ARTÍCULO

Programación Declarativa con Restricciones

Antonio J. Fernández

Dpto. Lenguajes y Ciencias de la Computación,
E.T.S.I.I., Universidad de Málaga,
29071 Málaga, Spain,
afdez@lcc.uma.es

Resumen

Recientemente, esta revista ha publicado una excelente monografía, [65], dedicada a los problemas de satisfacción con restricciones. La monografía cubre muchos de los aspectos relacionados con estos problemas pero obvia un área tradicionalmente muy importante en la comunidad de las restricciones como es la integración de restricciones en los lenguajes de programación declarativos (especialmente los lógicos).

Este artículo describe el *estado-del-arte* de la programación declarativa con restricciones (PDC) con especial énfasis en la integración de restricciones en los lenguajes de programación lógicos. El artículo está dirigido tanto a personas con conocimientos de PDC como a aquellos interesados en conocerla, y cubre sus orígenes históricos, los fundamentos teóricos y las instancias más populares dependientes del dominio de computación.

Palabras clave: Satisfacción de Restricciones, Programación Lógica, Programación Funcional, Resolutor.

1. Introducción

Recientemente, esta revista ha publicado una estupenda monografía [65] enfocada sobre los problemas de satisfacción con restricciones (CSP, *Constraint Satisfaction Problem*) y en la cual se abordan las técnicas clásicas empleadas para su resolución y, más particularmente, otras técnicas usadas sobre áreas específicas de aplicación (tales como la diagnosis, las bases de datos o la recuperación de información). La monografía cubre muchos de los aspectos relacionados con los CSPs pero obvia un área muy importante como es la integración de restricciones en los lenguajes declarativos (especialmente los lógicos). En realidad, la programación lógica con restricciones (CLP, *Constraint Logic Programming*) [110, 179] es uno de los campos de investigación más activos en la comunidad de las restricciones puesto que está directamente relacionado con el modelado (a alto nivel) de soluciones a los problemas de satisfacción con restricciones; más aún, como ya fue apuntado en [54][capítulo 15], uno no puede (ni debe) asumir que todas las restricciones estarán disponibles al comienzo del proceso de búsqueda de las soluciones y el usuario podría querer construir el CSP a resolver de forma incremental y "representar las restricciones a través de fórmulas pertenecientes a un cierto lenguaje de programación en vez de representar éstas mediante conjuntos de tuplas de valores". Este aspecto no fue considerado en [65], y ésa es precisamente la razón que ha generado este artículo.

Este artículo describe el *estado-del-arte* de la *programación declarativa con restricciones*, con especial énfasis en la *programación lógica con restricciones* ya que la naturaleza relacional de las restricciones hace que los lenguajes que guardan algún componente lógico sean más adecuados para la integración de las mismas.

2. Programación con restricciones

La programación con restricciones (CP, *Constraint Programming*) [137, 163] está suscitando un interés creciente en la comunidad investigadora debido a varios motivos:

- El paradigma se sustenta sobre unos fundamentos teóricos muy sólidos [169], lo cual hace de CP un paradigma de programación robusto;
- CP es un campo de investigación muy heterogéneo que abarca desde tópicos puramente teóricos en la lógica matemática hasta aplicaciones totalmente prácticas en la industria. Como consecuencia de ello, el paradigma CP está atrayendo un amplio interés comercial pues es muy apropiado para el modelado de una extensa gama de problemas de optimización y, en particular, esos problemas que involucran restricciones heterogéneas y búsqueda combinatoria.

Básicamente, una restricción es una relación mantenida entre las entidades (e.g., variables u objetos) de un problema. Las restricciones se usan para modelar los problemas reales mediante un enfoque idealizado de la interacción entre los componentes del problema. La forma en la cual esta interacción se define depende de la capacidad y la experiencia del programador.

Un *Problema de Satisfacción de Restricciones* (CSP, *Constraint satisfaction problem*) es un conjunto de restricciones definidas sobre un número finito de variables que están restringidas a tomar valores en un conjunto de dominios de computación. Normalmente, el modelado de un CSP consta de los siguientes pasos [40]:

1. Analizar el problema a resolver para comprender cuales son sus componentes, es decir, identificar las variables restringidas y sus dominios de computación.
2. Determinar la relación entre las variables mediante la identificación de las restricciones que se deben mantener entre ellas.
3. Declarar esas restricciones en forma simbólica, o sea, en forma de ecuaciones, desigualdades, restricciones de (alto-bajo) nivel, restricciones predefinidas, restricciones definidas por el usuario, etc.

Resolver un CSP significa encontrar un conjunto de valores posibles en los dominios de computación de las variables restringidas que garanticen la satisfacción de todas las restricciones que modelan el problema. Pueden existir varios casos

- EL CSP tiene una solución.
- El CSP tiene múltiples soluciones.

  EJEMPLO 1 *Considera un CSP definido con una única restricción $x < y$ donde x e y están inicialmente restringidas a tomar valores en el dominio discreto $\{0, 1, 2\}$. Este CSP tiene entonces tres soluciones:*

$$x = 0,\ y = 1;$$
$$x = 0,\ y = 2;$$
$$x = 1,\ y = 2.$$

- EL CSP no tiene solución. Esto puede ser debido a la imposibilidad de satisfacer todas las restricciones al mismo tiempo (es lo que se llama un *problema sobre-restringido*).

La resolución de CSPs normalmente requiere de tres etapas [23, 184, 185]:

1. La construcción de un modelo para el problema.
2. La resolución del modelo.
3. La comprensión de la solución.

La última etapa pertenece mayormente al área del análisis de programas, y está fuera del alcance de este artículo. La segunda etapa fue abordada recientemente en [65] donde queda claro que la resolución de CSPs suele ser comprendida como la tarea de buscar una solución simple para el problema, aunque a veces se requiere encontrar todo el conjunto de soluciones, y en ciertos casos, a causa del coste para encontrar una solución, el objetivo consiste en encontrar la mejor solución o una aproximación a ésta con respecto a unos recursos limitados (e.g., un tiempo razonable). Estos CSPs suelen llamarse *CSPs parciales* [73].

Actualmente, la resolución de CSPs puede ser realizada mediante el uso de diferentes técnicas,

desde técnicas tradicionales hasta otras mucho más modernas. Existen diferentes métodos de resolución de CSPs tal y como quedó reflejado en [65], y en general estos métodos para generar una solución se clasifican en cuatro categorías [151]: (1) las variantes de la búsqueda por backtracking (con sus versiones más o menos inteligentes) [118], (2) los métodos que minimizan la redundancia en el proceso de búsqueda mediante la eliminación, en los dominios de computación, de aquellos valores que nunca pueden ser parte de una solución (i.e., los *valores inconsistentes*) (estos métodos suelen denominarse *algoritmos de filtrado*, *algoritmos de arco consistencia*, o incluso *algoritmos de propagación*), (3) las técnicas de *forward checking* y las de *look ahead*, los cuales son algoritmos que combinan un método de propagación dentro de un algoritmo de backtracking y (4) los llamados *algoritmos guiados por la estructura* los cuales explotan la estructura del grafo de restricción del problema [151]. Existen otras propuestas tales como los algoritmos que descomponen (con frecuencia mediante el uso de técnicas bien conocidas de la teoría de grafos [158]), el problema en subproblemas los cuales son a su vez resueltos por los métodos descritos anteriormente [72]. La mayoría de estas técnicas (y otras que complementan la propagación como los métodos de enumeración) fueron descritas en [65] por lo que tampoco son consideradas en este artículo.

Sin embargo la primera etapa en la resolución de CSPs, es decir, la construcción de un modelo para el problema no fue debidamente abordada en [65] y parcialmente[1] es considerada en este artículo pues la integración de los mecanismos de resolución de CSPs anteriormente citados dentro del paradigma declarativo ha facilitado el modelado a alto nivel de los CSPs. El resto del artículo es dedicado a mostrar la mayor parte del trabajo que se ha realizado hasta la fecha en este sentido.

3. Programación declarativa

El objetivo fundamental de los lenguajes de programación declarativa en sentido amplio, es proporcionar un alto nivel de abstracción, de forma que *la especificación de un problema sea un programa capaz de resolver el problema.* En definitiva se intenta liberar al programador de describir detalladamente la secuencia de acciones que debe realizar la máquina para obtener el resultado buscado, como es habitual en los tradicionales lenguajes imperativos, en los cuales el programador no sólo tiene que especificar las restricciones, sino que además tiene que definir cómo se resolverán esas relaciones.

EJEMPLO 2 *Considera la siguiente relación entre variables numéricas*

$$a = 2b + c \tag{1}$$

En un lenguaje de programación tradicional, un programador no puede usar esta relación directamente y tiene que definirla dependiendo de los valores conocidos de las variables a, b y c. Por ejemplo, en un entorno imperativo, si b y a son conocidos, entonces esta relación es implementada como sigue:

$$c \leftarrow a - 2b$$

entendida como la asignación a la variable c del resultado obtenido al calcular $a - 2b$. Como consecuencia, el programador requiere un esfuerzo adicional ya que todas las posibles asignaciones derivadas de la relación (1) tienen que ser definidas explícitamente.

Los lenguajes declarativos permiten abstraerse de los detalles concretos del hardware y, en general, los programas son más breves y más sencillos de mantener que los programas imperativos.

Existen varios paradigmas que representan a la programación declarativa. Los dos más importantes, el *lógico* [127] y el *funcional* [148] han evolucionado de forma independiente, pero manteniendo como prioridad común la *expresividad.* Como resultado se han forjado dos estilos de programación distintos que intentan aprovechar las ventajas que ofrece uno u otro enfoque. Recientemente ha aparecido otro paradigma que se incluye dentro de la programación declarativa, es el llamado paradigma *lógico-funcional* [94] que surge como amalgama de las dos vertientes anteriormente citadas, la lógico y la funcional, y en la que se pretende reunir las principales ventajas de ambos paradigmas en uno nuevo. El mecanismo operacional de los lenguajes lógico-funcionales es el resultado de combinar los que utilizan los lenguajes lógicos y los funcionales. En este sentido hay

[1] Observe la calificación de 'parcialmente' puesto que el modelado de CSPs es un área de creciente expansión que tiene que ver no sólo con los lenguajes de programación declarativos sino además con otras muchas cuestiones que permanecen fuera del alcance de este artículo.

varias propuestas, pero uno de los mecanismos más estudiados y extendidos es la "reescritura con unificación" o, más comúnmente, *narrowing (estrechamiento)* [88, 138].

La restricciones han sido integradas en paradigmas de programación muy diferentes, aunque algunos son mas adecuados para ello. Particularmente, como ya se ha hecho notar, las restricciones poseen una naturaleza relacional que hace que los lenguajes con componentes lógicos parezcan más apropiados para su integración. En particular, en el paradigma declarativo, los lenguajes lógicos y los lógico-funcionales parecen ser los más apropiados, mientras que los lenguajes funcionales (precisamente por carecer de un componente lógico) no se amoldan especialmente bien al paradigma de la programación con restricciones. A continuación realizamos un recorrido por las propuestas mas interesantes de integración de restricciones dentro de la programación declarativa, considerando sus instancias más importantes.

4. Programación lógica con restricciones

La programación lógica con restricciones (CLP) [26, 110, 137] nació como una mezcla de dos paradigmas: la programación con restricciones (CP) [14, 54] y la programación lógica (LP, *logic programming*) [127], y su éxito radica en que combina la declaratividad de LP con la eficiencia de CP [65].

4.1. Programación lógica

Este artículo no pretende describir la programación lógica pues es un concepto ampliamente estudiado y conocido. En realidad se espera que el lector este familiarizado, o al menos la conozca en cierta medida[2]. Aún así, incluiremos una breve descripción (i.e., un recordatorio) sobre este tipo de programación.

La signatura del cálculo de LP consta de un conjunto arbitrario de símbolos de predicado y de función, y el paradigma LP se basa en una idea declarativa donde los programas se construyen a partir de implicaciones lógicas entre colecciones de predicados [127]. Un programa lógico consiste en un conjunto de reglas, llamadas *cláusulas*, de la forma $H :\!-B$, donde H se denomina *la cabeza de la regla* y $B = B_1, \ldots, B_n$ $(n \geq 0)$ el *cuerpo de la regla* (el cual se interpreta como una conjunción de átomos). Tanto H como B_i (para $1 \leq i \leq n$) son átomos con la forma $p(t_1, \ldots, t_m)$ $(m \geq 0)$, siendo p un símbolo de predicado y los t_i's son términos, o sea, variables, constantes o una función de aridad m aplicada a m términos. Si $n = 0$, se dice que la cláusula $H :\!-B$ es un *hecho*.

La idea intuitiva de una regla $H :\!-B$ es que, si la conjunción de $B_1, \ldots, B_n$ es verdadera, entonces H también es verdadero. H y B pueden contener variables y el significado de la cláusula es una implicación desde B a H interpretada de la siguiente manera: para todos los valores posibles que puedan tomar las variables que aparecen en H, existen valores que pueden tomar las variables que aparecen en B de tal forma que $B \Rightarrow H$. Observa que si la cláusula es un hecho, esto quiere decir que H es verdadero para cualquier valor de sus variables.

