan lugar a un conjunto insoluble de restricciones respuesta.

Si bien muchas estructuras algebraicas satisfacen el criterio de la solución compacta, estas se eligen como base para la implementación de una instancia en particular de un lenguaje **CLP(X).** Cualquier dominio restringido nuevo también necesita satisfacer tanto el criterio práctico como el técnico. Por lo tanto

- el poder expresivo del dominio de cálculo tiene que ser suficiente para justificar cualquier esfuerzo de implementa-ción.

[7] Información con respecto a esta característica se presenta mas adelante.

- debe existir un resolvedor de restricciones. El resolvedor tiene que ser completo[8].
- tiene que existir una amplia área de aplicaciones.

Un problema práctico surge de la naturaleza incremental de los programas lógicos por restricciones. Las restricciones se acumulan progresivamente y el resolvedor de restricciones se tiene que aplicar tan pronto como sea posible para eliminar la búsqueda sobre subespacios donde no se pueda encontrar la solución (debido a la insatisfactibilidad de las restricciones). Una condición adicional es la existencia de un **resolvedor incremental**.

1.1.4 Semánticas del Punto Fijo[9]

Las semánticas del punto fijo surgen a partir de funciones consecuentes de un paso $\mathbf{T}^{D}_{P}$ y $\mathbf{S}^{D}_{P}$, y el operador estrecho **[[P]]** generado por $\mathbf{T}^{D}_{P}$. Las funciones $\mathbf{T}^{D}_{P}$ y **[[P]]** se trazan sobre las interpretaciones **D**.

Sea $\mathbf{T}^{D}_{P}(\mathbf{I}) = \{ p(\tilde{d}) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ como una regla de $\mathbf{P}$, $a_i \in \mathbf{I}$, $i=1,\ldots,n$; $\mathbf{v}$ es una valoración de **D** tal que $\mathbf{D} \models \mathbf{v}(c)$, $\mathbf{v}(\tilde{x}) = \tilde{d}$ y $\mathbf{v}(b)=a_i$, $i = 1, \ldots, n\}$

[[P]] es el operador estrecho generado por $\mathbf{T}^{D}_{P}$. Este representa un cierre deductivo basado en las reglas de **P**. Denotamos a **Id** como la función identidad, y definimos $(\mathbf{f} + \mathbf{g})(\mathbf{x}) = \mathbf{f}(\mathbf{x}) \cup \mathbf{g}(\mathbf{x})$. Entonces **[[P]](I)** es el menor punto fijo de $\mathbf{T}^{D}_{P} + \mathbf{Id} > \mathbf{I}$, y el menor punto fijo de $\mathbf{T}^{D}_{P \cup I}$.

La función $\mathbf{S}^{D}_{P}$ se define en los conjuntos de hechos, la cual forma un enrejado bajo el subconjunto ordenado. Se denota al operador estrecho generado por $\mathbf{S}^{D}_{P}$ como **<< P >>**. Ambas funciones son continuas.

Sea $\mathbf{S}^{D}_{P}(\mathbf{I}) = \{ p(\tilde{x}) \leftarrow c \mid p(\tilde{x}) \leftarrow c', b_1, \ldots, b_n$ una regla de $\mathbf{P}$, $a_i \leftarrow c_i \in \mathbf{I}$, $i = 1,\ldots,n$ la regla y los hechos se renombran aparte, $\mathbf{D} \models c \leftrightarrow c' \wedge \Lambda^{n}_{i=1} c_i \wedge a_i = b_i \}$

Sea el mínimo punto fijo de una función **f**, **lfp(f)** y el máximo punto fijo **gfp(f)**. Para una función **f** trazando interpretaciones **D**, se define la iteración ascendente y descendente de **f** como sigue:

$\mathbf{f} \uparrow 0 = \phi$
$\mathbf{f} \uparrow (\alpha + 1) = \mathbf{f}(\mathbf{f} \uparrow \alpha)$
$\mathbf{f} \uparrow \beta = \mathbf{U}_{\alpha < \beta}\, \mathbf{f} \uparrow \alpha$ si β es un límite ordinal

[8] El resolvedor es completo si es capaz de decidir la satisfactibilidad de cualquier conjunto de restricciones del dominio de cálculo.
[9] Ver J. Jaffar, M.J. Maher. *Constraint Logic Programming: A survey.* págs. 16-17

$f \downarrow 0 = B_D$
$f \downarrow (\alpha + 1) = f(f \downarrow \alpha)$
$f \downarrow \beta = \cap_{\alpha < \beta} f \downarrow \alpha$ si β es un límite ordinal

Basándose en el hecho de que los modelos **D** de **P** son puntos fijos de **[[P]]** y los modelos **D** de **P*** son puntos fijos de T^D_P, se tienen las siguientes conexiones entre las semánticas de punto fijo y las semánticas lógicas, sólo como programación lógica estándar.

Proposición 1

Sean **P**, P_1, P_2 programas **CLP** y **Q** un conjunto de hechos sobre un dominio restringido **D** con la teoría correspondiente **T**. entonces :

- $T^D_P \uparrow \omega = lfp(T^D_P) = [lfp(S^D_P)]_D = [[P]] (\phi)$
- $lm(P,D) = [\{h \leftarrow c \mid P^*, D \models (h \leftarrow c) \}]_D = [\{ h \leftarrow c \mid P^*, T \models (h \leftarrow c) \}]_D$
- $lm(P^*,D) = lm(P, D) = lfp(T^D_P)$
- $gm(P^*,D) = gfp(T^D_P)$
- $[[P]] ([Q]_D) = [[P \cup Q]] (\phi) = lm(P \cup Q, D)$
- $<<P>> (Q) = <<P \cup Q>> (\phi) = lfp(S^D_{P \cup Q})$
- $D \models P_1 \leftrightarrow P_2$ si $[[P_1]] = [[P_2]]$

P se dice que es (**D,L**) **canónica** si $gfp(T^D_P) = T^D_P \downarrow \omega$.
Los programas lógicos canónicos [JP88] muestran que cada programa lógico es equivalente a un programa lógico canónico. Como muchas otras clases de resultados en la programación lógica, estos resultados se extienden comúnmente a **CLP** de una manera directa.

1.1.5 Ejecución Descendente (o Top - Down)

La ejecución descendente (o top - down), cubre múltiples modelos operacionales. La semántica operacional se presenta como un sistema de transición de estados: tuplas **<A,C,S>** donde **A** es un conjunto de átomos y restricciones, **C** y **S** son conjuntos de restricciones. Las restricciones **C** y **S** se refieren a cómo se almacenan las restricciones y, las implementaciones se actualizan por un resolvedor de restricciones.

A es una colección de restricciones y átomos todavía invisibles, **C** es la colección de restricciones las cuales juegan un **papel activo** (o están excitadas), y **S** es una colección de restricciones que juegan un **papel pasivo** (o están diferidas). Existe otro

estado denotado por **falla**[10]. Se asume una **regla de cálculo** la cual selecciona un tipo de transición y un elemento apropiado de **A** para cada estado. El sistema de transición también se limita por un predicado **consistente** y una función **inferir**. Una **meta** inicial **G** se representa como un estado por **<G, ϕ, ϕ>.** Las transiciones en el sistema de transición son:

< A $\cup$ a, C, S > $\rightarrow_r$ < A $\cup$ B, C, S $\cup$ (a = h) >

si **a** se selecciona por una regla de cálculo, **a** es un átomo, **h $\leftarrow$ B** es una regla de **P**, renombrada para nuevas variables, y **h** y **a** tienen el mismo predicado simbólico. **a** se reescribe en esta transición.

< A $\cup$ a, C, S > $\rightarrow_r$ falla

si se selecciona **a** por la regla de cálculo, **a** es un átomo y, para cada regla **h $\leftarrow$ B** de **P, h** y **a** tienen diferentes predicados simbólicos.

< A $\cup$ c, C, S > $\rightarrow_c$ < A, C, S $\cup$ c >

si se selecciona la restricción **c** por medio de la regla de cálculo.

< A, C, S > $\rightarrow_i$ < A, C', S' > si **(C',S') = inferir(C, S)**
< A, C, S > $\rightarrow_s$ < A, C, S > si **consistente(C)**

<A, C, S> $\rightarrow_s$ falla si **$\neg$consistente(C).**

La transición $\rightarrow_r$ surge de la resolución de la transición $\rightarrow_c$ la cual introduce restricciones dentro del resolvedor de restricciones; la transición $\rightarrow_s$ examina si las restricciones activas son consistentes; y, finalmente, la transición $\rightarrow_i$ infiere más restricciones activas (y quizás modifica las restricciones pasivas) de la colección actual de restric-ciones. Se escribe $\rightarrow$ para referir una transición de tipo arbitrario.

El predicado **consistente(C)** expresa un examen de consistencia para **C**. Se define por: **consistente(C)** si **D** $\models \tilde{\exists} C$ esto es, un examen de consistencia completa. Sin embargo los sistemas pueden emplear un examen conservativo pero incompleto (o parcial) si **D** $\models \tilde{\exists} C$ entonces **consistente(C)** se mantiene, pero algunas veces **consistente(C)** se mantiene a través de **D** $\models \neg \tilde{\exists} C$ **.**

La función **inferir(C,S)** calcula del conjunto actual de restricciones, un nuevo conjunto de restricciones activas **C'** y restricciones pasivas **S'.** Estas se agregan a **C** para formar **C'** y **S** simplifica a **S'**. Denotándose que **D** $\models$ **(C** $\wedge$ **S)** $\leftrightarrow$ **(C'** $\wedge$ **S'),** tal que la información sea perdida o supuesta por la inferencia.

[10] La definición de este concepto se dará más adelante.

El papel que juega la **inferencia** varía ampliamente de sistema a sistema. En **Prolog**, no hay restricciones pasivas y se puede definir **inferir(C,S) = (C ∪ S, ϕ).** En **CLP®** las restricciones no lineales son pasivas e **inferir** simplemente pasa una restricción de **S** a **C'** cuando la restricción se vuelve lineal en el contexto de **C**, y elimina la restricción en **S**.

Generalmente, las restricciones activas se determinan sintácticamente. En **Prolog** todas las ecuaciones son activas, en **CLP®** todas las restricciones lineales están activas, en el dominio finito de **CHIP** todas las restricciones unarias (restricciones con una sola variable) están activas, y en el intervalo aritmético de **BNR-Prolog** solamente los intervalos están activos.

Un **sistema CLP** se determina por su dominio restrictivo y una semántica operacional detallada. Lo anterior implica una regla de cálculo y definiciones para la **consistencia** y la **inferencia**.

Algunas propiedades significativas de los sistemas **CLP** [JL86][JL87] son:

Definición 4

Sea $\rightarrow_{ris} = \rightarrow_r \rightarrow_i \rightarrow_s$ y $\rightarrow_{cis} = \rightarrow_c \rightarrow_i \rightarrow_s$. Se dice que un sistema **CLP** es de **verificación rápida** si su semántica operacional se puede describir por $\rightarrow_{ris}$ y $\rightarrow_{cis}$. Un sistema **CLP** es **progresivo** si, para cada estado con una colección vacía de átomos, cada derivación a ese esta-do falla, conteniendo una transición $\rightarrow_r$ o conteniendo una transición $\rightarrow_c$. Un sistema **CLP** es **ideal** si este es de verificación rápida, progresivo, la inferencia se define por **inferir(C,S) = (C ∪ S,ϕ)** y **consistente(C)** se mantiene si **D** $\models \tilde{\exists} C$.

En un sistema de verificación rápida, la inferencia de nuevas restricciones activas se ejecuta y se hace un examen de consistencia cada vez que la colección de restricciones en el solucionador de restricciones cambia. Así, aún con las limitaciones de la **consistencia** y la **inferencia**, este encuentra la inconsistencia tan pronto como sea posible.

Una **derivación** es una secuencia de transiciones **< A_1, C_1, S_1 > → … → < A_i, C_i, S_i>.** Un estado el cual no puede derivarse más se llama **estado final**. Una derivación es **exitosa** si esta es finita y el estado final tiene la forma **<ϕ, C, S>.**

Sea **G** una meta con variables libres $\tilde{x}$ **,** las cuales inician una derivación y producen un estado final **<ϕ, C, S>.** Entonces $\exists_{-\tilde{x}} C \wedge S$ es llamada la **restricción respuesta** de la derivación.

Una derivación es **falla** si esta es finita y el estado final falla. Una derivación es **imparcial** si esta es falla ó, para cada **i** y cada $\mathbf{a} \in \mathbf{A_i}$, **a** se reescribe en una transición después. Una regla de cálculo es **imparcial** si da lugar únicamente a derivaciones imparciales.

Una meta **G** es **falla finita** si, para cualquier regla de cálculo imparcial, cada derivación de **G** en un sistema **CLP** ideal, falla. Esto muestra que si una meta es falla finita entonces cada derivación im-parcial en un sistema **CLP** ideal es falla. Una derivación se **esfuerza inútilmente** si esta es finita y el estado final tiene la forma **<A, C, S>** donde $\mathbf{A \neq \phi}$.

El **árbol de cálculo** de una meta **G** para un programa **P** en un sistema **CLP** es un árbol con nodos etiquetados por estados y ramas etiquetadas por $\rightarrow_r$, $\rightarrow_c$, $\rightarrow_i$ o $\rightarrow_s$ tal que : la raíz esté etiquetada por **<G, ϕ, ϕ>**; para cada nodo, todas las ramas externas tiene la misma etiqueta; si un nodo **S** tiene una rama externa etiquetada por $\rightarrow_c$ $\rightarrow_i$ o $\rightarrow_s$ entonces el nodo tiene exactamente un hijo, y el estado que etiqueta ese hijo se puede obtener de **S** vía una transición $\rightarrow_c$, $\rightarrow_i$ o $\rightarrow_s$ respectivamente; si un nodo etiquetado por un estado **S** tiene una rama externa etiquetada por $\rightarrow_r$ entonces el nodo tiene un hijo para cada regla en **P**, y el estado que etiqueta cada hijo es el estado obtenido de **S** por la transición $\rightarrow_r$ para la regla; para cada rama $\rightarrow_r$ y $\rightarrow_c$, la correspondiente transición usa el átomo o la restricción seleccionada por la regla de cálculo.

Cada rama de un árbol de cálculo es una derivación y, dada una regla de cálculo, cada derivación seguida de esta regla es una rama del correspondiente árbol de cálculo. Diferentes reglas de cálculo pueden surgir para árboles de cálculo de tamaños diferentes.

Sea la transición $\mathbf{< A, C, S > \rightarrow_g < A, C', \phi >}$ donde **C** es un conjunto en L* tal que $\mathbf{D \models C' \rightarrow (C \wedge S)}$ y, para cada variable **x** que ocurra en **C** o **S, C'** contiene una ecuación **x = d** para alguna constante **d**. Se define a $\rightarrow_g$ como el estado aterrizado. **< A, C, S>** $\mathbf{\rightarrow_g}$ **falla** si **C'** no existe ($\mathbf{C \wedge S}$ es insatisfactible en **D**). Una **derivación aterrizada** es una derivación compuesta de $\rightarrow_r$ $\rightarrow_g$ o $\rightarrow_c$ $\rightarrow_g$.

Tres conjuntos que cristalizan los aspectos de la semántica operacional son: el conjunto exitoso **SS(P)** el cual consta de las restricciones respuesta para simplificar las metas **p(**$\tilde{x}$**),** el conjunto de falla finita **FF(P)** que consta del conjunto de metas simples finitas, y finalmente el conjunto de falla finita aterrizada **GFF(P)** el cual consta del conjunto de átomos aterrizados; todas aquellas derivaciones imparciales aterrizadas son falla. Su representación simbólica es:

SS(P) = { p($\tilde{x}$**)** $\leftarrow$ **c |** $\mathbf{< p(\tilde{x}), \phi, \phi > \rightarrow < \phi, c', c'' >}$, $\mathbf{D \models c \leftrightarrow \exists_{-\tilde{x}} c' \wedge c''}$ **}.**
FF(P) ={ p($\tilde{x}$**)** $\leftarrow$ **c |** para cada derivación imparcial, $\mathbf{< p(\tilde{x}), c, \phi> \rightarrow}$ **falla }.**

GFF(P)={p($\tilde{d}$)| para cada derivación imparcial aterrizada**,< p($\tilde{d}$),ϕ,ϕ> → falla }.**

1.1.6 Semánticas Operacional y Declarativa

Esta sección muestra la principal relación entre la semántica declarativa y la semántica operacional descendente. Para hacer esto sencillo, se considerará únicamente a los sistemas **CLP** ideales.

La validez de los resultados se mantiene para cualquier sistema **CLP**, dadas las restricciones de **consistencia** e **inferencia**. Los resultados completos para derivaciones exitosas requieren únicamente que el sistema **CLP** sea progresivo.

Teorema 1

Sea **P** un programa en un lenguaje **CLP** determinado por la tupla (∑**,D,L,T**) donde **D** y **T** son correspondientes a **L** y se ejecutan sobre un sistema **CLP** ideal. Entonces:

1. **SS(P) = lfp(S^D_P)** y **[SS(P)]**$_D$ **= Im(P,D)**
2. si la meta **G** tiene una derivación exitosa con la restricción respuesta **c**, entonces **P,T** ⊨ **c** → **G.**
3. sea **T** una satisfacción completa **L**. Si **G** tiene un árbol de cálculo finito con restricciones respuesta $c_1, \ldots, c_n$ entonces **P*,T** ⊨ **G** ↔ $c_1 \vee \ldots \vee c_n$
4. si **P, T** ⊨ **c** → **G** entonces hay derivaciones para la meta **G** con restricciones respuesta $c_1, \ldots, c_n$ tal que **T** ⊨ **c** → $\bigvee_{i=1}^{n} c_i$. Si al agregar (**D,L**) existe independencia de restricciones negadas entonces los resultados se mantienen para **n = 1**
5. sea **T** una satisfacción completa **L**. Si **P*,T** ⊨ **G** ↔ $c_1 \vee \ldots \vee c_n$ entonces **G** tiene un árbol de cálculo con restricciones respuesta $c_1', \ldots, c_m'$ (y posiblemente otras) tal que **T** ⊨ $c_1 \vee \ldots \vee c_n \leftrightarrow c_1', \ldots, c_m'$.
6. sea **T** una satisfacción completa **L**. La meta **G** es falla finita para **P**, si **P*,T** ⊨ ¬**G**
7. **gm(P*, D) = B_D**[11] **- GFF(P).**
8. sea (**D,L**) una solución compacta. $T^D_P \downarrow \omega = B_D$ **- [FF(P)]**$_D$
9. sea (**D,L**) una solución compacta. **P** será (**D, L**) canónica si **[FF(P)]**$_D$ **= [{ h** ← **c | P*,D** ⊨ ¬ **(h** ∧ **c) }]**$_D$.

Los resultados anteriores se han tomando de [JL86][JL87], así como de [GL91][MM87][MM93].
Las derivaciones exitosas completas [MM87] proveen una salida interesante en la teoría de la programación lógica, esto muestra que en **CLP** es necesario considerar y combinar diversas derivaciones exitosas (y sus respuestas) para establecer que **c** →

[11] Ver pág. 12

G se mantiene, mientras que en la programación lógica, únicamente se necesita una derivación exitosa.

En resumen, “si reemplazamos al universo de Herbrand por un dominio restrictivo arbitrario **D**, el mecanismo de unificación por la satisfacción de restricciones; la mayoría de los resultados (y aún sus pruebas) se disiparían al pasar de **LP** a **CLP**. Además, la mayoría de los aspectos operacionales de **LP** (y **Prolog**), se pueden interpretar como operaciones lógicas, y consecuentemente estas operaciones también se disiparían para **CLP** ” [MM93].

Finalmente, la filosofía del **esquema CLP** [JL87] da prioridad a la estructura **D** sobre la cual se ejecuta el cálculo, y menos prominencia a la teoría **T**. Sin embargo, también es posible iniciar con una teoría de satisfacción completa **T** [MM87] sin hacer referencia a una estructura.

1.2 **Implementación de un Sistema CLP**

La principal innovación requerida para implementar un sistema **CLP** claramente es la manipulación de las restricciones. En esta parte se considera el problema de extender la máquina de inferencia de la **Programación Lógica** para tratar con restricciones.

1.2.1 Algoritmos para solucionar restricciones

Existen varias operaciones que implican la implementación de las restricciones. Estas incluyen: un examen de satisfactibilidad, para implementar la **consistencia** e **inferencia**; un examen de vinculación, para implementar **metas protegidas**; y una **proyección** de la restricción almacenada dentro de un conjunto de variables, para calcular el estado final.

El resolvedor de restricciones tiene que ser capaz de anular los efectos de agregar nuevas restricciones cuando la máquina de inferencia retrocede.

1.2.1.1 Incrementabilidad

De acuerdo al estilo de **CLP**, los algoritmos para las implementaciones **CLP** tienen que ser **incrementales** para ser prácticos. Sin embargo esta descripción no es totalmente satisfactoria, dado que el término incremental puede usarse en dos sentidos diferentes.

Por un lado, la incrementabilidad se usa para referir la **naturaleza** del algoritmo. Esto es, un algoritmo es incremental si este acumula un estado inicial y una nueva entrada se procesa en combinación con el estado interno. Tales algoritmos son algunas

veces llamados algoritmos en línea. Por otro lado, la incrementabilidad es usada algunas veces para referirse a la **ejecución** del algoritmo. Esta sección sirve para aclarar esta última noción de incrementabilidad.

Denotemos al estado del solucionador de restricciones consistente de un almacén de restricciones como **C**, a un grupo de restricciones **G** que están por ser vinculadas, y algunos puntos de retroceso. En el estado inicial, denotado por ϕ no hay restricciones ni puntos de retroceso. El resolvedor de restricciones reacciona para una secuencia de operaciones, dando como resultado un nuevo estado y una respuesta.

Por ejemplo, consideremos la aplicación de una secuencia de operaciones $o_1,\ldots,o_k$ sobre un estado Δ; sea $F(\Delta, o_1,\ldots,o_k)$ el estado actualizado y $G(o_1, \ldots,o_k)$ la secuencia de operaciones respuestas [VF90]. Para una expresión $exp(\tilde{o})$ que denota una función $\tilde{o}$, se define $AV[exp(\tilde{o})]$ como el valor promedio de $exp(\tilde{o})$, sobre todas las secuencias de operaciones $\tilde{o}$.

Sea Δ el símbolo que denota a $F(\phi, o_1,\ldots,o_k)$. Sea "**A**" un algoritmo el cual aplique una secuencia de operaciones sobre el estado inicial, dando la misma respuesta que el resolvedor de restricciones, pero no necesariamente calculando el nuevo estado. Esto es, "**A**" es la versión de partida (off - line) del resolvedor de restricciones. Intuitivamente "**A**" representa el mejor algoritmo de partida disponible.

Consideremos un algoritmo para **F** y **G** que sea relativamente "**no incremental"** a "**A**"**,** si el costo promedio de aplicar una operación extra o_k a Δ no es mejor que el costo del método directo de usar "**A**" sobre $o_1,\ldots,o_k$, entonces:

$$AV[\, costo(\Delta, o_k)\,] \geq AV[\, costo_A(o_1,\ldots,o_k)\,].$$

Por otro lado, si el algoritmo para **F** y **G** es **"perfectamente incremental"** a "**A**", su costo no es peor que el de "**A**"**.** En otras palabras, no incurre costo alguno en la naturaleza incremental del algoritmo, esto es:

$$AV[\, costo(\phi, o_1,\ldots,o_{k-1}) + costo(\Delta, o_k)] \leq AV[\, costo_A(o_1,\ldots, o_k)].$$

En general, cualquier algoritmo reside algunas veces entre estos dos extremos. Este último no será perfectamente incremental, a menos que:

$$AV[\, costo(\phi, o_1, \ldots, o_{k-1}) + costo(\Delta, o_k)\,] = AV[\, costo_A(o_1, \ldots, o_k)\,] + costo\_extra(o_1, \ldots, o_k)$$

donde el término adicional $costo\_extra(o_1,\ldots, o_k)$ denota el costo extra incurrido por el algoritmo en línea sobre el mejor algoritmo considerado.

Por lo tanto, una posible "definición" de un algoritmo incremental en un sistema **CLP**, es simplemente que su factor de **costo extra** sea insignificante.

1.2.1.2 Satisfactibilidad

Es crucial que el algoritmo que determina la satisfactibilidad de una nueva restricción almacenada sea incremental. Sea una secuencia de restricciones $\mathbf{c_1, \ldots, c_k}$ de aproximadamente igual tamaño **N**. Una aplicación sencilla de este algoritmo de tiempo lineal es decidir que $\mathbf{c_1}$, entonces $\mathbf{c_1 \wedge c_2,}$ y finalmente $\mathbf{c_1 \wedge \ldots \wedge c_k}$ podrían incurrir en un costo proporcional a $\mathbf{Nk^2}$ en promedio. En contraste, un algoritmo incremental perfecto tiene un costo **O(Nk)** en promedio.

En la práctica, la mayoría de los algoritmos representan restricciones de alguna clase del **modelo solución,** un formato en el cual la satisfacción de restricciones es evidente. Así el problema de satis-factibilidad se reduce esencialmente dentro del modelo solución. Una propiedad importante del modelo solución es que este defina una representación apropiada de la **proyección** del espacio solución con respecto a un conjunto de variables. Mas específicamente, cada variable se puede igualar con una expresión que contenga única-mente variables paramétricas, esto es, variables cuyas proyecciones sean sobre el espacio entero. Esta propiedad ayuda a la incrementabilidad.
Un algoritmo de satisfacción que usa un modelo solución es:

Algoritmo 1 :
Dado **C**

(a) reemplazar las restricciones **c** más recientes por $\mathbf{c\theta}$, donde θ es la sustitución definida por **C**.
(b) escribir $\mathbf{c\theta}$ dentro de las ecuaciones de la forma $\mathbf{x = \ldots,}$ esto implica seleccionar **x** y los términos cambiados. La insatis-facción se detecta en esta etapa.
(c) si el paso previo existe, usar las nuevas ecuaciones para substituir todas las ocurrencias de **x** en **C**.
(d) finalmente, agregar las nuevas ecuaciones a **C** para obtener un modelo de solución para $\mathbf{C \wedge c}$.

Nótese que no se necesita de la presencia de variables de eliminación para la substitución de expresiones en (b) para asegurar que las nuevas ecuaciones estén en un modelo solución, y en (c) para asegurar que al aumentar **C** con las nuevas ecuaciones, este permanece en un modelo solución. El que esta metodología conduzca a un algoritmo incremental se basa en el hecho de que el costo de tratar con **c** está estrechamente relacionado con el tamaño de **c** (el cual es menor en promedio) que el de **C**. Esto es:

- la substitución de expresiones para las variables eliminadas en **c**, las cuales determinan en gran parte el tamaño de $\mathbf{c\theta,}$ con frecuencia tienen un tamaño independiente al tamaño de **C**, y

- el número de ocurrencias de la nueva variable eliminada **x** en **C**, la cual determina en gran parte el costo de substitución externo **x** en **C**, es menor en comparación al tamaño de **C**.

Un algoritmo de satisfacción tomado de [AC82] es:

Sea **x** una variable y, **s** y **t** términos no variables:

(a) descartar cada $x = x$;

(b) para cualquier $x = y$, reemplazar **x** por **y**;

(c) reemplazar $t = x$ por $x = t$;

(d) reemplazar $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$, $n \geq 0$, por **n** ecuaciones $s_i = t_i$, $1 \leq i \leq n$;

(e) reemplazar $f(.) = g(.)$ por falso (y así la colección entera de restricciones es insatisfactible);

(f) reemplazamos cada par de ecuaciones $x = t_1$, $x = t_2$ y decimos que t_1 no es mayor que t_2 para $x = t_1$, $t_1 = t_2$.

1.2.1.3 Vinculación

Dada la satisfacción **C**, tal que ninguna restricción protegida **G** sea vinculada por **C** (o una nueva restricción **c**), el problema a tratar es la determinación del subconjunto G_1 de **G** de las restricciones vinculadas por $C \wedge c$.

Una regla para determinar si un algoritmo de vinculación es incremental (en el sentido discutido anteriormente), es que su factor importante no sea el número de restricciones vinculadas después de un cambio en el almacenamiento, sino más bien, el número de restricciones no vinculadas. Esto es, el algoritmo tiene que ser capaz de ignorar las últimas restricciones de tal forma que el costo incurrido dependa únicamente del número de restricciones vinculadas, y no del número total de restricciones protegidas.

Como en el caso de la satisfactibilidad, la propiedad de vinculación es crucial para la implementación práctica de los sistemas **CLP**.

Sea **FT** el dominio y supongamos que **G** contiene únicamente restricciones protegidas de la forma $x = t$ donde **t** es el término aterrizado. Agregamos a la implementación estándar de un algoritmo de unificación una estructura indexada que trace variables **x** solo para aquellas restricciones protegidas en **G** las cuales implican **x**. Esto da lugar a un algoritmo incremental debido a que únicamente las restricciones protegidas que se esperan son aquellas $x = t$ para lo cual **x** se ha vuelto aterrizada, y no la colección entera de **G**[12].

En resumen, el problema de detectar la vinculación no se limita sólo al costo de determinar si una restricción en particular está vinculada. La incrementabilidad es

[12] Ejemplo tomado de [CF87].

crucial, y su propiedad puede definirse rigurosamente como la limitante del costo dependiendo del número de restricciones protegidas que se afecten a cada cambio en el almacén. En particular, tratar con la colección entera de restricciones protegidas cada vez que el almacén cambie es inaceptable.

1.2.1.4 Proyección

El problema consiste en obtener una representación útil de la proyec-ción de las restricciones **C**. Más formalmente: dadas las variables meta $\tilde{x}$ y las restricciones **C(**$\tilde{x}$ **,** $\tilde{y}$ **)** que implica variables de $\tilde{x}$ y $\tilde{y}$, expresamos que $\exists \tilde{y}$ **C(**$\tilde{x}$ **,** $\tilde{y}$ **)** en el modelo más utilizable.

Una fase importante en un sistema **CLP**, es la proyección de las restricciones respuesta con respecto a las variables meta. Otra fase es la meta-programación, donde la descripción del almacenamiento actual puede desearse para más manipulación. La proyección provee las bases lógicas para eliminar variables del conjunto acumulativo de restricciones, una vez que se conocen estas no se vuelven a referir de nuevo.

Existen pocos principios generales que guíen el diseño de algoritmos de proyección a través de varios dominios restrictivos. La razón es que, estos algoritmos tienen que estar íntimamente relacionados con el dominio a manipular.

Un algoritmo relativamente simple se puede obtener usando el método de eliminación de Gauss. Este asume que existe alguna categoría sobre las variables y asegura que las variables de menor prioridad se presenten en términos de variables de más alta prioridad. Esta categoría es arbitraria, excepto para el hecho de que las variables meta puedan ser de mayor prioridad que las otras variables.

Algoritmo de Proyección para Ecuaciones Lineales.

Sea m_1, …, m_n las variables meta;
para(i=1; i ≤ n; i=i+1) {
    si(m_i es un parámetro) **continua**;
    sea e la ecuación m_i = r.h.s.(m_i) a manipular;
    si(r.h.s.(m_i) contiene una variable s de menor prioridad que m_i) {
        selecciona la s de menor prioridad;
        reescribe la ecuación e dentro de la forma s = t;
        si(s es una variable meta) marca la ecuación e como final;
        substituye t para s en las otras ecuaciones;

} **delocontrario** marca la ecuación e como final
} **regresa** todas las ecuaciones finales.

Más detalles sobre este algoritmo se pueden encontrar en [JMS93].

1.2.1.5 Retroceso

El método aquí consiste en realmacenar el estado del resolvedor de restricciones a un estado previo (o al menos, a un estado equivalente). La técnica más común es el *arrastre* de restricciones cuando estas se modifican por el resolvedor de restricciones y el realmacenamiento de estas restricciones actualiza el retroceso.

En **Prolog**, las restricciones son ecuaciones entre términos, represen-tadas internamente como vínculos variables. Dado que las variables se implementan como apuntadores a sus variables ligadas, el retroceso se facilita por un mecanismo simple de *arrastre* sin etiqueta [DHD83] [HA91]. Este identifica al conjunto de variables, las cuales se han ligado desde el último punto de selección. Para actualizar el retroceso, estas variables simplemente se resetean para volverse no ligadas. Así en **Prolog**, la única información *arrastrada* es aquella en la cual las variables que se han vuelto a vincular, y el no *arrastrarlas* simplemente no liga estas variables.

Para **CLP** en general, es necesario **salvar** los cambios de las restricciones. Mientras en **Prolog** una variable simplemente se vuelve más y más instanciada durante la ejecución; en **CLP**, una expresión puede tomar otro valor diferente a su modelo original.

El *arrastre* de las expresiones es, en general, más costoso que el *arrastre* de las variables solas. Por esta razón, es útil evitar el *arrastre* cuando no hay un punto de selección entre las veces en que una variable cambia de valor de una expresión a otra.

Una técnica estándar para facilitar esto es el uso de ***marcas de tiempos***; una variable está en *marca de tiempo* con respecto al tiempo en que el último valor cambió, y cada punto de selección está también en *marca de tiempo* cuando este se crea. Justo antes de que cambie el valor de la variable (llamémosla **n**), su *marca de tiempo* se compara con la *marca de tiempo* **m** de la mayoría de los puntos de selección recientes, y si **n > m**, no es necesario hacer el *arrastre*.

Ahora consideramos el uso de una ***tabla de referencia cruzada*** para los modelos solución. Esta es una estructura indizada la cual traza cada variable paramétrica a una lista de ocurrencias en el modelo solución. Tal estructura es particularmente útil, y puede aún ser crucial en la eficiencia de los procesos para substituir una variable paramétrica. Sin embargo, su uso aumenta el problema del retroceso.
Un método directo es simplemente *arrastrar* las entradas en esta tabla. Sin embargo, debido a que estas entradas son en general grandes, y dado que la tabla de

referencia cruzada es redundante desde el punto de vista semántico, un método útil es **reconstruir** la tabla actualizando el retroceso. Este método de reconstrucción tiene la ventaja de incurrir en el costo únicamente cuando el retroceso de hecho toma lugar.

En resumen, el retroceso en **CLP** es substancialmente más complejo que en **Prolog**. Algunos de los conceptos a agregar en la máquina de **Prolog** son los siguientes: un *arrastre* de valores; *marcas de tiempos* para evitar repetidamente el *arrastre* de una variable durante el tiempo vida del mismo punto de selección; y finalmente, la reconstrucción de las referencias cruzadas, más que del *arrastre*.

1.2.2 Máquina de Inferencia

1.2.2.1 Diferir/Excitar metas y restricciones

El problema a resolver es, determinar cuando una meta diferida será excitada o cuando una restricción pasiva se vuelve activa. El criterio para tal evento se da por medio de una *restricción protegida* que es, excitar a la meta o activar a la restricción cuando se vincula a la *restricción protegida* por el almacenamiento.

El método de implementación fundamental, en lo que al resolvedor de restricciones respecta, es cómo procesar de manera eficiente, justo aquellas restricciones protegidas que se afectan como resultado de una nueva restricción introducida. Específicamente, para lograr la *incrementabilidad*, el costo de procesar un cambio al grupo actual de restricciones protegidas se puede relacionar con las restricciones protegidas afectadas por el cambio, y no con todas las restricciones protegidas. Los siguientes dos aspectos parecen haber logrado este fin.

El primer aspecto es la representación de la mayoría de las restric-ciones necesarias tal que una restricción dada se vincule. Una **restricción diferida** no se despierta por sí sola, sino por la conjunción de diversas restricciones de entrada. Cuando un subconjunto de tales restricciones de entrada se ha encontrado, el tiempo de corrida de la estructura relaciona la restricción diferida justo para las clases de restricciones que permanecen, las cuales serán excitadas.
El segundo aspecto, requiere indizar algunas estructuras a fin de permitir un acceso inmediato a las restricciones protegidas afectadas como resultado de una nueva restricción introducida. El principal logro será mantener tal estructura en presencia del retroceso. Por ejemplo, si los cambios de la estructura fueran *arrastrados* usando alguna adaptación de las técnicas de **Prolog** [DHD83], entonces se incurriría en un costo proporcional al número de entradas, no obstante el hecho de que se afecten las restricciones no protegidas.

1.2.2.2 Sistemas Excitados

Se define a una instancia de una restricción de la forma **p($1,…,$n)∧C** donde **p** es la n-ésima restricción simbólica a manipular, **$1,…, $n** son variables usadas como patrones para los argumentos de **p**, y **C** es una restricción (la cual determina los valores de **$1,...,$n**)[JMY91].

Un **grado excitado** representa un subconjunto de **p** restricciones, y un **sistema excitado** consiste de un conjunto de grados excitados; estos grados se organizan dentro de un autómata donde las transiciones entre los grados se etiquetan por restricciones llamadas **condiciones excitadas**.

Una transición ocurre cuando una condición excitada se vincula por el almacenamiento. Existe un grado llamado **excitar** (woken), el cual representa las **p** restricciones activas.

Considérese por ejemplo la restricción en **CLP®**, **pow(x,y,z)** (ver la figura 1). Un grado excitado se especifica por el significado de las restricciones contenidas en **$1, $2, $3**, las cuales describen a los tres argumentos, y algunas meta - constantes **#, #1, #2,** que describen los valores no especificados. Así, por ejemplo **$2 = #** especifica que el segundo argumento está aterrizado.

Tal meta - lenguaje puede usarse para especificar las condiciones excitadas. Así por ejemplo, la condición excitada **$2 = #, #<>0, #<>1** (ligada en la parte inferior de la figura), representa la transición de una restricción **pow($1, $2, $3) ∧ C ∧ c**, donde **C** no está aterrizada a **$2**, y donde **C ∧ c** no están aterrizados a **$2** con un número diferente de 0 y 1.

