# Análisis de Sistemas de Programación Lógica por Restricciones

Anatoli Koulinitch*
Wendy Yaneth García Martínez**

## Resumen

*La* Programación Lógica por Restricciones (CLP) *es la unión natural de dos paradigmas declarativos: la* solución de restricciones *y la* programación lógica*. Esta combinación ayuda a hacer programas* expresivos *y* flexibles*, y en algunos casos más eficientes que otros programas, dado que reducen dramáticamente el tiempo de ejecución mientras logran una eficiencia similar a los lenguajes procedimentales.*

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 por una estructura algebraica (el dominio de cálculo, las funciones y las relaciones sobre ese dominio). Las funciones especiales y los predicados simbólicos se interpretan sobre un dominio seleccionado fijo, formando los* símbolos interpretados*; las relaciones sobre el dominio de cálculo se llaman* restricciones*, éstas se formulan involucrando las funciones especiales y los predicados simbólicos.*

*Algunos de los lenguajes más significativos del esquema CLP son* CHiP*,* Prolog III *y* CLP®*, en los cuales se pueden ver diferencias 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.*

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 sólo restricciones resolubles, sino también puede presentar una salida en relación a las restricciones que no fueron del todo satisfactorias.*

*Trabajar con* CLP® *resultó ser mucho más fácil de lo que esperaba, aunque se podría considerar a su dominio de cálculo como una desventaja. Debido a 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®*.*

## Abstract

Constrained Logical Programming *(CLP) is the natural union of two declarative paradigms:* Constrained Solutions *and* Logical Programming*. This combination helps make programs expressive and flexible. In some cases, CLP is more efficient than other programs given that CLP drastically reduces the command execution time while achieving a similar efficiency to other conventional languages.*

*CLP defines a family of languages of* Logical Programming*, where each language is an instance obtained through specifying a calculus structure. The language is characterized by an algebraic structure. Such an algebraic structure encompasses calculus and the various functions that act upon it. The special functions and symbolic rules are interpreted over a fixed chosen range forming* interpreted symbols*. The relations over calculus are called* constraints*. These are formed by combining special functions and symbolic rules.*

*Some of CLP´s more significant languages are CHiP, Prolog III, and CLP®. In these, a difference can be seen in the calculus used, in the use of a differed engine, in the type of constraints they act upon and in the form that they illustrate the output of the solvable constraints.*

*CLP® is a complete language now that it includes a mechanism that integrates non linear constraints. In contrast, CHiP and Prolog III rely on a special form of discerning device. Another advantage of CLP® is the form that it presents its results. Since it is not limited to showing only solvable constraints but can also produce an output with respect to the constraints that are partially unsatisfactory.*

*CLP® is very user friendly, although its use of calculus could be considered a hindrance. Since working with real numbers often creates imprecise outcomes, CLP® makes the numeric difference minimal thereby resolving the problem. This, however, does not imply that either CHiP or Prolog III are "inferior" to CLP®.*

## Introducción

Durante las últimas décadas, el paradigma de la *Programación Lógica* ha alcanzado su grado de madurez. 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 se tiene que hacer?) y el perfil del mecanismo de cálculo se usa

* Profesor-investigador de tiempo completo de la U.T.M
** Profesora-investigadora de tiempo completo de la U.T.M

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. La regla de cálculo en la *Programación Lógica* es el origen de un problema y el procedimiento de *búsqueda a profundidad* resulta de la aplicación del paradigma *genera/examina*, dando lugar a una serie de problemas en la búsqueda a grandes espacios.

La *Programación Lógica por Restricciones* CLP (Constraint Logic Programming) intenta cubrir las fallas de la Programación Lógica integrando a un lenguaje como *Prolog* mecanismos que solucionen restricciones. Curiosamente estas fallas se pueden disipar usando *restricciones*.

Por lo tanto, CLP es la unión natural de dos paradigmas declarativos: la *solución de restricciones* y la *programación lógica*. Esta combinación ayuda a hacer programas CLP expresivos flexibles y en algunos casos más eficientes que otros programas, dado que reducen dramáticamente el tiempo de ejecución mientras logran una eficiencia similar a los lenguajes procedimentales.