Operacionalmente hablando, ejecutar un programa lógico consiste en preguntar el valor de verdad (i.e., verdadero o falso) de cierta sentencia $:\!-G$, llamada el *objetivo*, que contiene una conjunción de átomos (i.e., $G = G_1, \ldots, G_n$). Esto se interpreta como "preguntar por los valores adecuados de las variables que aparecen en G para que la conjunción $G_1, \ldots, G_n$ sea verdad con respecto al programa lógico". La respuesta a un objetivo es un conjunto, posiblemente vacío si no hay solución, de substituciones (i.e., asignaciones de variables a términos) que hacen que, cuando cada variable es substituida por su valor correspondiente según la asignación, el objetivo G sea verdadero con respecto al programa.

Uno de los puntos clave en la resolución de objetivos es la existencia de *un paso de unificación* realizado en cada una de las etapas del proceso global. Este paso consiste en unificar, a través de una substitución σ, un átomo G_i, que es parte de un (sub-)objetivo `:-G` (e.g., $G = G_1, \ldots, G_n$) con la cabeza H de una cláusula, digamos, $H :\!-B$. En este paso, σ es el unificador más general entre G_i y H, y el objetivo $:\!-G$, es reemplazado por un nuevo (sub-)objetivo (e.g., $:\!-(B, G_2, \ldots, G_n)\sigma$ para $i = 1$) en proceso de resolución.

EJEMPLO 3 *Considera el siguiente programa de una sola cláusula escrito en el lenguaje de programación lógica estándar, Prolog [45, 48]:*

[2]Hoy día, en cualquier carrera informática universitaria hay al menos una asignatura relacionada con la programación lógica.

```
menor_que_tres(X):-integer(X), X < 3.
```

El objetivo `:- menor_que_tres(2)` *devuelve la substitución* {*X=2*} *la cual hace que el cuerpo* {`integer(2), 2 < 3 `} *sea verdad.*

Muy básicamente, la resolución integra otras etapas, tales como el renombramiento de variables o la elección del átomo G_i en el objetivo, que no consideramos en esta sección. No insistiremos más en los detalles de LP y animamos al lector a repasar la bibliografía clásica [166].

Es fácil intuir que LP constituye un marco natural en el cual los CSPs pueden ser modelados a alto nivel pues las restricciones son relaciones entre las variables y los programas lógicos contienen relaciones de variables en forma de cláusulas.

4.2. El esquema CLP

A continuación presentamos el esquema básico de CLP. Este esquema fue presentado por Jaffar y Lassez en 1987 [109], y supone un marco formal, basados en restricciones, para las semánticas operacionales, lógicas y algebraicas de una clase extendida de programas lógicos. A pesar de que los programas de CLP dependen del dominio de aplicación (i.e., el dominio de cómputo), este esquema de CLP fue ideado para la creación de lenguajes lógicos compartiendo el mismo mecanismo de evaluación.

La idea básica en CLP es la de reemplazar la unificación clásica de LP por la resolución de restricciones (i.e., un resolutor) sobre un dominio de cómputo específico. Esta idea dio lugar al esquema CLP($\mathcal{X}$) descrito en [109]. En este esquema, $\mathcal{X}$ representa al dominio de cómputo sobre el cual las restricciones serán resueltas. Las diferentes instancias de $\mathcal{X}$ (e.g., $\Re$, $\mathcal{Z}$, conjuntos, Booleanos, etc.) generan las diferentes instancias del esquema CLP($\mathcal{X}$) (ver Sección 4.5) el cual permite además que CLP pueda resolver problemas que LP no puede por lo que es especialmente interesante para el paradigma declarativo.

EJEMPLO 4 *Considera el programa mostrado en el Ejemplo 3. Un objetivo tal como* `:-menor_que_tres(Y)` *devuelve, en general, un error de instanciación en cualquier lenguaje lógico puesto que la variable Y no esta ligada (a un termino, en este caso un valor entero) previamente a la realización del proceso de resolución. Sin embargo, si lanzamos este objetivo en un sistema lógico con restricciones de enteros, el objetivo devuelve un conjunto de múltiples respuestas indicado por la relación* $Y \in [-\infty, 2]$.

En cada una de las instancias del esquema, el lenguaje de programación lógico subyacente es extendido con un conjunto de operaciones y estructuras que pueden ser usadas sobre el dominio de cómputo y que pueden ser directamente utilizadas por el usuario.

El esquema CLP($\mathcal{X}$) se caracteriza asimismo por un mecanismo de resolución de las restricciones inmerso en el lenguaje lógico subyacente. Este mecanismo, llamado el *resolutor de restricciones*, constituye un procedimiento de decisión capaz de comprobar si una restricción o conjunto de restricciones puede ser satisfecha. En el proceso clásico de resolución de objetivos en LP, el resolutor reemplaza a la unificación estándar por un algoritmo que decide la satisfiabilidad de las restricciones, donde la satisfiabilidad de las restricciones significa decidir la consistencia de las mismas. La mayoría de los resolutores actuales inmersos en lenguajes lógicos son incompletos por lo que suelen estar definidos mediante algún algoritmo de propagación de restricciones [122].

En un contexto general, podemos decir que un lenguaje de CLP puede considerarse un lenguaje lógico donde se han incorporado restricciones y métodos de resolución de restricciones. En este sentido, la diferencia entre LP y CLP estriba en la interpretación operacional de las restricciones y no en su interpretación declarativa.

Resumiendo, la idea crítica del esquema CLP es que, tanto el lenguaje lógico subyacente como sus semánticas operacionales y declarativas pueden ser parametrizadas por un dominio de cómputo $\mathcal{X}$ y las operaciones sobre este dominio.

4.3. Aplicaciones de CLP

CLP ha sido aplicada con bastante éxito en la resolución de problemas reales como por ejemplo aplicaciones clásicas resueltas por técnicas OR como la búsqueda combinatoria [61], problemas de "cutting stock" [60] y problemas de planificación [51]. Otras áreas donde CLP ha demostrado su potencia son la diagnosis de fallos en circuitos electrónicos [17, 162], redes neuronales [117], aplicaciones industriales [38], visualización de relaciones con datos biológicos [86], manejo de robots

[156], reconstrucción de secuencias originales de ADN a partir de fragmentos de particiones de enzima [187] y reconstrucción tridimensional de modelos de proteínas [168] entre otras muchas aplicaciones. Más información sobre las aplicaciones prácticas de CLP puede encontrarse en [161] y [181].

4.4. Algunas referencias clave

A continuación proporcionamos algunas referencias que pueden ayudar al lector interesado a una mejor comprensión de CLP. Para empezar, [81] y [123] proporcionan una introducción informal a CLP, y [46] presenta una introducción breve (con una visión histórica) a los lenguajes CLP.

Desde un punto de vista más general, [50] y [110] describen los aspectos globales más interesantes de CLP en términos de los conceptos fundamentales (cubriendo teoría, práctica e implementación de lenguajes de restricciones).

Desde un punto de vista pedagógico, [40] sirve como curso de introducción a CLP. Esta guía está principalmente dirigida a programadores industriales con poca o nula experiencia con CP.

[78][Capítulos 4-5] y [54][Capítulo 15] contienen, de una manera muy clara y concisa, descripciones de las semánticas operacionales y declarativas de CLP, y cubren como otros aspectos relacionados, aunque para obtener una descripción más profunda de las semánticas de CLP, proponemos [110] y [111]. También, [137] dedica una amplia parte del libro a los lenguajes CLP.

4.5. Instancias de CLP

Como se ha declarado en la Sección 4.2, el esquema de CLP está parametrizado con respecto al dominio de cómputo subyacente sobre el cual las restricciones se resuelven, y cada dominio da lugar a una instancia del esquema. Los resolutores son específicos para este dominio por lo que diferentes dominios demandan diferentes mecanismos de resolución para la satisfacción de restricciones.

Esta sección realiza un recorrido por las instancias de CLP más importantes, lo cual ayudará a comprender los aspectos más relevantes de CLP.

4.5.1. El dominio finito

De los dominios de CLP, el dominio finito (FD, *Finite Domain*) [173] es uno de los más estudiados pues es un marco adecuado para la resolución de problemas de satisfacción de restricciones en los cuales las variables toman valores en dominios discretos. La importancia de los lenguajes de CLP con restricciones FD radica en su impacto en la industria ya que muchos problemas reales involucran variables tomando valores en dominios finitos. La consecuencia es que los lenguajes de CLP para el FD pueden resolver muchas aplicaciones prácticas en el mundo industrial. FD es particularmente útil para modelar problemas en áreas tales como la investigación operacional, el diseño de hardware o la inteligencia artificial. Problemas tales como planificación, empaquetado, composición de horarios, y otros pueden ser modelados mediante variables FD por lo que muchos sistemas CLP actuales dan soporte para este tipo de resolutores.

Los algoritmos de programación existentes en este tipo de sistemas CLP suelen llamarse *técnicas de consistencia* o *algoritmos de filtrado* [122] y surgieron como una alternativa a la ineficiencia de los procedimientos *generar-y-chequear* y el backtracking estándar de LP. Estas técnicas se basan en la idea de una poda *a priori* i.e., las restricciones se usan para reducir el espacio de búsqueda antes de encontrar un fallo; esto permite reducir el número de backtrackings así como el de los chequeos de la satisfiabilidad de las restricciones. Estas técnicas fueron derivadas a partir del algoritmo de filtrado de Waltz [182] y el algoritmo del resolutor REF-ARF [69]. Con posterioridad, estos trabajos fueron desarrollados y extendidos en [71, 83, 133]. A su vez, [141, 149] proporcionan una descripción general de los trabajos pioneros sobre los algoritmos de satisfacción con restricciones.

Las técnicas de consistencia de los sistemas de CLP se han descrito, de forma excelente, en [173], donde además se enseña como se pueden integrar en los lenguajes de programación lógica sin pérdida de declaratividad. El principio fundamental de LP se mantiene para CLP pues el "programador no necesita preocuparse sobre si la poda del árbol de búsqueda se realiza *a priori* (i.e., vía las técnicas de consistencia) o *a posteriori* (i.e., vía backtracking clásico)".

El primer lenguaje de CLP para FD fue CHIP [62, 172] y su propósito era el de resolver, de

manera eficiente y flexible, una clase amplia de problemas combinatorios [61]. En realidad, CHIP no sólo se diseñó para FD sino que además admitía otros dos dominios de cómputo, el Booleano y el de los racionales. Cada uno de los dominios de cómputo soportados por CHIP tenía asociado su mecanismo de resolución específico que se implementó internamente de forma opaca para el usuario. Por ejemplo, en el dominio de los racionales, se empleaba un algoritmo como el del Simplex, mientras que en el dominio Booleano se empleaba la unificación Booleana. El éxito de CHIP fue enorme, de tal manera que CHIP fue tomado como una referencia para mostrar las posibilidades de las técnicas de consistencia [173].

Desde la aparición de CHIP, la propagación de restricciones en FD ha sido ampliamente estudiada, y obviamente mejorada. En particular, el sistema más exitoso de CLP sobre FD es clp(FD) [58] el cual proporciona una única restricción primitiva sobre la cual el usuario puede diseñar nuevas restricciones definidas a alto nivel (es lo que se suele denominar *transparencia*). Su eficiencia mostrada en la resolución de problemas popularizó el sistema dentro de la comunidad de las restricciones [57]. Otro sistema transparente (o sea, que permite el modelado de nuevas restricciones por parte del usuario) alternativo, es el implementado en el lenguaje B-Prolog [188, 190], que da soporte a unas construcciones específicas para la resolución de restricciones de dominio finito, y además ha sido demostrado que es bastante eficiente [66].

Para más información, proponemos leer [99] que da una idea general sobre el uso de la programación de restricciones en el dominio finito para la resolución de problemas combinatorios.

4.5.2. El dominio continuo

Muchas aplicaciones requieren cálculos numéricos en el dominio de los reales. El primer sistema de CLP que permite la resolución de restricciones sobre los reales fue el sistema CLP($\Re$) [113]. Este sistema es en sí mismo una instancia del esquema CLP, por lo que su semántica operacional es similar a la de Prolog, aunque como es usual en CLP, la unificación se reemplaza por un mecanismo de satisfacción con restricciones, el cual, en el caso de CLP($\Re$), consiste en una resolución de restricciones sobre el dominio de los funtores sin interpretar aplicados a términos reales aritméticos. Básicamente, CLP($\Re$) es un lenguaje de programación lógica con restricciones de aritmética real. Su implementación contiene un resolutor opaco que resuelve ecuaciones lineales y que permite retrasar la evaluación de restricciones no lineales hasta que éstas lleguen a ser lineales [112]. Este mecanismo de retraso se describe detalladamente en [115] mientras que [114] describe una máquina abstracta para el sistema. Por supuesto, CLP($\Re$) también puede usarse como un lenguaje de programación lógica de propósito general (pues CLP($\Re$) está basado en Prolog).

