UNIVERSIDAD POLITECNICA DE VALENCIA
Departamento de Sistemas Informáticos y Computación

---

# EL LENGUAJE CLP(*H/E*): UNA APROXIMACION BASADA EN RESTRICCIONES A LA INTEGRACION DE LA PROGRAMACION LOGICA Y ECUACIONAL

## TESIS DOCTORAL

Presentada por:
María Alpuente Frasnedo

Dirigida por:
Prof. Dr. Giorgio Levi
Prof. Dr. Isidro Ramos Salavert

Valencia, Noviembre de 1991.

# TESIS DOCTORAL

## EL LENGUAJE CLP(*H/E*): UNA APROXIMACION BASADA EN RESTRICCIONES A LA INTEGRACION DE LA PROGRAMACION LOGICA Y ECUACIONAL

Memoria presentada por María Alpuente Frasnedo para la obtención del grado de Doctor en Informática por la Universidad Politécnica de Valencia.

Valencia, Noviembre de 1991.

# EL LENGUAJE CLP(*H/E*): UNA APROXIMACION BASADA EN RESTRICCIONES A LA INTEGRACION DE LA PROGRAMACION LOGICA Y ECUACIONAL

Tesis Doctoral en Informática presentada por

María Alpuente Frasnedo
Licenciada en Ciencias Físicas por la Universidad de Valencia

**Directores:**

Prof. Dr. Giorgio Levi — Università di Pisa
Prof. Dr. Isidro Ramos Salavert — U. Politécnica de Valencia

**Tribunal de Lectura:**

| | | |
|---|---|---|
| Presidente: | Prof. Dr. Fernando Orejas Valdés | U. Politécnica de Cataluña |
| Vocales: | Prof. Dr. Mario Rodríguez Artalejo | U. Complutense de Madrid |
| | Prof. Dr. Vicente Hernández García | U. Politécnica de Valencia |
| | Prof. Dr. Moreno Falaschi | Università di Pisa |
| Secretario: | Prof. Dr. Juan José Moreno Navarro | U. Politécnica de Madrid |

*a José Antonio*

*"... it is exciting to realize that the research activities that started twenty years ago from very different roots, finally meet and merge."*

*B. Buchberger, 1985.*

# Indice

# Indice

# Indice

# Resumen

Uno de los problemas que, en los últimos años, han atraído con más fuerza el interés de los investigadores en el campo de la Programación Lógica es el de la integración de dos de las familias más prometedoras de lenguajes declarativos: los lenguajes lógicos y los ecuacionales. Este problema ha sido afrontado con técnicas y planteamientos muy distintos. Una de las aproximaciones de mayor relevancia define un programa lógico - ecuacional $(P, E)$ como un programa lógico positivo $P$ (cláusulas de Horn definidas) aumentado con una teoría ecuacional de Horn $E$. La ventaja de esta aproximación es que, dado que la teoría ecuacional también es un conjunto de cláusulas de Horn definidas, se cumple la conocida propiedad de intersección de modelos y, así, la teoría ecuacional genera una relación de congruencia menor o más fina sobre el universo de Herbrand asociado al programa. Tomando como dominio de interpretación el cociente del universo de Herbrand módulo esta congruencia, el programa lógico extendido con la teoría ecuacional admite una semántica de modelo mínimo y una semántica por menor punto fijo. De esta forma, las principales propiedades semánticas de los programas lógicos tradicionales se preservan en un contexto más general de programación lógico - ecuacional. Concretamente, se mantiene la existencia de un dominio de computación canónico sobre el que es posible definir varias semánticas formales que no sólo son simples y elegantes sino que, además, coinciden: semántica operacional, semántica por teoría de modelos y semántica por punto fijo. Por otro lado, el paradigma de programación lógica pura ha sido generalizado recientemente a un contexto más general de programación lógica con restricciones (C*onstraint* L*ogic* P*rogramming,* CLP). CLP es un marco general, un esquema genérico para la introducción de restricciones en programación lógica. Cada instancia CLP($\chi$) del esquema es un lenguaje de programación que surge especificando una estructura $\chi$ de computación. El esquema CLP garantiza que las propiedades semánticas de los programas lógicos convencionales son heredadas por cualquier lenguaje que pueda formalizarse como una instancia del esquema. Además, existen modelos, correspondientes a diferentes comportamientos observables de los programas lógicos, que han sido desarrollados para el esquema y que se parametrizan también para cada una de sus instancias.

El principal argumento discutido en esta tesis es que en el marco de CLP es posible formalizar la deseada integración entre programación lógica y ecuacional, admitiendo la definición de funciones y proporcionando un tratamiento adecuado de la relación de igualdad. La tesis define una instancia del esquema CLP especializada en resolver ecuaciones bajo una teoría ecuacional de Horn $E$. La estructura de computación viene dada justamente por la menor partición $H/E$ inducida por $E$ sobre el universo de Herbrand $H$ para el programa. $=$ es el único símbolo de predicado para las restricciones, interpretado como igualdad semántica en el dominio. El lenguaje propuesto, CLP($H/E$), aúna el estilo de programación lógica basado en cláusulas Horn, el paradigma basado en ecuaciones

# Resumen

(condicionales) y la programación con restricciones. La ventaja de esta nueva aproximación a la integración es que, ya que el lenguaje se define como una instancia del esquema, todas las propiedades semánticas mencionadas anteriormente son heredadas automáticamente por él. Además, si se desarrolla un procedimiento eficiente para resolver restricciones en la estructura *H*/*E,* éste puede ser incorporado fácilmente a un sistema CLP general y cooperar con otros algoritmos de resolución de restricciones. En el diseño de tal procedimiento hemos considerado los siguientes aspectos, de importancia fundamental en el contexto CLP:

- cómo verificar la satisfacibilidad de las restricciones usando un procedimiento correcto y completo de unificación semántica, tal como "narrowing".
- cómo lograr incrementalidad en el proceso de computación.
- cómo simplificar las restricciones en una secuencia de computación.
- cómo utilizar las restricciones de forma activa para reducir la talla del espacio de búsqueda.
- cómo beneficiarse de las derivaciones falladas finitamente como un heurístico para optimizar los algoritmos.

La organización de la tesis es la siguiente. En los capítulos uno y dos se estudia el problema de la integración de la programación lógica y ecuacional, las motivaciones para la misma y las propuestas clásicas de mayor significación. También se revisa el paradigma de programación lógica con restricciones y se analiza la posibilidad de formalizar la integración en dicho marco. Los capítulos tres y cuatro constituyen el núcleo de este trabajo. En ellos se define formalmente el lenguaje CLP(*H*/*E*) y se describe un algoritmo incremental para verificar la satisfacibilidad de las restricciones, como base para la definición de una semántica operacional para el lenguaje. El algoritmo, basado en un procedimiento de "narrowing", no sólo verifica la satisfacibilidad de las restricciones sino que también las simplifica. Describimos el procedimiento de "narrowing" como un cálculo que es guiado heurísticamente por las sustituciones descartadas (en las computaciones previas) durante la búsqueda de soluciones a nuevas restricciones. Se presenta también otro cálculo que, además, construye incrementalmente el árbol de búsqueda para las restricciones. Las propiedades de corrección y de completitud se estudian para todos ellos. En el capítulo cuatro se definen de forma directa, a partir de los resultados del esquema, la semántica por punto fijo y la semántica por teoría de modelos del lenguaje. Definimos modelos capaces de capturar completamente el comportamiento operacional de los programas y de expresar, por tanto, diferentes propiedades observables de los mismos. También se presentan los esperados resultados de corrección y de completitud que establecen la equivalencia entre las tres semánticas: operacional, teoría de modelos y punto fijo. Como una aplicación interesante de las técnicas desarrolladas, en el capítulo cinco se estudia la utilidad de las mismas para la validación de

# Resumen

requerimientos de aplicaciones de bases de datos. Se muestra que dichas técnicas son capaces de operar con especificaciones ejecutables en dos modos distintos, uno de los cuales envuelve una capacidad inferencial que puede utilizarse para resolver problemas de generación automática de planes. Finalmente, se establecen algunas conclusiones y se presentan posibles líneas de continuación de este trabajo. Se incluye como anexo un intéprete Prolog para el lenguaje.

Algunas notas de caracter bibliográfico: gran parte del material presentado en los capítulos tres y cuatro apareció en [Alpuente Falaschi 91, Alpuente et al 91]. La técnica de especificación desarrollada en el capítulo cinco fue presentada en [Alpuente Ramírez 90]. La aproximación CLP(*H*/*E*) al prototipado automático de aplicaciones de bases de datos se presenta en [Alpuente Ramírez 91].

Quiero expresar mi agradecimiento a los profesores Giorgio Levi e Isidro Ramos por su apoyo constante durante la elaboración de esta tesis, gracias al cual ha resultado una experiencia riquísima. Agradezco profundamente el cálido acogimiento encontrado durante mi estancia en Pisa y los numerosos estímulos que allí recibí. Mi gratitud para Moreno Falaschi por su amistad y por los diferentes resultados que son fruto de nuestra colaboración. Un agradecimiento también especial para Mª José Ramírez, con quien he compartido todos los momentos del desarrollo del trabajo y cuya participación ha sido muy importante. Mi gratitud para Vicente Gisbert, cuyas valiosas indicaciones y análisis me han permitido profundizar en numerosos puntos. Gracias a todos mis amigos y, en particular, a Javier Oliver, en quien he encontrado siempre afecto y apoyo. Mi agradecimiento a Juan Carlos Casamayor, Asunción Casanova, Matilde Celma, Joan Climent, Pierpaolo Degano, Maurizio Gabbrielli y Juanjo Moreno por la atenta revisión de diferentes partes del trabajo y por tantas sugerencias interesantes y útiles. Mi reconocimiento a Javier Piris y Mª José Ramis por su inestimable ayuda en las tareas de experimentación. También quiero agradecer el apoyo encontrado en el Departamento de Sistemas Informáticos y Computación para llevar adelante el trabajo.

# 1 Introducción

Una de las áreas de mayor interés y actividad actualmente en el campo de la ingeniería del software centra sus esfuerzos en el desarrollo de conceptos y herramientas que den soporte a la utilización de técnicas de prototipado automático, cuya importancia es bien conocida dentro del campo [Agresti 86, Balzer 85]. Es una tesis ampliamente aceptada que este objetivo puede alcanzarse de forma elegante mediante el uso de técnicas de especificación formal. La lógica matemática subyacente aporta su sintaxis, su semántica y su mecanismo de deducción, que debe ser correcto y completo. Se puede así definir el significado de la especificación en términos de modelos y su ejecución en términos de deducción, lo cual permite considerar la propia especificación (ejecutable) como un prototipo (de muy alto nivel) del sistema. Existen muchos trabajos en la literatura que apuntan esta conveniencia (ver, e.g., [Henderson 86, Levi 86]). Sistemas tales como ASSPEGIQUE [Choppy 87], RAP [Geser Hussmann 86] y SLOG [Choquet et al 86] son ejemplos claros de esta tendencia, que está perdiendo su aire académico y condiciona ya no sólo el software (herramientas CASE de alto nivel, modelización conceptual, bases de datos deductivas, procesamiento del lenguaje natural, ...) sino también el hardware (el proyecto de máquina de reglas de reescritura [Goguen Meseguer 88], la llamada 5ª generación de computadores [Furukawa 87, Furukawa 91], ...).

Desde el momento en que se dispone de herramientas para ejecutar la especificación, no existen diferencias importantes entre los lenguajes de programación de un cierto nivel y los lenguajes de especificación ejecutables de un cierto estilo, con una semántica operacional clara, eficiente y bien definida. El estilo lógico o declarativo se sitúa en ese punto en el que la frontera entre especificación y programa parece desdibujarse. Cabe señalar que, en lo que sigue, no haremos distinción entre especificaciones y programas, ya que entendemos que el lector tiene un amplio conocimiento del ámbito de estudio y no encontrará confusión ni ambigüedad en esta terminología.

## 1.1. Semántica de los lenguajes de programación.

Uno de los principales requisitos de un buen lenguaje de programación es el de disponer de una semántica definida de un modo sencillo, flexible y con un buen soporte formal. La tendencia actual es asociar a los lenguajes de especificación una semántica formal definida en un estilo ya estándar, semántica declarativa y semántica operacional, con resultados de equivalencia entre ambas. De esta forma, las especificaciones pueden ser consideradas como teorías que se pueden animar para verificar el hecho de que su modelo se corresponde con las intenciones del programador, para verificar su corrección (y otras propiedades) respecto de la funcionalidad esperada.[1]

### 1.1.1. Semántica declarativa.

El término "declarativa" indica un tipo de semántica que especifica el significado de los objetos sintácticos por medio de su traducción en elementos y estructuras de un dominio matemático conocido. Se han propuesto muchos métodos que se basan en este principio (ver [Palamidessi 88] para una clasificación). En relación a esta tesis, algunos de los más interesantes, que examinaremos en los párrafos siguientes, son la semántica por teoría de modelos "a la Tarski", la semántica algebraica, la semántica por punto fijo y la semántica denotacional. A menudo se tiende a identificar el método denotacional y el método del punto fijo por la importancia que tiene este último en la aproximación denotacional. Sin embargo, conceptualmente son independientes y conviene diferenciarlos ya que la semántica por punto fijo se aplica también a otro tipo de aproximaciones, como discutiremos posteriormente. También es habitual considerar la semántica algebraica como una variante de la semántica denotacional en la que los modelos se definen en base a construcciones algebraicas provenientes de la teoría de categorías o del álgebra universal [Manes Arbib 86]. Sin embargo, la apariencia es muy distinta puesto que en la aproximación algebraica clásica no se intenta definir cada función semántica sino caracterizar por medio de ecuaciones su comportamiento. En dicha aproximación no es posible poner en evidencia el contenido informativo asociado a los objetos sintácticos; es decir, no se puede derivar de la semántica el hecho de que un objeto está indefinido en un programa o que está más o menos definido que otro, etc. Este problema se puede afrontar imponiendo una estructura de orden sobre el dominio, como en el estilo denotacional de [Scott 82] o en la aproximación basada en álgebras continuas de [Goguen et al 78].

[1] Por ejemplo, en el caso de las especificaciones algebraicas con semántica inicial, se suelen considerar las ecuaciones como reglas que permiten la evaluación de expresiones por reducción a su forma normal. En el caso de que el conjunto de reglas sea canónico, el conjunto de formas normales que se obtienen por reescritura se corresponde exactamente con el álgebra inicial (a una misma clase de equivalencia pertenecen todos los términos que pueden ser probados iguales mediante reescritura). Las técnicas de reescritura permiten también estudiar algunas propiedades interesantes de la especificación, como su consistencia o su completitud.

#### 1.1.1.1. Semántica por teoría de modelos.

La semántica por teoría de modelos está basada en las nociones de *interpretación* y de *modelo*. Una interpretación está constituida por un cierto dominio y una función que asocia a los símbolos de base del lenguaje (símbolos de constante, de función y de predicado) elementos, funciones y relaciones del dominio, respectivamente. Las conectivas lógicas tienen una interpretación predefinida. A cada fórmula se le asigna un valor de verdad y se definen como modelos de un conjunto de axiomas todas las interpretaciones en las que los axiomas resultan verdaderos. El significado que se da a un programa es el conjunto de sus consecuencias lógicas, i.e. el conjunto de fórmulas que son verdad en todos los modelos del programa. La teoría de modelos ha producido resultados importantes, como el teorema de Löwenheim-Skolem (una teoría consistente tiene al menos un modelo numerable, que puede ser construido de un modo totalmente sintáctico), el teorema de completitud de Gödel (equivalencia entre derivabilidad |- e implicación semántica |=) y el teorema de Gödel sobre la indecidibilidad de todas las teorías "interesantes" (es decir, tan potentes, al menos, como la aritmética). En el caso particular de los programas de cláusulas de Horn definidas se cumple también el teorema de existencia del modelo mínimo (de Herbrand) que, en lo que respecta a la verdad de ciertas fórmulas, puede ser considerado como representante de la clase entera de modelos y, por tanto, es utilizado para caracterizar el significado de este tipo de programas.

#### 1.1.1.2. Semántica algebraica.

El método algebraico se usa frecuentemente para la especificación de los tipos de datos [Ehrigh Mahr 85, Wirsing 90] y para la definición semántica de los lenguajes ecuacionales de primer orden [Goguen Meseguer 85, Mahr Makowsky 84, Wirsing 90]. El método se basa en nociones de la teoría de categorías y del álgebra universal. Un resultado fundamental en este campo es el teorema de completitud de Birkhoff, que puede considerarse como la versión algebraica del teorema de completitud de Gödel.

En esta aproximación, la semántica de un programa se define mediante una clase particular de álgebras (de una determinada signatura algebraica) que satisfacen las ecuaciones del programa según un criterio de satisfacción dado. Dependiendo de las características del programa, se puede elegir entre una amplia gama de posibilidades. Pueden tomarse álgebras generadas finitamente, álgebras parciales, álgebras de error, álgebras de géneros ordenados, álgebras continuas, etc. Lo mismo ocurre con la elección de la clase de álgebras. La técnica más usual se basa en la elección de un objeto inicial

(semántica inicial) o de un objeto final (semántica final) de la categoría de álgebras que son modelo de las ecuaciones.

Para cualquier conjunto *E* de ecuaciones, el álgebra inicial tiene una construcción muy simple como el álgebra cociente del álgebra de términos sin variables (términos "ground") o universo de Herbrand *H* entre la relación de congruencia $\equiv_E$, definida como:

$$s \equiv_E t \;\; sii \;\; E \models s = t$$

Para una demostración de la inicialidad de $H/\equiv_E$[2] ver, e. g., el trabajo del grupo ADJ [Goguen et al 78] o el "survey" [Meseguer Goguen 85].

Un álgebra inicial está caracterizada de forma única (salvo un único isomorfismo) por la propiedad de que sólo lo que es demostrable es cierto y el resto es falso. Esta hipótesis de mundo cerrado se define también como *ausencia de basura* (entidades superfluas) y *de confusión* [Meseguer Goguen 85]. Informalmente, la condición de *no basura* equivale a inducción estructural sobre la signatura y la de *no confusión* equivale al hecho de que no hay en el dominio dos elementos iguales que no puedan ser probados como tales.

La variación que introduce la semántica final respecto a la hipótesis de mundo cerrado es que en ésta es cierto todo lo que no puede demostrarse como falso a partir de *observaciones* (los modelos de una especificación con semántica final no tienen *basura* pero sí *confusión*). Los trabajos relacionados con la semántica final pueden considerarse pioneros en el estudio de la noción de comportamiento, pero la integración entre ambos conceptos no ha resultado muy buena por tratarse, como en el caso de la semántica inicial, de una aproximación monomórfica (la clase de modelos se define por isomorfía). Esto excluye la posibilidad de que álgebras no isomorfas puedan considerarse equivalentes en comportamiento y, en particular, excluye como modelos de una especificación sus implementaciones más habituales. El desarrollo de técnicas de estructuración, especialmente *parametrización e implementación,* ha conducido a otras aproximaciones, como la basada en semánticas laxas ("loose"), donde se toman todas las álgebras de la signatura algebraica seleccionada, y las semánticas de comportamiento y observacionales, donde se toman clases de álgebras cerradas por comportamiento (todas las álgebras que "se comportan de una determinada forma"). También han merecido especial atención otras generalizaciones lógicas, como el concepto de "canon" o "data

[2] a partir de ahora, por simplicidad, denotado *H/E*.

constraint" [Ehrigh Mahr 89, Reichel 87]. Para una discusión sobre nuevas tendencias en especificación algebraica ver, e.g., [Ehrigh Orejas 89].

La semántica algebraica y la de los modelos tienen muchos puntos en común. Es bien conocido que el álgebra inicial de un sistema de ecuaciones es isomorfa al modelo mínimo de la correspondiente teoría ecuacional. Esta consideración es de gran importancia cuando se plantea la integración de los paradigmas de programación lógica y ecuacional, como se pondrá de manifiesto más adelante. En ambos casos, sea para construir el álgebra inicial o el modelo mínimo, es posible aplicar el método del mínimo punto fijo de una oportuna transformación continua (transformación que trabaja sobre las relaciones de equivalencia en el primer caso, sobre las interpretaciones en el segundo [Kaplan 84, Lloyd 87]).

#### 1.1.1.3. Semántica por punto fijo.

En la aproximación por punto fijo se define el significado de un programa recursivo (o iterativo) como un determinado punto fijo $\mu_T$ de una transformación T, continua, asociada a dicho programa. La continuidad de T implica la existencia de un punto fijo mínimo y garantiza una contrapartida computacional inmediata, que consiste en poder construir tal objeto de un modo iterativo, en base a repetir la aplicación de la transformación a partir del elemento más pequeño del dominio.

No siempre se requiere la continuidad de tales transformaciones (sobre algunas estructuras, como los retículos completos, la monotonía es suficiente para asegurar la existencia de puntos fijos) y no siempre se elige el punto fijo mínimo. Algunos autores critican la elección del punto fijo mínimo por considerarla arbitraria [Manna Shamir 77]. Para cierto tipo de especificaciones (como, por ejemplo, la semántica de los procesos perpetuos) se han intentado, con cierto éxito, aproximaciones del tipo "máximo punto fijo". En cualquier caso, la elección de puntos fijos distintos del mínimo presenta el grave inconveniente de que éstos no son calculables.

En el caso general, la semántica por punto fijo es la abstracción de un proceso computacional y no proporciona, por sí misma, una semántica declarativa. En el caso de los programas en lógica de cláusulas de Horn definidas la situación es muy diferente. Informalmente, la semántica por punto fijo de un programa lógico viene a expresar cómo construir inductivamente un modelo para el mismo en una forma ascendente: dados los hechos y las reglas, se aplican las reglas a los hechos para generar nuevos hechos, se repite la operación hasta que no se puedan generar hechos nuevos. El menor punto fijo así

obtenido coincide con el menor modelo, como se mostró en [van Emden Kowalski 76]. Así pues, está asociado a consecuencias lógicas y tiene un significado declarativo claro. Por otra parte, la caracterización por punto fijo proporciona un enlace entre la semántica por teoría de modelos y la semántica operacional, lo cual permite simplificar las correspondientes pruebas de equivalencia entre ambas.

#### 1.1.1.4. Semántica denotacional.

La aproximación denotacional, que tiene sus orígenes en los trabajos de Scott, Strachey y de Bakker (ver, e.g., [Scott 82]), se articula en las siguientes fases:

i) Subdivisión de los objetos linguísticos en varios conjuntos (categorías sintácticas o *dominios sintácticos*) y clasificación de los operadores linguísticos (i.e., de las construcciones que permiten componer objetos linguísticos para obtener otros objetos).
ii) Definición de los oportunos *dominios semánticos*, algunos de los cuales corresponden a las categorías sintácticas mientras que otros son auxiliares. Esta definición viene dada constructivamente, partiendo de los dominios de base y especificando dominios cada vez más complejos por medio de *operadores de dominio*. Una técnica alternativa, desarrollada por Scott, consiste en especificar los dominios mediante *ecuaciones de dominio*. La teoría de Scott se basa, esencialmente, en los conceptos de orden parcial, completitud, algebricidad y continuidad (para una revisión, consultar [Barendregt 84] o [Mosses 90]).
iii) Definición de algunas funciones de valuación semántica, que asocian a cada objeto linguístico un elemento del correspondiente dominio semántico y a cada operador un elemento del espacio funcional sobre los correspondientes dominios semánticos.

Algunas características particularmente interesantes de esta aproximación son las siguientes:

- La estructura de orden impuesta a los dominios de base (que induce a su vez un orden sobre los dominios obtenidos constructivamente, según la técnica antes comentada) permite formalizar matemáticamente conceptos ligados al contenido informativo de los objetos sintácticos, como el hecho de que un objeto da lugar a una computación que no termina.
- La aproximación denotacional es, a diferencia de la algebraica, típicamente "de orden superior". De hecho, el significado de un operador sintáctico puede venir especificado como la solución de una ecuación en un lenguaje que de cuenta del orden superior (por ejemplo, el $\lambda$-cálculo [Barendregt 90]).

• El requerimiento de continuidad[3] de las funciones semánticas, además de un puntal para la construcción de la teoría de dominios de Scott, es importante para poder usar el método del punto fijo en la especificación de los objetos semánticos.

### 1.1.2. Semántica operacional.

La semántica operacional es, posiblemente, la más antigua de las definiciones semánticas. El significado del programa se define en términos de las acciones que serían ejecutadas por un modelo abstracto de máquina que, naturalmente, debe haber sido definido con anterioridad. La definición semántica operacional de un lenguaje de programación equivale a la definición de un intérprete para el mismo independiente de cualquier posible implementación. En el caso de la lógica de predicados, e.g., el procedimiento de demostración se comporta como tal intérprete. Una implementación concreta de un lenguaje no tiene por qué seguir los mismos mecanismos operacionales de su definición semántica, pero sí ha de obtener los mismos resultados. Una de las aproximaciones más relevantes a la definición de la semántica operacional es la definida por Plotkin [Plotkin 81] como "Structural Operational Semantics" (SOS), basada en un concepto de máquina verdaderamente abstracta y muy simple: el formalismo de los *Sistemas de Transición*. La semántica de las diversas construcciones del lenguaje se especifica, por inducción estructural, mediante reglas de inferencia. Esta presentación es de gran utilidad cuando se intenta demostrar propiedades de los programas puesto que, a menudo, es posible probar dichas propiedades utilizando conjuntamente inducción estructural e inducción sobre la longitud de las secuencias de transición.

## 1.2. Lenguajes lógicos.

Como se ha comentado en la sección anterior, nuestro concepto de lenguaje lógico es aquél en el que los programas son teorías en una cierta lógica y en el que, además, tres conceptos resultan equivalentes: computación en la máquina, deducción en la lógica y satisfacción en un modelo estándar de la teoría. Por supuesto, un requerimiento adicional de eficiencia y la existencia de un mecanismo efectivo de extracción de respuestas son necesarios para establecer una distinción entre demostración de teoremas y programación lógica.

Se han propuesto varios sistemas lógicos como posible base para un lenguaje de especificación (lógica de primer orden, lógica modal...). Sin embargo, la mayoría de las

[3]Intuitivamente, continuidad significa que nuestra información acerca del valor de una función es proporcional a nuestra información acerca del valor de sus argumentos.

lógicas son demasiado expresivas (hasta la fecha, no abordables computacionalmente). Por ello, si el lenguaje debe ser ejecutable, deberá estar basado en un subconjunto apropiado.

Uno de los ejemplos clásicos de mayor relevancia, en el caso de la lógica de primer orden, es el subconjunto definido por el lenguaje de la lógica de cláusulas de Horn definidas (HCL), cuyas fórmulas son disyunciones de literales conteniendo un único literal positivo [Lloyd 87]. Claramente, con esta limitación no se pueden ya representar todas las fórmulas de primer orden. Sin embargo, desde el punto de vista de la calculabilidad, HCL sí tiene toda la potencia de la lógica de primer orden [Apt 90]. Además, la lógica de cláusulas de Horn constituye el mayor sublenguaje de la lógica de primer orden tal que todos los conjuntos de fórmulas tienen modelo inicial [Makowsky 85]. Este resultado se extiende a la lógica heterogénea y/o con igualdad [Goguen Meseguer 87, Hölldobler 89] y resulta especialmente apropiado para soportar aplicaciones de bases de datos, donde el modelo inicial puede verse como el mundo cerrado del cual el programa habla.

Los fundamentos teóricos para la computación en HCL provienen de los trabajos de Hill [Hill 74] quien demostró, para cláusulas de Horn, la completitud de la resolución lineal con función de selección no restringida (LUSH-resolución), más conocida como resolución lineal con función de selección para cláusulas definidas (SLD-resolución, [Apt van Emden 82]). Posteriormente, [van Emden Kowalski 76], [Clark 78] y [Apt van Emden 82] desarrollaron las propiedades semánticas fundamentales de la programación en HCL, que son:

- la existencia de un dominio de computación canónico.
- la existencia de una semántica de menor y mayor modelo.
- la existencia de una semántica por menor y mayor punto fijo.
- resultados de corrección y completitud fuerte para las derivaciones con éxito y las derivaciones que fallan finitamente.
- resultados de corrección y completitud para la regla de
- negación como fallo.

La regla de negación como fallo se introduce para deducir información negativa de un programa HCL, ya que las consecuencias lógicas de tales programas son siempre positivas. Para una revisión de ésta y otras soluciones clásicas al tratamiento de la negación, ver [Sheperdson 88].

Los últimos trabajos de investigación en el campo de programación lógica en HCL (también conocida como programación lógica relacional [Goguen Meseguer 86]) se han centrado en proporcionar una semántica más adecuada y rica para los programas. Mencionaremos aquí [Falaschi et al 88, Falaschi et al 89], donde se presenta una aproximación a la semántica de los programas lógicos que caracteriza los modelos capturando tanto sus propiedades lógicas como su comportamiento operacional. Por ejemplo, se consigue expresar tanto el conjunto de las consecuencias lógicas atómicas como el conjunto de las respuestas calculadas para un objetivo dado. La técnica se generaliza a programas lógicos con negación.

La presencia de variables lógicas, soportada por el algoritmo de unificación [Knight 89, Lassez et al 88, Robinson 65], es una característica potente y altamente expresiva e, incluso, puede ser vista como un mecanismo primitivo de comunicación en un eventual modelo concurrente del lenguaje. Por otro lado, es el origen de la mayor parte de las diferencias entre los lenguajes lógicos relacionales (HCL) y los ecuacionales (lógica ecuacional), donde el enlace de las variables se produce en un único sentido.

Para un estudio profundo de la lógica de clásulas de Horn como lenguaje de programación, ver [Kowalski 74], [Lloyd 87] o [Apt 90].

En el caso de la lógica ecuacional, que se define como la restricción del cálculo de predicados de primer orden con identidad obtenida suprimiendo de dicho lenguaje toda conectiva lógica y todo símbolo de predicado distinto de la igualdad [Huet 77], una posibilidad es trabajar con teorías ecuacionales canónicas, donde, como ya se ha comentado, es la reescritura o reducción quien juega el papel de intérprete completo y una construcción (algebraica) apropiada (cfr. subsecciones §1.1.1.2., §1.1.1.4) el papel de modelo estándar. La reescritura se entiende como deducción ecuacional usando las ecuaciones sólo de izquierda a derecha. Un paso de computación se ejecuta buscando un subtérmino de la expresión a reducir que sea sintácticamente igual a una instancia de la parte izquierda de la definición de una función ("pattern-matching") y, si tal redex existe, reemplazando el subtérmino por la correspondiente instancia de la parte derecha.

Un formalismo de este tipo resulta ser el cuadro natural desde el que establecer las especificaciones abstractas de una amplia clase de tipos abstractos de datos. Por otro lado, su bajo nivel sitúa a la lógica ecuacional en un lugar adecuado para el análisis de la conexión entre la programación declarativa y la procedural (ASSPEGIQUE [Choppy Kaplan 90], EXCALIBUR [Botella et al 88]). Para un estudio en profundidad sobre

técnicas de reescritura y sistemas ecuacionales, ver [Dershowitz Jouannaud 90, Jouannaud Lescanne 87, Huet Oppen 80, O'Donnell 77, O'Donnell 85].

Los estilos lógicos relacional (HCL) y ecuacional (lógica ecuacional[4]) son bastante similares. En ambos se reemplazan secuencias de símbolos por otras hasta que se alcanza una forma normal o irreducible. En el caso ecuacional, esto queda indicado por una expresión que no contiene redexes, mientras que en el caso HCL es la derivación de la cláusula vacía quien señala el fin de la computación. Tanto en uno como en otro las computaciones ejecutadas son totalmente sintácticas. Cabe señalar aquí que en un contexto más amplio es habitual denominar a la programación lógica relacional "programación lógica" y a la programación lógica ecuacional "programación ecuacional"[5]. En lo que sigue vamos a mantener esta terminología en todos aquellos casos en los que no se preste a confusión.

El representante paradigmático de la familia de los lenguajes lógicos es Prolog, del que existen diversas variantes e implementaciones (ver [Cohen 88] para un análisis sobre sus orígenes y desarrollo). Algunos aspectos del lenguaje que han merecido especial atención han sido "backtracking" inteligente [Bruynooghe Pereira 84, Wolfram 86], metafacilidades [Sterlig Shapiro 86], mecanismos de control [Naish 86], técnicas de compilación eficiente [Campbell 84, Hogger 84, Maier Warren 87], técnicas de programación [Bratko 86, Sterling Shapiro 86], interpretación abstracta [Abramski Hankin 87, Bruynooghe 88, Marriot Sondergaard 89], tipos y módulos [Hanus 90b, Mishra 84, Mycroft O´Keefe 84, Smolka 89] y paralelismo [Clark Gregory 86, Hermenegildo 86, Shapiro 89].

En la familia de los lenguajes ecuacionales de orden superior, los lenguajes más puros y mejor establecidos son Haskell [Hudak 89], Hope [Burstall 80], Miranda [Turner 85] y ML [Milner et al 90]. Entre los de primer orden (donde, generalmente, la potencia del orden superior se obtiene mediante módulos parametrizados), cabe destacar ACT [Ehrigh Mahr 85], CLEAR [Burstall Goguen 77, Burstall Goguen 80], LOOK [Zilles et al 82], el lenguaje en [Hoffmann O'Donnell 82] y OBJ [Futatsugi et al 85, Goguen Meseguer82, Goguen Tardo 79, Goguen Winkler88, Kirchner et al 88], una de cuyas

[4] englobamos aquí también a los lenguajes basados en (algún tipo de) λ-cálculo, que puede verse como una lógica ecuacional de orden superior [Meseguer 89].