## 1. Esquema de los Lenguajes CLP

La *Programación Lógica por Restricciones* (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, las funciones y las relaciones sobre ese dominio). Las funciones especiales y los predicados simbólicos se interpretan sobre un dominio seleccionado fijo, formando los *símbolos interpretados*; las relaciones sobre el dominio de cálculo se llaman *restricciones*, éstas se formulan involucrando las funciones especiales y los predicados simbólicos.

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.

Puede decirse, entonces, que la *Programación Lógica por Restricciones* implica la incorporación de restricciones con métodos que *solucionan* éstas en un lenguaje basado en la *lógica*, donde su esquema resultante define las clases de lenguajes CLP(X) obtenidos al instanciar el parámetro X. Este parámetro se interpreta como la tupla (å,D,L,T), donde å 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].

### Semántica Algebraica

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.
   $\forall d \in X$, $d = \bigcap C_i$, 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 $C_i$;
   $\forall C_i \widetilde{C} = \cup C_i$ por razones de la negación requerimos que la teoría T, la cual corresponde a $X_T$, sea una satisfacción completa (complete satisfaction), esto es:
   $T \not\vdash \bot C$ siempre que no $T \not\vdash \$C$;
   decimos que la teoría T corresponde a la estructura X si:
   $X \not\vdash T$, X modela T y $X \not\vdash \widetilde{\exists} C$ implica que $T \not\vdash \$C \forall C$

Si bien muchas estructuras algebraicas satisfacen el criterio de la solución compacta, éstas se eligen como base para la implementación de una instancia en particular de un lenguaje CLP(X). Cualquier dominio restrictivo nuevo 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 implementación.
- debe existir un resolvedor de restricciones. El resolvedor tiene que ser completo[1].
- tiene que existir una amplia área de aplicaciones.

## Semánticas Lógicas

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

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

como la fórmula lógica $\forall \bar{x}, \bar{y}\, p(\bar{x}) \vee \neg b_1 \vee \ldots \vee \neg b_n$, donde $\bar{x} \cup \bar{y}$ 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:

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

entonces la fórmula asociada a p es:

$$\forall \bar{x}\, p(\bar{x}) \leftrightarrow$$
$$\exists \bar{y}_1 B_1$$
$$\vee \exists \bar{y}_2 B_2$$
$$\ldots$$
$$\vee \exists \bar{y}_n B_n$$

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

1. El resolvedor es completo si es capaz de decidir la satisfactibilidad de cualquier conjunto de restricciones del dominio de cálculo.

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. Esto se hace dada la existencia de derivaciones infinitas que dan lugar a un conjunto insoluble de restricciones respuesta.

## Semánticas del Punto Fijo

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

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

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

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

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

## Semántica Operacional

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 que 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*. 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, f, f>.

Las transiciones en el sistema de transición son:
<A È a, C, S> ®$_r$ <A È B, C, S È (a=h)>

si a se selecciona por una regla de cálculo, a es un átomo, h ¬ 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 È a, C, S> ®$_r$ falla

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

<A È c, C, S> ®$_c$ <A, C, S È c>

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

<A, C, S> ®$_i$ <A, C', S'> si (C', S')=inferir(C, S)
<A, C, S> ®$_s$ <A, C, S> si consistente(C)
<A, C, S> ®$_s$ falla si consistente(C).

La transición ®$_r$ surge de la resolución de la transición ®$_c$, la cual inserta restricciones dentro del resolvedor de restricciones; la transición ®$_s$ examina si las restricciones activas son consistentes y, finalmente, la transición ®$_i$ infiere más restricciones activas (y quizás modifica las restricciones pasivas) de la colección actual de restricciones. Se escribe ® para referir una transición de tipo arbitrario.

El predicado *consistente(C)* expresa un examen de consistencia para C. Se define por: *consistente(C)* si DÆ$\exists$C, esto es, un examen de consistencia completa. Sin embargo, los sistemas pueden emplear un examen conservativo pero incompleto (o parcial) si DÆ$\exists$C, entonces *consistente(C)* se mantiene, pero algunas veces *consistente(C)* se mantiene a través de DÆØ$\exists$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 DÆ(C Ù S) « (C' Ù S'), tal que la información sea perdida o supuesta por la inferencia.