Un programa CLP($\Re$) es una colección de reglas en el sentido de las cláusulas de Prolog y con la diferencia de que el cuerpo de una cláusula puede contener también restricciones definidas sobre variables reales. Las restricciones son básicamente ecuaciones o desigualdades de la forma

$$Expresion_1 \bullet Expresion_2,$$

donde $\bullet \in \{=, >, \geq, <, \leq\}$, y las expresiones a ambos lados se construyen a partir de constantes reales, variables, términos reales negados (i.e., aplicando el signo $-$) y términos más complejos construidos mediante la aplicación arbitraria de los operadores aritméticos clásicos tales como $+, -, *$ y $/$. Así,

$$X + 3{,}0 * Y \leq Z \qquad \text{y} \qquad 3{,}451 + W = 24 * Y$$

son ejemplos de restricciones en CLP($\Re$).

Este sistema fue extendido para incluir capacidades para la meta-programación con restricciones [98]. Desde un punto de vista práctico, CLP($\Re$) ha demostrado su potencia en áreas muy diversas tales como la biología molecular [186], el chequeo de protocolos [90], la ingeniería eléctrica [97], la diagnosis de circuitos analógicos basada en modelos [30] y en problemas de "option trading" [105] entre otras muchas aplicaciones prácticas.

El éxito del sistema CLP($\Re$) fue propagado a varios sistemas que integraron el manejo de restricciones reales aritméticas en sistemas de programación lógica. Por ejemplo, clp(Q,R) [103] es otro lenguaje histórico que permite la resolución de ecuaciones lineales sobre variables reales y racionales, y que además aporta un tratamiento perezoso de las ecuaciones no lineales y un algoritmo de decisión para las desigualdades lineales que detecta la ecuaciones implícitas, elimina redundancias, ejecuta proyecciones (i.e., eliminación de cuantificadores) y proporciona optimización en la resolución. Este sistema forma parte hoy día de los más importantes sistemas de programación lógica tales como SICStus Prolog [39], ECL^iPS^e [4] y CIAO Prolog [100].

También CHIP, el sistema pionero de CLP sobre el dominio finito, posibilitaba cierta clase de resolución de restricciones sobre términos racionales, donde un término racional se construye a partir la combinación de valores racionales (i.e.,constantes) con las operaciones + y *. CHIP resuelve entonces ecuaciones lineales y desigualdades a través de un algoritmo especializado tipo Simplex, por lo que el tipo de problema que resuelve involucra la programación lineal.

La resolución de restricciones no lineales sobre los números reales es una tarea compleja que no fue realmente resuelta en CLP($\Re$) puesto que el proceso de resolución era retrasado hasta que las ecuaciones se hacían lineales. Este es un método de implementación eficiente pero tiene la desventaja de que algunas veces las respuestas calculadas son insatisfacibles o se entra en un bucle infinito. Fue por estas razones por las que, partiendo desde el enfoque original de CLP($\Re$), otros enfoques de CLP intentan mejorar la resolución de las restricciones no lineales.

[104] experimentó la combinación de CLP con dos métodos algebraicos, el método de *descomposición cilíndrica parcial* [16] y el de *las bases de Gröbner* [35], para la resolución de restricciones reales no lineales. Estos métodos fueron aplicados con éxito en décadas anteriores en la disciplina del álgebra de los ordenadores. Se creó una implementación prototipo, plasmada en el lenguaje RISC-CLP(Real), donde se demostró que la combinación fue provechosa.

También [93] propuso otro método alternativo para resolver el problema de retrasar la evaluación de las restricciones no lineales. La propuesta consistía en definir un resolutor más especializado, a la manera de ése implementado en el lenguaje RISC-CLP(Real), y aceptar un compromiso entre los dos extremos, identificando en cierta forma una clase de programas CLP($\Re$) para los cuales todas las restricciones no lineales retrasadas llegan a ser lineales en tiempo de ejecución. Los programas pertenecientes a esta clase pueden ser ejecutados, de forma segura, con el método eficiente de CLP($\Re$), mientras que los restantes programas necesitan un resolutor mucho más poderoso.

Posteriormente, el sistema QUAD-CLP($\Re$) [147] fue implementado encima del sistema CLP($\Re$), y su objetivo era evitar de nuevo el retraso incondicional de las restricciones no lineales, concentrándose en las restricciones cuadráticas las cuales reescribe de tal manera que puede decidirse si se genera una aproximación conservadora de las mismas (manteniéndolas retrasadas) o si se mejora el control sobre el proceso de cómputo. En ambos casos, la idea consiste en manejar las restricciones en una forma más sencilla. QUAD-CLP($\Re$) representa un resolutor incompleto de restricciones no lineales adecuado para la resolución de problemas de cierto tamaño.

La investigación sobre la resolución de restricciones no lineales es todavía uno de los asuntos más activos en la comunidad de C(L)P.

4.5.3. Conjuntos

El conjunto finito es otro de los dominios de cómputo más empleados en CLP pues los conjuntos permiten el modelado de problemas combinatorios y de problemas de procesamiento del lenguaje natural. Además, muchos problemas reales pueden ser expresados mediante relaciones o grafos sobre conjuntos o multiconjuntos.

El primer intento de utilizar conjuntos en CLP fue realizado en [180] donde se proponen los conjuntos regulares sobre *palabras* como un nuevo dominio de cómputo en el lenguaje CLP(Σ^*). Los conjuntos regulares son definidos como conjuntos finitos compuestos de cadenas de caracteres finitas, y las restricciones en CLP(Σ^*) tienen la forma de restricciones de intervalo. Por ejemplo, la restricción

$$A \; in \; (X.\text{“}ab\text{”}.Y)$$

asocia la variable A a cualquier cadena de caracteres que contenga la subcadena “*ab*”. Aunque los conjuntos regulares no representan, en el concepto general, a los conjuntos, sí que podemos afirmar que CLP(Σ^*) fue la primera iniciativa de manejo de conjuntos en el marco CLP.

Posteriormente, CLPS [125] fue otro intento, basado en la noción de conjuntos de profundidad finita sobre los términos de Herbrand (e.g., el conjunto $\{1, 2, 3, 4\}$ tiene profundidad 1, el conjunto $\{\{1, 2\}, 3, 4\}$ profundidad 2, el conjunto $\{\{\{1, 2\}\}, 3, 4\}$ profundidad 3, etc.). La satisfacción de restricciones consiste en el chequeo de la consistencia de las variables de dominio restringidas en el dominio de estos conjuntos.

Uno de los trabajos más influyentes fue descrito en [84] donde un lenguaje de CLP, llamado *Conjunto*, permitía el manejo de restricciones de intervalo de conjuntos. La motivación para este lenguaje fue el de combinar las técnicas de sat-

isfacción con restricciones con la declarativid de Prolog para resolver CSPs definidos sobre conjuntos o grafos (e.g., división - particionamiento - de conjuntos, empaquetado de conjuntos, cubrimiento de conjunto máximo-mínimo, etc.). En esta línea, [85] define un procedimiento determinista de unificación de conjuntos consistente en chequear, en tiempo polinomial, la igualdad entre las variables de los conjuntos ligados. Para ello, en vez de usar el dominio de conjuntos como dominio de cómputo, se definió una aproximación, el dominio de los intervalos cerrados (especificados por su cota inferior y su cota superior) de conjuntos lo que garantizaba un ordenamiento parcial. En este contexto, una variable de conjunto s es asociada a un intervalo de conjunto mediante una restricción de la forma

$$s \in [l, u] \qquad \text{con } l \subseteq u$$

donde l y u son conjuntos de enteros e.g., $s \in [\{1\}, \{1, 3, 5, 7\}]$. Este enfoque significa una mejora importante con respecto a los trabajos previos puesto que proporciona la posibilidad de usar técnicas de consistencia para razonar en términos de variaciones de intervalo.

Más recientemente, [119, 120] proponen usar restricciones para definir un lenguaje de CLP sobre conjuntos de términos ligados. Este enfoque generaliza de alguna forma la programación lógica clásica sobre el dominio de Herbrand. También, [63] desarrolla el lenguaje $\{log\}$ asentando las bases para el uso de restricciones de la forma $\{x\} \cup Set$ en un lenguaje lógico. La satisfacción de las restricciones se basa en una selección no determinista de las restricciones en la cual se tienen en cuenta todas las posibles substituciones entre los elementos de dos conjuntos. Desafortunadamente, esta propuesta da lugar a un crecimiento exponencial del árbol de búsqueda.

Actualmente, el dominio de los conjuntos se encuentra incluido dentro de los principales sistemas de CLP, tales como ECL^iPS^e [4] u Oz[3] [140]. Para conseguir más información sobre los conjuntos como dominio de cómputo en (C)LP, recomendamos leer [85, 167].

4.5.4. (Pseudo-)Booleanos

El dominio Booleano es otro de los mas estudiados dentro de CP; la razón es la utilidad de este dominio puesto que muchas aplicaciones y problemas reales se formulan naturalmente empleando variables Booleanas (a veces también llamadas *variables 0-1*) [12]. El estudio de los problemas Booleanos [184] surge del modelado, mediante variables Booleanas, de problemas en campos diversos tales como la demostración automática de teoremas, la verificación y la diagnosis de circuitos, la inteligencia artificial, etc. Por ello, no es sorprendente que este dominio haya dado lugar a una instancia del esquema CLP, en el cual a veces se ha empleado un resolutor específico y especialmente dedicado a la resolución de problemas Booleanos, y otras veces se ha proporcionado en forma de librería para el manejo de restricciones Booleanas.

El procesamiento básico de un conjunto de restricciones Booleanas consiste en determinar la satisfiabilidad de las mismas. Un resolutor de restricciones Booleanas normalmente da soporte para la resolución de, al menos, las siguientes fórmulas lógicas:

$$X \vee Y \equiv Z,$$
$$X \wedge Y \equiv Z,$$
$$X = Y \text{ y}$$
$$X \equiv \neg Y$$

en las cuales las variables X, Y y Z son variables Booleanas que por lo tanto toman valores en el dominio Booleano. Por supuesto, es deseable que el resolutor además proporcione soporte a otras fórmulas tales como el *or exclusivo*, el *not and*, el *not or*, la *equivalencia* o la *implicación*. No obstante, con las primeras cuatro fórmulas es posible definir el resto.

Merece la pena clarificar la distinción entre un resolutor Booleano específico, es decir, esos dedicados exclusivamente a la resolución de fórmulas Booleanas, y esos proporcionados por un sistema de CLP como un módulo de librería con la capacidad de permitir el uso de restricciones Booleanas y resolverlas. En esta última categoría se encuentra, el ya nombrado, lenguaje CHIP [36] y el lenguaje Prolog III [25], que ofrecen algoritmos Booleanos de propósito especial en los cuales el procesamiento de restricciones Booleanas se realiza en el paso de unificación (como es habitual en CLP). La única restricción realmente necesaria es la igualdad entre los términos Booleanos. En estos lenguajes, un término Booleano se compone mediante la combinación de constantes, variables

[3]Oz, hoy en día llamado Mozart, debe ser considerado más como un lenguaje multiparadigmático que como un lenguaje de CLP (véase la Sección 8).

y los operadores lógicos *and, or, not, xor, nand* y *nor*. El algoritmo de unificación (i.e., ése para calcular el unificador más general entre dos términos Booleanos) está basado en el concepto de eliminación de variable.

Otros algoritmos especializados para la resolución de restricciones Booleanas, son nombrados a continuación. Por ejemplo, [157] propone un algoritmo para el lenguaje CAL [5, 6] (en realidad, es una aplicación del algoritmo de Buchberger [34] sobre anillos Booleanos). El lenguaje CAL se basa en un álgebra con valores simbólicos, donde la igualdad entre fórmulas Booleanas expresa la equivalencia en el álgebra. También, otros lenguajes de LP tales como GNU-Prolog [59], Prolog IV [142], IF/Prolog [107] SICStus [160] y B-Prolog [189] incorporan módulos de librería para el manejo de restricciones Booleanas.