[5] Algunos autores llaman "lenguajes funcionales" a la clase de lenguajes ecuacionales que distinguen entre funciones definidas y constructores de datos [Reddy 85]. Aunque esta terminología goza de amplia difusión, no será hecha explícita en este trabajo. También es habitual reservar el término "ecuacional" para la programación en lógica ecuacional de primer orden pero, como ya se ha comentado, en este trabajo se utiliza con un caracter más general, para enfatizar el hecho de que los lenguajes de esta clase utilizan ecuaciones recursivas para la definición de funciones y algún tipo de razonamiento ecuacional (e.g., reglas de reducción) para la deducción.

implementaciones más robustas es AXIS [Coleman et al 88] (ver [Wirsing 90] para una revisión de éstos y otros lenguajes ecuacionales). Algunos trabajos interesantes sobre técnicas de compilación para lenguajes ecuacionales son [Geser et al 88, Kaplan 87, Peyton-Jones 87].

Los últimos avances en las técnicas de compilación para cláusulas de Horn y lenguajes ecuacionales permiten desarrollar compiladores que hacen la eficiencia de estos lenguajes aceptable en comparación con la de lenguajes más convencionales ejecutados sobre las mismas máquinas. También, la aparición de nuevos modelos y arquitecturas para computación paralela contribuye a incrementar sustancialmente la velocidad de los cálculos desde el momento que el paralelismo está presente implícitamente en la semántica operacional de los lenguajes lógicos y ecuacionales. Una primera consecuencia es que la misma caracterización semántica sirve para las versiones secuencial y paralela de dichos lenguajes: los conceptos de consecuencia lógica, sustitución de respuesta computada, paso de resolución, forma normal, ... son independientes de la forma (secuencial o paralela) en que son evaluados o computados.

En el caso de lógicas no clásicas, un ejemplo interesante es el constituido por el sistema MOLOG [Fariñas 86], basado en una extensión modal de la lógica de cláusulas de Horn, que constituye una herramienta (razonablemente) eficiente para la mecanización de este tipo de lógicas.

## 1.3. Integración de lenguajes lógicos.

Como es bien conocido, muchos conceptos importantes de los distintos estilos de programación son capturados de manera natural por diferentes lógicas. Por ejemplo, la lógica ecuacional da cuenta de un estilo de programación funcional mientras que la lógica de cláusulas de Horn lo hace de un estilo de programación relacional, la lógica heterogénea da soporte formal a sistemas de tipos y módulos mientras que la lógica de géneros ordenados da cuenta de mecanismos de herencia, la lógica temporal (y, más recientemente, la así llamada lógica de reescritura condicional [Meseguer 91]) proporciona modelos para la concurrencia, la lógica modal permite razonar acerca de cambios en un conjunto de creencias, etc. En la literatura se propone "a hundred thousand" de lenguajes de especificación [Goguen 86], basados en sistemas lógicos como éstos o en otros muy diversos. En [Levi 86] se estudian algunas de las debilidades generales de los sistemas lógicos, desde el punto de vista de la especificación y el prototipado. Dichas debilidades hacen referencia, fundamentalmente, a los siguientes puntos: la importancia del entorno de especificación (que debería facilitar el desarrollo y

prueba del prototipo, incorporando herramientas similares a las usadas para el desarrollo de programas convencionales y otras adicionales que sirvan de base, e.g., a una metodología de transformación y verificación formal de programas), la carencia, en muchos de los sistemas propuestos, de mecanismos bien establecidos de estructuración y modularización de programas (necesarios para escribir y comprender especificaciones de sistemas reales) y, más problemático si cabe, la no existencia de una lógica efectiva capaz de dar soporte a todo lo que sería de utilidad expresar (relaciones, funciones, concurrencia, no-terminación, tipos de datos, orden superior, herencia...).

En relación al último punto, si la elección de una lógica apropiada para un cierto tipo de aplicación permite disponer de un lenguaje de especificación adecuado a ese tipo de aplicaciones, la combinación de diferentes sistemas lógicos permitirá definir un lenguaje más potente, adecuado para un rango más amplio de aplicaciones. Recientemente, se ha propuesto la combinación de diferentes lenguajes lógicos, correspondientes a distintos paradigmas de especificación (o programación), en un lenguaje coherente único con el que abordar cada problema en el formalismo más apropiado. La manera más adecuada de unificar lenguajes basados en sistemas diversos es unificar sus lógicas, introduciendo una noción de modelo y de deducción lo bastante generales como para englobar los distintos estilos e incluirlos como un caso particular. Pero esto es posible sólo si los distintos lenguajes tienen los mismos mecanismos operacionales básicos y una estructura similar o compatible para los modelos [Barbutti et al 84]. En [Goguen Burstall 84] se introduce la noción de institución como una formalización categorial de la noción intuitiva de sistema lógico o como una teoría de modelos (verdaderamente) abstracta, que facilita la transferencia de resultados de un sistema lógico a otro y proporciona un marco teórico apropiado para combinar diferentes lenguajes en un lenguaje sencillo[6]. Algunos ejemplos de instituciones son la lógica de primer orden, la lógica ecuacional condicional, la lógica de cláusulas de Horn y la lógica de cláusulas de Horn con igualdad, que generaliza las dos anteriores y es la base de una de las aproximaciones más populares a la integración de las dos familias más prometedoras de lenguajes declarativos: los lenguajes lógicos y los ecuacionales.

En el próximo capítulo se discute el problema de la integración de los paradigmas de programación lógica y ecuacional, con la intención de desarrollar un sistema de especificación combinado que sirva de soporte a una técnica de prototipado más potente, y se revisan algunas de las propuestas que, en nuestra opinión, han tenido

[6]Con la intención de proporcionar una axiomatización de la noción de programación lógica, [Meseguer 89] introduce, entre otras, la noción de *lógica general*, que consta de dos componentes, una modelística, que viene dada por las instituciones, y otra, la deductiva, dada por su noción de sistema deductivo ("entailment system") y la noción de morfismo entre lógicas.

mayor influencia en este campo. En particular, examinaremos el problema del tratamiento de la igualdad, que es la noción sobre la cual se basa una de las aproximaciones de mayor significación. Cualquier concepto definido o utilizado de manera informal en el próximo capítulo puede consultarse en §3.1, donde se agrupan todas las definiciones técnicas de interés para la tesis.

# 2 Programación lógico-ecuacional con restricciones

## 2.1. Integración lógico - ecuacional: motivaciones y propuestas.

En los últimos años se ha dedicado un gran esfuerzo al problema de la integración de la programación lógica y ecuacional (consultar [Bellia Levi 86], [Dershowitz Plaisted 88], [Guo et al 90], [Moreno 89] y [Reddy 86] para una revisión, análisis y clasificación de las propuestas más significativas; ver también la colección en [DeGroot Lindstrom 86], que constituye una referencia estándar en este campo). Ambos estilos se benefician de las características que les brinda su naturaleza declarativa, aunque ello no impide que existan importantes diferencias estilísticas y de fundamento entre ellos. En las dos próximas subsecciones se revisan las principales diferencias entre los dos estilos y se discuten algunas de las motivaciones que hacen interesante su integración. A partir de este análisis es posible establecer una clasificación de las diferentes propuestas existentes, que será ofrecida en la subsección tercera.

### 2.1.1. Las características de los lenguajes lógicos y ecuacionales.

Como se ha comentado en el capítulo anterior, el estilo lógico se caracteriza por la presencia de variables lógicas, i.e. unificación, y por el no determinismo, i.e. computación basada en búsqueda. La bidireccionalidad de la unificación, en contraste con la unidireccionalidad del "pattern-matching", permite inversibilidad de definiciones y estructuras de datos parcialmente definidas ("non ground output") y, por tanto, un estilo de programación más compacto y flexible. El estilo ecuacional comparte con el lógico su simplicidad formal pero, además de diferir en aspectos notacionales, los lenguajes ecuacionales ofrecen una variedad de conceptos de programación de gran utilidad, como funciones evaluables, estrategias de computación "innermost/outermost", tipos y polimorfismo, orden superior, estructuras de datos infinitas y evaluación perezosa. En la tabla que se muestra a continuación se resumen de forma esquemática las diferencias entre ambos estilos (en sus versiones más estándar), que suponemos conocidas. Para un estudio en profundidad ver, e.g., [Darlington et al 86, Hudak 89].

| PROGRAMACION LOGICA | PROGRAMACION ECUACIONAL |
|---|---|
| NOTACION: cláusulas que definen relaciones | NOTACION: ecuaciones que definen funciones |
| S. DECLARATIVA: teoría de modelos | S. DECLARATIVA: algebraica o denotacional |
| S. OPERACIONAL: SLD-resolución | S. OPERACIONAL: reescritura |
| unificación | "pattern-matching" |
| variables lógicas, cuantif. existencialmente | variables cuantificadas universalmente |
| no direccionalidad entrada/salida | direccionalidad |
| inversibilidad de definiciones | uso único de definiciones |
| estructuras de datos parcialmente definidas | datos completos |
| no determinismo (computación con búsqueda) | determinismo (computación sin búsqueda) |
| sin tipos | tipos y polimorfismo |
| no datos potencialmente infinitos | datos potencialmente infinitos |
| evaluación voraz | evaluación perezosa |
| primer orden | orden superior |

### 2.1.2. ¿Por qué integrar lenguajes lógicos y ecuacionales?

Resulta evidente que cada estilo tiene mucho que ofrecer al otro. Los lenguajes lógicos carecen de la lógica de la igualdad y de las funciones no libres; los lenguajes ecuacionales carecen de la lógica de las relaciones y de las variables lógicas. Su integración es, por tanto, muy deseable, puesto que el lenguaje resultante podría beneficiarse ampliamente de las facilidades de la lógica (funciones, relaciones e igualdad), permitiendo a sus usuarios utilizarlas separadamente o combinarlas de la forma más adecuada para cada aplicación particular. La integración de estos dos estilos, por otro lado, proporciona facilidades adicionales que enriquecen el campo de la modelización conceptual de aplicaciones de bases de datos, como se informó en [Alpuente Ramírez 90, Alpuente Ramírez 91] y será recordado en el capítulo cinco de esta tesis.

Su unificación parece, por otro lado, bastante natural, en cuanto que ambos estilos están soportados por mecanismos operacionales basados en operaciones íntimamente relacionadas ("pattern-matching" y unificación) que operan sobre las mismas estructuras de datos (términos bien formados) y sus modelos pueden definirse a partir de una misma construcción básica, el álgebra de términos sin variables o universo de Herbrand.

Los intentos de integrar la programación lógica y ecuacional han seguido diferentes direcciones. Se observa un creciente interés por la introducción de

características de orden superior como un ingrediente esencial para el lenguaje integrado, generalmente soportadas por una axiomatización del λ-cálculo como una teoría ecuacional de primer orden (confrontar, e.g., [Aït-Kaci Nasr 89, Bellia et al 87, Cheng et al 90, González et al 90, Miller Nadathur 86, Smolka 88]). Sin embargo, la tendencia inicial, que centró sus esfuerzos en intentar combinar programación lógica (HCL) y programación ecuacional (lógica ecuacional) de primer orden, todavía está considerada como una aproximación significativa, cuyos principales resultados teóricos y prácticos pueden ser explotados en posibles extensiones al orden superior. La ventaja de la restricción al primer orden es que la integración puede ser lograda en un marco formal puramente lógico, el de la lógica de cláusulas de Horn con igualdad. Por otro lado, dado que las definiciones de función de primer orden son, básicamente, igualdades dirigidas (es decir, reglas de reescritura), algunos resultados importantes bien conocidos en el dominio de los sistemas de reescritura de términos pueden ser reutilizados, con algunas adaptaciones, en el diseño de un lenguaje integrado.

### 2.1.3. Taxonomía de la integración.

La integración de la programación lógica y ecuacional en un marco formal unificado ha sido aproximada desde ambos extremos, el lógico y el ecuacional, con diferentes grados de rigor matemático y de claridad semántica. En las primeras propuestas se proporcionaba simplemente una interfaz entre reducción de objetivos (resolución) y evaluación de términos (reescritura), sin integrarlos en un marco unificado y sin desarrollar una semántica uniforme. Algunos ejemplos son APPLOG [Cohen 86], QLOG [Komorowski 82], POPLOG [Mellish Hardy 84] y LOGLISP [Robinson Sibert 82]. En estos lenguajes, los términos se reescriben a su forma normal antes de intentar la unificación y la definición de funciones no da lugar a la instanciación de variables libres durante la computación. Una situación similar ocurre en el lenguaje Le Fun [Aït-Kaci et al 87], basado en un mecanismo de residuación o retraso de la unificación hasta la instanciación de todas las variables de la ecuación, lo cual permite la reducción de cada miembro antes de intentar la unificación. Los lenguajes así definidos no son completos, en el sentido de que no necesariamente son encontradas todas las soluciones a un objetivo.

De mayor significación son las propuestas en las que se intenta incorporar a cada estilo la lógica de la cual carece y que confiere al otro la potencia que le caracteriza (la lógica de la igualdad y las funciones no libres para el estilo ecuacional; la lógica de las variables lógicas y del no determinismo para el lógico). Bajo esta consideración, la

siguiente clasificación refleja sólo las aproximaciones que consideramos más interesantes, aquéllas que dan soporte a las técnicas de integración mejor establecidas:

ι) Aproximación ecuacional + lógica:

En esta aproximación, la integración se entiende como la adición de variables lógicas y unificación a un lenguaje ecuacional para obtener un lenguaje lógico con sintaxis ecuacional (e.g., reglas de reescritura condicionales[7] [Bergstra Klop 86, Kaplan 84]).

El procedimiento de "narrowing", que fue introducido originalmente en el campo de la demostración automática de teoremas [Lankford 75, Slagle 74] y posteriormente usado para resolver problemas de "matching" y unificación semántica o generalizada [Dincbas van Hentenryck87, Fages Huet 83, Fay 79, Hullot 80, Plotkin 72, Siekmann Szabo 82] ha sido incorporado como núcleo de la semántica operacional de los lenguajes de esta clase. Consultar, e.g., [Nutt et al 89] para una buena revisión sobre "narrowing". El "narrowing" puede ser visto como una extensión propia de la reescritura que puede ser implementada eficientemente sustituyendo "matching" por unificación en el procedimiento de reducción. Citando [Hullot 80], "el narrowing de una expresión es aplicarle la mínima sustitución tal que la expresión resultante es reducible y entonces reducirla; la sustitución se encuentra unificando la expresión con la parte izquierda de las ecuaciones". En la subsección §3.3.1.2 se incluye una descripción formal de este procedimiento.

El proceso de "narrowing" es no determinista, con diferentes sustituciones para las variables lógicas correspondientes a diferentes caminos. Se tienen dos grados de libertad: uno para la elección del subtérmino dentro de la expresión y otro para la elección de la ecuación cuya parte izquierda unifica con el subtérmino. Esta situación es, esencialmente, la misma que se presenta en programación lógica respecto de la selección del subobjetivo y de la cláusula a resolver. De hecho, como se apunta en [Bosco et al 87], "narrowing" y resolución son dos reglas de inferencia distintas pero basadas en un mismo tipo de mecanismo: la unificación entre una porción de un objetivo (un subobjetivo, en el caso de la resolución, un subtérmino de una ecuación, en el caso de "narrowing") y una porción de una regla (la cabeza de la cláusula, en el caso de la resolución, la lhs de la regla de reescritura, en el caso de "narrowing"), la aplicación del unificador al objetivo y

[7] Se debe notar que, si se usan las ecuaciones condicionales para representar clásulas de Horn, la condición habitual de ausencia de variables extra (variables que no están en la parte izquierda (lhs) de la cabeza de la cláusula y sí en las correspondientes partes derechas (rhs) o en la condición) no se mantiene en el caso general.

a la regla y, finalmente, la construcción de un nuevo objetivo combinando adecuadamente (las instancias de) las dos porciones restantes del objetivo y de la regla. El procedimiento de "narrowing" no sólo subsume al de reducción (en el sentido de que, si se aplica a una expresión sin variables cuantificadas existencialmente, se obtiene la misma secuencia de reducciones que se obtendría por reescritura (i.e., si $e \rightarrow d$ es una reducción, entonces $e \Rightarrow^{id} d$ es un "narrowing", con *id* la sustitución identidad [Reddy 86])) sino también al de SLD-resolución (cuando los predicados son vistos como funciones booleanas y los objetivos como expresiones "non-ground" que deben ser reducidas a true). De este modo, se puede manipular la componente lógica pura usando el mismo mecanismo computacional que para la ecuacional. El procedimiento de "narrowing" es completo para teorías ecuacionales que tienen las propiedades de confluencia y terminación finita [Hullot 80]. También se conocen resultados de completitud para "narrowing" condicional permitiendo incluso el uso de variables extra en las condiciones de las reglas y aún en ausencia de la propiedad de terminación [Giovannetti Moiso 86], lo cual resulta especialmente importante para el tratamiento de funciones y relaciones definidas sobre estructuras de datos infinitas, pues hace innecesario garantizar tal propiedad. Estos resultados son particularmente interesantes en el caso de teorías con disciplina de constructores [Giovannetti Moiso 86, Levi et al 87, Moreno Rodríguez 89].

Los lenguajes ecuacionales con semántica por "narrowing" tienen la potencia expresiva de los lenguajes lógicos y algunas de sus características, como invertibilidad, no determinismo y estructuras de datos parcialmente definidas. El principal inconveniente es, en general, la falta de una notación adecuada para las relaciones que, normalmente, suelen ser tratadas como funciones booleanas.

Esta clase incluye SLOG [Fribourg 84b, Fribourg 85b, Choquet et al 86], RAP [Hussmann 86], BABEL [Moreno Rodríguez 88, Moreno Rodríguez 89], QUTE [Sato Sakurai 84] y los lenguajes en [Dershowitz Plaisted 85] y [Reddy 85, Reddy 87].

Hay otros métodos para extender lenguajes ecuacionales con características lógicas, como los basados directamente en el procedimiento de complección, que puede interpretarse como el intento de saturar un conjunto de ecuaciones mediante reglas de inferencia, con la esperanza de derivar un conjunto finito sin consecuencias no triviales. El procedimiento computa "pares críticos" (obtenidos cuando las partes izquierdas de dos reglas se superponen). El procedimiento de complección fue introducido inicialmente para derivar sistemas de reescritura canónicos para teorías ecuacionales [Knuth Bendix 70] y posteriormente utilizado para probar la validez de ecuaciones en el álgebra inicial

de una variedad ecuacional [Goguen 80, Huet Hullot 82, Jouannaud Kounalis 89]. Para una revisión sobre complección ecuacional ver, e.g., [Bachmair et al 86, Buchberger 87, Jouannaud Kirchner 86]. Esta aproximación, que se generaliza fácilmente a cláusulas de Horn [Dershowitz Josephson 84, Dershowitz 91, Kounalis Rusinowitch 88, Paul 86], no ha merecido excesiva atención en el contexto de la integración de la programación lógica y ecuacional, ya que cualquier forma de complección más general que "narrowing" (que es, esencialmente, una complección lineal), aunque interesante desde el punto de vista de la demostración de teoremas, no resulta adecuada para programación lógica[8] [Bellia Levi 86]. Por ejemplo, la semántica operacional de SLOG [Fribourg84b, Fribourg 85b] es superposición clausal, una restricción lineal de la complección que puede verse, esencialmente, como "narrowing" [Bellia Levi 86]. El procedimiento de complección subsume tanto el "narrowing" como la regla de resolución. De hecho, un paso de resolución puede ser interpretado como una superposición y un resolvente como un par crítico [Paul 84, Buchberger 87].

Otra aproximación bastante diferente, basada en la noción de abstracciones de conjunto ("set abstraction"), considera los predicados como funciones valuadas en conjuntos que devuelven el conjunto de tuplas de la relación [Darlington et al 86, Robinson Sibert 82]. La solución técnica utilizada es, nuevamente, "narrowing", donde el no determinismo se reemplaza ahora por unión de conjuntos.

ιι) Aproximación lógica + ecuacional:

Una de las características de la programación lógica tradicional es el hecho de que las estructuras de datos sobre las que trabaja un programa se componen únicamente de constructores. Esto significa que cada objeto coincide con su valor; es decir, la única igualdad reconocida es la identidad sintáctica. En términos declarativos, esto significa que el universo de Herbrand coincide con el universo de Herbrand cociente; es decir, cada clase de equivalencia en el algebra inicial contiene un único elemento. En términos computacionales, esto implica que, para la completitud de la regla de inferencia de resolución respecto de la semántica declarativa del lenguaje, basta determinar el unificador más general (mgu) sintáctico (que puede calcularse con uno de los varios algoritmos existentes [Herbrand 71, Martelli Montanari 82, Paterson Wegman 78]). La situación es muy diferente si queremos considerar los objetos de un programa lógico

[8]Es ampliamente aceptado que un intérprete para un lenguaje de programación, a diferencia de un demostrador de teoremas, no debe deducir nuevos axiomas por comparación entre las sentencias del programa ni entre subobjetivos alternativos generados en una computación no determinista [Dershowitz Plaisted 88] (un compilador, por otra parte, podría hacer ambas cosas por razones de eficiencia). Esta consideración es comparable a la idea que condujo a la utilización de resolución lineal como la base de un intérprete para programación lógica pura.

como representantes de un valor distinto (del de sí mismos como estructuras sintácticas) y queremos que objetos sintácticamente diferentes puedan tener el mismo valor.

La idea central en esta aproximación es incorporar igualdad a un lenguaje lógico para obtener un lenguaje lógico con características funcionales, como la posibilidad de definir y evaluar funciones. Introduciendo la relación de igualdad, la notación funcional es inmediata y trae consigo la noción de "función ejecutable", permitiendo que aparezcan símbolos de función interpretados como argumentos de las relaciones y que variables cuantificadas existencialmente aparezcan como argumentos de las funciones. Si no estamos interesados en un lenguaje capaz de representar funciones y predicados de orden superior, i.e. si nos restringimos al primer orden, es posible formalizar la integración en un contexto puramente lógico: el de la lógica de cláusulas de Horn con igualdad. El lenguaje integrado permitirá una combinación apropiada de átomos ecuacionales y predicados ordinarios (átomos relacionales).

El verdadero problema con la igualdad es la complejidad del procedimiento de refutación. Resulta necesario imponer algunas restricciones a su uso para preservar las propiedades computacionales de los programas lógicos tradicionales, que los hacen tan simples y potentes. La primera restricción general es el uso dirigido de la igualdad lo cual requiere, si se pretende preservar la propiedad de completitud (como ocurre en el caso de las técnicas de reescritura), que los programas disfruten de la propiedad de confluencia [Bergstra Klop 86, Huet 80]. Esta propiedad puede ser lograda tras una fase de complección previa (que, desafortunadamente, no siempre termina) o asegurada por medio de restricciones sintácticas (que valen incluso en ausencia de la propiedad de terminación, como en [Huet Levy 79]). Si se permiten variables extra en los cuerpos de las reglas, la confluencia ya no es suficiente para garantizar completitud, incluso en presencia de la propiedad de terminación. En este caso se requiere alguna condición más fuerte, tal como la confluencia separada de cada uno de los (posiblemente infinitos) niveles de reescritura que conforman la relación de reescritura global [Giovannetti Moiso 86]. Aún en este caso existen caracterizaciones sintácticas suficientes [Levi et el 87, Moreno Rodríguez 89].

La importancia de la igualdad en deducción automática fue reconocida ya con anterioridad a los orígenes de la programación lógica. Incorporar igualdad dentro del proceso de deducción ha añadido gran potencia a los demostradores automáticos de teoremas, siendo precisamente éste el campo donde se desarrollaron las principales propuestas para su tratamiento.

La propuesta más simple consiste en incluir explícitamente la teoría ecuacional $E$ y los axiomas de la igualdad $E_q$[9] como parte de la teoría lógica [Chang Lee 73, Kowalski 70]. Con este método no hay necesidad de cambiar el sistema de inferencia; los resultados de corrección y completitud no se ven afectados [Furbach et al 89, Hölldobler 89]. Sin embargo, desde un punto de vista operacional, los axiomas que representan las propiedades lógicas de la igualdad conducen a un crecimiento desmesurado del árbol de búsqueda, con numerosas ramas infinitas e inútiles, lo que se traduce en intratabilidad (consultar, e.g., [van Emden Yukawa 87] para un ejemplo que permite apreciar la complejidad del espacio de búsqueda en presencia de los axiomas de la igualdad).

Siguiendo las ideas en [Chang Lee 73, Plotkin 72, Siekmann 84], será necesario evitar la introducción explícita de estos axiomas en el programa, incorporándolos de forma implícita dentro de la maquinaria deductiva. Si se tiene en cuenta que las principales componentes de un demostrador de teoremas por resolución son:

- un algoritmo de unificación
- una implementación de la regla de resolución,

esta consideración conduce a pensar en varias posibles estrategias para extender las técnicas de demostración de teoremas por resolución a teorías con igualdad (evitando todas ellas una representación explícita de los axiomas de la igualdad).

Una primera posibilidad consiste en mantener la resolución como la única regla de inferencia, pero sustituir el algoritmo de unificación estándar (sintáctico) por un procedimiento de unificación bajo la correspondiente teoría de la igualdad (asociatividad, conmutatividad...) [Plotkin 72].

Otra posibilidad está en mantener el algoritmo de unificación tradicional pero extender la otra componente, la regla de inferencia, bien generalizándola a una resolución semántica con igualdad predefinida [Robinson 68] o mediante la introducción de una regla de inferencia adicional, como la paramodulación [Robinson Wos 69], para manipular la igualdad. La paramodulación sustituye varios pasos de resolución por una instanciación seguida del reemplazo de un subtérmino [Furbach et al 89, Padawitz 88, Rusinowitch 87]. Cuando se combina con resolución, proporciona un conjunto completo

[9] $E_q$ = { $X = X \leftarrow$. (reflexividad), $X = Y \leftarrow Y = X$. (simetría), $X = Z \leftarrow X = Y, Y = Z$. (transitividad),
$f(X_1, X_2, ..., X_n) = f(Y_1, Y_2, ..., Y_n) \leftarrow X_1 = Y_1, X_2 = Y_2, ..., X_n = Y_n$.
para cada símbolo de función $n$-ario $f$ (substitutividad de funciones),
$p(X_1, X_2, ..., X_n) \leftarrow p(Y_1, Y_2, ..., Y_n), X_1 = Y_1, X_2 = Y_2, ..., X_n = Y_n$.
para cada símbolo de predicado $n$-ario $p$ (substitutividad de predicados) }

de reglas de inferencia para la lógica de primer orden con igualdad, suponiendo que se añaden también el axioma de la reflexividad y los axiomas reflexivos funcionales (de la forma $f(X_1, X_2, ...,X_n) = f(X_1, X_2, ..., X_n)$, para cada símbolo de función $n$-ario $f$ que ocurre en el programa) [Chang Lee 73, Gallier86, Hölldobler88]. Para una discusión acerca de las condiciones bajo las cuales dichos axiomas no son necesarios, consultar [Hölldobler 88, Hölldobler 89]. La regla de paramodulación también puede considerarse subsumida por el procedimiento de complección ya comentado, como se argumenta en [Zhang Kapur 88].

La aproximación basada en paramodulación, aunque de especial relevancia en el campo de la demostración automática de teoremas, no ha trascendido al campo de la programación lógica debido a que la talla del espacio de búsqueda resulta ser todavía excesivamente grande. Sin embargo, también en este caso el procedimiento de "narrowing" puede verse como la versión restringida de la paramodulación (donde sólo se permite unificación con las partes izquierdas de las ecuaciones y únicamente en ocurrencias no variables de la expresión a reducir) que ha sido reconocido como procedimiento clave para incorporar igualdad en el proceso de deducción [Hölldobler 89].

Una última propuesta, que también ha tenido enorme influencia, se basa en los trabajos de Brand sobre el así llamado *método de descomposición*. En [Brand 75] se propone una transformación de un conjunto de cláusulas con igualdad que consiste en eliminar la composición funcional, reemplazándola por el operador lógico "and" mediante el uso de variables auxiliares. Por ejemplo, una expresión con un anidamiento funcional como $p(g(a))$, se transformaría en la conjunción de átomos $a=Y, g(Y)=X, p(X)$. Esta transformación de aplanamiento ("flattening") subsume algunos de los axiomas de la igualdad (transitividad y sustitutividad de funciones y de predicados [Cox Pietrzykowski 85, Hoddinot Elcock 86, Elcock 89]). Con una sencilla extensión de la transformación, resulta posible subsumir también el axioma de la simetría. Sin embargo, resulta inconveniente (e innecesario si el conjunto de reglas es confluente puesto que, como ya se ha comentado, la confluencia autoriza a utilizar las ecuaciones en un sólo sentido, sin pérdida de completitud). El axioma de la reflexividad, por el contrario, no puede ser evitado, en el caso general, si se pretende preservar la completitud. Posteriormente analizaremos las condiciones que autorizan su eliminación. Una transformación similar (transformación a la forma homogénea) fue utilizada para la prueba de la corrección de Prolog II en [van Emden Lloyd 84] y en los trabajos de Yamamoto sobre combinación de resolución y "narrowing" [Yamamoto 87].

Tal y como se ha discutido anteriormente, el verdadero problema con la igualdad está en la complejidad del procedimiento de refutación. Como se comenta en [van Emden Yukawa 87], una de las razones por las que la programación lógica tuvo éxito donde otros métodos de demostración de teoremas fracasaron (en cuanto a su utilidad con fines de programación) fue la explícita prohibición del uso de la relación de igualdad. Sólo muy recientemente ha sido reconocida la importancia de la igualdad en programación lógica y, con la experiencia ganada en el campo de la deducción automática, se han desarrollado diferentes propuestas para su tratamiento.

Los métodos computacionales que han sido propuestos para la adición de igualdad en programación lógica son la versión refinada de las técnicas comentadas anteriormente. Como especialmente significativos, comentaremos los métodos de flat-SLD-resolución y SLDE-resolución.

ιι.1) flat-SLD-resolución:

Esta aproximación se basa en la transformación de aplanamiento descrita anteriormente. Como veremos, flat-SLD-resolución equivale a SLD-resolución sobre programas transformados aumentados, si es necesario, con la cláusula ($X = X \leftarrow$.) [Bellia et al 87, Bosco et al 87, Deransart 83, Levi et al 87, Tamaki 84, van Emden Maibaum 81, van Emden Yukawa 87].

La principal diferencia entre el "narrowing" y la resolución está en el hecho de que el primero, para intentar la unificación, toma del interior del objetivo a resolver cualquier subtérmino del literal seleccionado, mientras que la resolución toma el literal entero. Para simular "narrowing" mediante resolución basta, por tanto, con aplanar los términos, tanto en el subobjetivo como en las reglas del programa. Es decir, hay que desanidar los subtérminos para que la resolución pueda requerir también su unificación [Bosco et al 87]. Por consiguiente, cada cláusula de Horn con igualdad es sometida a una transformación sintáctica en la que se eliminan los anidamientos funcionales, reemplazando recursivamente cada llamada funcional $f(t_1, t_2, ..., t_n)$ por una variable auxiliar nueva, digamos $X$, y añadiendo el átomo funcional $f(t_1, t_2, ..., t_n) = X$ en el cuerpo de la cláusula. El símbolo = es considerado un predicado ordinario tras la transformación. El axioma de la reflexividad ($X = X \mathfrak{R}$.) se añade, si es necesario, al programa transformado. La aplicación de este axioma corresponde al paso final de unificación sintáctica en cualquier secuencia de "narrowing" y también a un paso de "narrowing" nulo (equivalente a la posibilidad de no seleccionar un subtérmino). El axioma resulta innecesario cuando se trabaja con teorías con disciplina de constructores

si, además, las funciones están totalmente definidas sobre cada valor que sus argumentos pueden tomar (i.e., las cláusulas que definen cada función cubren todo su dominio [Bosco et al 87, Fribourg 84a] y, consideradas como reglas de reescritura, definen un sistema en el que las formas normales no contienen funciones definidas; de esta forma, cualquier término "ground" se reduce a un término "data", formado sólo por constructores). En [Echahed 88, Fribourg85a, Huet Hullot 82] se dan condiciones suficientes para la satisfacción de esta propiedad, conocida como "principio de definición".

Es conocido que, bajo ciertas condiciones y al precio del preproceso de "compilación" (aplanamiento) descrito anteriormente, flat-SLD-resolución es semánticamente equivalente a un "narrowing" refinado (para cada paso de "narrowing" a partir del objetivo original en el programa original se puede encontrar un paso de resolución equivalente en el programa y objetivo transformados) y más eficiente que un "narrowing" ordinario [Bosco et al 88] puesto que, debido a la estrategia de selección, se evitan algunas computaciones redundantes.

Una regla de selección "innermost" puede implementarse fácilmente a través de la regla de selección "leftmost" habitual de Prolog, si la transformación de aplanamiento va colocando los literales planos en el orden adecuado. La estrategia "outermost", en cambio, no puede ser implementada por medio de una compilación trivial ya que el orden de selección de átomos no se conoce estáticamente (sólo puede ser determinado en tiempo de ejecución). Una definición completa de la estrategia "outermost" se puede encontrar en [Levi et al 87] (ver también [Moreno et al 90]). Esta regla selecciona los átomos explotando la relación "productor-consumidor" expresada, en el programa original, por el anidamiento funcional, sin requerir la explícita resolución contra la cláusula $X = X \leftarrow$. (que, en otro caso, sería necesaria). Una optimización adicional simula una estrategia de simplificación basada en normalización.

La aproximación basada en "flattening" resulta muy agradable, ya que resolver ecuaciones y resolver otros objetivos merece el mismo tratamiento respecto al mecanismo de deducción; de esta forma, las dos componentes del lenguaje, lógica y ecuacional, son soportadas por un mecanismo de inferencia único, SLD-resolución. Esta situación puede considerarse simétrica de la que se plantea en el caso de la aproximación que hemos llamado ecuacional + lógica. Además, tampoco hay necesidad de modificar el algoritmo de unificación tradicional. Otro aspecto especialmente atrayente de esta aproximación es el hecho de que se permite que predicados y funciones sean definidos en una forma mutuamente recursiva en un mismo programa; es decir, la igualdad y otras relaciones pueden aparecer en cualquier punto dentro de una cláusula. Para una

disertación acerca de la técnica de flat-SLD-resolución comparada con "narrowing", consultar [Bosco et al 91] y [Hölldobler 89].