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 È S, f). En *CLP®* las restricciones no lineales son pasivas e inferir simplemente pasa de 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.

Finalmente, 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 son:

Sea ®$_{ris}$ = ®$_r$ ®$_i$ ®$_s$ y ®$_{cis}$ = ®$_c$ ®$_i$ ®$_s$. Se dice que un sistema CLP es de *verificación rápida* si su semántica operacional se puede describir por ®$_{ris}$ y ®$_{cis}$.

Un sistema CLP es *progresivo* si para cada estado con una colección vacía de átomos, cada derivación a de ese estado falla, conteniendo una transición ®$_r$ o conteniendo una transición ®$_c$.

Un sistema CLP es *ideal* si éste es de verificación rápida, progresivo; la inferencia se define por inferir(C,S)=(C È S,f) y *consistente(C)* se mantiene si DÆ$\exists$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, aun con las limitaciones de la consistencia y la inferencia, éste encuentra la inconsistencia tan pronto como sea posible.

Una *derivación* es una secuencia de transiciones $<A_1, C_1, S_1> \rightarrow \ldots \rightarrow <A_i, C_i, S_i>$. Un estado que no puede derivarse más se llama *estado final*. Una derivación es *exitosa* si ésta es finita y el estado final tiene la forma <f, C, S>.

Una derivación es *falla* si ésta es finita y el estado final falla. Una derivación es *imparcial* si ésta es falla o para cada i y cada a $\in A_i$, 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 imparcial en un sistema CLP ideal es falla.

## 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*.

### 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.

### Incrementabilidad

De acuerdo con el 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 éste 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. Dicha 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 f, 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, sea D el símbolo que denota a $F(f, o_1,\ldots,o_k)$. Sea "*A*" un algoritmo que 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.

Considérese 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 D no es mejor que el costo del método directo de usar "*A*" sobre $o_1,\ldots,o_k$, entonces:

$AV[costo(D, o_k)] \geq AV[costoA(o_1,\ldots,o_k)]$.

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(f, $o_1$,...,$o_{k-1}$)+costo(D, $o_k$)]£AV[ $costo_A$($o_1$,..., $o_k$)].

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

AV[costo(f, $o_1$, ..., $o_{k-1}$)+costo($_D$, $o_k$)]=AV[$costo_A$($o_1$, ..., $o_k$)]+costo_extra($o_1$, ..., $o_k$)

donde el término adicional costo extra($o_1$,..., $o_k$) 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.

### Satisfactibilidad

Es crucial que el algoritmo que determina la *satisfactibilidad* de una nueva restricción almacenada sea *incremental*. Sea una secuencia de restricciones $c_1$, ..., $c_k$ de aproximadamente igual tamaño N. Una aplicación sencilla de este algoritmo de tiempo lineal es decidir que $c_1$, entonces $c_1$ Ù $c_2$, y finalmente $c_1$ Ù ... Ù $c_k$ podrían incurrir en un costo proporcional a $Nk^2$ 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 *satisfactibilidad* 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.

### 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 G1 de G de las restricciones vinculadas por C Ù 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 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.

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 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.

### Proyección

El problema consiste en obtener una representación útil de la proyección de las restricciones C. Más formalmente: dadas las variables meta $\bar{x}$ y las restricciones C($\bar{x}$,$\bar{y}$) que implica variables de $\bar{x}$ y $\bar{y}$, expresamos que $\exists\bar{y}$ C($\bar{x}$, $\bar{y}$) 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 variables, 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.

### Retroceso

El método 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 éstas se modifican por el resolvedor de restricciones y el realmacenamiento de estas restricciones actualiza el retroceso.

En *Prolog* las restricciones son ecuaciones entre términos, representadas 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. Este identifica al conjunto de variables, las cuales se han ligado desde el último punto de selección. Para actualizar el retroceso, las variables simplemente se "resetean" para volverse no ligadas. Así, en *Prolog* la única información *arrastrada* es aquella en la cual las variables se han vuelto a vincular y el no *arrastrarlas* simplemente no liga estas variables.

En general, para CLP 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 éste 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*.

En resumen, el retroceso en CLP es substancialmente más complejo que en *Prolog*. Algunos de los conceptos para 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 de vida del mismo punto de selección y, finalmente, la reconstrucción de las referencias cruzadas, más que del *arrastre*.