La idea de considerar el dominio Booleano como una instancia del subconjunto $\{0, 1\}$ de los enteros, es decir, como un caso particular de dominio finito, fue por primera vez empleada en el lenguaje CHIP. Esta idea permite una generalización de las fórmulas lógicas como por ejemplo la idea de las restricciones reificadas [137][capítulo 8]. Este enfoque de CHIP consiguió tanto éxito que llegó a convertirse en el estándar en la versión comercial de CHIP, mientras que el resolutor Booleano especializado mediante una algoritmo complejo podía ser escogido como opción auxiliar. Las restricciones primitivas Booleanas de CHIP fueron codificadas internamente de forma opaca para el usuario (i.e., lo que se denomina *enfoque de caja negra*). Posteriormente, la idea fue extendida al sistema clp(FD/B) [42]. Esta extensión consistió en la integración del resolutor Booleano en el resolutor de restricciones de dominio finito del sistema clp(FD). Las restricciones Booleanas tales como *and*, *or* y *not* fueron descompuestas en expresiones más simples de la forma X *in* r donde r toma valores en el dominio finito $\{0, 1\}$. El esquema de propagación era muy simple y convertía el enfoque de caja negra de CHIP en un enfoque transparente. Además, [44] demostró que clp(FD/B) presentaba más eficiencia (sobre un orden de magnitud en velocidad) que la del resolutor Booleano de CHIP. Más aún, este enfoque transparente era incluso más eficiente que algunos resolutores Booleanos de propósito especial. Por ello, el sistema clp(FD/B) se especializó para el dominio Booleano y dio lugar al sistema clp(B) [43], el cual posee optimizaciones específicas para dicho dominio y elimina las estructuras de datos del sistema clp(FD/B) que se usaban en el dominio finito pero no en el Booleano. El sistema resultante fue un resolutor simple y compacto, más eficiente que el sistema anterior.

Recientemente, una generalización de las restricciones Booleanas que abarca las llamadas restricciones *pseudo-Booleanas* está siendo muy estudiada. La restricciones pseudo-Booleanas son ecuaciones o desigualdades entre polinomios enteros multilineales definidas en variables 0-1 (i.e., variables en las cuales un 0 representa el valor *falso* y un 1 el valor *verdadero*) [22, 33]. Las restricciones pseudo-Booleanas son por tanto una forma restringida de las restricciones de dominio finito.

EJEMPLO 5 *Para modelar la interacción de n objetos* $ob_1, \ldots, ob_n$*, cada uno de los cuales puede ser seleccionado o no, es natural usar una función cuadrática pseudo-Booleana de la forma*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} X_i X_j.$$

donde a_{ij} *mide la interacción entre los objetos* ob_i *y* ob_j *y las variables 0-1 de decisión* X_i *indican si* ob_i *se selecciona o no [23]. Como la mayoría de los resolutores existentes no pueden manejar restricciones no lineales pseudo-Booleanas, la técnica estándar consiste en linearizarlas y resolverlas entonces en un resolutor de restricciones 0-1.*

En la literatura encontramos varios lenguajes de CLP desarrollados para el manejo de restricciones pseudo-Booleanas. Por ejemplo, el lenguaje clp($\mathcal{PB}$) que permite el modelado y la resolución de problemas 0-1, fue presentado en [32] y una implementación prototipo fue descrita en [21]. Dado un conjunto de restricciones Booleanas 0-1, el resolutor calcula un conjunto equivalente de cláusulas más simples las cuales son suministradas a una resolutor de restricciones 0-1 para su resolución.

4.5.5. Intervalos reales

El dominio real es un dominio continuo y los algoritmos para resolver restricciones definidas en el dominio real se estudian en un marco idealizado. Sin embargo, en la práctica, la validez de estos algoritmos no es aceptada pues los números reales son aproximados por los números en formato punto flotante, lo que impide una precisión absoluta en la resolución de restricciones reales, y por tanto existirán errores de aproximación. En realidad, los métodos convencionales de cálculo numérico

requieren de aproximaciones punto flotante a un conjunto finito de valores reales, es decir, básicamente encuentran soluciones aproximadas (dentro de los límites de una cota de error) a un conjunto de restricciones.

Como alternativa aparecen los métodos basados en aritmética de intervalos y que calculan una solución como una unión de intervalos de tal forma que la verdadera solución permanece en alguno de ellos. Sólo aquellos valores que no pertenecen a la solución son eliminados y por tanto es posible garantizar que los valores reales de una solución están el intervalo computado como solución. Para trabajar con métodos de intervalo, uno tiene que encontrar la correcta abstracción de números en punto flotante en términos de números reales.

La primera publicación sobre la aritmética de intervalos es atribuida a Moore [139]. Moore reemplaza cada constante real por un intervalo que la contiene, y extiende las operaciones de los reales a operaciones de intervalo de tal forma que el error de redondeo queda limitado en cada operación. Si el mérito de la aritmética de intervalos es para Moore, el origen de las restricciones de intervalo es atribuido a Waltz y su artículo [183] en el cual las restricciones eran propagadas con el fin de reducir los conjuntos de valores posibles en una solución. El paso siguiente fue la aplicación de los algoritmos de propagación sobre las restricciones reales de intervalo [52].

El mérito de integrar la aritmética de intervalo en el paradigma de programación lógica es para Cleary [41] que extiende el enfoque típico de la aritmética de intervalo en un marco funcional a la teoría relacional. Independientemente, [106] también descubrió las restricciones de intervalo y la aplicación de la aritmética de intervalos a la resolución de restricciones. Los métodos básicos de las restricciones de intervalo fueron mejorados mediante la incorporación del método de Newton [28, 176]. Más recientemente, Hickey [101, 102] ha avanzado en la búsqueda de un marco unificado para las restricciones de intervalo y la aritmética de intervalo.

En general, la idea básica de la instancia CLP(Interval) consiste en evaluar cada expresión numérica usando intervalos en vez de números en punto flotante, evitando con ello la pérdida de precisión numérica. Se ha demostrado que CLP(Interval) es una instancia muy potente para la resolución de restricciones no lineales.

Desde un punto de vista de la implementación, la instancia CLP(Interval) se consigue construyendo en el lenguaje de LP un resolutor de restricciones de intervalo que garantice la completitud de las soluciones (i.e., todas las soluciones del problema son retenidas). CHIP [124, 172] fue el sistema pionero que mostró que la idea de la restricciones de intervalo era aplicable de manera amplia. Posteriormente, el sistema BNR-Prolog [146] combinó las ideas de Davis y Cleary con un lenguaje lógico (i.e., Prolog). En general, es posible decir que, históricamente, han existido dos enfoques fundamentales para implementar resolutores de restricciones con aritmética de intervalos [27]. Uno de ellos, el enfoque de *consistencia de casco* (i.e., *hull consistency*) está representado por el sistema CLP(BNR) [24, 145, 146, 29], y el otro enfoque, el enfoque de *consistencia de caja* (i.e., *box consistency*) está representado por los sistemas Newton [28, 175] y Numerica [174, 176].

- *Consistencia de casco.*

  Este enfoque se basa en la traducción de las restricciones aritméticas complejas en restricciones primitivas para posteriormente ejecutar una contracción de estas restricciones (i.e., una reducción de dominios). El mecanismo de propagación es el usual, o sea, las restricciones comparten variables de manera que la contracción de restricciones normalmente tiene que ser ejecutada varias veces sobre cualquier restricción: cada vez que alguna restricción hace que el intervalo de una variable se reduzca, todas aquellas restricciones que contengan esa variable tienen que reactivarse para dar lugar a nuevas contracciones. A menudo, la terminación de este proceso se garantiza teniendo en cuenta que los reales son números en punto flotante y, por lo tanto, habrá como mucho un número finito de contracciones.

  La principal desventaja de este enfoque es que la descomposición de las restricciones complejas, introduce nuevas variables lo cual da lugar a aproximaciones innecesarias.

  CLP(BNR) es un lenguaje lógico basado en Prolog que incorpora un algoritmo de arco consistencia para restricciones de intervalo, permitiéndose el manejo de restricciones algebraicas sobre variables reales, enteras y/o Booleanas. Esto permite que los programadores puedan expresar sistemas de ecuaciones no lineales sobre intervalos reales que además pueden combinarse de forma arbitraria con ecuaciones definidas

sobre variables enteras o Booleanas. En CLP(BNR) cada restricción se descompone en restricciones primitivas, y un mecanismo de resolución de restricciones es invocado repetidamente para contraer cada restricción primitiva hasta que cierta condición de terminación sea satisfecha.

- *Consistencia de caja.*

  Los sistemas basados en este enfoque incorporan un algoritmo de resolución de restricciones como una combinación de métodos numéricos tradicionales tales como los métodos de procesamiento de intervalo y las técnicas de satisfacción de restricciones. En este sentido, la consistencia de caja permite el procesado eficiente de restricciones complejas sin descomposición tales como la resolución de sistemas de desigualdades y ecuaciones no lineales así como problemas de optimización.

Recientemente han aparecido algunos lenguajes que combinan los dos enfoques tales como el lenguaje DecLIC [91].

El lector interesado puede consultar referencias estándares en restricciones de intervalo tales como [29, 144, 171]. Para obtener una visión general de las aplicaciones de la aritmética de intervalo en un entorno relacional se puede consultar [135]. Otras referencias tradicionales son [11] y [92].

4.5.6. Árboles

CLP ha sido también aplicada al dominio de los árboles puesto que éstos facilitan el modelado de problemas que con otros dominios no se pueden modelar, y además, los árboles pueden ser recorridos y procesados de formas diferentes mostrando una gran capacidad para la resolución de restricciones. Más aún, en árboles ordenados, la búsqueda puede ser relativamente barata.

El dominio de Herbrand es el único dominio de LP, y por lo tanto está presente en todos los lenguajes CLP. En realidad, LP puede ser considerada como CLP aplicada al dominio de Herbrand, donde los términos de Herbrand son una representación de *árboles finitos* y las restricción principal es la igualdad. Algunos sistemas existentes pueden manejar restricciones sobre el universo de Herbrand. Por ejemplo, el sistema HAL [55, 56], el cual es un nuevo lenguaje de programación lógica específicamente diseñado para soportar *la construcción de* y *la experimentación con* resolutores de restricciones, proporciona restricciones (i.e.,'tests') para chequear la igualdad de términos ligados.

El sistema de restricciones FT [10] también ofrece una estructura de datos universales basada en árboles, y presenta una alternativa a las restricciones de Herbrand sobre los árboles de constructores. Los constructores en FT son más generales que los de Herbrand, y las restricciones de FT son de granularidad más fina y de diferente expresividad. La novedad esencial de FT es suministrada por atributos funcionales llamados *features* que permiten representar datos como registros "extensibles", una forma más flexible que la ofrecida por los constructores de aridad fija de Herbrand.

Hoy día, existen lenguajes de LP tales como Prolog III [47] y SICStus [160] que dan soporte a un dominio de computación basado en *árboles racionales*. La importancia de este dominio se demuestra por el hecho de que Prolog III y SICStus utilizan la unificación de árbol racional como el resolutor por defecto [116]. Un árbol racional es un árbol con un número de nodos posiblemente infinito pero donde el número de ramas que emanan de cada nodo es finito. El uso de estos árboles acelera la unificación (debido a la omisión de occurs-check) e incrementa la declaratividad. Desafortunadamente, su uso también da lugar a un sorprendente número de problemas [18]. Por ejemplo, muchos predicados predefinidos están definidos de forma anómala para estos árboles y necesitan ser suplementados con chequeos en tiempo de ejecución cuyo coste puede ser muy significativo. Obsérvese también que el dominio de los árboles finitos es el dominio de computación por excelencia de la mayoría de los lenguajes de LP y, por lo tanto, algunas técnicas de manipulación de programas ampliamente utilizadas, asumen este dominio de computación. Esto quiere decir que estas técnicas o no son aplicables a los árboles infinitos o no han sido probadas a ser correctas en el dominio de los árboles racionales.

5. Programación funcional con restricciones

La programación funcional consiste en otro paradigma declarativo bastante diferente del enfoque lógico y que tiene una serie de características muy diferenciadoras con respecto al en-

foque imperativo. Entre éstas encontramos algunas tales como el tratamiento uniforme de los programas como funciones, el tratamiento de las funciones como datos, la limitación de los efectos laterales y el uso de manejo automático de memoria. Como resultado, un lenguaje funcional proporciona una gran flexibilidad, programas concisos y semánticas simples.

Los lenguajes funcionales poseen una serie de características, tales como la recursión, la abstracción funcional, los tipos y las funciones de orden superior, que han influenciado, o han llegado a ser parte de, muchos lenguajes de programación modernos y que deberían considerarse en la implementación de cualquier lenguaje se programación. La fortaleza de estos lenguajes se ve reforzada además por la evaluación perezosa, concepto que dota a estos lenguajes de una característica no existente en ningún lenguaje lógico.

Sin embargo, como ya hemos comentado, la programación funcional por sí misma no parece ser un marco demasiado adecuado para la integración de restricciones. En efecto, como tendría que haber quedado claro, las restricciones guardan un componente relacional que parece no tener cabida en el paradigma funcional, a pesar de poseer éste un componente eminentemente declarativo. Este hecho hace que las restricciones no puedan ser insertadas de forma natural, como ocurría en el marco lógico, en los lenguajes funcionales aunque existen diversas propuestas que son descritas a continuación.

5.1. Programación funcional

En esta sección comentamos brevemente los fundamentos teóricos de la programación funcional pura. El concepto predominante en la programación funcional es el de función. Muy básicamente, una *función* es una regla que asocia a cada valor x de un conjunto $\mathcal{X}$ de valores un único valor y de otro conjunto $\mathcal{Y}$ de valores. En notación matemática, si f es el nombre de una función, entonces escribimos:

$$f :: \mathcal{X} \rightarrow \mathcal{Y}$$
$$f(x) = y$$

El conjunto $\mathcal{X}$ es denominado el *dominio* de f y el conjunto $\mathcal{Y}$ el *rango* de f. En los lenguajes de programación se debe distinguir entre *definición de función* y entre *aplicación de función*. Lo primero consiste en describir cómo será evaluada la función usando parámetros formales, mientras que lo segundo consiste en una llamada a la función donde los parámetros actuales reemplazan a los formales en la definición de la función.