La clase de lenguajes con semántica operacional por flat-SLD-resolución incluye EUROPA [Alpuente Ramírez 90], LEAF [Barbutti et al 86], K-LEAF [Giovannetti et al 91, Levi et al 87] y el lenguaje en [Tamaki 84], basado en la definición de un predicado de reducibilidad. K-LEAF es el núcleo de un sistema más general, el lenguaje IDEAL [Bellia et al 87, Bosco Giovannetti 86], que se compila a K-LEAF y que incorpora tipos y polimorfismo, orden superior y ejecución paralela. En [Deransart 83] y [van Emden Yukawa 87] también se define una transformación similar a la de "flattening", donde el aplanamiento se basa en la compilación de las funciones en predicados introduciendo un argumento adicional para representar el resultado y donde, a diferencia de K-LEAF (que utiliza una estrategia "outermost" perezosa), se adopta una estrategia "innermost". En el lenguaje SLOG [Fribourg84b, Fribourg 85b] también se utilizan ecuaciones en forma plana con una estrategia "innermost" pero, como ya se ha comentado, el sistema de inferencia es diferente.

ιι.2) SLDE-resolución (resolución ecuacional):

En esta aproximación se mantiene SLD-resolución como la única regla de inferencia, pero se reemplaza el algoritmo de unificación sintáctica tradicional por unificación semántica [Siekmann Szabo 82] (a veces llamada en la literatura unificación ecuacional, universal, generalizada, extendida o *E*-unificación). Esto supone utilizar, separadamente, ecuaciones condicionales (para la definición de funciones) y cláusulas (para la definición de predicados). Esta sintaxis, conocida como "well-behaved", admite un procedimiento para la enumeración de un conjunto completo de *E*-unificadores de una ecuación (informalmente, aquél tal que cualquier otro *E*-unificador de la ecuación puede obtenerse como una instancia (semántica) de alguno de los elementos del conjunto) [Hölldobler 90] y define la clase de programas más grande tal que es posible usar SLDE-resolución como base para el desarrollo de un intérprete eficiente. La inclusión de igualdad para clases de programas más grandes parece comprometer irremediablemente la eficiencia [Gallier Raatz 89]. Algunos de los trabajos donde se elaboran técnicas de SLDE-resolución son [Bosco et al 87, Gallier Raatz 86, Gallier Raatz 89, Goguen Meseguer 86, Hölldobler 89, Jaffar et al 84a, Jaffar et al 86a].

En el siguiente ejemplo, tomado de [Palamidessi 88], se ilustran algunos de los principales problemas que se presentan en esta aproximación, examinando el caso de

superponer a un programa lógico puro una teoría ecuacional de primer orden $E$ (cfr. §3.1).

**Ejemplo 2.1.3.1.** Consideremos el programa $P=\{p(a)., p(b)., q(x) \leftarrow p(f(x)).\}$, y supongamos que en $E$ se tiene $\{f(b)=a\}$. Una semántica adecuada para $P \cup E$ debe satisfacer los siguientes requisitos:

- declarativamente, $q(b)$ es una consecuencia lógica del programa $P \cup E$. Una noción eventual de modelo mínimo debe representar este hecho.
- operacionalmente, $\leftarrow q(b)$ debe dar lugar a una refutación.

El primer problema puede resolverse, por ejemplo, considerando la intersección de todos los modelos de Herbrand del programa que, además, sean modelo de los axiomas de la igualdad (intersección conocida como *E-modelo* de Herbrand mínimo) [Chang Lee 73, Fribourg 84b, Hölldobler 89]. Desafortunadamente, esta caracterización no preserva una propiedad semántica importante de la programación lógica: no admite un dominio canónico de interpretación y una asignación funcional fija [Hölldobler 89]. Si la teoría ecuacional es una teoría ecuacional de Horn (ver §3.1), es bien conocido que admite una congruencia menor o más fina $\equiv_E$. Una posibilidad mejor, en este caso, es considerar sólo los modelos que tienen como dominio el universo de Herbrand $H$ módulo la congruencia $\equiv_E$ asociada a la teoría $E$. Dicho dominio se denota como $H/E$ y los correspondientes modelos se conocen como $H/E$-modelos de Herbrand) [Jaffar et al 84a, Jaffar et al 86a, Hölldobler 89]. Dado que la propiedad de intersección también se cumple para estos modelos, la semántica del programa lógico-ecuacional puede definirse como el $H/E$-modelo de Herbrand mínimo.

El segundo aspecto requiere extender el concepto de unificación ya que, claramente, la regla de SLD-resolución tradicional, con unificación sintáctica, aplicada a $\leftarrow q(b)$ no puede tener éxito. La unificación semántica o generalizada es el problema general de unificar un par de términos en presencia de una teoría ecuacional (i.e., unificación en un contexto en el que dos términos que no son sintácticamente idénticos pueden ser considerados iguales bajo la teoría ecuacional). La teoría se describe, usualmente, por medio de un conjunto de ecuaciones. El unificador semántico de dos términos $s$ y $t$ es una sustitución $\upsilon$ tal que $E \models s\upsilon = t\upsilon$ [10] (ver, e.g., [Siekmann Szabo 82, Siekmann 89]).Diremos también que $\upsilon$ es un $E$-unificador o una solución de la ecuación $s = t$. Informalmente, $\upsilon$ se dice maximal si no existen unificadores (semánticos)

[10] En lo que sigue, consideramos siempre que las interpretaciones obedecen los axiomas de la igualdad; por tanto, satisfacibilidad y consecuencia lógica se asumen definidas con respecto a estos axiomas.

más generales. Para una revisión en profundidad sobre unificación extendida, consultar [Dincbas van Hentenryck 87, Siekmann Szabo 82, Siekmann 89].

En [Jaffar et al 84a, Jaffar et al 86a, Jaffar Stuckey 86a] se demuestra que el conjunto de éxitos de la SLDE-resolución para un programa lógico $P$ extendido con una teoría ecuacional $E$ es igual al menor $H/_E$-modelo del programa. En [Gallier Raatz 89] se da también un resultado de completitud para la SLDE-resolución, que sirve en el caso en el que $E$ sea una teoría ecuacional no condicional, y se ilustran las implicaciones computacionales de incluir la igualdad para ciertas clases más generales de cláusulas de Horn con igualdad. Los resultados en [Hölldobler 90] extienden ambos resultados mostrando, para el caso condicional, la independencia respecto de la función de selección (un resultado que fue ya conjeturado en [Beierle Pletat 87]).

El principal problema computacional en esta aproximación es que el conjunto de $E$-unificadores de un par de términos es sólo semidecidible (incluso para teorías no condicionales y canónicas, ver §3.1) de manera que el proceso de unificación semántica puede no terminar y que, en general, no es posible encontrar una representación finita del conjunto de todos los posibles unificadores semánticos de dos términos (es decir, el conjunto de unificadores de máxima generalidad de dos términos puede ser infinito). Esto, sin embargo, no compromete la posibilidad de tener un sistema de inferencia completo: si se dispone de un procedimiento efectivo para generar todas las $E$-soluciones de las ecuaciones (como "narrowing" o flat-SLD-resolución), basta estar atentos a combinar adecuadamente este procedimiento con el procedimiento habitual de generación de las sustituciones de respuesta computada de la SLD-resolución. Claramente, no se puede efectuar enteramente lo primero en cada paso de derivación. Este problema se resuelve elegantemente en el marco de programación lógica con restricciones, como veremos en los próximos capítulos.

La clase de lenguajes con semántica por SLDE-resolución incluye Aflog [Shin et al 87], Denali [Zachary 88], ELP [Hölldobler 87], EQLOG [Goguen Meseguer 86], FUNLOG [Subrahmanyam You 86], LPG [Bert Echahed 86], Unicorn [Bandes 84] y los lenguajes en [Kornfeld 86], [Hölldobler 88] y [Yamamoto 87].

En ocasiones puede resultar muy deseable aislar el problema de la unificación ecuacional de la regla de resolución, para explotar así una variedad de algoritmos conocidos para distintos problemas de unificación ecuacional (optimizando, de esta forma, la implementación del lenguaje, como se propone en [Zachary 88]). Aunque generalmente se usa "narrowing" para obtener el efecto de la $E$-unificación, algunos

lenguajes definen sus propias versiones del algoritmo de *E*-unificación. Es el caso del algoritmo de "unificación semántica" definido en FUNLOG [Subrahmanyam You 84], el de "unificación canónica" de Aflog [Shin et al 87], el de "residuation" de Le Fun [Aït-Kaci et al 87] o el de "unificación extendida" presentado en la definición del lenguaje FHCL en [Furbach Hölldobler 86]. Ninguno de estos mecanismos instancia variables lógicas durante la computación funcional.

Las aproximaciones más relevantes al problema de computar el conjunto de *E*-unificadores de dos términos son:

- flat-SLD-resolución [Bosco et al 88].
- paramodulación [Robinson Wos 69] o alguna forma especial de la misma, tal como superposición [Fribourg 85a] o "narrowing" [Hullot 80, Nutt et al. 89].
- conjuntos completos de transformaciones [Dershowitz Sivakumar 88, Gallier Snyder 87, Hölldobler 90, Kirchner 84, Martelli et al 86].

La última propuesta está basada en la transformación de un conjunto de ecuaciones mediante una extensión de la regla introducida en la tesis de Herbrand y posteriormente utilizada en [Martelli Montanari 82] para el cálculo del mgu de un par de expresiones $t_1$ y $t_2$. Este cálculo se considera equivalente a transformar el conjunto de ecuaciones $\{t_1 = t_2\}$ en un sistema en forma resuelta que representa explícitamente el unificador. En la extensión al caso ecuacional, se incorporan las reglas de la propuesta original (reglas de unificación de tres tipos: reglas de descomposición de términos, reglas de eliminación de variables y reglas de eliminación de ecuaciones triviales) y otras nuevas, cuya aplicación puede ser perezosa, pudiéndose incorporar estrategias de optimización, como normalización y reglas de rechazo. Si la transformación de "flattening" puede verse, básicamente, como la aplicación de los axiomas de la sustitutividad en tiempo de compilación, las reglas de transformación los aplican explícitamente en tiempo de ejecución.

### 2.1.4. Síntesis de las aproximaciones tradicionales.

A modo de síntesis, los métodos computacionales que se han propuesto para la ejecución de lenguajes basados en lógica de cláusulas de Horn con igualdad están basados, en general, en refinamientos lineales de los procedimientos de resolución y de complección (i.e., SLD-resolución y "narrowing", respectivamente). Entre ellos destacan "narrowing" condicional, flat-SLD-resolución y SLDE-resolución. Como hemos visto,

los dos primeros resultan equivalentes y pueden utilizarse para obtener el efecto de la unificación semántica en la tercera aproximación.

En cuanto a la semántica declarativa, se puede dar en cualquiera de los estilos comentados en la subsección §1.1.1. Claramente, en aquellas propuestas que provienen de la programación ecuacional de primer orden, la semántica declarativa suele definirse en un estilo algebraico. Es el caso, e.g., de EQLOG [Goguen Meseguer 86] y RAP [Hussmann 86]. La formalización rigurosa de la completitud del mecanismo operacional de EQLOG puede encontrarse en [Gallier Raatz 89, Hölldobler 89]. En las aproximaciones que parten de la tradición en programación lógica, intentando extender un lenguaje "a la Prolog" (o, más generalmente, un lenguaje HCL) con características funcionales, la semántica declarativa es una semántica por teoría de modelos clásica (que extiende los conceptos de interpretación, modelo y consecuencia lógica para dar cuenta de la relación de igualdad, preservando siempre las buenas propiedades de los programas Horn puros: propiedad de intersección de modelos, existencia de un modelo mínimo y caracterización por punto fijo). Es el caso de la semántica de SLOG [Fribourg 84b, Fribourg 85b], la del lenguaje en [Hölldobler 87] y la definida en [Jaffar et al 84a, Jaffar et al 86a] (que se comentó en la subsección anterior y que será revisada en profundidad en la subsección §2.2.3.1), donde se da una caracterización genérica muy elegante de los programas lógicos enriquecidos con una teoría ecuacional. Cuando se parte de una tradición en programación funcional de orden superior, la semántica declarativa es típicamente denotacional (como en los lenguajes que se definen en [Reddy 85, Reddy 87] y [Darlington et al 86]). Cuando se pretende definir una semántica capaz de dar cuenta de funciones no estrictas (es decir, funciones que pueden terminar incluso sobre argumentos no definidos totalmente, por ejemplo sobre valores que provienen de funciones que no terminan) se deben considerar como dominios de interpretación órdenes parciales completos (cpo's). Es el caso de los lenguajes BABEL [Moreno Rodríguez 88, Moreno Rodríguez 89] y K-LEAF [Giovannetti et al 91, Levi et al 87], basados en una aproximación mixta que define una semántica de modelo mínimo que adopta algunos de los principios fundamentales de la semántica denotacional ("cpo's a la Scott" en lugar de conjuntos sin estructura). Las clases de modelos resultantes son más complejas que las que se definen mediante la caracterización en [Jaffar et al 84a, Jaffar et al 86a] pero también la semántica es más rica, en el sentido de que permite dar cuenta de algunas de las características del mecanismo computacional (e.g., computaciones que no terminan y evaluación perezosa).

En opinión de la autora de esta tesis, de entre todos los lenguajes revisados merecen especial consideración BABEL, EQLOG, K-LEAF, RAP y SLOG por el

sencillo y riguroso modelo que ofrecen, acompañado de resultados formales de corrección y completitud. La siguiente tabla resume las características más importantes de dichos lenguajes en relación a este trabajo, a la vez que ofrece una comparación entre ellos (para una comparación de otras características, como tratamiento de la negación, disciplina de constructores, tipos, orden superior, evaluación perezosa, eficiencia o aspectos de implementación, ver [Moreno 89]). Se ha incluido el lenguaje CLP(*H*/*E*), objeto de este trabajo, para una primera comparación con las aproximaciones clásicas, aunque este punto será examinado en profundidad en los próximos capítulos.

| | BABEL | EQLOG | K-LEAF | RAP | SLOG | CLP(*H/E*) |
|---|---|---|---|---|---|---|
| Notac. | reglas condicionales | cl. Horn + ecs. no cond. | cl. Horn con = | reglas condicionales | teoría ec. Horn | cl. Horn + teoría ec. Horn |
| Sem. Oper. | nwing. cond. perezoso | SLDE-res. | outermost flat-SLD-res. | nwing. cond. full/innermost | superposición clausal innerm. | SLD-res. + constr. solving |
| Sem. Declar. | modelo minimo sobre cpo's | algebraica | modelo mínimo sobre cpo's | algebraica | *E*-modelo mínimo de Hb. | $H/E$ -modelo mínimo de Hb. |

## 2.2. Programación lógica con restricciones.

Una restricción sobre un dominio de computación dado puede entenderse como una formulación declarativa de una relación entre objetos del dominio y también como un mecanismo computacional para hacer cumplir dicha relación. La programación lógica con restricciones (en inglés "Constraint Logic Programming", CLP) es un marco general, un esquema genérico, para la introducción de restricciones en programación lógica (ver [Lassez 87] para una presentación informal). El esquema CLP proporciona un contexto unificado en el que pueden estudiarse diferentes extensiones a la programación lógica pura, como características de orientación a objetos [Aït-Kaci Podelski 90, Beringer Porcher 89], negación constructiva [Stuckey 91a, Stuckey 91b], tratamiento de la relación de igualdad [Alpuente Falaschi 91, Alpuente et al 91, Darlington et al 91], aritmética real [Jaffar et al 88] o términos infinitos y desigualdades [Colmerauer 84, Jaffar Stuckey 86b].

La programación lógica con restricciones es, actualmente, uno de los campos de mayor actividad en programación lógica (ver, e.g., las actas de las últimas reuniones y conferencias internacionales sobre programación lógica, como ALP, ICFGCS, ICLP,

ILPS, PLILP...[11]). El interés por estas investigaciones se ha iniciado sólo muy recientemente, alrededor del año 1987. En el nuevo modelo, un lenguaje lógico se ve como un lenguaje de restricciones sobre el cual es posible definir relaciones por medio de cláusulas definidas [Höhfeld Smolka 88]. Empleando diferentes lenguajes de restricciones es posible definir diferentes lenguajes de programación lógica. Por ejemplo, la programación lógica tradicional se obtiene utilizando un lenguaje de ecuaciones que se interpretan sobre el álgebra de términos sin variables (o universo de Herbrand).

Esta sección consiste en una breve exposición de tres aspectos básicos de la programación lógica con restricciones. En el nivel teórico discutiremos en qué sentido CLP es un esquema y cómo, consecuentemente, las diferentes instancias del mismo heredan sus propiedades semánticas, las mismas de que disfrutan los programas lógicos tradicionales. En los niveles de aplicación e implementación se revisarán los factores a considerar cuando se seleccionan algoritmos para verificar la satisfacibilidad de las restricciones en combinación con un sistema CLP general. Comenzaremos con una revisión del trabajo reciente desarrollado en el campo del diseño, definición semántica e implementación de lenguajes CLP y su relación con nuestro trabajo.

### 2.2.1. De unificación a satisfacción de restricciones.

Si la programación lógica puede ser vista desde dos perspectivas, una de ellas relacionada con la lógica matemática y las técnicas de demostración automática de teoremas, la otra vinculada al desarrollo y uso de lenguajes de programación basados en la lógica, el creciente interés por la programación lógica mediante restricciones puede ser entendido, similarmente, desde este doble punto de vista [Cohen 90]. En el contexto de la lógica matemática, CLP representa el esfuerzo por establecer una clase de teorías de primer orden que preservan las propiedades computacionales básicas de la lógica de cláusulas de Horn. Desde la perspectiva de los lenguajes de programación, el propósito es establecer una clase genérica de lenguajes lógicos donde la computación se realiza directamente sobre diferentes dominios de interés, como el de los reales, los booleanos, los conjuntos regulares, las listas o los árboles [Colmerauer 90, Jaffar Michaylov 87, Walinsky 89]. Esta flexibilidad contrasta con la rigidez del modelo tradicional, donde la computación se realiza siempre sobre un dominio simbólico artificial, el universo de Herbrand.

[11] ALP: Int'l Conf. on Algebraic and Logic Programming, ICFGCS: Int'l Conf. on Fifth Generation Computer Systems, ICLP: Int'l Conf. on Logic Programming, ILPS: Int'l Logic Programming Symp. (antes llamado the North American Conf. on Logic Programming, NACLP), PLILP: Int'l Symp. on Principles of Programming Language Implementation and Logic Programming.

El esquema CLP viene a establecer que la programación lógica no depende en ningún modo esencial de la unificación ni del universo de Herbrand. Esto se entiende fácilmente si se tiene en cuenta que la unificación tiene dos aspectos que conviene diferenciar. Uno de ellos es que constituye un procedimiento de decisión que revela si una ecuación $s = t$ es satisfacible. El otro simplifica la ecuación, transformándola en una ecuación en forma resuelta que equivale a un unificador idempotente (el mgu), que representa explícitamente el conjunto de todas sus soluciones [Lassez et al 88]. Pero es lícito argumentar que la ecuación $s = t$ también representa el conjunto de todas sus soluciones. Se puede considerar el mgu como una representación explícita de todas sus soluciones y la ecuación $s = t$ como una representación implícita equivalente del mismo conjunto. ¿Por qué, entonces, es necesario el mgu?. Lo es, básicamente, por conveniencia sintáctica; e.g. es más fácil manejar $X = a$ que $g(X,X) = g(a,a)$. Pero es posible renunciar a la simplicidad sintáctica sin afectar a los resultados de corrección y completitud, punto fijo y menor modelo ni a sus demostraciones. El único precio a pagar es que la salida del programa lógico es ahora un sistema no simplificado de ecuaciones, en vez de un mgu, que puede resultar más conciso. Así que lo fundamental en el concepto de unificación es el aspecto satisfacibilidad, no su aspecto mgu, que puede ser interpretado como una mera cuestión de implementación. Este punto resuelve una debilidad fundamental de la programación lógica tradicional, ya que para muchos dominios de interés no existe la mencionada representación explícita del conjunto de soluciones a una ecuación (y, en general, a una restricción). La capacidad de devolver respuestas simbólicas es una característica importante de los lenguajes CLP. Responder a una pregunta se entiende en términos de solubilidad de restricciones y no de devolver un valor. En el nuevo esquema, por tanto, la computación se entiende como la verificación de la solubilidad en el modelo.

Como ya se ha mencionado, uno de los principales requisitos de un buen lenguaje de programación es el de disponer de una semántica definida en modo sencillo y con un buen soporte formal. En el paradigma de programación lógica con restricciones se preservan las propiedades semánticas únicas de los lenguajes lógicos tradicionales que los hacen tan simples y potentes; es decir, la existencia de varias semánticas (operacional, teoría de modelos y punto fijo) que no sólo resultan simples y elegantes sino que, además, coinciden. Más aún, dado que la computación se realiza sobre un dominio particular, los programas gozan adicionalmente de una semántica algebraica que complementa las anteriores; es decir, una semántica definida directamente sobre la estructura algebraica del dominio. Trabajar sobre álgebras específicas puede resultar muy intuitivo y las diferentes implementaciones del modelo pueden beneficiarse ampliamente

de la extensa literatura referente al problema de la resolución de restricciones sobre diferentes estructuras. La programación lógica con restricciones debe, por tanto, entenderse como una generalización de la programación lógica tradicional, en el sentido de que se preservan sus propiedades semánticas fundamentales sin la restricción al universo de Herbrand ni tampoco a las igualdades entre términos de dicho dominio, que no son sino una clase particular de restricciones, ni siquiera a la noción de unificación, que puede verse como un caso muy especial de resolución de restricciones.

### 2.2.2. Programación con restricciones.

Independientemente de la programación lógica, la utilidad de las restricciones ha sido probada en gran variedad de aplicaciones, incluyendo planificación, análisis y diseño de dispositivos mecánicos y circuitos eléctricos, simulación de experiencias físicas, investigación operativa, técnicas de representación gráfica,... [Freeman-Benson et al 90, Leler 87, Stefik 84]. En el campo de la programación lógica, algunos de los trabajos pioneros más conocidos sobre restricciones son [Steele Sussman 79] y [Borning 81]. En el primero se introduce una variante de las técnicas de evaluación perezosa, conocida como propagación local de restricciones, que define una estrategia de reordenación, en tiempo de ejecución, de un sistema de restricciones para encontrar una secuencia de asignaciones que hace "ground" todas las variables del sistema. El trabajo de Borning se sitúa en un terreno más aplicado; en él se definen jerarquías de restricciones y su tratamiento, con aplicaciones a sistemas gráficos y simulación. También es conocido que muchas de las características frecuentemente utilizadas en algunos lenguajes CLP, como la eliminación del test del "occur-check", la verificación de la satisfacibilidad de un sistema de ecuaciones o el no determinismo, estaban ya presentes en el lenguaje Absys, desarrollado a finales de los años 60 (ver [Elcock90]).

Pero es comúnmente aceptado que el primer intento de introducción de restricciones en un lenguaje lógico es el realizado por Colmerauer con el diseño e implementación de Prolog II [Colmerauer 82], un sistema basado en la resolución de ecuaciones e inecuaciones (que no son sino otra clase particular de restricción) sobre árboles infinitos racionales. Esto permite expresar directamente en el objetivo derivado en un paso de computación la igualdad que debe ser verificada entre los argumentos del átomo seleccionado (en el objetivo precedente) y los de la cabeza de la cláusula utilizada. Prolog II puede ser entendido como un lenguaje lógico que manipula estructuras infinitas, como un Prolog estándar sin "occur-check" [Lloyd 87]. En [van Emden Lloyd 84] se demuestra (mediante la técnica de transformación a la forma homogénea revisada en la subsección §2.1.3) la corrección del lenguaje respecto a la clase de modelos que son

consistentes con una cierta teoría de la igualdad en la que las relaciones recursivas tienen siempre una solución. Posteriormente, [Jaffar et al 84b, Jaffar et al 86b] han demostrado que esta clase de modelos admite un modelo mínimo, que está basado en un universo de Herbrand extendido con términos infinitos. Este modelo puede ser tomado, por tanto, como la definición de la semántica declarativa de Prolog II. En [Jaffar et al 86b] se demuestra la completitud del lenguaje respecto a esta semántica.

### 2.2.3. El esquema CLP de Jaffar y Lassez.

Jaffar y Lassez fueron los primeros en identificar el nuevo modelo. En [Jaffar Lassez 86, Jaffar Lassez 87] desarrollaron un marco general, un esquema genérico acuñado como CLP($\chi$), que se parametriza respecto al dominio de computación deseado, y demostraron resultados sobre la corrección y la completitud para una semántica operacional genérica (abstrayendo, obviamente, de la corrección y completitud del cálculo de las soluciones de las restricciones). Cada instancia del esquema es un lenguaje de programación que se obtiene especificando una estructura $\chi$ de computación. La estructura define el dominio de discurso subyacente (que, por abuso de notación, también se denota como $\chi$) y las operaciones y relaciones sobre este dominio, dando así una interpretación semántica al mismo. Las diferentes implementaciones de una instancia del esquema pueden diferir por la elección de algoritmos de resolución de restricciones específicos.

La programación lógica tradicional puede verse como una instancia de este esquema, donde la correspondiente estructura toma como dominio de interpretación el universo de Herbrand para el programa y donde el único símbolo de predicado es la igualdad, que se interpreta como identidad sintáctica sobre el dominio. El algoritmo de unificación es visto ahora como un algoritmo de resolución de restricciones especializado en verificar la solubilidad de ecuaciones sobre el universo de Herbrand.

#### 2.2.3.1. El esquema de lenguaje de programación lógica.

CLP($\chi$) tiene sus orígenes en un esquema previo, el "esquema de lenguaje de programación lógica" [Jaffar et al 84a, Jaffar et al 86a, Jaffar Stuckey 86a] que, como se adelantó en la subsección §2.1.4, captura las propiedades semánticas esenciales de los programas lógicos extendidos con teorías ecuacionales. Un programa lógico en el esquema se ve como un par *(P,E),* donde *P* es un conjunto de cláusulas de Horn definidas y *E* es una teoría de la igualdad, también presentada mediante un conjunto de cláusulas de Horn definidas. La sintaxis del esquema es, por tanto, la sintaxis de HCL y su

dominio de computación se deja sin especificar aunque se asume que puede ser definido mediante una teoría ecuacional *completa respecto a la unificación*. Una teoría ecuacional $E$ se dice completa respecto a la unificación si cada posible solución de una ecuación dada puede ser representada mediante un $E$-unificador de la ecuación. Es decir, si para toda ecuación $s = t$, $E \models ((s = t) \rightarrow \bigvee_{\theta} \exists(\hat{}, \theta_i))$, donde $\{\exists(\hat{}, \theta)_i\}$ es el conjunto (eventualmente vacío o infinito) de $E$-unificadores de $s$ y $t$, vistos como un conjunto de ecuaciones, y $\bigvee_i$, $i \in \{0, 1, ..., \omega\}$, representa la disyunción de tales ecuaciones. Si el conjunto es vacío, la disyunción se asume falsa. Informalmente, este requerimiento significa que en cualquier modelo de $E$ se cumple la siguiente propiedad: si una valoración de las variables que aparecen en los términos $s$ y $t$ es tal que $s = t$ en el modelo, entonces uno, al menos, de los unificadores (vistos como conjuntos de ecuaciones) es también verdad en ese modelo con esa misma valoración. Consecuentemente, cuando no hay $E$-unificador, $E \models \neg(s=t)$. El intérprete de los lenguajes que se obtienen como instancia de este esquema se basa en SLD-resolución con un algoritmo de unificación generalizada apropiado, $E$-unificación, correspondiente a una igualdad semántica dependiente del problema. Las propiedades semánticas fundamentales de las cláusulas definidas se cumplen para el esquema y todas sus instancias.

Esta generalización se basa en la observación de que muchas de las extensiones de la programación lógica tradicional se pueden obtener reemplazando el dominio clásico de computación simbólica (el universo de Herbrand), basado en la identidad sintáctica, por una estructura axiomatizada mediante una adecuada teoría ecuacional $E$. Ahora, en lugar de establecer uno a uno los varios resultados semánticos para una extensión dada de la programación lógica tradicional, pueden obtenerse todos en un movimiento. Prolog II puede verse como una instancia de este esquema desde el momento que puede definirse una teoría ecuacional que describe el dominio de los términos infinitos racionales, es decir, de las soluciones de las ecuaciones recursivas [Jaffar et al 86b]. Esencialmente, se procede en dos pasos: primero se define la teoría ecuacional, cuyo modelo estándar es el dominio de los árboles racionales, y después se muestra que esta teoría es completa respecto a la unificación.

La propiedad fundamental que debe satisfacer la teoría ecuacional para que se cumpla el teorema de existencia del modelo mínimo es la de admitir una congruencia mínima o más fina; es decir, la relación definida como:

$$s \equiv_E t \text{ sii } E \models s = t$$

debe ser una congruencia. Dos términos son considerados iguales si pertenecen a la misma clase de $\equiv_E$-*equivalencia* Muchas teorías satisfacen esta condición; en particular, las teorías de Horn ecuacionales y las teorías de la igualdad de Prolog y Prolog II. En tales casos, se define la estructura $H/E$ (el conjunto de las clases de equivalencia de $\equiv_E$ sobre el universo de Herbrand $H$ del programa), cuyo dominio es el dominio de interpretación deseado, y se generaliza el concepto de interpretación y de modelo en el modo natural ($H/E$−interpretaciones y $H/E$-modelos). Estos modelos juegan el papel de los modelos tradicionales en programación lógica convencional. En [Jaffar et al 86a] se demuestra que para cualquier programa $P$ y cualquier átomo "ground" $A$ se cumple una propiedad análoga al teorema de Löwenheim-Skolem para modelos de Herbrand:

$$(P,E) \models A \text{ sii } P \models_{H/E} A$$

es decir, la verdad en los modelos equivale a la verdad en los $H/E$−modelos.

Llegado este punto, la construcción es análoga a la del caso estándar:

- se demuestra la existencia del $H/E$−modelo mínimo, sobre el que se define la semántica de $P$.
- se define una transformación $T(P,H/E)$ sobre las $H/E$-interpretaciones que se demuestra continua y cuyo mínimo punto fijo corresponde al $H/E$−modelo mínimo.

En lo referente al aspecto computacional se demuestra, para el esquema y para todas sus instancias, la corrección y completitud de la regla de SLD-resolución, con el algoritmo de unificación sintáctica sustituido por $E$-unificación, respecto de la semántica declarativa del menor $H/E$−modelo. El uso de la negación como fallo finito para obtener información negativa de un programa lógico generalizado requiere la oportuna generalización del concepto de programa lógico completo, lo que se obtiene definiendo la complección $(P^*, E^*)$ de un programa lógico extendido $(P,E)$. En el programa completo, $P^*$ se obtiene usando la transformación de Clark [Clark 78] mientras que $E^*$ es una extensión completa respecto a la unificación que extiende a $E$. Por ejemplo, la teoría de la igualdad de Clark[12] es una extensión completa respecto a la unificación de la teoría de la igualdad sintáctica con los axiomas habituales. El esquema incluye resultados de

[12] formada por los axiomas: $c(X_1, X_2, ..., X_n) \neq d(Y_1, Y_2, ..., Y_m)$.
$c(X_1, X_2, ..., X_n) = c(Y_1, Y_2, ..., Y_n) \rightarrow X_1 = Y_1, X_2 = Y_2, ..., X_n = Y_n$.
$X \neq t(X)$.
donde $c$ y $d$ son símbolos de constructor distintos, de aridades $n$ y $m$, respectivamente, y $t(X)$ es un término no variable conteniendo una ocurrencia de la variable $X$.

corrección y completitud para las derivaciones con éxito (respectivamente, con fallo finito) respecto a los átomos "ground" positivos (respectivamente, negativos) que son consecuencia lógica del programa completado $(P^*, E^*)$.

### 2.2.3.2. El esquema de Programación Lógica con restricciones (CLP).

Mientras el esquema anterior logra el objetivo de englobar, en un marco unificado, lenguajes lógicos cuyos dominios pudieran ser definidos mediante una teoría ecuacional, está íntimamente ligado y, por tanto, restringido a la unificación. Sin embargo, cuando se trabaja al margen de la unificación y de las ecuaciones no es, en principio, tan aparente que un esquema similar exista. Por ejemplo, el problema de las inecuaciones en Prolog II no puede ser acomodado en el viejo esquema. Consecuentemente, en [Jaffar Lassez 86, Jaffar Lassez 87] se considera una extensión del esquema, donde el concepto de unificación (generalizada) se reemplaza por el concepto de resolución de restricciones. Además de la ventaja de ser más general, los lenguajes CLP satisfacen tres criterios que hacen de un lenguaje de programación lógica un lenguaje útil: tienen un buen soporte formal (del que se heredan las propiedades semánticas esenciales), un alto poder expresivo (es posible trabajar directamente en el dominio en el que se sitúa el discurso) y es posible desarrollar implementaciones eficientes (que se benefician del uso de técnicas conocidas para resolución de restricciones en el dominio de computación específico seleccionado).