## 3. Máquina de Inferencia

### Diferir/Excitar metas y restricciones

El problema a resolver es determinar cuándo una meta diferida será excitada o cuándo 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, respecto al resolvedor de restricciones, 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 restricciones 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*, entonces se incurriría en un costo proporcional al número de entradas, no obstante el hecho de que se afecten las restricciones no protegidas.

### Sistemas Excitados

Un *grado excitado* representa un subconjunto de p restricciones y un *sistema excitado* consiste en 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.

En suma, 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 cuándo 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.

### 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 primera 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(.) Ù C' apuntando a L:

(a) Se borran todos los nodos de ocurrencia apuntando a C.
(b) Se inserta la nueva restricción diferida C''=p(.) Ù C' Ù x=5 con un apuntador a C.
(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 éste solamente requiere 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.

Hasta aquí se han mostrado las características más sobresalientes de los sistemas CLP. A continuación se detallan sólo algunos de los sistemas que trabajan bajo este paradigma de programación.

## 4. Sistemas Existentes

Al surgir la Programación Lógica por Restricciones, diversos lenguajes basados en ésta 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.

### 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 las 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.

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 tie-

nen 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 que 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.

Existen muchos métodos para solucionar un conjunto dado de restricciones. Un método es la *propagación local*. 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.

Aún cuando es posible aumentar cualquier lenguaje con características restrictivas, un hecho importante es cómo el lenguaje interactúa con dichas características; en algunos lenguajes con restricciones esta interacción es tosca, pues se requiere que el usuario proporcione una gran cantidad de información acerca de cómo están siendo recolectadas y solucionadas aquellas. En los lenguajes CLP esta interacción es clara.

### Parámetros Clave

Con respecto a sus parámetros clave, los lenguajes CLP se caracterizan 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 en 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 aquéllos 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.

### 4.1. CHiP (Constraint Handling in Prolog)

CHiP se basa en el concepto del uso *activo* de restricciones [HG85][PVHD86] [MD86]. Difiere de los len-

guajes 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.

CHiP trabaja con tres dominios de cálculo:

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

Para cada uno de estos dominios, CHiP usa algoritmos especializados. Mientras que las técnicas de verificación de consistencia se usan para los dominios finitos, las 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.

### Dominios Finitos

La característica básica de CHiP para solucionar problemas combinatorios 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 /.

### 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[2], son restricciones bien formadas de CHiP.

### Restricciones simbólicas

CHiP no se restringe a las restricciones aritméticas, 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 éstos.
- 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 importante, por lo tanto, 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.

2. *X* diferente de *Y*.

### 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 combinatorios 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.

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 proporcionan 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.

### Términos Booleanos

Dos formas de representar a los términos booleanos son la *representación 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.

### 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.

### Términos racionales

Los términos racionales son términos lineales sobre números y variables racionales (las cuales toman sus valores de los números racionales). Las relaciones aritméticas usuales se pueden definir sobre términos racionales.

Dados dos términos racionales X y Y, las 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.

### 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 éstos son satisfactibles o no. Este procedimiento es una adaptación 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 implementación, 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 directamente 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 SÈ C. CHiP transformará la solución de S dentro de una solución S È C, si ésta existe. Tal 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*. Contrariamente 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:

a) *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.

b) *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 que *optimicen* una función de evaluación lineal.

En resumen, muchos problemas a partir de campos como la *Investigación 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 por qué CHiP usa algoritmos basados en Simplex y no los algoritmos de Khachian[LK79] y Karmarkar[NK84]. Estos algoritmos 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 (éste 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 normalmente 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.

### Propagación Local

En la *propagación local* las restricciones se usan para encontrar los valores de las variables no instanciadas a partir de variables instanciadas. 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*.

### Propagación Condicional

La *propagación condicional* es una técnica de propagación manejada por la satisfactibilidad o insatisfactibilidad de las restricciones. Declarativamente, ésta es una simple construcción *if_then_else*.

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

El procedimiento provee un mecanismo de manejo de demonios eficiente, ésta 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 procedimiento 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 ésta 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 puede 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.