En la programación funcional pura no existen variables en la aplicación de una función, y sólo se admiten constantes y valores (o llamadas a otras funciones en el caso de orden superior). En general se pierden tanto las variables como la asignación, aunque esto sólo ocurre en la programación funcional pura. Además, si en los lenguajes lógicos, el mecanismo operacional de evaluación se basa en la unificación y en la resolución, en los lenguajes funcionales juega un papel fundamental el proceso de reescritura, que básicamente consiste en reemplazar definiciones más amplias con otras cada vez más concretas y definidas.

Tradicionalmente la programación funcional, al igual que la lógica, ha sido considerada como ineficiente en comparación con la programación imperativa. Las causas son varias: por un lado, a causa de su naturaleza dinámica, los lenguajes funcionales han sido históricamente interpretados, más que compilados. Incluso cuando los compiladores llegaron a ser accesibles a estos lenguajes, la velocidad obtenida no es la adecuada. Recientemente se ha avanzado mucho en las técnicas de compilación que, junto los avances en las técnicas de interpretación, han hecho estos lenguajes muy atractivos para el usuario. Aún así, el problema de la eficiencia no está totalmente paliado y hay otras causas tales como la dependencia de las llamadas a funciones y el manejo de la memoria dinámica [131]. Para solventar la ineficiencia han surgido nuevos enfoques, como la recolección generacional de basura y nuevas técnicas de traducción, que incrementan el rendimiento de estos lenguajes. Algunos artículos que cubren estas cuestiones son [82] y [165].

Existen multitud de libros sobre programación funcional. Si se tiene interés en lenguajes específicos de programación funcional podemos citar unos cuantos tales como ML [170], Haskell [31, 64], o Scheme [3], entre otros. Con respecto a las técnicas generales para programación funcional recomendamos [121] y [143], y con respecto a la implementación [13] y [148]. También, para cualquier persona interesada en la programación funcional, es importante comprender el lambda-cálculo [19], pues muchos lenguajes funcionales (tales como Scheme, ML y Haskell) están basados en él (el lambda-cálculo proporciona una for-

ma simple y concisa de cómputo y fue inventado por Lorenzo Church).

5.2. Funciones y restricciones

Como ya se ha comentado, la integración de restricciones en este paradigma no ha provocado tanto éxito como en la programación lógica por lo que en la literatura existen pocas propuestas a tener verdaderamente en cuenta con respecto a la integración de restricciones en lenguajes funcionales. Una de ellas, la ofrecida en el lenguaje FT, ya ha sido descrita en la Sección 4.5.6. La otra ofrece un enfoque alternativo consistente en extender el lambda-cálculo [19, 20] (i.e., el sistema estándar de reescritura que proporciona el mecanismo de evaluación para muchos lenguajes funcionales) para que incluya un almacén de restricciones globales. La restricciones son enviadas a este almacén a través de la aplicación de funciones. El problema ahora es cómo determinar el rol activo de este almacén en la evaluación de los objetivos del programa. [49] describió un método llamado *el lambda-cálculo restringido*, en el cual únicamente los valores definidos pueden ser comunicados desde el almacén. El almacén se usa entonces para determinar el valor de las variables de forma que si el valor de una variable está definido entonces reemplaza la variable por su valor en la lambda-expresión. El problema actual es que el almacén juega un papel demasiado pasivo en este proceso (especialmente con respecto al proceso de búsqueda ya que no puede guiarlo).

A pesar de no ser un marco adecuado para la satisfacción de restricciones, sí que han surgido propuestas híbridas que tienden a combinar funciones y restricciones dentro de un enfoque multiparadigmático (en vez de considerar simplemente un enfoque funcional) que de alguna manera guarda un componente lógico. Estas propuestas son analizadas en las sección siguiente.

6. Programación lógico-funcional con restricciones

Como ya hemos comentado, recientemente ha surgido un nuevo paradigma declarativo especialmente interesante que consiste en la programación lógico-funcional. Siguiendo el enfoque de las secciones anteriores, primero introducimos los conceptos fundamentales de este paradigma declarativo para describir posteriormente los intentos de integración con el mundo de las restricciones.

6.1. Programación lógico-funcional

La programación lógico-funcional surge como una combinación de los paradigmas lógicos y funcional en la que se pretende reunir las principales ventajas de ambos paradigmas en uno nuevo [94]. El mecanismo operacional de los lenguajes lógico-funcionales [89] es el resultado de combinar los que utilizan los lenguajes lógicos (i.e., unificación y resolución) y los funcionales (i.e., básicamente reescritura). En este sentido existen diversas propuestas aunque una de los mecanismos más estudiados y extendidos es la "reescritura con unificación" o, más comúnmente denominada *estrechamiento* (i.e., narrowing). El origen de este método lo hallamos en [150]. No todos los lenguajes lógico-funcionales se basan en este método pues algunos como *ESCHER* [128] se basan básicamente en un sistema de reescritura, sin embargo los dos principales representantes de los lenguajes lógico-funcionales, Curry [95] y $\mathcal{TOY}$ [37, 87, 130], se basan en técnicas de estrechamiento.

El *narrowing* presenta un alto grado de indeterminismo debido a la elección de la expresión a reducir en un paso de cómputo y de la regla de reescritura a utilizar, lo que supone que el espacio de búsqueda generado puede ser muy grande. Para reducir este espacio se utilizan *estrategias de estrechamiento* con el fin de conseguir cómputos más deterministas, y en consecuencia, más eficientes. En particular, es posible guiar la evaluación de una función estudiando la demanda de patrones de las reglas que la definen, y en este sentido surge la *Estrategia Guiada por la Demanda* [129]. Esta estrategia responde a la idea del *estrechamiento perezoso* [150] y su funcionamiento consiste en retrasar la evaluación de los argumentos de la función de llamada, mientras no sean necesarios para continuar el cómputo.

6.2. Restricciones en este marco

La *programación lógico-funcional con restricciones* (CFLP, en inglés *Constraint Functional Logic Programming*) nació a partir del deseo de integrar restricciones en los lenguajes del paradigma lógico-funcional. Como ya se ha comentado,

este paradigma, a pesar de poseer características funcionales, también posee un componente lógico similar al de los lenguajes de programación lógica por lo que las restricciones se adaptan más naturalmente que en los lenguajes puramente funcionales.

El objetivo pretendido con la integración es similar al buscado en el paradigma LP, o sea, combinar la expresividad de los lenguajes lógico-funcionales con la eficiencia que ofrece el paradigma de la programación con restricciones. En definitiva lo que se busca es ampliar el abanico de aplicaciones prácticas que un lenguaje lógico-funcional pueda resolver, pues la integración de resolutores de satisfacción de restricciones en dominios concretos puede minimizar en parte la ineficiencia innata inherente a los lenguajes de programación declarativos.

Hasta la fecha se han incluido, con mayor o menor éxito, restricciones de dominio finito y restricciones reales [15, 67, 68, 132]. En particular, la resolución de restricciones sobre el dominio real está convirtiéndose en una cuestión realmente importante en el contexto de los lenguajes lógico-funcionales. Ya en [15], se presenta un lenguaje declarativo, denominado CFLP($\Re$), que combina aspectos de la programación funcional perezosa con la programación lógica y la resolución de restricciones sobre el dominio real. El mecanismo de ejecución del lenguaje consiste en una combinación de evaluación perezosa y resolución de restricciones. La principal desventaja de este lenguaje radica en su implementación pues el método consiste en traducir los programas CFLP($\Re$) a programas de un lenguaje lógico que da soporte a un mecanismo de resolución de restricciones aritméticas reales. También, más recientemente, en [96] se propone la integración de las llamadas reglas de manejo de restricciones (CHRs; ver la sección 7) en el lenguaje lógico-funcional Curry, aunque esta integración está todavía en una fase que debe madurar.

7. Reglas de manejo de restricciones

Al hablar de programación declarativa con restricciones, no podemos olvidar en este artículo una mención importante a las denominadas *reglas de manejo de restricciones* (CHR, *Constraint handling rules*) [74, 75, 76] que son una propuesta para dotar de mayor flexibilidad a los sistemas de restricciones. En concreto, las CHRs representan una extensión de los lenguajes declarativos especialmente diseñada para escribir restricciones definidas por el usuario, y constituyen esencialmente un lenguaje de elección-comprometida (i.e., 'committed-choice language') consistente en reglas con guardas con múltiples cabezas donde las restricciones se reescriben es otras más simples hasta que se alcanza un forma resuelta. Un lenguaje de CHR permite "múltiples cabezas", i.e., conjunciones de restricciones en la cabeza de una regla, lo que representa una característica fundamental para resolver la conjunción de las restricciones puesto que, si las reglas tuviesen una cabeza simple, la insatisfiabilidad de las restricciones no podría ser probada (e.g., $X < Y, Y < X$) y la satisfacción de restricciones en general no podría ser alcanzada.

Como lenguaje de propósito especial, las CHRs extienden un lenguaje (normalmente lógico) huésped con una capacidad para la resolución de restricciones. Los cálculos auxiliares en los programas con CHRs son directamente ejecutados como sentencias del lenguaje huésped. En esta sección no entraremos en detalles del lenguaje huésped y nos ceñiremos exclusivamente a las CHRs.

Una restricción en un lenguaje lógico se considera que es un predicado especial de primer orden (fórmula atómica). Ahora hay que considerar dos clases disjuntas de símbolos de predicado: una clase para las restricciones predefinidas y otra para las restricciones definidas por el usuario (CHRs). Las primeras son manejadas por el resolutor de restricciones predefinidas que ya existe en el lenguaje huésped (o fuente), normalmente lógico. Las segundas, o sea, las restricciones CHRs, son aquellas formuladas en un programa CHR que consiste simplemente en un conjunto finito de CHRs. Puesto que las sentencias que aparecen del lenguaje huésped suelen ser declarativas, podemos considerarlas como restricciones predefinidas (dentro de un resolutor incompleto que es el propio lenguaje huésped).

A continuación presentamos una visión general sobre la sintaxis de las CHRs. Para obtener información más profunda sobre las semánticas puede consultarse [1, 2].

Tradicionalmente existen tres clases de CHRs (aunque la tercera es en realidad una combinación de las dos primeras):

• La CHR de *Simplificación* que tiene la forma

$$label \ @ \ H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k;$$

• La CHR de *Propagación* tiene la forma siguiente

$$label \ @ \ H_1, \ldots, H_i \Rightarrow G_1, \ldots, G_j \ \mid B_1, \ldots, B_k;$$

• y la CHR llamada de *Simpagación* (i.e., Simplificación + Propagación) que tiene la forma

$$label \ @ \ H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Leftrightarrow \ G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

donde $i > 0, j \geq 0, k \geq 0, l > 0$. El componente, que llamaremos, *multi-cabeza*, $H_1, \ldots, H_i$ es una lista no vacía de CHRs, la guarda $G_1, \ldots, G_j$ es una secuencia de restricciones predefinidas y el cuerpo $B_1, \ldots, B_k$ es una secuencia de restricciones predefinidas y de CHRs. *label* es simplemente un identificador que etiqueta la regla. La regla de *Simplificación* reemplaza las restricciones $H_1, \ldots, H_i$ por las restricciones más simples $B_1, \ldots, B_k$, siempre que la conjunción de las guardas $G_1, \ldots, G_j$ pueda ser demostrada como cierta. La regla de *Propagación* añade al almacén de restricciones nuevas restricciones $B_1, \ldots, B_k$ las cuales son lógicamente redundantes con respecto a las restricciones $H_1, \ldots, H_i$ pero que, aún así, pueden causar una posterior simplificación (de nuevo siempre que la conjunción de las guardas $G_1, \ldots, G_j$ pueda ser demostrada como cierta). La regla de la *Simpagación* es una abreviación de la regla de simplificación

$$label \ @ \ H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \Leftrightarrow \\ G_1, \ldots, G_j \ \mid B_1, \ldots, B_k, H_1, \ldots, H_l.$$

EJEMPLO 6 *Considera la restricción $X \geq Y$. Entonces, las siguientes CHRs definen una restricción de orden parcial $\geq$.*

$$\textit{reflexividad} \ @ \ X \geq Y \Leftrightarrow X = Y \mid \textit{true}.$$
$$\textit{antisimetría} \ @ \ X \geq Y, \ Y \geq X \Leftrightarrow X = Y.$$
$$\textit{transitividad} \ @ \ X \geq Y, Y \geq Z \Rightarrow X \geq Z.$$

'true' representa la secuencia vacía de CHRs. Estas reglas definen el mecanismo de propagación de la restricción $\geq /2$. Por ejemplo, la regla reflexividad *declara que si el almacén implica $X = Y$ entonces una restricción tal como $X \geq Y$ puede ser simplificada a 'true' (y por lo tanto puede ser eliminada del almacén). La regla* antisimetría *declara que si el almacén, implica dos restricciones $X \geq Y$ y $Y \geq X$, entonces ambas pueden ser reemplazadas por una restricción más simple tal como $X = Y$. Por último, la regla* transitividad *añade la restricción $X \geq Z$ al almacén siempre que coexistan en éste dos restricciones de la forma $X \geq Y$ e $Y \geq Z$.*

La aplicación repetida de las CHRs simplifica y, posiblemente, resuelve las restricciones definidas por el usuario.