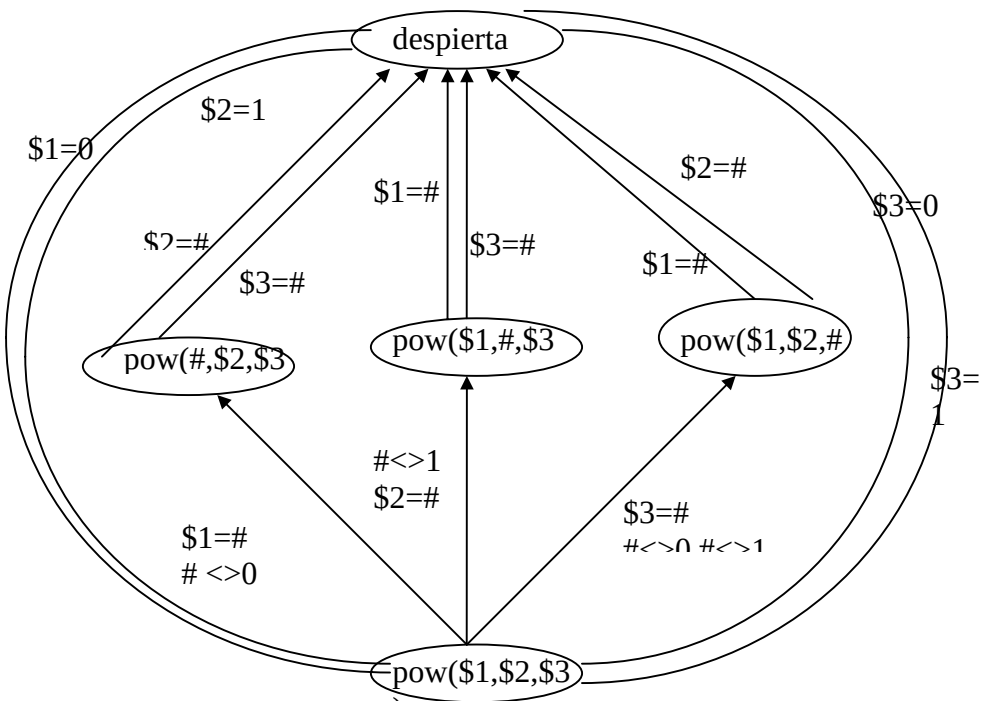


Figura 1. Sistema Excitado para pow (potencia) en CLP®.

La condición excitada **$2 = 1**, representa una transición al grado despierta, esta representa el hecho de que **pow($1, 1, $3)** es una restricción activa. Similarmente **$3 = 1** representa la transición **$1 = $2** activa. Nótese que no hay ninguna condición excitada para **$2 = 0**, ya que **pow($1, 0, $3)** no está activada. En general, existen ciertos requerimientos sobre la estructura de tal autómata para asegurar que define un trazado de **p** restricciones dentro de los grados excitados, y que éste trazado satisface ciertas propiedades tales como: definir una partición; trazar únicamente restricciones activas dentro de una excitación (woken); ser consistente con las condiciones excitadas que especifican transiciones.

En resumen, los sistemas excitados son un modelo intuitivo para especificar la organización de las restricciones protegidas. Los grados excitados representan los diferentes casos de una restricción diferida. Asociado con cada grado está un número de condiciones excitadas las cuales especifican cuando una nueva restricción cambia el grado de una restricción diferida. Así los sistemas excitados representan una parte importante en la implementación de las cláusulas protegidas, dado que un átomo protegido se puede ver como una cláusula protegida para un predicado anónimo.

1.2.2.3 Estructuras en el tiempo de corrida

Existen tres grandes operaciones con las metas diferidas o restricciones diferidas las cuales corresponden a las acciones de diferir, excitar y retroceder:

1. agregar una meta o restricción diferida a la colección actual de restricciones almacenadas;
2. despertar una meta diferida o restricción diferida como resultado de la introducción de una nueva restricción (activa);
3. realmacenar la estructura en el tiempo de corrida a un estado previo, esto es, realmacenar la colección de metas diferidas y restricciones diferidas, y por consiguiente realmacenar todas las estructuras auxiliares.

La primer estructura es una pila que contiene restricciones diferidas. Así para implementar la operación simplemente se requiere de una operación de insertar (push). En general, el grupo de restricciones diferidas contenidas en el sistema, se describen como el subgrupo de restricciones apiladas.

Para la implementación de la operación 2, es necesario tener algún acceso a la estructura y a la restricción vinculada para ajustar aquellas restricciones diferidas asociadas. Dado que hay en general un número infinito de posibles restricciones vinculadas, se requiere de una clasificación finita de ellas.

En una estructura indizada la cual traza una restricción diferida dentro de una lista doblemente enlazada de nodos de ocurrencia, cada nodo apunta a un elemento de la pila conteniendo una restricción diferida. Correspondiente a cada nodo de ocurrencia se tiene un apuntador de reversa del elemento de la pila al nodo de ocurrencia. Llamaremos a esta lista asociada con una restricción diferida DW una **lista DW**, y llamaremos a cada nodo en la lista un **nodo de ocurrencia DW**. Inicialmente la estructura de acceso está vacía.

A continuación se detallan cada uno de los mecanismos usados:

Diferir (delay). Inserta la restricción **C** dentro de la pila, y para cada condición excitada asociada con **C**, crea la protección correspondiente y la lista DW. Todos los nodos de ocurrencia aquí están apuntando a **C**.
Vinculación de procesos. Si se vincula **x = 5**, encontrar todas las protecciones que se implementen por **x = 5**. Si no hay ninguna, el proceso está completo. De lo contrario, para cada lista DW, **L** corresponde a cada una de las condiciones, y para cada restricción **C=p(.)** $\wedge$ **C'** apuntando a **L**,

(a) se borran todos los nodos de ocurrencia apuntando a **C**
(b) se inserta la nueva restricción diferida **C" = p(.)** $\wedge$ **C'** $\wedge$ **x = 5** con un apuntador a **C**, y finalmente
(c) se construye la nueva lista **DW** correspondiente a **C"** para la operación de retraso.

Retroceso. Realmacenar la pila durante el retroceso es fácil dado que este solamente requiere de una serie de salidas (pops). Realmacenar la lista de la estructura, sin embargo, no se hace directamente debido a que se ejecuta el *arrastre*/*salvación* de los cambios. La operación de retroceso es la siguiente:

(a) se sale de la pila y se deja que **C** denote la restricción recién sacada.
(b) se borran todos los nodos de ocurrencia apuntados por **C**. Si no hay apuntador desde **C** a otra restricción más adentro en la pila (o la restricción no está diferida), entonces no se necesita hacer nada más.
(c) si hay un apuntador de **C** a otra restricción **C'**, entonces se ejecutan modificaciones al acceso de la estructura como si **C'** estuviera siendo metida a la pila.

Estas modificaciones implican calcular las protecciones pertinentes para **C'**, insertando nodos de ocurrencia y colocando los apuntadores de reversa. La figura 2

ilustra una estructura al tiempo de corrida después de ejecutar dos restricciones **pow(x, y, z)** y **pow(y, x, y)** almacenadas en este orden. La figura 3 ilustra una estructura después de que una nueva restricción hace que **x = 5** se vincule.

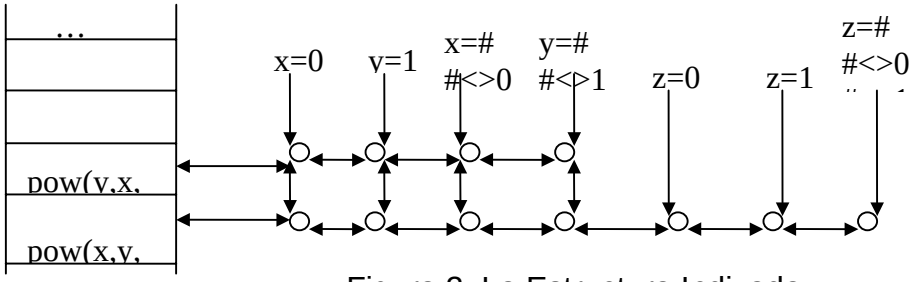


Figura 2. La Estructura Indizada

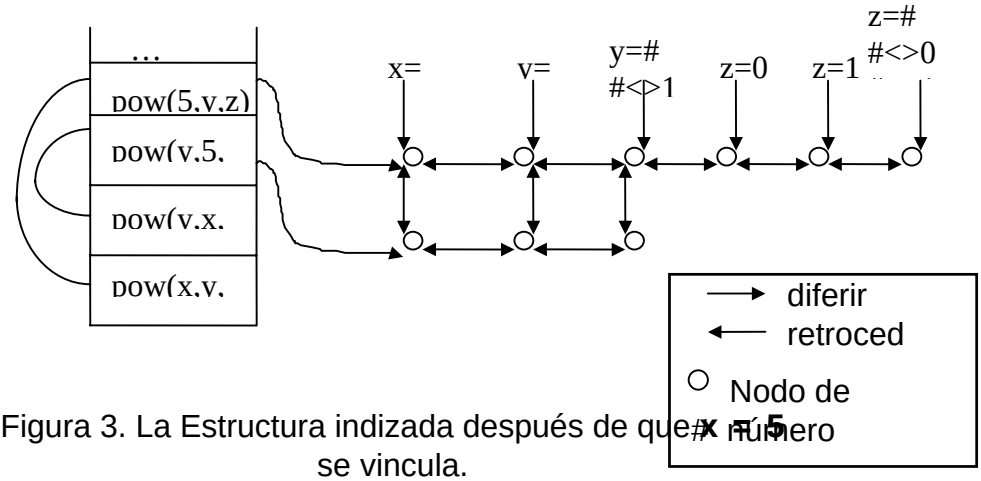


Figura 3. La Estructura indizada después de que **x = 5** se vincula.

En resumen, una pila se usa para almacenar restricciones diferidas y sus formas reducidas. Un acceso a la estructura traza un número finito de protecciones a la lista de restricciones diferidas. El resolvedor se asume idéntico para aquellas condiciones las cuales se vinculan. El costo de una operación primitiva sobre restricciones diferidas se vincula por el tamaño (fijo) del sistema excitado. El costo total de operación, esto es, diferir una nueva restricción, procesar una restricción vinculada y retroceder sobre las restricciones diferidas, es proporcional al número de restricciones diferidas afectadas por la operación.

Hasta aquí se han tratado las características más importantes con respecto a la **programación lógica por restricciones**, sin embargo su estudio no estaría completo si no se adentrara en su clasificación y desarrollo. En el siguiente capítulo se hace mención de 3 tipos de sistemas desarrollados bajo la filosofía **CLP**.

2

Sistemas Existentes

Al surgir la **Programación Lógica por Restricciones**, diversos lenguajes basados en esta se han ido diseñando e implementando. Muchos de estos lenguajes tratan con restricciones aritméticas. El concepto central es que las **restricciones** se usen no solamente para representar la relación entre objetos, sino también para calcular valores basados en estas relaciones.

Este capítulo muestra algunos modelos de sistemas **CLP** existentes como son **CHiP**, **Prolog III** y **CLP®**. Primeramente se identifican los parámetros clave que caracterizan a un lenguaje por restricciones. Posteriormente, se muestran cada uno de los lenguajes a analizar, y finalmente, se muestra porqué **CLP®** conviene a la evolución de los lenguajes por restricciones dentro de los lenguajes de programación lógica de propósito general.

2.1 Clasificación de un Lenguaje de Programación por Restricciones

Un parámetro que caracteriza al lenguaje por restricciones se refiere a cómo se interpretan las restricciones que aparecen en un programa. Esto es, representar a las restricciones como tales o crear restricciones temporales en el tiempo de corrida. Clasificamos a las primeras como **restricciones estáticas**, a las últimas como **restricciones dinámicas**, y a copias en el tiempo de corrida de tales restricciones como **instancias**.

Por ejemplo, considerando cómo afecta la forma en que se modela un sistema físico como un circuito eléctrico, si solamente se tienen restricciones estáticas, las restricciones tendrían que escribirse para cada componente, sin embargo, la habilidad para programar con restricciones dinámicas permite definir una **restricción temporal** para cada tipo de componente, y entonces construir una copia para cada instancia del componente temporal llenado posiblemente con valores extras. Esto induce a la noción de restricciones instancias en el tiempo.

En THINGLAB [AB81], las restricciones son estáticas. El lenguaje de Steele [35] es ligeramente más general; este permite diseñar programas con **macro-restricciones** definidas. Sin embargo, también prohibe la recursión, así que, el número de instancias de cada restricción y su relación es fija.

Un segundo parámetro que caracteriza a un lenguaje por restricciones se refiere a cómo las restricciones afectan el control del programa. Más específicamente, cómo una colección de restricciones puede influir en la colección futura de más restricciones. En la mayoría de los lenguajes con restricciones, las restricciones mismas no tienen efecto sobre el control del programa. Por lo tanto, estos lenguajes sacrifican una ventaja clave de las restricciones usadas.

Un tercer parámetro se refiere a cómo el algoritmo es usado para solucionar restricciones - el **resolvedor de restricciones,** y la extensión para lo cual las restricciones se solucionan. Este punto se refiere al hecho de que muchos dominios liberan restricciones que son impracticables para la solución en general.

De este modo, para obtener un lenguaje con restricciones práctico y un sistema sobre tal dominio, es necesario seleccionar prudentemente una subclase de restricciones las cuales se solucionen de manera práctica. Ejecutar un programa con restricciones implica tanto la ejecución de la colección de restricciones como la determinación de si las restricciones son satisfactorias (así como construir un modelo para representar restricciones una vez que estas se determinan para su solución).

Existen muchos métodos para solucionar un conjunto dado de restricciones. Un método es la **propagación local**. Este es el único método para solucionar restricciones aritméticas en el trabajo de Steele [35,34], y en sistemas Prolog incluyendo MU-Prolog [27] y NU-Prolog [39], entre otros.

Un sistema con restricciones se soluciona por propagación local si todas las variables en el sistema se determinan después de un número finito de pasos. Un **paso de propagación local** ocurre cuando una restricción ha determinado un número suficiente de variables para alguna de sus otras variables a ser determinadas. Estas variables ya determinadas nuevamente pueden precipitarse más adelante en pasos de propagación local para otras restricciones.

En general, sistemas que solucionan restricciones implican ciclos inter-dependientes que requieren del uso de técnicas más poderosas que la propagación local. Para solucionar restricciones aritméticas generales, las cuales pueden ser no lineales, se requieren de técnicas más poderosas, tales como símbolos algebraicos en paquetes como REDUCE[28] y MACSYMA[25]. Sin embargo estos paquetes son usual-mente lentos como para ser incorporados en un lenguaje de programación de propósito general. Un resultado crítico en sistemas con restricciones como EL/ARS[33,37] y SYN[37] (los cuales son usados con MACSYMA como su resolvedor de restricciones básico), fue el desarrollo de un retroceso inteligente para reducir la cantidad de trabajo efectuado por MACSYMA.

Aún cuando es posible aumentar cualquier lenguaje con características restrictivas, un hecho importante es cómo el lenguaje interactúa con dichas características restrictivas. En algunos lenguajes con restricciones esta interacción es tosca, pues requieren que el usuario les dé una gran cantidad de información acerca de cómo están siendo recolectadas y solucionadas las restricciones. En los lenguajes **CLP**, esta interacción es clara.

2.1.1 Parámetros Clave

Con respecto a sus parámetros clave, los lenguajes **CLP** se carac-terizan como sigue:

- Estos aplican reglas recursivas, que permiten instancias para cada restricción y cuya relación entre estas instancias se determina en el tiempo.
- El control de cálculo para solucionar restricciones es inherente al modelo operacional; dado que, a cada etapa, las restricciones se verifican para ver si son satisfactorias.
- El esquema **CLP** no tiene un algoritmo específico para solucionar restricciones; sin embargo, los sistemas **CLP** incorporan algoritmos más poderosos que aquellos basados en la propagación local.

El aspecto más importante de los lenguajes **CLP**, sin embargo, es que el esquema **CLP** define una clara interacción entre el esquema de la programación lógica y la forma en que las restricciones son utilizadas. Esto es, introduce técnicas para la solución de restricciones en la programación lógica (LP) con el uso de algunas herramientas matemáticas como Simplex para resolver restricciones numéricas, y el uso de técnicas de verificación de consistencia y propagación de restricciones para resolver restricciones simbólicas.

Algunos lenguajes basados en el esquema **CLP** son:

- **CHiP**, el cual es un lenguaje lógico que combina los aspectos declarativos de la programación lógica con la eficiencia de las técnicas de solución de restricciones. Este se ha diseñado para resolver problemas restrictivos en el dominio finito.
- **Prolog III**, el cual usa un algoritmo como Simplex para resolver ecuaciones lineales; y provee un método de saturación para tratar con términos booleanos.
- **CLP®**, el cual manipula ecuaciones lineales y no lineales en el dominio de los números reales.

A continuación se presentan con más detalle cada uno de estos lenguajes.

2.2 El lenguaje **CHiP** (Constraint Handling in Prolog)

Muchos problemas en la vida real como prevención, localización, distribución, diagnósticos de faltantes y verificación de hardware, se pueden ver como problemas de **búsqueda por restricciones**. La mayoría de ellos son del tipo de problemas **NP-completos** [GJ79]. El método más común para solucionar estos problemas consiste en escribir programas especiales en lenguajes estructurados, esto requiere de esfuerzos substanciales en su desarrollo, y los programas resultantes son muy difíciles de mantener, modificar y extender.

CHiP (Constraint Handling in Prolog **-** Manipulación de Restricciones en Prolog) intenta sobrellevar las dificultades de los problemas NP-completos, proveyendo además de la eficiencia de los métodos proposicionales, la principal característica de las herramientas de la quinta generación: la **declaratividad** y la **flexibilidad**. Este es un lenguaje de programación lógica como **Prolog** ampliado por las técnicas de solución de restricciones numéricas y simbólicas.

2.2.1 Dominios de Cálculo

CHiP se basa en el concepto del uso **activo** de restricciones [HG85] [PVHD86][MD86]. Difiere de los lenguajes lógicos usuales en dos aspectos: el **dominio de cálculo** sobre el cual trabaja, y la **manipulación de restricciones** y los procedimientos de búsqueda mejorados que provee.

El poder de un lenguaje de programación lógica como **Prolog** radica en tres mecanismos: **modelo racional**, **unificación** y **cálculo no determinista**. **Prolog** lleva el cálculo en el universo de Herbrand. La **unificación** se usa para resolver ecuaciones en este universo (sobre términos **no interpretados**). Por lo tanto, cuando se modela un problema, se tiene que hacer un mapeo del dominio dirigido al universo de Herbrand, esto causa la pérdida no solamente de la naturaleza de la expresión del problema sino también de la eficiencia de esta resolución. **CHiP** da lugar a dominios de cálculo más ricos que el universo de Herbrand, dado que permite manipular términos más expresivos, esto implica ampliar la unificación con algunos símbolos funcionales; como se sabe, las restricciones en **Prolog** se usan de una manera **pasiva** a través del paradigma **genera/examina** el cual causa ineficiencia en los problemas por restricciones, la ampliación consiste entonces en introducir técnicas que resuelvan restricciones en dominios de cálculo especializados.
La pregunta que da lugar en este punto es: ¿qué **dominio de cálculo** se tiene que introducir (con sus correspondientes técnicas) en la programación lógica para solucionar restricciones?. Tres grandes criterios para esta opción son: el **poder expresivo** del dominio de cálculo, la existencia de **técnicas eficientes** para resolver restricciones y su beneficio para posibles **aplicaciones**.

Siguiendo estos criterios, se seleccionaron tres dominios de cálculo para **CHiP**:

- términos restrictivos en el dominio finito,
- términos booleanos,
- términos de la aritmética racional lineal.

Para cada uno de estos dominios de cálculo, **CHiP** usa algoritmos especializados. Mientras que las técnicas de verificación de consistencia se usan para los dominios finitos, herramientas matemáticas (como la solución de ecuaciones en el álgebra booleana y un algoritmo simbólico como Simplex) se usan para la aritmética racional y booleana. Nótese que estas extensiones no afectan al aspecto declarativo de la programación lógica.

2.2.1.1 Dominios Finitos

Dominios Variables

La característica básica de **CHIP** para solucionar problemas combina-torios discretos, es su habilidad para trabajar sobre dominios variables, (variables que tienen un rango sobre dominios finitos) [PVHD86].

CHiP difiere entre dos clases de tales variables: aquellas que se clasifican sobre las constantes, y aquellas que se clasifican sobre un conjunto finito de números naturales. También tiene la habilidad de tratar con términos aritméticos sobre dominios variables, a partir de números naturales y operadores +, -, * y /.

2.2.1.2 Restricciones sobre dominios finitos

CHiP provee una gran variedad de restricciones sobre dominios variables. Este no contiene únicamente **restricciones aritméticas** sino también **restricciones simbólicas** y aún **definidas por el usuario**.

Restricciones aritméticas

En lo que concierne a restricciones aritméticas, **CHiP** permite las relaciones usuales entre los términos aritméticos sobre dominios variables. Por ejemplo, para cualquier término X y Y,

X > Y, X >= Y, X < Y, X =< Y, X = Y, X ~= Y[13]
son restricciones bien formadas de **CHiP**.

Restricciones simbólicas

[13] **X** diferente de **Y**.

CHiP no se restringe a las restricciones aritméticas, sino también contiene (y es parte de su originalidad), restricciones simbólicas sobre dominios variables. Ejemplos de algunas restricciones simbólicas (aparte de **X ~= Y**) son:

- **element(Nb, List, Var)** el cual se mantiene si **Var** es el **Nb**-ésimo elemento de **List**; esta restricción se puede usar cuando **Nb** y **Var** son dominios variables o constantes, y **List** es una lista de estos.
- **alldistinct(List)** el cual se mantiene si todos los elementos de **List** son diferentes; esta restricción se puede usar si la lista tiene elementos que son dominios variables o constantes.

Las restricciones de este tipo, son esencialmente herramientas para solucionar muchos problemas combinatorios discretos. Por ejemplo, la restricción "**element**" se puede usar para forzar una relación entre dos variables. Esta permite declarar problemas de forma natural y dar solución a los problemas más eficientemente.

Restricciones definidas por el usuario

No todas las restricciones se pueden proporcionar como primitivas en un lenguaje por restricciones. Es por lo tanto importante, permitirle al programador definir sus propias restricciones. En **CHiP**, es posible especificar que un predicado en particular se ha manipulado como una restricción usando **técnicas de consistencia**. El único requisito para que un predicado se manipule como una restricción es que sus instancias sean fallas finitas o exitosas. **CHiP** es el único lenguaje lógico por restricciones que permite restricciones definidas por el usuario.

Extensiones de orden mayor

CHiP también incluye algunas extensiones de orden mayor para encontrar soluciones optimizando alguna función de evaluación (minimización/maximización). Estos predicados pueden usarse para solucionar problemas de optimización combinatoria. Por ejemplo, el predicado:

minimize(Meta, Función)

donde **Meta** es un predicado y **Función** un término aritmético sobre dominios variables que se usa para encontrar la solución de **Meta** que minimice a **Función**. Esto se implementa usando una técnica de **ramifica** y **salta**.

2.2.2 Técnicas de consistencia

Todas las restricciones que implican dominios variables se resuelven a través de **técnicas de consistencia**, un paradigma poderoso que emerge de la inteligencia artificial para resolver problemas combina-torios discretos [UM74][AM77][HE80].

El principio en que se basan estas técnicas es el uso de restricciones para reducir los dominios variables y así el tamaño del espacio de búsqueda. Diferentes clases de podas (reducción de dominios) se han identificado y formas eficientes para lograrlo se han ideado. Sin embargo, las técnicas de consistencia no son capaces de resolver las restricciones por sí mismas, para solucionar un problema combinatorio discreto por ejemplo, se iteran los siguientes dos pasos:

- se propagan las restricciones tanto como sea posible,
- se crea una opción hasta que una solución se alcance.

Es también válido mencionar que las restricciones se solucionan a través de técnicas de consistencia como programadas en un modelo de manejo de datos. Un esquema de cálculo que implanta técnicas de consistencia dentro de la programación lógica se define en [PVH87] [PVH-87]. Este consiste en tres reglas de inferencia: la regla de inferencia de verificación hacia adelante (**FCIR**); la regla de inferencia de búsqueda hacia adelante (**LAIR**) y la regla de inferencia de búsqueda parcial hacia adelante (**PLAIR**), cada una definiendo una forma particular de poda.

Todas las restricciones primitivas sobre los dominios variables en **CHiP** se pueden ver como aplicaciones eficientes de estas reglas de infe-rencia. Por ejemplo, X~=Y se puede ver como una aplicación de **FCIR**. Otras relaciones aritméticas en términos aritméticos sobre dominios variables se pueden ver como aplicaciones en **PLAIR** dado el significado del razonamiento sobre variación de intervalos. Por ejemplo, dada la restricción,

R + E + 1 = 10 + T

con **R** $\in$ **{0,1}** y **E** y **T** $\in$ **{0, 2, 3, 4, 5, 6, 7, 8, 9}** inferirá que **T=0** y **E** $\in$ **{8, 9}.**

Las reglas de inferencia son la base de los **mecanismos de control general** para manipular restricciones definidas por el usuario. En resumen, los dominios variables son una extensión significativa para la programación lógica y las técnicas de consistencia ya que propor-cionan un paradigma uniforme y eficiente para solucionar restricciones sobre los dominios finitos, no importando si las restricciones son aritméticas, simbólicas o definidas por el usuario.

2.2.3 Unificación Booleana

La **unificación booleana** resuelve simbólicamente ecuaciones sobre **términos booleanos**. Se proponen diferentes algoritmos para la unificación booleana en

[BS87][MN86]. En esta sección, se usa el algoritmo de eliminación variable de [BS87].

2.2.3.1 Términos Booleanos

Dos formas de representar a los términos booleanos son la represen-tación interna y externa. En la **representación externa**, los términos booleanos se crean de valores ciertos (0 y 1), a partir de constantes (átomos), variables y operadores booleanos & (and), ! (or), # (xor), nand, nor, not. Las **constantes** denotan valores de entrada simbólicas, las **variables** denotan valores de salida o intermedios. Internamente, los **términos booleanos** se representan como gráficas acíclicas directas.

2.2.3.2 Uso de la unificación booleana

Para decirle al sistema qué argumentos deseamos usar en la unifica-ción booleana, se tienen que expresar **declaraciones booleanas** para los predicados. Expresando una declaración

```
?- declare and(h, h, bool, bool, bool).
and(M, N, X, Y, X & Y).
```

para el predicado and, se usa la unificación normal sobre los términos de Herbrand en los dos primeros argumentos, y la unificación booleana para los tres últimos argumentos. El predicado declara que el último argumento tiene que ser igual a la lógica and del tercer y cuarto argumento.

Como un ejemplo sencillo del uso de la unificación booleana en **CHiP** se verifica una compuerta **Xor** (figura 1). La descripción del circuito se explica con más detalle en [SD87][SND88]:

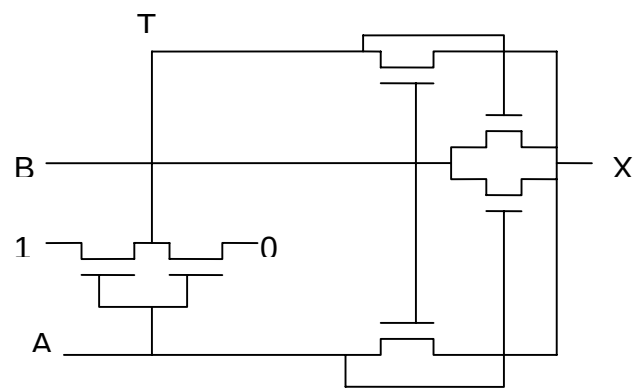


Figura 1: Diagrama del circuito de la compuerta Xor

```
? declare eq(bool, bool).
eq(X, X).

? declare n_switch(bool, bool, bool).
n_switch(Drain, Gate, Source)  :- eq(Drain & Gate, Gate & Source).

? declare p_switch(bool, bool, bool).
p_switch(Drain, Gate, Source) :- eq(Drain & not(Gate), not(Gate) & Source).

? declare xor(bool, bool, bool).
xor(A, B, X) :- p_switch(1, A, T1),
                n_switch(0, A, T1),
                p_switch(B, A, X),
                n_switch(B, T1, X),
                p_switch(A, B, X),
                n_switch(T1, B, X).
```

Si se desea calcular simbólicamente la salida de este circuito, se tiene que solucionar la ecuación booleana para cada interruptor. Esta instancia a las variables X y T1 para algunos términos booleanos. Se muestran los valores de X y T1 después de cada paso. Las variables que inician con un **'_'** (underscore) son variables libres introducidas por la unificación booleana.

```
?- xor(a, b, X).

1) T1= 1 # a #  _A & a
2) T1= 1 # a
3) X = b #  _C & a  #  a & b
4) X = b #  _C & a  #  a & b
5) X = a # b #  _D & a & b
6) X = a # b

X = a # b
```

Al final del cálculo, vemos al valor **X = a # b** como resultado.

2.2.4 Aritmética Racional

Consideremos ahora la parte de **CHiP** que manipula problemas continuos (problemas donde hay un número infinito de puntos en el espacio de búsqueda), modelándolos a través de números racionales.

2.2.4.1 Términos racionales

Los términos racionales son términos lineales sobre números y varia-bles racionales (variables las cuales toman sus valores en los números racionales).

Las relaciones aritméticas usuales se pueden definir sobre términos racionales. Dados dos términos racionales **X** y **Y**, las siguientes restricciones

X > Y, X >= Y, X < Y, X =< Y, X = Y, X ~= Y

son todas restricciones bien formadas en **CHiP**. Debido a la restricción impuesta sobre los términos racionales, únicamente las restricciones lineales son posibles.

2.2.4.2 Solución de términos

CHiP provee un procedimiento de decisión para restricciones sobre términos del tipo racional lineal. Esto significa que para cualquier conjunto de restricciones (sobre términos racionales) **CHiP** puede decidir si estos son satisfactibles o no. Este procedimiento es una adap-tación del algoritmo Simplex [GD63][SG85] basado en la eliminación de variables y no sobre la manipulación de matrices.

Dado un conjunto de restricciones; el procedimiento puede tanto fallar (las restricciones no son satisfactibles), como producir un conjunto de ligaduras para las variables y un conjunto de restricciones en un modelo simplificado normal. Desde el punto de vista de la implemen-tación, este procedimiento tiene ciertas propiedades deseables:

Integración: El solucionador de restricciones se establece por completo en el lenguaje **CHiP** y no es un módulo separado para el cual las restricciones se transmiten y los resultados se buscan.

Variables no fijas: El algoritmo garantiza que todas las variables que aparecen en los términos racionales puedan tomar un número infinito de valores. Si una variable puede tomar sólo un valor (en este caso decimos que la variable está fija), el procedimiento asigna directa-mente este valor a la variable. Esta propiedad permite decidir restricciones eficientes de la forma **X ~= Y**.

Incrementabilidad: El procedimiento es incremental. Si se tiene un conjunto de restricciones S, el agregar una nueva restricción C a S no requerirá tener que solucionar a partir de cero al conjunto $S \cup C$. **CHiP** transformará la solución de S

dentro de una solución S ∪ C, si esta existe. Esta propiedad es de gran importancia dado que las restricciones en **CHiP** se crean dinámicamente.

Uniformidad: Las ecuaciones y desigualdades se resuelven de una manera uniforme al introducir variables de **poca actividad**. Contraria-mente al algoritmo Simplex, no es necesario retransformar igualdades dentro de las desigualdades introduciendo variables artificiales. Desde el punto de vista del usuario, la inclusión de este procedimiento dentro de **CHiP** ofrece también propiedades atractivas como:

> **Soluciones simbólicas:** Las soluciones devueltas son siempre más generales que las de cualquier otro sistema procedimental, dado que **CHiP** puede representar un número infinito de soluciones de manera finita.
>
> **Desigualdades estrictas:** El usuario puede expresar su problema no únicamente en términos de desigualdades, sino también en términos de desigualdades estrictas (X > Y). Este poder expresivo surge de la habilidad de **CHiP** para resolver restricciones de la forma X ~= Y.

Satisfactibilidad y Optimización: CHiP puede usarse tanto para decidir si un conjunto de restricciones es satisfactible, como para encontrar la solución más general a un conjunto de restricciones las cuales optimicen una función de evaluación lineal.

En resumen, muchos problemas a partir de campos como la Investiga-ción de operaciones y el Análisis de circuitos eléctricos caen dentro del esquema de la aritmética racional de **CHiP**, abriendo nuevas áreas de aplicación a la programación lógica. A continuación, se discuten algunas opciones del diseño básico de estas aplicaciones.

Algoritmos polinomiales vs. Simplex

Es asombroso el porqué **CHiP** usa algoritmos basados en Simplex y no los algoritmos de Khachian[LK79] y Karmarkar[NK84]. Estos algorit-mos tienen por supuesto la propiedad de interesarse en la complejidad polinomial en el peor de los casos, contrariamente al algoritmo Simplex. “Estudios experimentales han mostrado que Simplex se desarrolla muy bien en el caso promedio (este es casi lineal), reforzando su posición como una de las herramientas más importantes en la investigación de operaciones. El algoritmo de Khachian, fue el primer algoritmo polinomial en la programación lineal, induce a una generalidad inaceptable ya que parece tener un potencial mayor pero no es del todo claro al crear la incrementabilidad” [PVH88].

Números reales vs racionales

En **CHiP** se ha optado por incluir a la aritmética racional como en **Prolog III** [AC87] y no toda la representación de la aritmética racional de números de punto flotante

como en **CLP®** [JM87]. Esta opción se ha hecho dado que, para las restricciones lineales, la aritmética racional tiene la propiedad deseable de estabilidad numérica.

Los números racionales se pueden representar exactamente sobre una computadora mientras que los números reales se representan normal-mente por números de punto flotante que introducen errores de redondeo. Esto indica que trabajar con números racionales preserva la **validez** del sistema, mientras que no es del todo cierto si se trabaja con números de punto flotante. Por supuesto, trabajar con números racionales induce a algunas **desventajas** en términos del consumo de memoria y de la eficiencia del tiempo.

Términos no lineales vs lineales

Las restricciones no lineales no se han incluido en **CHiP**, la razón es que no hay un método general analítico disponible para solucionarlos. Los métodos numéricos interactivos son necesarios en la mayoría de los casos. Por lo tanto, sufren de insatisfactibilidad numérica y a la vez, no se adaptan muy bien con la filosofía de la programación lógica.

2.2.5 Construcción de Demonios (Demon)

2.2.5.1 Declaración "*Delay*"

CHiP contiene un mecanismo diferido, las declaraciones ***delay***, las cuales habilitan de alguna manera el manejo de demonios. Por ejemplo, el siguiente programa define un elemento almacenado del tipo *cerrojo* usado en la simulación de circuitos secuenciales que hace uso de una declaración *delay*. El significado de esta definición es diferir el llamado del predicado *cerrojo* hasta que el primer argumento no sea una variable.

```
?- delay cerrojo(nonvar, any, any, any, any).
cerrojo( [ ], [ ], [ ], [ ], _ ).
cerrojo( [ CLK | CLK1 ], [ Ld | Ld1 ], [D|D1], [Q|Q1],S):-
                                la1( CLK, Ld, D, Q, S),
                                cerrojo( CLK1, Ld1, D1, Q1, Q).
?- delay la1(nonvar, nonvar, any, any, any).
la1( CLK, 1, D, D, S).
la1( CLK, 0, D, S, S).
```

2.2.5.2 Propagación Local

En la propagación local, las restricciones se usan para encontrar los valores de las variables no instanciadas a partir de variables instan-ciadas. Esta técnica ha sido

usada ampliamente en Inteligencia Artificial; el lenguaje por **restricciones** de Sussman y Steel [SS80] se basa en este paradigma de cálculo, mientras que muchos algoritmos en el área del diseño de hardware y sistemas gráficos se basan en principios similares a los de [AB81][JK76][RD84]. La **propagación local** se puede implementar a través de mecanismos diferidos; sin embargo, esta solución no es ni elegante ni eficiente dadas las características de cada uno de ellos. En **CHiP** la propagación local se logra a través de un mecanismo general llamado **demonio**. El siguiente programa ilustra su uso en un circuito lógico y una compuerta.

```
?- demon and/3.
and( 0, Y, Z )   :-  Z = 0.
and( Y, 0, Z )   :-  Z = 0.
and( 1, Y, Z )   :-  Z = Y.
and( X, 1, Z )   :-  Z = X.
and( X, Y, Z )   :-  Z = Y.
and( X, Y, 1 )   :-  X = 1, Y = 1.
and( X, Y, 0 )   :-  one_zero( X, Y ).

? demon one_zero/2.
one_zero( 0, _ ).
one_zero( _, 0 ).
one_zero( X, 1 ) :- X = 0.
one_zero( 1, Y ) :- Y = 0.
```

Los predicados sometidos a las declaraciones **demonio** se usan como reglas re-escritas. Estos son determinísticos y una meta se resuelve por sí sola cuando se iguala (en el sentido de la unificación de un solo camino) a una cabeza en una de las cláusulas definidas. Por ejemplo, meta ← and(1,0,Z) se resuelve con la segunda o tercera cláusula, una de estas cláusulas se detecta y la meta se escribe como Z = 0.

2.2.5.3 Propagación Condicional

La propagación condicional es una técnica de propagación manejada por la satisfactibilidad o insatisfactibilidad de las restricciones. Declara-tivamente, esta es una simple construcción **if_then_else.**

if condición **then** meta1 **else** meta2

El procedimiento provee un mecanismo de manejo de demonios eficiente, esta es una extensión del **if_then_else** de Mu-Prolog [LN84] y se manipula de igual manera.

Para cualquier condición permitida, **CHiP** hace uso de un procedi-miento capaz de decidir si la condición es siempre cierta o siempre falsa para todas las instancias de la condición o si la condición es cierta para algunas instancias y falsa para otras. Por lo tanto, mostrando tal restricción **CHiP** usa el procedimiento adecuado para evaluar la **condición**; si esta siempre evalúa a cierto, **meta1** se ejecuta; si evalúa a falso, **meta2** se ejecuta; de lo contrario, la construcción **if_then_else** se difiere, en espera de más información.

Cualquier restricción sobre los dominios variables, términos booleanos y términos racionales pueden crear una condición permitida. La propagación condicional ha sido una herramienta importante para la simulación en la industria de circuitos electromecánicos y aplicaciones en el razonamiento cualitativo.

2.2.6 Aplicaciones en CHiP

Diferentes problemas del mundo real se han resuelto en **CHiP**. Algunos de ellos previamente resueltos en un lenguaje procedimental requiriendo de un esfuerzo y un tiempo de desarrollo grandes. **CHiP** reduce drásticamente este tiempo logrando una eficiencia similar. La riqueza de las aplicaciones muestran la flexibilidad de **CHiP** para adaptarse a diferentes tipos de problemas. A continuación se presentan sólo algunos de ellos.

2.2.6.1 Aplicaciones en planeación y prevención

La Investigación de Operaciones (I.O) es una fuente interminable de problemas de investigación interesantes. Muchos problemas de I.O se han solucionado en **CHiP**, especialmente problemas de planeación y prevención a gran escala.