### 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.

### Aplicaciones en planeación y prevención

La *Investigación de Operaciones* (I.O.) es una fuente interminable de problemas de investigación muy 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 telecomunicaciones. 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.

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 componentes).

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 del 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 y
- análisis y planeación financiera.

### 4.2 Prolog III

*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 sólo nodo. Algunos tienen un nombre particular, mientras que otros solamente pueden diseñarse usando una operación, que 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 y
- el signo especial "< >".

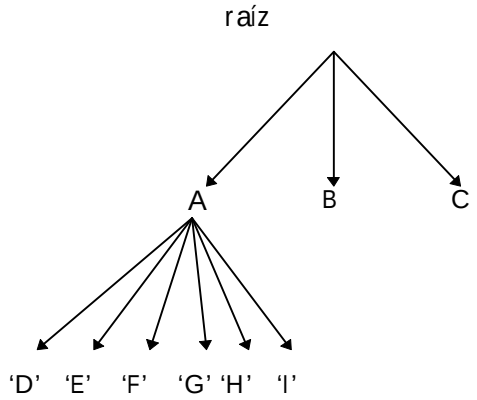
Se debe tener cuidado de no confundir los árboles con los términos. Los *árboles* son elementos en el dominio de *Prolog III*, los *términos* son construcciones sintácticas.

Los árboles cuya etiqueta inicial sea un identificador se llaman *árboles de hechos* y aquéllos cuya etiqueta inicial es el signo "<>" son llamados *tuplas*, cuyos elementos son llamados *cadenas*[3]. Al combinar la generalidad de las listas y la eficiencia de los vectores, las *tuplas* representan una herramienta flexible y poderosa para definir estructuras de datos.

3. En Prolog III las cadenas se representan por comas invertidas (') y las tuplas se representan por los paréntesis cuadrados (< Hasting,1066,1 >)

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 y nombre será *hoja*. *Prolog III* no distingue entre una hoja y las etiquetas que acarrea. Consecuentemente, los identificadores, caracteres, valores booleanos y numéricos son considerados como árboles de tipo especial.

Un *árbol* en *Prolog III* se representa de la siguiente manera:




## 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 con 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:

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

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.

## 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): $(a_1, a_2) \rightarrow a_1 \mid a_2$
(4) implica: $(a_1, a_2) \rightarrow a_1 \Rightarrow a_2$
(5) equivalente: $(a_1, a_2) \rightarrow a_1 \Leftrightarrow a_2$

## 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[4]: $(a_1, a_2) \rightarrow a_1 * a_2$
(6) división[5]: $(a_1, a_2) \rightarrow a_1 / a_2$

## 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: $(a_1, a_2,..., a_n) \rightarrow a_1(a_2,..., a_n)$ solamente se define si $n \geq 2$ y $a_1$ es una hoja.

4. Para aligerar la notación, Prolog III permite $a_1a_2$ en lugar de $a_1*a_2$.
5. La división solamente se define si $a_2$ no es cero.

(2) tupla construcción: $(a_1,..., a_n) \rightarrow \rangle a_1,..., a_n \langle$ se define si la tupla ($a_1$, ..., $a_n$) donde n ³ 1. n se dice que es la *longitud* de la tupla construida. La siguiente igualdad se deriva de la definición misma de las tuplas: $\rangle a_1,..., a_n \langle$ =<> $(a_1,..., a_n)$

(3) constructor general de árboles: $(a_1, a_2) \rightarrow a_1[a_2]$ esta operación sólo se define si $a_1$ es una *hoja* y $a_2$ es una *tupla*. Por definición se tiene la siguiente igualdad: $a_1[\rangle b_1,..., b_m \langle] = a_1(b_1,..., b_m)$
El constructor general de árboles se usa para representar cualquier árbol usando únicamente su etiqueta inicial y la tupla formada por sus hijos inmediatos[6].

(4) concatenación de tuplas: $(a_1, a_2) \rightarrow a_1 \cdot a_2$ solamente se define si a1 y a2 son ambas tuplas. Por definición se tiene la igualdad:
$\rangle b_1,..., b_m \langle \cdot \rangle c_1,..., c_k \langle = \rangle b_1,... b_m, c_1,.., c_k \langle$

## Variables y Términos

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 carácter 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.