Las CHRs han sido ya integradas en un gran número de lenguajes de CLP como por ejemplo SICStus Prolog, SWI Prolog, ECL^iPS^e Prolog, HAL, XSB Prolog, YAP Prolog, etc, y han sido utilizadas para definir una amplia variedad de resolutores de restricción aplicados sobre dominios de cómputo diversos como el dominio finito, el dominio conjunto y otros dominios novedosos tales como los "features trees" y dominios adecuados para el razonamiento terminológico y temporal [75]. Debido a su gran flexibilidad, las CHRs han sido también muy usadas para modelar aplicaciones del mundo real [79, 80, 77, 78]. A pesar de la flexibilidad para el modelado de CSPs, existe una gran desventaja al usar CHRs: su ineficiencia tal y como quedo demostrado en [66]. No obstante, desde la realización de la citada comparativa, las CHRs han evolucionado mucho y su eficiencia está directamente relacionada con la del sistema sobre las cuales estén implementadas.

8. Programación declarativa multiparadigma con restricciones

En general la programación multiparadigmática se basa en la idea de combinar varios paradigmas en un mismo marco de programación. Con respecto a la programación declarativa con restricciones a continuación mostramos varias propuestas existentes en la literatura que consideran además otros paradigmas.

8.1. Marco concurrente

La *programación concurrente con restricciones* (CCP, *concurrent constraint programming*) se basa en la comunicación asíncrona entre "agentes" mediante el proceso de "implicación de restricciones" (i.e., constraint entailment). Una

restricción definida por el usuario es considerada como un *proceso*, y un estado está directamente relacionado con una red de procesos enlazados a través de variables compartidas mediante el almacén de restricciones. Los procesos pueden comunicarse, mediante la adición de restricciones al almacén de restricciones, y sincronizarse, a través de esperas que son agregadas mientras se cumpla alguna condición de retraso relativa al contenido del almacén.

En [134], los lenguajes lógicos concurrentes [159] fueron generalizados para incluir restricciones; la idea consistió en considerar el operador básico de sincronización usado en estos lenguajes como un proceso de implicación de restricciones. Posteriormente, en 1989, [153] describió un modelo teórico muy elegante para los lenguajes concurrentes con restricciones, y acuñó el termino de "programación concurrente con restricciones". A partir de entonces, se ha producido un considerable progreso sobre diferentes aspectos de la CCP [154, 155].

En particular, los lenguajes lógicos concurrentes con restricciones (i.e., lenguajes de CCL) posibilitan que los procesos puedan interaccionar los unos con los otros; la comunicación y la sincronización se realiza mediante la inserción y el chequeo de restricciones. Los lenguajes de CCL más influyentes están principalmente basados en dos condiciones de retraso que tradicionalmente son llamadas *ask* y *tell*. Estas condiciones fueron definidas en [152, 153]. Una condición de retraso *ask* tiene la forma $ask(C)$ y se activa cuando el almacén de restricciones implica la restricción C. Por ejemplo, *listavacía(Y)* es equivalente a $ask(Y = [])$. Más aún, las condiciones *asks* pueden incluir variables locales mediante el uso de cuantificadores existenciales. Por ejemplo, la condición *listanovacía(Y)* es equivalente a $ask(\exists X \exists L.Y = [X \mid L])$ puesto que ésta se activa siempre que existan valores para las variables X y L tales que el almacén de restricciones implique $Y = [X \mid L]$. La otra condición de retraso es la condición *tell* la cual tiene la forma $tell(C)$ y se activa si la restricción C es consistente con el almacén de restricciones.

La discrepancia más importante entre los lenguajes de CCP y CCL se encuentra en la forma en la que manejan (i.e., resuelven) las múltiples reglas que definen el mismo predicado. En los lenguajes de CCL, se hace un intento por cada regla hasta que se encuentra una respuesta i.e., se emplea el conocido determinismo "don't know". En los lenguajes de CCP, el mecanismo de evaluación retrasa la selección de la regla a usar hasta que al menos una de las guardas se activa[4]. Si existen más de una regla para seleccionar, entonces se elige una arbitrariamente. Independientemente de lo que ocurra en el futuro, nunca se realiza backtracking de manera que el programador tiene la responsabilidad de dotar a cada regla con una guarda adecuada que asegure que al menos una vez se active. Por lo tanto los lenguajes de CCP proporcionan una especie de no determinismo "don't care" (i.e., si hay más de una guarda que se pueda habilitar entonces no importa ni preocupa cual de las reglas correspondientes será seleccionada pues cualquiera de ellas será correcta).

Otro enfoque alternativo fue la idea presentada inicialmente en [164] y posteriormente implementada para producir el lenguaje *Oz* [177], el cual hoy día ha evolucionado al lenguaje Mozart [178]. Este lenguaje combina características de los lenguajes de CLP, lenguajes funcionales y lenguajes concurrentes. Con respecto a la programación con restricciones, la búsqueda es implementada de una forma bastante diferente a como se había hecho en los lenguajes de CLP, puesto que la búsqueda es programable. Además, en vez de seguir el típico enfoque de primero en profundidad y de izquierda a derecha, las estrategias de búsqueda en el lenguaje *Oz* están codificadas en los llamados *procedimientos de búsqueda* con los que se explora el espacio de posibles soluciones (i.e., el espacio de búsqueda). Además, el cómputo puede ser suspendido o retrasado con respecto a las elecciones a realizar en el procedimiento de exploración, hasta que el programador especifique explícitamente un procedimiento de búsqueda. En general *Oz* integra los paradigmas de CLP y el de la programación concurrente suministrando un enfoque más flexible a la programación con restricciones.

[53] proporciona un "survey" bastante completo que muestra los principales motivos para extender el marco concurrente al paradigma de (C)LP, y analiza además el paradigma de CCP con sus fundamentos semánticos básicos.

8.2. Orientación a objetos.

Otro enfoque declarativo multiparadigmático con restricciones es el ofrecido en el lenguaje LIFE [9]

[4]Una guarda consiste básicamente en una condición *tell* más una condición *ask*.

que es un lenguaje experimental que propone integrar la programación lógica, la programación funcional, la programación orientada a objetos y un mecanismo de resolución sobre un dominio ordenado de árboles de *features*, admitiéndose restricciones tales como la igualdad (i.e., la unificación) y el "entailment"(i.e., matching) sobre términos "features". Es el precursor de otros lenguajes tales como LOGIN [8] y Le Fun [7].

9. Paradigmas relacionados

El éxito real de los lenguajes declarativos con restricciones (especialmente CLP) ha hecho que el concepto de restricción (y su resolución) se integre en otros paradigmas. Los conceptos originales de CLP fueron ajustados para dar un mejor servicio en diferentes áreas de aplicación.

Éste es el caso de la llamada *programación con restricciones imperativa* que básicamente busca integrar el concepto de restricción en un entorno imperativo de programación. En particular, algunos lenguajes orientados a objetos permiten que el programador especifique algunas restricciones y, por lo tanto, formulen de una manera concisa la relación entre variables diferentes. Esta integración garantiza la eficiencia de la resolución del programa, como quedó demostrado en [70].

Además, otros lenguajes orientados a objetos ofrecen un mecanismo de satisfacción incremental de restricciones muy similar al proporcionado por los lenguajes de CLP. En realidad esto quiere decir que las restricciones son integradas mediante resolutores preconstruidos específicos en un lenguaje imperativo que actúa como el huésped [108]. Todavía queda mucho por estudiar en lo relativo a esta integración de las restricciones en los lenguajes imperativos.

10. Comentarios finales

El objetivo de este articulo es el de ofrecer una visión general sobre el "estado-del-arte" con respecto a la integración de restricciones en la programación declarativa, considerando los principales paradigmas declarativos, tales como el lógico, el funcional y el lógico-funcional, y mostrando otras propuestas novedosas como por ejemplo las reglas de manejo de restricciones y los enfoques multiparadigmáticos. Hemos enfatizado especialmente el esquema de la programación lógica con restricciones (CLP), precisamente por ser éste el campo declarativo de investigación más activo en cuanto a la integración de restricciones.

Además, hemos hecho hincapié en las diferentes instancias del esquema de CLP, instancias generadas por los diferentes dominios de cómputo sobre el cual las restricciones son satisfechas. Las principales instancias de este esquema han sido descritas y se han mostrado las líneas de investigación fundamentales realizadas sobre cada una de ellas en los últimos años.

Otro de los propósitos de este artículo es el de resaltar la naturaleza multidisciplinar del paradigma de la programación con restricciones, el cual ha sido aplicado en áreas muy diferentes tales como la inteligencia artificial, la investigación operativa, las matemáticas, la computación simbólica y otras muchas entre las cuales se encuentra la programación declarativa.

Agradecimientos

El autor agradece los valiosos comentarios de los revisores anónimos que han permitido mejorar la presentación de este artículo. Este trabajo ha sido subvencionado parcialmente por los proyectos TIC2002-04498-C05-02 y TIN2004-7943-C04-01 financiados por el MCyT y FEDER.

Referencias

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, number 1330 in LNCS, pages 252–266, Linz, Austria, 1997. Springer-Verlag.

[2] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.

[3] Abelson et al. Revised report on the algorithmic language Scheme. *Symbolic Computation*, 11(1):5–105, August 1998.

[4] A. Aggoun, D. Chan, P. Dufresne, E. Falvey, H. Grant, A. Herold, G. Macart-

ney, M. Meier, D. Miller, S. Mudambi, B. Perez, E. Van Rossum, J. Schimpf, Periklis, A. Tsahageas, and D.H. de Villeneuve. ECL^iPS^e 3.5, user manual. European Computer -Industry Research Centre (ECRC). Munich, December 1995.

[5] A. Aiba and K. Sakai. CAL: a theoretical background of constraint logic programming and its applications. *Journal of Symbolic Computation*, 8:589–603, 1989.

[6] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. The constraint logic programming language CAL. In ICOT, editor, *International Conference on Fith Generation Computer Systems (FGCS'88)*, pages 263–276, Tokyo, Japan, 1988. Ohmsha Ltd. and Springer-Verlag.

[7] H. Aït-kaci, P. Lincoln, and R.Ñasr. Le Fun: logic, equations and functions. In *1987 Symposium on Logic Programming (SLP'87)*, pages 17–23, San Francisco, California, 1987. IEEE-CS.

[8] H. Aït-kaci and R.Ñasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[9] H. Aït-kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993. A preliminary version appeared in [136], pp:255-274.

[10] H. Aït-kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, 1994.

[11] G. Alefeld and J. Herzberger. *Introduction to interval computations*. Academic Press, London and San Diego, 1983.

[12] C. Ansoategui and F. Manyà. Una introducción a los algoritmos de satisfiabilidad. *Inteligencia Artificial. Revista Iberoamericana de IA*, (20):43–55, 2003.

[13] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[14] K.R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[15] P. Arenas, M.T. Hortalá, F.J. López-Fraguas, and E. Ullán. Real constraints within a functional logic language. In P. Lucio, M. Martelli, and M.Ñavarro, editors, *Joint Conference on Declarative Programming (APPIA-GULP-PRODE'96)*, Donostia-San Sebastian, Spain, July 1996.

[16] D.S. Arnon, G.E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: the basic algorithm. *SIAM Journal on Computing*, 13(4):865–877, 1984.

[17] F. Azevedo and P. Barahona. Modelling digital circuits problems with set constraints. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís-Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *1st International Conference on Computational Logic (CL2000)*, number 1861 in LNCS, pages 414–428, London, UK, July 2000. Springer-Verlag.

[18] R. Bagnara, R. Gori, P.M. Hill, and E. Zaffanella. Finite-tree analysis for constraint logic-based languages. In P. Cousot, editor, *8th International Symposium on Static Analysis (SAS'01)*, volume 2126 of *LNCS*, pages 165–184, Paris, France, 2001. Springer-Verlag, Berlin.

[19] H.P. Barendregt. *The lambda calculus - Its syntax and semantics*. Studies in Logic and the Foundations of Mathematics, 103. North-Holland, Netherlands, 1984. Second Revised Edition.

[20] H.P. Barendregt. Functional programming and lambda calculus. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 323–363, Netherlands, 1990. Elsevier. J. van Leeuwen editor.

[21] P. Barth. Short guide to CLP($\mathcal{PB}$). Available at ftp://www.mpi-sb.mpg.de/pub/tools/CLPPB/clppb.html, 1994.