Sucesión de Carros: Este problema ocurre en los inventarios para la línea de ensamblaje en la manufactura de un carro. Cada carro requiere de un conjunto diferente de opciones, y la línea de ensamblaje, de las restricciones de capacidad para tales opciones. El problema consiste en generar una sucesión de carros que satisfagan las restricciones de capacidad.

Asignación de Tráfico Óptimo para Satélites: Este problema concierne a la prevención de sistemas de interrupción a bordo de satélites de tele-comunicaciones. El problema se formula como: dada una intersección en la matriz tráfico, determinar los modos de interrupción sucesiva para todos los requerimientos de tráfico en un tiempo mínimo.

Planeación de Inversiones: El programa selecciona entre diferentes tipos de inversión para minimizar o maximizar una función meta sobre un período dado. Gracias al método Simplex, el programa produce la solución más general y el usuario puede

entonces interactuar con él consiguiendo la mejor solución con respecto a sus necesidades.

2.2.6.2 Aplicaciones en el diseño de circuitos

Otro dominio de aplicación muy promisorio para **CHiP** es el diseño de circuitos. **CHiP** hace posible resolver grandes clases de problemas para grandes circuitos.

Simulación de Circuitos: Los mecanismos de propagación de restricciones y demonios en **CHiP**, permiten la simulación de grandes circuitos secuenciales y combinatorios (con varios cientos de compo-nentes).

Diagnósticos de Falla: El problema consiste en localizar un componente defectuoso en un circuito a partir de fallas en la entrada y salida. Se hace una sola suposición de la falla, el diagnóstico se basa en un modelo de razonamiento para encontrar la falla usando un método de propagación de restricciones combinado con una técnica de etiquetado consistente.

Simulación de dispositivos electromecánicos: Para analizar circuitos híbridos, se desarrolló en **CHiP** un simulador cuantitativo usando modelos de dispositivos analógicos. El sistema se aplica exitosamente en circuitos del mundo real llegando hasta la industria de aeroespacio.

Otras áreas de aplicación de **CHiP** son:

- planeación de recursos,
- localización estratégica de almacenes,
- problemas de recorte de existencias,
- prevención en la línea de ensamblaje,
- análisis y planeación financiera.

Más información acerca de estos problemas se puede encontrar en [SD-87][SD87][SDN88][PVH-87][DSH][DSH88].

2.3 Prolog III

Los primeros lenguajes que ajustaron con el esquema **CLP** - aunque no fueran descritos como tales - fueron presentados por Colmerauer. Este pionero trabajó en **Prolog** describiendo **Prolog II** [AC82], un sistema basado en la solución de ecuaciones y desigualdades sobre árboles infinitos. Posteriormente trabajó en el lenguaje **Prolog III** [AC88], donde otros problemas dominaron, tales como la aritmética lineal, aritmética booleana y cadenas. La nueva característica que surgió de **Prolog II** y **Prolog III** fue la existencia de programas con restricciones primitivas,

las cuales son un conjunto distinguido de predicados sobre el problema dominante. Desde el punto de vista teórico, la relación entre estos Prologs y **CLP** es simple: cada uno de estos lenguajes es una instancia del esquema **CLP**.

Prolog III obtiene mejoras con respecto a su predecesor teniendo en cuenta dos observaciones. Primero, la estructura de una lista; un árbol construido por el significado de un símbolo funcional binario, hace que las operaciones de concatenación sean extremadamente costosas. Segundo, cualquier acceso al **n**-ésimo elemento de una lista puede aprovecharse vía una ruta secuencial. Estas dos desventajas se eliminan al introducir restricciones sobre tuplas, equipadas con una operación de concatenación cuyas propiedades corresponden a lo que los usuarios esperan.

2.3.1 Árboles

Prolog III es un lenguaje completamente coherente y uniforme: Todos los objetos manipulados son **árboles**. Algunos son muy complejos o aún infinitos, mientras que otros pueden reducirse a un solo nodo. Algunos tienen un nombre particular, mientras que otros solamente pueden diseñarse usando una operación la cual los construya o un sistema de restricciones del cual ellos son solución.

Un **árbol** es visto como una **unidad jerárquicamente organizada**. Esta jerarquía abarca un conjunto de posiciones o nodos, y un conjunto de flechas las cuales enlazan estas posiciones. Los nodos localizados en la parte final de una flecha son referidos como los hijos del nodo **n**. El nodo al inicio de la jerarquía es llamado la **raíz** o nodo inicial del árbol. Un **árbol** incluye una **etiqueta** y una secuencia finita de subárboles.

Las etiquetas pueden ser:

- identificadores,
- caracteres,
- valores booleanos de 0 y 1,
- valores numéricos,
- el signo especial "**<>**"

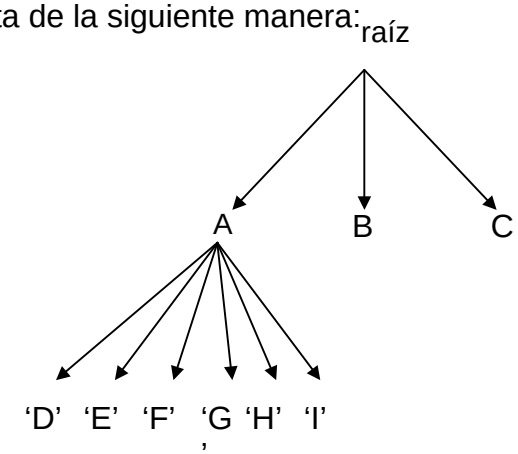
Se debe tener cuidado de no confundir los árboles con los términos. Los **árboles** son elementos en el dominio de **Prolog III**; los **términos** son construcciones sintácticas.

Los árboles cuya etiqueta inicial sea un identificador se llaman **árboles de hechos**, y aquellos cuya etiqueta inicial es el signo "**<>**" son llamados **tuplas**, cuyos elementos son llamados **cadenas**[14]. Las **tuplas** representan una herramienta flexible y

[14] En Prolog III, las cadenas se representan por comas invertidas (`) y las tuplas se representan por los paréntesis cuadrados (< Hasting,1066,1 >)

poderosa de definir estructuras de datos, al combinar la generalidad de las listas y la eficiencia de los vectores.

El número de hijos en un nodo es siempre finito e independiente de la naturaleza de su etiqueta. El número de hijos del nodo puede ser cero, cuyo nombre será **hoja**. **Prolog III** no distingue entre una hoja y las etiquetas que acarrea. Consecuentemente los identificadores, carac-teres, valores booleanos y numéricos, son considerados como árboles de tipo especial. Un árbol en **Prolog III** se representa de la siguiente manera:



Constantes

Las **constantes** son elementos en el dominio de **Prolog III** las cuales tienen nombres especiales. Una hoja no es necesariamente una constante: un árbol reducido a un simple número fraccional como **755/10** es una hoja, pero este no tiene un nombre específico y solamente puede expresarse por el significado de la división de la cual se produjo. Consecuentemente, no todas las constantes representan necesariamente hojas.

Identificadores

Un **identificador** es una secuencia que contiene letras, dígitos y los caracteres (**'**) y (_), no tienen la sintaxis de un nombre de variable (inicia con al menos dos letras).

Caracteres

Todos los **caracteres** conocidos se pueden usar en un programa de **Prolog III**. Estos se pueden expresar encerrados por (**'**) : **'A', 'a', '<',** los caracteres no imprimibles y aquellos que tienen un papel especial se pueden indicar usando el caracter de escape (****).

Valores booleanos

Los **valores booleanos** son los dos elementos del álgebra booleana: **0** y **1**, estos no se interpretan como cierto o falso, dado que estos términos se usan para describir restricciones (las restricciones son evaluadas en cierto o falso para indicar si son o no satisfechas).

2.3.2 Operaciones sobre Árboles

Para denotar árboles que no son constantes se tiene que escribir alguna fórmula que combine las constantes y los operadores de acuerdo a una sintaxis. Estas fórmulas expresan el resultado de una **operación**. Una **operación** se define sobre un conjunto de tres tuplas asociando un árbol a cada una de esas tuplas:

$$f:(a_1,\ldots,a_n)\longrightarrow f(a_1,\ldots,a_n)$$

Existen tres tipos de operaciones: **booleanas, aritméticas** y **árboles de construcción**. Un punto esencial aquí es que las operaciones booleanas y aritméticas tienen su usual significado matemático.

En otros lenguajes, las operaciones escritas con **+, *, &**, pueden estar definidas; sin embargo no tienen un significado matemático; estas son simplemente operaciones entre árboles de construcción. Así en **Prolog II**, la fórmula **2 + 3** denota un árbol cuya etiqueta es **+** y los hijos son **2** y **3**. Un predicado construido puede entonces interpretar este árbol y extraer el número **5** de él, pero esto es extraño para **Prolog II**, el cual no ejecutó algún proceso específico. En **Prolog III** sin embargo, las operaciones aritméticas y booleanas son conocidas como el núcleo del lenguaje, el cual da a estos un significado lógico o aritmético.

2.3.2.1 Operaciones Booleanas

Las operaciones booleanas se definen si los operandos son valores booleanos (hojas etiquetadas por constantes booleanas). Estas son:

(1) **not :** $a_1 \rightarrow \neg a_1$
(2) **and :** $(a_1, a_2) \rightarrow a_1 \& a_2$
(3) **or** (no exclusivo) : $(a_1, a_2) \rightarrow a_1 | a_2$
(4) **implica :** $(a_1, a_2) \rightarrow a_1 \Rightarrow a_2$
(5) **equivalente :** $(a_1, a_2) \rightarrow a_1 \Leftrightarrow a_2$

2.3.2.2 Operaciones Aritméticas

Las operaciones aritméticas solamente se definen si los operandos son valores numéricos (las hojas se etiquetan con números). Si ninguno de los números es número flotante, la operación se considera una operación exacta sobre números racionales. Si al menos uno de los operandos es flotante, entonces el otro se transforma en un número flotante también (a menos que ya lo fuera), y la operación se considera una operación correspondiente a los números flotantes.

las operaciones aritméticas son:

(1) **neutral :** $a_1 \rightarrow +a_1$

(2) **cambio de signo** : $a_1 \rightarrow \sim a_2$

(3) **suma :** $(a_1, a_2) \rightarrow a_1 + a_2$

(4) **substracción :** $(a_1, a_2) \rightarrow a_1 - a_2$

(5) **multiplicación**[15] **:** $(a_1, a_2) \rightarrow a_1 * a_2$

(6) **división**[16] **:** $(a_1, a_2) \rightarrow a_1 / a_2$

2.3.2.3 Operaciones sobre árboles de construcción

Las operaciones definidas sobre árboles en **Prolog III** son las **operaciones de construcción**. Como en **Prolog II**, aquí se encuentra un **árbol constructor**, pero hay dos nuevos constructores: la **tupla constructor** y el **constructor general de árboles**, los cuales se usan para denotar cualquier árbol de una manera totalmente genérica. Estas operaciones son usadas para construir árboles no reducidos a hojas. Estas son:

(1) **árbol constructor :** $(a_1, a_2, \ldots, a_n) \rightarrow a_1(a_2, \ldots, a_n)$

solamente se define si **n ≥ 2** y $\mathbf{a_1}$ es una hoja.

(2) **tupla construcción:** $(a_1, \ldots, a_n) \rightarrow \langle a_1, \ldots, a_n \rangle$

se define si la tupla $\mathbf{(a_1, \ldots, a_n)}$ donde **n≥1**. **n** se dice que es la **longitud** de la tupla construida. La siguiente igualdad se deriva de la definición misma de las tuplas:

$$\langle a_1, \ldots, a_n \rangle = <> (a_1, \ldots, a_n)$$

(3) **constructor general de árboles :** $(a_1, a_2) \rightarrow a_1[a_2]$

---

[15] para aligerar la notación, Prolog III permite $\mathbf{a_1 a_2}$ en lugar de $\mathbf{a_1 * a_2}$.

[16] la división solamente se define si a2 no es cero.

esta operación solo se define si $\mathbf{a_1}$ es una hoja y $\mathbf{a_2}$ es una tupla. Por definición se tiene la siguiente igualdad:

$$a_1\left[\left\langle b_1,\ldots,b_m\right\rangle\right]=a_1\left(b_1,\ldots,b_m\right)$$

El constructor general de árboles se usa para representar cualquier árbol usando únicamente su etiqueta inicial y la tupla formada por sus hijos inmediatos[17].

(4) **concatenación de tuplas :** $\left(a_1,a_2\right)\rightarrow a_1\cdot a_2$

solamente se define si $\mathbf{a_1}$ y $\mathbf{a_2}$ son ambas tuplas. Por definición se tiene la igualdad:

$$\left\langle b_1,\ldots,b_m\right\rangle\cdot\left\langle c_1,\ldots,c_k\right\rangle=\left\langle b_1,\ldots b_m,c_1,..,c_k\right\rangle$$

2.3.3 Variables

En esencia, las **variables** son simplemente otra forma de representar árboles. Desde el punto de vista sintáctico, la principal característica que distingue a una variable es que su nombre es una secuencia de letras, dígitos y los caracteres (**‘**) y (_), iniciando con una letra y algún caractér si cualquiera de ellos no es una letra.

Existe una profunda diferencia entre el significado de las variables en la mayoría de los lenguajes procedimentales y lo que representan en **Prolog III**. En los lenguajes procedimentales, un valor se liga a cada variable; en estos lenguajes, una variable siempre denota un objeto conocido o de lo contrario el uso de la variable es ilegal. Una variable en **Prolog III** representa árboles desconocidos.

2.3.4 Términos

Los **términos** son fórmulas que representan árboles en un programa. Su sintaxis permite todos los tipos de términos a escribir. Existen dos restricciones importantes con respecto a los términos en **Prolog III**. La primera se refiere a los términos encabezados por un operador aritmético: las expresiones aritméticas tienen que ser **lineales**; la segunda restricción se refiere a la operación de **concatenación**: en una concatenación, la longitud del operando a manipular tiene que ser conocida.

¿Que representan los términos?

Los **términos** representan árboles en los programas de **Prolog III**. Esto puede entenderse de dos maneras:

- **Un término determina un árbol parcialmente conocido**. Un término generalmente representa un árbol en el cual solo ciertos componentes son conocidos, y el objetivo del programa es revelar los otros componentes.
- **un término representa un conjunto de árboles.** El conjunto obtenido al dar todos los valores posibles a las variables que contiene.

[17] No se pueden insertar espacios entre los términos que representan la etiqueta del árbol y los paréntesis que abren del constructor. Esta es una de las raras ocasiones en Prolog III en la cual los espacios se prohiben.

Cuando se define una **asignación** del conjunto de variables, simplemente se selecciona un valor para cada variable pertinente. Si el conjunto es:

$$V = \{x_1, x_2, \ldots, x_n\}$$

entonces una **asignación** será el conjunto de pares **(x_i, a_i)**, escritos como $x_i \leftarrow a_i$ por lo tanto:

$$A = \{x_1 \leftarrow a_1, x_2 \leftarrow a_2, \ldots, x_n \leftarrow a_n\}$$

significa que el árbol a_1 da el valor de la variable x_1, el árbol a_2 el valor de la variable x_2, y así sucesivamente. Una **asignación** es por lo tanto el **mapeo** de un conjunto de variables en el dominio de **Prolog III**, definida por su extensión (elemento por elemento).

Denotemos a **t** como un término; cada asignación **A** del conjunto de variables que aparecen en **t** permiten que **t** se transforme en un árbol **a = t/A**. Esto es suficiente para reemplazar cada variable de **t** con su valor en **A**. Para definir **t/A** en cada modelo del término **t**, se tiene que

- si **t** se reduce a una constante **k**, entonces **a** es el árbol de **k**,
- si **t** se reduce a una variable **x**, entonces **a** es el árbol dado como el valor de **x** en la asignación **A**,
- si **t** tiene la forma $<t_1, t_2, \ldots, t_n>$ y si $a_1, a_2, \ldots, a_n$ son los árboles t_1/A y $t_2/A, \ldots, t_n/A$ entonces **a** es el árbol $<a_1, a_2, \ldots, a_n>$
- si **t** tiene la forma $t_1(t_2, \ldots, t_n)$ y si $a_1, a_2, \ldots, a_n$ son los árboles $t_1/A, t_2/A, \ldots, t_n/A$ entonces
  - si a_1 es una hoja, entonces **a** es un árbol $a_1(a_2, \ldots, a_n)$,
  - de lo contrario, **a** es **indefinido**.
- si **t** tiene la forma $t_1(t_2)$ y si a_1 y a_2 son los árboles t_1/A y t_2/A entonces
  - si a_1 es una hoja y a_2 una tupla, entonces **a** es el árbol $a_1[a_2]$,
  - de lo contrario, **a** es **indefinido**,

2.3.5 Tuplas

El doble signo "**<>**" es un símbolo convencional usado para etiquetar árboles los cuales no están etiquetados desde un punto de vista semántico. Estos árboles pueden así reducirse a la secuencia de sus hijos, en **Prolog III** estos implementan el concepto de una secuencia finita de árboles, los cuales se llaman **tuplas**.

Las **tuplas** se escriben como $<a_1, \ldots, a_n>$; la **tupla vacía** es por lo tanto escrita como **<>**. Con frecuencia, el número de hijos de una **tupla** se conoce como **tamaño**.

La única operación definida sobre las **tuplas** es la operación de **concatenación** denotada por un punto (.). Esta operación usa dos tuplas **T1** y **T2** cuyos tamaños respectivos son **n** y **m** para construir la tupla **T1.T2** de tamaño **m + n**, en la cual los primeros **n** hijos pertene-cen a **T1**, y los subsecuentes **m** hijos pertenecen a **T2**.

2.3.6 Relaciones

En **Prolog III**, las relaciones binarias y unarias expresan condiciones aplicadas a los árboles, tales como "**<aa, bb, cc>** es diferente de **aa(bb, cc)**" o " **1** es mayor que **0**".

Las relaciones no son operaciones. La característica distintiva de una **operación** aplicada a una **tupla** es producir un árbol; la característica distintiva de una **relación** aplicada a una **tupla** es ser o no verificada. Como las operaciones, las relaciones son **parciales**.

Las **relaciones** son:

Igualdad y Desigualdad : La condición $a_1 = a_2$ se lee como "el árbol a_1 y a_2 son iguales". La condición $a_1 \# a_2$ se lee como "el árbol a_1 y a_2 no son iguales". La relación de igualdad de árboles se define recursiva-mente al declarar que dos árboles a_1 y a_2 son iguales si:

- las etiquetas de a_1 y a_2 son iguales como etiquetas,
- a_1 y a_2 tienen el mismo número **n** de hijos,
- (si $n \neq 0$) el primer hijo de a_1 es igual al primer hijo de a_2, el segundo hijo de a_1 es igual al segundo hijo de a_2, el **n**-ésimo hijo de a_1 es igual al **n**-ésimo hijo de a_2.

Para que dos **etiquetas** sean iguales, estas tienen que ser:

- del mismo tipo (dos identificadores, dos números, dos caracteres),
- iguales con respecto al criterio de igualdad correspondiente a su tipo.

Implicación: La condición $a_1 => a_2$ se lee como "los árboles a_1 y a_2 son ambos booleanos y si a_1 tiene el valor **1** entonces a_2 tiene el valor **1**".

Comparaciones numéricas: Las condiciones $a_1 < a_2$, $a_1 <= a_2$, $a_1 > a_2$, $a_1 >= a_2$ se verifican si los árboles a_1 y a_2 son ambos números y si a_1 es menor que (o menor o igual a, mayor que, mayor que o igual a) a_2.

Relaciones unarias: Las relaciones unarias con frecuencia se llaman relaciones de tipos dado que implican principalmente la naturaleza de la etiqueta ligada al nodo inicial del árbol al cual se aplica.

a !numt

a !chart
a !boolt
a !idt

Estas condiciones se verifican si el nodo inicial **a** está etiquetado por un número, un caracter, un booleano, o un identificador respectivamente.

a !tuple

Esta condición se verifica si el nodo inicial del árbol **a** es **<>**, en otras palabras si **a** es una **tupla**.

a :: n

Esta condición se lee como "**n** es un entero no negativo con **n** ramas desde el nodo inicial de **a**". **a** es una tupla de longitud **n**.

a !num
a !char
a !bool
a !id

Estas condiciones se verifican si **a** es una hoja y si su único nodo está etiquetado por un número.

2.3.7 Listas

Las **listas** en **Prolog III**, se basan en el concepto de **punto par** de la misma forma que existen en **Lisp** y otros **Prologs**. Se representan por **[]**. Sintácticamente, los siguientes términos se permiten:

- lista vacía : **[]**
- **[U | L]** representa una lista con cabeza **U** y resto **L**:
- **[U1,U2,...,Un | L]** representa una lista cuyos primeros elementos son **U1,U2,...,Un** y cuyo resto es **L**.

Es importante entender que la **lista** no es específicamente una opera-ción de construcción, sino simplemente otra manera sintáctica de representar en ciertos árboles: el **punto par** y **nulo** (nil).

2.3.8 Restricciones

Una **condición** consiste de la aplicación de una **relación** para un árbol o pares de árboles en el dominio de **Prolog III**. Una **restricción** será un objeto sintáctico, formado por el símbolo que expresa una relación, y un término o pares de términos. Es posible combinar con varias restricciones como **<,** ≤**, >** o ≥ suponiendo que todas son del tipo **<,** ≤ o del tipo **>,** ≥. Se puede escribir entonces **{ 1 < x < y** ≤ **3 }** en lugar de **{ 1 < x, x < y, y**≤ **3}**.

2.3.8.1 Sistemas restrictivos

En los programas de **Prolog III**, las restricciones siempre se expresan como **sistemas restrictivos**. Un **sistema restrictivo** es una secuencia finita de restricciones las cuales están separadas por comas y encerradas entre corchetes.

Un **sistema restrictivo** se dice que es **verificable** o mejor aún **resoluble**, si hay al menos un argumento en las variables del sistema las cuales se transforman dentro en un conjunto de condiciones verdaderas aplicadas a los árboles. Esta asignación es llamada una **solución** del sistema restrictivo. Por ejemplo, el sistema restrictivo

{ P ≥ 0, L ≥ 0, P + L = 16 }

es **resoluble** dado que entre otras cosas, la asignación

{ P ← 8, L ← 8 }

se transforma dentro del siguiente conjunto de condiciones:

{ 8 ≥ 0, 8 ≥ 0, 8 + 8 = 16 }

2.3.9 Reglas y Preguntas

Los programas en **Prolog III** consisten de **reglas** que tienen una cabeza la cual es un término, y un cuerpo formando por una secuencia de metas y un sistema restrictivo. La secuencia de metas así como el sistema restrictivo, pueden estar vacíos. Cuando el sistema restrictivo está vacío, este se puede omitir. Simbólicamente se tiene que:

< regla >
::= < cabeza >
- > { < meta > **} [,** < sistema restringido > **];**

< pregunta >
::= { < meta > **} [,** < sistema restringido > **];**

< meta >
::= /
::= < término >

< cabeza >
::= < término >

para activar un programa se tiene que componer una pregunta. Una **pregunta** es igual a la cabeza de la regla.

2.3.9.1 Significado de un programa

Un **programa** en **Prolog III** tiene un doble significado. Por un lado constituye la definición recursiva de un conjunto de árboles los cuales se consideran por el programador **hechos ciertos**; esto es llamado la **semántica declarativa** del lenguaje, y por otro lado, el programa se manifiesta a sí mismo como una serie de acciones a ejecutar de las cuales hay dos tipos: la solución de sistemas restrictivos y la llamada a procedimientos. Esto se llama la **semántica operacional** de **Prolog III**.

El conjunto de hechos definen un programa. Un programa consiste de reglas de la siguiente forma:

$$t_0 \rightarrow t_1 t_2 \ldots . t_n , S;$$

donde $\mathbf{t_0}$ es un término, $\mathbf{t_1 \ldots t_n}$ son términos y **S** es un conjunto de restricciones. Una regla como esta, genera un conjunto de reglas que actualizan estos árboles, una regla para cada asignación **A** que hace las condiciones **C = S/A** sean ciertas es de la siguiente forma:

$$a_0 \rightarrow a_1 a_2 \ldots a_n ;$$

tal regla sin variables expresa una propiedad lógica: si $\mathbf{a_1, a_2 \ldots a_n}$ son hechos, entonces $\mathbf{a_0}$ es un hecho. Cuando **n = 0** la regla no tiene cuerpo: $a_0 \rightarrow;$ la correspondiente propiedad lógica es simplemente "$\mathbf{a_0}$ es un hecho".

El conjunto que define a un programa en **Prolog III** es el conjunto más pequeño de hechos que satisface todas las propiedades lógicas generadas por las reglas del programa. El programa por lo tanto consiste de reglas las cuales declaran **hechos explícitos**, y todas las otras reglas, las cuales se pueden ver como **procesos deductivos** para producir hechos implícitos.

2.3.9.2 Ejecución de un programa

Dada una secuencia $\mathbf{t_1, t_2, \ldots, t_n}$ de términos y un sistema restrictivo **S**, encontrar los valores de las variables las cuales transformen los términos $\mathbf{t_i}$ en hechos definidos por el programa y las restricciones de **S** dentro de condiciones verificadas.

Este problema se introduce a la máquina en forma de una pregunta

$$t_1 t_2 \ldots t_n, S$$

donde hay dos posibilidades particularmente significativas:

- si la secuencia $\mathbf{t_1 t_2 \ldots t_n}$ está vacía, entonces la pregunta se suma al resto de la solución del sistema **S**.
- si el sistema **S** está vacío y $\mathbf{n = 1}$ entonces la pregunta se suma a la pregunta ¿qué valores de variables se requieren para que el término $\mathbf{t_1}$ se transforme en un hecho definido por el programa?.

Para explicar cómo **Prolog III** calcula la respuesta a estas preguntas se introduce una máquina abstracta. Esto consiste en una máquina no determinística cuya única instrucción básica se describe por las siguientes tres fórmulas:

(1) $\left(W, t_0 t_1 \ldots t_n, S\right)$

(2) $s_0 \rightarrow s_1 \ldots s_m, R$

(3) $\left(W, s_1 \ldots s_m t_1 \ldots t_n, S \forall R \forall \left\{ s_0 = t_0 \right\}\right)$

La fórmula (1) representa el conjunto de la máquina a un momento dado. La pregunta actual es $\mathbf{t_0 t_1 \ldots t_n, S}$. **W** es el conjunto de variables en las que estamos interesados. La fórmula (2) representa la regla en el programa que se usa para cambiar el estado de la máquina. Si es necesario, algunas de estas variables se renombran tal que ninguna sea común con (1). La fórmula (3) representa el nuevo estado de la máquina, cuando la regla (2) se ha aplicado. Esto es solamente posible al pasar este estado si el sistema restrictivo **S** $\forall$ **R** $\forall$ $\mathbf{\{s_0 = t_0\}}$ posee una solución en cada uno de los términos en secuencia $\mathbf{s_1 \ldots s_m t_1 \ldots . \ t_n}$ los cuales representan un árbol definido.

Para responder a la pregunta $\mathbf{t_0 \ldots t_n, S}$ la máquina comienza en el estado inicial $\mathbf{(W, t_0 \ \ldots \ t_n, \ S)}$, donde **W** es el conjunto de variables que aparecen en la pregunta, y va a través de todos los estados que puede atar repitiendo la operación básica. **Prolog** funciona de tal forma que todas las reglas son tratadas en sucesión, como fueron escritas en el programa. Cada vez que se alcanza un estado se tiene la forma **(W, S)** en el cual la secuencia de metas es vacía, la respuesta dada por **Prolog III** será la solución del sistema **S** sobre el conjunto de variables **W**.

2.3.10 Restricciones Numéricas

De todos los dominios para lo cual las restricciones se pueden aplicar en **Prolog III**, el dominio numérico promete ser uno de los más aplicables. Esto es dado que **Prolog III** hace posible el solucionar ecuaciones y sistemas de inecuaciones sobre números racionales y reales, por lo tanto habilita a la programación lógica a solucionar problemas en las áreas de planeación, prevención, redes, análisis financiero, entre otros.

Las **restricciones numéricas** son objetos sintácticos que expresan la relación entre dos expresiones numéricas. La simple presencia de una variable en una expresión numérica restringe a la variable para que represente un valor numérico.

La principal restricción aplicada a las restricciones numéricas es que, sólo las ecuaciones lineales y desigualdades se toman en cuenta al solucionar algoritmos. La multiplicación de dos variables o división de una variable se difiere hasta que el número de variables conocidas la vuelve una restricción lineal. En la mayoría de los casos esta linealidad se obtiene inmediatamente después de la unificación.

Números

Las restricciones numéricas en **Prolog III** se aplican a dos tipos de objetos: números racionales y números de punto flotante.

Los **números racionales** se representan con precisión perfecta (con tantos dígitos como se requieran expresar exactamente). Todos los números racionales se codifican en forma de un número fraccionario irreducible: un par de enteros de precisión perfecta representando su numerador y denominador. Así **Prolog III** asegura que cualquier representación del número racional es única.

Con respecto a los **números de punto flotante**, su representación interna (y su precisión y extensión), se determinan por las caracterís-ticas de la computadora. Su sintaxis es la misma que la usada en la mayoría de los lenguajes que incluyen este tipo de datos numéricos. Además los números de punto flotante pueden ser muy útiles, y aún indispensables cuando se solucionan muchos problemas, aunque también se pueden considerar al tener efectos dañinos, especialmente en la exactitud de los cálculos que implican.

Expresiones numéricas

Las expresiones numéricas son términos construidos de variables, constantes numéricas y operadores numéricos. En **Prolog III** hay dos tipos de constantes numéricas: enteros positivos o cero, y números de punto flotante positivo o cero.

Tanto los enteros negativos o números fraccionarios se consideran constantes. Sin embargo se pueden expresar como resultado de las operaciones sobre los enteros no negativos.

Operadores numéricos

Los operadores numéricos empleados por **Prolog III** son:

- **+ : signo mas**
- **- : signo menos**

+ : suma
- : resta
/ : división
*** : multiplicación**

Sintácticamente, **Prolog III** conserva una filosofía de notación matemá-tica dado que permite que se omita el signo de multiplicación cuando este no causa ambigüedad.

En la parte numérica de **Prolog III**, existen dos tipos de relaciones: **relaciones unarias** las cuales se usan para especificar el carácter numérico de ciertos árboles y **relaciones binarias** las cuales se usan para construir restricciones numéricas verdaderas.

Relaciones Unarias

X !numt restringe la variable **x** a representar un árbol etiquetado por un número real o racional.

X !num el cual es equivalente a la asociación de las dos restricciones **x !numt** y **x::0**

Relaciones Binarias

= : igualdad
: desigualdad
< : estrictamente menor que
> : estrictamente mayor que
<= : menor que o igual a
>= : mayor que o igual a

2.3.11 Restricciones Booleanas

El dominio Booleano es uno de los dominios en el cual se aplican las restricciones en **Prolog III**. Las constantes, operaciones y relaciones definidas con este tipo se pueden usar para expresar todas las fórmulas del cálculo proposicional, dando como resultado un poder expresivo el cual es considerablemente mayor que el ofrecido por las cláusulas de Horn o la producción de reglas. Sin embargo, hay un precio a pagar por este poder. El tiempo de ejecución de los algoritmos que solucionan restricciones booleanas son exponenciales.

Una **restricción booleana** es la relación unaria que restringe a una variable que representa un árbol etiquetado por un valor booleano; o una relación entre dos expresiones booleanas.

2.3.11.1 Expresiones Booleanas

Sea **V** el subconjunto de variables que representan valores booleanos. Sea **B=(0',1')** el conjunto de valores que los elementos de **V** puedan tomar. Estas son las representaciones de falso y cierto en **Prolog III**. Finalmente, consideremos las siguientes operaciones: **~** (no), **&** (y), **|** (o), **=>** (implica), **<=>** (equivalente).

Las expresiones Booleanas se definen como:

- **0** y **1** son expresiones booleanas,
- las variables booleanas son expresiones booleanas,
- si **f** y **g** son expresiones booleanas, entonces:
  ~ (f),
  (f) | (g),
  (f) & (g),
  (f) = > (g),
  (f) < = > (g), son expresiones booleanas

y cualquier expresión booleana en **Prolog III** es un **término**.

2.3.11.2 Asignaciones Booleanas

Una **asignación booleana** es un conjunto **X = {x_1:= a_1, x_2:= a_2,...}** donde un valor **a_i** tomado del conjunto **(0', 1')** se asocia con cualquier variable booleana **x_i**. Similarmente, un **sistema de restricciones booleano** es resoluble si existe al menos una asignación booleana la cual es solución de esta, o irresoluble en otro caso.

2.3.11.3 Simplificación de un sistema restrictivo

Referente a la simplificación de un sistema de restricciones, **Prolog III** generalmente cubre dos aspectos muy diferentes. El primer aspecto es el que hace al sistema tan indicativo como sea posible, removiendo los números redundantes para reducir el tamaño. El segundo aspecto concierne a la eliminación de variables innecesarias.

Prolog III no remueve todas las redundancias en las restricciones booleanas, sin embargo asegura que cada vez el dominio de valores posibles para una variable se reduzca a un elemento.

2.3.12 Técnicas de Retraso

Como en **Prolog II** y **Prolog II+**, **Prolog III** habilita la ejecución de una meta a ser diferida mientras espera por un término a ser conocido.

2.3.12.1 Términos conocidos

Un **término** es ***conocido*** cuando conocemos la etiqueta del árbol que representa y se sabe si el número de sus hijos es o no cero. Estos términos con frecuencia se reducen a una variable, conociéndose después como variables *conocidas*.

2.3.12.2 Restricciones diferidas

Prolog III difiere ciertas restricciones las cuales no corresponden a las restricciones impuestas por el lenguaje. Este es el caso para:

- el tamaño de las restricciones en el cual el tamaño no se conoce explícitamente.
- restricciones que implican concatenación en las cuales el tamaño del operador a la izquierda no es conocido.
- restricciones numéricas no lineales (implican productos de variables o divisiones por una variable).

Supóngase que se quiere construir un predicado **create(T1,T2,U)** el cual difiera la creación de una tupla **U** formada de dos elementos **T1** y **T2**

```
create(T1,T2,U ) ->
        freeze(T1, create(T1,T2,U));

create(T1,T2, <T1,T2>) -> ;
```

un primer ejemplo seria:

```
> create(illusion(A), T, U);
     { U = <illusion (A), T > }
>
```

Dado que el primer parámetro se conoce, el predicado "**create**" se ejecuta inmediatamente cuando se llama. Sin embargo

```
> create(T1, T2, U);
        {T1 = E[X],
           E[X]  !freeze(create' (E[X], T2, U) ) }
>
```

aquí el primer parámetro no se conoce en el tiempo de llamada, y nunca se conocerá hasta que deje de ejecutarse la pregunta. La salida de **Prolog III** indica que la variable **T1**, se

coloca dentro de un modelo normal por el significado del árbol general de construcción en la forma **E[X]**, el cual indica que el predicado diferido está sobre esta variable.

2.3.13 Técnicas de Control

En cada etapa de ejecución de un programa, el interprete de **Prolog III** realiza dos pasos: primero selecciona una meta a ejecutar de una secuencia de metas, y entonces selecciona la regla la cual se usará para ejecutar a esta. La meta seleccionada es siempre la primera en la secuencia de las metas a ejecutarse, y las reglas seleccionadas a ejecutarse son todas las reglas cuya cabeza se unifica con la meta pertinente. Dado que **Prolog III** es fundamentalmente un lenguaje declarativo, el concepto de control se reduce a un modelo simple (modificar o restringir selecciones). La forma en que **Prolog** hace estas selecciones puede inducir a algunos programas a entrar en ciclo y por lo tanto no realizar lo planeado.

2.3.13.1 El corte " / "

Normalmente, **Prolog III** intenta ejecutar una secuencia de metas en todas las formas posibles. Pero si una regla contiene un "**/**" (o corte) en una meta **q**, la ejecución de este "**/**" borrará todas las reglas seleccionadas las cuales permanezcan aún en espera de ejecutar **q**. Esto restringe el tamaño del espacio de búsqueda, se puede decir que "**/**" hace que Prolog **"**olvide" las otras formas posibles de ejecutar **q**.

El "**/**" es un parásito el cual puede aparecer únicamente en los términos que constituyen el lado derecho de la regla. Las reglas de selección restantes las cuales ejecutan un "**/**" son:

- las demás reglas que tengan la misma cabeza que la regla que contengan a "**/**".
- las demás reglas que puedan haberse usado para ejecutar los términos entre el inicio del cuerpo de la regla y el "**/**".

2.3.13.2 Interrupción

Un programa en **Prolog** se puede interrumpir a cualquier tiempo especificando una tecla, dependiendo del sistema usado (por ejemplo <Ctrl-C>). Un breve dialogo aparecerá:

User Interrumpt by (^C). C(ontinue, K(ill, tT(race, Q(uit?

habilitando al usuario a seleccionar una respuesta para continuar la ejecución (C), interrumpirla (K), continuar la ejecución en modo de trazos, o salir de la sesión (Q).

Existen otros predicados para interrumpir la ejecución de una meta, ejecutar un programa de control o verificar si una variable está ligada. Como es el caso de **block, block_exite, default, bound** y **free** respectivamente[18].

2.3.14 Entrada / Salida

Todos los dispositivos de **entrada/salida** usados por un programa en **Prolog III** se representan por entidades llamadas **unidades de entrada/salida**. La representación escrita de estas unidades es una cadena de caracteres; si es necesario la cadena también representará un campo de información de la unidad en relación al sistema operativo, nombre del archivo, entre otros.

A cualquier momento, el sistema **Prolog III** reconoce cierto número de unidades de entrada/salida. Estas son las unidades que se han abierto y que aún no se han cerrado. Los descritos de estas unidades son arreglos en una pila; la unidad en la parte superior de la pila se llama **unidad actual**. Todas las operaciones de entrada/salida se ejecutan sobre la unidad actual.

Cuando una unidad se cambia, la unidad actual no se cierra y permanece sin cambios, pero una nueva unidad se abre (si es necesario) y se coloca en la parte superior de la pila, antes de la unidad actual previa, de tal manera que más tarde pueda ser realmacenada cuando la unidad actual se cierre.

2.4 CLP®

Ahora describiremos la sintaxis de los programas **CLP®** y su semántica intuitiva.

2.4.1 La estructura $\mathfrak{R}$

El corazón de cualquier lenguaje **CLP** es una estructura, que define el dominio del discurso y las operaciones y relaciones sobre ese dominio. Para que este lenguaje se base formalmente en la semántica del esquema **CLP**, la correspondiente estructura tiene que ser una **solución compacta**[19].