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.

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 sólo 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.

## Asignación

Cuando se define una *asignación* del conjunto de variables, simplemente se selecciona un valor para cada variable pertinente. Si el conjunto es: $V = \{x_1, x_2,..., x_n\}$, entonces una *asignación* será el conjunto de pares ($x_i$, $a_i$) escritos como xi¬ ai; por lo tanto:

$A = \{x_1 \leftarrow a_1, x_2 \leftarrow a_2,..., x_n \leftarrow a_n\}$ 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* definido por su extensión (elemento por elemento).

## 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:

6. 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 de Prolog III en la cual los espacios se prohiben.

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 recursivamente 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 y
- (si n ¹ 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, éstas tienen que ser:

- del mismo tipo (dos identificadores, dos números, dos caracteres) e
- 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.

### 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 operación de construcción, sino simplemente otra manera sintáctica de representar en ciertos árboles: el *punto par* y *nulo* (nil).

### 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 ³ suponiendo que todas son del tipo <, £ o del tipo >, ³. Se puede escribir entonces {1< x < y £ 3} en lugar de {1< x, x < y, y £ 3}.

### Restricciones Numéricas

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 esa variable para que represente un valor numérico.

La principal restricción aplicada a las restricciones numéricas es que sólo las ecuaciones lineales y las 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.

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 requiera 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 determina por las características 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 aun indispensables cuando se solucionan muchos problemas, aunque también se pueden considerar al tener efectos dañinos, especialmente en la exactitud de los cálculos que implican.

### Restricciones Booleanas

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

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) Þ (g),
  (f) Û (g) son expresiones booleanas
  y cualquier expresión booleana en *Prolog III* es un *término*.

### 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. 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, identificándose después como variables *conocidas*.

### Restricciones diferidas

*Prolog III* difiere ciertas restricciones que 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).

### Técnicas de Control

En cada etapa de ejecución de un programa, el intérprete de *Prolog III* realiza dos pasos: primero selecciona una meta a ejecutar de una secuencia de metas y entonces selecciona la regla que se usará para ejecutar a ésta. 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.

## 4.3 CLP®

Esencialmente Â 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.

### Restricciones en CLP®

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 $t_1$, ...., $t_n$ son términos, entonces $f(t_1,...,t_n)$ es un término. Si $t_1$ y $t_2$ son términos aritméticos, entonces $t_1 = t_2$, $t_1 < t_2$ y $t_1 £ t_2$ son todas *restricciones* aritméticas. Sin embargo, si 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*.

## Programas en CLP®

Un *átomo* es de la forma $p(t_1, t_2, ..., t_n)$, donde p es un predicado distinto a los símbolos =, < y £; y $t_1$, ..., $t_n$ son términos. Una *regla* es de la forma: $A_o$ :- $a_1$, $a_2$, ..., $a_k$, donde cada $a_i$, 1 £ i £ 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:?- $a_1$, $a_2$, ..., $a_k$, donde cada $a_i$ 1 £ i £ k es una restricción primitiva o un átomo. Además, cada restricción primitiva de la meta se clasifica al ser solucionada o diferida. Una subcolección de átomos y restricciones en una meta es algunas veces llamada una *submeta* de la meta.

## Hacia una metodología de programación

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

## 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 requieren *restricciones globales* para describir la forma en que estas partes interactúan (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 más 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 independientes de algún sistema.

## 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 más sofisticadas en forma de ligaduras o unificadores con variables dadas. Sin embargo, éstos siguen un modelo explícito de salida. En contraste, *CLP®* provee salidas en forma de *restricciones respuesta* y ellas están inherentemente implícitas.

La salida implícita aumenta el poder expresivo de un lenguaje de programación en muchas formas. Esta per-

mite que respuestas complejas se representen de una manera simple y compacta describiendo objetos, los cuales no tienen una representación sintáctica y explícita.

Las *restricciones respuesta* encarecen no sólo la salida, sino también la metodología de programación. En particular, podemos pensar en una restricción respuesta como una evaluación parcial de un programa, esto es, una instancia más específica del programa original.

## Problemas de búsqueda combinatoria

La metodología *examina/genera* de CLP se aplica típicamente a problemas que satisfacen restricciones y 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 la 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...