[22] P. Barth. *Logic-based 0-1 constraint Programming*. Operations Research/Computer Science Interfaces. Kluwer, 1996.

[23] P. Barth and A. Bockmayr. Modelling 0-1 problems in CLP($\mathcal{PB}$). In M. Wallace, editor, *2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, pages 1–9, London, UK, 1996. Prolog Management Group.

[24] Bell Northern Research, Ottawa, Ontario, Canada. *CLP(BNR) reference and users manuals*, 1988.

[25] F. Benhamou. Boolean algorithms in Prolog III. In *[26]*, pages 307–325, Cambridge, Massachusetts, London, England, 1993. The MIT Press.

[26] F. Benhamou and A. Colmerauer, editors. *Constraint logic programming: selected research*. The MIT Press, Cambridge, MA, 1993.

[27] F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising hull and box consistency. In D. De Schreye, editor, *16th International Conference on Logic Programming (ICLP'99)*, pages 230–244, Las Cruces, New Mexico, USA, November-December 1999. The MIT Press.

[28] F. Benhamou, D.A. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In M. Bruynooghe, editor, *4th International Symposium on Logic Programming (ILPS'94)*, Logic Programming, pages 124–138, Ithaca, New York, November 1994. The MIT Press.

[29] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *The Journal of Logic Programming*, 32(1):1–24, July 1997.

[30] A. Biasizzo and F.Ñovak. Model-based diagnosis of analog circuits. In *International Mixed Signal Testing Workshop*, pages 95–100, Grenoble, France, 1995.

[31] R. Bird, T.E. Scruggs, and M-A. Mastropieri. *Introduction to Functional Programming*. Prentice Hall, Englewood Cliffs, N.J., 2000.

[32] A. Bockmayr. Logic programming with pseudo-Boolean constraints. In *[26]*, pages 327–350, Cambridge, MA, 1993. The MIT Press.

[33] A. Bockmayr. Solving pseudo-Boolean constraints. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 22–38, Châtillon-sur-Seine, France, 1994. Springer-Verlag.

[34] B. Buchberger. Applications of Gröbner bases in non-linear computational geometry. In *Trends in Computer Algebra*, number 296 in LNCS. Springer-Verlag, 1987.

[35] B. Buchberger. Gröbner bases: an introduction. In W. Kuich, editor, *19th International Colloquium on Automata, Languages and Programming (ICALP'92)*, number 623 in LNCS, pages 378–379, Vienna, Austria, July 1997. Springer-Verlag.

[36] W. Büttner and H. Simonis. Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(2):191–205, 1997.

[37] R. Caballero, F.J. López-Fraguas, and J. Sánchez. User's manual for $\mathcal{TOY}$. Technical report SIP-5797, Universidad Complutense de Madrid, Dpto. Lenguajes, Sistemas Informáticos y Programación, 1997.

[38] M. Carlsson and M. Brindal. Automatic frequency assignment for cellular telephones using constraint satisfaction techniques. In D.S. Warren, editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 647–665, Budapest, Hungary, 1993. The MIT Press.

[39] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In U. Montanari and F. Rossi, editors, *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, number 1292 in LNCS, pages 191–206, Southampton, UK, 1997. Springer-Verlag.

[40] M. Carro, M. Hermenegildo, F. Bueno, D. Cabeza, M.J. García, P. López, and G. Puebla. An introductory course of constraint logic programming. Computer Science School, Technical University of Madrid, UPM, 2000.

[41] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[42] P. Codognet and D. Diaz. Boolean constraint solving using clp(FD). In D. Miller, editor, *1993 International Symposium on Logic Programming (ILPS'93)*, pages 525–539, Vancouver, British Columbia, Canada, October 1993. The MIT Press.

[43] P. Codognet and D. Diaz. clp(B): combining simplicity and efficiency in Boolean constraint solving. In *6th International Symposium on Programming Languages Implementation and Logic Programming (PLILP'94)*, number 844 in LNCS, pages 244–260, Madrid, Spain, 1994. Springer-Verlag.

[44] P. Codognet and D. Diaz. Compiling constraints in clp(FD). *The Journal of Logic Programming*, 27(3):185–226, 1996.

[45] J. Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, January 1988.

[46] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[47] A. Colmerauer. An introduction to PROLOG III. *Communications of the ACM*, 33(7):69–90, July 1990.

[48] A. Colmerauer and P. Roussel. The birth of Prolog. *ACM SIGPLAN Notices as part of 2nd ACM SIGPLAN Conference on History of Programming Languages*, 28(3):37–52, March 1993. Cambridge, United States.

[49] J. Crossley, L. Mandel, and M. Wirsing. First-order constrained lambda calculus. In F. Baader and K. U. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (Frocos'96)*, volume 3 of *Applied Logic Series*, pages 339–356, Munich, Germany, March 1996. Kluwer Academic Publishers.

[50] J. Csontó and J. Paralič. A look at CLP: theory and application. *Applied Artificial Intelligence*, 11:59–69, 1997.

[51] S. Curtis, B. Smith, and A. Wren. Constructing driver schedules using iterative repair. In *2nd International Conference on The Practical Applications of Constraint Technology and Logic Programming (PACLP'2000)*, pages 59–78, Manchester, UK, April 2000. Practical Application Company.

[52] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, 1987.

[53] F.S. de Boer and C. Palamidessi. From concurrent logic programming to concurrent constraint programming. In *[126]*, pages 55–113. Oxford University Press, 1994.

[54] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[55] B. Demoen, M.G. de la Banda, W. Harvey, K. Marriott, and P. Stuckey. Herbrand constraint solving in HAL. Technical Report 1999/49, Monash University, Melbourne, 1999.

[56] B. Demoen, M.G. de la Banda, W. Harvey, K. Marriott, and P. Stuckey. Herbrand constraint solving in HAL. In D. De Schreye, editor, *16th International Conference on Logic Programming (ICLP'99)*, pages 260–274, Las Cruces, New Mexico, USA, November-December 1999. The MIT Press.

[57] D. Diaz. *Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système clp(FD)*. PhD thesis, l'université d'Orléans, 1995.

[58] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In D.S. Warren, editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

[59] D. Diaz and P. Codognet. GNU Prolog: beyond compiling Prolog to C. In E. Pontelli and V.Santos Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in LNCS, pages 81–92, Boston, USA, 2000. Springer-Verlag.

[60] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In R.A. Kowalski and K.A. Bowen, editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 42–58, Seattle, Washington, August 1988. The MIT Press.

[61] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1):75–93, January-March 1990.

[62] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In ICOT, editor, *International Conference on Fith Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, November-December 1988. Ohmsha Ltd. and Springer-Verlag.

[63] A. Dovier, E.G. Omodeo, E. Pontelli, and G. Rossi. {log}: a language for programming in logic with finite sets. *Journal of Logic Programming*, 28(1):1–44, July 1996.

[64] S.P. Peyton-Jones (editor). *Haskell 98 Language and Libraries: The revised report*. Cambridge University Press, 2003.

[65] F. Barber (editor). Special issue: Constraint satisfaction problems. *Inteligencia Artificial. Revista Iberoamericana de IA*, (20), 2003.

[66] A. J. Fernández and P.M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints*, 5(3):275–301, 2000.

[67] A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In P. Wadler and V. Dahl, editors, *5th International Symposium on Practical Aspects of Declarative Languages (PADL'2003)*, number 2562 in LNCS, pages 320–338, New Orleans, Louisiana, USA, 2003. Springer-Verlag.

[68] A. J. Fernández, M. T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado Vírseda. Constraint functional logic programming over finite domains. *Theory and Practice of Logic Programming*, 2005. Accepted for publication. Available on-line in http://arXiv.org/abs/cs/0601071.

[69] R.E. Fikes. *A heuristic program for solving problems states as non-deterministic procedures*. PhD thesis, Comput. Sci. Dept., Carnegie-Mellon University, Pittsburgh, PA, 1968.

[70] B. Freeman-Benson and A. Borning. Integrating constraints with an object-oriented language. In O. Lehrmann Madsen, editor, *European Conference on Object-Oriented Programming (ECOOP'92)*, number 615 in LNCS, pages 268–286, Utrecht, The Netherlands, 1992. Springer-Verlag.

[71] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(1):958–966, 1978.

[72] E.C. Freuder and P. Hubbe. Extracting constraint satisfaction subproblems. In *14th International Joint Conference on Artificial Intelligent (IJCAI'95)*, pages 548–557, Québec, Canada, August 1995. Morgan Kaufman.

[73] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(21-70):21–70, 1992.

[74] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 90–107, Châtillon-sur-Seine, France, 1994. Springer-Verlag.

[75] T. Frühwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37:95–138, 1998.

[76] T. Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In G.Ñadathur, editor, *International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 117–133, Paris, France, 1999. Springer-Verlag.

[77] T. Frühwirth and S. Abdennadher. The Munich rent advisor: a success for logic programming on the internet. *Theory and Practice of Logic Programming*, 1(3):303–319, January 2001.

[78] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Cognitive Technologies Series. Springer, 2003.

[79] T. Frühwirth and P. Brisset. Optimal planning of digital cordless telecommunication systems. In M. Wallace, editor, *3rd International Conference on the Practical Application of Constraint Technology (PACT'97)*, pages 165–176, London, UK, 1997. Prolog Management Group.

[80] T. Frühwirth and P. Brisset. Optimal placement of base stations in wireless indoor telecommunication. In M. Maher and J-F. Puget, editors, *4th International Conference on Principles and Practice of Constraint Programming (CP'98)*, number 1520 in LNCS, pages 476–480, Pisa, Italy, October 1998. Springer-Verlag.

[81] T. Frühwirth, A. Herold, V. Kuechenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming - an informal introduction. In G. Comyn, M. Ratcliffe, and N. Fuchs, editors, *2nd International Logic Programming Summer School (LPSS'92): Logic programming in Action*, number 636 in LNAI, Zurich, Switzerland, 1993. Springer-Verlag.

[82] R.P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge Massachusetts, 1985.

[83] J. Gaschnig. A constraint satisfaction method for inference making. In *12th Annual Allerton Conference on Circuit System Theory*, pages 866–874, Illinois University, 1974.

[84] C. Gervet. Conjunto: constraint logic programming with finite set domains. In M. Bruynooghe, editor, *1994 International Symposium on Logic programming (SLP'94)*, pages 339–358, Ithaca, New York, November 1994. The MIT Press.

[85] C. Gervet. Interval propagation to reason about sets: definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[86] D. Gilbert, M. Schroeder, and J. van Helden. Interactive visualisation and exploration of biological data. In G. Levi and M. Martelli, editors, *5th Joint Conference on Information Sciences (Stream on Biomolecular Informatics)*, Atlantic City, New Jersey, USA, 2000.

[87] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *The Journal of Logic Programming*, 40(1):47–87, July 1999.

[88] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In E. Börger, G. Jäger, H.K. Büning, S. Martini, and M.M. Richter, editors, *6th workshop in Computer Science Logic, CSL'92*, number 702 in LNCS, pages 216–230, San Miniato, Italy, September 1992. Springer.

[89] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *14th International Conference on Logic Programming (ICLP'97)*, pages 153–167. The MIT Press, 1997.

[90] M.M. Gorlick, C.F. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a logical methodology for testing. *Journal of Logic Programming*, 8(1):95–119, 1990.

[91] F. Goualard, F. Benhamou, and L. Granvilliers. An extension of the WAM for hybrid interval solvers. *The Journal of Functional and Logic Programming*, 1999(1):1–36, April 1999. Special issue of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages.

[92] E. Hansen. *Global optimization using interval analysis*. Marcel Dekker, 1992.

[93] M. Hanus. Analysis of nonlinear constraints in CLP($\Re$). In D.S. Warren, editor, *10th International Conference on Logic Programming (ICLP'93)*, pages 83–99, Budapest, Hungary, 1993. The MIT Press.

[94] M. Hanus. The integration of functions into logic programming: A survey. *The Journal of Logic Programming*, 19-20:583–628, 1994. Special issue: Ten Years of Logic Programming.

[95] M. Hanus. Curry: a truly integrated functional logic language. http://www.informatik.uni-kiel.de/∼curry/, 1999.

[96] M. Hanus. Adding constraint handling rules to curry. In *20th Workshop on Logic Programming (WLP 2006)*, Vienna, Austria, 2006.

[97] N. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9(2):231–260, 1992.

[98] N. Heintze, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. On meta-programming in CLP($\Re$). In E.L. Lusk and R.A. Overbeek, editors, *the 1989 North American Conference on Logic Programming (NACLP'89)*, pages 52–66, Cleveland, Ohio, October 1989. The MIT Press.

[99] M. Henz and T. Müller. An overview of finite domain constraint programming. In *5th Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore, 2000.

[100] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M.G. de la Banda, P. López-García, and G. Puebla. The Ciao logic programming environment. In J.W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagiv, and P.J. Stuckey, editors, *1st International Conference on Computational Logic (CL'2000)*,

number 1861 in LNCS, London, UK, July 2000. Springer-Verlag. Tutorial.