Esencialmente, $\mathfrak{R}$ es una estructura doblemente ordenada donde una estructura pertenece a los números reales, y la otra al conjunto de árboles sobre funciones no interpretadas y los números reales.

2.4.1.2 Restricciones en CLP®

[18] Los detalles de estos predicados se pueden consultar en [PIII-93].
[19] Los detalles de esta propiedad se describieron en el Capítulo 1.

Constantes reales y variables reales son ambos **términos aritméticos**. Si t_1, t_2 son términos aritméticos, entonces $(t_1 + t_2)$, $(t_1 - t_2)$ y $(t_1 * t_2)$ lo son. Las constantes no interpretadas y los términos aritméticos son **términos** y por consiguiente son cualquier variable.

El resto de los términos se definen inductivamente como sigue: si **f** es la n-ésima función no interpretada y $t_1, \ldots, t_n$ son términos, entonces $f(t_1,\ldots,t_n)$ es un término. Si t_1 y t_2 son términos aritméticos, entonces $t_1 = t_2$, $t_1 < t_2$ y $t_1 \leq t_2$ son todas **restricciones** aritméticas. Si sin embargo, ninguno de los términos t_1 y t_2 son términos aritméticos, entonces únicamente la expresión $t_1 = t_2$ es una restricción.

Dado que estas restricciones tienen significados predefinidos, algunas veces las llamaremos **restricciones primitivas**.

2.4.1.3 Programas en CLP®

Un **átomo** es de la forma $p(t_1, t_2, \ldots, t_n)$ donde **p** es un predicado distinto a los símbolos **=, <** y ≤; y $t_1, \ldots, t_n$ son términos.

Una **regla** es de la forma:

$$A_o \text{ :- } \alpha_1, \alpha_2, \ldots, \alpha_k$$

donde cada α_i, $1 \leq i \leq k$, es una restricción primitiva o un átomo. El átomo A_o se llama la **cabeza** de la regla, mientras que los átomos restantes y las restricciones primitivas son conocidas colectivamente como el **cuerpo** de la regla. En caso de que no haya átomos en el cuerpo, podemos llamar a la regla como **hecho** o **regla unitaria** (A_o).

Un **programa CLP®** se define como una colección finita de reglas. Estas reglas en **CLP®** tienen el mismo formato en **Prolog**, excepto que las restricciones primitivas pueden aparecer junto con otros átomos en el cuerpo. Una **meta** en **CLP®**, es de la forma:

$$?\text{- } \alpha_1, \alpha_2, \ldots, \alpha_k$$

donde cada α_i, $1 \leq i \leq k$ es una restricción primitiva o un átomo. Además, cada restricción primitiva de la meta se clasifica a ser solucionada o diferida. Una subcolección de átomos y restricciones en una meta es algunas veces llamada una **submeta** de la meta.

2.4.2 Hacia una metodología de programación

Los lenguajes de programación lógica en general admiten un amplio rango de técnicas de programación [36]. Dado que **CLP®** admite un rango más rico, se especula que lo siguiente aumente la metodología al escribir programas en un lenguaje como **Prolog**.

2.4.2.1 Razonamiento jerárquico y programación con restricciones

Una **restricción primitiva** típicamente representa la propiedad local del subproblema a tratar. Mientras estas restricciones representan la relación entre varios parámetros de una parte en particular del problema (por ejemplo, la ley de Ohm para una resistencia en un circuito), se requiere de **restricciones globales** para describir la forma en que estas partes interactuan (por ejemplo, el uso de la ley de Kirchoff en un análisis de nodos).

En **CLP®**, las propiedades globales de un problema se representan por medio de reglas, y las restricciones globales se asocian con las restricciones respuesta del programa.

La forma mas directa de representar propiedades globales en un programa es la composición jerárquica. Por ejemplo, puede usarse una regla de la forma:

p(...) :- restricciones, ..., p_1(...), p_2(...),...,p_n(...)

donde **p_1, ..., p_n** son las definiciones de grandes partes indepen-dientes de algún sistema.

2.4.2.2 Restricciones como salida

En los lenguajes de programación tradicional, la salida es de una naturaleza explícita en el sentido de que podemos imprimir solamente los valores de las variables. Los lenguajes funcionales y lógicos proveen salidas mas sofisticadas en forma de ligaduras, o unificadores con variables dadas. Sin embargo, estos siguen un modelo explícito de salida. **CLP®** en contraste provee salidas en forma de **restricciones respuesta** y estas están inherentemente implícitas.

La salida implícita aumenta el poder expresivo de un lenguaje de programación en muchas formas. Esta permite que respuestas complejas se representen de una manera simple y compacta descri-biendo objetos, los cuales no tienen una representación sintáctica y explícita.

Las **restricciones respuesta** encarecen no solo la salida, sino también la metodología de programación. En particular, podemos pensar en una restricción respuesta como una evaluación parcial de un progra-ma, esto es, una instancia mas específica del programa original. Consideremos, por ejemplo, el siguiente programa el cual relata los parámetros clave de una hipoteca:

```
hipoteca(P, Tiempo, IntRazón, MP, B ) :-
        Tiempo > 0,
        Tiempo < = 1,
        Interés = Tiempo * ( P * IntRazón / 1200),
        B = P + Interés - (Tiempo * MP).
```

```
hipoteca( P, Tiempo, IntRazón, MP, B ) :-
        Tiempo > 1,
        Interés = P * IntRazón / 1200,
        hipoteca( P + Interés - MP, Tiempo -1, IntRazón, MP, B).
```

donde los principales parámetros son: vida de la hipoteca (en meses), porcentaje de interés anual (% mensual), pago mensual y finalmente, el balance sobresaliente. La meta

```
?- hipoteca( 100000, 360, 12, MP, 0 ).
```

pregunta cuanto cuesta financiar una hipoteca de $100,000 al 12% de interés en 30 años, y la respuesta obtenida es **MP=1028.61**.

se puede volver a preguntar: **`?- hipoteca(P, 360, 12, 1028.61, 0).`**

Lo cual nos da la respuesta esperada **`P = 100,000`**. El punto principal de este ejemplo sin embargo, es que podemos preguntar, no solo por los valores, sino también por la **relación entre P**, **MP** y **B**. Por ejemplo:

```
?- hipoteca( P, 120, 12, MP, B).
```

Da la respuesta

```
B= 0.302995 + P - 69.700522 * MP
```

Este ejemplo en particular ilustra el punto de que las restricciones respuesta pueden ser vistas como una evaluación parcial del programa. En este caso, la anterior ecuación es el resultado de evaluar parcialmente el programa con respecto al Tiempo=120 e Interés=12. Finalmente la meta

```
?- hipoteca(999, 3, Int, 0, 400).
```

dará lugar a restricciones no lineales, donde el sistema regresará la restricción respuesta:

```
400 = (-400 + (599 + 999 * Int) *  (1 + Int )  * (1 + Int) ).
```

Aún cuando la implementación en **CLP®** no puede determinar si esta ecuación es satisfactoria, la ecuación de hecho describe la respuesta correcta.

2.4.2.3 Problemas de búsqueda combinatoria

“La metodología **examina/genera** de **CLP** se aplica típicamente a problemas que satisfacen restricciones y de búsqueda combinatoria (CSP) para los cuales no se conoce una solución

buena y por lo tanto, hacen uso de alguna técnica de búsqueda" [PVHD88]. Esto implica buscar a través del espacio solución del problema y hacer uso de las restricciones para guiar la búsqueda tanto por generación activa de valores como por medio de la poda cuando las restricciones se vuelven insatisfactibles.

En **CLP®**, la estructura general de un programa **examina/genera** es:

```
p(...) :-
        ... restricciones primitivas ....
        ... restricciones expresadas con predicados ....
        ... generadores para las variables ...
```

donde los generadores se usan para instanciar las variables cuyo rango se encuentra sobre un dominio en particular.

Esta es generalmente una mejor estrategia de búsqueda que la de **genera**/**examina** ya que las restricciones se examinan antes de que se generen, evitando así la generación cuando ya se conoce que las restricciones son inconsistentes. Un ejemplo de búsqueda combinatoria en **CLP®,** es el algoritmo criptoaritmético de **SEND + MORE = MONEY.**

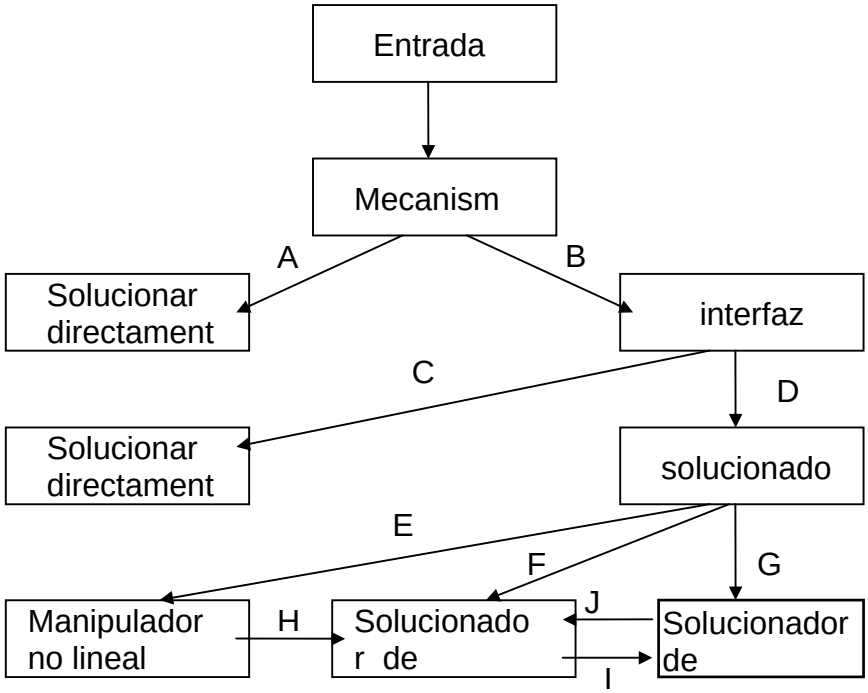
2.4.3 El Intérprete de CLP®

El propósito original del diseño y construcción del sistema **CLP®** fue dar evidencia del potencial práctico del esquema **CLP**. La presente sección muestra el esquema de un intérprete escrito en alrededor 15,000 líneas en código C. Este intérprete se organiza en 6 partes principales.

- un **mecanismo de inferencia,** el cual controla la ejecución de pasos de derivación y mantiene las variables ligadas;
- una **interfaz,** la cual evalúa expresiones aritméticas comple-jas y transforma restricciones a un pequeño número de formas estándar.
- un **solucionador de ecuaciones,** el cual trata con ecuaciones aritméticas lineales que son demasiado complicadas como para manipularse en el mecanismo y la interfaz.
- un **solucionador de desigualdades,** el cual trata con desigualdades lineales.
- un **manipulador de restricciones no lineales,** en el que se almacenan ecuaciones no lineales, este envía ecuaciones al solucionador de ecuaciones lineal mientras que estas se vuelven lineales; y
- un **módulo de salida,** el cual convierte las restricciones representadas internamente en un modelo simplificado.

La principal novedad en la implementación del sistema **CLP®** radica en el **resolvedor de restricciones** y el **módulo de salida**.

A continuación se muestra gráficamente al intérprete de **CLP®.**



A: restricciones provenientes del mecanismo
B: demasiado complicado para el mecanismo
C: reducciones al formato del mecanismo
D: demasiado complicado, invoca al solucionador
E: no lineal, diferir
F: ecuaciones lineales
G: desigualdad lineal
H: lo no lineal se excitó, enviar al solucionador
I: ecuaciones que afectan a las desigualdades
J: igualdad inferida

Intérprete de CLP®

2.4.3.1 El mecanismo

El **mecanismo** es una adaptación de una estructura del intérprete de **Prolog**. Se tienen dos estructuras de datos centrales, las cuales están apiladas. La operación mayor ejecutada por el mecanismo es la creación de ligaduras de ciertas variables; este se activa tanto en la reducción de pasos en una submeta, como cuando se encuentran ciertos tipos de ecuaciones.

Los tipos de restricciones, las cuales se solucionan directamente en el mecanismo son :

- una ecuación entre dos variables no solucionadas,
- una ecuación entre una variable no solucionada y una variable solucionada,
- una ecuación que implica una función no interpretada,
- una ecuación o desigualdad entre dos números, y
- una ecuación entre una variable no solucionada y un número

Como es usual, una variable puede estar atada tanto de un apuntador a otro término de la estructura, como a otra variable formando una cadena de referencia. A diferencia de **Prolog** donde una variable puede ser liberada o atada a un término únicamente, las variables en **CLP®** pueden pertenecer a uno de los diferentes tipos descritos. En sí, una variable puede estar atada de todos modos por una ligadura implícita dando como resultado la unificación. Los diferentes casos que se dan al realizar la unificación son:

variable no solucionada y variable no solucionada. Consiste en elegir una variable la cual puede depender de la estructura actual, y entonces ligarla a otra variable.

variable no solucionada y variable solucionada. Para evitar invocar al solucionador de restricciones, la variable no solucionada se “liga” a la variable solucionada y viceversa. Esta opción es crucial para la eficiencia.

una función y cualquier término. En caso de que el término sea una variable no solucionada, se crea un vínculo directo de esta variable al otro término. En caso de que el término contenga un factor, sus principales factores se verificarán para ver si son iguales. Si lo son, entonces sus correspondientes argumentos se unificarán recursivamente.
número y número. Dado que se usa una implementación de punto flotante, decimos que dos números son iguales si están lo suficien-temente “cerca” el uno del otro.

variable no solucionada y número. Dado que una variable no solucionada no aparece en el solucionador, la variable puede simple-mente vincularse al número.

variable solucionada y variable solucionada o número. La interfaz, y entonces quizás el resolvedor de restricciones, son invocados.

término aritmético y cualquier otro. Aquí también la interfaz, y entonces quizás el resolvedor de restricciones, son invocados.

Pueden surgir vínculos especiales entre una variable y un número, por ejemplo en:

- el **mecanismo**; cuando una variable no solucionada se iguala con un número el cual aparece en el programa.
- la **interfaz**, cuando una variable no solucionada se iguala a una expresión aritmética evaluable directamente.
- el **resolvedor de ecuaciones**, cuando un sistema de ecuaciones es solucionado y este sigue con ciertas variables hasta que toman un valor específico.
- el **resolvedor de desigualdades**, como una secuencia de igualdad implícita, esto es, una ecuación vinculada por un número de desigualdades no estrictas.

Las ligaduras en los primeros dos casos son de naturaleza explícita, y tienen sus contrapartes en los sistemas de programación lógica. En los dos casos restantes, las ligaduras ocurren de una manera implícita.

Para un acceso más rápido a todos estos valores, estos se agrupan en una estructura de datos central y un arreglo de ligaduras implícito, así esta estructura se accesa desde todos los módulos en el sistema. Donde el mecanismo juega el papel de una memoria “caché”.

2.4.3.2 La Interfaz

Este módulo es llamado desde el mecanismo siempre que una restricción contenga un término aritmético. La interfaz primero simpli-fica la restricción de entrada evaluando las expresiones aritméticas. Si como resultado de esto la restricción se aterriza, entonces se crea un examen apropiado. Si aparece exactamente una variable no solucio-nada en las restricciones y si la restricción es una ecuación, entonces se crea un vínculo implícito. En todos los otros casos, el resolvedor es invocado.

Antes de que la interfaz invoque al resolvedor, este transforma la restricción simplificada dentro de una de las siguientes formas:

- una desigualdad de la forma **Variable > 0**;
- una desigualdad de la forma **Variable** $\geq$ **0**;
- una ecuación de la forma

$$\sum_{i=1}^{n} c_i X_i = d$$

donde $n \in \{1,2,3,4\}$ y c_i y **d** son números reales.

- una ecuación de la forma **Variable = c * Variable * Variable** donde **c** es un número real.

2.4.3.3 El resolvedor de ecuaciones

Este módulo se invoca de 3 formas. Primero vía la interfaz con una ecuación lineal en la forma descrita anteriormente; segundo, vía el resolvedor de desigualdades con una igualdad implícita; y tercero, vía el manipulador de restricciones no lineales con una ecuación que haya sido excitada.

El **resolvedor de ecuaciones** retorna **cierto** si las restricciones satisfactibles ya almacenadas en el resolvedor, son consistentes con la ecuación de entrada.

La estructura de datos central en el resolvedor de ecuaciones es una tabla la cual almacena en cada fila, la representación de alguna variable lineal de las variables paramétricas. El método básico de solución es un modelo de eliminación Gaussiano. Una operación crítica surge de substituciones simplificadas, esto es, el reemplazo de una variable paramétrica seleccionada por una expresión paramétrica equivalente. Otra operación crítica es la comunicación entre los resolvedores de ecuaciones y desigualdades.

2.4.3.3.1 Tabla de ecuaciones

El conjunto de variables resolubles enviadas desde la interfaz se particionan por el resolvedor de ecuaciones en dos conjuntos de variables: **paramétricas** y **no paramétricas**.

Las ecuaciones lineales se almacenan en un **modelo de solución paramétrica** usando por conveniencia una notación matricial, esta es posiblemente una colección vacía de ecuaciones, de las cuales la i-ésima es de la forma:

$X_i = b_i + c_{i,1}*T_{i,1} + c_{i,2}*T_{i,2} + ... + c_{i,n}*T_{i,n}$

donde $n \geq 0$, X_i es una variable no paramétrica, $c_{i,j}$ $1 \leq j \leq n$ son coeficientes de números reales, y $T_{i,j}$ $1 \leq j \leq n$ son variables paramétricas. La variable X_i se llama el **sujeto** de la ecuación.

2.4.3.3.2 Agregando una nueva ecuación lineal

Llamaremos al modelo de solución paramétrica inicial (vacío) θ_o. se describe un modelo de solución paramétrica θ_{j-1} y una ecuación lineal de entrada E_j, $j \geq 1$, entonces ¿cómo determinar si $\theta_{j-1} \wedge E_j$ son resolubles?; si lo son, se obtiene un nuevo modelo solución θ_j.

Sea C_j el resultado de substituir[20] la salida de cada variable no paramétrica **X** en E_j, donde **X** aparece en la tabla θ_{j-1}, con el correspondiente modelo paramétrico de **X**.

(1) Si C_j es ahora de la forma **O=O**, entonces $\theta_j = \theta_{j-1}$,
(2) Si C_j es de la forma O_i**=c** donde **c** es un número real diferente de cero, entonces la colección $E_1 \wedge E_2 \wedge ... \wedge E_j$ es inconsistente, y así θ_j no existe. Simplemente se retorna negatividad.
(3) Si C_j contiene variables no paramétricas[21], decimos $X_1, X_2, ..., X_m$, $m \geq 1$, se selecciona X_1 y se reescribe C_j con X_1 como el sujeto (las variables restantes $X_2,...,X_m$ ahora se vuelven nuevas variables paramétricas). La ecuación resultante se suma a θ_{j-1} para obtener θ_j.
(4) Si C_j no contiene variables diferentes a las variables paramétricas, se elige **T**, y se reescribe C_j con **T** como el sujeto. Por razones de eficiencia, la elección de **T** no es arbitraria:
    - Se elige **T**, tal que esta no aparezca en la tabla del resolvedor de ecuaciones.
    - Sí, sin embargo, todas las variables en C_j aparecen en el resolvedor de desigualdades, entonces se elige si es posible **T**, tal que **T** sea una variable **básica** en el resolvedor de desigualdades;
    - Si sin embargo, todas las variables en C_j aparecen en el resolvedor de desigualdades como variables no básicas, entonces se puede elegir arbitrariamente una variable cualquiera para que sea **T**.

La ecuación resultante C_j**'** define la substitución que habilita la **eliminación paramétrica**. Esto es, θ_j se obtiene de θ_{j-1} substituye la salida de la variable **T** usando esta substitución, y entonces se agrega la ecuación C_j**'** a la tabla.

2.4.3.3.3 Excitando restricciones diferidas

Cuando las variables se vuelven nulas, es necesario comunicar este hecho al manipulador de restricciones no lineal. Sin embargo, es importante que el módulo no

[20] Las substituciones implican tanto reemplazar términos en las variables así como la colección de términos comunes.
[21] Estas variables son nuevas para el resolvedor.

lineal se invoque únicamente **después** de que la ecuación actual haya sido tratada por completo. Esto es, porque el módulo no lineal puede por sí mismo invocar al resolvedor de ecuaciones. Surgen sin embargo, algunas complicaciones en la implementación debido a estos dos hechos.

El módulo de solución paramétrica se representa principalmente por una tabla de listas enlazadas. Haciendo uso de una tabla de **referencia cruzada** que mapea cada variable paramétrica a la lista de ocurrencias de la tabla principal. Las operaciones críticas que surgen entonces son:

Substituciones paramétricas: Dado que una variable paramétrica **T** se reemplaza en la tabla, la primera operación es obtener, desde la lista en la tabla de referencia cruzada correspondiente a **T**, la lista de ocurrencias **T** en la tabla. Las substituciones actuales tendrán un costo proporcional a la longitud de esa lista.

Retroceso: En retroceso se presenta una diferencia muy importante entre **CLP®** y **Prolog**. En este último, los términos sintácticos no se cambian realmente durante el cálculo, mas bien, estos son meramente instanciados. Así el retroceso simplemente requiere que las variables que hayan sido ligadas a partir del último punto de selección, estén libres.

En **CLP®**, sin embargo, las substituciones realmente cambian el modelo de una ecuación paramétrica. Así, cuando cada fila de la tabla se cambia por primera vez después de un punto de selección, el modelo previo tiene que almacenarse. Un segundo punto es que el resolvedor de restricciones tiene sus propios puntos de selección a gestionar.

2.4.4 El resolvedor de desigualdades

Dos clases de restricciones entran al resolvedor de desigualdades:

(a) desigualdades lineales desde la interfaz, y
(b) ecuaciones desde el resolvedor de ecuaciones.

El **resolvedor de desigualdades** decide si una restricción lineal que se introduce, es consistente con las restricciones ya almacenadas tanto en el resolvedor de ecuaciones como en el resolvedor de desigualdades. Si no es así, simplemente retorna negatividad. Por otro lado, si se agrupan las restricciones en la tabla y se determina el conjunto de igualdades implícitas liberadas tanto de las restricciones almacenadas, como de la restricción de entrada, este entonces comunica estas igualdades al resolvedor de igualdades.

2.4.4.1 Tabla de desigualdades

Todas las restricciones se representan como ecuaciones en la tabla. La i-ésima columna en cada fila, almacena el coeficiente de la i-ésima variable en la ecuación. Donde **R** es una fila, se escribe **coef(X, R)** para denotar la entrada de la variable **X** en **R**. Existen otras dos columnas en la tabla; una para la constante la cual aparece en todas las ecuaciones, llamada la columna constante; y otra, llamada la columna simbólica, la cual representa las ecuaciones liberadas de las desigualdades estrictas.

Cada variable en la tabla es tanto una **variable no restrictiva**, en cuyo caso puede asumir cualquier valor de número real, o una **variable ociosa**, en cuyo caso puede asumir cualquier valor de número real no negativo. Las variables ociosas se crean en el resolvedor de desigualdades y nunca se encuentran fuera de su módulo.

Se dice que una tabla de **n** filas está solucionada si hay **n** columnas distintas, llamadas columnas básicas, tal que:

- la columna constante contiene únicamente números no negativos;
- la columna simbólica contiene únicamente números no negativos si la correspondiente entrada en la columna constante es cero;
- cada columna básica asociada con una variable no restringida contiene exactamente una entrada diferente de cero en la tabla.
- cada columna básica asociada con una variable ociosa contiene exactamente una entrada diferente de cero en la tabla, y esta entrada es positiva;
- una fila contiene una variable ociosa básica que contiene solamente variables ociosas.

2.4.4.2 Aumentando la tabla con una nueva restricción

Cuando una nueva restricción se introduce, el resolvedor de desigualdades intenta inmediatamente regresar después de registrar el hecho de que esta restricción se encontró. Si falla, intenta simplemente agregarla a la tabla. Solamente cuando estos dos intentos fallan, el resolvedor de desigualdades necesita tratar con la tabla original.

Las restricciones que pueden introducirse al resolvedor de desigualdades son:

a) una desigualdad lineal estricta o no estricta,
b) una ecuación lineal de todas aquellas variables que aparecen en la tabla como variables no básicas,
c) una ecuación lineal de todas aquellas variables que aparecen en la tabla, si al menos una de ellas, decimos **X**, es básica.

Caso (a): [desigualdad]. Sea la desigualdad **C** de la forma general

$$c_1 * Y_1 + c_2*Y_2 + \ldots + c_n*Y_n \geq c, \text{ ó}$$
$$c_1 * Y_1 + c_2*Y_2 + \ldots + c_n*Y_n > c$$

donde ninguna de las Y_i son básicas. Se crea una nueva variable ociosa **s**, y se re-escribe la desigualdad de la forma

$$c_1 * Y_1 + c_2*Y_2 + \ldots + c_n*Y_n - s = c$$

en el caso de que **c** sea una desigualdad no estricta, y

$$c_1 * Y_1 + c_2*Y_2 + ... + c_n*Y_n - s = c + 1 + EPS$$

en el caso de que **c** sea una desigualdad estricta.

Si **c<0**, se normaliza la ecuación multiplicándola por -1, y se agrega a la tabla como una nueva fila. Entonces,

- si la ecuación contiene una nueva variable no restrictiva, se hace una variable básica, y se retorna afirmativo,
- si la ecuación contiene únicamente variables ociosas, una de estas es nueva y tiene un coeficiente positivo, se hace una variable básica, y se retorna afirmativo.

Si ninguna de estas dos situaciones (examinadas en el orden dado) sucede, entonces se procede al algoritmo primario de la siguiente subsección.

Caso (b): [igualdad conteniendo variables no básicas]. Denotemos a la ecuación de entrada como:

$$X = c_1*X_1 + c_2*X_2 + ... + c_n*X_n + c$$

donde todas las variables son “no básicas”. Usando esta ecuación como una substitución de **X**, se reemplazan todas las ocurrencias de **X** en la tabla. Es posible que algunas filas **R** sean de **valor(R)<(0,0)**; si es así, se multiplica **R** por -1, y se retorna afirmativo.

Caso (c): [igualdades que contienen una variable básica]. Se escribe la ecuación de entrada usando la variable **X** como sujeto, obteniendo:

$$X = c_1*Y_1 + c_2*Y_2 + ... + c_m*Y_m + c$$

dado que **X** es “básica”, esta aparece en exactamente una fila. Sin perder la generalidad, se asume que esta fila es la última fila. Usando la ecuación anterior como una substitución de **X**, se crea otra substitución reemplazando **X** en la última fila por

$$c_1 * Y_1 + c_2 * Y_2 + ... + c_m * Y_m + c$$

llamando a la fila resultante **R**. Si **valor(R)<(0,0),** entonces se normaliza **R**. Se verifica si la ecuación contiene una variable no restrictiva (la cual se hace “básica”), y si es así, se retorna afirmativo.

De lo contrario si:

(i) **valor(R) > (0,0)** : se procede con el algoritmo primario de la siguiente subsección;

(ii) **valor(R) = (0,0)**: el resolvedor de desigualdades puede ahora retornar afirmativo (dado que cualquiera de las **Yi** puede tomar el valor 0).

2.4.4.3 Algoritmo primario

Consideramos como entrada, una tabla de **N** ≥ **0** filas tal que:

- las primeras **N-1** filas de una subtabla y la **N**-ésima o última fila, no contienen variables básicas;
- la subtabla no contiene igualdades implícitas[22], esto es, las restricciones representadas por la subtabla no implican **X=0** para cualquier variable **X.**

La salida del algoritmo primario puede ser:

- **irresoluble**; indicando que la restricción de entrada es inconsistente con la subtabla.
- **solución sin igualdades implícitas**, indicando que la restricción de entrada es consistente con la subtabla, y que la nueva tabla, la cual se coloca dentro del modelo solucionado, no implica igualdad alguna;
- **solución con igualdades implícitas**, indicando que la restricción de entrada es consistente con la subtabla y que la nueva tabla, la cual se coloca dentro del modelo solucionado, implica algunas igualdades.

El **algoritmo primario**, cuyo trabajo es encontrar una variable básica para la última fila de la tabla, se resume como :

```
repite{
        seleccionar una nueva variable básica X de la última fila;
        si(tal X no se puede seleccionar) {
                si(valor(última_fila)==(0, 0))
                                regresa solución con igualdades implícitas;
                delocontrario regresa insoluble;
        }
        seleccionar una fila R en la cual X es básica;
        pivot(X,R);
}hastaque (R es la última fila);
regresa solución sin igualdades implícitas;
```

2.4.4.3.1 Igualdades Implícitas

El proceso se invoca cuando una ecuación de entrada hace que una fila se modifique dentro de una nueva fila **R** tal que **value(R) = (0,0)**, o cuando una respuesta de igualdad implícita se ha obtenido del algoritmo primario. El algoritmo para igualdades implícitas, es el siguiente:

```
Sea S el conjunto inicial de variables nulas dadas por la última fila;
borra esta última fila de la tabla; permite IE=0;
```

[22] Alternativamente, el espacio descrito por la subtabla es por completo dimensional.

```
repite {
        selecciona una variable s ∈ S;
        inserta la igualdad ímplicita asociada con s dentro de IE;
        borra s de S;
        si ( ∃ un fila R ∈ relevant_rows(s) tal que coef(s,R) < 0) {
                asume w.l.o.g que R es la última fila;
                usando R como una substitución de s, susbstituyendo s de las otras filas;
                borrar s de R;
                aplica el algoritmo primario a la subtabla relevant_rows(s);
                si (solución con igualdades implícitas) {
                        agregas las variables R a S;
                        borrar R de la tabla;
}}}hastaque (S es vacío);
para( cada variable básica Y en la subtabla homogénea) {
        substituye la salida de Y de la subtabla homogénea;
} salida IE;
```

En esencia, el algoritmo determina cuales son las variables nulas **consecuencia** de alguna variable específica **s**. Este identifica primero la subtabla "**relevant_rows(s)**", de este conjunto se selecciona una fila **R** en la cual **s** puede estar aterrizada. Entonces se invoca al algoritmo primario para obtener una base alternativa para **R**. En otras palabras, se realiza una minimización de **s**. Cualquier invocación al algoritmo primario desde el algoritmo para igualdades implícitas, no puede dar como resultado una respuesta **irresoluble**. Dado que la tabla ha sido previamente determinada como consistente.

Si la respuesta es "**igualdades no implícitas**", entonces la tabla junto con el hecho de que **s = 0**, no implica que alguna variable sea igual a cero. Si la respuesta es "**igualdades implícitas**", entonces las variables encontradas en la última fila representan un subconjunto de las variables **X** tal que la tabla, junto con el hecho de que **s = 0** implica que **X = 0**. El punto importante aquí es que este subconjunto contiene al menos una variable (diferente de **s**).

2.4.5 Manipulador de restricciones no lineales

El **manipulador no lineal** tiene dos funciones principales. Primero, almacena ecuaciones no lineales de la forma $\mathbf{X_0 = c * X_1 * X_2}$ las cuales se obtienen de la interfaz. Segundo, recibe del resolvedor de igualdades, ecuaciones de la forma **variable = número**. Entonces verifica su almacén de ecuaciones no lineales para determinar cual de estas será excitada, esto es, cual de ellas se volverá lineal como resultado de la ecuación aterrizada. Específicamente, una ecuación no lineal $\mathbf{X_0 = c * X_1 * X_2}$ se vuelve lineal solamente como resultado de una ecuación de la forma $\mathbf{X_1}$**=número** o $\mathbf{X_2}$**= número**. Cada ecuación no lineal excitada se envía al resolvedor de desigualdades.

2.4.5.1 Pila no lineal

Al inicio, las ecuaciones no lineales se almacenan en una pila. Donde dos tipos de nodos componen sus elementos:

- un nodo **no lineal**, el cual contiene la representación de la ecuación no lineal almacenada, y
- un nodo **excitado**, el cual indica que la restricción que está asociada con un nodo no lineal en la pila, ha sido excitada como resultado de una ecuación del resolvedor de igualdades. El nodo excitado también indica qué variable en la ecuación no lineal se volverá aterrizada.

2.4.5.2 Tabla excitada de referencia cruzada

Como en las tablas de los resolvedores de igualdad y desigualdad, se tiene aquí una **tabla de referencia cruzada**, indizada por variables **X** en una lista cuyos nodos indican ocurrencias de **X** en la pila no lineal. Esto es, cada **ocurrencia** del nodo **M** apunta a un nodo no lineal el cual:

(a) contiena una ocurrencia de **X**, y
(b) no tiene un nodo excitado asociado con este.

Recíprocamente, cada nodo no lineal representa una ecuación de la forma $X_0 = c * X_1 * X_2$ con dos apuntadores a los dos nodos de ocurrencias asociadas con X_1 y X_2.

La pila no lineal entonces, hace uso de tres operaciones básicas. La primera consiste en aumentar la pila con una nueva ecuación no lineal $X_0 = c * X_1 * X_2$. Para insertar en la pila un nodo no lineal que represente esta ecuación, se agregan dos nodos de ocurrencia, uno para "**cross_ref(X_1)**" y el otro para "**cross_ref(X_2)**". Finalmente, en el nodo no lineal, se actualizan los apuntadores a estos dos nodos de ocurrencia. La segunda operación ocurre cuando una ecuación aterrizada, decimos **X = número**, se obtiene del resolvedor de igualdades. Si "**cross_ref(X)**" está vacía, entonces nada más se hace. De lo contrario, para cada nodo de ocurrencia de esta lista, **N** denota el nodo no lineal asociado con la ecuación no lineal en cuestión. Entonces:

(1) se introduce un nodo excitado **W** en la pila no lineal;
(2) se actualiza un apuntador en **W** para **N**, denotando que la restricción no lineal asociada con **N** ha sido excitada;
(3) se borran los dos nodos de ocurrencias apuntados por **N.**

La tercera operación ocurre durante el retroceso; esto es, el proceso de restauración de la tabla de referencias cruzadas cuando un nodo se ha sacado de la pila no lineal. Al sacar un nodo no lineal **N**, los dos nodos de ocurrencia apuntados por **N** se borran. En el caso de un nodo excitado **W** se aplica:

(1) accesar al nodo no lineal **N** al cual apunta **W**;
(2) insertar dos nuevos nodos de ocurrencia correspondientes a las dos variables en la restricción asociada con **N**;
(3) sacar el nodo **W**.

En resumen, una **pila no lineal** se usa para grabar, en orden cronológico, los eventos de diferir y excitar una restricción no lineal. Una **tabla de referencia cruzada** se usa para un rápido acceso a las restricciones no lineales que se afectan por la anulación de una variable dada.

2.4.6 Modulo de Salida

Llamado también conjunto de variables distinguidas. Se invoca automáticamente después de que una meta sucede, con las variables meta como distinguidas. También disponible como un predicado de la forma **dump([X, Y, ..., Z])**[23].

El propósito de este módulo es proyectar, con respecto a las variables distinguidas, la solución definida por las restricciones presentadas en el sistema.

2.4.6.1 Restricciones iniciales

Considérese primero al subconjunto de variables distinguidas las cuales están tanto "ligadas" o "desligadas" a una función no aritmética. Por conveniencia, se asume que si la variable **X** se liga al término **t**, entonces **t** contiene símbolos de funciones no aritméticas y variables resolubles (si hay) las cuales aparecen en las ecuaciones de la tabla igualdad. Entonces:

(1) las **variables primarias** son variables distinguidas así como variables satisfactorias las cuales aparecen en una restricción no aritmética.
(2) las **variables auxiliares** son aquellas que pertenecen a una variable primaria de las ecuaciones no lineales. Más específicamente, una variable auxiliar es aquella que aparece en una ecuación no lineal con una variable primaria u, otra variable auxiliar.
(3) el resto de las variables son las **variables sin salida**.

Por ejemplo, si **X, Y** y **Z** fueran variables distinguidas y las restricciones a manipular fueran:

C_1 = X = f(a,M)
C_2 = N = 2 * T
C_3 = Y = 4 * T
C_4 = Z = R + T
C_5 = M = N * R

entonces **X, Y, Z** y **M** son variables primarias, **N** y **R** son variables auxiliares y la variable restante **T** es la variable sin salida.

2.4.6.2 Fases del módulo de salida

La primera fase del módulo de salida aplica una serie de operaciones de pivoteo. Primero, todas las ecuaciones lineales, que sean variables no salida, se borran. Después, mientras hay una variable no salida **X** en alguna ecuación lineal **E**, la operación **pivot(X,E)** se

[23] Se hablará con mas detalle sobre este predicado en el siguiente capítulo.

ejecuta. Finalmente, mientras haya una variable auxiliar **X** en alguna ecuación lineal **E**, la operación **pivot(X,E)** se ejecuta. La segunda fase del módulo de salida tiene dos aspectos principales: desechar igualdades redundantes y eliminar variables sin salida. Esta inicia borrando todas las desigualdades que contengan solamente variables sin salida.

Redundancia

Una restricción **c** es **redundante** con respecto a una colección C de restricciones, si la colección que se obtiene de borrar **c** de C vinculando **c**, es C **- c = c**.

Una clase simple de redundancia es la **redundancia paralela**, esta ocurre cuando C contiene dos desigualdades las cuales definen hiperplanos paralelos y donde una desigualdad implica a otra. Así una de estas restricciones es redundante con respecto a C . Para desechar redundancias, se eliminan todas las desigualdades que crean redundancias paralelas con respecto a las restricciones almacenadas. Eliminar redundancias es importante no sólo para obtener una representación mínima de las restricciones, sino también para reducir la explosión de desigualdades producidas por algún algoritmo de eliminación variable.

Salida

Primero, las variables distinguidas no aritméticas son tratadas. Si tales variables no están ligadas, simplemente se ignoran. De lo contrario, se imprimen de la manera obvia, con una excepción: si la estructura contiene una variable aritmética **X** y si todavía hay una ecuación en la tabla de igualdades con **X**, entonces se imprime la expresión paramétrica correspondiente a **X**, de lo contrario, simplemente se imprime **X**.