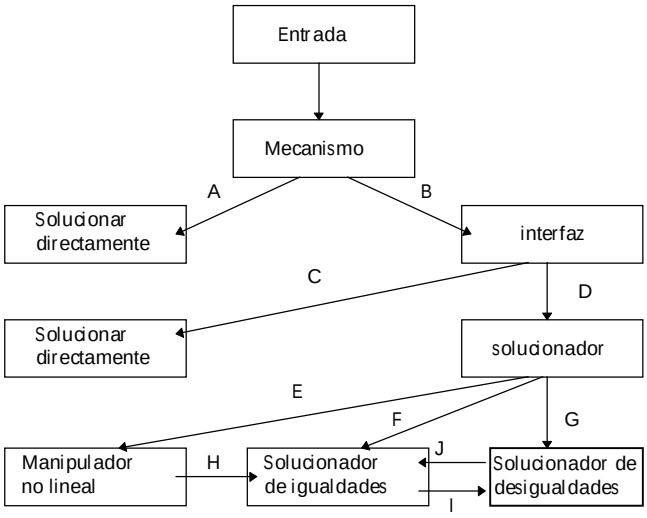
donde los generadores se usan para instanciar las variables, cuyo rango se encuentra sobre un dominio en particular. Esta es generalmente una mejor estrategia de búsqueda que la de *genera/examina*, ya que las restricciones se examinan antes de que se generen, evitando así la generación cuando ya se conoce que las restricciones son inconsistentes.

## 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 complejas 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, éste envía ecuaciones al solucionador de ecuaciones lineal, mientras que éstas 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: 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®

### El mecanismo

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

### La Interfaz

Este módulo es llamado desde el mecanismo, siempre que una restricción contenga un término aritmético. La *interfaz* primero simplifica 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 solucionada 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, éste transforma la restricción simplificada dentro de una de las siguientes formas:

- una desigualdad de la forma Variable > 0;
- una desigualdad de la forma Variable ³ 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 y
- una ecuación de la forma Variable=c * Variable * Variable donde c es un número real.

### El resolvedor de ecuaciones

Este módulo se invoca de 3 formas: vía la interfaz con una ecuación lineal en la forma descrita anteriormente; vía el resolvedor de desigualdades con una igualdad implícita y 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 central de datos en el resolvedor de ecuaciones es una tabla que 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, es decir, 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.

### 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 lineal se invoque únicamente *después* de que la ecuación actual haya sido tratada por completo, porque el módulo no lineal puede por sí mismo invocar al resolvedor de ecuaciones; sin embargo, surgen 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, más bien éstos 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. De esta forma, 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.

### 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, entonces el resolvedor de desigualdades comunica esas desigualdades al resolvedor de igualdades.

### Manipulador de restricciones no lineales

El *manipulador no lineal* tiene dos funciones principales. Primero, almacena ecuaciones no lineales de la forma $X_0=c*X_1*X^2$, 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 cuál de éstas será excitada, esto es, cuál de ellas se volverá lineal como resultado de la ecuación aterrizada.

Específicamente, una ecuación no lineal $X_0=c*X^1*X^2$ se vuelve lineal solamente como resultado de una ecuación de la forma $X^1$=número o $X^2$=número. Cada ecuación no lineal excitada se envía al resolvedor de desigualdades.

## 5. CLP® como una aproximación al esquema CLP

La implementación de *CLP®* es una aproximación al modelo operacional del esquema *CLP* por cuatro 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 éstas sean lo suficientemente simples[7].

¿Por qué programar en CLP®?

Dada la fuerza y disponibilidad de *CLP®* que permite 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.

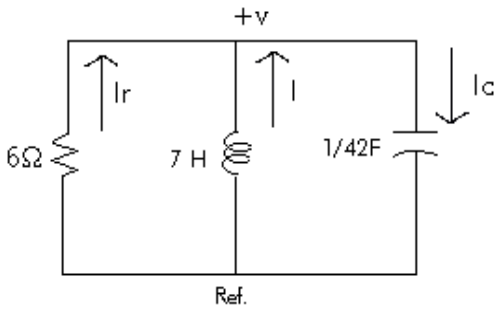
A continuación se presenta cómo *CLP®* resolvería un circuito eléctrico RLC en paralelo a corriente directa.