Informalmente, un programa CLP consta de un conjunto finito de cláusulas de la forma $H \leftarrow c \ \square$. o $H \leftarrow c \ \square \ B_1, \ldots, B_n$., donde $H$ (la cabeza) y $B_1, \ldots, B_n$ (el cuerpo), $n \geq 0$, son átomos y $c$ es una conjunción finita (posiblemente vacía) de restricciones. Un objetivo es una cláusula sin cabeza. El símbolo $\square$ se utiliza sólo para diferenciar las dos partes y debe interpretarse como la conectiva lógica de conjunción. La posibilidad de incluir restricciones en los objetivos y en los cuerpos de las cláusulas incrementa sustancialmente la potencia expresiva (podría incluso compararse a la introducción de objetivos negados en el cuerpo de las cláusulas de un programa de cláusulas definidas).

La semántica declarativa de un programa CLP puede verse tanto en términos de consecuencia lógica como con un estilo algebraico. La respuesta a un objetivo $G$ no es una sustitución, sino una conjunción de restricciones $c$ tal que:

$$(P, \mathfrak{I}) \models (\forall)(c \Rightarrow G) \qquad \text{(versión lógica)}$$
$$P \models_{\mathfrak{R}} (\forall)(c \Rightarrow G) \qquad \text{(versión algebraica)}$$

donde $P$ es un programa CLP, $\mathcal{R}$ la estructura y $\mathcal{T}$ es una teoría que axiomatiza $\mathcal{R}$. La teoría CLP impone restricciones sobre $\mathcal{R}$, $\mathcal{T}$ y sus relaciones para establecer equivalencias entre las semánticas (cfr. §3.1).

El intérprete CLP está otra vez basado en SLD-resolución y un apropiado algoritmo de resolución de restricciones (un algoritmo que (semi-)decide la satisfacibilidad de las mismas) en el dominio de computación correspondiente. Definir un lenguaje de esta clase equivale a especificar el dominio de computación y construir el algoritmo de resolución de restricciones apropiado a dicho dominio. Intuitivamente, una computación CLP puede verse como una secuencia de pasos de reducción en la que se acumulan restricciones (posiblemente simplificadas), previa comprobación de su satisfacibilidad. La computación termina con un conjunto satisfacible de restricciones.

El nuevo esquema garantiza también que las propiedades semánticas fundamentales de la programación en lógica de cláusulas definidas son preservadas en todas sus extensiones. La existencia de un dominio de computación canónico, la semántica de menor y mayor modelo, la existencia de una semántica de menor y mayor punto fijo, los resultados de corrección y completitud para derivaciones con éxito y los resultados de corrección y completitud para derivaciones que fallan finitamente y para la regla de negación como fallo finito son heredados por cualquier extensión que pueda ser formalizada como una instacia del esquema. Por otra parte, en [Gabbrielli Levi 91a, Gabbrielli Levi 91b] se define, como generalización al caso CLP de los resultados en [Falaschi et al 88, Falaschi et al 89], una nueva noción de modelos, que corresponden a diferentes comportamientos observables y que capturan adecuadamente la semántica operacional de los programas CLP. Estos modelos, aún no siendo minimales (en el caso general), pueden obtenerse como puntos fijos de operadores de consecuencia inmediata continuos apropiadamente definidos. En particular, existe un operador consecuencia inmediata, definido en términos de restricciones resolubles, cuyo menor punto fijo es un modelo (no minimal) correspondiente a una semántica de *restricción respuesta computada* ("computed answer constraint"). También estos resultados se aplican a cada instancia del esquema.

#### 2.2.3.3. Instancias y extensiones del esquema CLP.

El primer lenguaje explícitamente propuesto como una instancia del esquema CLP fue CLP($\mathcal{R}$) [Jaffar Michaylov 87, Jaffar et al 88] en el que el dominio de computación es el de los reales $\mathcal{R}$ y las restricciones que se manejan son ecuaciones e

inecuaciones lineales. Muchas de las extensiones de Prolog, como Prolog II y Prolog III [Colmerauer 82, Colmerauer 87, Colmerauer 90], pueden también entenderse como instancias del esquema, aunque no fueron desarrolladas como tales. Por ejemplo, Prolog III permite restricciones ecuacionales sobre términos booleanos y ecuaciones lineales, desigualdades e inecuaciones entre términos racionales. La parte racional podría verse como CLP($Q$), aunque no fue presentado originalmente en esta forma. Lenguajes como CLP($\Im^*$) (CLP sobre conjuntos regulares, cuyos elementos son cadenas sobre un alfabeto finito) [Walinsky 89], Cosylog [Beringer Porcher 89] (presentado como CLP(*teoría conceptual*)) o Trilogy [Voda 88] (que puede verse como CLP($Z$)) son instancias diferentes del esquema. El esquema CLP proporciona también una base formal para lenguajes como CAL [Aiba et al 88], ConsLOG [Hao Chabrier 90], CHIP [van Hentenryck 89] o el lenguaje de representación del conocimiento Login [Aït-Kaci Nasr 86], que comentaremos posteriormente. Cabe advertir que, aunque todos estos lenguajes pueden ser acomodados en el marco de CLP, la motivación de los mismos es, claramente, diferente. La característica más significativa común a todos ellos es su gran potencia expresiva, unida a su eficiencia para la resolución de problemas de satisfacción de restricciones (CSP). Las técnicas para resolución de restricciones sobre dominios finitos desarrolladas para CHIP y usadas posteriormente en los lenguajes CHARME [Leconte Lelong 91], Prolog-XT [Büttner et al 90] y PECOS [Albert Puget 91] conducen a una gran eficiencia en ejecución, varios órdenes de magnitud superior a la de Prolog. El punto clave está en que las restricciones se usan de manera activa para reducir la talla del especio de búsqueda (a diferencia del estilo pasivo "generate and test" característico de Prolog), propagando información sobre los dominios de las variables tan pronto como se dispone de ella.

En [Borning et al 89, Freeman-Benson et al 90] se describe un esquema de lenguaje de programación lógica con restricciones (HCLP) que introduce jerarquías de restricciones en el contexto de CLP. El esquema HCLP viene parametrizado tanto por el dominio de las restricciones como por un comparador ($C$) que permite comparar diferentes soluciones y seleccionar la "mejor".

Para que un lenguaje esté formalmente basado en la semántica del esquema de Jaffar y Lassez, la correspondiente estructura debe ser *compacta respecto a las soluciones* [Jaffar Lassez 86, Jaffar Lassez 87] (ver definición técnica en sección §3.1). Informalmente, esta propiedad significa que cualquier elemento del dominio puede ser definido como la única solución de un conjunto (finito o no) de restricciones y que el lenguaje de restricciones es lo bastante preciso como para distinguir un objeto que satisface una restricción de los que no. Todo dominio finito o numerable goza

trivialmente de esta propiedad. Algunos dominios no numerables, como los reales y los árboles infinitos, también la poseen. Según [Jaffar Lassez 87], este requerimiento es muy débil y si un dominio y su lenguaje de restricciones no fueran compactos, entonces no deberían ser considerados para propósitos computacionales, puesto que su sintaxis no permitiría una definición precisa de los objetos. En la sección §2.5, sin embargo, revisaremos el esquema para programación lógica con restricciones de Höhfeld y Smolka, que generaliza al de Jaffar y Lassez y donde no se requiere esta propiedad (lo que, en particular, hace el esquema aplicable a problemas de representación del conocimiento).

Para razonar acerca de la negación se debe extender la teoría $\mathfrak{T}$ que axiomatiza la estructura $\mathfrak{R}$ para hacerla *completa respecto a la satisfacción*. Si ésto no es posible, no se podrán heredar del esquema los resultados correspondientes al fallo finito y a la negación como fallo. Informalmente, esta propiedad significa que ha de ser demostrable tanto la satisfacibilidad como la insatisfacibilidad de las restricciones. Este requerimiento se corresponde con el concepto de *completitud respecto a la unificación* utilizado en el viejo esquema de lenguaje de programación lógica [Jaffar et al 84a, Jaffar et al 86a]. La principal diferencia entre ambos conceptos y la noción lógica clásica de completitud de una teoría es la clase de fórmulas a las que se aplican. En completitud clásica se exige $\mathfrak{T} \models \phi$ o $\mathfrak{T} \models \neg\phi$ para cualquier fórmula $\phi$. En *completitud respecto a la satisfacción* se exige $\mathfrak{T} \models \tilde{\exists}\, C$ o $\mathfrak{T} \models \neg C$; es decir, se aplica sólo a fórmulas de la forma $\exists X_1,X_2,..., X_m\,(c_1 \wedge c_2 \wedge ... \wedge c_n)$, donde las $c_i$, $i = 1, 2, ..., n$, son restricciones. Por tanto, es posible que una teoría sea completa respecto a la satisfacción con respecto a un lenguaje de restricciones dado sin ser la teoría entera decidible. La diferencia entre las propiedades de *completitud respecto a la unificación* y *completitud respecto a la satisfacción* es que la primera requiere que todas las soluciones sean cubiertas por unificadores. Esto hace aconsejable utilizar preferiblemente el nuevo esquema puesto que si una teoría puede producir un conjunto infinito de unificadores de máxima generalidad para un conjunto de ecuaciones dado (e.g., la teoría de una función asociativa), entonces la teoría no es *completa respecto a la unificación*. Esto se debe a que el teorema de compacidad de la lógica de primer orden implica que si una fórmula equivale a una disyunción infinita, también equivale a una disyunción finita. En consecuencia, el axioma que define la propiedad de completitud respecto a la unificación no es expresable en lógica de primer orden [Maher 91].

Para terminar esta sección se presenta una tabla que resume las características más importantes de los principales lenguajes que pueden considerarse como instancia del esquema CLP y que han sido ya comentados. Se ha incluido también el lenguaje

CLP(*H*/*E*), objeto de este trabajo, cuya definición constituye el tema principal de la disertación que se ofrece en los próximos capítulos.

Las características comparadas han sido:

- DI Dominio de Interpretación
- $\Pi_C$ Conjunto de símbolos de predicado para restricciones
- $\Sigma$ Conjunto de símbolos de función
- LR Lenguaje de Restricciones
- ASR Algoritmos de verificación de la solubilidad de las restricciones en el dominio correspondiente

| LENGUAJE | DI | $\Pi_C$ | $\Sigma$ | LR | ASR |
|---|---|---|---|---|---|
| Prolog | Arb. finitos | = | Func. libres | cierre conj.[13] | Alg. unificación |
| Prolog II | Arb. ∞ rac. | = , ≠ | Func. libres | cierre conj. | Alg. forma resuelta |
| Prolog III | Arb. ∞ rac. | = , ≠ | func. libres + constr. árb. | cierre conj. | Alg. forma resuelta |
| | Racionales | = , ≠, <, >, ≤, ≥ | +, -, * | cierre conj. | Simplex incremental |
| | Booleanos | = , ≠, ⇒ | 0,1,¬,v,∧,⇔ | w.f.f | Trad. forma clausal + método saturación |
| | Listas | = , ≠ | constr. listas | cierre conj. | Alg. forma resuelta |
| CLP($\mathcal{R}$) | Arb. finitos | = | func. libres | cierre conj. | Eliminac. Gaussiana |
| | Reales | <, >, ≤, ≥ | +, * | cierre conj. | Simplex incremental |
| CLP($\Sigma^*$) | Conjuntos regulares | In | func. libres + ops. Cjts. regs. | cierre conj. | Cálculo específico |
| CHIP[14] | Arb. finitos | = | func. libres | cierre conj. | Alg. unificación |
| | Racionales | =,≠,<, >, ≤, ≥ | +,-,*,/ | cierre conj. | Elimin. Gaussiana + Simplex incremental |
| | Booleanos | = , ≠ | 0,1,&(and), !(or),#(xor), nand, nor,not | cierre conj. | Alg. unific. booleana |
| | D. finitos (ctes. y *N*) | =,≠, <, >, ≤, ≥ y const. simbolic. | +,-,*,/ (*N*) | cierre conj. | Consistency checking |

[13] cierre conjuntivo del conjunto de restricciones que se pueden formar con los símbolos de $\Pi_C$ y $\Sigma$.

| | | | | | |
|---|---|---|---|---|---|
| CLP(*H*/*E*) | *H* / *E* | = | func. libres + func.definidas | cierre conj. | Algoritmo "nwing" incremental |

### 2.2.4. El esquema de Höhfeld y Smolka.

Recientemente, Höhfeld y Smolka han generalizado el esquema de Jaffar y Lassez para hacerlo aplicable a representación del conocimiento [Höhfeld Smolka 88]. La motivación original del trabajo fue el desarrollo de una fundamentación semántica formal para el lenguaje Login [Aït-Kaci Nasr 86], en el que se definen relaciones sobre un lenguaje de restricciones basado en los llamados Ψ-términos. En el esquema de Smolka no se requiere que el lenguaje de restricciones sea un sublenguaje de la lógica de predicados y puede venir provisto de más de una interpretación. Tampoco se requiere que el dominio subyacente a las diferentes interpretaciones sea compacto respecto a las soluciones.

En [Darlington et al 91] se define una extensión de este esquema, el llamado "paradigma de programación lógica funcional con restricciones, CFLP", y se introduce el lenguaje FALCON (Functional And Logic language with CONstraints) como una instancia concreta del modelo propuesto.

El lenguaje LIFE ("Logic, Inheritance, Functions and Equations") se define como una composición de tres instancias del esquema de Höhfeld y Smolka: clausal, algebraica y teoría de tipos [Aït-Kaci Lincoln 88, Aït-Kaci Podelski 90]. LIFE fue desarrollado con la intención de explorar las ventajas de combinar programación lógica, funcional y orientada a objetos. En consecuencia, el lenguaje tiene tres componentes: la componente funcional está basada en el λ-cálculo, la relacional lo está en un cálculo de primer orden restringido a cláusulas de Horn y la orientada a objetos en el Ψ-cálculo de Aït-Kaci [Aït-Kaci Podelski 90]. Las componentes de LIFE fueron combinadas dos a dos anteriormente en los lenguajes Login, FOOL y Le Fun. Login [Aït-Kaci Nasr 86] puede verse como una extensión de Prolog donde los términos de primer orden se reemplazan por Ψ-términos (con definiciones de tipos) para dar cuenta de la relación de herencia. Similarmente, FOOL [Aït-Kaci Podelski 90] es un lenguaje de programación funcional con Ψ-términos. En el lenguaje Le Fun [Aït-Kaci et al 87] se integra programación lógica y funcional, como ya fue comentado en la subsección §2.1.3. En LIFE, que integra Login, FOOL y Le

[14] El lenguaje CHIP ofrece también la posibilidad de generar restricciones elementales y compuestas a partir de las restricciones primitivas del lenguaje y de programas lógicos. El algoritmo de resolución de restricciones es, intencionalmente, incompleto.

Fun como sublenguajes, es posible especificar tipos, funciones y relaciones. Dentro de las funciones y relaciones se usan Ψ-términos para especificar tipos, subtipos y herencia.

En [Saraswat 89, Saraswat Rinard 90] se desarrolla, partiendo del esquema de Höhfeld y Smolka, la noción de programación lógica concurrente con restricciones (cc). Los lenguajes de esta familia son paramétricos respecto al sistema de restricciones subyacente. En [Gabbrielli Levi 90] se presenta una nueva semántica operacional y por punto fijo para un lenguaje en esta familia, que modela el conjunto de éxitos, bloqueos y fallo finito. Se presenta también un resultado de equivalencia entre las dos semánticas.

Por último mencionar que también en el campo de la demostración automática de teoremas en lógica de primer orden con igualdad se han introducido técnicas de resolución de restricciones a partir del esquema de Höhfeld y Smolka (ver, e.g., [Kirchner et al 91]).

### 2.2.5. Los algoritmos de resolución de restricciones.

Una cuestión fundamental a afrontar en el diseño de un lenguaje CLP es la selección de los algoritmos adecuados para la verificación de la solubilidad de las restricciones en cada dominio. Un requerimiento importante a considerar en esta elección es que la eficiencia en el procesamiento de los programas que no utilizan las facilidades añadidas del lenguaje debería aproximarse a la de los intérpretes de programación lógica convencionales.

En esta sección vamos a introducir varios factores que deben ser considerados en la selección de los algoritmos de resolución de restricciones:

- incrementalidad
- simplificación
- forma canónica

• El primer factor es una propiedad esencial que permite incrementar la eficiencia: El conjunto de restricciones en los objetivos de una derivación aumenta su talla de forma monótona. En cada paso de derivación se demuestra la satisfacibilidad del conjunto de restricciones generadas por el último objetivo resuelto. En el paso siguiente aparecen nuevas restricciones a añadir al conjunto y resulta inaceptable tener que rehacer otra vez todo el trabajo del paso previo. Es decir, una vez que ha sido probada la satisfacibilidad

de una restricción, el problema de satisfacibilidad resultante de añadir una restricción adicional debería resolverse evitando computaciones inútiles, minimizando el esfuerzo computacional requerido para verificar si la fórmula continúa siendo satisfacible o no. El coste de resolver el nuevo conjunto de restricciones debería ser aproximadamente igual al coste de resolver las restricciones añadidas más el coste de "combinar" la nueva solución con la solución anterior. Los intérpretes Prolog clásicos tienen esta propiedad, ya que las unificaciones realizadas previamente no son recomputadas en cada paso de inferencia. Algunas modificaciones del método de Gauss para la resolución de ecuaciones lineales cumplen también esta propiedad. También el método Simplex para la verificación de la satisfacibilidad de inecuaciones y el metodo de SL-resolución [Kowalski Kuehner 71] utilizado para decidir la satisfacibilidad de ecuaciones booleanas pueden ser redefinidos para satisfacer la propiedad de incrementalidad. En [Cohen 90] y [Freeman-Benson et al 90] se revisa el trabajo reciente en esta materia y en [Maher Stuckey 89] se presentan algunas extensiones a la interfaz de interrogación de lenguajes CLP para obtener incrementalidad, examinado el uso de jerarquías de restricciones en este marco.

Por último, comentar que en [Hao Chabrier 90, van Hentenryck 90] se considera en general el problema de la satisfacción incremental de restricciones, mostrando que una aproximación basada en "backtracking" resulta inadecuada, ya que sólo puede usar pasivamente las nuevas restricciones para comprobar si son satisfechas por las nuevas soluciones tentativas generadas. Como alternativa, se presenta un esquema basado en reejecución y poda del árbol de búsqueda de la SLD-resolución.

- El segundo factor requerido, simplificación, hace referencia a la sustitución del conjunto de restricciones otro equivalente pero más simple, lo cual sólo será posible en ciertos dominios. La simplificación consume tiempo, pero puede resultar necesaria. Una decisión importante durante la actividad de diseño de un nuevo lenguaje CLP es definir el nivel de simplificación a efectuar en cada paso de computación, puesto que si un conjunto de restricciones es probado insatisfacible todo el trabajo de simplificación de los pasos anteriores se pierde. La simplificación sólo es absolutamente esencial en el paso final, cuando la restricción computada debe ser mostrada en la forma más clara y legible posible.

- Un problema relacionado con el de la simplificación es el de encontrar una forma concisa y fácilmente manipulable, una forma estándar o canónica, para representar las restricciones de un conjunto dado. La representación (interna) de las restricciones en forma normal facilita tanto el test de la satisfacibilidad como las posteriores simplificaciones. Por tanto, la elección de una noción de forma canónica apropiada es

también una consideración de diseño importante para lenguajes CLP. En esta línea, en [Darlington et al 91] se define el concepto de *algoritmo de resolución de restricciones* como el procedimiento para computar formas resueltas para sistemas de restricciones.

En ciertos dominios es posible representar un conjunto soluble de restricciones en forma resuelta. Por ejemplo, la estructura $H = (\{=\} \cup \Pi_P, \Sigma)$, donde = es el único símbolo de predicado para las restricciones, interpretado como igualdad sintáctica en el universo de Herbrand $H = \tau(\Sigma)$ asociado a un programa $P$, goza de la propiedad de existencia de una forma canónica única (salvo isomorfismo) para las $(\{=\}, \Sigma)$-restricciones. Esta forma canónica es la definida en [Lassez et al 88] como *conjunto de ecuaciones en forma resuelta* y equivale al mgu idempotente del conjunto de restricciones ecuacionales. Esta propiedad permite identificar sintácticamente $(\{=\}, \Sigma)$-restricciones $H$-equivalentes. Para el caso de la estructura $H_{II} = (\{=,\neq\} \cup \Pi_P, \Sigma)$, que corresponde esencialmente a la estructura de Prolog II, se tienen también resultados análogos. Si $\Sigma$ es un conjunto infinito, los resultados presentados en [Common Lescanne 89, Kirchner Lescanne 87] muestran la existencia de una forma canónica para conjuntos de igualdades y desigualdades y dan un algoritmo que la computa. También en dominios aritméticos es posible definir formas canónicas. Por ejemplo, en [Lassez McAloon 88, Lassez McAloon 89] se define una forma canónica para restricciones lineales generalizadas que juega el mismo papel que el del mgu para igualdades interpretadas en el universo de Herbrand.

En el caso del universo de Herbrand cociente por una teoría de la igualdad $=_E$ no existe, en el caso general, una representación explícita finita de una restricción en forma resuelta. Como representación de la forma resuelta se adopta, en este caso, el concepto de *conjunto completo de soluciones,* que ya no permite la identificación sintáctica de restricciones $H/E$-equivalentes. En este caso, habrá que introducir una definición semántica explícita del concepto de equivalencia de restricciones.

En el capítulo tres se tratará el problema de la satisfacción incremental de restricciones ecuacionales y se introducirá una representación simplificada apropiada para restricciones solubles.

## 2.3. Programación lógico-ecuacional con restricciones.

El principal argumento discutido en esta tesis es que en el marco de CLP es posible formalizar la deseada integración entre programación lógica y ecuacional, admitiendo la definición de funciones y proporcionando un tratamiento adecuado de la relación de igualdad. En este marco, la programación lógico-ecuacional puede

considerarse como una extensión de la programación lógica pura que usa ecuaciones para expresar restricciones entre símbolos de función. El uso multimodo de las definiciones de función resulta posible mediante técnicas de resolución de restricciones[15].

El lenguaje que se define en este trabajo es una instancia del esquema CLP que resuelve ecuaciones bajo una teoría de la igualdad presentada mediante una teoría ecuacional de Horn *E*. La estructura distinguida es el álgebra cociente *H*/*E*. La principal ventaja de esta aproximación a la integración es que, dado que el lenguaje es una instancia del esquema, todas las propiedades semánticas de los lenguajes lógicos tradicionales son heredadas automáticamente por él. Por otra parte, si se desarrolla un algoritmo eficiente para resolver ecuaciones, éste puede ser incorporado fácilmente a un sistema CLP más general y cooperar con otros algoritmos de resolución de restricciones, ofreciéndose así la posibilidad de trabajar, conjuntamente, con dominios simbólicos y con otros dominios. El algoritmo de "narrowing" o algún otro procedimiento correcto y completo de *E*-unificación puede ser considerado el núcleo del mecanismo de resolución de restricciones que semidecide la solubilidad de las mismas en la estructura *H*/*E*. Por los motivos discutidos durante la presentación del mecanismo de SLDE-resolución (subsección §2.1.3.), aunque la solubilidad de las restricciones ha de ser verificada, las restricciones no pueden ser resueltas enteramente en cada paso de computación. En el próximo capítulo veremos cómo este problema ha sido elegantemente resuelto en CLP(*H*/*E*).

[15]El lenguaje Unicorn [Bandes 84] puede ser considerado el primer lenguaje lógico-ecuacional que combinó técnicas de unificación semántica y simplificación de restricciones. Sin embargo, Unicorn nunca fue formalizado.

# 3 El lenguaje CLP(*H/E*) y su semántica operacional

Este capítulo está dedicado a la definición de la sintaxis y de la semántica operacional del lenguaje CLP(*H*/*E*), una instancia del esquema CLP especializada en resolver ecuaciones bajo una teoría de la igualdad presentada mediante una teoría ecuacional de Horn *E*. La estructura correspondiente a esta instancia del esquema es *H*/*E*, la partición más fina inducida por la teoría de la igualdad sobre el universo de Herbrand *H* para el programa. El único símbolo de predicado para las restricciones es =, que es interpretado como igualdad semántica en el dominio.

El lenguaje CLP(*H*/*E*) aúna el estilo de programación lógica basado en cláusulas de Horn, el paradigma basado en ecuaciones (condicionales) y la programación con restricciones. Respecto a otros lenguajes lógico - ecuacionales, CLP(*H*/*E*) tiene una notable potencia expresiva (gracias a la posibilidad de incluir restricciones) y un eficiente modelo computacional, basado en el uso de un refinamiento incremental del procedimiento de "narrowing" básico [Bosco et al 88, Hölldobler 89, Hullot 80, Nutt et al 89] como núcleo del mecanismo de resolución de restricciones que semidecide la solubilidad de las mismas en la estructura *H*/*E*. Para la definición de la semántica operacional del lenguaje seguiremos la aproximación basada en el formalismo de los *Sistemas de Transición* definido por Plotkin [Plotkin 81]. La caracterización de la semántica declarativa del lenguaje se presentará en el capítulo cuatro donde, además, se ofrecen los resultados de corrección y de completitud correspondientes.

Este capítulo está organizado como sigue:

La sección primera reformula algunos de los conceptos básicos del marco conceptual de CLP en términos de *transiciones* y de *configuraciones*, para facilitar el posterior uso de la aproximación de Plotkin a la definición de la semántica operacional del lenguaje. También se formalizan algunas de las nociones y resultados básicos sobre ecuaciones, sistemas de reescritura condicional y unificación universal, que fueron revisados informalmente en los capítulos de introducción y serán utilizados en el resto de la tesis. Se ha preferido concentrar todas las definiciones técnicas en este único apartado para facilitar su posible consulta.

La sección segunda describe la sintaxis de los programas CLP($H/E$). Informalmente, dichos programas constan de dos conjuntos de cláusulas de Horn definidas. Uno de ellos, la parte relacional, contiene sólo cláusulas cuya cabeza es un átomo (no ecuacional) y cuyo cuerpo puede contener tanto átomos como restricciones (ecuacionales). La parte funcional contiene sólamente cláusulas constituidas enteramente por ecuaciones (cláusulas de Horn ecuacionales o, equivalentemente, ecuaciones condicionales).

La sección tercera describe el mecanismo de cómputo del lenguaje. Definimos varias estrategias relacionadas con los aspectos de incrementalidad y de simplificación comentados en la subsección §2.2.5. Los siguientes aspectos, de importancia fundamental en el contexto CLP, han sido considerados:

• cómo verificar la solubilidad de las restricciones en la estructura $H/E$ usando algún procedimiento correcto y completo de unificación semántica, tal como "narrowing".

• cómo lograr incrementalidad en el proceso de computación.

• cómo simplificar las restricciones en una secuencia de computación.

• cómo utilizar las restricciones de forma activa para reducir la talla del espacio de búsqueda.

• cómo beneficiarse de las derivaciones falladas finitamente como un heurístico para optimizar los algoritmos, para lograr una mejora comparable a la de un "backtracking" inteligente [Bruynooghe Pereira 84, Wolfram 86].

Para la definición formal del mecanismo computacional del lenguaje presentamos varios cálculos, que serán descritos como conjuntos de reglas de inferencia. El nivel superior define abstractamente el intérprete CLP. Se construye sobre la base de un procedimiento incremental que no sólo comprueba la satisfacibilidad de las restricciones sino que, además, las simplifica. Este procedimiento utiliza activamente las restricciones para reducir la talla del espacio de búsqueda. Como núcleo de éste se describe otro cálculo, el cual define un procedimiento de "narrowing" que es guiado de forma heurística por las sustituciones descartadas (en computaciones previas) durante la búsqueda de soluciones a nuevas restricciones. Finalmente presentamos un cálculo que, además, construye incrementalmente el árbol de búsqueda para las restricciones. Las

propiedades de corrección y de completitud se estudian para todos ellos. Una subsección con ejemplos de computaciones muestra la simplicidad conceptual del mecanismo de resolución de objetivos de CLP(*H/E*).

## 3.1. Preliminares.

Sea $SORT = \cup SORT_i$ un conjunto finito de géneros ("sorts"). Una *signatura* de un símbolo de función, predicado o variable $n$-ario $f$ es una secuencia de, respectivamente, $n+1$, $n$ o $1$ elementos de $SORT$ que sirve para indicar tanto el número de argumentos como el género de cada uno. Por simplicidad escribiremos $f/n$ para indicar que el símbolo $f$ es $n$-ario. El último elemento en la *signatura* del símbolo de función $f$ indica el *género de f*. Por $\Sigma$, $\Pi$ y $V$ (posiblemente con subíndices) denotamos, respectivamente, colecciones numerables de símbolos de función, símbolos de predicado y símbolos de variable con sus signaturas. Asumimos que cada género es no vacío. $\tau(\Sigma \cup V)$ y $\tau(\Sigma)$ denotan, respectivamente, los conjuntos de términos y términos sin variables construidos sobre $\Sigma$ y $V$. Un $(\Pi,\Sigma)$-átomo es una expresión de la forma $p(t_1, \ldots, t_n)$ donde $p \in \Pi$ es $n$-ario y $t_i \in \tau(\Sigma \cup V)$, $i=1, \ldots, n$. Una $(\Pi,\Sigma)$-restricción es un conjunto finito (posiblemente vacío) de $(\Pi,\Sigma)$-átomos. Intuitivamente, una restricción es una conjunción de $(\Pi,\Sigma)$-átomos. La restricción vacía será denotada por *true*. El símbolo $\tilde{}$ denotará una secuencia finita de símbolos.

$\tau(\Sigma)$ suele ser llamado el Universo de Herbrand ($H_\Sigma$) sobre el alfabeto $\Sigma$. Eliminaremos el subíndice $\Sigma$ si viene fijado por el contexto y nos referiremos a $\tau(\Sigma)$ simplemente como $H$.

**Definición 3.1.1.** *(programa CLP)*

Sea $\Pi = \Pi_C \cup \Pi_B$, con $\Pi_C \cap \Pi_B = \varnothing$[16]. Un $(\Pi,\Sigma)$-programa es un conjunto de cláusulas de la forma

$$H \leftarrow c \,\square. \quad \text{o} \quad H \leftarrow c \,\square\, B_1, \ldots, B_n.$$

donde $c$ es una $(\Pi_C,\Sigma)$-restricción finita (posiblemente vacía), y $H$ (la cabeza) y $B_1, \ldots, B_n$ (el cuerpo), $n \geq 0$, son $(\Pi_B,\Sigma)$-átomos. Un *objetivo* es una cláusula de programa sin cabeza.

[16] Informalmente, $\Pi_C$ representa el conjunto de símbolos de predicado para las restricciones y $\Pi_B$ el conjunto de símbolos de predicado para los átomos en el cuerpo de las cláusulas.

**Definición 3.1.2.** *(átomo restringido)*

Sea $\Pi = \Pi_C \cup \Pi_B$, con $\Pi_C \cap \Pi_B = \varnothing$. Un $(\Pi,\Sigma)$-átomo restringido es un objeto de la forma

$$c \square p(\tilde{X})$$

$\tilde{X}$

donde $c$ es una $(\Pi_C,\Sigma)$-restricción y $p(\tilde{X})$ es un $(\Pi_B,\Sigma)$-átomo.

**Definición 3.1.3.** *(expresión)*

Una expresión es tanto un término, una secuencia de términos, un átomo o una conjunción de átomos. Se dice que una expresión es básica ("ground") si no contiene variables.

En el marco conceptual de CLP(χ), χ representa la estructura que especifica el dominio sobre el que se efectúa la computación, dando la interpretación semántica para símbolos de función y de predicado (de las restricciones), y es el elemento clave para la semántica algebraica.

**Definición 3.1.4.** *(estructura)*

Sea $\Pi = \Pi_C \cup \Pi_B$, con $\Pi_C \cap \Pi_B = \varnothing$. A continuación definimos el concepto de estructura $\chi = \Re(\Pi,\Sigma)$ sobre los alfabetos de símbolos de predicado y función $\Pi$ y $\Sigma$, donde $\Pi$ contiene el símbolo igualdad, que se supone sobrecargado y no necesita signatura. $\Re(\Pi,\Sigma)$ consta de:

1. una colección $D\Re$ de conjuntos no vacíos $D\Re_s$, con $s \in SORT$.
2. una asignación a cada símbolo de función $n$-ario $f \in \Sigma$ de una función $D\Re_{s_1} \times ... \times D\Re_{s_n} \rightarrow D\Re_s$ donde $(s_1, s_2, \ldots, s_n, s)$ es la *signatura* de $f$.
3. una asignación a cada símbolo de predicado $n$-ario $p \in \Pi_C$, distinto de la igualdad, de una función $D\Re_{s_1} \times ... \times D\Re_{s_n} \rightarrow \{True, False\}$, donde $(s_1, s_2, \ldots, s_n)$ es la *signatura* de $p$.

Una $\Re(\Pi,\Sigma)$-*valoración* es una aplicación $\theta: V \rightarrow D\Re$, donde $V = \cup_{s \in SORT} V_s$ es el conjunto de todas las variables y $X_s\theta \in D\Re_s$, siendo $s$ el género de la variable $X_s$.

Los $(\Pi,\Sigma)$-programas, $(\Pi_B,\Sigma)$-átomos y $(\Pi_C,\Sigma)$-restricciones serán llamados programas, átomos y restricciones, respectivamente. Además, $\mathcal{R}$ $(\Pi,\Sigma)$ será denotada por $\mathcal{R}$. La noción de $\mathcal{R}$ $(\Pi,\Sigma)$-valoración se extiende de forma homomórfica al caso de términos y restricciones. Si $C$ es un conjunto (posiblemente infinito) de restricciones atómicas, escribiremos $\mathcal{R} \models C\theta$ sii $\forall c \in C$ se cumple que $\mathcal{R} \models c\theta$ ($c\theta$ es $\mathcal{R}$-*equivalente* a *true*). Si $A = p(t_1, \ldots, t_n)$ es un $(\Pi_B,\Sigma)$-átomo y $\theta$ es una $\mathcal{R}$ $(\Pi,\Sigma)$-*valoración* para $t_1, \ldots, t_n$ entonces $A\theta$ denota $p(t_1\theta, \ldots, t_n\theta)$.