2.5 CLP® como una aproximación al esquema CLP

La implementación de **CLP®** es una aproximación al modelo operacional del esquema **CLP** por 4 razones principales:

- Usa un algoritmo similar al algoritmo de unificación proposicional para solucionar ecuaciones entre términos no aritméticos.
- **CLP®** emplea una adaptación de izquierda a derecha en la estrategia de selección de submetas, en la cual una restricción no lineal se difiere hasta que la restricción a solucionar sea lineal.
- Hace uso de una representación de punto flotante con números reales. Usando una tolerancia pequeña y específica en la comparación de los números.
- El resolvedor de restricciones no determina la satisfacción de todas las restricciones no lineales.

Los dos primeros puntos pertenecen a la implementación de **Prolog**, así como a la de **CLP®** . El tercer punto, usar números de punto flotante, claramente puede dar lugar a fallas. Sin embargo, se estima que las implementaciones alternativas (números racionales o métodos basados en intervalos) sean costosamente inaceptables en un sistema de propósito general.

Una técnica general que trata con restricciones difíciles de resolver, consiste en emplear la técnica **retrasar/diferir** para que las restricciones a manipular se distribuyan únicamente cuando estas sean lo suficientemente simples[24].

Las consecuencias de no decidir qué hacer con las restricciones no lineales son:

a) demandar una secuencia infinita cuando existe un hecho de derivación exitosa, o cuando todas las derivaciones tienden a fallar finitamente;
b) obtener una colección de restricciones respuesta las cuales pueden no ser resolubles.

2.6 Análisis de diferencias

Desde el punto de vista de los lenguajes de programación, **CLP®** y **Prolog III** tienen significados diferentes. Una diferencia importante se basa en el método que cada uno usa para solucionar restricciones. En general, las restricciones se definen sobre una estructura dada, tal que la satisfacción del problema resulte ser intratable. Por ejemplo, en la aritmética de enteros, las restricciones no tienen un procedimiento decisivo, aún cuando se piensa que estas son útiles en cualquier lenguaje de programación.

Prolog III corta la generalidad de las restricciones dentro de la definición del lenguaje. Por ejemplo, las ecuaciones de cadenas son ciertas variables **X** que se pueden acompañar por una restricción que especifica que la longitud de **X** es un número en particular. Otro ejemplo se puede ver en su aritmética racional, la multiplicación simplemente está prohibida.

Los métodos tomados en **CLP®** y **CHiP** son similares, ambos emplean un mecanismo **difiere/excita** el cual se especifica de una manera simple:

- condición **difiere (delay)**: Esta es una subclase de restricciones las cuales son diferidas, esto es, tales restricciones simplemente son almacenadas y no se consideran para la satisfactibilidad.
- condición **excita (wakeup):** Define cuando una restricción diferida puede ser excitada; esto es, la restricción se remueve de su almacén de restricciones diferidas y entra al resolvedor.

Mientras esta técnica puede parecer similar al uso de primitivas diferidas encontradas en **Prolog**, las variantes ofrecen un átomo variable en la regla seleccionada; la diferencia es que la condición diferida en **CLP®** no se define en el programa.

En **CLP®**, la condición diferida se dirige a las ecuaciones no lineales y la condición despierta se vuelve lineal como resultado de las otras restricciones lineales en el sistema. **Prolog III** por su parte, hace uso de un predicado **freeze([X_1, X_2, ..., X_n],p)** cuyo efecto

[24] La definición exacta de **suficientemente simple** depende del poder del algoritmo resolvedor de restricciones.

es agregar la submeta **p** a la meta cuando las variables $X_1, X_2, \ldots, X_n$ hayan sido todas puestas a nulo.

Otra diferencia importante, es que en el sistema **Prolog III** se consideran a los números racionales (no a los números reales). Así, por ejemplo, no hay un método directo que implemente la función de la raíz cuadrada en **Prolog III**. Consecuentemente, existen varios tipos de aplicaciones, por ejemplo el modelado de sistemas físicos, para lo cual **Prolog III** no esta debidamente adaptado.

CHiP por su parte, tiene su principal aplicación en la búsqueda de problemas combinatorios y trata con restricciones enteras. Este también trata con los tipos de restricciones en **Prolog III**.

CLP®, sin embargo, escoge la representación convencional de punto flotante para números reales; dado los métodos de números racionales, punto flotante y no lineales, **CLP®** es la implementación mas cercana a la meta de razonamiento en la aritmética real, mientras que **Prolog III** y **CHiP** son una implementación perfecta de una aproximación a esta meta.

Como se mostró antes, el poder de un lenguaje de Programación Lógica como **Prolog** radica en tres mecanismos: **modelo racional**, **unificación** y **cálculo no determinista**. **Prolog** lleva el cálculo en el universo de Herbrand. La **unificación** se usa para resolver ecuaciones en este universo (términos **no interpretados**). Por lo tanto cuando se modela un problema, se tiene que usar un mapeo del dominio dirigido al universo de Herbrand. Esto causa la pérdida de no solamente la naturaleza de la expresión del problema sino también de la eficiencia de esta resolución.

Con **CHiP**, sin embargo, se pueden proveer, dominios de cálculo más ricos que el universo de Herbrand, y manipular términos más expresivos. Esto implica extender la unificación en **Prolog** para tomar en cuenta las interpretaciones proyectadas para algunos símbolos funcionales.

Prolog III resuelve restricciones de una manera **pasiva** a través del paradigma de **genera/examina** el cual causa su legendaria ineficiencia sobre grandes problemas.

CHiP y **CLP®** por su parte, emplean una metodología **examina/genera**, lo cual implica buscar a través del espacio de solución del problema y hacer uso de las restricciones para guiar la búsqueda tanto por generación activa de valores como por medio de la poda cuando las restricciones se vuelven insatisfactibles. Esta es generalmente una mejor estrategia de búsqueda que la de **genera/examina** ya que las restricciones se examinan antes de que se generen, evitando así la generación cuando ya se conoce que las restricciones son inconsistentes.

Por lo tanto, **CLP®** tiene el suficiente potencial como para resolver problemas que sus contemporáneos no pueden resolver del todo bien. Este es un sistema flexible y eficiente con características adicionales. Una de las mas significativas es su mecanismo, el cual permite a la máquina distinguir restricciones que se puedan manipular sin el resolvedor de restricciones a partir de aquellas que requieren del resolvedor.

La interfaz entre el mecanismo y el resolvedor de restricciones transforma a las restricciones en un modelo proposicional. En esta transformación, la restricción de entrada se divide en varias partes que son manipuladas por diferentes módulos del resolcedor, disminu-yendo así la necesidad de usarlo.

El manipulador de restricciones no lineal tiene un claro propósito. Mientras es importante que el retraso o almacenamiento de ecuaciones no lineales sea eficiente, es crucial que el despertar de las ecuaciones no lineales se haga eficientemente. Esto, porque es necesario checar las ecuaciones almacenadas a excitar cada vez que se activa el resolvedor de ecuaciones, y en general, porque la mayoría de estas no tratarán de excitarse. Así, lo que se requiere es un mecanismo eficiente que detecte cuando un mecanismo **excitado** no este siendo llamado.

El propósito del módulo de salida es doble. Primero, delimita las restricciones, tanto como sea posible, para que las restricciones respuesta reflejen la relación entre las variables y la meta solamente.

Segundo, transforma esas restricciones resultantes dentro de un modelo simplificado. Esto sirve al propósito de estandarización, esto es, dos conjuntos de restricciones definiendo el mismo espacio de solución podrían parecer semejantes, y se cumple con el propósito de tener una salida tan compacta como sea posible.

En resumen, la filosofía del sistema **CLP®** es asegurar una implementación práctica y manejable; el costo de satisfacer restricciones es proporcional a la dificultad inherente de las restricciones mismas, lo que se logra dadas las características de su mecanismo el cual manipula una gran cantidad de restricciones sin necesidad de usar siempre al resolvedor de restricciones. Sin embargo, con un resolvedor de restricciones perfecto, las restricciones por sí solas serían suficientes para obtener las respuestas sin necesidad de recurrir a los generadores usados en **CLP®**, lo que lo haría mas práctico con respecto a otros dado que, los resolvedores de restricciones perfectos sobre dominios discretos, son típicamente imprácticos de usar o aún imposibles de obtener.

El siguiente capítulo resalta aún más las características de este sistema y lenguaje.

3
Sistema y Lenguaje CLP®

El lenguaje **CLP®** (como ya se definió), es una instancia del esquema de la **programación lógica por restricciones** definido por Jaffar y Lassez [JL87]. Su modelo operacional es similar al de **Prolog**, donde la mayor diferencia es que la unificación se reemplaza por un mecanismo más general: *solucionar restricciones en el dominio de las funciones no interpretadas sobre términos aritméticos reales*.

Este capítulo denota las características del lenguaje. Se basa en la implementación del compilador de **CLP®** versión 1.2

Más información técnica sobre **CLP®** se puede encontrar en: diseño del lenguaje e implementación [JL87][JC92], meta - programación [HE89] y mecanismos diferidos [JJ91].

3.1 Características generales

El lenguaje **CLP®**, es un sistema interactivo que compila todos los programas y metas en un código CLAM, el cual se interpreta por un emulador código - byte que es parte del sistema. Este es portátil en el sentido de que corre virtualmente sobre todas las máquinas UNIX a 32 bits con un compilador estándar de **C**, así como en muchos otros.

3.1.1 Sintaxis

Un programa **CLP®** es una colección de reglas. La definición de regla es similar a lo que en **Prolog** es una cláusula, pero diferente en dos aspectos:

- las reglas pueden contener restricciones, así como átomos en el cuerpo, y
- la definición de términos es más general.

Una meta es una regla sin cabeza, como es usual.

El cuerpo de una regla puede contener un sin número de restricciones aritméticas, separadas por comas. Las restricciones son ecuaciones o desigualdades, formadas de constantes reales, variables, **+**, **-**,*****, **/** y **=**, **>=**, **<=**, **>**, **<**; donde estos símbolos tienen el significado usual, y donde los paréntesis se usan para resolver ambigüedad.

Se dice que cualquier variable que aparece en una restricción aritmética, es una **variable aritmética**, y no puede tomar un valor que no sea aritmético.

Los comentarios en el programa inician con un **%** y terminan al final de la línea; o bien, de la forma en que lo hace **C**, iniciando con un **/*** y terminando con un ***/**. A diferencia de los comentarios en **C**, estos pueden anidarse tal que el código ya contenga comentarios que se relacionen fácilmente.

3.1.2 Términos

Sintácticamente, un **término** es tanto un término simple o un término compuesto construido de términos simples. Un término es entonces un **término aritmético** o un **factor término**. Los términos simples son: **variables términos**, las cuales empiezan con un caracter en mayúscula o un subrayado " _ ".

Estas variables son llamadas **variables anónimas** y siempre representan una variable nueva. Las variables que contienen mas de un caracter y que empiezan con un subrayado, son variables ordinarias (estas no son variables anónimas), excepto que se ignoran durante la verificación de estilos.

Término constante numérico. Es un número real con un punto decimal opcional y un exponente entero (también opcional) el cual puede ser positivo o negativo.

Constantes numéricas simbólicas. Denotan valores constantes espe-ciales, por ejemplo:

pi = 3.14159265358979323846
pi / 2 = 1.57079632679849661923
pi / 4 = 0.78539816339744830962
e = 2.7182818284590452354
sqrt{ 2 } = 1.41421356237309504880
1/sqrt{ 2 } = 0.70710678118654752440
c = 2.99792458 * 10 ^ 8 (velocidad de la luz en el vacío)
g = 9.80665 (aceleración de la gravedad)
h = 6.626176 * 10 ^ -34 (constante de Planck)
e = 1.6021892 * 10 ^ -19 (carga elemental)

Términos cadena . Son cualquier secuencia de caracteres delimitadas por comillas (**"**).[25]

Termino aritmético. Un término aritmético es tanto una variable, una constante numérica o un término compuesto construido de términos aritméticos, el cual incorpora los símbolos de las funciones aritméticas : **+**, **-**, *****, **/**, **sin**, **arcsin**, **cos**, **arccos**, **pow**, **abs**, **min** y **max**. por ejemplo:

X, 3.14159, X + Y , sin(X + 2.0), (X + Y) /4

son términos aritméticos válidos. Sin embargo,

f(a), c + 5.0, cos(f (3))

no los son. Los términos aritméticos se interpretan como expresiones aritméticas.

3.1.3 Funciones

[25] En el presente, la interpretación de la sintaxis de la cadena no se ha determinado y todas las cadenas se tratan al inicio como factores constantes. Esto difiere de algunos Prologs los cuales usan esta sintaxis como una notación alternativa para las listas.

Son una secuencia de caracteres alfanuméricos (en letras minúsculas o " _ "), o bien, una secuencia de caracteres del conjunto { **\ & * + - . / : ; < = > ? { } \ ~** }. Así como cualquier secuencia de caracteres delimitados por las cotas simples { ' }. Por ejemplo, **'comida + bar'** es una función constante (átomo) . La constante especial **[ ]** denota la lista vacía o nula.

Funciones término. Las funciones término son tanto variables, funciones término (términos constantes) o términos compuestos. Una función término compuesta tiene la forma **f(t_1, t_2, …, t_N)** donde **N ≠ 0** , **f** es la función **n**-ésima no interpretada y **t_1, t_2, …, t_N** son términos.

Que una función sea **no interpretada**, significa que la función simplemente es tratada como una constante simbólica, opuesta a los términos aritméticos, los cuales son interpretados.

3.1.4 Restricciones

Una **restricción** es tanto una restricción aritmética o una función restrictiva. La primera se define de la forma **t_1 Delta t_2** donde **t_1** y **t_2** son términos aritméticos y **Delta** es una de las relaciones aritméticas **=**, **>=**, **<=**, **>** y **<**. Por ejemplo,

X > 5.0, X + Y + Z = 3, X <= Y

Una función restrictiva es de la forma **t_1 = t_2** donde **t_1** y **t_2** son tanto una variable o una función término.

3.1.5 Tipos de resultados

Informalmente, uno de los dos tipos de resultados emitidos en **CLP®**, se da por medio de los números reales, y el otro por los residuos de los términos aterrizados (variables libres).

Estrictamente hablando, **CLP®** es un lenguaje de tipo estático en el sentido de que las variables, funciones no interpretadas y predicados en un programa, tienen que usarse de manera consistente con respecto a su tipo. Esto es, cada variable y cada argumento de cada predicado y función no interpretada se conoce primero por su tipo.
Sin embargo, por razones de conveniencia, **CLP®** no ejecuta una verificación de tipos al momento de compilar. Esta decisión se basa en el hecho de que con frecuencia se sobrecarga al símbolo; por ejemplo, uno puede desear una base de datos **p** tanto de números como de letras:

p(1). p(2). p(a). p(b).

y poder correr una meta conteniendo **p(X)** y algunas restricciones con la base de datos. Al no ejecutar la verificación de tipos, se puede tener un error al tiempo de corrida. Esto es, una secuencia de ejecución la cual falla debido a un **choque de tipos**. Con frecuencia tales fallas indican que hay un error en el programa. El sistema **CLP®** no distinguirá fallas a partir de los errores obtenidos de las restricciones correctamente escritas.

Una forma correcta de pensar acerca del tipo usado cuando se escriben programas en **CLP®**, es que aún cuando un término aritmético aparezca en una regla, por cada variable **X** que haya, se agregue un átomo correspondiente al cuerpo de la regla **real(X)**. El sistema de predicados **real** es **cierto** sólo en caso de que haya una solución real para **X** en el conjunto actual de restricciones.

3.2 Programando en CLP®

Supongamos que todas las restricciones aritméticas son lineales. El cálculo inicia con una meta y un grupo de restricciones inicialmente vacío. La regla de selección del átomo (de izquierda a derecha) se usa para seleccionar tanto una restricción aritmética, como un átomo en cada etapa.

Cuando se selecciona una restricción, esta se agrega al conjunto de restricciones almacenadas y se determina si el conjunto resultante tiene solución. Si no hay solución, se da un retroceso. Por otro lado, cuando se selecciona un átomo, el conjunto de reglas se recorre en un módelo descendente, igualando este átomo con la cabeza de alguna regla.

Tal equiparación se realiza por medio de una ecuación entre estos dos átomos; la ecuación es tratada como cualquier ecuación entre tér-minos. En general, primero se unifican las partes sintácticas de los términos. Sin embargo, estos términos pueden contener términos aritméticos.
Como los términos aritméticos tienen un significado especial, estos no se unifican sintácticamente, sino que se soluciona una ecuación entre ellos en el dominio de la aritmética real. Por ejemplo, consideremos un programa que incluye tanto términos aritméticos como restricciones explícitas:

```
p( 10, 10 ).
q( W, c(U, V) )   :-   W - U + V = 10,
                       p( U, V ).
```

cuya meta es :

```
?- q( Z, c( X + Y, X - Y ) ).
```

lo cual nos da la ruta exitosa

```
?- q( Z, c( X + Y, X - Y ) ).
q(Z, c( X + Y,X - Y))= q(W, c(U,V )) ?- W - U + V = 10, p(U,V).
q(Z, c( X + Y,X - Y))= q(W,c(U,V)), W - U + V = 10  ?- p(U,V).
p(U,V) = p(10,10) ?- .
```

la respuesta para esta derivación es:

```
Y = 0,   X = 10,  Z = 10.
```

como se esperaba, esta respuesta no hace mención alguna de las variables **U, V** y **W**. Sin embargo, las respuestas no necesariamente muestran los valores de las variables ya que es posible obtener una respuesta como :

X + Y + Z = 0, X > Y.

Esta es una característica importante y muy útil del sistema **CLP®** como se verá mas adelante.

3.2.1 Retraso de las restricciones no lineales

Supóngase que un conjunto de restricciones tiene solución, pero una restricción que se agrega hace que este se complique demasiado, tal que no sea práctico decidir si aún permanece resoluble. Un método "ingenuo" al tratar con este tipo de problemas, es simplemente no permitir expresiones que puedan dar lugar a tal complejidad. Esto equivale a prohibir todas las restricciones no lineales. La pérdida de poder expresivo es, sin embargo, inaceptable.
CLP® permite todas las restricciones no lineales pero las mantiene en un conjunto de restricciones diferidas. Más precisamente, en cada paso operacional, en lugar de agregar ciegamente cada restricción al conjunto de restricciones e incurrir en el costo de ejecutar un exámen de satisfactibilidad, **CLP®** remueve las restricciones que hacen que el conjunto se complique demasiado, almacenándolas en un conjunto de restricciones diferidas.

También, a cada paso, es posible que algunas restricciones (en el conjunto de restricciones diferidas), no necesiten seguir diferidas dada la nueva información. En ese caso, estas se mueven del conjunto de restricciones diferidas al conjunto de restricciones y se crea un examen de satisfactibilidad[26].

El conjunto de restricciones diferidas puede contener otras restricciones, las cuales permanecerán ahí por mucho más tiempo. Debido a este mecanismo de retraso, se puede continuar a través de una secuencia de cálculo, aún pensando que el conjunto de restricciones diferidas y el conjunto de restricciones almacenadas no tienen solución. En el peor de los casos, se puede dar un ciclo infinito. Este es el precio que se paga por un algoritmo eficiente.

En el sistema **CLP®**, una ecuación lineal o desigualdad siempre se considera lo suficientemente simple como para ser solucionada de inmediato, pero las restricciones no lineales se difieren hasta que estas se vuelven lineales. Esto incluye a las funciones **seno**, **arcoseno**, **coseno**, **arcococoseno**, **potencia**, **máximo**, **mínimo** y **absoluto**, las cuales se difieren hasta que se vuelven evaluaciones simples en una dirección u otra.

3.3 Modelo Operacional

Una meta **G** se escribe de la forma **{ C, D, ?-, E}**, donde **C** es el conjunto de **restricciones satisfactibles**, **D** el conjunto de restricciones no lineales llamadas

[26] En general, la noción de qué expresiones son "demasiado complicadas" es dependiente de la implementación. En CLP®, solamente las restricciones no lineales se difieren.

restricciones diferidas, y la restricción diferida **E** una secuencia de átomos y restricciones. Al reducir una meta **{ C, D, ?-,E }**, **CLP®** toma un elemento de **E**, llamando a esto, una "**reducción hacia adelante"**, o selecciona una restricción desde **D**, llamando a esta una **reducción excitada**. Inicialmente, **C** y **D** están vacías, y **CLP®** intenta hacer una reducción hacia adelante.

3.3.1 Reducción hacia adelante

Si **E** está vacío, decimos que la meta es terminal y ninguna reducción de la meta es posible. Si **D** también está vacío, entonces la derivación es exitosa; de lo contrario, la derivación es condicionalmente exitosa (dependiendo de las restricciones no lineales).

Considerando el caso donde **E** no está vacio; **E0** denota el primer elemento de **E** y **E2** denota el resto de **E**. Si **E0** es un átomo, entonces se seleccionará **E0** para la reducción del átomo. Primero, se selecciona una regla apropiada al programa. El átomo y la cabeza de la regla serán entonces igualados, dando lugar a un conjunto de restricciones, las cuales se escribirán como **{M1 & M2}** donde **M1** consiste únicamente de restricciones lineales y **M2** de las no lineales. La nueva meta consiste de:

- **{C & M1}** en el primer componente;
- **{D & M2}** en el segundo componente, y
- el cuerpo de la regla y **E2** (en este orden), en su tercer componente.

Si **E0** es una restricción lineal, entonces la meta reducida es **{ C & E0, D ?- E2 }** tal que **{ C & E0 }** es satisfactible; de lo contrario no es una meta reducible y la derivación es falla finita. Finalmente, si **E0** es una restricción lineal, entonces la meta reducida es **{ C, D & E0 ?- E2 }**. Esto es, la restricción **E0** simplemente se difiere.

3.3.2 Reducciones excitadas

Consideremos a **{ C, D ?- E }** como la meta a manipular**.** Este paso de reducción inicia considerando que hay una restricción diferida **D0** en **D** la cual es lineal. Esto es, **C** implica que **D0** es equivalente a una restricción lineal. Si no hay tal restricción diferida, entonces no se ejecuta ninguna reducción.

Por otro lado, consideremos el caso en el cual **C** es inconsistente con esta restricción lineal. Aquí la reducción no es posible y se obtiene una derivación de falla finita. Sin embargo, si **C** es consistente con la restricción lineal, entonces la meta reducida es **{C & D0, D2 ?- E}** donde **D2** es el resultado de borrar **D0** desde **D**.

3.4 Meta - Programación

En el contexto de **Prolog**, la meta - programación se refiere a la destrucción y construcción de reglas y términos, y al examen y modificación de la base de reglas. Del mismo modo lo interpreta **CLP®**. Sin embargo, son necesarias algunas características

extras debido a la naturaleza especial de los términos aritméticos y en las restricciones. Ya que, sin tales características y modificaciones extras, no habría forma de que un programa en **CLP®** distinguiera los términos **p(3-1)** y **p(1+1)** dado que son semánticamente idénticos.

Más específicamente, las características y modificaciones extras son necesarias para: a) hacer que los términos aritméticos se interpreten sintácticamente introduciendo un modelo codificado; b) convertir los modelos codificados de los términos aritméticos en los términos aritméticos apropiados; c) obtener un modelo codificado del conjunto de restricciones actual; d) agregar restricciones apropiadas para hacer valer las reglas y, e) examinar por completo la base de reglas.

3.4.1 Macro-operadores "*quote/eval*"

El argumento del operador ***quote*** se traduce a una versión en la cual todos los operadores aritméticos se convierten a un formato de código especial, el cual no es, por otro lado, directamente accesible al programador. Este formato de código se puede tratar como una función término. Además, el operador ***quote*** pasa a través de todos los símbolos de las funciones, constantes, variables, sin modificarlos. Así por ejemplo, la regla

```
q( X, Y ) :- X = quote( f( g( Y ), 2 * Y ) ).
```

se vuelve

```
( X, Y ) :- X = f( g( Y ), 2 * Y ).
```

La forma original de la regla siempre se muestra cuando se enlista la base de datos, pero cuando se imprime un término, los símbolos de las funciones codificadas se imprimen precedidas por un **^.**

El operador ***eval***[27], convierte un término codificado en el término que codifica. Este pasa a través de los símbolos de las funciones no interpretadas, sin afectarlas, del mismo modo que para las constantes y símbolos de funciones interpretadas. Por ejemplo, la meta

```
?- X = f( a, g( c ) ), U = eval( X ).
```

resulta que tanto **U** y **X** se vuelven **f(a, g(c))**. Sin embargo en

```
?- X = f( Y, g( c ) ), U = eval( X ).
```

resulta que **U** se vuelve **f(eval(Y),g(c))**, siendo esta la mejor represen-tación de los términos que contienen a ***eval***.

27 Este operador es una aproximación al mecanismo propuesto en [HE89].

CLP® implementa un algoritmo parcial el cual mantiene las restricciones, tal que ***eval*** aparece únicamente de la forma **X = eval(Y)**, estas ecuaciones se difieren hasta que el argumento de ***eval*** se construye. De hecho, el retraso de tal ecuación ***eval*** se implementa de la misma forma como lo son las ecuaciones no lineales. Por ejemplo, consideremos la meta

```
?- X = quote( U + 1 ), eval( X ) = 5, Y = eval( U ) - 5.
```

Después de la primera restricción, **X** es igual a **U + 1**, pero después de la segunda restricción, ***eval*** va a través de **X**, tal que obtenemos la restricción **eval(U) + 1 = 5**, lo cual se simplifica a **eval(U)=4**. Aquí la tercera restricción nos da por resultado que **Y** se vuelve **-1**.

Sin embargo, si la meta fuera cambiada a

```
?- eval( X ) = 5, Y = eval( U ) - 5, X = quote( U + 1 ).
```

La primera y la segunda restricción de ***eval*** se diferirían. La tercera restricción excitaría al primer ***eval*** diferido dado que **X** está ahora aterrizado, dando como resultado la restricción **eval(U)+1=5**, la cual, junto con el segundo ***eval*** diferido - el cual está ahora excitado - darían como resultado que **Y** se vuelva **-1** otra vez.

3.4.2 Predicados "*rule*", "*retract*" y "*assert*"

Como el predicado **clause** de **Prolog**, la cláusula aquí se vuelve el predicado ***rule*** tal que la meta **?- rule(H, B)** se comporta como si fuera el hecho **rule(E,F)** para cada regla **E :- F** en el programa (y por supuesto **rule(A, true)** para cada hecho **A**).

Hay sin embargo, un aspecto de ***rule*** el cual no es análogo a **clause**; los símbolos de las funciones aritméticas se codifican. Más precisa-mente, el predicado ***rule*** se vuelve un hecho **rule(quote(E),quote(F))** para cada regla **E :- F** en la base de reglas (y **rule(quote(A),true)** para cada hecho **A**).

En un modelo similar, el predicado ***retract*** de **CLP®** es como su similar en **Prolog** pero se diferencian en que **CLP®** iguala los símbolos de las funciones aritméticas con sus modelos codificadas.

assert en **CLP®** difiere del de **Prolog** en que no solamente codifica términos, adicionalmente las restricciones se pueden agregar a la regla que se está haciendo valer. De forma más general, este predicado aporta una técnica de evaluación parcial de restricciones.

Esta técnica consiste en realizar una pregunta y entonces, usando el formato simplificado de la restricción respuesta construir nuevas reglas. Estas nuevas reglas representan una rama del programa con respecto a esa pregunta. Por ejemplo:

```
resistor( V, I, R ) :- V = I * R.

?- resistor( V, I1, R1 ), resistor( V, I2, R2 ), I = I1 + I2,
assert( parallel_resistors( V, I, R1, R2 ) ).
```

Los resultados en la aserción de la regla describen la relación equivalente del voltaje - corriente de un par de resistencias conectadas en paralelo.

```
parallel_resistors(V, I, R1, R2) :- V =I2 * R2, V =(I - I2)*R1.
```

3.5 Salida en falso (Dump)

Una característica importante del sistema **CLP®** es su habilidad para mostrar la salida de las restricciones de una derivación exitosa en un formato más simple. En una derivación típica, cientos de restricciones pueden agruparse e imprimir una salida sin haberse simplificado, lo cual conduciría a una respuesta inútil.

Cuando una derivación tiene éxito, se invoca al módulo de salida de **CLP®** para imprimir las restricciones que relacionan a las variables en la meta. El módulo también puede invocarse usando el predicado **dump([X, Y, …, Z])**. El sistema **CLP®** intenta simplemente **proyectar** las restricciones de dos formas: proyectando las restricciones sobre el conjunto de variables objetivo (aquellas que aparecen en la meta original o son dadas por el usuario en el argumento de ***dump***), y eliminando la redundancia en las restricciones[28].

3.5.1 Perfil del algoritmo

Recordemos que hay cuatro formas diferentes de restricciones: funciones restrictivas (**X = f(Y, a, g(Y))**); ecuaciones lineales (**3*X + 4*Y = 6**); desigualdades lineales (**3*X > 4 + Y**); y ecuaciones no lineales (**X = Y * Z, T = pow(U, V), U = eval(V)**). Las funciones se manipulan primero y de la misma forma que en **Prolog**. Las restricciones se almacenan en un formato resoluble usando vinculaciones e imprimiendo la variable objetivo en su representación terminal. Por ejemplo

```
?- X = f( Y, Z ),  Z = g( a, Y ), dump( [ X, Y ] ).
```

da como resultado la salida

[28] Idealmente las restricciones de la salida solamente implicarán variables objetivo y estarán libres de redundancia, pero esto no siempre es posible.

X = f(Y, g(a, Y)).

Nótese que no hay ecuación para **Y** debido a que esta es su propia representación terminal. Con las funciones restrictivas, no siempre es posible presentar la salida en términos de variables objetivo solamente, y algunas variables no objetivo se imprimen usando un nombre interno.
Por ejemplo,

?- X = f(Y, Z), Z = g(a, Y), dump([X]).

da una salida como:

X = f(_h6, g(a, _h6)).

Las **ecuaciones lineales** se usan para sustituir la salida de las variables no objetivo de la siguiente manera: Si **E** es una ecuación lineal seguida de una variable no objetivo **X**, entonces **E** será de la forma **X = t** y sustituimos **X** en todas las demás restricciones (incluyendo las funciones, ecuaciones no lineales y desigualdades). Por ejemplo

?- T = 3 + Y, X = 2 * Y + U, Z = 3 * U + Y, dump([X, T, Z]).

Primero eliminamos **Y** usando la primera ecuación **Y = 3-T,** entonces

X = 2 * T - 6 + U, Z = 3 * U + T - 3.

eliminamos **U** usando la primera ecuación y obtenemos

Z = 3 * X - 5 * T + 15.

esta es la respuesta final debido a que solamente las variables **T** y **Z** permanecen. Esta clase de eliminación continúa, y eventualmente la ecuación final se usa para eliminar **T**. Esta es la respuesta correcta ya que no hay restricción directa entre **X** y **Z**, o más formalmente, dado que en cualquier par de valores para **X** y **Z,** las restricciones se vuelven satisfactorias.

Las **desigualdades lineales** son más difíciles de manipular que las ecuaciones lineales [SC86][JB92]. La eliminación de variables desde desigualdades puede ser cara y la proyección puede contener un número exponencial de desigualdades. Primero, estas no permiten eliminar variables de otra parte que no sea el conjunto de restricciones. Segundo, mientras en las restricciones manipuladas anteriormente, la redundancia se elimina automáticamente por el resolvedor, la redundancia de las desigualdades lineales está presente y es difícil de remover.

La eliminación completa de la redundancia sobre conjuntos con desigualdades lineales no es una tarea trivial y es bastante cara computacionalmente. Una solución fue descrita por Kohler [KO67]; quien desarrolló un examen - no caro - que detecta muchas restricciones redundantes producidas en la eliminación Fourier-Motzkin[SC86], el cual permite detectar

redundancia antes de que esta se produzca. En el sistema **CLP®**, se usa una adaptación del método de Kohler [JB92].

Finalmente tratamos con las **ecuaciones no lineales**. En general, el algoritmo simplemente muestra cada ecuación no lineal a menos que se use como sustitución. Recordemos que cada restricción no lineal tiene la forma

X = Y * Z, X = sin(Y), X = cos(Y), X = pow(Y, Z),
X = max(Y, Z), X = min(Y, Z) o **X = abs(Y).**

cada una de estas ecuaciones se pueden usar para sustituir **X,** si **X** es una variable no objetivo. Por ejemplo,

```
?- Y = sin( X ), Y = cos( Z ), dump( [ X, Z ] ).
```

Conduce a la salida

```
sin( X ) = cos( Z ).
```

Como en el caso de las funciones restrictivas, en la práctica no podemos eliminar todas las variables no objetivo que aparecen en las restricciones no lineales. Como antes, se despliega cualquier variable no objetivo usando un nombre interno.

3.5.2 Sistema de predicados {dump}

La característica básica para la salida en **CLP®** es el sistema de predicados **dump**, cuyo argumento es una lista de variables objetivo. Para usar este predicado, las variables objetivo tienen que aparecer explícitamente en el argumento (como en **dump([A, B])** y no ser pasados (**X = [A, B], dump(X)**). Esto es porque los nombres de las variables objetivo son de hecho usadas en la salida. El orden de las variables en la lista se usa para especificar una prioridad con respecto a las variables posteriores que tienen una prioridad mayor.
Desde las restricciones salida de ***dump***, hay muchas formas equivalentes del mismo conjunto de restricciones y la prioridad nos da un modelo de control con respecto a la salida ***dump***.
El predicado **{dump}** es un refinamiento de ***dump***, que resulta para ser más flexible. Su primer argumento, como antes, es una lista de variables objetivo. Su segundo argumento es una lista de constantes usadas en lugar de las variables objetivo originales en la salida. Por ejemplo,

```
?- Nombre=[a,b], Destino=[X, Y],  X > Y,  dump(Destino,Nombre).
```

da una salida **a > b**. Este predicado es útil cuando los nombres de las variables objetivo se conocen únicamente en el tiempo de corrida.

Más precisamente, la operación ***dump*** es como sigue: supóngase que el primer y el segundo argumento es **[t_1, …, t_n]** y **[u_1, …, u_n],** donde **t_i** y **u_i** son términos arbitrarios. Se construyen nuevas variables **T_1, …, T_n** y se agregan a la colección

actual de ecuaciones restrictivas **T_1 = t_1 … T_n = t_n**. Ahora obtenemos una proyección de las restricciones aumentadas **T_1, …, T_n**. Finalmente, la salida de esta proyección renombra cada variable objetivo **T_i** por su nuevo nombre **u_i**.

En la meta-programación puede ser útil obtener la forma codificada de las restricciones con respecto a las variables objetivo dadas. Esta característica se proporciona por el sistema de predicados **{dump}**.

{dump} consiste de tres argumentos: las variables a ser proyectadas (1er argumento) y las variables que reciben la forma codificada (3er argumento). El 2do argumento es una lista de términos que sirven para reemplazar las variables originales, y de aquí que la longitud de las dos listas tenga que ser la misma. Por ejemplo:

```
?- NuevaVariable = [ A, B, C ], Destino = [ X, Y, Z ], X > Y + Z,
dump( Destino, NuevaVariable, Respuesta ).
```

Dando como resultado **Respuesta = [-A + B + C < 0]**.

Existen dos razones para tener tal segundo argumento. Primero, es muy inconveniente manipular formas codificadas que contengan variables que todavía tienen las restricciones aritméticas originales.

Segundo, en muchos casos es más conveniente manipular las representaciones nulas de los modelos codificados. Esto es, constantes sintácticas que reemplazan a las variables. Los términos resultantes pueden ser substituidos entonces por las variables originales (u otras) más fácilmente.

3.6 Usando el Sistema

La interfaz de usuario en **CLP®** es muy parecida al estilo del intérprete de **Prolog**. En otras palabras, es muy posible que al usar este sistema casi se ignore por completo el hecho de que está basado en un compilador. Todas las metas se compilan (rápidamente) antes de que se ejecuten y cualquier archivo consultado es compilado inmediata-mente. La base de reglas siempre está disponible para inspeccionarse (excepto con reglas protegidas) y se puede modificar dinámicamente en tanto las relaciones se hayan declarado dinámicas.

3.6.1 Argumentos en la línea de comandos

La sintaxis de una línea de comando es

clpr [opciones] [nombre del archivo]

donde el nombre del archivo contiene un programa **CLP®**. Las opciones disponibles son:

ouritem{ -cs < n > } especifica el tamaño del código (default 128,000)
ouritem{ -hs < n > } especifica el tamaño de la pila (default 200,000)

ouritem{ -ls < n > } especifica el tamaño de la pila local (default 100,000)
ouritem{ -ss < n > } especifica el número máximo de variables solubles (default 128,000)
ouritem{ -ts < n > } especifica el tamaño de arrastre (default 100,000)
ouritem{ -z < r > } coloca la noción interna de cero a este número[29].
ouritem{ -r < int > } especifica una secuencia de números aleatorios.

3.6.2 Nombres de archivos

Los nombres de archivo consultados o leídos como una cadena de entrada pueden tener implícito un sufijo opcional el cual se agrega al nombre del archivo. El sufijo por default usualmente es **'.clpr'** (**'.clp'** para **MS/DOS** u **OS/2**) dependiendo de la instalación. Este se puede cambiar usando la variable de ambiente **CLPRSUFFIX**, la cual puede colocar una lista de sufijos separados por comillas, ("**.clpr : .clp**"). Primero, el nombre del archivo original se prueba y si este no puede ser leído entonces se agrega un sufijo en el orden especificado por la lista de sufijos.

3.6.3 Respuestas posibles

Después de que el sistema se ha inicializado, este solicitará del usuario una pregunta. Continuamente aceptará metas del usuario y las solucionará hasta que se termine la sesión con un alto o bien, si encuentra un fin de archivo. (**ctrl-D** en UNIX o **ctrl-Z** en MS-DOS).

Si el usuario falla en la meta, entonces se desplegará un mensaje de {***** No**}, de lo contrario la pregunta es exitosa y el resultado es una restricción (las restricciones sobre las variables en la meta).

Una meta exitosa también desplegará un mensaje de {***** Yes**}, pero si hay otras alternativas por tratar en la meta, entonces se desplegará un mensaje de {***** Retry**} esperando a que se teclee **enter**, '**.**' o '**n**' para aceptar las respuestas, o bien '**;**' o '**y**' para causar un retroceso.

Un prompt diferente se despliega si permanecen aún restricciones (no lineales) diferidas al final de la ejecución. El mensaje {*****Maybe**} reemplaza al {***** Yes**} y {*****(Maybe)Retry?**} reemplaza al {*****Retry?**} para indicar que la satisfactibilidad de las restricciones no lineales que permanecen aún no ha sido decidida por **CLP®**.