[101] T. Hickey, Q. Ju, and M.H. van Emden. Interval arithmetic: from principles to implementation. CS Technical report CS-99-202, Brandeis University, July 1999.

[102] T.J. Hickey. CLIP: a CLP(Intervals) dialect for metalevel constraint solving. In E. Pontelli and V.Santos Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, number 1753 in LNCS, pages 200–214, Boston, USA, 2000. Springer-Verlag.

[103] C. Holzbaur. OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[104] H. Hong. RISC-CLP(Real): logic programming with non-linear constraint over reals. In *[26]*, pages 133–159, Cambridge, MA, 1993. The MIT Press.

[105] T. Huynh and C. Lassez. A CLP($\Re$) options trading analysis system. In R.A. Kowalski and K.A. Bowen, editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 59–69, Seattle, Washington, August 1988. The MIT Press.

[106] E. Hyvönen. Constraint reasoning based on interval arithmetic. In N. S. Sridharan, editor, *11th International Joint Conference on Artificial Intelligent (IJCAI'89)*, pages 1193–1198, Detroit, MI, USA, August 1989. Morgan Kaufman.

[107] If/Prolog. *IF/Prolog V5.0A, constraints package*. Siemens Nixdorf Informationssysteme AG, Munich, Germany, 1994.

[108] Ilog SOLVER, reference manual, version 3.1, 1995.

[109] J. Jaffar and J.L. Lassez. Constraint logic programming. In *14th ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[110] J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994.

[111] J. Jaffar, M. Maher, K. Marriot, and P. Stuckey. The semantics of constraint logic programs. *The Journal of Logic Programming*, 37:1–46, 1998.

[112] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *4th International Conference on Logic Programming (ICLP'87)*, pages 196–218, Melbourne, Australia, 1987. The MIT Press.

[113] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\Re$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[114] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. An abstract machine for CLP($\Re$). *SIGPLAN Notices*, 27(7):128–139, July 1992. Publication of the Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI'92), San Francisco, California.

[115] J. Jaffar, S. Michaylov, and R.H.C. Yap. A methodology for managing hard constraints in CLP systems. *SIGPLAN Notices*, 26(6):306–316, June 1991. Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91), Toronto, Ontario, Canada.

[116] A. King. Pair-sharing over rational trees. *Journal of Logic Programming*, 46(1-2):139–155, November 2000.

[117] J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Evolutionary training of CLP-constrained neural networks. In M. Wallace, editor, *2nd International Conference on the Practical Application of Constraint Technology (PACT'96)*, pages 129–142, London, UK, 1996. Publisher Prolog Management Group.

[118] G. Kondrak and P. Van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89(1-2):365–387, January 1997.

[119] D. Kozen. Set constraints and logic programming. In J-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics (CCL'94)*, number 845 in LNCS, pages 302–303, Munich, Germany, September 1994. Springer-Verlag. Extended version published in [120].

[120] D. Kozen. Set constraints and logic programming. *Information and Computation*, 142(1):2–25, 1998.

[121] G. Lapalme and F. Rabhi. *Algorithms: A Functional Programming Approach*. Reading. Addison-Wesley, Massachusetts, 1999.

[122] Javier Larrosa and Pedro Meseguer. Algoritmos para satisfacción de restricciones. *Inteligencia Artificial. Revista Iberoamericana de IA*, (20):31–42, 2003.

[123] C. Lassez. Constraint logic programming: a tutorial. *BYTE magazine*, pages 171–176, August 1987.

[124] J.H.M. Lee and M.H. van Emden. Interval computation as deduction in CHIP. *The Journal of Logic Programming, Special Issue:Constraint Logic Programming*, 16(3-4):255–276, 1993.

[125] B. Legeard and E. Legros. Short overview of the CLPS system. In J. Maluszynski and M. Wirsing, editors, *3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'97)*, number 528 in LNCS, pages 431–433, Passau, Germany, August 1991. Springer-Verlag.

[126] G. Levi, editor. *Advances in logic programming theory*, volume 1. Oxford University Press, UK, 1994.

[127] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, Berlin, Heidelberg, 1987.

[128] J.W. Lloyd. Declarative programming in Escher. Technical report cstr-95-013, University of bristol, 1995.

[129] R. Loogen, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *International Symposium on Programming Languages Implementation and Logic Programming (PLILP'93)*, number 714 in LNCS, pages 184–200. Springer-Verlag, 1993.

[130] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In P.Ñarendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, number 1631 in LNCS, pages 244–247, Trento, Italy, 1999. Springer-Verlag.

[131] K.C. Louden. *Programming languages, Principles and Practice*. Thomson Brooks/Cole, 2003.

[132] W. Lux. Adding linear constraints over real numbers to Curry. In A. Middeldorp, H. Kuchen, and K. Ueda, editors, *5th International Symposium on Functional and Logic Programming (FLOPS'2001)*, number 2024 in LNCS, pages 185–200, Tokyo, Japan, March 2001. Springer-Verlag.

[133] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[134] M. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *4th International Conference on Logic Programming (ICLP'87)*, pages 858–876, Melbourne, Australia, 1987. The MIT Press.

[135] S. Majumdar. Application of relational interval arithmetic to computer performance analysis: a survey. *Constraints*, 2(2):215–235, October 1997.

[136] J. Maluszynski and M. Wirsing, editors. *3rd International Symposium in Programming Language Implementation and Logic Programming (PLILP'91)*, volume LNCS 528. Springer-Verlag, Passau, Germany, August 1991.

[137] K. Marriot and P. J. Stuckey. *Programming with constraints*. The MIT Press, Cambridge, Massachusetts, 1998.

[138] A. Middeldorp and S. Okui. Deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[139] R.E. Moore. *Interval analysis*. Prentice hall, Englewood Cliffs, NJ, 1966.

[140] T. Müller and M. Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13th Workshop on Logic Programming*, pages 104–115, München, 17–19 September 1997. Technische Universität.

[141] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[142] S.Ñ'Dong. Prolog IV ou la programmation par contraintes selon PrologIA. In *Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97)*, pages 235–238, Orléans, France, 1997. Edition HERMES.

[143] C. Oksaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, U.K., 1999.

[144] W. Older. Interval arithmetic specification. Technical report, Bell-Northern, Research Computing Research Laboratory, Ottawa, Ontario, Canada, 1989.

[145] W. Older and F. Benhamou. Programming in CLP(BNR). 1st International Workshop on Principles and Practice of Constraint Programming (PPCP'93), Informal Proceedings, pages: 228-238, Brown University, Newport, Rode Island, 1993.

[146] W. Older and A. Vellino. Constraint arithmetic on real intervals. In *[26]*, pages 175–195, Cambridge, MA, 1993. The MIT Press.

[147] G. Pesant and M. Boyer. QUAD-CLP($\Re$): adding the power of quadratic constraints. In A. Borning, editor, *2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, number 874 in LNCS, pages 95–108, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[148] S.P. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, N.J., 1987.

[149] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[150] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *1985 Symposium on Logic Programming (SLP'87)*, pages 138–151, Boston, Massachusetts, July 1985. IEEE-CS.

[151] Z. Ruttkay. Constraint satisfaction-a survey. *CWI Quaterly*, 11(2-3):163–214, 1998.

[152] V.A. Saraswat. A somewhat logical formulation of CLP synchronisation primitives. In R.A. Kowalski and K.A. Bowen, editors, *5th International Conference and Symposium of Logic Programming (ICLP/SLP'88)*, pages 1298–1314, Seattle, Washington, August 1988. The MIT Press.

[153] V.A. Saraswat. *Concurrent constraint programming languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, January 1989. Also published in [155].

[154] V.A. Saraswat. Concurrent constraint programming: A brief survey. Unpublished, August 1992.

[155] V.A. Saraswat. *Concurrent constraint programming*. The MIT Press, Cambridge, MA, 1993. Doctoral Dissertation Award and Logic programming Series.

[156] S. Sato and A. Aiba. An application of CAL to robotics. In *[26]*, pages 161–173, Cambridge, MA, 1993. The MIT Press.

[157] S. Sato and A. Aiba. A study on Boolean constraint solvers. In *[26]*, pages 253–267, Cambridge, MA, 1993. The MIT Press.

[158] R. Sedgewick. *Algorithms*. Series in Computer Science. Addison-Wesley, USA, 1984.

[159] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[160] Sicstus manual. *SICStus Prolog user's manual, release* 3#5. By the Intelligent Systems Laboratory, Swedish Institute of Computer Science, 1994.

[161] H. Simonis. Applications of constraint logic programming. In L. Sterling, editor, *12th International Conference on Logic Programming (ICLP'95)*, pages 9–11, Tokyo, Japan, June 1995. The MIT Press. Advanced Tutorials.

[162] H. Simonis and M. Dincbas. Using logic programming for fault diagnosis in digital circuits. In K. Morik, editor, *11th German Workshop on Artificial Intelligence (GWAI'87)*, pages 139–148, Geseke, 1987. Springer-Verlag.

[163] B.M. Smith. A tutorial on constraint programming. Research Report 95.14, University of Leeds, School of Computer Studies, England, April 1995.

[164] G. Smolka. The Oz programming model. In Jan Van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 324–343, Berlin, 1995. Springer.

[165] G. Steele. Debunking the 'expensive procedure call' myth. In *National Conference of the ACM*, pages 153–162, New York, 1977. ACM Press.

[166] L. Sterling and E. Shapiro. *The art of Prolog.* Series in Logic Programming. The MIT Press, Cambridge, MA, 1986.

[167] F. Stolzenburg. Membership-constraints and complexity in logic programming with sets. In F. Baader and K.U. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (FroCos'96)*, volume 3 of *Applied Logic*, pages 285–302, Munich, Germany, March 1996. Kluwer Academic.

[168] M.T. Swain and G.J.L. Kemp. A CLP approach to the protein side-chain placement problem. In T. Walsh, editor, *Principles and Practice of Constraint Programming (CP'01)*, number 2239 in LNCS, pages 479–493, Paphos, Cyprus, 2001. Springer-Verlag.

[169] E. Tsang. *Foundations of constraint satisfaction.* Academic Press, London and San Diego, 1993.

[170] J. Ulmann. *Elements of ML Programming.* Prentice Hall, Englewood Cliffs, N.J., ML97 edition, 1998.

[171] M.H. Van Emdem. Value constraints in the CLP scheme. *Constraints*, 2(2):163–183, October 1997.

[172] P. Van Hentenryck. Tutorial on the CHIP system and applications. In *Workshop of Constraint Logic Programming*, Rehovot, Israel, 1988. Weizmann Institute of Science.

[173] P. Van Hentenryck. *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA, 1989.

[174] P. Van Hentenryck. A gentle introduction to NUMERICA. *Artificial Intelligence*, 103(1-2):209–235, 1998.

[175] P. Van Hentenryck, L. Michel, and F. Benhamou. Newton - constraint programming over nonlinear constraints. *Science of Computer Programming*, 20(1-2):83–118, 1988.

[176] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a modeling language for global optimization.* The MIT Press, Cambridge, MA, 1997.

[177] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):717–763, November 2003.

[178] P. Van Roy and S. Haridi. *Concepts, techniques and models of computer programming.* The MIT Press, Cambridge, MA, 2004.

[179] H. Vandecasteele. *Constraint logic programming: applications and implementation.* PhD thesis, Catholic University of Leuven, 1999.

[180] C. Walinsky. CLP(Σ^*): constraint logic programming with regular sets. In G. Levi and M. Martelli, editors, *6th International Conference on Logic Programming (ICLP'89)*, pages 181–196, Lisbon, Portugal, 1989. The MIT Press.

[181] M. Wallace. Practical applications of constraint programming. *Constraints*, 1(1-2):139–168, September 1996.

[182] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, MA, November 1972.

[183] D. Waltz. Understanding line drawings in scenes shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91, UK, 1975. McGraw-Hill.

[184] H.P. Williams. *Model building in mathematical programming.* J. Wiley and Sons, New York, USA, 1993. Revised edition.

[185] H.P. Williams. *Model solving in mathematical programming.* J. Wiley and Sons, New York, USA, 1993.

[186] R.H.C. Yap. Restriction site mapping in CLP($\Re$). In Koichi Furukawa, editor, *8th International Conference on Logic Programming (ICLP'91)*, pages 521–534, Paris, France, June 1991. The MIT Press.

[187] R.H.C. Yap. A constraint logic programming framework for constructing DNA restriction maps. *Artificial Intelligence in Medicine*, 5:447–464, 1993.

[188] N-F. Zhou. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In J. Jaffar, editor, *Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 70–84, Manchester, UK, June 1996. The MIT Press.

[189] N-F. Zhou. B-Prolog user's manual (version 2.1). Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka, Japan, 1997.

[190] N-F. Zhou and I.Ñagasawa. An efficient finite-domain constraint solver in beta-Prolog. *Journal of Japanese Society for Artificial Intelligence*, 9:275–282, 1994.