**Definición 3.1.5.** (*$\mathcal{R}$-solubilidad)*

Dada una estructura $\mathcal{R}$, se dice que una restricción $c$ es $\mathcal{R}$-soluble (en símbolos, $\mathcal{R} \models \tilde{\exists}$ ración $\theta$ tal que $\mathcal{R} \models c\theta$. Se dice entonces que $\theta$ es una $\mathcal{R}$-solución de $c$. También se dice que $c$ es consistente.

A continuación definimos la subclase de estructuras que pueden acomodarse dentro del esquema CLP. Una estructura $\mathcal{R}(\Pi,\Sigma)$ es *compacta respecto a las soluciones* si para todo género $s \in SORT$ y elemento $d \in D_{\mathcal{R}_s}$ se cumple que:

*($SC_1$)* $d$ puede definirse unívocamente como la única solución de un conjunto finito o infinito de $(\Pi,\Sigma)$-restricciones.

*($SC_2$)* si $d$ no puede ser definido de forma unívoca mediante un conjunto finito de $(\Pi,\Sigma)$-restricciones ($d$ es un elemento límite) y el conjunto (infinito) de restricciones $C_\omega$ define $d$, entonces para cualquier restricción $c$:

$c \cup C_\omega$ no es $\mathcal{R}$-soluble sii $c \cup c_0$ no es $\mathcal{R}$-soluble
para alguna restricción finita $c_0$ en $C_\omega$.

Informalmente, una estructura es *compacta respecto a las soluciones* si todo elemento del dominio es la única solución de un conjunto (finito o no) de restricciones y si un conjunto infinito de restricciones (del cual un subconjunto define un elemento límite) es irresoluble sii hay un subconjunto finito del mismo que es irresoluble. La primera propiedad es comparable a una condición de *no basura* (todos los elementos del dominio pueden verse como la interpretación de algún objeto sintáctico). La segunda garantiza que el complemento de cualquier restricción puede especificarse por medio de un número (posiblemente infinito) de restricciones. Esta condición es necesaria para establecer muchos resultados que conciernen al fallo finito y a la negación como fallo.

Cualquier estructura sin elementos límite es trivialmente *compacta respecto a las soluciones* [Jaffar Lassez 87]. Esto incluye, en particular, cualquier estructura finita y algunas en general no finitas, como los reales y las estructuras $H$, $H_{II}$ y $H/E$ definidas en la subsección §2.2.5.

A continuación se revisa el concepto que permitirá establecer la correspondencia entre la semántica lógica y la semántica algebraica.

En programación lógica tradicional, los programas gozan de la siguiente propiedad: $P \cup \neg G$, donde $P$ es un programa y $G$ un objetivo, es insatisfacible sii es insatisfacible en modelos de Herbrand. Esto permite que la computación se realice en el universo de Herbrand. El siguiente concepto resulta necesario para elevar este resultado al caso CLP.

**Definición 3.1.6.**

Se dice que una teoría de primer orden $\mathfrak{T}$ *corresponde* a una estructura $\mathfrak{R}$ *compacta respecto a las soluciones* si

- $\mathfrak{R} \models \mathfrak{T}$, i.e. $\mathfrak{R}$ es modelo de $\mathfrak{T}$,
- $\mathfrak{R} \models \tilde{\exists} c$ implica $\mathfrak{T} \models \tilde{\exists}\ c$ para toda restricción $c$

La correspondencia entre teoría y estructura es suficiente para el caso de objetivos que se siguen de un programa. Sin embargo, para el problema complementario de razonar acerca de objetivos negativos, se necesita una condición más fuerte:

Se dice que una teoría de primer orden $\mathfrak{T}$ es *completa respecto a la satisfacción* sii $\mathfrak{T} \models \neg C$ cuando no es el caso que $\mathfrak{T} \models \tilde{\exists}\ C$. Informalmente, esta propiedad significa que tanto la satisfacibilidad como la insatisfacibilidad de las restricciones ha de ser demostrable (cfr. §2.2.3.3).

**Definición 3.1.7.** *(substitución)*

Una $(\Pi,\Sigma)$-substitución ("ground") es un endomorfismo $\theta$: $\tau(\Sigma \cup V) \rightarrow \tau(\Sigma \cup V)$ sobre el álgebra de términos $\tau(\Sigma \cup V)$ que puede ser representado mediante un conjunto finito de pares $\{X_1/t_1, \ldots, X_k/t_k\}$ donde

- $t_i$, $i = 1, \ldots, k$, son términos ("ground") sobre $\Sigma$ que no contienen ninguna ocurrencia de $X_j$, $j = 1, \ldots, k$.
- $X_i$ y $t_i$ tienen el mismo género, $i = 1, \ldots, k$.

La representación ecuacional de una substitución $\theta \{X_1/t_1, \ldots, X_k/t_k\}$ es el conjunto de ecuaciones $\mathcal{O}(\hat{},\theta) = \{X_1 = t_1, \ldots, X_k = t_k\}$. La substitución vacía se denota por $\varepsilon$. Para cualquier substitución $\theta$ y conjunto de variables $W$ ($\subseteq V$), denotamos por $\theta_{|W}$ ($\theta$ restringido a $W$) el conjunto $\{(X/t): (X/t) \in \theta \wedge X \in W\}$. $Var(e)$ es el conjunto de las variables distintas que aparecen en la expresión $e$. Por abuso de notación, llamamos a $Dom(\theta) = \{X \in V: X\theta \neq X\}$ el *dominio de* $\theta$ y a $Cod(\theta) = \{X\theta : X \in Dom(\theta)\}$ el *codominio de* $\theta$. Denotamos por $VCod(\theta) = Var(Cod(\theta))$ el conjunto de variables introducido por $\theta$.

Las nociones de aplicación, composición y generalidad relativa de se definen en la forma usual [Lassez et al 88, Lloyd 87, Siekmann 89]. Por ejemplo, decimos que una $(\Pi,\Sigma)$-substitución $\theta$ es más general que $\gamma$, y lo denotamos como $\theta \leq \gamma$, si existe una $(\Pi,\Sigma)$-substitución $\sigma$ tal que $\gamma = \theta\sigma$.La intersección del preorden $\leq$ con su inversa es una relación de equivalencia (sobre expresiones) llamada *varianza* ($\approx$). En otras palabras, dos expresiones $E_1$ y $E_2$ son variantes ($E_1 \approx E_2$) sii existen dos substituciones $\theta$ y $\gamma$ tales que $E_1\theta = E_2$ y $E_1 = E_2\gamma$. En este caso, $\theta$ (resp. $\gamma$) se llama substitución de renombramiento de variables con respecto a $E_1$ (resp. $E_2$).

**Definición 3.1.8.** *(mgu)*

Una substitución $\theta$ es un unificador de un conjunto de ecuaciones $c = \{t_1 = t'_1, \ldots, t_n = t'_n\}$ sii $t_1\theta = t'_1\theta \wedge \ldots \wedge t_n\theta = t'_n\theta$. Decimos que $\theta$ es un unificador lo más general posible (mgu) para $c$ sii $\forall$ unificador $\theta'$ de $c$, $\exists\, \gamma$ tal que $\theta' = \theta\gamma$ ($\theta$ es divisor de cualquier unificador $\theta'$).

**Definición 3.1.9.**

Sea $c$ un conjunto de ecuaciones. Denotamos por $mgu(c)$ el conjunto de unificadores más generales de $c$. De la definición de *mgu* se sigue que los elementos de este conjunto son el mismo, módulo renombramiento de las variables [Lloyd 87].

Como ya se ha comentado, en este capítulo seguimos la aproximación operacional a la semántica basada en *Sistemas de Transición*. Este estilo de presentación de la

definición semántica operacional de un lenguaje de programación se basa en la especificación de un conjunto $\Gamma$ de *configuraciones*, que definen los estados por los que puede pasar un programa durante la ejecución, una relación $\rightarrow \subseteq \Gamma x \Gamma$ que describe la relación de *transición* entre las *configuraciones* y un conjunto $T \subseteq \Gamma$ de configuraciones terminales. Un sistema de transición (no etiquetado) es un triplete $< \Gamma, T, \rightarrow >$. Para sistemas etiquetados es necesario, además, definir un conjunto $\Lambda$ de etiquetas y entonces la relación de transición $\rightarrow \subseteq \Gamma x \Lambda x \Gamma$. En lo que sigue usaremos siempre la terminología estándar [Plotkin 81].

A continuación reformulamos algunos de los conceptos básicos del marco conceptual de CLP en términos de *transiciones* y de *configuraciones*. Para especificar el sistema de transición CLP($\chi$) debemos mostrar el conjunto $\Gamma$ de *configuraciones* (estados), la relación de transición $\rightarrow_{CLP(\chi)}$ y el conjunto de configuraciones terminales y etiquetas.

**Definición 3.1.10.** *configuraciones* CLP($\chi$)

Sea $\chi = \Re(\Pi,\Sigma)$ una estructura sobre los alfabetos $\Pi$ y $\Sigma$. Sea $P$ un $(\Pi,\Sigma)$-*programa* y $G = \leftarrow c \square A_1, \ldots, A_n$. un $(\Pi,\Sigma)$-*objetivo*. Definimos una $(\Pi,\Sigma)$-*configuración* como un par

$$C = < \leftarrow s[c] \Diamond A_1, \ldots, A_n >$$

donde *s[c]* denota un estado de la "máquina" que resuelve las restricciones ("constraint solver"), cuya estructura se deja sin especificar, puesto que depende del algoritmo específico elegido, pero que incluye, al menos, la restricción *c*. Cuando *c* viene fijada por el contexto, denotaremos *s[c]* simplemente por *s*.

**Definición 3.1.11.** *relación de transición* $\rightarrow_{CLP(\chi)}$

La regla que describe un paso de $(P,\Re)$-*computación* a partir de una $(\Pi,\Sigma)$-configuración $< \leftarrow s_i \Diamond A_1, \ldots, A_n >$, donde $s_i$ incluye la restricción $c_i$, viene dada por:

$$\frac{s_i \xrightarrow{\tilde{c}} s_{i+1} \qquad \tilde{B}}{< \leftarrow s_i \Diamond A_1, \ldots, A_n > \rightarrow_{CLP(\chi)} < \leftarrow s_{i+1} \Diamond \tilde{B}_1, \ldots, n >}$$

si existen $n$ variantes de cláusulas en $P$, $H_j \leftarrow c'_j \square \tilde{B}_j$., $j=1, \ldots, n$, sin variables en común con $\leftarrow c_i \square A_1,\ldots, A_n$. ni con el resto de cláusulas, y $\tilde{c} = \{c'_1,\ldots, c'_n, A_1 = H_1, \ldots, A_n = H_n\}$. La condición $s_i \quad s_{i+1}$ significa que algún "constraint solver" puede efectuar un movimiento verificando la $\mathfrak{R}$–solubilidad de la restricción $c_i \cup \tilde{c}$ y devolviendo el nuevo estado del "constraint solver" $s_{i+1}$ que incluye (una versión posiblemente simplificada de) esta restricción, $c_{i+1}$.

**Definición 3.1.12.** *configuración* CLP(χ) *inicial* ($C_0$)

Sea $G_0 = \leftarrow c_0 \ \tilde{B}$. un (Π,Σ)-*objetivo* y sea $s_0$ un estado distinguido (estado *vacío*) del "constraint solver". Si $c_0$ no es una restricción vacía y $s_0 \overset{c_0}{\longrightarrow} s$ entonces $C_0 = < \leftarrow s \ \Diamond \ \tilde{B} >$ es la *configuración* CLP(χ) *inicial*. Si $c_0$ es vacía entonces $C_0 = < \leftarrow s_0 \ \Diamond \ \tilde{B} >$.

**Definición 3.1.13.** *configuraciones* CLP(χ) *terminales* ($C$)

Una configuración terminal $C$ es de la forma $C = < \leftarrow s[c] \ \Diamond >$, donde $c$ representa la *restricción respuesta computada* correspondiente a esa derivación.

Se debe notar que cuando el "constraint solver" se diseña simplemente para verificar la $\mathfrak{R}$(Π,Σ)-solubilidad de la restricción $c_i \cup \tilde{c}$, la relación de transición definida anteriormente describe el paso de ($P,\mathfrak{R}$)-*derivación* estándar, tal como se define en [Jaffar Lassez 86]:

$$\frac{\mathfrak{R}(\Pi,\Sigma) \models \tilde{\exists}(c_i \cup \tilde{c})}{< \leftarrow c_i \square A_1, \ldots, A_n. > \rightarrow_{CLP(\chi)} < \leftarrow c_i \cup \tilde{c} \square \tilde{B}_1, \ldots, \tilde{B}_n.>}$$

En este caso, los conceptos de ($P,\mathfrak{R}$)-*derivación* y ($P,\mathfrak{R}$)-*derivaciones con éxito y derivaciones falladas finitamente* se pueden definir en la forma usual [Jaffar Lassez 86]. La restricción en el último objetivo de una derivación con éxito es la *restricción respuesta computada* en dicha derivación. En [van Hentenryck Deville 91] se sigue también una aproximación estructural para la definición de la semántica operacional de (una clase de) lenguajes CLP. Pero nuestra formalización es más general y capaz de dar soporte a estrategias incrementales.

A continuación revisamos brevemente algunas nociones y resultados básicos sobre ecuaciones, sistemas de reescritura condicional y unificación universal. En este

apartado no se suministra ninguna demostración, ya que todo el material es bien conocido. Para más detalles ver, e.g., [Kaplan 84, Siekmann 84].

**Definición 3.1.14.**

Sea $\rightarrow$ una relación binaria sobre un conjunto $S$. Usaremos $\rightarrow^*$ para denotar el cierre reflexivo y transitivo de $\rightarrow$. La relación $\rightarrow$ se dice *confluente* si $\forall s_1, s_2, s_3 \in S$ tales que $s_1 \rightarrow^* s_2 \wedge s_1 \rightarrow^* s_3$, $\exists s \in S$ tal que $s_2 \rightarrow^* s \wedge s_3 \rightarrow^* s$. La relación $\rightarrow$ se dice *noetheriana* si no hay cadenas infinitas de la forma $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow ...$

**Definición 3.1.15.** *(ecuación, teoría ecuacional de Horn, E-igualdad)*

Una $\Sigma$-*ecuación* $s = t$ es un par de términos $s$ y $t$ del mismo género ($s,t \in \tau(\Sigma \cup V)$). Una $\Sigma$-*teoría* ecuacional de Horn $E$ consiste en un conjunto finito de cláusulas de Horn ecuacionales, de la forma $l = r \leftarrow e_1, e_2, ..., e_n,\ n \geq 0$. La $\Sigma$-*ecuación* $l = r$ en la cabeza de la cláusula se considera implícitamente orientada de izquierda a derecha y los literales $e_i$ en el cuerpo son $\Sigma$-*ecuaciones* ordinarias (no orientadas). A menudo llamaremos a las $\Sigma$-*ecuaciones* y a las $\Sigma$-*teorías*, *ecuaciones* y *teorías*, respectivamente.

Una teoría ecuacional de Horn $E$ puede verse como un sistema de reescritura $R$ donde las reglas son las cabezas y las condiciones son los respectivos cuerpos. Si todas las cláusulas en E tienen cuerpo vacío entonces se dice que $E$ y $R$ son no condicionales; si no, se dicen condicionales. Se dice que la teoría ecuacional $E$ es canonica si la relación binaria de reescritura $\rightarrow_R$ definida por $R$ es noetheriana y confluente.

La relación de reescritura $\rightarrow_R$ es la unión de un conjunto infinito de relaciones

$$\rightarrow_R \ = \bigcup^{i \geq 0} \ \overset{i}{\longrightarrow}_R$$

donde

$\overset{i}{\longrightarrow}_R$ es la relación vacía.

$t \overset{n+1}{\longrightarrow}_R s$ se satisface si hay una regla $l = r \leftarrow t_1 = t'_1, ..., t_m = t'_m,\ m \geq 0$, una ocurrencia no variable $u$ de $t$, una substitución $\sigma$ y $m$ términos $\gamma_i$ tal que $t/u = l\sigma,\ t_i\sigma \overset{n}{\longrightarrow}_R{}^* \gamma_i \ {}^* \overset{n}{\longleftarrow}_R t'_i\sigma, n \geq 0,$ y $s = t[u \leftarrow r\sigma]$.

Se dice que una relación de reescritura $\rightarrow_R$ es *confluente por niveles* si la relación definida anteriormente $(\overset{n}{\longrightarrow}_R, n \geq 0)$ es confluente separadamente. Por supuesto, confluencia por niveles implica confluencia pero lo contrario no es cierto. Por ejemplo, el

siguiente sistema de reescritura [Middeldorp 91a] es confluente pero no es confluente por niveles.

$$\begin{array}{lll} f(X) & \rightarrow a \quad \leftarrow & X = b,\ X = c. \\ d & \rightarrow b. & \\ d & \rightarrow c. & \\ b & \rightarrow c \quad \leftarrow & f(d) = a. \end{array}$$

Para caracterizaciones sintácticas de teorías condicionales confluentes (y/o confluentes por niveles) consultar [Bergstra Klop 86, Giovannetti Moiso 86, Huet Levy 79, Levi et al 87, Moreno Rodríguez 89].

Se dice que un término $t_1$ está en forma normal si no existe un $t_2$ tal que $t_1 \rightarrow_R t_2$. Si $R$ es canónico, todos los términos tienen una y sólo una forma normal. El concepto de forma normal se extiende al caso de substituciones. Una substitución $\sigma$ está normalizada si $\forall X.\ (X\sigma)$ está en forma normal.

Se dice que $E$ tiene *variables extra* [Bosco et al 91, Hölldobler 89] cuando se permite que variables que no están en la parte izquierda (lhs) de la cabeza de la cláusulas sí aparezcan en la correspondiente parte derecha (rhs) y/o en el respectivo cuerpo.

Dada una teoría ecuacional de Horn $E$ y una ecuación $e$, escribimos $E \models e$ para denotar que $e$ es verdad en todos los modelos de $E$. Cada teoría ecuacional de Horn $E$ genera una relación de congruencia más fina $\equiv_E$, llamada *E-igualdad*, sobre $\tau(\Sigma \cup V)$ (la menor teoría que contiene todos los pares $s = t$ tales que $E \models s = t$ [Burckert et al 87]). Se dice que $E$ es una presentación o axiomatización de la teoría de la igualdad $\equiv_E$ y, por abuso, se habla a veces de la teoría de la igualdad $E$ para denotar la teoría axiomatizada por $E$. Denotaremos como H/E la partición más fina $\tau(\Sigma)/\equiv_E$ inducida por $E$ sobre el conjunto de términos "ground" $\tau(\Sigma)$. A continuación introduciremos el problema de la unificación bajo una teoría ecuacional de Horn $E$ como el problema de la existencia de una substitución tal que la correspondiente instancia de la ecuación se sigue lógicamente de la teoría. Formularemos la solución a este problema como un conjunto completo de unificadores bajo la teoría ecuacional.

**Definición 3.1.16.** *(E-unificador)*

Dados dos términos $s$ y $t$, decimos que $s$ y $t$ son *E-unificables* (o *E-iguales*) sii existe una substitución $\sigma$ tal que $s\sigma$ y $t\sigma$ están en la congruencia $\equiv_E$, abreviado $s\sigma \equiv_E t\sigma$

(o, equivalentemente, $E \models s\sigma = t\sigma$). La substitución $\sigma$ se llama un *E-unificador* de $s$ y $t$.

Dado un conjunto de variables $W \subseteq V$, extendemos el concepto de *E-igualdad* al caso de substituciones en la manera estándar: $\sigma =_E \theta\ [W]$ sii $X\sigma =_E X\theta \quad \forall X \in W$. Cuando $W$ coincida con $V$, será omitido.

**Definición 3.1.17.** *(E-instancia)*

Se dice que la substitución $\sigma'$ es una *E-instancia* de $\sigma$ (y que $\sigma$ es más general que $\sigma'$) sobre $W$ (en símbolos, $\sigma \leq_E \sigma'\ [W]$) sii $(\exists \rho)$ tal que $\sigma' =_E \sigma\rho\ [W]$. Se dice que dos substituciones $\sigma, \sigma'$ son *E-equivalentes* sobre $[W]$ (en símbolos, $\sigma =_E \sigma'\ [W]$) sii $\sigma \leq_E \sigma'\ [W]$ *y* $\sigma' \leq_E \sigma\ [W]$.

El conjunto de *E-unificadores* de una ecuación es cerrado bajo *E-instanciación* (cualquier *E-instancia* de un *E-unificador* es también un *E-unificador*). Por este motivo se utilizan conjuntos representativos de *E-unificadores,* con la propiedad de que la unión de todas las *E-instancias* de los elementos del conjunto es exactamente el conjunto de todos los *E-unificadores* de la ecuación.

**Definición 3.1.18.**

Se dice que un conjunto $U_E$ de substituciones es un *conjunto completo* y *minimal* de *E-unificadores* de una ecuación $e$ (sobre $W$) sii

1. $\forall \sigma \in U_E$, $\sigma$ es un *E-unificador* de $e$. (coherencia)
2. $\forall$ *E-unificador* $\sigma$ de $e$, $\exists \sigma' \in U_E$, tal que $\sigma' \leq_E \sigma\ [W]$. (completitud)
3. $\forall\ \sigma, \sigma' \in U_E, \sigma \leq_E \sigma'\ [W] \Rightarrow \sigma = \sigma'$ (minimalidad)

Por razones prácticas resulta útil exigir un requerimiento técnico adicional:

4. Dado un conjunto de variables $Z \supseteq W$, $(\forall \sigma \in U_E)\ Dom(\sigma) \subseteq W$ y $VCod(\sigma) \cap Z = \varnothing$ (pureza o protección de $W$)

Informalmente, un conjunto completo de *E-unificadores* es aquel tal que cualquier otro *E-unificador* puede obtenerse como instancia (semántica) de algun elemento del conjunto. Si el conjunto contiene sólo estos unificadores más generales y no

otros, entonces se dice minimal. Las substituciones en un conjunto minimal de *E-unificadores* son mutuamente independientes. La necesidad de considerar un conjunto *Z* viene del hecho de que los términos de la ecuación *e* pueden ser subtérminos de términos mayores, conteniendo variables que no estén en la ecuación, y no se quiere confundir estas variables con las de $Vcod(\sigma)$. Se debe notar que este requerimiento adicional de pureza implica que los *E-unificadores* en $U_E$ son idempotentes ($\sigma\sigma = \sigma$ ya que $Dom(\sigma) \cap VCod(\sigma) = \emptyset$.

Dado que en teorías presentadas por un conjunto finito de axiomas el conjunto de *E-unificadores* de una ecuación es recursivamente enumerable [Gallier Raatz 89], siempre existe un conjunto completo de *E-unificadores* aunque dicho conjunto no tiene por qué ser finito (no lo es, e.g., para $a * X = X * a$ en teorías donde * es asociativa). Si es finito, siempre es posible obtener a partir de él un conjunto completo y minimal, filtrando los elementos redundantes [Fages Huet 83]. Si no lo es, es posible que no exista un conjunto completo y minimal, ni siquiera infinito [Fages Huet 83, Plotkin 72]. Una posible razón es que la relación de orden inducida por $\leq_E$ sobre el conjunto de los *E-unificadores* de una ecuación no es bien fundada (no toda cadena estrictamente decreciente es finita) [Bürckert et al 87, Gallier Raatz 89, Siekmann 89], aunque esta condición de noetherianidad no es necesaria para la existencia de un conjunto minimal. Si existe un conjunto completo y minimal entonces es único, módulo la relación de equivalencia inducida por $\leq_E$, lo cual autoriza a computar sólo uno como representante de la clase entera.

Obviamente, cuando *E* es vacía, sí existen siempre conjuntos completos y minimales, que son vacíos o se reducen a un único elemento, que antes hemos denominado el *unificador más general* (*mgu*) de la ecuación. Por analogía con este caso, en la literatura sobre unificación universal, a un conjunto completo y minimal de *E-unificadores* de una ecuación se le llama también *conjunto de E-unificadores maximalmente generales* [Jaffar et al 86] o *conjunto de unificadores más generales (mgu) bajo E* [Siekmann 84] ("separados de *Z*"). Cuando este conjunto tiene un único elemento (módulo la congruencia $\equiv_E$) se dice que la ecuación tiene un único *E-mgu*.

**Definición 3.1.19.** *(E-mgu)*

Sea *e* una ecuación y $W = Var(e)$. Un *E-unificador* $\sigma$ de *e* se dice un *E-unificador más general (E-mgu)* de *e* (sobre *W*) si para cualquier *E-unificador* $\sigma'$ de *e* se cumple que $\sigma \leq_E \sigma'$ *[W]*.

El problema de la *E-unificación* puede verse como el proceso de resolver una ecuación respecto de una teoría ecuacional $E$. Ya que la *E-unificación* es sólo semidecidible (incluso en teorías no condicionales y canónicas como se demuestra, e.g., en [Hölldobler 89]), un algoritmo de *E-unificación* puede verse como un procedimiento que semidecide la solubilidad de las restricciones ecuacionales (cuantificadas existencialmente) en el cociente $H/E$. Cada instancia de un *E-unificador* representa una solución sobre esta estructura (una $H/E$-solución). En la literatura sobre *E-unificación* es bastante común llamar *solución* al propio *E-unificador* (ver, e.g., [Bürckert 88, Bürckert et al 87]).

Un procedimiento de *E-unificación* es completo si genera un conjunto completo de *E-unificadores* para cualquier ecuación de entrada. Se han desarrollado numerosos procedimientos de *E-unificación* para tratar con teorías ecuacionales condicionales (ver, e.g., [Dershowitz Plaisted 85, Fribourg 85b, Giovannetti et al 91, Hölldobler 90, Hussmann 86, Kaplan 86]). Muchos de ellos se basan en versiones más o menos refinadas del algoritmo de "narrowing" condicional. Su completitud ha sido demostrada para teorías condicionales que satisfacen diferentes restricciones [Dershowitz Plaisted 85, Giovannetti Moiso 86, Hölldobler 89, Kaplan 86]. En general, los algoritmos de *E-unificación* no producen un conjunto completo y minimal aún si éste existe. También es conocido que el algoritmo puede no terminar, incluso si existe un conjunto completo, minimal y finito de *E-unificadores* de la ecuación. Para el caso de teorías no condicionales, [Hullot 80] presenta condiciones suficientes para la terminación pero se basan en comprobar que cualquier derivación que empieza a partir de las partes derechas de las cabezas de las reglas termina lo cual, desafortunadamente, es una tarea de la misma naturaleza que decidir el problema original.

**Definición 3.1.20.** *(símbolos irreducibles)*

Sea $E$ una teoría ecuacional de Horn. Un símbolo de función $f \in \Sigma$ se dice *irreducible* sii no existe ninguna cláusula $(l = r \leftarrow e_1, e_2, ..., e_n.) \in E$ tal que $l \in V$ ni $f$ ocurre como el símbolo de función "outermost" en $l$; en otro caso $f$ es un símbolo de función *definido*.

En teorías donde se hace la distinción anterior, la signatura $\Sigma$ queda particionada como $\Sigma = C \cup F$, donde $C$ es el conjunto de símbolos de función irreducibles y $F$ es el conjunto de símbolos de función definidos. Los miembros de $C$ se llaman también *constructores*. El conjunto de cláusulas $\{ f(t_1,\ldots,t_n) = r \leftarrow e_1, e_2, ..., e_n.\} \subset E$ se denomina *definición de f*. La distinción entre constructores y funciones definidas

corresponde a la distinción, presente en los lenguajes de programación convencionales, entre estructuras de datos y algoritmos.

El significado operacional del símbolo predefinido = puede ser especificado ahora por medio de cláusulas [Fribourg 85b, Moreno Rodríguez 89] que pueden ser generadas de forma automática y orientadas como reglas que pueden usarse para la simplificación y también para la deducción de información negativa [Fribourg 85b, Josephson Dershowitz 86]:

$c = c \rightarrow true.$ $\forall$ símbolo de función irreducible $c/0$

$c(X) = c(Y) \rightarrow X = Y.$ $\forall$ símbolo de función irreducible $c/1$

$c(X_1, X_2, \ldots, X_n) = c(Y_1, Y_2, \ldots, Y_n) \rightarrow ANDn( X_1=Y_1, X_2=Y_2, \ldots, X_n=Y_n).$

$\forall$ símbolo de función irreducible $c/n, n > 1$

$c(X_1, X_2, \ldots, X_n) = d(Y_1, Y_2, \ldots, Y_m) \rightarrow false.$

$\forall$ símbolo de función irreducible $c/n, d/m, c \neq d, n \geq 0, m \geq 0$

$ANDn(X_1, \ldots, X_{i-1}, true, X_{i+1}, \ldots, X_n) \rightarrow$

$ANDn\text{-}1(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n).$ $\forall\ n \geq 2$

$AND2(true, X) \rightarrow X .$

$AND2(X, true) \rightarrow X .$

$ANDn(X_1, \ldots, X_{i-1}, false, X_{i+1}, \ldots, X_n) \rightarrow false.$ $\forall\ n > 1, 1 \leq i \leq n$

Las diferentes técnicas de optimización basadas en el uso de símbolos irreducibles afectan a la talla del árbol de búsqueda de los *E-unificadores* de una ecuación. En algunos casos, un árbol de búsqueda infinito puede quedar reducido a uno finito [Hölldobler 89, Hussmann 86] (para un ejemplo, ver §3.3.1.3).

**Definición 3.1.21.** *(disciplina de constructores)*

Se dice que una teoría ecuacional de Horn $E$ es una teoría con *disciplina de constructores* si cada ecuación en la cabeza de cualquier cláusula en $E$ tiene la forma $f(t_1, t_2, \ldots t_n) = t$ donde $f \in F$ y no ocurren símbolos de función definidos en ninguno de los $t_1, t_2, \ldots t_n$. Esta condición implica que no se permiten anidamientos funcionales en la parte izquierda de la cabeza de las cláusulas ni axiomas entre constructores. Todo término formado sólo por constructores y variables (término "data") está siempre en forma normal.

Cuando la teoría ecuacional $E$ satisface una disciplina de constructores, la propiedad de confluencia puede ser garantizada por las siguientes condiciones computables sobre la sintaxis de las cláusulas (cfr. [González et al 90, Moreno 89]):

- Salida definida: En cada cabeza ecuacional $f(t_1, t_2, \ldots t_n) = t$, todas las variables en $t$ deben ocurrir también en $f(t_1, t_2, \ldots t_n)$.

- Linealidad por la izquierda: Las variables del lado izquierdo de una regla aparecen sólo una vez en ésta.

- No ambigüedad: Dadas dos cláusulas en la definición del símbolo de función $f$:

$$f(t_1, t_2, \ldots t_n) = M \;:\!-\; B.$$
$$f(s_1, s_2, \ldots s_n) = N \;:\!-\; C.$$

se satisface alguna de las siguientes condiciones:

ι) No superposición: $f(t_1, t_2, \ldots t_n)$ y $f(s_1, s_2, \ldots s_n)$ no son unificables

ιι) Fusión de cuerpos: $f(t_1, t_2, \ldots t_n)$ y $f(s_1, s_2, \ldots s_n)$ tienen un mgu $\theta$ tal que $M\theta$ y $N\theta$ son idénticos.

ιιι) Incompatibilidad de guardas: $f(t_1, t_2, \ldots t_n)$ y $f(s_1, s_2, \ldots s_n)$ tienen un mgu $\theta$ tal que $B\theta$ y $C\theta$ son incompatibles, i.e., no pueden ser evaluados simultáneamente a *true*.

**Definición 3.1.22.** *(teoría definida completamente)*

Sea $E$ una teoría ecuacional de Horn y $f$ un símbolo de función definido. Se dice que $f$ está completamente definido ("everywhere defined" [Bosco et al 88]) sobre su dominio

sii no ocurre en ningún término "ground" en forma normal. Se dice que $E$ es una teoría completamente definida sii cada símbolo de función definido está completamente definido. En [Echahed 88, Fribourg 85a, Huet Hullot 82] se proponen condiciones suficientes que garantizan esta propiedad, conocida también como "principio de definición". En una teoría completamente definida con disciplina de constructores, el conjunto de términos "ground" en forma normal es el conjunto de términos sobre $C$ ($\tau(C)$). Dicho conjunto constituye el dominio sobre el que se desarrolla la semántica declarativa, representando además los valores finales que pueden tomar las expresiones del lenguaje. Los programas de esta clase admiten, por tanto, una semántica por modelo mínimo de Herbrand similar a la de los programas de cláusulas definidas, donde el universo de Herbrand está formado sólo por constantes y constructores.

**Definición 3.1.23.**

Los términos serán considerados como árboles etiquetados siguiendo la terminología habitual [Huet Oppen 80, Hullot 80]. Un árbol etiquetado es una función parcial del conjunto $N^{+*}$ de secuencias de enteros positivos a $\Sigma \cup V$, cuyo dominio finito $O(t)$ satisface:

- $O(t)$ es no vacío y cerrado bajo prefijo.
- si $u \in O(t)$ y $t[u]$ es un símbolo n-ario, entonces $u.i \in O(t)$ sii $i \in \{1, ..., n\}$.

$O(t)$ denota el conjunto de ocurrencias (secuencias de enteros positivos describiendo un camino de acceso en un término, con la secuencia vacía representada por $\lambda$) de un término y está ordenado parcialmente por la relación de *prefijo*: $u \leq v$ sii $\exists w$ tal que $u.w = v$. Usaremos $t/u$ para denotar el subtérmino de $t$ a la ocurrencia $u \in O(t)$, definido como sigue:

- $t/\lambda = t$
- $f(t_1, t_2, \ldots t_n)/i.u = t_i/u$

y $t[u \leftarrow s]$ para denotar el término que se obtiene de $t$ reemplazando el subtérmino $t/u$ por $s$. Por $O'(t)$ denotamos el conjunto de ocurrencias no variables de $t$, i.e. $O'(t) = \{u \in O(t): t/u \notin V\}$.