## 6. 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[8].

Supongamos que se tiene el siguiente circuito sin fuentes, cuyas condiciones iniciales son Io=10, Vo = 0.




al correr el programa circuito_RLC_paralelo[9] 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 = 0.000635601I L= -0.000544801  IC=-9.08001e-05
V = 0.00381361  A2 = -84          A1 = 84
S2 = -6         S1 = -1           Omeg= 2.44949
Alfa = 3.5
*** Retry?
```

Los valores se calculan fácilmente indicando que se trata de un *sistema sobreamortiguado*; dado que el valor de *Alfa < Omega*, la respuesta natural es:

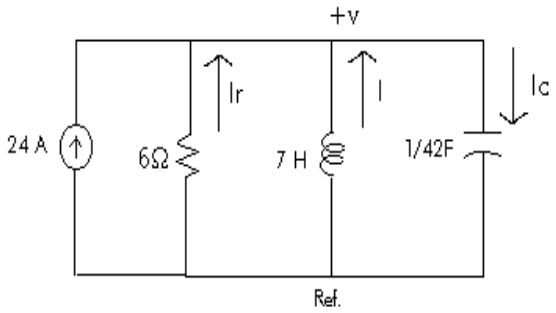
7. La definición exacta de suficientemente simple depende del poder del algoritmo resolvedor de restricciones.

8. V [GW98]. Wendy Y. García Martínez, *Análisis de Sistemas de Programación Lógica por Restricciones,* Cap. 4.

9. V [GW98]. Wendy Y. García Martínez, op. cit., Apéndice B.

$$v(t) = 84e^{-t} - 84e^{-6t}$$

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, pues 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 sólo basta 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 :

$$v(t) = 296.985e^{-2t}\,sen1.4142t$$

o bien

$$v(t) = 210\sqrt{2}e^{-2t}\,sen\sqrt{2}t$$

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 ejem-

plo 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:

$$v(t) = 420\, te^{-2.44505\, t}$$

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?
```

## Bibliografía


[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.

[AC87] A. Colmerauer. *Opening the Prolog III Universe.* BYTE magazine, 12(9). August 1987. pags. 177-182

[AM77] A.K Mackworth. *Consistency in Networks of Relations.* AI Journal, 8 (1), 99 - 118, 1977.

[GD63] G.B Dantzing. *Linear Programming and Extensions.* Princeton University Press. Princenton, New Jersey, 1963.

[GW98] García Mtz. Wendy Y. *Análisis de Sistemas de Programación Lógica por Restricciones.* Tesis Profesional. Universidad Tecnológica de la Mixteca. México, 1998.

[HE80] R.M Haralick, G.L. Elliot. *Increasing Tree Search Efficiency for Constraints Satisfaction Problems.* Artificial Intelligence, 14: 263 - 313, 1980.

[HG85] H. Gallaire. *Logic Programming: Further Developments.* Simposium de IEEE sobre Programación Lógica, pp. 88-99. Julio 1985.

[JK76] J. De Kleer. *Local Methods of Localizing faults in Electronic Circuits.* Technical Report AIM - 394, Artificial Intelligence Laboratory, MIT, Cambridge, USA, 1976.

[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.

[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.

[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.

[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.

[NK84] N. Karmarkar. *A New Polynomial-Time Algorithm for Lineal Programming.* Combinatoria, 4 (4), 373-395, 1984.

[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.

[RD84] R. DAVIS. *Diagnostic Reasining Based on Structure and Behavior.* Artificial Intelligence, 347- 410, 1984.

[RK79] R. A. KOWALSKI. *Algorith = Logic + Control.* Communications of the ACM, 22, pp 424-431, 1979.

[SG85] S.I GASS. *Linear Programming.* McGraw-Hill. New York, 1985.

[SS80] G. STEELE, G.J. SUSSMAN. *CONSTRAINTS - A Language for Expressing Almost Hierarchical Descriptions.* Artificial Intelligence 14(1), 1-39, 1980.

[UM74] U. MONTANARI. *Networks of Constraints: fundamental Propierties and Applications to Picture Processing.* Informal Science, 7 (2), 95- 132, 1974.