La ejecución de una pregunta se puede interrumpir a cualquier tiempo usando la tecla de interrupción (**Ctrl-C** usualmente)[30] . Un buffer de al menos 50 metas permanece, y se puede examinar por el predicado **history** (o **h**). Una pregunta antigua se puede ejecutar únicamente introduciendo su número de historia como una meta (**?- 5.**).

3.6.4 carga/consulta y reconsulta de programas

[29] Los números entre **r** se toman a ser equivalentes a cero.
[30] No es seguro interrumpir la ejecución de una pregunta ya que puede ocasionar que el sistema se vuelva inconsistente internamente.

Un programa fuente **CLP®** se puede cargar usando el predicado "**consul**" o la notación más conveniente [lista de nombres de archivos]. Al cargar un programa se compilan todas las reglas de ese programa, haciendo que los nuevos predicados estén disponibles para su uso y también se ejecuta cualquier meta implantada. A diferencia de algunos sistemas de **Prolog** donde los archivos consultados se interpretan y se compilan, todos los predicados consultados en **CLP®** se recompilan (usualmente de forma más rápida).

Los nombres de archivo constan de caracteres en minúscula y de cualquier otra clase de nombre (una ruta, la cual se tendría que poner entre comillas simples **'**).

Al reconsultar un archivo con "**reconsult**" o la notación **[lista de archivos]**, si se encuentran definiciones previas, estas serán borradas y se reemplazaran por las nuevas variables[31]. Algunos sistemas en **Prolog** usan una notación alternativa, **[-nombre_del_archivo]** pero en **CLP®** se tendrían conflictos con el signo menos. También en algunos sistemas, el consultar y reconsultar se combinan. En **CLP®** consultar un archivo previamente consultado con definiciones activas dará como resultado un mensaje de advertencia y las redefiniciones se ignorarán. El nombre de archivo especial **user** denota que el archivo a ser consultado o reconsultado se lee desde una entrada estándar. Este permite la entrada directa de reglas las cuales están accesibles a modificaciones rápidas desde el nivel superior de la pregunta.

3.6.5 Verificación y advertencias

Los programas en **CLP®** se pueden verificar de manera opcional contra algunas convenciones estilísticas, también llamadas **verificación de estilos**. El propósito de la verificación de estilos es dar una advertencia de que el programa puede contener algunas **fallas** comunes cuando las reglas de estilos no se siguen[32] .

Hay tres clases diferentes de verificación de estilos: **single_var, discontiguous, name_overload**.

single_var. Advierte si una variable se usa una sola vez con una regla e indica que una variable ha sido suplantada. Las variables anónimas (_) y también las variables mezcladas con un subrayado se ignoran. Un ejemplo de este error es la regla **p(X,Y)** que da el siguiente mensaje:

```
Warning: Style check, singleton variables, rule 1 of q/2+++ X, Y
```

discontiguous. Asume que todas las reglas diferentes que definen un predicado ocurren de manera contínua con un archivo y advierte si hay otra regla interviniendo. La falla común (la cual resulta cuando la verificación de estilos no se sigue) puede suplantar el nombre de una regla, o substituir un '**.**' al final de la regla cuando en su lugar corresponde una '**,**'. Por ejemplo el programa

```
p(X) :- X > 0.   q( X ).   p( 0 ) :- r( X ).
```

[31] Por default, un predicado el cual se redefine, genera una advertencia. Esta se puede deshabilitar ejecutando el sistema de predicado {warning} (redefine_off).

[32] es importante recordar que estos son solamente advertencias y que un programa puede ser perfectamente correcto.

donde debería haber una coma antes de **q** dando como resultado, el siguiente mensaje:

Warning, < stdin > : 1 Style check, **p** is not contiguous.

name_overload. Verifica si el mismo nombre del predicado se define con diferentes aridades. Por ejemplo el programa **p(0,0). p(1). p(2,2)**. Nos da el siguiente mensaje de advertencia:

Warning: rule overloading, same name, different arity: +++p.

Nótese que cuando esta opción se ha deshabilitado y luego rehabilitado, las reglas que se hayan definido antes generarán advertencias.

La opción "**all**" abarca los tres estilos. Por default, la verificación de estilos se encuentra activada y la verificación de estilo individual se puede activar o desactivar con "**style_check**", "**no_style_check**". Las advertencias adicionales pueden deshabilitarse usando el sistema de predicados especial "**clear_style_check**". "**style_check(all_reset)**" limpia todas las advertencias previas y activa la verificación de estilos.

Otra clase de advertencia se da cuando una regla se define en más de un archivo. La unidad básica de compilación es un solo archivo y todas las ocurrencias de la regla para un predicado tienen que definirse sin ser del mismo archivo.
La excepción se da cuando un archivo es reconsultado; las nuevas definiciones reemplazan a las antiguas. El compilador simplemente ignora todas las adiciones a un predicado existente y por defecto dará una advertencia.

3.6.6 Sesión ejemplo

Esta es una sesión ejemplo del sistema **CLP®**. Se da información extra en forma de comentario después del carácter **%**.

```
clpr

CLP( R ) Versión 1.2 (c) Copyright International Business
Machines Corporation 1989 (1991) All Rights Reserved

1 ?- f( X, Y ) = f( g( A ), B ). % una simple "unificación"
B = Y, X = g( A )
*** Yes

2 ?- X = Y + 4, Y = Z - 3, Z = 2. %una simple evaluación aritmética
Z = 2   Y = -1   X = 3
*** Yes

3 ?- X + Y < Z,  3 * X - 4 * Y = 4, 3 * X + 2 * Y = 1.
Y = -0.5   X = 0.666667   0.166667 < Z
*** Yes
```

4 **?- X + Y < Z, 3 * X - 4 * Y = 4, 2 * X + 3 * Z = 1.**
Y = -1.125 * Z - 0.625 X = -1.5 * Z + 0.5 - 0.0344828 < Z
***** Yes**

5 **?- history.**
1 f(X, Y) = f(g(A), B).
2 X = Y + 4, Y = Z - 3, Z = 2.
3 X + Y < Z, 3 * X - 4 * Y = 4, 3 * X + 2 * Y = 1.
4 X + Y < Z, 3 * X - 4 * Y = 4, 2 * X + 3 * Z = 1.
***** Yes**

6 **?- 2**. % corre la segunda meta otra vez
X = Y + 4, Y = Z - 3, Z = 2.
Z = 2 Y = -1 X = 3
***** Yes**
7 **?- ['examples / fib'].** % consulta (carga) un programa ejemplo
goal: go
***** Yes**

8 **?- ls fib.** % visualiza el programa
fib(0, 1). fib(1, 1).
fib(N, X1 + X2) :- N > 1, fib(N -1, X1), fib(N - 2, X2).
***** Yes**

9 **?- fib(5, F).** % solamente una única respuesta
F = 8
***** Retry?;**
***** No**

10 **?- F > 7, F < 9, fib(N, F).** % pregunta por la primer respuesta
N = 5 F = 8
***** Retry?**

11 **?- ['examples / mortgage'].** % usa ' para reconsultar los ejemplos
goals: go1, go2
***** Yes**

12 **?- ls.** % visualiza toda la base de reglas

```
h :- history.

fib( 0, 1 ).  fib( 1, 1 ).
fib( N, X1 + X2 ) :- N > 1, fib( N - 1, X1 ),  fib( N - 2, X2 ).
```

```
go :- printf( \n Fib( 14 ) = , [ ] ), ztime, fib( 14,X ), ctime( T1 ), printf( ( Time = ) \n, [ X, T1 ] ), printf( Fib-1 ( 610 ) =, [ ] ), ztime, fib( Y, 610 ), ctime( T2 ), printf( ( Time = ) \n, [ Y, T2 ] ).

mg(P, T, I, B, MP) :- T = 1, B = P + P * I - MP.
mg(P ,T ,I ,B, MP) :- T > 1, mg(P*(1 + I) - MP, T - 1, I, B, MP).

go1:-ztime, mg(999999, 360, 0.01, 0, M), ctime(T), printf(Time =, M \n ,[ T, M ]).
go2:-ztime, mg( P, 720, 0.01B, M ), ctime( T ), printf( Time = \n,  [T ] ), dump( [ P, B, M] ).
*** Yes
13 ?- [ 'examples / mortgage' ].
Warning:  mg / 5 has been redefined
Sample goals: go1, go2
*** Yes

14 ?- go2. Time = 0.25
M = -7.74367e  - 06 * B + 0.0100077 * P.
*** Retry?-

15 ?- [ user ].
p( X )  :- writeln( X ).
^D
*** Yes

16 ?- p( hello ).
hello
*** Yes
```

3.7 Organización de Archivos Consultados

Un **archivo** consiste de un número de partes. Cada **parte** consiste de cero o más reglas seguidas por una meta. Esto es, una meta siempre encierra una parte, y el final de un archivo cierra la última parte si una meta no se ha efectuado.

Una **relación** puede no expandirse a más de una parte a menos que este se declare como dinámico antes de que la primera regla lo defina. Definir una relación estática en más de una parte generará un mensaje de advertencia declarando que las nuevas definiciones serán ignoradas. Sin embargo, si se está reconsultando, las nuevas definiciones se reemplazarán en las partes previos.

Puede haber tres clases de metas en cualquier archivo consultado. Las tres se consideran idénticas cuando se encuentran en un archivo fuente que está siendo consultado.

Sin embargo, son diferentes cuando un archivo fuente se compila primero y cuando el archivo **.clam** se consulta.

:- goal. Se ejecuta durante la compilación de un archivo fuente.
::- goal. Se ejecuta durante la consulta de un archivo {**.clam**}
?- goal. Se ejecuta durante la compilación y en el tiempo de corrida.
El primer tipo de meta se usa para directivas de compilación y mensajes que comprueben la equiparación mientras algún código está siendo compilado. El segundo tipo se usa para hacer que un programa corra directamente después de que éste se ha cargado. Finalmente, la tercer clase de meta es útil para declarar al operador, el cual necesita presentarse al resto del programa para desarrollarse correctamente y también cuando el programa está corriendo, tal que los términos se impriman correctamente.

3.8 Código estático y dinámico

Un programa en **CLP®** se divide en **reglas estáticas**, las cuales no cambian, y **reglas dinámicas**, las cuales permiten que la base de reglas se modifique vía "**assert**" y "**retract**" así como por la consulta. Las reglas/código estático no se pueden expandir a más de un pedazo. El código dinámico por otro lado puede definirse en cualquier lugar y las reglas dinámicas se pueden agregar haciéndose valer durante la ejecución o por la consulta de un archivo, el cual se comporta como si estas definiciones fueran válidas.
El único requerimiento para las reglas dinámicas es que el nombre de un predicado en particular haya sido predeclarado usando "**dynamic**", lo cual asegura que el uso de ese predicado sea dinámico. Por ejemplo **:- dynamic(foo, 2)**. El primer argumento es el nombre del predicado y el segundo es su aridad[33].

Cada declaración dinámica tiene que efectuarse antes de que se haga uso de cualquier predicado dinámico, de lo contrario se generará un error con cualquier predicado precedido de este, el cual se asume que es estático.

El declarar un predicado como dinámico nos permite usar el predicado "**rule**" para inspeccionar la base de reglas, "**assert**" para agregar nuevas reglas y "**retract**" para borrar reglas.

La semántica operacional de la familia del sistema de predicados de "**assert**", "**rule**" y "**retract**", es que cualquier modificación a la base de reglas esté disponible inmediatamente para su uso. "La semántica operacional del código dinámico puede variar considerablemente entre los diferentes sistemas **Prolog** dado que no se podría colocar un hecho confiable en este" [LI87].

Finalmente, no hay diferencia entre el código estático y el dinámico ya que ambos pueden usarse de manera intercambiable (ambos pueden enlistarse con "**ls**"). El código dinámico también se compila pero general-mente no es tan eficiente como el código estático y también es menos determinístico.

3.9 Depuración

[33] La mayoría de los Prologs usan la convención nombre/aridad para especificar esto, pero este puede confundirse con la división, de aquí que los dos argumentos se declaren de esta forma.

Las características de depuración en esta versión de **CLP®** son rudimentarias.

codegen_debug. Esta es una directiva del compilador, el cual incluye instrucciones de depuración en un código generado consecuentemente. Debe activarse antes de que el archivo a ser depurado se consulte.

codegen_nodebug. Esta es una directiva del compilador que desactiva la generación del código depurado en una compilación subsecuente.

spy. Asegura que todas las relaciones compiladas bajo "**codegen_debug**" sean visibles al depurador. Las reglas protegidas nunca son visibles.

spy(+P, +A). Asegura que la relación del predicado **P** con aridad **A** sea visible al depurador si este fuera compilado bajo "**codegen_debug**". No puede aplicarse a relaciones protegidas.

spy([P1(+A1), …,Pn(+An)]). Como "**spy**", excepto que se suministra una lista de predicados a ser visualizados, donde P_i son los nombres del predicado y A_i su aridad.

nospy. Hace todas las relaciones invisibles al depurador.

nospy(+P, +A). Hace la relación del predicado **P** con aridad **A**, invisibles al depurador.

nospy([P1(+A1),…,Pn(+An)]). Como "**nospy**", excepto que se suministra una lista de predicados a ser invisibles.

trace. Activa la impresión. Todos los intentos subsecuentes para buscar una relación visible al depurador resultarán en un mensaje impreso. El mensaje es el mismo a pesar de que este sea su primer o subsecuente intento de satisfacer una meta.

notrace. Desactiva la impresión.

Notas sobre la eficiencia

Algunas características clave que pueden afectar significativamente a la eficiencia pueden ser en general inadvertidas, y por lo tanto se deben tomar con mayor cuidado cuando se usan.

Indexación: CLP® emplea primero un argumento de indización para construir las funciones término así como a los números reales, el usar el indizado puede dar como resultado un pequeño aumento en la velocidad.

Vinculación Recursiva: La última llamada a optimización se emplea, y por lo tanto hace que la vinculación recursiva no incremente el uso de pila local. La disyunción lógica (**;**) e **if-then-else** (**->**), se implementan a un meta nivel y por lo tanto, no son del todo eficientes.

Igualdades implícitas: La solución a desigualdades que implican algunas desigualdades implícitas, se puede controlar usando los predicados: "**implict**", "**noimplicit**", y "**partial_implicit**" los cuales se detallan más adelante.

Manteniendo la regla: El predicado "**assert**", implica incorporar restric-ciones que relacionen a las variables en esa regla. Esto es menos eficiente que si las restricciones no fueran tomadas en cuenta. La familia de predicados especiales "**fast_assert**", ejecuta el manteni-miento sin incorporar restricciones aritméticas como en **Prolog**.

3.10 Sistema de Predicados

3.10.1 Base de reglas

op(+P,+T,+S). Declara el átomo **S** como un operador del tipo **T** con precedencia **P**. El tipo puede usarse para especificar el prefijo, postfijo y los operadores binarios usando la notación posicional: **fy, fx, yf, xf, yfy, xfy, yfx, xfx**; donde **f** especifica el operador, **y** y **x** los argumentos. Una **y** especifica que la función/operador más próxima en la subexpresión será de igual o de menor precedencia que el operador **f**, y **x** especifica que es estrictamente menor. La precedencia tiene un rango de 0 a 1200 donde una precedencia 0 remueve al operador.

listing. Lista las reglas de la base de reglas que son visibles.

listing +P. ls +P. Lista las reglas visibles actuales para el predicado **P**.

consult(+F). [+F] . Lee el archivo **F** y agrega las reglas que contenga a la base de reglas. Si el archivo se especifica como "**user**", entonces se usa la entrada estándar en lugar de un archivo. El formato **[F]** toma una lista de archivos mientras que "**consult**" toma solamente un archivo.

reconsult(+F). Lo mismo que "**consult**", pero si un predicado ya tiene reglas que lo definan, estas se borran antes de que las nuevas reglas se agreguen, y se imprime un mensaje de advertencia[34].

[34] Notése que **[-F]**, el cual es el equivalente para { reconsult } en los sistemas Prolog, no se puede usar dado que esto significa **F** negativo.

retract_all. Borra por completo la porción de la base de reglas que no esté protegida.

retract_all(+H). Borra todas las reglas visibles con cabeza igual a **H**. El código estático no se puede borrar con "**retract_all**".

asserta(+R). Agrega la regla **R** a la base de reglas antes de que todas las demás definan al mismo predicado[35] .

assertz(+R). Agrega la regla **R** a la base de reglas después de que todas las demás definan al mismo predicado.
rule(+H, ?B). Cierto si la regla **H :- B** está actualmente en la parte visible de la base de reglas. Encontrando la siguiente regla igual en el retroceso. Nota que las reglas en la base de reglas se codifican antes de que la equiparación se realice.

deny(+H, ?B). Borra la regla equiparada a **H :- B** desde la parte visible de la base de reglas. Es similar a "**retract**" y tanto **H** como **B** son términos codificados.

retract(+R). Borra la regla equiparada **R** de la parte visible de la base de reglas. Como "**rule**", esta tiene una vista codificada de la base de reglas.

prot(+P, +A). Protege todas las reglas para el predicado **P** con aridad **A** en la base de reglas. Esto se parece a un sistema de predicados del usuario. En particular, no se pueden listar, agregar o borrar.

prot([P1(+A1), …, Pn(+An)]. El mismo efecto que "**prot**", sólo que este toma una lista de predicados **Pi** con aridades **Ai**.

Control { ! } { cut }. El predicado corte (o **cut**). Como es usual, su uso no se recomienda. Con frecuencia, es más apropiado usar "**once**".

fail. Siempre falla.

true. Siempre exitoso.

repeat. Siempre exitoso, aún en retroceso

+B1, +B2. Conjunción lógica.

+B1; +B2. Disyunción lógica

3.10.2 Meta-nivel

call(+X). El meta - nivel "**call**", se comporta como si el predicado **X** apareciera directamente en el cuerpo de una regla o meta. Tanto el código estático como el dinámico

[35] Los términos codificados se vuelven no codificados en la base de reglas.

se pueden usar con "**call**". En esta versión, un "**cut**" en lugar de un "**call**" se ignora. También, "**printf**" y "**dump**" no se pueden usar en lugar de "**call**".
not(+X) { call }. Negación insegura. Esta se implementa usando "**call**" así que es más lenta.

dump(+L1, ?L2, ?L3). Similar a ***dump***; el primer argumento **L1** representa las variables destino y el segundo argumento **L2** representa variables nuevas. La diferencia con ***dump*** es que la proyección está codificada y no se muestra como la salida, sino que esta se construye como el tercer argumento **L3**[36].

once(+X). Equivalente a { **call(X),** ! }. Solamente la primera respuesta a la pregunta **X** se considera.

nonground(?X). Cierto si **X** no es un término nulo.

ground(?X). Cierto si **X** es un término nulo.

nonvar(?X). Cierto si **X** no es una variable.

var(?X). Cierto si **X** es una variable. Esta puede encontrarse en una restricción aritmética que no se ha construido o vuelto nula.

?X == ?Y. Cierto si **X** y **Y** son exactamente el mismo término. En particular, variables en posiciones equivalentes tienen que ser idénticas.

atom(?X). Cierto si **X** es un átomo. Esto es, una función constante (incluyendo la lista vacía).

atomic(?X). Cierto si **X** es un átomo o número real.

functor(?X). Cierto si **X** se construye con una función.

real(?X). Impone como restricción que **X** tome un valor real; esto es equivalente a cualquier tautología aritmética de una restricción que implique **X**. Por ejemplo **X + 0 = X**.

arithmetic(?X). Cierto si **X** se restringe a tener un valor real. Esto es sólo un examen pasivo, opuesto a "**real**".

?T = .. ?L . T es un término y **L** es un término expandido a una lista (también conocido como "**univ**"). Este predicado puede usarse tanto para descomponer como construir términos. Para su uso, el primer argumento tiene que construirse o bien, el segundo argumento tiene que ser una lista de longitud fija cuyo primer elemento sea una función constante.

[36] Nota que **dump** no cambia la colección actual de restricciones.

functor(?T, ?F, ?A). **T** es un término, **F** y **A** son el nombre y aridad de la función principal **T**. **T** tiene que construirse o **F** tiene que ser una función constante (no un número real) y **A** tiene que ser un entero no negativo.

arg(+N, +T, ?A). **A** es el n-ésimo argumento del término **T**. **N** tiene que ser un entero positivo y **T** un término compuesto. Si **N** está fuera del rango, la llamada falla.

occurs(-V, ?T). **V** es una variable ocurriendo en el término **T**.

floor(+R, -I). **R** tiene que ser un número real, e **I** es el entero más grande menor que o igual a **R**.

dynamic(+P, +A). Declara el predicado **P** con aridad **A** como dinámico, tal que se pueden agregar o borrar reglas.

3.11 Entrada/salida

En esta sección, las variables no nulas se imprimirán con un nombre específico (como los del argumento de "**dump**"). Si no se especifica el nombre, entonces se imprimirán con los siguientes formatos:

- **_h\d** pila variable
- **_s\d** pila local variable
- **_t\d** variable paramétrica en el resolvedor
- **_S\d** variable lenta en el resolvedor

Las características de la **entrada/salida** son las siguientes:

dump(+L). Enlista el conjunto de restricciones en la salida actual, proyectadas con respecto a las variables destino de la lista **L**. La lista **L** tiene que ser explícitamente suministrada, esto es, se escribe sintáctica-mente como el argumento de "**dump**". El orden de las variables en la lista se usa para representar la prioridad de las variables destino.

dump(+L1, +L2). Una versión más flexible de "**dump**", sin su restricción sintáctica. Su primer argumento **L1** representa las variables destino, y su segundo argumento **L2**, el cual tiene que ser nulo, representa los nuevos nombres a usarse en la salida. Los elementos de estas dos listas pueden ser términos arbitrarios.

nl. Envía un carácter de nueva línea a la salida actual.

print(?T), write(?T). Imprime el término **T** de acuerdo a las declaraciones en la salida actual.

writeln(?T). lo mismo que **{write(T), nl}**.

printf(+F,+L). Imprime los términos de la lista **L** de la salida actual con el formato dado por la cadena **F**. El desarrollo es similar a la función "**printf**" en **C**. Todos los caracteres

excepto el carácter especial **escape** o la ruta de los argumentos se imprimen sin cambio alguno en la salida.

Los caracteres especiales de escape empiezan con un "**sl**" y son:

center{ |I|I| } { iXXX }. El carácter es representado por el número octal **iXXX,**
{sl n}: una línea nueva,
{sl r}: un return,
{sl b}: retroceso (backspace),
{sl f}: una forma alimentada,
{sl { t X }}: cualquier otro carácter, **{i X}** aparecerá sin cambio en la línea **h.**

El patrón de los argumentos inicia con "****" y denota el formato para cada término correspondiente a la lista **L**. Un "****" denota un simple porcentaje. De lo contrario, el formato toma el modelo de un ancho de campo y una precisión opcional seguido por uno de los caracteres de conversión impresos en **C**.
Más precisamente este puede describirse con la expresión regular :

[[-] [0 - 9] *] [sl. [0 - 9] *] [\ ]

Los especificadores integrales imprimirán los números reales, los cuales han sido redondeados a enteros usando la regla de redondeo par.
Una lista vacía es necesaria si ninguna variable se ha impreso. Por conveniencia, un solo "****" puede usarse en lugar de un formato específico de un argumento y un formato apropiado de ese argumento en particular (con números, el formato por default es **\g**). Por ejemplo, **printf(" X = Y = 3.2 g{ sl } n ", [X, Y])**.

printf_to_atom(?A, +F, +L). Como "**printf**" excepto que en lugar de imprimir **A**, esta se iguala con un átomo cuya cadena es la misma que lo que podría imprimirse por otro lado.

read(-X). Lee un término de la entrada actual y liga la variable **X** a esta. Cualquier variable en el término de entrada se considera que está separada de las variables que aparecen en la regla. Si se lee un fin de archivo, el término "**?-end**" es devuelto. Finalmente, el término obtenido se encuentra entre comillas, esto es, cualquier operador aritmético es tratado sintácticamente.

see(+F). Hace a **F** el archivo de entrada actual.

seeing(?F). Cierto cuando **F** es el archivo de entrada actual.

seen. Cierra el archivo de entrada actual. Vuelve a la entrada estándar de "**user**".

tell(+F). Hace **F** el archivo de salida actual.

telling(?F). Cierto cuando **F** es el archivo de salida actual.

told. Cierra el archivo de salida actual. Regresa a la salida estándar "**user**".

flush. Suministra al buffer asociado con el archivo de salida actual.

3.12 Predicados con carácterísticas asociadas a UNIX

fork. Divide los procesos actuales. Falla en un suceso y es exitoso en el otro.

pipe(+X). Crea un bus denominado **X**. Para usarse con "**see**", "**tell**".

edit(+F). Invoca al editor por default en el archivo **F**, y entonces reconsulta al archivo. Bajo el ambiente UNIX, el editor por default es **vi**, bajo MS/DOS y OS/2 este es **edit**.

more(+F). Corre el archivo **F** en páginas.

halt. Salida del sistema **CLP®**

clpr. Cierto. Usado para verificar si el programa está ejecutándose en el sistema **CLP®.**

abort. Aborta la ejecución de la meta actual.

sh. Invoca una imagen de **sh** sobre sistemas UNIX. En MS/DOS u OS/2, inicia un subshell de command.com

csh. Invoca una imagen de **csh** bajo sistemas UNIX. Bajo MS/DOS u OS/2 se desarrolla igual que **sh**.

oracle(+F, +P1, +P2). Corre el archivo binario ejecutable **F** y coloca un "bus" **P1** para escribir los procesos y un bus **P2** para leer desde los procesos. Estos "buses" se atan a los procesos de entrada y salida estándar respectivamente.

3.12.1 Predicados diversos

history. Imprime las últimas 50 metas en líneas de comandos.

history +N. Imprime las últimas **N** metas en líneas de comandos.

h. Abreviatura para history.

N. Corre la meta de la línea de comando a la posición **N** en la lista de "**history**". Este puede usarse únicamente como comando de alto nivel.

new_constant(+A,+N). Coloca la constante **A** al valor **N**. El nombre de la constante se especifica sin un # (**?.- new_constant(my_constant, 5)**). Se imprime una advertencia si el valor de una constante conocida se cambia. La advertencia puede desactivarse usando "**warning_off**".

srand(+X). Coloca un número aleatorio descendiente al número real **X**.
rand(-X). Genera un número aleatorio **0** y **1** inclusive, y lo liga a **X**.

ztime{statistics}. Inicializa a cero el contador del CPU.

ctime(-T){statistics}. Pasa a**T** al tiempo transcurrido del CPU a partir de que el contador fue puesto a cero. **T** puede no estar instanciado.

style_check(+A){contiguous}name_overload. La verificación de estilos advierte acerca de posibles errores. Este puede usarse con **A** siendo esta "**single_var**", "**discontiguous**", "**name_overload**" y "**all**". Una advertencia se da cuando una regla de verificación de estilos han sido violada. La opción "**all**" activa todos los estilos. La opción especial "**reset_all**", limpia las advertencias previas y pendientes que pueden haberse acumulado, si la verificación de estilos fue desactivada y activada en una verificación total.

no_style_check(+A). Lo contrario a "**style_check**". Desactiva las opciones para "**single_var**", "**discontiguous**", "**name_overload**" y "**all**".

clear_style_check. Limpia cualquier advertencia de verificación de estilos pendiente que pueda ocurrir cuando una verificación de estilos haya sido desactivada y activada.

warning(+A). El comportamiento cuando ocurre un error se puede modificar con un "**warning**". Cuando un error ocurre, se imprime un mensaje de advertencia de error y la ejecución es abortada. Las opciones para **A** pueden ser: "**abort**", "**continue**", "**warning_on**", "**warning_off**", "**redefine_on**" y "**redefine_off**". Las opciones "**continue/abort**" controlan la ejecución de abortar en un error. La impresión de mensajes se controlan por medio de "**warning_on**" y "**warning_off**", mientras que "**redefine_on**" y "**redefine_off**" controlan las definiciones de los predicados durante una reconsulta emitiendo una advertencia. La opción "**abort**" anula un "**warning_on**" y los mensajes de advertencia se despliegan cuando "**abort**" está activado.

3.12.1 Predicados especiales

fassert(+R). Como "**assert**" pero no toma en cuenta las restricciones a meta-nivel o restricciones aritméticas. Consecuentemente, es más rápido que "**assert**" pero menos sensitivo cuando hay restricciones implicadas. Cuando las reglas son nulas, "**fassert**" se comporta de la misma manera que "**assert**".

fasserta(+R), fassertz(+R). Igual que "**asserta**" y "**assertz**"

call(+X). Llamada a meta-nivel sobre un único predicado definido por el usuario. No permite computar metas o sistemas de predicados.

implicit. Detecta igualdades implícitas. Un conjunto de desigualdades algunas veces puede ser equivalente a algunas ecuaciones; y estas se conocen como igualdades implícitas. Un ejemplo trivial de una ecuación implícita es: **X >= 0, X <= 0.**

La bandera "**implicit**" controla estas ecuaciones implícitas que son detectadas por el solucionador de restricciones. Una ventaja al usar estas banderas es que se pueden activar o desactivar para aplicarlas entre diferentes metas que se estén ejecutando y no durante una sola ejecución actual. Otro punto importante es que, cuando hay restricciones no lineales, desactivar las ecuaciones implícitas puede conducir a restricciones diferidas que no están siendo excitadas.

noimplicit. Desactiva la detección de igualdades implícitas. Su conse-cuencia es que, restricciones diferidas las cuales pueden ser excitadas, podrían continuar diferidas.

partial_implicit. Detecta solamente algunas ecuaciones implícitas. Esta puede ser más rápida que "**implicit**"**.**

set_counter(+C, +V). set_counter. counter. Contador global el cual no cambia con el retroceso. Inicializa el contador con el nombre atómico **C** al valor de número real **V**. El nombre del contador puede ser cualquier nombre atómico.

counter_value(+C, ?V). V se iguala con el valor del contador **C**

add_counter(+C, +V). El contador **C** se incrementa por **V**.

3.12.3 Predicados faltantes

compile(+F1, +F2). El archivo **F1** invoca al compilador produciendo el archivo de tipo clam **F2**.

display(?T). Imprime **T** en el formato preestablecido en la salida actual.

printrule(+R). Imprime la regla **R** en un formato substituible en la salida actual.

printgoal(+G). Imprime la meta **G** en un formato substituible en la salida actual.

eof(+X). Cierto cuando **X** es el término significativo de la condición de fin de archivo.

libdir? D. Liga **D** al nombre del directorio en que se encuentran las librerías del archivo.

lib +F. Consulta el archivo **F** desde el directorio de librerías.

true. Indica siempre exitoso.

abort. Detiene la ejecución de la pregunta actual y retorna al comando de línea.

int(+X). Cierto si **X** es un entero.

string(+X). Cierto si **X** es una cadena.

funstr(?F, ?S). Convierte una constante **F** en una cadena **S**.

shell(+X). Invoca al shell por default con la cadena **X** como entrada.

3.13 Restricciones No Lineales y Diferidas

A continuación se describen las condiciones diferidas en varios ejemplos de restricciones no lineales. En algunas de las funciones de abajo, "**sin**", "**arcsin**", "**cos**", "**arccos**", pueden haber valores de **X** y **Z**, los cuales caigan fuera del rango de la función.
Tales valores inválidos causarán que las restricciones fallen y por default se generará un valor **fuera de rango**.

Z = X * Y . Se difiere hasta que **X** o **Y** estén aterrizados.
Z = sin(X). Se difiere hasta que **X** esté aterrizado.
Z = arcsin(X). Se difiere hasta que **X** o **Z** estén aterrizados.
Z = cos(X). Se difiere hasta que **X** está aterrizado.
Z = arccos(X). Se difiere hasta que **X** o **Z** estén aterrizados.
Z = pow(X, Y). Se difiere hasta que:

- **X** y **Y** estén aterrizados, o
- **X** y **Z** estén aterrizados, o
- **X = 1**, o
- **Y = 0**, o
- **Y = 1**.

Z = abs(X). Se difiere hasta que:

- **X** esté aterrizado, o
- **Z = 0**, o
- **Z** esté aterrizada y negativo.

Z = min(X, Y). Se difiere hasta que **X** y **Y** están aterrizados.
Z = max(X, Y). Similar a lo anterior
Z = eval(X). Se difiere hasta que **X** se construye.


```
::- op( 21, fy, '-' ).          ::- op( 21, yfx, * ).
::- op( 21, yfx, / ).           ::- op( 31, yfx, ( - ) ).
::- op( 31, yfy, + ).           ::- op( 37, xfx, < ).
```

```
::- op( 37, xfx, <= ).          ::- op( 37, xfx, > ).
::- op( 37, xfx, >= ).          ::- op( 40, xfx, = ).
::- op( 40, xfx, =.. ).         ::- op( 60, fx, once ).
::- op( 252, xfy, ' , ' ).      ::- op( 253, xfy, ; ).
::- op( 254, xfy, ( -> ) ).     ::- op( 255, fx, ( :- ) ).
::- op( 255, fx, ( ::- ) ).     ::- op( 255, fx, ( ?- ) ).
::- op( 255, xfx, ( :-) ).
```

4
Aplicaciones en CLP®

En este capítulo se muestra el uso práctico del lenguaje **CLP®**. Se presentan dos problemas diferentes; el primero trata con el Análisis de un Sistema Financiero (Swap), y el segundo representa la simulación (a nivel teórico) de un Circuito RLC en Paralelo a Corriente Directa (cd).

4.1 ¿ Porqué programar en CLP® ?

Dada la fuerza y disponibilidad de **CLP®** para permitir cálculos más rápidos y flexibles para solucionar ecuaciones lineales con cualquier número de variables desconocidas, una red permutada por ejemplo, se puede construir sólo parcialmente sin necesidad de ligar todos los parámetros de entrada, en donde el sistema retornará la relación entre sus variables. Lo mismo se puede hacer con un sistema eléctrico, un sistema matemático; ya que una de las características más sobresalientes de **CLP®** es su **flexibilidad** para programar y representar la salida.

Más información sobre sus diversas aplicaciones se puede encontrar en: la Ingeniería eléctrica [HE87][MO91], ecuaciones diferenciales [HA87][HO91], razonamiento temporal [AB92][BR91], exámenes de protocolo [GO8], análisis estructural y de síntesis [LH89], modelos basados en diagnósticos [YU91], teoría musical [TO88] y biología molecular [RY91] entre otros.

4.2 Swaps en CLP®

Los **swaps** son instrumentos financieros que permiten que dos partes intercambien pagos de interés en diferentes ocurrencias. Un **swap** es una estructura desde la cual se pueden dar intercambios como resultado de la redistribución de riesgos y suministros económicos. Generalmente un intermediario (una institución financiera), diseña e implementa la red swap, tomando un poco de la ganancia como pago. Un criterio clave para una red swap viable es que ninguna parte se torne riesgosa mas allá de su riesgo prefijado.

Hull [JH89], Shapiro [AS92], y Macfarlane [JM85], son solo unos cuantos de los expositores generales acerca de los swaps.

4.2.1 Swaps del porcentaje de interés

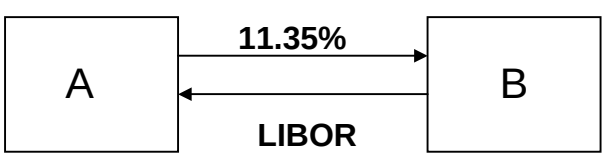


Figura 4.2.1

La figura 4.2.1 ilustra el swap del porcentaje de interés donde dos partes **A** y **B** tienen préstamos de la misma cantidad **P**. El tipo de préstamo considerado se extiende a los periodos t_i para $1 \leq i \leq n$. Los pagos se hacen (de acuerdo al interés) cada periodo t_i seguido por un pago global del último periodo t_n.

El swap respecto a **A** fija el pago de interés al **11.35%** para **B** en el intercambio de recepción flotante **LIBOR**[37] a partir de **B**.

A partir de esta simple estructura, se pueden crear redes más complejas. La siguiente figura muestra un swap dual a través de un intermediario **B**. **A** transforma un préstamo flotante a uno fijo, y **C** transforma un préstamo fijo a uno flotante. **B** anula su riesgo al paso de los pagos flotantes a través de **C** a **A**.

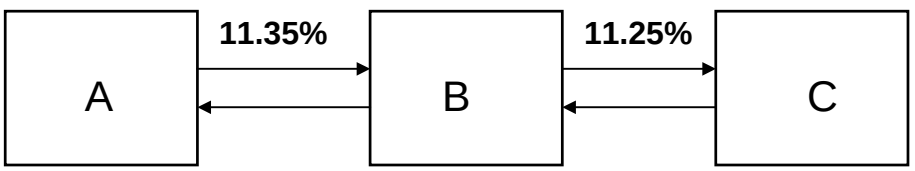


figura 4.2.2

Si se asume tanto **A** o **C** por defecto, entonces **B** no tendrá riesgos, y tomará una ganancia del **0.1%** por su servicio. Para este ejemplo, la ganancia de **B** se calcula directamente; sin embargo, en redes complejas, se necesita de una fórmula más general. Para simplificar las cosas, se puede asumir que la razón de mercado es fija a lo largo del préstamo, de lo contrario, dará lugar a ecuaciones no lineales. He aquí el uso práctico de **CLP®**.

[37] **L**ondon **I**nter**b**ank **O**ffer **R**ate - Porcentaje propuesto por el Banco de Londres

4.2.2 Análisis en CLP®

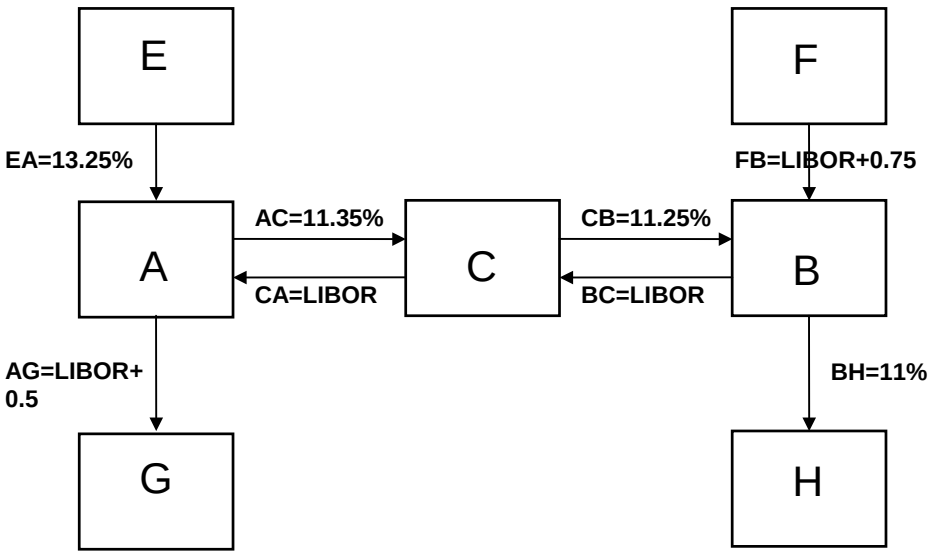


Figura 4.2.2.1

```
net1(Info, [Pi_A, Pi_B, Pi_C]) :-
        Info = [AC, CA, CB, BC, AG, BH, FB, EA],
        Pi_A = CA - AC - AG + EA,
        Pi_B = FB - BH - BC + CB,
        Pi_C = AC + BC - CA - CB.
```

La figura 4.2.2.1 muestra el swap en la razón de interés y su inserción directa a un programa en **CLP®**, donde las estructuras del préstamo son idénticas excepto para el porcentaje. Esencialmente cada nodo en la red corresponde a una ecuación que balancea la razón de interés entrando/saliendo de ese nodo.