Los conceptos de *conjunto de ocurrencias, subtérmino* y *reemplazo de subtérminos* pueden ser extendidos de manera inmediata al caso de ecuaciones y conjunciones de

ecuaciones [Hussmann 86]. Por ejemplo. la definición de *conjunto de ocurrencias* para un sistema de ecuaciones sería:

$$O(t_1 = t'_1, ..., t_n = t'_n) = \{i.1.u:\ i \in \{1, 2, ...n\}, u \in O(t_i)\} \cup \{i.2.u:\ i \in \{1, 2, ...n\}, u \in O(t'_i)\}$$

Los conceptos extendidos de *conjunto de ocurrencias, subtérmino* y *reemplazo de subtérminos* permiten tratar el problema de la *E-unificación* de conjunciones de ecuaciones como simple *E-unificación*. Es decir, se pueden utilizar los mismos algoritmos que se usan para resolver una ecuación única.

### 3.2. Programas lógicos CLP(*H/E*).

CLP(*H/E*) es un lenguaje lógico-ecuacional basado en el esquema de programación lógica con restricciones (CLP). Recordemos que para que un lenguaje pueda verse como instancia del esquema CLP, la estructura correspondiente debe de ser *compacta respecto a las soluciones* (ver definición 3.1.4). La estructura *H/E* es trivialmente *compacta respecto a las soluciones* ya que carece de elementos límite [Jaffar Lassez 86, Jaffar Lassez 87]. También es conocido que la teoría ecuacional *E* y la estructura *H/E* se corresponden en el sentido definido en la sección anterior [Jaffar Stuckey 86a, Jaffar et al 86a];

- $H/E \models E$, i.e. *H/E* es modelo de *E,*
- $H/E \models \tilde{\exists}\, c$ implica $E \models \tilde{\exists}\, c$ para toda restricción *c*

es decir, una restricción *c* es *E-unificable* sii es *H/E-soluble*. Los programas CLP(*H/E*) disfrutan por tanto de la siguiente propiedad [Jaffar et al 86a]: existe un dominio de computación canónico, *H/E*, sobre el cual es posible establecer una clase canónica de interpretaciones respecto a la cual examinar la verdad de las fórmulas. Este resultado es análogo a la canonicidad de las interpretaciones de Herbrand para los programas lógicos tradicionales.

Asumiremos que las restricciones son ecuaciones a resolver en *H/E* por medio del algún algoritmo de *E-unificación* apropiado, que será descrito posteriormente.

Como ya hemos comentado, para heredar del esquema los resultados acerca de la negación, la teoría *E* debería ser extendida a una teoría $\Im$ *completa respecto a la satisfacción*, i.e. $\Im \models \neg\, c$ cuando no es el caso que $\Im \models \tilde{\exists} c$ (cfr. §2.2.3.3 y §3.1).

Desafortunadamente, en nuestro caso, esto no es posible puesto que el problema de la unificación bajo una teoría ecuacional no es decidible. Por tanto, los resultados de corrección y completitud para el fallo finito y la negación como fallo no podrán ser heredados.

La sintaxis concreta del lenguaje CLP(*H/E*) está basada en la sintaxis del esquema. Un programa CLP(*H/E*) consta de dos conjuntos de cláusulas de Horn definidas:

**Definición 3.2.1.** *programas* CLP(*H/E*)

Sea $\Pi_C = \{=\}$, $\Pi = \Pi_C \cup \Pi_B$ y $\Pi_C \cap \Pi_B = \varnothing$. Definimos un $(\Pi,\Sigma)$-programa CLP*(H/E)* como un $(\Pi,\Sigma)$-programa (CLP) $P$ extendido con una $\Sigma$-teoría ecuacional de Horn canónica $E$ sin variables extra.

Algunas observaciones acerca de la definición del lenguaje:

• CLP(*H/E*) no implementa negación *por fallo*. Preferimos manejar la negación incorporando la información negativa en forma de reglas de reescritura, que se usan para reducir las ecuaciones a *false*. La aproximación ecuacional a la negación se sigue también en [Dershowitz Plaisted 85, Goguen Meseguer 86, Fribourg 85b].

• Las cabezas de las cláusulas ecuacionales $f(t_1, t_2, \ldots, t_n) = t$ pueden orientarse explícitamente y escribirse como $f(t_1, t_2, \ldots, t_n) \rightarrow t$. El símbolo $\leftarrow$ en cualquier cláusula puede substituirse por *:-*.

• Usaremos las siguientes formas azucaradas de las cláusulas ecuacionales:

| | |
|---|---|
| $f(t_1, t_2, \ldots, t_n) \rightarrow false.$ | $\neg f(t_1, t_2, \ldots, t_n).$ |
| $f(t_1, t_2, \ldots, t_n) \rightarrow false$ :- $B.$ | $\neg f(t_1, t_2, \ldots, t_n)$ :- $B.$ |
| $f(t_1, t_2, \ldots, t_n) \rightarrow true.$ | $f(t_1, t_2, \ldots, t_n).$ |
| $f(t_1, t_2, \ldots, t_n) \rightarrow true$ :- $B.$ | $f(t_1, t_2, \ldots, t_n)$ :- $B.$ |

Los siguientes ejemplos ilustran la definición del lenguaje CLP(*H/E)*. Se han escrito comentarios para facilitar la lectura. Los comentarios se escriben en la forma habitual: comentarios cerrados (parentizados entre los símbolos /* y */) y comentarios en línea (encabezados por el símbolo %). Los nombres de las variables, símbolos de función y símbolos de predicado tienen el significado esperado.

El siguiente programa lógico-ecuacional (*Cats*) está tomado de [Hölldobler 89].

**Ejemplo 3.2.2.** *Cats*.

```
%  SPEC Cats

%  CONSTRUCTORS
%       0          /   0
%       succ       /   1
%  FUNCTIONS
%        +         /   2
%        *         /   2
%  PREDICATES
%       cats       /   2

%  RELATIONAL CLAUSES
       /* p1 */     cats( NCats, NBirds, NCats + NBirds, 4 * NCats + 2 * NBirds)  :-
                                                                                  □.
%  FUNCTIONAL CLAUSES
       /* e1 */     X + 0  →  X.
       /* e2 */     X + succ( Y )  →  succ( X + Y ).
       /* e3 */     X * 0  →  0.
       /* e4 */     X * succ( Y )  →  X + X * Y.
%  ENDSPEC
```

Explicamos ahora brevemente la intención de los símbolos y reglas propuestos. *0* y *succ* son símbolos irreducibles que representan, respectivamente, el cero y el sucesor de los números naturales. + define la suma de naturales y * define el producto. La relación *cats( C, B, H, L)* se satisface si $C$ gatos y $B$ pájaros tienen juntos $H = C + B$ cabezas y $L = 4 * C + 2 * B$ patas. Por conveniencia notacional, abreviamos $succ^n(0)$ por $n$. Este programa puede ser interrogado de diferentes formas. Por ejemplo, el objetivo ←*true* □*cats( 1, 2, H, L)* computa como respuesta la restricción $H = 3, L = 6$ mientras que el objetivo ←*true* □*cats( C, B, 4, L)* tiene como respuesta $4 = C + B, L = 4 * C + 2 * B$.

El siguiente ejemplo está basado en [Fribourg 86], donde se sigue un método de especificación algebraica para construir especificaciones de protocolos. El sistema que se pretende modelar queda caracterizado por la secuencia de eventos (constructores del estado) ocurridos en el sistema. La especificación no es más que la definición completa de una relación *ok* que establece si una secuencia de eventos es aceptable o no, si satisface o no el protocolo. La definición de la relación *ok* requiere varias definiciones intermedias.

**Ejemplo 3.2.3.** *Protocolo de Lectores y Escritores*.

En este ejemplo se especifica el protocolo de exclusión mutua entre varios lectores y un único escritor que comparten los mismos recursos. Las condiciones de exclusión son:

- sólo puede haber un escritor en cada momento.
- sólo se permite un número máximo de lectores actuando simultáneamente.
- las operaciones de lectura y escritura son mutuamente exclusivas.

Los eventos (constructores del estado) que pueden afectar al sistema son:

- *rbegin* (nuevo lector).
- *rend* (fin lector)
- *wbegin* (nuevo escritor).
- *wend* (fin escritor).

La definición de la relación *ok* requiere tres funciones intermedias para expresar las condiciones de exclusión mutua:

- *rmax:* indica el número máximo de lectores activos.
- *rnumber:* indice el número actual de lectores activos.
- *won:* indica que un escritor está utilizando el recurso.

La relación *ok* se define completamente por recursión estructural sobre términos formados sólo por variables y constructores mediante las siguientes restricciones:

- para *rbegin:* no debe haber ningún escritor activo y sólo un número de lectores menor que *rmax*
- para *rend:* ningún escritor activo y un número de lectores mayor que cero.
- para *wbegin:* ningún escritor y ningún lector.
- para *wend:* un escritor y ningún lector.

Para simplificar el ejemplo, no se considera el problema de la *inanición* (la posibilidad de que algún proceso sea pospuesto indefinidamente). Se supone, además, que *rmax* es dos.

```
%  SPEC Readers-Writers

%  CONSTRUCTORS
%       sinit       /   0
%       rbegin      /   1
%       rend        /   1
%       wbegin      /   1
%       wend        /   1
%       0           /   0
%       succ        /   1

%  FUNCTIONS
%       rmax        /   0
%       rnumber     /   1
%       won         /   1
```

```
%       pred        /   1
%       <           /   2

%   PREDICATES
%       ok          /   1

%   RELATIONAL CLAUSES
        /* p1 */    ok(sinit).
        /* p2 */    ok(rbegin(S))   :-  ¬ won(S), rnumber(S) < rmax   □  ok(S).
        /* p3 */    ok(rend(S))     :-  ¬ won(S), 0 < rnumber(S)      □  ok(S).
        /* p4 */    ok(wbegin(S))   :-  ¬ won(S), 0 = rnumber(S)      □  ok(S).
        /* p5 */    ok(wend(S))     :-    won(S), 0 = rnumber(S)      □  ok(S).
```

```
% FUNCTIONAL CLAUSES
    /* e1 */    rmax                    →   succ(succ(0)).
    /* e2 */    rnumber(sinit)          →   0.
    /* e3 */    rnumber(rbegin(S))      →   succ(rnumber(S)).
    /* e4 */    rnumber(rend(S))        →   pred(rnumber(S)).
    /* e5 */    rnumber(wbegin(S))      →   0.
    /* e6 */    rnumber(wend(S))        →   0.
    /* e7 */    ¬ won(sinit).
    /* e8 */    ¬ won(rbegin(S)).
    /* e9 */    ¬ won(rend(S)).
    /* e10 */   won(wbegin(S)).
    /* e11 */   ¬ won(wend(S)).
    /* e12 */   pred(0)                 →   0.
    /* e13 */   pred(succ(X))           →   X.
    /* e14 */   0 < succ(X).
    /* e15 */   ¬ succ(X) < 0.
    /* e16 */   succ(X) < succ(Y)       →   X < Y.

% ENDSPEC
```

## 3.3. Semántica operacional del lenguage CLP(*H*/*E*).

Un intérprete para un lenguaje CLP se basa en una generalización de la regla de SLD-resolución utilizada en programación lógica convencional y en un algoritmo apropiado para resolver restricciones en el dominio de computación correspondiente. Intuitivamente, una computación CLP puede verse como una secuencia de pasos de reducción (de objetivos) en la que se acumulan restricciones (simplificadas), previa comprobación de su satisfacibilidad. En esta sección se describe el mecanismo de cómputo del lenguaje CLP(*H*/*E*). El algoritmo que resuelve las restricciones en el dominio *H*/*E* se basa en una versión incremental de la extensión al caso condicional del algoritmo de "narrowing" básico, descrito en [Bosco et al 88, Hullot 80] y revisado en [Hölldobler 89, Nutt et al 89].

### 3.3.1. Satisfacción incremental de restricciones (*ICS*).

La manera estándar de integrar "narrowing" y resolución es usar "narrowing" para generar las soluciones que son examinadas después por el programa lógico [Bosco et al 87, Goguen Meseguer 86]. Este estilo *generación y test* resulta ineficiente en la mayoría de casos (puesto que tiende a buscar ciegamente las soluciones en un espacio de búsqueda generalmente grande, lo que conduce rápidamente a una explosión combinatoria [Fribourg 85b, Hao Chabrier 90, van Hentenryck 90]) y requiere complejas interacciones entre el "backtracking" de los algoritmos de "narrowing" y de resolución. Este mecanismo no cabe, por otra parte, dentro del esquema CLP, como argumentaremos

en lo que sigue. En su lugar, "narrowing" puede usarse como un procedimiento para verificar la consistencia de una restricción buscando una solución a la misma ya que, de ser encontrada ésta, la satisfacibilidad de la restricción habrá sido demostrada.

A primera vista, se podría pensar en construir un sistema CLP(*H/E*) mediante la simple incorporación de algún procedimiento correcto y completo de unificación semántica, tal como "narrowing", como el algoritmo de resolución de restricciones dentro del sistema CLP. Los estados de este sencillo "constraint solver" no contendrían nada más que restricciones (conjuntos de ecuaciones). El algoritmo podría describirse mediante la siguiente relación de transición:

$$\frac{\textit{existe un E-unificador de } c \cup \tilde{c}}{c \xrightarrow{\tilde{c}} c \cup \tilde{c}}$$

Sin embargo, esta definición envuelve demasiada computación redundante ya que el conjunto de restricciones crece de forma monótona y la definición anterior requiere la comprobación de la satisfacibilidad del conjunto total en cada transición. El problema está en el hecho de que el coste de comprobar la satisfacibilidad es, en general, significativo por lo que resulta necesario considerar cómo se puede reutilizar el trabajo realizado en los pasos anteriores para reducir el coste de resolver el nuevo conjunto. Una aproximación que no intentara explotar la información que puede ser recogida en los pasos de computación previos para guiar la búsqueda no sería práctica. Para una discusión más amplia sobre este tema, revisar la subsección §2.2.5.

En el contexto de CLP(*H/E*), parece bastante natural intentar reutilizar la solución $\theta$ a la restricción $c$ encontrada en el paso anterior para comprobar la satisfacibilidad de la nueva restricción $c \cup \tilde{c}$. Los estados *s[c]* de este "constraint solver" deberían de contener esta información adicional. Expresaremos dicho estado como un par $s[c] = < \theta, c >$. Describimos la relación de transición de este nuevo "constraint solver" mediante las siguientes dos reglas.

(1) Regla de éxito

$$\frac{\textit{existe un E-unificador } \theta' \textit{ de } \tilde{c}\theta}{< \theta, c > \xrightarrow{\tilde{c}} < \theta\theta', c \cup \tilde{c} >}$$

(2) Regla de reinicio

$$\frac{\text{no existe } E\text{-unificador de } \tilde{c}\theta \ \wedge \text{ existe un } E\text{-unificador } \theta' \text{ de } c \cup \tilde{c}}{< \theta, c > \xrightarrow{\tilde{c}} < \theta', c \cup \tilde{c} >}$$

Se debe notar que si se aplica la regla (2), entonces no existe ninguna instancia de la substitución $\theta$ que pueda resolver la restricción $c \cup \tilde{c}$. Esta consideración sugiere un método para podar el árbol de búsqueda cuando se añade incrementalmente una nueva restricción y se ha de comprobar su consistencia, como mostraremos posteriormente. Otra consideración importante relacionada con la incrementalidad será cómo representar una restricción satisfacible en una forma simplificada.

En [van Hentenryck 90] también se estudia el problema de la satisfacción incremental de restricciones. Se muestra que una aproximación *generación y test* basada en "backtracking" es inadecuada ya que sólo es capaz de usar pasivamente las nuevas restricciones para comprobar si la solución tentativa que fue generada para el conjunto de restricciones anterior sigue siendo útil (comprobando si resuelve también el nuevo conjunto). En el contexto de CLP parece más apropiado usar activamente las nuevas restricciones para guiar la búsqueda hacia la solución. Consecuentemente, en [van Hentenryck 90] se presenta un esquema basado en reejecución y poda del árbol de búsqueda de la SLD-resolución. Nosotros tenemos un problema similar pero relacionado con el árbol de búsqueda del algoritmo de resolución de restricciones.

En la próxima subsección se presenta un algoritmo incremental para estudiar la solubilidad de las restricciones en la estructura *H/E* y se define una representación simplificada y apropiada para las mismas. Dicho algoritmo, *ICS* ("*I*ncremental *C*onstraint *S*olver"), se describe por medio de un sistema de transición etiquetado sobre el conjunto de las $(\Pi_C,\Sigma)$-restricciones y diseñado especialmente para ser incremental.

### 3.3.1.1. El sistema de transición *ICS*.

**Lema 3.3.1.1.1**

Sea $c$ una restricción *H/E-soluble*. Supongamos que $c$ puede escribirse de la forma $c_1 \cup c_2$ (pudiendo ser cualquiera de ellas vacía) donde $c_1$ tiene un único *E-unificador más general (E-mgu)* $\theta$. Entonces $H/E \models (c \Leftrightarrow \backslash O(\hat{},\theta) \cup c_2)$, i.e., la restricción $c$ tiene el mismo conjunto de *soluciones* sobre la estructura *H/E* que la restricción simplificada $\backslash O(\hat{},\theta) \cup c_2$.

**Demostración.**

Denotamos el conjunto (posiblemente vacío) de *H/E-soluciones* de una restricción $c$ por $Solns_E(c)$. Es inmediato que $Solns_E(c' \cup c'') = Solns_E(c') \cap Solns_E(c'')$ [Lassez et al 88]. Por tanto, $Solns_E(c) = Solns_E(c_1 \cup c_2) = Solns_E(c_1) \cap Solns_E(c_2) = Solns_E(\O(\hat{},\theta)) \cap Solns_E(c_2) = Solns_E(\O(\hat{},\theta) \cup c_2)$. □

A continuación definimos una forma simplificada para las restricciones.

**Definición 3.3.1.1.2.** *(ICS-representación de una restricción H/E-soluble).*

Sea $c$ una restricción *H/E-soluble*. Supongamos que $c$ puede escribirse de la forma $c_1 \cup c_2$ (pudiendo ser cualquiera de ellas vacía) donde $c_1$ tiene un único *E-mgu* $\theta$. Entonces $r_c = (\O(\hat{},\theta), c_2)$ es la *ICS-representación* de la restricción simplificada $\O(\hat{},\theta) \cup c_2$. La *ICS-representación* de una restricción vacía es $(\varnothing, \varnothing)$, que será abreviada como $( )$. Se debe notar que si $r_c = (\O(\hat{},\theta), \varnothing)$ entonces la restricción $c$ tiene un único *E-mgu*.

**Definición 3.3.1.1.3.** *(ICS-estados)*

Definimos un estado $s$ del "Incremental Constraint Solver" como un triplete $s = < \theta, f, r_c >$, donde $r_c$ es la *ICS-representación* de una $(\Pi_C,\Sigma)$-restricción $c$, $\theta$ representa un *E-unificador* de $c$ y $f$ es un conjunto de $(\Pi,\Sigma)$-substituciones.

Informalmente, para el algoritmo de resolución de restricciones que será descrito a continuación, $r_c$ representa la restricción simplificada acumulada y $f$ representa el conjunto de substituciones que ya han sido probadas sin éxito por el "constraint solver" y que son útiles para una búsqueda heurística de otras soluciones.

**Definición 3.3.1.1.4.** *(ICS-estado vacío, ICS-estados terminales).*

El triplete $< \varepsilon, \varnothing, ( ) >$ representa el estado *vacío* del "constraint solver". Cualquier *ICS-estado* puede ser un *ICS-estado* terminal.

**Definición 3.3.1.1.5.** *(ICS-etiquetas).*

Una *ICS-etiqueta* $\tilde{c}$ es una $(\Pi_C,\Sigma)$-restricción. Representa una nueva restricción $\tilde{c}$ que se ha de añadir a la restricción simplificada acumulada $c$ una vez comprobada la solubilidad del nuevo conjunto $c \cup \tilde{c}$.

**Lema 3.3.1.1.6.**

Sean $c$, $\tilde{c}$ dos restricciones *H/E-solubles*. Supongamos que $\theta$ es un *E-unificador* de $c$. Si $\theta'$ es *E-unificador* de la restricción $\tilde{c}\theta$ entonces $\theta\theta'$ es un *E-unificador* de $c \cup \tilde{c}$.

**Demostración.**

Debemos mostrar que $\theta\theta'$ es un *E-unificador* de $c \cup \tilde{c}$. Resulta inmediato que si una substitución $\gamma$resuelve una ecuación entonces cualquier instancia $\gamma\gamma'$ de la misma también la resuelve [Bürckert 88, Jaffar Stuckey 86a]. Dado que $\theta$ es un *E-unificador* de $c$, también $\theta\theta'$ es un *E-unificador* de $c$. Además, puesto que $\theta'$ es *E-unificador* de $\tilde{c}\theta$, entonces $\theta\theta'$ es un *E-unificador* de $\tilde{c}$. En consecuencia, $\theta\theta'$ es un *E-unificador* tanto de $c$ como de $\tilde{c}$. □

**Lema 3.3.1.1.7.**

Sean $c$, $\tilde{c}$ dos restricciones. Supongamos que $c$ tiene un único *E-mgu* $\theta$. Si $\theta'$ es el único *E-mgu* de la restricción $\tilde{c}\theta$ entonces $\theta\theta'$ es el único *E-mgu* de $c \cup \tilde{c}$. Si no existe un *E-unificador* de $\tilde{c}\theta$ entonces tampoco $c \cup \tilde{c}$ es *E-unificable*.

**Demostración.**

Si $\theta'$ es el único *E-mgu* de la restricción $\tilde{c}\theta$ entonces $Solns_E(c \cup \tilde{c}) = Solns_E(c) \cap Solns_E(\tilde{c}) = Solns_E(\O(\hat{},\theta)) \cap Solns_E(\tilde{c}) = Solns_E(\O(\hat{},\theta)) \cap Solns_E(\tilde{c}\theta) = Solns_E(\O(\hat{},\theta)) \cap Solns_E(\O(\hat{},\theta)') = Solns_E(\O(\hat{},\theta) \cup \O(\hat{},\theta)') = Solns_E(\O(\hat{}, \theta\theta'))$ lo que demuestra la primera parte del lema.

Si no existe un *E-unificador* de $\tilde{c}\theta$ entonces $Solns_E(c \cup \tilde{c}) = Solns_E(\O(\hat{},\theta)) \cap Solns_E(\tilde{c}\theta) = Solns_E(\O(\hat{},\theta)) \cap \varnothing = \varnothing$ lo que completa la prueba. □

A continuación se describe el sistema de transición *ICS*. Sea $s = < \theta, f, r_c >$ *un ICS-estado*. Informalmente, para conseguir incrementalidad, el coste de resolver el nuevo conjunto de restriciones $c \cup \tilde{c}$ debería ser aproximadamente igual al coste de resolver la nueva restricción $\tilde{c}$ más el coste de "combinar" la nueva solución con la solución anterior. Dado que la substitución $\theta$representa un *E-unificador* de la restricción acumulada simplificada $c$, esta consideración sugiere buscar un *E-unificador* del nuevo

conjunto $c \cup \tilde{c}$ buscando soluciones para $\tilde{c}\theta$. Si no existe un *E-unificador* de $\tilde{c}\theta$ habrá que empezar de cero y buscar las soluciones de la restricción total $c \cup \tilde{c}$. La substitución $\theta$ puede ser entonces añadida al conjunto $f$ de substituciones ensayadas sin éxito. Utilizamos un procedimiento de "*N*arrowing" *C*ondicional *H*eurístico (HCNC) *N*, que será descrito posteriormente, para la búsqueda de una solución $\theta'$ para $\tilde{c}\theta$ y combinamos adecuadamente $\theta$ y $\theta'$ para obtener una nueva solución acumulada y una nueva restricción acumulada simplificada.

**Definición 3.3.1.1.8.** *(ICS-relación de transición*[17] $\xrightarrow{\tilde{c}}_{ICS}$*)*

Reglas de solución única

(1)

$$\frac{N(\tilde{c}\theta, f) \text{ tiene única solución } \theta'}{< \theta, f, (\backslash O(\hat{}, \theta), \varnothing) > \xrightarrow{\tilde{c}}_{ICS} < \theta\theta', f, (\backslash O(\hat{}, \theta\theta'), \varnothing) >}$$

(2)

$$\frac{N(\tilde{c}\theta, f) \text{ tiene única solución } \theta' \wedge N((c_2 \cup \tilde{c})\theta_1, f) \text{ tiene única solución } \theta''}{< \theta, f, (\backslash O(\hat{}, \theta)_1, c_2) > \xrightarrow{\tilde{c}}_{ICS} < \theta_1\theta'', f, (\backslash O(\hat{}, \theta_1\theta''), \varnothing) >}$$

Reglas de solución múltiple

(3)

$$\frac{N(\tilde{c}\theta, f) \text{ tiene única solución } \theta' \wedge N((c_2 \cup \tilde{c})\theta_1, f) \text{ tiene una primera solución } \theta''}{< \theta, f, (\backslash O(\hat{}, \theta)_1, c_2) > \xrightarrow{\tilde{c}}_{ICS} < \theta\theta', f, (\backslash O(\hat{}, \theta)_1, c_2 \cup \tilde{c}) >}$$

(4)

$$\frac{N(\tilde{c}\theta, f) \text{ tiene una primera solución } \theta'}{< \theta, f, (\backslash O(\hat{}, \theta)_1, c') > \xrightarrow{\tilde{c}}_{ICS} < \theta\theta', f, (\backslash O(\hat{}, \theta)_1, c' \cup \tilde{c}) >}$$

Regla de reinicio

(5)

[17] Por simplicidad, en este conjunto de reglas $c_2$ denota una restricción no vacía.

$$\frac{N(\tilde{c}\theta, f) \text{ no tiene solución} \wedge \; < \varepsilon, f \cup \theta_{|Var(G0)}, ( ) > \; \xrightarrow{(c_2 \cup \tilde{c})\theta_1}_{ICS} \; s}{< \theta, f, (\backslash O(\hat{}, \theta)_1, c_2) > \; \xrightarrow{\tilde{c}}_{ICS} \; s}$$

Se debe notar que cada transición de este sistema depende de la terminación del procedimiento de "narrowing" *N*. Asumimos que si *N(c, f)* termina devolviendo la salida *no solución* entonces la restricción $c$ no es *E-unificable,* si la salida es *única solución* $\theta$ entonces $\theta$ es el único *E-mgu* de $c$ y si la salida es *primera solución* $\theta$ entonces $\theta$ es un *E-unificador* de $c$ pero, posiblemente, no es el único *E-mgu* de *c*. A continuación definiremos un sistema de transición para CLP(*H*/*E*) que depende del cálculo *ICS*. La relación de transición para CLP(*H*/*E*) dependerá de un paso de transición único del sistema *ICS*. Así pues, no es necesario incluir una prueba de la terminación del sistema *ICS*. El siguiente teorema establece la corrección del cálculo *ICS*.

**Teorema 3.3.1.1.9.** *(Corrección de ICS)*

Si es posible probar una transición $< \theta, f, r_c > \; \xrightarrow{\tilde{c}}_{ICS} \; < \theta', f', r'_c >$, entonces la restricción $c \cup \tilde{c}$ es *H*/*E*-soluble. $r'_c$ representa la forma simplificada de esta restricción y $\theta'$ representa un *E-unificador* de la misma. $f'$ representa el conjunto de substituciones intentadas sin éxito.

**Demostración.**

Demostraremos que la afirmación es cierta para cada regla. Se debe notar que en cada paso se puede elegir una y sólo una regla.

- Supongamos que se ha aplicado la regla (1). Esta regla sólo se puede aplicar si se tiene una restricción $c$ cuyo único *E-mgu* es $\theta$. Por el lema 3.3.1.1.7, si $\tilde{c}\theta$ tiene un único *E-mgu* $\theta'$ entonces $c \cup \tilde{c}$ es *H/E-soluble* y $\theta\theta'$ es su único *E-mgu*.

- Supongamos que se ha aplicado la regla (2). Esta regla sólo se puede aplicar si se tiene una restricción $c$ que ha sido simplificada a $\backslash O(\hat{}, \theta)_1 \cup c_2$ y para la cual se conoce un *E-unificador* $\theta$, y si se ha encontrado un único *E-mgu* $\theta'$ para $\tilde{c}\theta$. El lema 3.3.1.1.6 garantiza entonces que $\theta\theta'$ es un *E-unificador de* $c \cup \tilde{c}$. Además, dado que $\backslash O(\hat{}, \theta)_1$ tiene un único *E-mgu* $\theta_1$ y que $\theta''$ es el único *E-mgu* de $(c_2 \cup \tilde{c})\theta_1$, el lema 3.3.1.1.7 garantiza que $\theta_1\theta''$ es el único *E-mgu* de

$(\O(\hat{},\theta)_1 \cup \O(c_2 \cup \tilde{c}))$ y, por el lema 3.3.1.1.1, la restricción simplificada $\O(\hat{}, \theta_1\theta'')$ tiene el mismo conjunto de *H/E-soluciones* que $(\O(\hat{},\theta)_1 \cup (c_2 \cup \tilde{c}) = c \cup \tilde{c}$.

- Supongamos que se ha aplicado la regla (3). Esta regla sólo se puede aplicar si se tiene una restricción $c$ que ha sido simplificada a $\O(\hat{},\theta)_1 \cup c_2$ y para la cual se conoce un *E-unificador* $\theta$, y si se ha encontrado un único *E-mgu* $\theta'$ para $\tilde{c}\theta$. El lema 3.3.1.1.6 garantiza entonces que $\theta\theta'$ es un *E-unificador de* $c \cup \tilde{c}$. Además, dado que $\O(\hat{},\theta)_1$ tiene un único *E-mgu* $\theta_1$, el lema 3.3.1.1.1 garantiza que la restricción $c \cup \tilde{c}$ tiene el mismo conjunto de *H/E-soluciones* que la restricción simplificada $(\O(\hat{},\theta)_1 \cup (c_2 \cup \tilde{c})$.

- Supongamos que se ha aplicado la regla (4). Esta regla sólo se puede aplicar si se tiene una restricción $c$ que ha sido simplificada a $\O(\hat{},\theta)_1 \cup c'$ y para la cual se conoce un *E-unificador* $\theta$, y si se ha encontrado un *E-unificador* $\theta'$ para $\tilde{c}\theta$. El lema 3.3.1.1.6 garantiza entonces que $\theta\theta'$ es un *E-unificador de* $c \cup \tilde{c}$ Además, dado que $\O(\hat{},\theta)_1$ tiene un único *E-mgu* $\theta_1$ el lema 3.3.1.1.1 garantiza que la restricción $c \cup \tilde{c}$ tiene el mismo conjunto de *H/E-soluciones* que la restricción simplificada $(\O(\hat{},\theta)_1 \cup (c' \cup \tilde{c})$.

- En la condición de la regla (5) se requiere demostrar la transición $< \varepsilon, f \cup \theta_{|Var(G_0)}, (\,) \xrightarrow{(c_2 \cup \tilde{c})\theta_1}_{ICS} s$. Dado que $(\,) = (\varnothing,\varnothing)$, las reglas (1) y (4) son las únicas que pueden ser aplicadas para derivar esta transición[18]. Por tanto, la regla (5) puede reemplazarse equivalentemente por las siguientes dos reglas:

$(5^{i})$

$$\frac{\tilde{c}\theta, f) \text{ no tiene solución} \wedge N((c_2 \cup \tilde{c})\theta_1, f \cup \theta_{|Var(G_0)}) \text{ tiene única solución } \theta'}{< \theta, f, (\O(\hat{},\theta)_1, c_2) > \xrightarrow{\tilde{c}}_{ICS} < \theta', f \cup \theta_{|Var(G_0)}, (\O(\hat{},\theta)', \varnothing) >}$$

$(5^{ii})$

$$\frac{\tilde{c}\theta, f) \text{ no tiene solución} \wedge N((c_2 \cup \tilde{c})\theta_1, f \cup \theta_{|Var(G_0)}) \text{ tiene una primera solución } \theta'}{< \theta, f, (\O(\hat{},\theta)_1, c_2) > \xrightarrow{\tilde{c}}_{ICS} < \theta', f \cup \theta_{|Var(G_0)}, (\O(\varnothing), \O(\hat{},\theta)_1 \cup (c_2 \cup \tilde{c}) >}$$

[18] Se debe notar que en la regla (4), c' será vacía.

- Supongamos que se ha aplicado la regla (5$^{i}$). Esta regla sólo se puede aplicar si se tiene una restricción $c$ que ha sido simplificada a $\O(\hat{},\theta)_1 \cup c_2$ y para la cual se conoce un *E-unificador* $\theta$. Si no hay solución para $\tilde{c}\theta$ entonces ninguna instancia de $\theta$ puede resolver $c \cup \tilde{c} = \O(\hat{},\theta)_1 \cup \O(c_2 \cup \tilde{c})$. Por tanto, se debe descartar la solución acumulada $\theta$ y estudiar la solubilidad de la restricción completa $(\O(\hat{},\theta)_1 \cup \O(c_2 \cup \tilde{c}))$. La substitución $\theta_{|Var(G_0)}$ ha de añadirse al conjunto $f$ de substituciones que servirán para guiar la búsqueda de otras soluciones. Finalmente, si se encuentra una única solución para $(c_2 \cup \tilde{c})\theta_1$, por un argumento similar al usado para la regla (2), la prueba está completa.

- Supongamos que se ha aplicado la regla (5$^{ii}$). Analogamente a (5$^{i}$), la substitución $\theta_{|Var(G_0)}$ ha de añadirse al conjunto $f$ de substituciones que servirán para guiar la búsqueda de otras soluciones. Además, esta regla sólo se puede aplicar si no hay solución para $\tilde{c}\theta$ y se encuentra una solución para $(c_2 \cup \tilde{c})\theta_1$. De nuevo, por un argumento similar al usado para la regla (4), la propiedad enunciada se satisface y $(\O(\varnothing), \O(\hat{},\theta)_1 \cup \O(c_2 \cup \tilde{c}))$ representa la restricción en forma no simplificada. □

En general, el algoritmo descrito por el cálculo *ICS* no es completo porque el procedimiento de "narrowing" puede no terminar durante la evaluación de la condición de una regla no sólo cuando la restricción no es *E-unificable* sino también cuando existe un único *E-mgu* que ya ha sido encontrado y se intenta encontrar otra solución.

### 3.3.1.2. Algoritmo de "Narrowing" *C*ondicional *H*eurístico.

En este apartado describimos el algoritmo de "narrowing" condicional que se utiliza como núcleo del mecanismo incremental de resolución de restricciones *ICS* descrito en la subsección §3.3.1.1. El algoritmo de "narrowing" se define mediante un cálculo ("*H*euristic *C*onditional *N*arrowing *C*alculus", *HCNC*) que es guiado de forma heurística por las sustituciones descartadas (en computaciones previas) durante la búsqueda de soluciones a otras restricciones. *HCNC* es una adaptación del Algoritmo de "Narrowing" Condicional CNA presentado en [Hussmann 86], donde se demuestra su corrección y completitud para sistemas de reescritura condicionales canónicos sin variables extra. En este apartado extendemos el algoritmo CNA utilizando activamente información heurística para converger rápidamente hacia una solución.

Como se comentó en la subsección §2.1.3, el procedimiento de "narrowing" fue introducido originalmente en el campo de la demostración automática de teoremas

[Lankford 75, Slagle 74] como una versión restringida de la regla de paramodulación [Robinson Wos 69] para sistemas de reescritura canónicos. En [Fay 79] se describió un procedimiento de "narrowing" para resolver problemas de unificación bajo una teoría ecuacional de primer orden definida por un sistema de reescritura de términos $R$ no condicional y canónico. Dicho procedimiento fue mejorado posteriormente por Hullot [Hullot 80].

Técnicamente, "narrowing" combina unificación de términos y reescritura. Realizar un paso de "narrowing" sobre un término $t$ supone reemplazar $t$ por $(t[u \leftarrow r])\theta$, donde $t/u$ es un subtérmino no variable de $t$, $l \rightarrow r$ es una copia de una regla en $R$ con las variables renombradas con nombres de variable nunca antes usados ("standardized apart") y $\theta$ es el mgu del subtérmino $t/u$ y la parte izquierda (lhs) $l$ de la regla. La relación de "narrowing" $\Rightarrow_R$ así obtenida extiende la relación de reescritura $\rightarrow_R$ ya que cualquier paso de reescritura (sobre términos sin variables) también es un paso de "narrowing". Para términos sin variables, reducción y "narrowing" coinciden. Sin embargo, la primera relación no siempre es un subconjunto de la segunda. Por ejemplo, si $R = \{X \rightarrow a\}$ entonces $f(Y) \rightarrow_R f(a)$ pero $\neg(f(Y) \Rightarrow_R f(a))$.

El procedimiento de unificación ecuacional de Fay define una estrategia de "narrowing" *normalizante* (donde el término obtenido después de cada paso de "narrowing" propio se reescribe a su forma normal). La ventaja de esta estrategia es que reduce el espacio de búsqueda y ha sido usada, entre otros, por [Krisker Bockmayr 91, Réty et al 85] en el caso no condicional y por [Dershowitz Plaisted 85, Fribourg 85b, Hussmann 86, Hölldobler 89] en el condicional. Para implementaciones de "narrowing" con normalización ver, e.g., [Hanus 90a, Cheong Fribourg 91]. Hullot introdujo la noción de "narrowing" *básico,* que también conduce a una reducción del espacio de búsqueda, y que se define restringiendo el conjunto de ocurrencias a las que se permite aplicar pasos de "narrowing" (sólo se aplica "narrowing" a términos que no han sido introducidos por instanciación en un paso de "narrowing" previo). La prueba de la completitud del "narrowing" *básico* se basa en el paralelismo que se puede establecer entre la secuencia de reducciones aplicadas a una instancia de un término y la secuencia de pasos de "narrowing" que es aplicable a dicho término. Este resultado es comparable al lema del "lifting" en la teoría de la resolución [Robinson 65]. Sin embargo, es posible restringir aún más el conjunto de ocurrencias sin perder completitud, eliminando también aquellas que quedaron a la izquierda de la ocurrencia en la que se aplicó el anterior paso de "narrowing". Esta estrategia es la base del "narrowing *left-to-right*" que se discute en [Herold 86] y en [Krisker Bockmayr 91], donde se presentan además resultados cuantitativos acerca de su eficiencia.

En [Bosco et al 87, Bosco et al 88] se introduce la estrategia de "narrowing" *selección,* basada en una idea semejante a la que condujo al refinamiento del procedimiento de resolución con función de selección: en cada paso se selecciona sólo un subtérmino y el resto de posibles elecciones de subtérmino para el mismo paso no son tomadas en consideración para la búsqueda. [Middeldorp 91b] y [Yamamoto 87] complementan la prueba de completitud de Bosco y Giovannetti.

La simple combinación de "narrowing" *básico* y *normalizante* no es completa porque la computación del conjunto de ocurrencias básicas de una secuencia de reescrituras es más complicada que la de las derivaciones por "narrowing" [Réty 87]. Nutt, Réty y Smolka [Nutt et al 89] generalizaron la definición de "narrowing" *básico* y estudiaron su completitud y sus optimizaciones cuando se combina con "narrowing" *normalizante*. Mostraron también cómo obtener la estrategia "*left to right*" de Herold y la estrategia de "narrowing" *selección* de Bosco a partir de la definición general. En [Padawitz 88] se examinan diferentes refinamientos del "narrowing" y se demuestra la corrección de las optimizaciones. [Echahed 88] y [Padawitz 87] estudiaron criterios para asegurar la completitud de distintas estrategias de selección para "narrowing".

La estrategia de "narrowing" *perezoso* proviene de [Reddy 85] y se revisa exhaustivamente en [Pull 90]. Existe una caracterización exacta de las derivaciones realmente perezosas. Son de la forma "outside-in". Este concepto fue introducido por [Huet Levy 79] y ha sido adaptado más tarde a la disciplina de constructores por [You 89].

[Dershowitz Plaisted 85], [Fribourg 85a], [Giovannetti Moiso 86], [Hölldobler 89], [Hussmann 86] y [Kaplan 87] investigaron el "narrowing" para sistemas de reescritura condicionales. [Hussmann 86] extiende el trabajo de Fay/Hullot a teoría definidas por sistemas confluentes (posiblemente no terminantes) condicionales y presenta un resultado de completitud respecto a soluciones normalizadas que se satisface si el sistema es no trivial (no hay ninguna regla cuya lhs es una variable) y se impone la condición habitual de ausencia de variables extra [Hölldobler 89]. Si el sistema es noetheriano, el resultado de completitud se extiende a todas las soluciones. [Giovannetti Moiso 86] estudiaron las consecuencias de permitir la presencia de variables extra en los cuerpos de las reglas y dieron una condición suficiente para garantizar la completitud, basada en el requerimiento de *confluencia por niveles* (ver definición 3.1.15). [Fribourg 85a] define un procedimiento de "narrowing" guiado por una estrategia "innermost". En [Hölldobler 89] se generalizan los resultados de Fribourg, Hullot y Hussmann.

A continuación revisamos el Algoritmo de Narrowing Condicional CNA presentado en [Hussmann 86].

Llamamaremos (sub-)objetivo a una expresión de la forma $c$ *with* $\tau$ donde $c$ es un conjunto de ecuaciones y $\tau$ es una substitución. Asumimos que la substitución $\tau$ está restringida a las variables de $c$ y que las variables de $c$ han sido estandarizadas aparte, i.e., el conjunto de variables en $c$ y en las cláusulas ecuacionales del programa son disjuntos. Para resolver $c$, el algoritmo empieza con el objetivo $c$ *with* $\varepsilon$ e intenta derivar subobjetivos (recordando en las partes *with* las substituciones aplicadas) hasta que se alcanza un objetivo terminal, de la forma $\varnothing$ *with* $\theta$. Cada substitución $\theta$ en un objetivo terminal es un *E-unificador* de $c$. Por abuso de notación, se le llama *solución* [Bürckert 88, Bürckert et al 87].

Sea $O'(c)$ el conjunto de ocurrencias no variables de $c$. Definimos la siguiente relación:

$$narred(\ c,u,k,\sigma\ ) \Leftrightarrow\ (u \in O'(c) \wedge\ (l_k = r_k \leftarrow \tilde{e}_k\ ) \in E \wedge\ \sigma\ = mgu(c/u,\ l_k))$$

El cálculo CNA se define por medio de las siguintes dos reglas:

regla de unificación CNA

$$\frac{c \text{ unifica sintácticamente con mgu } \sigma}{c \text{ with } \tau\ \rightarrow_{CNA}\ \varnothing \text{ with } \tau\sigma}$$

regla de narrowing CNA

$$\frac{narred(\ c,\ u,\ k,\ \sigma)}{c \text{ with } \tau\ \rightarrow_{CNA}\ \tilde{e}_k\ ,\ c[u \leftarrow r_k]\ )\sigma \text{ with } \tau\sigma}$$

Este cálculo define un algoritmo ya que todas las CNA-*derivaciones* (secuencias de estados) a partir un *objetivo* cualquiera pueden ser enumeradas fácilmente. Estas derivaciones pueden representarse mediante un árbol ramificado finitamente (aunque posiblemente infinito). Los nodos de este árbol han de ser visitados siguiendo una estrategia de búsqueda completa (e.g. *primero en amplitud*, "breadth first") durante la búsqueda de nodos unificables.

En lo que sigue presentamos el algoritmo de "*N*arrowing" *C*ondicional *H*eurístico (*HCNC*).

**Definición 3.3.1.2.1.** (*N-estado*)

Un *N-estado* es un triplete $< n, \theta, L >$, donde $n$ es un entero positivo $\in \{0, 1, 2\}$, $\theta$ es una $(\Pi,\Sigma)$-*substitución* y $L$ es una lista (posiblemente vacía) $[g_i]_{i=1}^{n}$ de subobjetivos $g_i = c_i$ *with* $\tau_i$, donde $c_i$ es un conjunto de $(\Pi_C,\Sigma)$-*restricciones (*una conjunción de ecuaciones*)* y $\tau_i$ es una $(\Pi,\Sigma)$-*substitución*. Denotaremos los constructores de listas mediante $[\ ]$ y $\bullet$. Por abuso de notación, asumimos que el operador $\bullet$ ha sido extendido homomórficamente como operador de concatenación de listas.

Informalmente, la primera componente de un *N-estado* representa el número actual de soluciones. La segunda componente representa la solución a devolver y la tercera componente representa la lista de subobjetivos que quedan aún por reducir. Los nodos del árbol de búsqueda de "narrowing" se almacenan en esta lista. La lista es tratada como una cola para simular una estrategia de búsqueda *en amplitud* en el árbol.

Para resolver la conjunción de ecuaciones $c$ utilizando la información heurística $f$, el algoritmo empieza en el *N-estado* inicial $N_0 = < 0, \varepsilon, [c \text{ with } \varepsilon] >$ e intenta derivar subobjetivos (recordando en las partes *with* las substituciones aplicadas) hasta alcanzar un *N-estado* terminal $< n, \theta, [\ ] > /\!\rightarrow_{\text{HCNC}}$.

El conjunto $f$ de substituciones ensayadas sin éxito por el "constraint solver" va a ser utilizado para una búsqueda heurística de otras soluciones. Si $\rho$ es una substitución que pertenece a $f$ entonces, por el teorema de compacidad de la lógica de primer orden, $\rho$ no puede resolver la nueva restricción acumulada $c$ ya que hay un subconjunto de $c$ que no puede ser resuelto. Más aún, tampoco podrá resolver $c$ ninguna substitución $\rho'$ tal que, restringida a las variables de $\rho$, sea instancia de $\rho$.

Definimos la siguiente relación $\preceq$ entre sustituciones:

$$\rho \preceq \rho' \Leftrightarrow (\exists\alpha).\ \rho'_{|Var(\rho)} = \rho\alpha$$

Sea $< n, \theta, L >$ un *N-estado* y sea $c_i$ *with* $\tau_i$ un subobjetivo incluído en la lista $L$. Sea $S = \{< u, k, \sigma >: narred(\ c_i,u,k,\sigma\ )\}$. Si $< u', k', \sigma' > \in S$ y $(\exists\rho \in f)$ tal que $\rho \not\preceq \tau_i\sigma'$ entonces el correspondiente subárbol del árbol de búsqueda que tiene como raiz el subobjetivo $(\ \tilde{e}_{k'},\ c_i[u' \leftarrow r_{k'}]\ )\sigma'$ *with* $\tau_i\sigma'$ no contiene ninguna solución.

Dado que es inútil explorar este subárbol, podaremos el árbol de búsqueda eliminando la representación de su raiz ($< u', k', \sigma' >$) del conjunto *S*.

Podemos considerar el conjunto $f$ de substituciones descartadas como una caracterización de "no unificabilidad". Así, la poda basada en el conjunto $f$ supone una mejora comparable a la de un "backtracking" inteligente perfecto. En programación lógica, las técnicas de "backtracking" inteligente [Bruynooghe Pereira 84, Wolfram 86] analizan los fallos del procedimiento de unificación para evitar el efecto de "thrashing", que surge cuando el "backtracking" no detecta que ciertas decisiones tomadas anteriormente nunca conducirán a una solución y, por tanto, se exploran subárboles del espacio de búsqueda que, como era predecible, no contienen ninguna rama de éxito. Dado que el número de nodos en éstos subárboles puede crecer exponencialmente con la altura del árbol, las técnicas de "backtracking" inteligente prometen obtener mejoras sustanciales al reducir la posibilidad de ocurrencia de este fenómeno. La característica común con nuestra aproximación está en el hecho de analizar el fallo del procedimiento de unificación y en el aprender de este fallo para prevenir su repetición, usándolo para la poda del árbol de búsqueda. Sin embargo, hay una cuestión básica que compromete la viabilidad de las técnicas de "backtracking" inteligente (y, en general, la de la mayoría de los refinamientos de los métodos de búsqueda) que no concurre en nuestra aproximación: el conflicto que se establece entre la ganancia obtenida con el refinamiento del método y el precio pagado, que puede ser prohibitivo (e.g., la diferencia entre la sobrecarga en tiempo de ejecución debida al refinamiento y el tiempo que hubiera sido requerido para la búsqueda en las ramas de fallo). Sin embargo, nosotros obtenemos la mejora con el simple test de la relación $\leq$ entre substituciones, que tiene una complejidad computacional lineal con la longitud de la $\rightarrowtail_{CLP(H/E)}$*-derivación.*

**Definición 3.3.1.2.2.** *C*álculo de "Narrowing" *C*ondicional *H*eurístico (*HCNC*)

El cálculo *HCNC* se define por medio del siguiente sistema de transición[19] estratificado:

Reglas de ramificación heurística

(1)

$$\frac{\{< u, k, \sigma >: \ narred(\ c, u, k, \sigma\ )\} = \varnothing}{c \text{ with } \tau \ \rightarrow_{Branch}\ [\ ]}$$

(2)

[19]Los estados del sistema de transición para *Branch* son subobjetivos o listas de subobjetivos.

$$\frac{[< u_i, k_i, \sigma_i >]_{i=1}^{n} = s\_to\_l\ (\{< u, k, \sigma >:\ narred(\ c, u, k, \sigma\ ) \wedge \{\rho \in f:\ \rho \leq \tau\sigma\} = \varnothing\})}{c\ with\ \tau \rightarrow_{Branch} [(\backslash O(e_{k_i}, \sim), c[u_i \leftarrow r_{k_i}]\ )\sigma_i\ with\ \tau\sigma_i]_{i=1}^{n}}$$

Reglas de éxito

(3)

$$\frac{c\ with\ \tau \rightarrow_{Branch} L' \wedge\ c\ unifica\ sintácticamente\ con\ mgu\ \sigma}{< 0, \varepsilon,\ c\ with\ \tau \bullet L > \rightarrow_{HCNC} < 1, \tau\sigma,\ L \bullet L' >}$$

(4)

$$\frac{c\ with\ \tau \rightarrow_{Branch} L' \wedge\ c\ unifica\ sintácticamente\ con\ mgu\ \sigma \wedge\ \neg(\tau\sigma \leq, \theta) \wedge\ \neg(\theta \leq \tau\sigma)}{< 1, \theta,\ c\ with\ \tau \bullet L > \rightarrow_{HCNC} < 2, \theta,\ [\ ] >}$$

Reglas de narrowing

(5)

$$\frac{c\ with\ \tau \rightarrow_{Branch} L' \wedge\ c\ unifica\ sintácticamente\ con\ mgu\ \sigma \wedge\ \theta \leq \tau\sigma}{< 1, \theta,\ c\ with\ \tau \bullet L > \rightarrow_{HCNC} < 1, \theta,\ L \bullet L' >}$$

(6)

$$\frac{c\ with\ \tau \rightarrow_{Branch} L' \wedge\ c\ unifica\ sintácticamente\ con\ mgu\ \sigma \wedge\ \tau\sigma \leq \theta}{< 1, \theta,\ c\ with\ \tau \bullet L > \rightarrow_{HCNC} < 1, \tau\sigma,\ L \bullet L' >}$$

(7)

$$\frac{c\ with\ \tau \rightarrow_{Branch} L' \wedge\ mgu(c) = \varnothing \wedge\ n \in \{0,1\}}{< n, \theta,\ c\ with\ \tau \bullet L > \rightarrow_{HCNC} < n, \theta,\ L \bullet L' >}$$

donde la función de elección *s_to_l(S)* devuelve una lista constituida por los elementos del conjunto *S*.

**Definición 3.3.1.2.3.** (comportamiento de *HCNC*)

Sea $N_0$ un *N-estado inicial*. Definimos la función:

$N(c, f) =$ *no solución si* $N_0 \rightarrow^*_{HCNC} <0, \varepsilon, [\ ] >$,
$N(c,f) =$ *única solución* $\theta$ *si* $N_0 \rightarrow^*_{HCNC} < 1, \theta, [\ ] >$,
$N(c,f) =$ *primera solución* $\theta$ *si* $N_0 \rightarrow^*_{HCNC} < 2, \theta [\ ]$

Se debe notar que si $f = \varnothing$ entonces el algoritmo anterior explora el prefijo del árbol de búsqueda de CNA que hubiera sido explorado siguiendo una estrategia de búsqueda *primero en amplitud* hasta que las condiciones que caracterizan un *N-estado*

*terminal* hubieran sido alcanzadas. Si *HCNC* no termina o termina con $L = [\ ]$ y $n \in \{0,1\}$ entonces $N(c, f)$ es equivalente a CNA dado que explora exactamente el mismo árbol.

Cuando $f \neq \varnothing$, la poda heurística basada en $f$ sólo elimina subárboles que no contienen ninguna solución, así que está garantizado que no se pierde ninguna solución.

El siguiente teorema establece la corrección del algoritmo de "Narrowing" *C*ondicional *H*eurístico $N(c, f)$.

**Teorema 3.3.1.2.4.** (corrección del "Narrowing" *C*ondicional *H*eurístico)

Sea $E$ una teoría ecuacional de Horn (condicional) canónica sin variables extra. Sea $f$ un conjunto de substituciones tales que ninguna instancia de ninguna de ellas puede resolver $c$. Si $N(c, f)$ devuelve la salida *no solución* entonces la restricción $c$ no es *E-unificable*, si la salida es *única solución* $\theta$ entonces $\theta$ es el único *E-mgu* de $c$, si la salida es *primera solución* $\theta$ entonces $\theta$ es un *E-unificador* de $c$.

**Demostración.**

- Supongamos que se devuelve la salida *no solución*. Entonces el árbol entero de CNA ha sido explorado (exceptuando subárboles que no contenían ninguna solución). Dado que CNA es completo, no hay ninguna solución para la restricción, i.e. la restricción $c$ no es *E-unificable*.

- Supongamos que se devuelve la salida *única solución* $\theta$. Entonces el árbol entero de CNA ha sido explorado (exceptuando subárboles que no contenían ninguna solución) y se ha encontrado una solución $\theta$ más general que todas las demás soluciones posiblemente encontradas (si existen). Dado que CNA es correcto y completo, $\theta$ es el único *E-mgu* de $c$.

- Supongamos que se devuelve la salida *primera solución* $\theta$. Entonces un prefijo del árbol de búsqueda de CNA ha sido explorado (exceptuando subárboles que no contenían ninguna solución) y se ha encontrado una solución $\theta$. Dado que CNA es correcto, $\theta$ es un *E-unificador* de $c$ aunque, posiblemente, no es el único *E-mgu* de $c$. □

Se debe notar que cuando *N(c, f)* devuelve la salida *primera solución* $\theta$, $\theta$ podría incluso ser el único *E-mgu* de *c,* si éste existe. De hecho, el test de generalidad relativa entre soluciones que se realiza en las reglas (4) y (6) es sólo sintáctico mientras que dicha comparación debería de ser semántica. Esto significa que el algoritmo incremental de resolución de restricciones *ICS* posiblemente simplificará las restricciones menos de lo que sería posible. El algoritmo de "Narrowing" *C*ondicional *H*eurístico *N(c, f)* que acabamos de definir es un procedimiento para comprobar la solubilidad de las restricciones sobre la estructura *H/E*. Si la restricción es *H/E-soluble* entonces el procedimiento encuentra una solución (un *E-unificador*) para ella y busca otro *E-unificador* sintácticamente incomparable. Si la restricción no es *E-unificable* o si no hay una segunda solución incomparable, el procedimiento puede no terminar.

Incluso en el caso de teorías canónicas, el uso de "narrowing como procedimiento de unificación semántica presenta varios inconvenientes como la posible no terminación cuando no existen (más) soluciones o el no garantizar la generación de un conjunto minimal de *E-unificadores*. "Narrowing" puede derivar un conjunto infinito de *E-unificadores* incluso en el caso de que la ecuación admita un conjunto completo, minimal y finito [Hullot 80]. Este problema ya fue heredado por las distintas técnicas desarrolladas para resolución ecuacional, como SLDE-resolución, donde un simple paso de resolución puede entrar en bucle sin producir ninguna respuesta. En [Hullot 80] se da una caracterización suficiente para la terminación que sirve en el caso de teorías canónicas no condicionales.

Para superar estas dificultades se han desarrollado varias propuestas. En [Hölldobler 89] se introduce una regla de resolución perezosa que retrasa el problema de la *E-unificación* de un conjunto de ecuaciones, añadiéndolo como una restricción al objetivo derivado. Es decir, en vez de resolver el problema antes de que la regla pueda ser aplicada se añaden las ecuaciones como restricciones sin comprobar su satisfacibilidad. Se asume la existencia de una estrategia de selección justa (una estrategia con la cual todos los átomos y restricciones del objetivo serán seleccionados tarde o temprano) que decidirá el momento en que debe tener lugar la resolución de las restricciones (decidirá qué partes del problema de unificación se deben resolver antes de un paso de resolución y cuáles retrasar hasta tener más información disponible, como en [Furbach Hölldobler 86] y [Zachary 88]). Esto podría considerarse similar a la posposición del test de satisfacibilidad de las restricciones no lineales en CLP($\Re$) [Jaffar Michaylov 87] y también al método de "residuación" en [Aït-Kaci et al 87], donde se retrasa el problema de la *E-unificación* hasta que las ecuaciones son "ground". Sin embargo, en estos trabajos se incorporan explícitamente estrategias de control (asíncronas) para

*despertar,* en el momento de producirse las instanciaciones esperadas, las restricciones cuyo test de satisfacibilidad había sido aplazado. Estas estrategias son comparables a las técnicas de retraso que se describen en [Naish 86] (y se utilizan, e.g., en Prolog II). Pero, a diferencia de éstas, el retraso es implícito y transparente al programador y la posterior invocación se realiza mediante técnicas de propagación de restricciones, como las descritas en [van Hentenryck 89] (donde la propagación se produce cuando la restricción tiene el suficiente número de variables "ground" como para ser seleccionada, lo cual provoca posteriores pasos de propagación sobre otras restricciones). Otra posibilidad es retrasar completamente la resolución de las restricciones hasta haber resuelto el resto de objetivos y entonces pasar el problema a un algoritmo de *E-unificación* completo, como en [Bürckert 87]. En este caso banal, obviamente, no tendría lugar plantear las optimizaciones derivadas del discurso de la incrementalidad.

#### 3.3.1.3. Optimizaciones del algoritmo de "narrowing".

El alto grado de indeterminismo inherente al "narrowing" causa a menudo la generación de muchas soluciones redundantes. Es evidente que el procedimiento de "narrowing" descrito anteriormente debe optimizarse apropiadamente antes de poder ser implementado e incorporado dentro de un sistema CLP(*H/E*). En esta subsección se proponen dos optimizaciones útiles (que se derivan de las reglas introducidas por [Huet Hullot 82] en su modificación del algoritmo de Knuth Bendix y que fueron recogidas, e.g., en [Fribourg 85b, Hussmann 86, Josephson Dershowitz 86, Nutt et al 89]) para HCNC que reducen significativamente la talla del árbol de búsqueda.

i) La primera optimización para evitar algunos caminos de "narrowing" superfluos se basa en el concepto de *símbolo de función irreducible* (ver definición 3.1.20) para detectar la irresolubilidad de una ecuación: dos términos encabezados por símbolos de función irreducibles nunca pueden ser iguales; cuando van encabezados por el mismo símbolo irreducible son iguales sii sus argumentos son iguales.

   Más formalmente, si un subobjetivo en un *N-estado* ocurriendo en *HCNC* contiene una ecuación $f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_n)$ y ocurre que $f$ y $g$ son irreducibles entonces se pueden usar las siguientes reglas de simplificación:

   - si $f \neq g$ entonces el subobjetivo es irresoluble y puede eliminarse. (*regla de rechazo*)

   - si $f = g$ entonces la ecuación puede reemplazarse por el conjunto de

ecuaciones: $t_1 = t'_1, \ldots, t_n = t'_n$. (*regla de descomposición*)

Esta optimización puede incluso reducir un árbol de búsqueda infinito a uno finito [Hussmann 86, Nutt et al 89, Hölldobler 89]. Además puede programarse fácilmente utilizando, e.g., una estrategia de normalización que haga reducciones voraces respecto a las propias reglas de simplificación, incluídas como cláusulas del programa, como se describe en [Fribourg 85b]. Las reglas (consultar definición 3.1.20) pueden ser generadas automáticamente como en [Josephson Dershowitz 86] o estar implícitas en la definición del "narrowing", como en [Barbutti et al 86, Reddy 85].

ii) Si un subobjetivo en un *N-estado* ocurriendo en *HCNC* contiene una ecuación $X = t$ o $t = X$ donde $X \in V$, $X \notin Var(t)$ y todos los símbolos en $t$ son irreducibles, entonces la ecuación puede ser eliminada. La substitución $\{X/t\}$ debe de ser aplicada al subobjetivo entero. (*regla de substitución*)

El siguiente ejemplo ilustra cómo estas optimizaciones pueden reducir drásticamente la talla del árbol de búsqueda de "narrowing".

**Ejemplo 3.3.1.3.1.** Reconsideremos la especificación de la función + en el programa *Cats* del ejemplo 3.2.2 y supongamos haber planteado el objetivo $0 = succ( 0 + Y )$ *with* $\varepsilon$, que es irresoluble. En este ejemplo, *HCNC* explora un árbol de prueba infinito mientras que *HCNC* con la optimización i) explora un árbol con sólo un nodo (un sólo objetivo). Si el objetivo a resolver es $(Y = 0, 0 = succ(0 + Y))$ *with* $\varepsilon$ que es, igualmente, irresoluble, *HCNC* explora un árbol de prueba infinito mientras que *HCNC* con la optimización ii) (incluso sin la optimización i) explora un árbol con sólo dos nodos.

Para otras optimizaciones, como el uso de reescritura en combinación con "narrowing" o el uso de reglas de subsumción, consultar [Hölldobler 89, Nutt et al 89, Padawitz 88]. Se pueden considerar también las diferentes estrategias para reducir el número de subtérminos a los que se puede aplicar un paso de "narrowing" [Bosco et al 88, Echahed 88, Fribourg 85a, Giovannetti Moiso 86, Herold 86, Nutt et al 89].

### 3.3.2. Modelo operacional de CLP(*H/E*).

En esta subsección se describe el mecanismo de reducción de objetivos del lenguaje CLP(*H/E*), construido sobre la base del cálculo *ICS* descrito en la subsección §3.3.1.

### 3.3.2.1. El sistema de transición CLP(*H/E*).

Las siguientes definiciones instancian las definiciones 3.1.11, 3.1.12 y 3.1.13 al caso de programas CLP(*H/E*).

**Definición 3.3.2.1.1** *relación de transición* $\rightarrow_{CLP(H/E)}$

Sea $P \cup E$ un $(\Pi,\Sigma)$-programa CLP(*H/E*). La relación de transición $\rightarrow_{CLP(H/E)}$ se define mediante la siguiente regla que describe un paso de computación a partir de una $(\Pi,\Sigma)$-configuración $< \leftarrow s_i \Diamond A_1,\ldots, A_n >$, donde $s_i = < \theta_i, f_i, r_{c_i} >$:

(1) Regla de reducción de objetivos

$$\frac{s_i \xrightarrow{\tilde{c}}_{ICS} s_{i+1}}{< \leftarrow s_i \Diamond A_1, \ldots, A_n > \rightarrow_{CLP(H/E)} < \leftarrow s_{i+1} \Diamond \tilde{B}_n, \ldots, \tilde{B}_n >}$$

si existen $n$ variantes de cláusulas en $P$, $H_j \leftarrow c'_j \square \tilde{B}_j$., $j=1,\ldots,n$, sin variables en común con $\leftarrow c_i \square A_1,\ldots, A_n$. ni con el resto de cláusulas, y $\tilde{c} = \{c'_1,\ldots, c'_n, A_1 = H_1, \ldots, A_n = H_n\}$.

**Definición 3.3.2.1.2.** *configuración* CLP(*H/E*) *inicial* $(C_0)$

Sea $G_0 = \leftarrow c_0 \square \tilde{B}$ un $(\Pi,\Sigma)$-*objetivo*. Si $c_0$ no es una restricción vacía, $s_0$ denota el estado vacío del "constraint solver" y $s_0 \xrightarrow{c_0}_{ICS} s$ entonces $C_0 = < \leftarrow s \Diamond \tilde{B} >$ es la *configuración* CLP(*H/E*) *inicial*. Si $c_0$ es vacía entonces $C_0 = < \leftarrow s_0 \Diamond \tilde{B} >$.

**Definición 3.3.2.1.3.** *configuraciones* CLP(*H/E*) *terminales (C)*

Una configuración terminal $C$ tiene la forma $C = < \leftarrow s \Diamond >$, donde $s = < \theta, f, (\backslash O(\hat{}, \theta)_1, c_2) >$ es un *ICS-estado* y $c_{ans} = (\backslash O(\hat{}, \theta)_1 \cup c_2)_{|Var(G_0)} \cup_{Var(c_2)}$ es la *restricción respuesta computada* correspondiente a esa derivación.

Análogamente al caso genérico, cuando el "constraint solver" se diseña simplemente para verificar la *H/E-solubilidad* de las restricciones, la relación de transición $\rightarrow_{CLP(H/E)}$ definida anteriormente se transforma en la definición estándar de *derivación* en programación lógico-ecuacional, tal como se define en [Jaffar et al 84, Jaffar et al 86 a]:

$$\frac{\text{existe un } E\text{-unificador de } c \cup \tilde{c}}{< \leftarrow c \ \square \ A_1, \ldots, A_n. > \rightarrow_{CLP(H/E)} < \leftarrow c \cup \tilde{c} \ \square \ \tilde{B}_1, \ldots, \tilde{B}_n.>}$$

### 3.3.2.2. Ejemplo de computación

Para ilustrar el funcionamiento del mecanismo de reducción de objetivos del lenguaje CLP(*H/E*), en esta subsección se muestra un ejemplo de computación para un programa y objetivo inicial dados. El ejemplo clarifica los puntos anteriores y motiva el próximo apartado.

**Ejemplo 3.3.2.2.1.** *Problema de la mochila 0/1.*

Consideremos el siguiente programa CLP(*H/E*), basado en un ejemplo en [Hölldobler 87]. Dada una lista de items con unos pesos asociados y una mochila con capacidad limitada, el programa obtiene todas las sublistas de items que pueden incluirse en la mochila de forma óptima, i.e. agotando la capacidad de la misma. Por simplicidad abreviamos $succ^n(0)$ como *n*

```
%  SPEC Knapsack

%  CONSTRUCTORS
%       0              /   0
%       succ           /   1
%       [ ]            /   0
%       [ | ]          /   2
%       a              /   0
%       b              /   0
%       c              /   0

%  FUNCTIONS
%       +              /   2
%       weight         /   1
%       addweight      /   1

%  PREDICATES
%       sublist        /   2
%       knapsack       /   3

%  RELATIONAL CLAUSES
```

```
/* p1 */   knapsack( M, L, W)   :-  addweight( M) = W   □  sublist( M, L).
/* p2 */   sublist( [ ], Z).
/* p3 */   sublist( [X|Y], [X|Z] ) :-                  □  sublist( Y, Z).
/* p4 */   sublist( Y, [X|Z] )      :-                 □  sublist( Y, Z).

%  FUNCTIONAL CLAUSES
/* e1 */   X + 0                  →  X.
/* e2 */   X + succ( Y )          →  succ( X + Y ).
/* e3 */   weight( a)             →  1.
/* e4 */   weight( b)             →  1.
/* e5 */   weight( c)             →  2.
/* e6 */   addweight( [ ] )       →  0.
/* e7 */   addweight( [X|Y] )     →  weight( X ) + addweight( Y ).

%  ENDSPEC
```

Explicamos ahora brevemente la intención de los símbolos y reglas propuestos. *0*, *succ*, *[ ]* y *[ | ]* son símbolos irreducibles que representan, respectivamente, el cero y el sucesor de los números naturales y los constructores de listas. La función *weight( X )* asigna peso a algunos items. *addweight( L )* calcula el peso total de los items de la lista *L*. *sublist ( M, L )* se satisface si la lista *M* puede obtenerse de la lista *L* eliminando algunos items. *knapsack( M, L, W )* establece que los items en la sublista *M* de la lista *L* pueden introducirse en una mochila que pesa exactamente *W*.