Esta metodología simple de “porcentaje” para evaluar swaps es posible dado que las bases del préstamo y los términos son idénticos. Este modelo asume una razón de mercado fija. Una pregunta típica para este programa es:

```
?- net1( [ 11.35, LIBOR, 11.25, LIBOR, LIBOR+0.5, 11.0, [LIBOR + 0.75, 13.25 ],Pi
).

Pi = [ 1.4, 1, 0.1 ]
```

cuando el swap llama a diferentes bases o términos, el flujo de dinero individual se tiene que calcular usando la fórmula de valuación válida. Este modelo se considera en la figura A, la cual muestra la implementación en **CLP®** del flujo de dinero de la red anterior. Se invoca a los procedimientos “**loan/5**” y “**loan/6**” con una base fija de **$100** y préstamo con **5** periodos.

```
net2 ( [P, R_mkt, T], Info, Libor, [Pi_A, Pi_B, Pi_C]):-
        Info = [ AC, CA, CB, BC, AG, BH, FB, EA ],
        Pi_A = AC_CF + AG_CF - CA_CF - EA_CF,
        Pi_B = BH_CF + BC_CF - FB_CF - CB_CF,
        Pi_C = CA_CF + CB_CF - AC_CF - BC_CF,
        loan(  P, EA,         R_mkt, T, EA_CF ),
        loan(  P, AC,         R_mkt, T, AC_CF ),
        loan(  P, CB,         R_mkt, T, CB_CF ),
        loan(  P, BH,         R_mkt, T, BH_CF ),
        floan( P, CA, Libor, R_mkt, T, CA_CF ),
        floan( P, AB, Libor, R_mkt, T, AG_CF ),
        floan( P, BC, Libor, R_mkt, T, BC_CF ),
        floan( P, FB, Libor, R_mkt, T, FB_CF ).
```

Fig. 4.2.2.2 : Modelo del flujo de dinero en **CLP®**

Claramente se podría dar a cada préstamo independiente, bases y longitudes si así se desea. No es necesario definir "**Libor**", ya que este será instanciado cuando se necesite y particionado entre los 4 préstamos flotantes.
Todo término desconocido de "**Libor**" será cancelado desde el resolvedor de ecuaciones. Por ejemplo, las preguntas típicas son:

```
?- net2( [100,10,5],[11.35,0,11.25,0,0.5,11.0,0.75,13.25 ],_,Pi).
Pi = [5.3071, 3.79079, 0.379079]

?- net2( [100,10,5], [X,0,11.25,0,0.5,11.0, Y, 13.25], _, Pi).
Pi=[-4.19247*X+53.454, 4.19247*Y+1.04812, 4.19247*X-47.1653]
```

estas soluciones están en dólares y son consistentes con la solución previa en términos de la razón de interés.

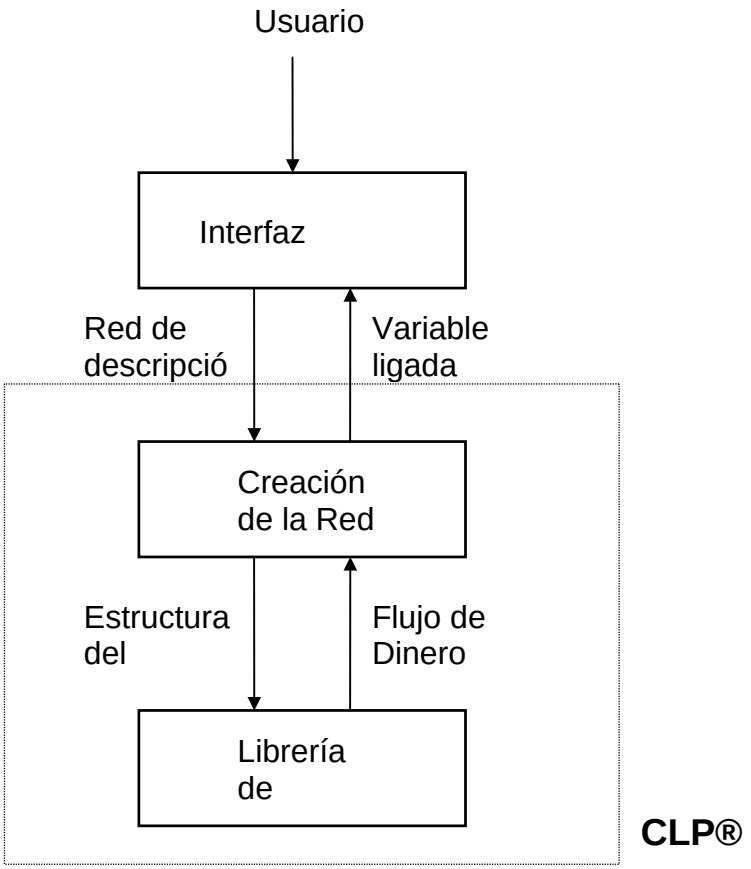


Figura 4.2.2.3 Vista global del analizador

La figura 4.2.2.3 muestra el sistema resumido del analizador financiero. La interfaz de usuario [DS94] escrita en C, acepta entradas gráficas de la red y las traduce dentro de una red de descripción aceptada por el analizador, escrita en **CLP®**. El listado del código fuente del analizador se encuentra en el apéndice A.

La red de descripción puede tener parámetros con nombres simbólicos, los cuales si se ligan serán retornados como soluciones. La ganancia se calcula y retorna para cada nodo la cual es la suma de sus flujos de dinero. Internamente al analizador mismo, la red de descripción invoca a las rutinas de la librería de préstamos que definen diferentes tipos de pagos, ya sean simples o amortizados. Estas invocaciones retornarán los valores del flujo de dinero necesarios para calcular las ganancias del valor presente.

La interfaz permite al usuario graficar específicamente la red swap, al introducir los parámetros para cada entidad (nodo) y préstamo (bordes). El usuario puede también especificar restricciones en términos tanto de parámetros de entrada como de ganancias.

La ventaja de trabajar con **CLP®** en programas financieros, como ya se ha visto, es su habilidad para diseñar y evaluar swaps de manera rápida y flexible bajo información incompleta. Las ganancias y los parámetros se pueden restringir simbólicamente para reducir el espacio de búsqueda, y el despliegue de las soluciones simbólicas ayudan a los

usuarios a intuir sus relaciones paramétricas. Estos atributos hacen a la herramienta superior a los métodos de análisis actuales, específicamente en la evaluación de **prueba y error** en diseños alternativos con hojas electrónicas[38].

4.3 Análisis de circuitos en CLP®

La simulación de circuitos con frecuencia se considera muy complicada y difícil de entender especialmente en el caso de circuitos grandes con componentes no lineales. Sin embargo, el aspecto básico de la simulación de circuitos no es complicada y de hecho, cualquiera que entienda el análisis de nodos y la teoría de la transformada de Laplace, puede fácilmente desarrollar un simulador de circuitos simple.

Este es un ejemplo de lo que **CLP®** puede hacer en la rama de los Circuitos Eléctricos. Se simula (de manera teórica) un circuito **RLC** en paralelo, caracterizado por el uso de ecuaciones diferenciales lineales de segundo orden.

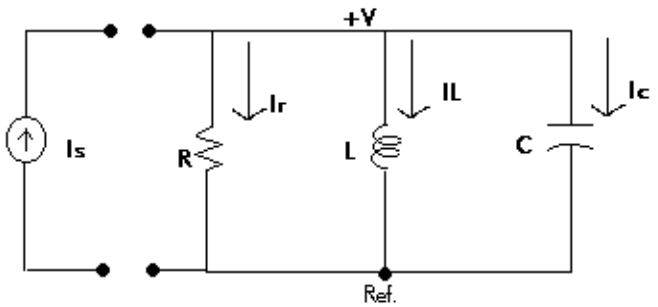
El sistema de segundo orden a analizar es, en esencia, el mismo que el de cualquier sistema mecánico. Los resultados obtenidos, por ejemplo, serían de utilidad directa para un ingeniero mecánico que esté interesado en conocer el desplazamiento de una masa sujeta a un resorte con amortiguamiento viscoso, es decir, un sistema que describa aproximadamente el movimiento vertical de un automóvil, con los amortiguadores como elementos disipativos; o para alguien interesado en el movimiento de un péndulo simple o de un péndulo de torsión.

4.3.1 El Circuito RLC

La presencia de inductores y capacitores en el mismo circuito produce por lo menos un sistema de segundo orden, es decir, un sistema caracterizado por una ecuación diferencial lineal que incluye una derivada de segundo orden, o bien dos ecuaciones diferenciales lineales de primer orden. Esto da como resultado una respuesta que tiene diferentes modelos funcionales para circuitos con la misma configuración, pero diferentes valores en sus elementos.

[38] La información presentada en esta sección se obtuvo de: Evan Tick, *Designing Financial Swaps with CLP®*. CIS-TR-91-21. November 1991. Departament of Computer and Information Science. University of Oregon.

El problema a tratar (circuito **RLC** en paralelo), se define en términos del circuito que se representa en la siguiente figura:



4.3.2 Circuito RLC en paralelo sin fuentes

El primer paso en el análisis de un sistema **RLC** consiste en calcular su **respuesta natural**. Esto se hace en forma más conveniente si se considera el circuito sin fuentes.

El primer paso para hallar la **respuesta natural**, es derivar la ecuación diferencial que debe satisfacer el voltaje **v**. Se escoge primero el voltaje porque es el mismo para todos los componentes. La corriente de una rama se puede determinar después de encontrar el voltaje, utilizando la relación **voltaje/corriente** del componente de la rama.

$$\frac{v}{R} + \frac{1}{L}\int_0^t v dt + Io + C\frac{dv}{dt} = 0. \qquad (4.1)$$

derivando con respecto a **t** y dividiendo entre la capacitancia **C** se obtiene[39]:

$$\frac{d^2 v}{dt^2} + \frac{1}{RC}\frac{dv}{dt} + \frac{v}{LC} = 0. \qquad (4.2)$$

la ecuación (4.2) se describe como una *ecuación diferencial ordinaria de segundo orden con coeficientes constantes.*

El enfoque clásico para resolver dicha ecuación es suponer que la solución tiene una forma exponencial, es decir, suponer que el voltaje tiene la forma:

$$v = Ae^{st}, \qquad (4.3)$$

[39] V.[WH93]. William H. Hayt & Jack E. Kemmerly. *Análisis de Circuitos en Ingeniería.* Quinta Edición. Pags. 196-200. McGraw Hill

donde **A** y **s** son constantes desconocidas.

Si la ecuación (4.3) es una solución de la ecuación (4.2), esta debe satisfacer la ecuación (4.2) para todos los valores de **t**. Al sustituir la ecuación (4.3) en (4.2) se obtiene la expresión:

$$As^2e^{st} + \frac{As}{RC}e^{st} + \frac{Ae^{st}}{LC} = 0,$$

o sea

$$Ae^{st}\left(s^2 + \frac{s}{RC} + \frac{1}{LC}\right) = 0, \qquad (4.4)$$

lo cual sólo puede satisfacerse para todos los valores de **t** si **A** es cero o el término entre paréntesis es cero, ya que $\mathbf{e^{st} \neq 0}$ para valores finitos de **st**.

No se puede usar **A = 0** como solución general, ya que el hacerlo implicaría que el voltaje es cero todo el tiempo, una imposibilidad física si hay energía almacenada en el inductor o en el capacitor. Por lo tanto, para que la ecuación (4.3) sea una solución de la ecuación (4.2), el término entre paréntesis de la ecuación (4.4) debe ser cero,

$$s^2 + \frac{s}{RC} + \frac{1}{LC} = 0. \qquad (4.5)$$

La ecuación (4.5) se denomina **ecuación característica** de la ecuación diferencial porque las raíces de esta expresión cuadrática determinan el carácter matemático de **v(t)**.

Las dos raíces de la ecuación son:

$$s_1 = -\frac{1}{2RC} + \sqrt{\left(\frac{1}{2RC}\right)^2 - \frac{1}{LC}}$$

y

$$s_2 = -\frac{1}{2RC} - \sqrt{\left(\frac{1}{2RC}\right)^2 - \frac{1}{LC}}$$

Si se sustituye cualquiera de las raíces en la ecuación (4.3), la solución satisface la ecuación diferencial dada, es decir, la ecuación (4.2). Por lo tanto:

$$v = A_1e^{s_1t} \quad \text{o} \quad v = A_2e^{s_2t}$$

satisface la ecuación (4.2). Si llamamos $\mathbf{v_1}$ y $\mathbf{v_2}$ respectivamente a estas dos soluciones, podemos demostrar que su suma también es una solución[40]. Por consiguiente, la respuesta natural del circuito **RLC** en paralelo tiene la forma:

$$v = A_1 e^{s_1 t} + A_2 e^{s_2 t}.$$

Por lo que, el comportamiento de **v(t)** depende de las raíces $\mathbf{s_1}$ y $\mathbf{s_2}$ las cuales se determinan por los parámetros **R**, **L** y **C**. Así que

$$s_1 = -\alpha + \sqrt{\alpha^2 - \omega_o^2}$$

y

$$s_2 = -\alpha - \sqrt{\alpha^2 - \omega_o^2}$$

donde

$$\alpha = \frac{1}{2RC}$$

y

$$\omega_o = \frac{1}{\sqrt{LC}}$$

$\mathbf{s_1}$ y $\mathbf{s_2}$ se conocen como frecuencias complejas, α se denomina frecuencia neper y ω_o se llama frecuencia resonante[41]. La naturaleza de las raíces $\mathbf{s_1}$ y $\mathbf{s_2}$ depende de los valores de α y ω_o.

Existen tres resultados posibles; primero, si $\omega_o^2 < \alpha^2$, ambas raíces serán reales y distintas, se dice que la respuesta está **sobreamortiguada**. Si $\omega_o^2 > \alpha^2$, tanto $\mathbf{s_1}$ como $\mathbf{s_2}$ serán complejas y además complejas conjugadas. En esta situación se dice que la respuesta en voltaje está **subamortiguada**. Si $\omega_o^2 = \alpha^2$, $\mathbf{s_1}$ y $\mathbf{s_2}$ serán reales e iguales. En esta situación se dice que la respuesta en voltaje está **amortiguada críticamente**.

4.3.3 Respuesta en voltaje sobreamortiguado

Cuando las raíces de la ecuación característica son reales y distintas, se dice que la respuesta en voltaje está **sobreamortiguada**. La solución para el voltaje tiene la forma

$$v(t) = A_1 e^{s_1 t} + A_2 e^{s_2 t}. \qquad (4.6)$$

donde $\mathbf{s_1}$ y $\mathbf{s_2}$ son las raíces de la ecuación característica. Las constantes $\mathbf{A_1}$ y $\mathbf{A_2}$ se determinan por las condiciones iniciales, específicamente por los valores de $\mathbf{v(0^+)}$ y $\mathbf{dv(0^+)/dt}$. Estos se determinan a partir del voltaje inicial en el capacitor, **Vo**, y la corriente inicial en el inductor **Io**[42].

4.3.4 Respuesta en voltaje subamortiguado

[40] Para mayor información sobre los cálculos realizados, favor de referirse a cualquier libro de circuitos eléctricos.

[41] El exponente **e** no debe tener dimensiones, de manera que $\mathbf{s_1}$ y $\mathbf{s_2}$ (y por lo tanto α y ω_o) tienen las dimensiones del inverso del tiempo, o frecuencia. Todas estas frecuencias tienen la misma dimensión, generalmente radianes por segundo (rad/s).

42 V. [WH93]. William H.Hayt & Jack E. Kemmerly. *Análisis de Circuitos en Ingeniería.* Quinta Edición. Pp 200-211. McGraw Hill

Cuando $\omega_o^2 > \alpha^2$, las raíces de la ecuación característica son complejas y se dice que la respuesta es **subamortiguada**. Por cuestiones de conveniencia, al estudiar la respuesta subamortiguada se expresan las raíces s_1 y s_2 como

$$s_1 = -\alpha + \sqrt{-\left(\omega_o^2 - \alpha^2\right)}$$
$$= -\alpha + j\sqrt{\omega_o^2 - \alpha^2}$$
$$= -\alpha + j\omega_d$$

y

$$s_2 = -\alpha - j\omega_d ,$$

donde

$$\omega_d = \sqrt{\omega_o^2 - \alpha^2}$$

ω_d se llama **velocidad angular amortiguada**.
A partir de la ecuación (4.6), la respuesta en voltaje subamortiguada en un circuito **RLC** en paralelo se puede expresar como

$$v(t) = B_1 e^{-\alpha t} \cos \omega_d t + B_2 e^{-\alpha t} \operatorname{sen} \omega_d t \qquad (4.7)$$

al pasar la ecuación (4.6) a la ecuación (4.7) se usa la identidad de Euler:

$$e^{\pm j\theta} = \cos\theta \pm j \operatorname{sen}\theta$$

De esta manera

$$\begin{aligned} v(t) &= A_1 e^{\left(-\alpha + j\omega_d\right)t} + A_2 e^{-\left(\alpha + j\omega_d\right)t} \\ &= A_1 e^{-\alpha t} e^{j\omega_d t} + A_2 e^{-\alpha t} e^{-j\omega_d t} \\ &= e^{-\alpha t}\left(A_1 \cos\omega_d t + jA_1 \operatorname{sen}\omega_d t + A_2 \cos\omega_d t - jA_2 \operatorname{sen}\omega_d t\right) \\ &= e^{-\alpha t}\left[\left(A_1 + A_2\right)\cos\omega_d t + j\left(A_1 - A_2\right)\operatorname{sen}\omega_d t\right]. \end{aligned}$$

En este punto de la transición de la ecuación (4.6) a (4.7) se sustituyen las constantes arbitrarias **$A_1 + A_2$** y **$j(A_1 - A_2)$** con nuevas constantes arbitrarias **B_1** y **B_2** para obtener :

$$\begin{aligned} v &= e^{-\alpha t}\left(B_1 \cos\omega_d t + B_2 \operatorname{sen}\omega_d t\right) \\ &= B_1 e^{-\alpha t} \cos\omega_d t + B_2 e^{-\alpha t} \operatorname{sen}\omega_d t. \end{aligned}$$

las constantes **B_1** y **B_2** son reales, no complejas, porque el voltaje es una función real. En este caso subamortiguado, **A_1** y **A_2** son conju-gadas complejas, y por consiguiente **B_1** y **B_2** son reales. La razón para definir la respuesta subamortiguada en términos de los coeficientes **B_1** y **B_2**, es que conduce a una expresión más sencilla para el voltaje **v**.

Las dos ecuaciones simultáneas para determinar **B_1** y **B_2** en la respuesta subamortiguada son :

$$v\left(0^+\right) = V_o = B_1$$

y

$$\frac{dv\left(0^+\right)}{dt} = \frac{i_c\left(0^+\right)}{C} = -\alpha B_1 + \omega_d B_2$$

4.3.5 Respuesta en voltaje amortiguado críticamente

Cuando **$\omega_o^2 = \alpha^2$** o **$\omega_o = \alpha$** el circuito se dice que está amortiguado críticamente, esto es, la respuesta está a punto de oscilar. En este caso las dos raíces de la ecuación característica son reales e iguales; es decir,

$$s_1 = s_2 = -\alpha = -\frac{1}{2RC}$$

la solución del voltaje ya no tiene la forma de la ecuación (4.6), pues si **$s_1 = s_2 = -\alpha$** esto conduce a

$$v = \left(A_1 + A_2\right)e^{-\alpha t} = A_0 e^{-\alpha t}$$

donde $\mathbf{A_0}$ es una constante arbitraria. La ecuación anterior no puede satisfacer las dos condiciones iniciales (**Vo, Io**) con sólo una constante arbitraria, por lo que la solución toma un modelo distinto:

$$v(t) = D_1 te^{-\alpha t} + D_2 e^{-\alpha t} \qquad (4.8)$$

El procedimiento para hallar $\mathbf{D_1}$ y $\mathbf{D_2}$, es el mismo que se sigue para los casos subamortiguado y sobreamortiguado. Se emplean los valores iniciales del voltaje y la derivada del voltaje con respecto al tiempo para escribir las dos ecuaciones simultáneas:

$$v(0^+) = V_0 = D_2$$

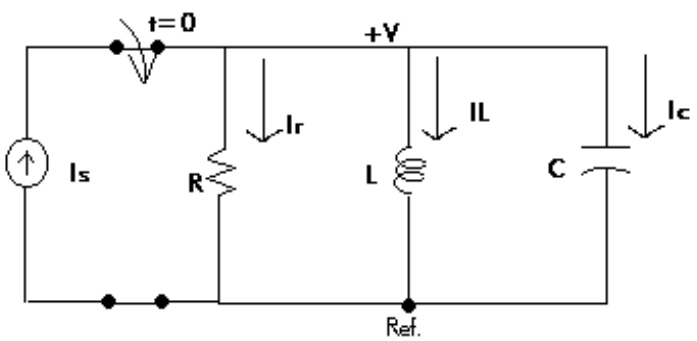
y

$$\frac{dv(0^+)}{dt} = \frac{i_c(0^+)}{C} = D_1 - \alpha D_2$$

4.3.6 Respuesta a un escalón

Aquí lo que interesa es el voltaje que aparece en las ramas paralelas como resultado de la aplicación repentina de una fuente de corriente continua o constante. Puede o no haber energía almacenada en el circuito en el momento de aplicar la corriente.

Ahora el circuito a tratar se define en términos de la siguiente figura:



Para desarrollar una estrategia general que sirva para encontrar la respuesta de un circuito de segundo orden a un escalón, se debe encontrar la corriente en la rama inductiva ($\mathbf{i_L}$)[43].

Como se quiere encontrar la respuesta a un escalón, se supone que la energía inicial almacenada en el circuito es cero. Esta suposición simplifica los cálculos y permite centrarse mejor en cómo hallar la respuesta a un escalón. Por la ley de corrientes de Kirchhoff tenemos :

[43] la corriente en el inductor es igual a la corriente en la fuente de corriente continua, **Is**.

$$i_L + i_R + i_C = I,$$

ó

$$i_L + \frac{v}{R} + C\frac{dv}{dt} = I.$$

como

$$v = L\frac{di_L}{dt},$$

se tiene

$$\frac{dv}{dt} = L\frac{d^2 i_L}{dt^2}.$$

substituyendo las ecuaciones y dividiendo por **LC**, obtenemos que :

$$\frac{d^2 i_L}{dt^2} + \frac{1}{RC}\frac{di_L}{dt} + \frac{i_L}{LC} = \frac{I}{LC}.$$

Al comparar la ecuación anterior con la ecuación (4.2), se observa que la presencia de un término distinto de cero en el lado derecho de la ecuación modifica el problema.

Por lo tanto, se debe despejar $\mathbf{i_L}$ en forma indirecta hallando primero el voltaje **v**, así

$$\frac{1}{L}\int_0^t v dt + \frac{v}{R} + C\frac{dv}{dt} = I.$$

derivando la ecuación con respecto a **t**, el lado derecho se anula porque **I** es una constante. Entonces

$$\frac{v}{L} + \frac{1}{R}\frac{dv}{dt} + C\frac{d^2 v}{dt^2} = 0,$$

ó

$$\frac{d^2 v}{dt^2} + \frac{1}{RC}\frac{dv}{dt} + \frac{v}{LC} = 0.$$

Como se demostró antes, la solución de **v** depende de las raíces de la ecuación característica. Por ello, las tres soluciones posibles son:

$$v = A_1 e^{s_1 t} + A_2 e^{s_2 t},$$
$$v = B_1 e^{-\alpha t}\cos\omega_d t + B_2 e^{-\alpha t}\,\text{sen}\,\omega_d t,$$

y

$$v = D_1 t e^{-\alpha t} + D_2 e^{-\alpha t}.$$

Al llegar a este punto es necesario hacer una advertencia. Puesto que hay una fuente en el circuito para **t > 0**, se debe tener en cuenta el valor de la fuente de corriente en **t = 0⁺** al calcular los coeficientes de las ecuaciones anteriores, por lo que:

$$i_L = I + A'_1\, e^{s_1 t} + A'_2\, e^{s_2 t},$$

$$i_L = I + B'_1\, e^{-\alpha t} \cos \omega_d t + B'_2\, e^{-\alpha t} \operatorname{sen} \omega_d t,$$

y

$$i_L = I + D'_1\, te^{-\alpha t} + D'_2\, e^{-\alpha t},$$

donde **A'₁, A'₂, B'₁, B'₂, D'₁** y **D'₂** son constantes arbitrarias.

Finalmente, la solución de la ecuación diferencial de segundo orden con una función constante forzada es igual a la **respuesta forzada más** la **respuesta natural**. Por ello, siempre se puede escribir la solución de la respuesta a un escalón en la forma:

$$i = I_f + \left\{ \begin{matrix} funcion\_de\_la\_misma\_forma \\ que\_la\_respuesta\_natural \end{matrix} \right\}$$

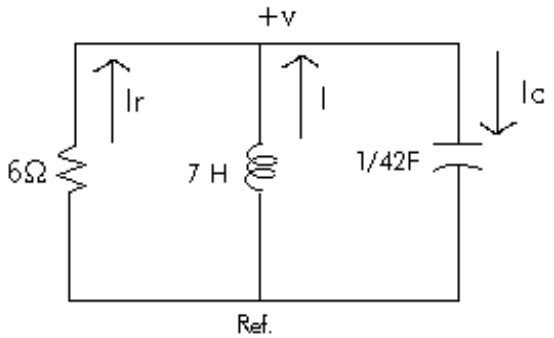
o

$$v = V_f + \left\{ \begin{matrix} funcion\_de\_la\_misma\_forma \\ que\_la\_respuesta\_natural \end{matrix} \right\},$$

donde $\mathbf{I_f}$ y $\mathbf{V_f}$ representan el valor final de la función respuesta. El valor final puede ser cero.

4.3.7 Ejemplos numéricos

Supongamos que se tiene el siguiente circuito sin fuentes, cuyas condiciones iniciales son **Io = 10, Vo = 0**.



al correr el programa **circuito_RLC_paralelo**[44] en **CLP®**, se tiene que :

CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991, 1992) All Rights Reserved

1 ?- **['PROGRAMAS/circuito_RLC_paralelo']**.
*** Yes

2 ?- **meta(6,7,1/42,10,0,Alfa,Omega,S1,S2,A1,A2).**

V(t) = 84'e'^-1t + -84'e'^-6t

Calcular en el tiempo? [s/n] s.
Tiempo : 10.

Ir = 0.000635601
IL = -0.000544801
IC = -9.08001e-05
V = 0.00381361

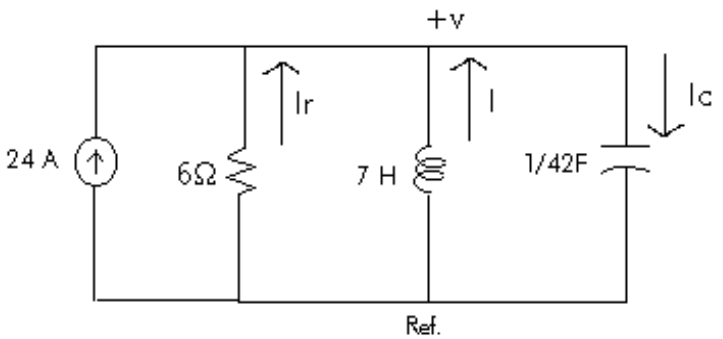
A2 = -84
A1 = 84
S2 = -6
S1 = -1
Omega = 2.44949
Alfa = 3.5
*** Retry?

[44] Ver Apendice B para el listado del programa.

Los valores se calculan fácilmente indicando que se trata de un **sistema sobreamortiguado**, dado que el valor de **Alfa < Omega**, la respuesta natural es:

$$v(t) = 84e^{-t} - 84e^{-6t}$$

el sistema pregunta si se desea calcular el voltaje con respecto al tiempo. Si la respuesta obtenida es '**n**' (No), la meta se detiene y el sistema está en espera de compilar otra meta; de lo contrario, se calcula el voltaje con respecto al tiempo (el cual es el mismo en todos los componentes dado que se trata de un sistema en paralelo), y se calculan las corrientes en cada uno de los componentes. Ahora, si el sistema es de la forma:



la nueva meta a correr será

3 ?- **meta(6,7,1/42,24,10,0,Alfa,Omega,S1,S2,A1,A2).**

IL(t) = 24 + -21.6'e'^-1t + 7.6'e'^-6t

Calcular en el tiempo? [s/n] s.
Tiempo : 10.

Ir = -0.00016344
IL = 0.000140092
IC = 2.33486e-05
V = -0.000980641

A2 = 7.6
A1 = -21.6
S2 = -6
S1 = -1
Omega = 2.44949
Alfa = 3.5
*** Retry?

Como se ve, la diferencia consiste en la inclusión de una fuente de corriente. Las constantes **A1** y **A2** son diferentes, dado que ahora toman en cuenta el valor de dicha fuente. La respuesta obtenida ahora, es la **respuesta a un escalón más la respuesta natural**.

Consideremos ahora el caso donde **Alfa > Omega** (voltaje subamortiguado), para obtener tal respuesta solo basta con modificar el valor de la resistencia; ahora **R=10.5, L=7** y **C=1/42**. Nuevamente se considera el caso donde la fuente está en corto circuito. Por lo tanto

4 ?- **meta(10.5,7,1/42,10,0,Alfa,Omega,S1,S2,A1,A2).**

V(t) = 0'e'^-2t cos1.41421t + 296.985'e'^-2t sen1.41421t

Calcular en el tiempo? [s/n] s.
Tiempo : 10.

Ir = -5.19883e-08
IL = -7.82897e-08
IC = 2.58917e-08
V = -5.45877e-07
S2 = 2
A2 = 296.985
A1 = 0
Alfa = 2
S1 = 1.41421
Omega = 2.44949
*** Yes

La meta es exitosa. Ahora la respuesta natural es :

$$v(t) = 296.985e^{-2t}\ \text{sen}\, 1.4142t$$

o bien

$$v(t) = 210\sqrt{2}e^{-2t}\ \text{sen}\, \sqrt{2}t$$

Si ahora tomamos en cuenta la fuente de corriente, la meta será:

5 ?- **meta(10.5,7,1/42,24,10, 0, Alfa,Omega,S1,S2,B1,B2).**

IL(t) = 24 + -14'e'^-2t cos1.41421t + -36.7696'e'^-2t sen1.41421t

Calcular en el tiempo? [s/n] s.
Tiempo : 5.

Ir = 0.000108301
IL = -1.747e-05
IC = -0.000114125
V = 0.00113717

S2 = 2
B2 = -36.7696
B1 = -14
Alfa = 2

S1 = 1.41421
Omega = 2.44949
*** Retry?

Nuevamente, los únicos valores que cambian son los valores para las constantes **B1** y **B2**. La respuesta obtenida se representa por la suma de la respuesta natural y forzada.

Finalmente, consideremos el caso en que el sistema comienza a oscilar, esto es **Alfa = Omega**, cuando se presenta un voltaje amortiguado críticamente. El valor de la resistencia ahora disminuye con respecto al ejemplo anterior, pero aumenta con respecto al primer ejemplo, esto es: **R=8.5732, L=7** y **C=1/42**.

6 ?- **meta(8.5732,7,1/42,10,0,Alfa,Omega,S1,S2,D1,D2).**

V(t) = 420t'e'^-2.44505t + 0'e'^-2.45394t

Calcular en el tiempo? [s/n] s.
Tiempo : 5.

Ir = 0.00117509
IL = 1.51313e-06
IC = -0.000539571
V = 0.0100743

D2 = 0
D1 = 420
S2 = -2.45394
S1 = -2.44505
Omega = 2.44949
Alfa = 2.44949
*** Retry?

Su respuesta natural será:

$$v(t) = 420te^{-2.44505t}$$

calculando con respecto a la fuente, tenemos que:

7 ?- **meta(8.5732,7,1/42,24,10,0,Alfa,Omega,S1,S2,D1,D2).**

IL(t) = 24 + -58.2929t'e'^-2.44505t + -14'e'^-2.45394t

Calcular en el tiempo? [s/n] s.
Tiempo : 5.
Ir=-0.000170927
IL=5.43007e-06
IC=7.88054e-05

V=-0.00146539
D2=-14
D1 = -58.2929
S2 = -2.45394
S1 = -2.44505
Omega= 2.44949
Alfa = 2.44949
*** Retry?

Conclusiones

Después de haber presentado cada uno de los lenguajes bajo el esquema de la programación lógica por restricciones, se concluye que:

Existen diferencias significativas en cada uno de los modelos, principalmente en el dominio de cálculo, el uso de un mecanismo diferido, el tipo de restricciones que manipulan, y la forma en que representan la salida de las restricciones resolubles. Para hacer mas claro este análisis, a continuación se muestra una tabla que refleja las similitudes y las diferencias entre cada uno de los lenguajes.

Lenguaje	**CHiP**	**Prolog III**	**CLP®**
Dominio de cálculo	• números finitos • booleanos • aritmética racional lineal	• aritmética racional lineal • booleanos • cadenas	• números reales
¿ es declarativo?	Si	Si	Si
¿ es flexible ?	Si	si	si
tipo de restricciones que manipula	• aritméticas • simbólicas • definidas por el usuario	• aritméticas • simbólicas	• aritméticas • simbólicas
técnicas usadas	Técnicas de consistencia	Técnicas de control	Técnicas de control
objetos manipulados	dominios variables	• árboles • tuplas	Números reales
como es su resolvedor de restricciones	general	máquina abstracta	resolvedor de restricciones no lineales y lineales
¿es incremental?	Si	no se pudo determinar	si
¿Manipula desigualdades ?	Si	si, pero solo entre árboles	sí
¿Cuenta con restricciones no lineales?	No	si	¡ por supuesto !
Como es su mecanismo diferido	usa un mecanismo de diferir/excitar, pero este tiene que estar definido en el programa (uso de	hace uso de un predicado “freeze”	el mecanismo esta integrado y se conoce como el mecanismo de

	demonios)		diferir/excitar
¿es aplicable?	Si	si	si
¿Cómo es su salida?	por medio de soluciones simbólicas	se toman como unidades de entrada/salida	en forma de relaciones (restricciones simbólicas) o bien, como resultados normales
tipo de metodología que usa	examina/genera	no se pudo determinar	examina/genera

Como se puede observar, **CLP®** es un lenguaje muy completo ya que cuenta con un mecanismo que integra restricciones no lineales mientras que **CHiP** y **Prolog III** tienen que hacer uso de algún tipo de predicado especial; otra ventaja de **CLP®** es el tipo de resultados que presenta puesto que no se limita a mostrar solo restricciones resolubles, sino también puede presentar una salida en relación a las restricciones que no fueron del todo resolubles.

El trabajar con árboles siempre ha sido un tanto tedioso (al menos en lo personal) por lo que trabajar con **CLP®** resultó ser mucho más fácil de lo que esperaba, y aúnque se podría considerar a su dominio de cálculo como una desventaja, dado que trabajar con números reales representa inexactitud en los resultados, **CLP®** resuelve este problema al hacer que la diferencia numérica sea mínima. Esto no implica que tanto **CHiP** como **Prolog III** sean “inferiores” con respecto a **CLP®**, cada quien tiene su particular punto de vista, pero en lo personal, recomiendo el uso del lenguaje **CLP®**.

APENDICE A : Programa Fuente Swaps

```
/*-------------------------------------------------------------------------------
Programa: Swaps (CLP®)
Autor: E. Tick
Fecha: Septiembre 15 1994
Traducción: Wendy Y. García Martínez
Notas:

1.- Preguntas:
     ?- top2( +Id, +Exchange, -Out).
        +Id = número de identificación del dato swap
        Exchange = 'no' si no se desea cambio alguno (en la respuesta final)
                   ' si' (Exchange_List) si se desean cambios (ver tabla 1)
        -Out = solución (lista de ganancias)

      ?- top1(+Id, -Out).
        Igual que la anterior, excepto que usa una tabla por defecto.
-------------------------------------------------------------------------------*/

% modo( ?, ^)
top1( Id, Out ) :-
        table( FX ),
        top2( Id, FX, Out ).

% modo( ?, ?, ^)
top2( Id, Exchange, Out ) :-
        info( Id,
                info( Market,
                      Libor,
                      const( Constraints ),
                      profits( Profs ),
                      vars( Vars ),
                      atoms( Atoms ),
                      graph( Graph ) ) ),
                market( Market, Mkl ),
                metamaker( Constraints ),          % cambia restricciones
                spawn( Graph, Libor, Mkt, Profits ),    % cambia préstamos
                filter( Exchange, Profits, NewProfits ),  % intercambia ganancias
                profit( NewProfits, Out0 ),        % combina ganancias
                profnamer( Profs, Out0 ),          % liga ganancias a variables
                !, unconstrain( Libor),                 % asegura que "Libor" no se ligue
                dump( Vars, Atoms, Out ). % variables de salida

% mode( ?, ?, ^)
% sin cambios para las ganancias
filter( no, Out, Out ).

% cambia todas las ganancias actuales a dólares $
```

```
filter( yes( Table ), In, Out ) :-
        filter1( In, Table, Out ).
% modo( ?, ?,^)
filter1( [], _, [] ).
Filter1( [In | Ins], Table, Out ) :-
        In = prof( Node1, Node2, Value, Currency ),
        lookup( Table, Currency, Rate ),
        Out = [ prof( Node1, Node2, Value+Rate, usd ) | Outs ],
        filter1( Ins, Table, Outs ).

% modo( ?, ?,^)
lookup( [], _, 1 ).
lookup( [ fx( Currency, Rate ) | _ ], Currency, Rate ) :- !.
lookup( [ _ | Rest ], Currency, Rate ) :-
        lookup( Rest, Currency, Rate ).

% modo( ?,^)
revolve( In, Out ) :-
        revolve1( In, In, Out ).

% modo( ?, ?,^)
revolve1( [], _, [] ).
revolve1( [ _ | Ins ], In, [ In, Outs ] ) :-
        In = [ A | As ],
        append( As, [ A ], NewIn ),
        revolve1( Ins, NewIn, Outs ).

unconstrain( Libor ) :-
        functor( Libor, libor, Arity ),
        unconstrain1( Arity, Libor ).

unconstrain1( 0, _ ).
unconstrain1( K, Libor ) :- K > 0,
        arg( K, Libor, X ),
        var ( X ),
        unconstrain1( K-1, Libor ).

% “profnamer/2” asegura los nombres de las variables definidas por el usuario
% para que las ganancias se unifiquen con sus correspondientes variables de
% programas. Este es el caso crítico para las restricciones del usuario.
%
% modo( ?, ?)
profnamer( [], _ ).
profnamer( [Profit | Profits ], Out ) :-
        profmem( Profit, Out ),
        profnamer( Profits, Out ).

% modo( ?, ?)
profmem( _, [] ) :- !.
profmem( P, [ P | _ ] ) :- !.
```

```
profmem( P, [ _ | Qm ] ) :- profmem( P, Qs ).



% modo( ?,^)
market( market( X ), Term ) :- !,
        functor( Term, market, 100 ),
        fill( 100, Term, X ).
market( Market, Market ).

fill( 0, _, _ ) :- !.
fill( K, T, E ) :- K > 0,
        arg( K, T, E ),
        fill( K-1, T, E ).

% modo( ? )
metamaker( [] ).
metamaker( [ C | Cs ] ) :-
        call ( C ),
        metamaker( Cs ).

% modo( ?, ?, ?,^)
spawn( [], _, _, [] ).
spawn( [ First | NewInfo ], Libor, Mkt, Profit ) :-
        First = info( Node1, Node2, Type, Cur, Princ, Start, End, Rate ),
        node( Type, Node1, Node2, Cur, Princ,
                Start, End, Rate, Libor, Mkt, [ Prof1, Prof2 ] ),
        Profit = [ Prof1, Prof2 | NewProfit ],
        spawn( NewInfo, Libor, Mkt, NewProfit ).

% modo( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,^)
node( one, Node1, Node2, Cur, Princ, Start, End, _, _, Mkt, Prof ) :-
        one( Princ, Start, End, Mkt, Cash ),
        pack( Node1, Node2, Cash, Cur, Prof ).

node( flat, Node1, Node2, Cur, Princ, Start, End, Rate, Libors, Mkt, Prof ) :-
        flat( Princ, Rate, Start, End, Libors, Mkt, Cash ),
        pack( Node1, Node2, Cash, Cur, Prof ).

node( amort, Node1, Node2, Cur, Princ, Strart, End, Rate, _, Mkt, Prof ) :-
        amort( Princ, Start, End, Rate, Mkt, Cash ),
        pack( Node1, Node2, Cash, Cur, Prof ).

pack( Node1, Node2, Cash, Cur, [ Prof1, Prof2 ] ) :-
        Prof1 = prof( Node1, Node2, (-Cash), Cur ),
        Prof2 = prof( Node2, Node1, Cash, Cur ).

% modo( ?,^)
profit( In, Out ) :-
```

```
        profit1( In, [], Out ).

% modo( ?, ?,^)
profit1( [], Out, Out ).
profit1( [ Profit | Profits ], Stack, Out ) :-
        prepper( Profit, Stack, NextStack ),
        profit1( Profits, NextStack, Out ).

% modo( ?, ?,^)
prepper( prof( N, _, Cash, Currency ), Stack, NextStack ) :-
        member( prof( N, _, Currency ), Stack, Ans, Rest ),
        ( Ans = no ->
          NextStack = [ prof( N, Cash, Currency ) | Stack ]
          ;
         Ans = prof( A, B, Currency ),
         NewCash = Cash + B,
         NextStack = [ prof( A, NewCash, Currency ) | Rest ] ).

% modo( ?, ?,^,^)
member( prof(A, _, E ), Stack, prof( A, B, C ), Out ) :-
        Stack = [ prof( A, B, C ) | Back ],
        C = E,
        Out = Back.
member( prof( A, _, E ), Stack, Rest, Out ) :-
        Stack = [ prof( B, C, D ) | Back ],
        not( A = B ),
        Out = [ prof( B, C, D ) | More ],
        member( prof( A, _, E ), Back, Rest, More ).

member( prof( A, _, E ), Stackl, Rest, Out ) :-
        Stack = [ prof( A, C, D ) | Back ],
        not( D = E ),
        Out = [ prof( A, C, D ) | More ],
        member( prof( A, _, E ), Back, Rest, More ).

member( _, [], no, [] ).


%=======================================
% Librerías para la variable del porcentaje de Mercado.
% ADVERTENCIA: ¡ No se puede solucionar para el mercado !

one( Principle, Start, End, Market, Value ) :-
        var ( Start ), !,
        one2( End-Start, End, Market, Principle, Value ).

one( Value, Start, End, _, Value ) :-
        Start >= End.

one( Principle, Start, End, Market, Value ) :-
```

```
        Start < End,
        one1( End-Start, Start + 1, Martket, Principle, Value ).

% porcentaje de mercado fraccionaria al FINALIZAR el período
one1( Time, Period, Market, In, Value ) :-
        Time > 0, Time <= 1,
        arg( Period, Market, MR ),
        Value = In / ( 1+ ( ( MR * Time ) / 100 ) ).



one1( Time, Period, Market, In, Value ) :-
        Time > 1,
        arg( Period, Market, MR ),
        Out = In / ( 1 + ( MR / 100 ) ),
        one1( Time - 1, Period + 1, Market, Out, Value ).

% porcentaje de mercado fraccionaria al INICIO del período
one2( Time, Period, Market, In, Value ) :-
        Time > 0, Time <= 1,
        arg( Period, Market, MR ),
        Value = In / ( 1 + ( ( MR * Time ) / 100 ) ).

one2( Time, Period, Market, In, Value ) :-
        Time > 1,
        arg( Period, Market, MR ),
        Out = In / ( 1 + MR / 100 ) ),
        one2( Time - 1, Period - 1, Market, Out, Value ).


%-------------------------------------------------------------------------------------
%
%       P               -P * LR                 -P * LR                 -P( 1 + LR )
%       |                 |                 |                   |
%       +-----------------+-----------------+-------------------+
%       |                 |                 |                   |
%      Start                                                   End
%      L( 1 )            L ( 2 )               L ( 3 )                 L ( 4 )
%     MR ( 1 )          MR( 2 )               MR ( 3 )          MR ( 4 )
%

flat( Principle, Rate, Start, End, Libors, Mkt, Value ) :-
        var( Start ), !,
        Value = Principle - Payments,
        flat2( End - Start, End, Principle, Rate, Libors, Mkt, Payments ).

flat( _, _, Start, End, _, _, 0 ) :- Start >= End.

flat( Principle, Rate, Start, End, Libors, Mkt, Value ) :-
        Start < End,
```

```
        Value = Principle - Payments,
        flat1( End - Start, Start +1, Principle, Rate, Libors, Mkt, Payments ).

flat1( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 0, Time <= 1,
        rate( Rate, Libors, Period, LR ),
        arg( Period, Mkt, MR ),
%       porcentaje aproximado del mercado final con período completo.
        Value = In * ( 1 + LR * Time ) / ( 1 + MR / 100 ).




flat1( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 1,
        rate( Rate, Libors, Period, LR ),
        arg( Period, Mkt, MR ),
        Out = In / ( 1 + MR / 100 ),
        Value = ( Out * LR ) + Rest,
        flat1 ( Time - 1, Period + 1, Out, Rate, Libors, Mkt, Rest ).

flat2( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 0, Time <= 1,
        rate( Rate, Libors, Period, LR ),
        arg( Period,  Mkt, MR ),
%       porcentaje aproximado del préstamo final con período final.
        Out = In / ( 1 + MR / 100 ),
%       escala final del porcentaje de préstamo con período fraccional.
        Value = Out * ( 1 + LR * Time ).

flat2( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 1,
        rate( Rate, Libors, Period, LR ),
        arg( Period, Mkt, MR ),
        Out = In / ( 1 + MR / 100 ),
        Value = ( Out * 1 + LR ) + Rest,
        flat3( Time - 1, Period - 1, Out, Rate, Libors, Mkt, Rest ).

flat3( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 0, Time <= 1,
        rate( Rate, Libors, Period, LR ),
        arg( Period, Mkt, MR ),
%       porcentaje aproximado de mercado final como período completo.
        Out = In / ( 1 + MR / 100 ),
%       escala final del porcentaje de préstamo como período fraccional.
        Value = Out * ( LR * Time ).

flat3( Time, Period, In, Rate, Libors, Mkt, Value ) :-
        Time > 1,
```

```
        rate( Rate, Libors, Period, LR ),
        arg( Period, Mkt, MR ),
        Out = In / ( 1 + MR / 100 ),
        Value = ( Out * LR ) + Rest,
        flat3( Time - 1, Period -1, Out, Rate, Libors, Mkt, Rest ).


%-------------------------------------------------------------------------------------------
% Préstamos amortizados solamente para porcentajes fijos, dando los
% períodos “Start” y “End”. Si no se conocen, entonces no se puede determinar
% si los pagos (Payments) se hacen en pocos o muchos periodos con un pago
% grande o pequeño.

amort( _, Start, End, _, _, 0 )  :- Start >= End.
amort( Principle, Start, End, fixed( Rate ), Mkt, Value ) :-
        Start < End,
        Value = Principle - Payments,
%       calcula primero el pago por período.
        mortgage( Start + 1, End, Principle, Rate, Payment ),
%       entonces calculamos el valor presente del flujo de dinero ...
        amort1( Start + 1, End, Payment, Mkt, Payments ).

amort1( End, End, In, Mkt, Value ) :- !,
        arg( End, Mkt, MR ),
        Value = In / ( 1 + MR / 100 ).

amort1( Period, End, In, Mkt, Value ) :-
        arg( Period, Mkt, MR ),
        Out = In / ( 1 + MR / 100 ),
        Value = Out + Rest,
        amort1( Period + 1, End, Out, Mkt, Rest ).

% al final, la balanza de pagos tiene que ser CERO ...
mortgage ( End, End, In, Rate, Payment ) :- !,
        0.0 = In * ( 1 + Rate / 100 ) - Payment.

mortgage( Start, End, In, Rate, Payment ) :-
        Out = In * ( 1 + Rate / 100 ) - Payment,
        mortgage( Start + 1, End, Out, Rate, Payment ).


%-------------------------------------------------------------------------------------------
% Funciones diversas

% modo( ?, ?,^)
append( [], L2, L2 ).
append( [ R | T ], L2, [ H | L3 ] ) :-
        append( T, L2, L3 ).

% modo( ?, ?, ?,^)
```

```
rate( fixed( Fix ), _, _, Fix / 100 ).
rate( float( Fix ), Libors, Period, Rate ) :-
        arg( Period, Libors, Float ),
        Rate = ( Float + Fix ) / 100.

% porcentaje de $ : x donde x es desconocido
table( yes [
        fx( yen, 0.01 ),
        fx( aud, 0.75 ),
        fx( fr, 0.30 ),
        fx( mrk, 0.50 )
          ] ) ).




info( 1, info(
        market( 3.5 ),
        libor( L1, L2, L3, L4, L5, L6, L7, L8, L9, L10 ),
        const( [ Pi3 = Pi4 ] ),
        profits( [ prof( 1, Pi1, usd ),
                prof( 2, Pi2, usd ),
                prof( 3, Pi3, usd ),
                prof( 4, Pi4, usd ),
                prof( 5, Pi5, usd ),
                prof( 6, Pi6, usd ),
                ] ),
        vars( [ Rate1, Rate2, Pi1, Pi2, Pi3, Pi4, Pi5, Pi6 ] ),
        atoms( [ rate1, rate2, pi1, pi2, pi3, pi4, pi5, pi6 ] ),
        graph( [
        info( 6, 4, flat, usd, 48, 0, 10, float( 0.0) ),
        info( 3, 6, flat, usd, 48, 0, 10, float( -0.2 ) ),
        info( 4, 6, flat, usd, 48, 0, 10, fixed( Rate2 ) ),
        info( 6, 5, flat, usd, 75, 0, 10, fixed( Rate1 ) ),
        info( 1, 6, one, usd, 27, 0, 0, _ ),
        info( 6, 1, one, aud, 38, 0, 0, _ ),
        info( 1, 6, one, aud, 70, 0, 10, _ ),
        info( 6, 1, one, usd, 37, 0, 10, _ ),
        info( 6, 3, one, aud, 68, 0, 0, _ ),
        info( 3, 6, one, aud, 130, 0, 10, _ ),
        info( 2, 5, one, aud, 106, 0, 0, _ ),
        info( 5, 2, one, aud, 200, 0, 10, _ ),
        info( 5, 6, one, aud, 106, 0, 0, _ ),
        info( 6, 5, one, aud, 200, 0, 10, _ ) ]
) ) ).
```

APENDICE B: Programa Fuente Circuitos Eléctricos

```
/*-------------------------------------------------------------------------------------------------

Programa : Circuito RLC en paralelo a cd. ( CLP® )
Autor : Wendy Y. García Martínez.
Fecha : Agosto 10, 1997
Notas:

1.- Preguntas:

      ?- meta(R,L,C,Io,Vo,Alfa,Omega,S1,S2,A1,A2).

          Valores de entrada :
            R = Valor de la Resistencia
            L = Valor de la Inductancia
            C = Valor de la Capacitancia
            Io, Vo = Condiciones iniciales para la corriente y el voltaje

          Valores de salida :
           Alfa = frecuencia Neper
            Omega = frecuencia resonante
            S1, S2 = raíces de la ecuación característica
            A1, A2 = constantes de la ecuación característica

      ?- meta(R,L,C,Is,Io,Vo,Alfa,Omega,S1,S2,A1,A2).
            igual que la anterior sólo que en este caso se introduce una
            fuente de corriente “Is”.

-------------------------------------------------------------------------------------------------*/

?- dynamic(meta,12).    % declara al predicado meta como dinámico

% respuesta natural
meta(R,L,C,Io,Vo,Alfa,Omega,S1,S2,A1,A2) :-
        DVt = (Io + (Vo /R) ) / C,
        calcula(R,L,C,Alfa,Omega,S1,S2),
         circuitoA(Vo,DVt,Alfa,Omega,S1,S2,A1,A2),
%       pregunta si se desea calcular con respecto al tiempo...
        pregunta(R,L,C,Alfa,Omega,A1,A2,S1,S2).

% respuesta forzada
meta(R,L,C,Is,Io,Vo,Alfa,Omega,S1,S2,AA1,AA2) :-
          DIt = Vo/L,
          calcula(R,L,C,Alfa,Omega,S1,S2),
          circuitoB(Is,Io,DIt,Alfa,Omega,S1,S2,AA1,AA2),
%       pregunta si se desea calcular con respecto al tiempo ...
          pregunta(R,L,C,Alfa,Omega,AA1,AA2,S1,S2).
```

```
calcula(R,L,C,Alfa,Omega,S1,S2) :-
        Alfa = 1/(2*R*C),
        Omega = 1/pow((L*C),0.5),
%       verifica que tipo de raíces se tienen ...
        comprueba(Alfa,Omega,S1,S2).

% en el caso de que las raíces sean reales ...
comprueba(Alfa,Omega,S1,S2):-
      Alfa >= Omega,
      Raiz = pow(((Alfa*Alfa)-(Omega*Omega)),0.5),
      S1 = - Alfa + Raiz,
      S2 = - Alfa - Raiz.

% en el caso de que las raíces sean complejas ...
comprueba(Alfa,Omega,S1,S2):-
      Alfa < Omega,
% S1 es la única raíz ( ω0 ), S2 sólo se ocupa como formalidad ...
      S1 = pow(((Omega*Omega)-(Alfa*Alfa)),0.5),
      S2 = Alfa.

% circuito SIN fuente ...
circuitoA(Vo,DVt,Alfa,Omega,S1,S2,A1,A2) :-
       abs(Alfa - Omega) < 0.001 ->
                critico(Vo,DVt,Alfa,S1,S2,A1,A2)
                ;
                 sinfuente(Vo,DVt,Alfa,Omega,S1,S2,A1,A2).

% caso con voltaje en amortiguamiento crítico sin fuente...
critico(Vo,DVt,Alfa,S1,S2,A1,A2) :-
      Ec1 = A2 - Vo,
      Ec2 = A1 - (Alfa*A2) - DVt,
      eval(Ec1)=0,
      eval(Ec2)=0,
      printf("\n V(t) = % t 'e' ^ % t + % 'e' ^ % t ",[A1,S1,A2,S2]).

% circuito CON fuente ...
circuitoB(Is,Io,DIt,Alfa,Omega,S1,S2,B1,B2) :-
      abs(Alfa - Omega) < 0.001 ->
                critico2(Is,Io,DIt,Alfa,S1,S2,B1,B2)
                ;
                confuente(Is,Io,DIt,Alfa,Omega,S1,S2,B1,B2).


% caso con voltaje en amortiguamiento crítico con fuente ...
critico2(Is,Io,DIt,Alfa,S1,S2,B1,B2) :-
               Ec1 = Is + B2 - Io,
               Ec2 = quote(Is + B1 - (Alfa*B2) - DIt),
               eval(Ec1)=0,
```

```
            eval(Ec2)=0,
             printf("\nI I(t) = % + % t 'e' ^ % t + %'e' ^ % t ",[Is,B1,S1,B2,S2]).


% caso con voltaje sobreamortiguado sin fuente ...
sinfuente(Vo,DVt,Alfa,Omega,S1,S2,A1,A2) :-
            Alfa > Omega,
            Ec1 = A1 + A2 - Vo,
           Ec2 = S1*A1 + S2*A2 - DVt,
            Ecf = quote(((-1)*Ec1) + ((-1)*Ec2)),
            eval(Ecf)=0,
           eval(Ec1)=0,
            printf("\n V(t) = %'e' ^ % t + % 'e' ^ % t ",[A1,S1,A2,S2]).


% caso con voltaje subamortiguado sin fuente ...
sinfuente(Vo,DVt,Alfa,Omega,S1,S2,A1,A2) :-
        Alfa < Omega,
        Ec1 = A1 - Vo,
        Ec2 = (-S2*A1) + (S1*A2) - DVt,
        eval(Ec1)=0,
        eval(Ec2)=0,
        printf("\nV(t) = % 'e' ^ - % t cos % t + % 'e' ^ - % t sen % t " , [A1,S2,S1,A2,S2,S1]).


% caso con voltaje sobreamortiguado con fuente ...
confuente(Is,Io,DIt,Alfa,Omega,S1,S2,B1,B2) :-
        Alfa > Omega,
        Ec1 = Is + B1 + B2 - Io,
        Ec2 = Is + S1*B1 + S2*B2 - DIt,
        Ecf = quote(((-1)*Ec1) + ((-1)*Ec2)),
        eval(Ecf)=0,
        eval(Ec1)=0,
        printf("\n Il(t) = % + %'e' ^ % t + % 'e' ^ % t  ",[Is,B1,S1,B2,S2]).


% caso con voltaje subamortiguado con fuente ...
confuente(Is,Io,DIt,Alfa,Omega,S1,S2,B1,B2) :-
        Alfa < Omega,
        Ec1 = Is + B1 - Io,
        Ec2 = Is + (-S2*B1) + (S1*B2) - DIt,
        eval(Ec1)=0,
        eval(Ec2)=0,
        printf("\nI I( t ) = % + % 'e' ^ - % t  cos % t + % 'e' ^ - % t sen % t " ,
        [Is,B1,S2,S1,B2,S2,S1]).

% pregunta si se desea calcular con respecto al tiempo ...
pregunta(R,L,C,Alfa,Omega,A1,A2,S1,S2) :-
         printf("\n\n\n Calcular en el tiempo? [s/n] ",[]),
         read(Resp),
```

```
        ( Resp == n -> !
        ;
         printf("\n Tiempo : ",[]),
         read(Tt),
         verifica(Tt,R,L,C,Alfa,Omega,A1,A2,S1,S2)
        ).
verifica(Tt,R,L,C,Alfa,Omega,A1,A2,S1,S2) :-
      abs(Alfa - Omega) < 0.001 ->
                corrienteA(Tt,R,L,C,Alfa,A1,A2)
                ;
                 corrientes(Tt,R,L,C,Alfa,Omega,A1,A2,S1,S2).

% calcula la corriente en el caso crítico ...
corrienteA(Tt,R,L,C,Alfa,A1,A2) :-
            e(-Alfa*Tt, E),
            Ir = (E *(A1*Tt + A2)) / R,
            Il = (( E / ((-Alfa*Alfa))*(-Alfa*Tt - 1)) - (A2/Alfa)*E) / L,
            Ic = C *((-Alfa*A1*E*Tt) + (A1*E) - (Alfa*A2*E)),
            V = E *(A1*Tt + A2),
           dump([Ir,Il,Ic,V]).

% calcula la corriente en el caso sobreamortiguado ...
corrientes(Tt,R,L,C,Alfa,Omega,A1,A2,S1,S2) :-
            Alfa > Omega,
            e(S1*Tt, E1), e(S2*Tt, E2),
            Ir = (A1*E1 + A2*E2) / R,
            Il = ((A1/S1)*E1 + (A2/S2)*E2) / L,
            Ic = C * ((S1*A1)*E1 + (S2*A2)*E2),
            V = A1*E1 + A2*E2,
            dump([Ir,Il,Ic,V]).

% calcula la corriente en el caso subamortiguado ...
corrientes(Tt,R,L,C,Alfa,Omega,B1,B2,S1,S2) :-
      Alfa < Omega,
      e(-S2*Tt, E3),
      X = E3 / ((-Alfa*Alfa) + (S1*S1)),
      X1 = cos(S1*Tt),
      X2 = cos((S1*Tt)+90),
      Ir = (E3 * ((B1*X1) + (B2*X2))) / R,
      Il = ((B1*X*((-Alfa*X1) + (S1*X2))) + (B2*X*((-Alfa*X2) - (S1*X1))))/L,
      Ic=C*(((-Alfa*E3)*((B1*X1)+(B2*X2)))+(E3*((-B1*S1*X2)+(B2*S1*X1)))),
      V = E3 * (B1*X1 + B2*X2),
      dump([Ir,Il,Ic,V]).

% calcula el valor exponencial ...
e(0,1).
e(Term,Rest) :- Rest= pow(2.718281,Term).

?- style_check(all_reset).
```

REFERENCIAS

[AB81] A. Borning. **The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory**, ACM Transactions on Programming Languages and Systems, 3(4), 252-387, October 1981.

[AB92] Aggoun A - Beldiceanu N. **Extending CHIP in Order to Solve Complex Scheduling and Placement Problems**. Tech.Rep., COSYTEC, 1992.

[AB93] A. Aggoun, N, Beldiceanu. **Overview of the CHIP Compilex System**, in : *Constraint Logic Programming: Selected Research.* MIT Press, 421 - 435, 1993.

[AC82] A. Colmerauer, **Prolog and Infinite Trees in Logic Programming**, K.L. Clark and S-A. Tarnlund (Eds), Academic Press, New York, 231-251, 1982.

[AC84] A. Colmerauer, **Equations and Inequations on Finite and Infinite Trees**, Proc. 2nd Int. Conf. On Fifth Generation Computer Systems, Tokyo, 85-99, 1984.

[AC87] A. Colmerauer. **Opening the Prolog III Universe**. BYTE magazine, 12(9). August 1987.

[AC87] A. Colmerauer. **Opening the Prolog III- Universe**. Byte, pags. 177-182. Agosto 1987.

[AC88] Alain Colmerauer. **Final Specifications for Prolog III**. ESPRIT P1219(1106). February 1988.

[AC90] A. Colmerauer. **An Introduction to Prolog III**. Communications of the ACM, 33 No. 7 pags. 52-68, 1990.

*[AC-93] A. Colmerauer, *invited talk at Workshop on the* **Principles & Practice of Constraint Programming**, Newport, RI, April 1993.

[AC93] A. Colmerauer, **Naive Solving of Non-linear Constraints**, Constraint Logic Programming: Selected Research, F. Benhamou and A. Colmerauer (Eds) MIT Press, 89-112, 1993.

[AM77] A.K Mackworth. **Consistency in Networks of Relations**. AI Journal, 8 (1), 99 - 118, 1977.

[AS88] A. Aiba, K. Sakai, Y. Sato. **Constraint Logic Programming Languages**. Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88), ICOT, Tokyo, 263 - 276, december 1988.

[AS92] A. Shapiro. **Multinational Financial Management**. Allyn and Bacon. 4th edition. 1992

[BM93] F. Benhamou, J. L. Massat, **Boolean Pseudo-equations in Constraint Logic Programming**, Proc. 10th International Conference on Logic Programming, 517-531, 1993.

[BS87] W. Büttner, H. Simonis, **Embedding Boolean Expressions into Logic Programming**, Journal of Symbolic Computation, 4, 191-205, 1987.

[CW89] C. Wantisky. **CLP(Z) Constraint Logic Programming with Regular Sets**. 6 ICLP, Proc. 6th International Conference on Logic Programming. MIT Press, Cambridge, MA, 1989.

[DHD83] D.H.D, Warren. **An Abstract PROLOG Instruction Set**. Technical note 309. AI Center. SRI International. Menlo Park. October 1983.

[DR93] A. Davier, G. Rossi. **Embedding Extensional Finite Set in CLP**. Proc. International Logic Programming Symposium, 540-556, 1993.

*[DS94] D. H. Scott. **A Graphical Editor for Designing Financial Swaps**. Bachelor's thesis. The University of Oregon. December 1994.

[DSH] M. Dincbas, H. Simonis, P. Van Hentenryck. **Solving Large Combinational Problems in Logic Programming**. Journal of Logic Programming.

[DSH87] M. Dincbas, H.Simonis, P. Van Hentenryck. **Extending Equation Solving and Constraint Handling in Logic Programming**. Colloquium on Resolution of Equations in Algebraic Structures (CREAS), Texas. May 1978.

[DSH88] M. Dincbas, H. Simonis, P. Van Hentenryck. **Solving the Car Sequencing Problem in Constraint Logic Programming**. European Conference on Artificial Intelligence (ECAI-88), Munich, W. Germany. August 1988.

[Fox83] M.S. Fox. **Constraint- Directed Search: A Case Study of Job-Shop Scheduling**. Technical Report CMU-CS-83-161, Carnegie-Mellon University, December 1983.

[GD63] G.B Dantzing. **Linear Programming and Extensions**. Princeton University Press. Princenton, New Jersey, 1963.

[GJ79] M.R. Garey and D.S. Johnson. **Computers and Intractability**. W.H* Freeman and Company, New York, 1979.

[GL89] G. Levi. **Declarative Modelling of the Operational Behavior of Logic Languages**. Theretical Computer Science, 289-318, 1989.

[GO8] J. Goguen, J. Meseguer. **Software for the Rewrite Rule Machine**. Proceedings of the International Conference on Fifth Generation Systems (ICOT). 628-637. 1988.

[HA91] H. Ait-Kaci, **Warren's Abstract Machine: A Tutorial Reconstruction**, MIT Press, 1991.

[HB89] H. Beringer, F. Porcher. **A relevant Scheme for Prolog Extensions: CLP (Conceptual Theory)**. Proceedings of the 6th International Conference on Logic Programming. MITP. 131-148. Jun 1989.

[HE80] R.M Haralick, G.L. Elliot**. Increasing Tree Search Efficiency for Constraints Satisfaction Problems**. Artificial Intelligence, 14: 263 - 313, 1980.

[HE87] N. Heintze, Spiro Michaylov, Peter Stuckey. **CLP® and Some Electrical Engineering Problems**. Logic Programmings: Proceedings of the 4th International Conference. Melbourne, Victoria, Australia. 675 - 703. May 1987.

[HE89]H. Gaifman, E. Shapiro, M. Maher. **Reactive Behavior Semantics for Concurrent Constraint Logic Programs**. Proc. North American Conference on Logic Programming. 553- 569. October 1989.

[HG85] H. Gallaire. **Logic Programming: Further Developments**. Simposium de IEEE sobre Programación Lógica, pags. 88-99. Julio 1985.

[HMS87] N.C Heintze, S. Michaylov, P.J. Stuckey. **CLP® and Some Electrical Engineering Problems**. Fourth International Conference on Logic Programming, Melbourne, Australia, May 1987.

[HO91] D. S. Homiak. **A Constraint Logic Programming System for Solving Partial Differential Equations with Applications in Options Valuation**. Master's project, DePaul University. 1991.

[JB92] J. Burg, **Parallel Execution Models and Algorithms for Constraint Logic Programming over a Real-number Domain**, Ph.D. thesis, Dept. Of Computer Science, University of Central Florida, 1992.

[JC90] J. Cohen, **Constraint Logic Programming Languages**, CACM, 33, 62-68, July 1990.

*[JC92] J. Jaffar, C. Yap, S. Michaylov, P. Stuckey. **The CLP® Language and Systems**. ACM Transactions on Programming Languages ans Systems. 14(3), 339 - 395. July 1992.

[JH89] J. Hull. *Options,* **Futures and Other Derivative Securities**. Prentice Hall. 1989.

[JJ91] Joxan Jaffar, R. Yap, S. Michaylov. **A Methodology for Managing Hard Constraints in CLP Systems**. Proceedings of the ACM, Sumposium on Programming Language Design and Implementation. Toronto Canada. 306 - 316. June 1991.

*[JJ-92] J. Jaffar, S. Michaylov, N. C. Heintze, R. Yap. **The CLP® Programmers Manual - ver. 1.2** . Tech. Rep., School of Computer Science, CMU, Pittsburg, PA, 1992.

[JK76] J. De Kleer. **Local Methods of Localizing faults in Electronic Circuits**. Technical Report AIM - 394, Artificial Intelligence Laboratory, MIT, Cambridge, USA, 1976.

*[JL86] J. Jaffar, J.L. Lassez. **Constraint Logic Programming**. Technical Report 86/73, Departament of Computer Science. Monash University. 1986.

*[JL87] J. Jaffar, J. L. Lassez. **Constraint Logic Programming**. In Proceedings 14th Anual ACM Symposium on Principles of Programming Languages, pags., 111-119, Munich, 1987.

[JM85] J. Macfarlane, D.R. Ross and J. Showers. **The Interest Rate Swap Market: Yield Mathematics**. Terminology and Conventions. Salomon Brothers Inc. Junio 1985

*[JM87] J. Jaffar, S. Michaylov. **Metodology and Implementation of a CLP system**. In 4 ICLP Proc. 4th International Conference on Logic Programming. MIT Press, Cambridge, MA, 1987.

*[JM92] J. Jaffar, S. Michaylov, P.J. Stuckey, R. Yap. **The CLP® Language and System**. ACM Transactions on Programming Languages and Systems. 14 No. 3, pp 339-395, 1992.

[JMS93] J. Jaffar, M. J. Maher, P.J. Stuckey, R. Yap. **Projecting CLP® Constraints**. New Generation Computing, 11, 449-469, 1993.

[JMY91] J. Jaffar, S. Michaylov, R. Yap. **A Methodology for Managaing Hard Constraints in CLP systems**. 4th Proceedings ACM. Conference on Programming language Design and Implementation, pp 306-316, Toronto, Ontario, 1991.

[JP88] Judea Pearl, Rina Dechter. **Network- Based Heuristics for Constraint- Satisfaction Problems**. Artificial Intelligence Magazine. 34(1), 1-38. January 1988.

[JR65] J.A Robinson. **A Machine-oriented Logic Based on the Resolution Principle**. Journal of ACM, 12 (1). 23 - 41, January 1965.

[KO67] D. A. Koher. **Projections of Polyhedral Sets**. Operations Research Center, University of California at Berkeley. August 1967.

[LH89] J. Lassez, L. Huynh, K. McAloon. **Simplification and Elimination of Redundant Linear Arithmetic Constraints**. In Proceedings North American Conference on Logic Programming, Cleeveland, pp 35 - 51. MIT Press, October 1989.

[LK79] L.G. Khachian. **A polynomial algorithm in linear programming**. Soviet Math. Dokl. 20(1), 191-194, 1979.

[LN84] L. Naish. **Mu-Prolog 3.1 db Reference Manual**. Melbourne University Edition, 1984.

[LS90] L.S. Sterling. **The Practice of Prolog**. The MIT Press, Cambridge, MA, 1990.

[MD86] M.Dincbas. **Constraint Logic Programming and Deductive Databases**. In Proceedings of France- Japan Artificial Intelligence and Computer Science Symposium. 1 - 27 ICOT, Tokyo, Japan October 1986.

[MM87] M.J Maher. **Logic Semantics for a Class of Committed-Choice Programs**. Proc. 4th International Conference on Logic Programming. 858-876, 1987.

*[MM93] M.J. Maher. **A Logic Programming View of CLP**. Proc. 10th International Conference on Logic Programming, 737-753, 1993. Versión completa: IBM Research Report, T.J. Watson Research Center.

[MN86] D. Miller, G. Nadathur. **Higher-Order Logic Programming**. Proc. 3rd International Conference on Logic Programming, 448-462, 1986.

[MN89] U. Martin, T. Nipkow. **Boolean Unification - the Story So Far**. Journal of Simbolic Computation, 7, 275-293, 1989.

[MO91] K. Marriott, M. Odersky. **Constraint - based Query Optimization for Spatial Databases**. Proc. 10th ACM Symp. On Principles of Database Systems, 181-191, 1991.

[NK84] N. Karmarkar. **A New Polynomial-Time Algorithm for Lineal Programming**. Combina - orica, 4 (4), 373 - 395, 1984.

[OB93] Older, W.J - Benhamou, F. **Applying Interval Arithmetic to Real Integer and Boolean Constraints**. Logic Programming. The ALP Newsletter 6, No. 2 pp 13-14, Mayo 1993.

*[PIII-93] **Prolog III ver 1.4. Manual de Referencia**. PrologIA. 1993

[PJS91] P.J. Stuckey. **Constructive Negation for Constraint Logic Programming**. Proc. Logic in Computer Science Conference. 328-339, 1991.

[PVH87] P. Van Hentenryck. **A Theoretical Framework for Consistency Techniques in Logic Programming**. IJCAI-87, 2 - 8, Milan Italy, August 1987.

[PVH-87] P. Van Hentenryck. **Consistency Techiniques in Logic Programming**. PhD thesis, University of Namur (Belgium), July 1987.

*[PVH-88] P. Van Hentenryck, M. Dincbas, H. Simonis, A. Aggoun, T, Graf, F. Berthier. **The Constraint Logic Programming Language: CHiP**. Proceedings of the International Conference on Fifth Generation Computer Systems. ICOT, 1988.

[PVHD86] P. Van Hentenryck, M. Dincbas. **Domains in Logic Programming**. AAAI-86, 759 - 765, Philadelphia, USA, August 1986.

*[PVHD88] P. Van Hentenryck, M. Dincbas, H. Simonis, A. Aggoun. **The Constraint Logic Programming Language CHIP**, Proceedings of the 2nd. International Conference of Fifth Generation Computer Systems, 249-264, 1988.

[RD74] R. J. Duffin. **On Fourier's analysis of linear inequality systems**. Math. Program. Stud. 1, 71-95, 1974.

[RD84] R. Davis. **Diagnostic Reasining Based on Structure and Behavior**. Artificial Intelligence, 24() : 347- 410, 1984.

[RK79] R. A. Kowalski. **Algorith = Logic + Control**. Communications of the ACM, 22, pp 424-431, 1979.

[RLL86] R. Layard - Liesching. **Swap Fever. Euromoney**. Pags. 108 - 113. Enero 1986. Suplemento.

[RY91] Roland Yap. **Restriction Site Mapping in CLP®.** Proceedings 8th International Conference on Logic Programming. MIT Press. June 1991. 521- 534.

[RY94] Roland Yap. **Contributions to CLP®.** PhD. Thesis. Department of Computer Science. Monash University. January 1994.

[SB92] R. Skuppin, T. Buckle. **CLP and Spacecraft Attitude Control**. Proc. JICSLP Workshop on Constraint Logic Programming. 45-54, 1992.

[SC86] Spiro Michaylov, C. Heintze, J. Jaffar, R. Yap... **The CLP® Programmers manual - Version 1**. Department of Computer Science, Monash University. June 1986.

[SD-87] H. Simonis, M. Dincbas. **Using an extended prolog for digital circuit design**. IEEE International Workshop on AI Applications to CAD Systems for Electronics. Munich, W. Germany, 165-188, October 1987.

[SD87] H. Simonis, M. Dincbas. **Using Logic Programming for Fault Diagnosis in Digital Circuits**. German Workshop on Artificial Intelligence (GWAI-87), Geseke, W. Germany, 139-148, September 1987.

[SG85] S.I Gass. **Linear Programming**. McGraw-Hill. New York, 1985.

[SND88] H. Simonis, H.N. Nguyen, M. Dincbas. **Verification of digital circuits using CHIP**. Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification. Glasgow, Scotland. July 1988.

[SS80] G. Steele, G.J. Sussman. **CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions**. Artificial Intelligence 14(1), 1-39, 1980.

[TH93] T. Hickey. **Functional Constraints in CLP Languages**. Constraint Logic Programming. Selected Research, F. Benhamou and A. Colmerauer (Eds.) MIT Press, 355-361 , 1993.

[TO88] Tobias II, Joseph C. **Knowledge in the Harmony Intelligent Tutoring sustems**. Departament of Computer Science, University of California at Los Angeles. 1988.

[UM74] U. Montanari. **Networks of Constraints: fundamental Propierties and Applications to Picture Processing.** Informal Science, 7 (2), 95- 132, 1974.

*[WH93] William H. Hayt & Jack E. Kemmerly. **Análisis de Circuitos en Ingeniería**. 5ª edición. McGraw Hill

[YJM91] R. Yap, J. Jaffar, S. Michaylov. **A Methodology for Managing Hard Constraints in CLP systems**. Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation, 306-316, 1991.

[YU91] Edward K. Yu. **MODIC: A program for model-based diagnosis that uses constraint logic programming**. Department of Computar Science, University of South Carolina (Columbia), 1991.

Nota: Los artículos precedidos por un * son los artículos conseguidos, las referencias restantes se conocieron de manera indirecta.