Consideremos el objetivo inicial $G_0 = \leftarrow \square$ *knapsack( K, [a,b], 2)*. Dado que la restricción en $G_0$ es vacía, la configuración CLP(*H/E*) inicial es:

$$C_0 = < \leftarrow < \varepsilon, \varnothing, ( ) > \lozenge \; knapsack( K, [a,b], 2) >.$$

Considerando la cláusula *p1*, la nueva restricción $\tilde{c}$ es

$$\tilde{c} = \{addweight( M ) = W, K = M, L = [a, b], W = 2\}$$

y, dado que $\tilde{c}\varepsilon$ tiene una primera solución $\theta$= *{M/[a,a], K/[a,a], L/[a,b], W/2}*, aplicando la regla (4) de la definición del sistema de transición *ICS*,

$$C_0 \rightarrow_{CLP(H/E)} C_1 = < \leftarrow < \theta, \varnothing, (\varnothing, \tilde{c}) > \lozenge \; sublist( M, L) >.$$

Considerando la cláusula *p3*, la nueva restricción $\tilde{c}'$ es

$$\tilde{c}' = \{M = [X|Y], L = [X|Z]\}$$

y, dado que $\tilde{c}'\theta$ tiene un único *E-mgu* $\theta' = \{X/a, Y/[a], Z/[b]\}$ y que $\tilde{c} \cup \tilde{c}'$ tiene una primera solución *{M/[a,a], K/[a,a], L/[a,b], W/2, X/a, Y/[a], Z/[b]}*, aplicando la regla (3) de la definición del sistema de transición *ICS,*

$$C_1 \rightarrow_{CLP(H/E)} C_2 = < \leftarrow < \theta\theta', \varnothing, (\varnothing, \tilde{c} \cup \tilde{c}') > \Diamond \; sublist(Y, Z) >.$$

Considerando de nuevo la cláusula *p3*, la nueva restricción $\tilde{c}''$ es

$$\tilde{c}'' = \{Y = [X'|Y'], Z = [X'|Z']\}$$

y, dado que $\tilde{c}''\theta\theta'$ no tiene solución y que $\tilde{c} \cup \tilde{c}' \cup \tilde{c}''$ tiene un único *E-mgu* $\theta'' = \{M/[a,b], K/[a,b], L/[a,b], W/2, X/a, Y/[b], Z/[b], X'/b, Y'/[\,], Z'/[\,]\}$, aplicando la regla ($5^i$) de la definición del sistema de transición *ICS,*

$$C_2 \rightarrow_{CLP(H/E)} C_3 = < \leftarrow < \theta'', \theta\theta', (\backslash O(\hat{},\theta)'', \varnothing) > \Diamond \; sublist(Y', Z') >.$$

Considerando la cláusula *p2*, la nueva restricción $\tilde{c}'''$ es

$$\tilde{c}''' = \{Y' = [\,], Z' = Z''\}$$

y, dado que $\tilde{c}'''\theta''$ tiene un único *E-mgu* $\theta''' = \{Z''/[\,]\}$, aplicando la regla (1) de la definición del sistema de transición *ICS,*

$$C_3 \rightarrow_{CLP(H/E)} C_4 = < \leftarrow < \theta''\theta''', \theta\theta', (\backslash O(\hat{}, \theta''\theta'''), \varnothing) > \Diamond \; >.$$

Finalmente, dado que se ha alcanzado una configuración CLP(*H/E*) terminal, la *restricción respuesta* correspondiente a esta derivación es

$$c_{ans} = (\backslash O(\hat{}, \theta''\theta''') \cup \varnothing)_{|Var(G_0)} \cup_{Var(\varnothing)} = \{K = [a,b]\}.$$

El ejemplo anterior pone de manifiesto el hecho de que la restricción acumulada se simplifica cuando se aplican las reglas (1), (2) y ($5^i$) del cálculo *ICS* y cómo se reutiliza la solución encontrada en el paso previo para intentar demostrar que la nueva restricción acumulada es satisfacible. De esta forma, la comprobación de su satisfacibilidad es menos costosa que el correspondiente test de satisfacibilidad para una restricción acumulada sin simplificar.

La siguiente sección propone varias estrategias de optimización del modelo operacional de CLP(*H/E*) basadas, respectivamente, en los resultados acerca del

"narrowing" *selección* presentados en [Bosco et al 88, Giovannetti Moiso 86] y en la introducción de una regla de propagación de las soluciones similar a la que se define en [Jaffar Lassez 86]. Ambos refinamientos preservan la corrección y completitud de la semántica operacional estándar del esquema CLP.

### 3.3.3. Optimizaciones del modelo operacional.

En la subsección §3.3.1.3 hemos presentado varias optimizaciones aplicables al algoritmo de "narrowing" heurístico que está en la base del "constraint solver" incremental descrito en §3.3.1.1. En este apartado presentaremos un sistema de inferencia más eficiente para ambos así como una regla de propagación que optimiza el cálculo que define el nivel superior del intérprete CLP(*H/E*).

#### 3.3.3.1. La regla de propagación.

El modelo operacional del lenguaje CLP(*H/E*) se ha definido como un sistema lógico que se estructura de forma jerárquica. El nivel superior describe abstractamente el intérprete CLP(*H/E*) (definición 3.3.2.1.1, relación de transición $\rightarrow_{CLP(H/E)}$) y está soportado por un procedimiento incremental (ICS) que comprueba la satisfacibilidad de las restricciones a la vez que las simplifica. A continuación extendemos el cálculo que define la relación de transición $\rightarrow_{CLP(H/E)}$ introduciendo la siguiente regla de inferencia adicional:

(2) Regla de propagación

$$< \leftarrow < \theta, f, (\backslash O(\hat{},\theta)_1, c_2) > \Diamond \tilde{B} > \rightarrow_{CLP(H/E)} < \leftarrow < \theta, f, (\backslash O(\hat{},\theta)_1, c_2) > \Diamond \tilde{B}\theta_1 >$$

Esta regla propaga la solución única a una restrición, resuelta por el procedimiento incremental ICS, al nivel superior (nivel CLP) para guiar las derivaciones en éste. La aplicación de esta regla se supone sujeta a un cierto control, que define el siguiente funcionamiento del intérprete optimizado: inmediatamente antes de la reducción de un objetivo utilizando la primera regla del cálculo $\rightarrow_{CLP(H/E)}$ (regla de reducción de objetivos) se aplica la regla de propagación. La corrección de esta estrategia se deduce como particularización inmediata de los resultados presentados en [Jaffar Lassez 86] sobre la relación (en nuestro caso, uno-uno) entre $(P,\mathfrak{I})$ y $(P,\mathfrak{R})$-*derivaciones correspondientes,* que revisamos a continuación brevemente. Se define la relación de $(P,\mathfrak{I})$-derivación sobre la teoría $\mathfrak{I}$ correspondiente a una estructura *compacta respecto a las soluciones* $\mathfrak{R}$ de la siguiente forma

$$\frac{c'_1\theta \text{ es } \mathfrak{I}\text{–satisfacible}^{20}}{< \leftarrow c \ \square \ A_1, \ldots, A_n. > \rightarrow_{CLP(\chi)} < \leftarrow c'_1\theta \ \square \ \tilde{B}_1\theta, \ldots, \tilde{B}_n\theta.>}$$

si existen $n$ variantes de cláusulas en $P$, $H_j \leftarrow c_j \ \square \ \tilde{B}_j.$, $j=1, \ldots, n,$ sin variables en común con $\leftarrow c \ \square \ A_1,\ldots, A_n.$ ni con el resto de cláusulas, y $\tilde{c} = \{c_1,\ldots, c_n, A_1 = H_1, \ldots, A_n = H_n\}$ y

(a) la restricción $\tilde{c}$ puede escribirse de la forma $c'_1 \cup c'_2$ (pudiendo ser cualquiera de ellas vacía) donde $c'_2$ tiene un conjunto completo y finito $S$ de satisfactores de máxima generalidad

(b) $\theta \in S$ (cuando $S$ es vacío, tomamos la substitución identidad).

De acuerdo con [Jaffar Lassez 86], la restricción respuesta asociada a una $(P,\mathfrak{I})$-derivación con éxito es $c_{ans} \cup$O($\hat{}$, $\theta_1$) $\cup$O($\hat{}$, $\theta_2$) $\cup \ldots \cup$O($\hat{}$, $\theta_m$) siendo $c_{ans}$ la restricción en el último objetivo de la derivación y O($\hat{}$, $\theta_i$, ), $i = 1, 2, \ldots, m,$ el $\mathfrak{I}$–satisfactor utilizado en el *i-ésimo* paso de la misma.

Para teorías $\mathfrak{I}$ en las que cualquier restricción tiene un conjunto completo y finito de $\mathfrak{I}$–satisfactores de máxima generalidad, todas las $(P,\mathfrak{I})$-derivaciones a partir de un objetivo $G$ dado pueden efectuarse de forma que todos los objetivos en la derivación, excepto $G$, contienen sólo restricciones vacías. El otro extremo se obtiene no utilizando nunca satisfactores en las $(P,\mathfrak{I})$-derivaciones. Entre ambos extremos se encuentra la posibilidad explotada por nuestra regla de propagación. En ella, siendo $\mathfrak{I}$ la teoría $E$ y conocido que una parte de la restricción acumulada tiene un conjunto completo y finito de *E-unificadores* de máxima generalidad (que, en nuestro caso particular está formado por un único elemento, un único *E-mgu*), éste se propaga a los átomos en el objetivo deducido a la vez que dicha solución se mantiene, en forma ecuacional, como parte de la restricción acumulada. En [Jaffar Lassez 86] se demuestra que un objetivo tiene una $(P,\leftarrow)$-derivación con éxito sii existe para el mismo una $(P,\mathfrak{I})$-derivación con éxito. De este resultado se deduce que, en caso de que el conjunto $S$ tenga un único elemento, ambas derivaciones deben producir restricciones respuesta equivalentes. Estos resultados se aplican, por tanto, a nuestro cálculo.

---

20 Una restricción $c$ es $\mathfrak{I}$*–satisfacible* si $\mathfrak{I} \models \tilde{\exists}\, c$. Una substitución $\theta$ tal que *(*$\forall$ *substitución* $\gamma$*)* $\mathfrak{I} \models c\theta\gamma$ se llama $\mathfrak{I}$*–satisfactor* de $c$.

Se debe notar que en nuestra formulación del cálculo ICS, en la cual se utiliza la substitución $\theta_1$ para instanciar las restricciones lanzadas al procedimiento de "narrowing" (reglas (2), (3), ($5^i$) y ($5^{ii}$) del cálculo), se incorpora, implícitamente, esta optimización. El siguiente ejemplo ilustra la mejora obtenida respecto al caso en que, en dichas reglas, se hubiera escrito $N(\backslash O(\hat{},\theta_1) \cup c_2 \cup \tilde{c})$ en el lugar de $N(c_2 \cup \tilde{c})\theta_1$.

**Ejemplo 3.3.3.1.** Consideremos el siguiente programa:

$P$ = $\{ p(b). \}$

$E$ = $\{ g(f(a)) = a.$
$f(a) = a.$
$f(f(X)) = f(X). \}$

junto con el objetivo inicial $G_0 = \leftarrow g(Y) = a \ \square \ p(f(Y))$.

Dado que la restricción $c_0$ en el objetivo inicial ($c_0 = \{g(Y) = a\}$) es no vacía y que, por tener solución única, el estado *vacío* del "constraint solver" $s_0 \overset{\tilde{c}_0}{\longrightarrow}_{I)CS} s$, donde $s_0 = < \varepsilon, \varnothing, ( ) >$ y $s = < Y / f(a), \varnothing, (Y = f(a), \varnothing) >$, la configuración CLP(*H/E*) inicial viene dada por:

$C_0 = < \leftarrow < \{Y / f(a)\}, \varnothing, (Y = f(a), \varnothing ) > \Diamond \ p(f(Y)) >$.

Considerando la única cláusula en $P$, la nueva restricción $\tilde{c}$ es

$\tilde{c} = \{f(Y) = b\}$.

Dado que $\tilde{c}\{Y / f(a)\}$ no tiene solución, se intenta la aplicación de la regla (5) del cálculo ICS que, con el cambio antes mencionado, entra en bucle en la evaluación de la condición, dado que el "narrowing" de la restricción $\backslash O(\hat{}, \theta_1) \cup c_2 \cup \tilde{c} = \{Y = f(a), f(Y) = b\}$ no termina.

### 3.3.3.2. Optimización del "constraint solver".

En este apartado presentamos una optimización del procedimiento que determina la solubilidad de las restricciones en la estructura *H/E*. Por simplicidad (y por completitud), presentaremos esta optimización aplicada al "constraint solver" más simple, aquel que verifica la solubilidad de forma no incremental. A continuación definimos,

sobre la base de este cálculo[21], un "constraint solver" incremental completo (que llamaremos iCS) para programas CLP(*H/E*).

El núcleo de iCS es un cálculo de "narrowing" condicional "innermost", *iCNC*, que explora (un prefijo de) un árbol de búsqueda que se construye de acuerdo con la estrategia de "narrowing" selección "innermost" definida en [Bosco et al 87, Bosco et al 88]. El árbol, además, se construye de forma incremental a medida que se añaden nuevas restricciones.

A continuación se define el cálculo *iCNC*, que busca un *E-unificador* $\theta$ para la restricción de entrada $c$ y devuelve como salida la lista $L$ de subobjetivos que quedan aún por reducir.

**Definición 3.3.3.2.1.** *C*álculo de "*N*arrowing" *C*ondicional "innermost" (*iCNC*)

Un *iCNC-estado* es un par $< n, L >$, donde $n \in \{0, 1\}$ y $L$ es una lista (posiblemente vacía) $[g_i]_{i=1}^{n}$ de subobjetivos $g_i = c_i \ with \ (\tau_i, Occ_i)$ donde $c_i$ es un conjunto de $(\Pi_C,\Sigma)$-*restricciones* (una conjunción de ecuaciones), $\tau_i$ es una $(\Pi,\Sigma)$-*substitución* y $Occ_i$ es el conjunto de ocurrencias de $c_i$ a las que puede aplicarse un paso de "narrowing" (ocurrencias que llamaremos *reducibles*).

El cálculo *iCNC* se define por medio de la siguiente relación de transición[22]:

Regla de ramificación "Don't-care"

(1)

$$\frac{u_0 = select\text{-}don't\text{-}care(Occ) \wedge \quad [< u_0, k_i, \sigma_i >]_{i=1}^{n} = s\_to\_l(\{< u_0, k, \sigma >: narred(c, u_0, k, \sigma)\})}{c \ with \ (\tau, Occ) \rightarrow_{DcB} c \ with \ (\tau, (Occ \sim \{u_0\})) \bullet [\ (c[u_0 \leftarrow r_{k_i}], \tilde{e}_{k_i})\sigma_i \ with \ (\tau\sigma_i, ((Occ \sim \{u_0\}) \cup \{u_0.v: v \in O'(r_{k_i})\} \cup O(c, \tilde{e}_{k_i}))) \ ]_{i=1}^{n}}$$

Regla de unificación

(2)

$$\frac{c \ unifica \ sintácticamente \ con \ mgu \ \sigma}{< 0, c \ with \ (\tau, \varnothing) \bullet L > \rightarrow_{iCNC} < 1, \varnothing \ with \ (\tau\sigma, \varnothing) \bullet L >}$$

[21] Las técnicas que vamos a describir pueden ser adaptadas fácilmente a la optimización del cálculo ICS pero, claramente, el "constraint solver" resultante no sería, en general, completo dado que *ICS* no lo es.

[22] Por simplicidad, en este conjunto de reglas, *Occ* denota un conjunto no vacío.

Regla de narrowing

(3)

$$\frac{c \text{ with } (\tau, Occ) \rightarrow_{DcB} L'}{< 0, c \text{ with } (\tau, Occ) \bullet L > \rightarrow_{iCNC} < 0, L \bullet L' >}$$

(4)

$$\frac{mgu(c) = \varnothing}{< 0, c \text{ with } (\tau, \varnothing) \bullet L > \rightarrow_{iCNC} < 0, L >}$$

donde la función *select-don´t-care(Occ)* selecciona de forma no determinista una ocurrencia "innermost" (i.e. una ocurrencia que no es prefijo de ninguna otra [Bosco et al 88, Fribourg 85b]) en *Occ*. De esta forma, en cada paso sólo se selecciona un subtérmino. Cualquier otra elección de subtérmino para el mismo paso no será tenida en cuenta en la búsqueda. La función $O(c, \tilde{c})$ en la regla (1) se define de la siguiente forma:

$$O(c, \tilde{c}) = \{ (|c| + j).k: \ j.k \in O'(\tilde{c}) \}.$$

donde $|c|$ denota la cardinalidad del conjunto de restricciones *c*.

**Definición 3.3.3.2.2.** *(comportamiento de iCNC)*

Definimos la función:

$$iN(L) = L' \ si \quad <0, L> \rightarrow^*_{iCNC} \quad <1, L'>$$

A continuación introducimos el concepto de árbol de búsqueda cuyos nodos se expanden de acuerdo a una estrategia de "narrowing" selección "innermost".

**Definición 3.3.3.2.3.** *(árbol de búsqueda "innermost")*

Un *árbol de búsqueda "innermost"* para una restricción *c* es un árbol cuyos nodos vienen etiquetados por subobjetivos $c_i$ *with* $(\tau_i, Occ_i)$, donde el subobjetivo en la raiz es *c with* $(\varepsilon, O'(c))$. Sea $g = c \text{ with } (\tau, Occ)$ un nodo del árbol ($Occ \neq \varnothing$). Sea $u_0 =$ *select-don´t-care(Occ)* y asumamos que $s\_to\_l (\{< u_0, k, \sigma >: narred(c, u_0, k, \sigma)\}) = < u_0, k_i, \sigma_i >]_{i=1}^{n}$. Entonces $g$ tiene un hijo $c \text{ with } (\tau, (Occ \sim \{u_0\}))$ y $n$ hijos $(c[u_0 \leftarrow r_{k_i}], e_{k_i})\sigma_i$ *with* $(\tau\sigma_i, ((Occ \sim \{u_0\}) \cup \{u_0.v: v \in O'(r_{k_i})\} \cup O(c, e_{k_i}))), i = 1, ..., n$.

**Definición 3.3.3.2.4.** *(representación de un árbol de búsqueda)*

Sea $t$ un árbol de búsqueda "innermost" para la restricción $c$. La lista $L_c$ de las hojas de $t$ será llamada *representación de t*. $L_c$ representa un estado de ejecución [Bosco et al 88] del algoritmo de "narrowing" selección "innermost" condicional.

**Definición 3.3.3.2.5.** *(iCS-estado)*

Un ICS-*estado* es un par $< c, L_c >$, donde $c$ es una restricción y $L_c$ es una lista de subobjetivos. El ICS-*estado vacío* es $< \varnothing, [\ ] >$.

Informalmente, en la definición anterior, $L_c$ es la representación en forma de lista de un árbol de búsqueda "innermost" para la restricción $c$.

**Lema 3.3.3.2.6.** Sea $L_c$ una lista de subobjetivos que representa un árbol de búsqueda "innermost" para la restricción $c$. Las siguientes afirmaciones son ciertas.

- $iN(L_c) = L' = \varnothing$ *with* $(\theta, \varnothing) \bullet L''$ sii existe un estado terminal $< 1, L' >$ tal que $\theta$ es una solución a $c$ y la lista $L'$ representa el estado de ejecución de la extensión a teorías condicionales del algoritmo de "narrowing" selección "innermost" de [Bosco Giovannetti 88] cuando la solución $\theta$ es encontrada.

- La visita a $L_c$ simula una estrategia de selección "innermost".

**Demostración.**

Demostraremos (simultáneamente) las dos afirmaciones por inducción sobre la longitud $n$ de la computación. Necesitamos tener en cuenta el siguiente hecho:

- Hecho(1): La regla (1) establece que si puede probarse la transición $c$ *with* $(\tau, Occ) \rightarrow_{DcB} L' = c$ *with* $(\tau, (Occ \sim \{u_0\})) \bullet [\ (c[u_0 \leftarrow r_{k_i}], e_{k_i})\sigma_i$ *with* $(\tau\sigma_i, ((Occ \sim \{u_0\}) \cup \{u_0.v: v \in O'(r_{k_i})\} \cup O(c, e_{k_i})))\ ]_{i=1}^{n}$ entonces se tiene lo siguiente. $Occ$ es el conjunto de ocurrencias en $c$ que pueden ser reducidas. Una de ellas ($u_0$) es elegida de forma no determinista por la función de selección *select-don´t-care(Occ)*. La lista $L'$ de subobjetivos corresponde a todos los posibles pasos de "narrowing" a partir de $c$ en la ocurrencia $u_0$ y $\tau\sigma_i$ es la composición de todas las sustituciones aplicadas a lo largo del *i-ésimo* camino de "narrowing".

Para demostrar el lema 3.3.3.2.6 necesitamos también probar, por inducción, que la substitución en la parte *with* de un subobjetivo corresponde, además, a la composición de todas las substituciones aplicadas a lo largo de un camino en un árbol de búsqueda "innermost".

Por inducción sobre la longitud $n$ de una computación:

- *n = 1)* Si se ha aplicado la regla (2), entonces $\tau = \varepsilon$ y $\sigma$ es un unificador sintáctico de la restricción $c$ siendo, por tanto, una solución a la misma.

  Si se ha aplicado la regla (3) y se ha probado la transición $< 0, c \text{ with } (\tau, Occ) \bullet L > \rightarrow_{iCNC} < 0, L \bullet L' >$ entonces, por la premisa de la regla y el hecho (1), $L'$ es la lista de subobjetivos que corresponde a la ramificación *don't-care* a partir de la restricción $c$ en alguna ocurrencia "innermost" $u_0$ usando todas las reglas de reescritura aplicables. Se debe notar que se expande el subobjetivo "leftmost" $c \text{ with } (\tau, Occ)$ de la lista correspondiente al tercer argumento del estado de partida y que su expansión, $L'$, se añade al final de la nueva lista $L \bullet L'$. Esto garantiza una visita "breadth-first" del árbol representado por la lista de partida. El hecho (1) garantiza también que los subobjetivos en la lista $L'$ contienen, en las partes *with*, la composición de las sustituciones $\tau\sigma_i$ aplicadas a lo largo de un camino de "narrowing".

- *n > 1)* La prueba del caso inductivo es perfectamente análoga a la del caso base:

  Si se ha aplicado la regla (2) entonces, por la hipótesis inductiva, sabemos que $\tau$ es la composición de las sustituciones a lo largo de un camino de "narrowing" "innermost" que parte de la restricción inicial y que $c$ es la próxima restricción a reducir. Dado que $\sigma$ es un unificador sintáctico de $c$, $\tau\sigma$ es una solución a la restricción original.

  Si se ha aplicado la regla (3) entonces, por la hipótesis inductiva, sabemos que $\tau$ es la composición de las sustituciones a lo largo de un camino de "narrowing" "innermost" que parte de la restricción inicial y, por el hecho (1), tenemos que la lista $L'$ consta de los subobjetivos cuya parte *with* contiene la composición de las sustituciones $\tau\sigma_i$, i.e. todas las sustituciones aplicadas a lo largo de ese camino de "narrowing".

- Si se ha aplicado la regla (4) y se ha probado la transición $< 0, c \text{ with } (\tau, \varnothing) \bullet L > \rightarrow_{iCNC} < 0, L >$ entonces se ha visitado el primer nodo $c \text{ with } (\tau, \varnothing)$ y, dado que la restricción no es unificable y que no tiene ninguna ocurrencia reducible, se ha de eliminar el nodo sin generar ninguna ramificación. □

**Definición 3.3.3.2.7.** *(iCS-relación de transición* $\overset{\tilde{c}}{\longrightarrow}_{iCS}$*)*

$$\frac{L' = iN(merge(L, \tilde{c}))}{< c, L > \overset{\tilde{c}}{\longrightarrow}_{iCS} < c \cup \tilde{c}, L' >}$$

Definimos a continuación la función $merge(L, \tilde{c})$, que combina la representación en forma de lista $L$ de un árbol de búsqueda "innermost" para la restricción $c$ con la nueva restricción $\tilde{c}$ para construir la representación de un árbol de búsqueda "innermost" para $c \cup \tilde{c}$.

$merge([\,], \tilde{c}) = [\tilde{c} \text{ with } (\varepsilon, O'(\tilde{c})]$.

$merge([c_i \text{ with } (\tau_i, Occ_i)]_{i=1}^{n}, \tilde{c}) =$
$[(c_i, \tilde{c}\tau_i) \text{ with } (\tau_i, Occ_i \cup O(c_i, \tilde{c}\tau_i))]_{i=1}^{n}$.

A continuación demostramos la corrección y la completitud del cálculo iCS utilizando los resultados de completitud del "narrowing" selección "innermost" condicional (conjeturados en [Giovannetti Moiso 86] y probados en [Middeldorp 91b])

**Lema 3.3.3.2.8.**

Sea $L_c$ la representación de un árbol de búsqueda "innermost" para la restricción $c$ denotando el estado de ejecución del algoritmo de "narrowing" condicional "innermost" cuando es encontrada la primera solución para $c$. Entonces, $merge(L_c, \tilde{c})$ representa un árbol de búsqueda "innermost" para la restricción $c \cup \tilde{c}$ correspondiente a ese mismo estado.

**Demostración.**

- Si $c$ es vacía, entonces $L_c = [\,]$. Dado que $merge([\,], \tilde{c}) = [\tilde{c} \text{ with } (\varepsilon, O'(\tilde{c})]$ y que, de acuerdo con la definición 3.3.3.2.3, $[\tilde{c} \text{ with } (\varepsilon, O'(\tilde{c})]$ representa un árbol de

búsqueda "innermost" para $\tilde{c}$, se sigue el resultado enunciado. De hecho, *merge([ ],* $\tilde{c}$*)* representa un árbol de búsqueda "innermost" para la restricción $c \cup \tilde{c}$ cuando no se ha realizado ningún paso de "narrowing".

• Si $c$ no es vacía, tenemos lo siguiente. Cualquier ocurrencia reducible e "innermost" de la restricción $c$ es, asimismo, una ocurrencia reducible e "innermost" de $c \cup \tilde{c}$. El hecho de que $L_c$ representa un árbol de búsqueda "innermost" para $c$ correspondiente al estado de ejecución del algoritmo de "narrowing" selección "innermost" cuando es encontrada la primera solución para $c$ garantiza entonces que *merge(*$L_c$, $\tilde{c}$*)* representa un árbol de búsqueda "innermost" para la restricción $c \cup \tilde{c}$ correspondiente al mismo estado. □

**Proposición 3.3.3.2.9.** [Middeldorp 91b]

Sea $E$ una teoría ecuacional condicional *confluente por niveles* y *noetheriana* (ver definición 3.1.15). Entonces el algoritmo de "narrowing" selección "innermost" (condicional) es un procedimiento de *E-unificación* completo. □

**Teorema 3.3.3.2.10.** *(Corrección y completitud de iCS)*

La transición $< c, L_c > \overset{c}{\longrightarrow}_{iCS} < c \cup \tilde{c}, L' >$ puede ser probada sii la restricción $c \cup \tilde{c}$ es soluble en *H/E*. La lista $L'$ en el estado de llegada representa un árbol de búsqueda "innermost" para $c \cup \tilde{c}$.

**Demostración.**

Sea $< c, L_c > \overset{c}{\longrightarrow}_{iCS} < c \cup \tilde{c}, L' >$ una transición iCS. Por la definición 3.3.3.2.7, $L'$ = *merge(*$L_c$, $\tilde{c}$*)*. Entonces, por el lema 3.3.3.2.8, *merge(*$L_c$, $\tilde{c}$*)* construye la representación $L_c \cup \tilde{c}$ de un árbol de búsqueda "innermost" para $c \cup \tilde{c}$. Por el lema 3.3.3.2.6, el sistema de transición *iCNC* emula una visita a dicho árbol e $\mathrm{iN}(L_c \cup \tilde{c}) = L'$ sii $L'$ representa un árbol de búsqueda "innermost" para $c \cup \tilde{c}$ cuando se encuentra una solución $\theta$ para la misma. Finalmente, aplicando la proposición 3.3.3.2.9, se obtiene el resultado esperado. □

# 4 Semántica declarativa de CLP(*H/E*)

Una de las principales motivaciones para formalizar un lenguaje de programación como una instancia del esquema CLP es que muchas de las propiedades semánticas fundamentales que se cumplen para el mismo pueden ser heredadas de forma automática por el lenguaje. En este capítulo se define la semántica declarativa de CLP(*H*/*E*) y se adaptan al lenguaje aquellos resultados que fueron demostrados para el esquema y que se aplican también a esta instancia. La definición semántica declarativa de CLP(*H*/*E*) se presentará en el estilo estándar: *teoría de modelos* y *punto fijo*.

Este capítulo se organiza en varias secciones. La primera sección introduce nuestra aproximación a la semántica declarativa del lenguaje CLP(*H*/*E*). Nuestra intención es definir modelos capaces de capturar completamente el comportamiento operacional de los programas y de expresar, por tanto, diferentes propiedades observables de los mismos, como el *conjunto de éxitos* "*ground*", *consecuencias atómicas* y *restricciones respuesta computadas*. La sección segunda describe la semántica por punto fijo del lenguaje, que proporciona un primer enlace entre la semántica operacional y el significado declarativo de los programas. Dicha semántica será relacionada con las varias propiedades (operacionales) observables de los programas CLP(*H*/*E*). En la sección tercera se caracterizan tales programas desde el punto de vista de la teoría de modelos y se presentan los esperados resultados de correción y completitud que establecen la equivalencia entre las tres semánticas: operacional, teoría de modelos y punto fijo.

## 4.1. Propiedades observables de los programas lógicos.

Una de las propiedades más atractivas de la programación lógica tradicional (y que se generaliza en el contexto de CLP) es la correspondencia entre la semántica declarativa (teoría de modelos y punto fijo) y la semántica operacional, mostrada por van Emden y Kowalski en [van Emden Kowalski 76]. En el caso de átomos "ground", esta correspondencia es completa, es una relación de equivalencia. De hecho, el modelo mínimo de Herbrand $M_P$ de un programa *P,* que caracteriza el conjunto de sus consecuencias atómicas "ground", es igual a su conjunto de éxitos. Sin embargo, la noción estándar de conjunto de éxitos en programación lógica

$$SS(P)= \{A: A \text{ es un átomo ground } y \leftarrow A \text{ tiene una refutación}\}$$

no es del todo adecuada como semántica operacional ya que oculta uno de los aspectos fundamentales de un programa lógico: la capacidad de computar respuestas [Falaschi et al 88, Falaschi et al 89].

Considérense, e.g., los programas $P_1 = \{p(X)., q(a).\}$ y $P_2 = \{p(a)., q(a).\}$. Ambos programas tienen las mismas interpretaciones y modelos de Herbrand. Sin embargo, desde un punto de vista operacional son diferentes [Falaschi et al 91]. De hecho, el objetivo $\leftarrow p(X).$ tiene una refutación con respuesta $\{X/a\}$ respecto a $P_2$ mientras la substitución de respuesta computada respecto a $P_1$ es vacía. Por otra parte, si se extiende el alfabeto del programa para contener la constante *b*, los programas dejan de tener el mismo conjunto de éxitos (el mismo modelo mínimo), puesto que el objetivo $\leftarrow p(b)$ es refutable en $P_1$ mientras que falla finitamente en $P_2$. Una definición más apropiada de conjunto de éxitos debería permitir caracterizar no sólo el conjunto de consecuencias atómicas "ground" de un programa sino también la diferencia, en cuanto a comportamiento operacional, de programas como $P_1$ y $P_2$ (caracterizando, e.g., el conjunto de consecuencias lógicas atómicas o el conjunto de átomos refutables, junto con sus respuestas computadas). Una definición más apropiada sería [Falaschi et al 89]:

$$SS'(P)= \{(A, \upsilon): A \text{ es un átomo } y \leftarrow A \text{ tiene una refutación con substitución de respuesta computada } \upsilon\}.$$

Desafortunadamente, en este caso ya no hay correspondencia entre $M_P$ y $SS'(P)$ debido al hecho de que el modelo mínimo no representa la noción de consecuencia lógica en el caso de átomos no "ground". En [Falaschi et al 88, Falaschi et al 89, Falaschi et al 91] se define una semántica por teoría de modelos que caracteriza exactamente el nuevo conjunto. La idea básica está en permitir que aparezcan variables (cuantificadas universalmente) en el universo de Herbrand del programa.

En [Gabbrielli Levi 91a] esta aproximación se extiende de forma natural al caso CLP, generalizando así la semántica algebraica de CLP propuesta en [Jaffar Lassez 86, Jaffar Lassez 87]. [Gabbrielli Levi 91a] introduce un marco formal para definir y comparar varias nociones de modelos, cada una correspondiente a diferentes propiedades observables de los programas CLP($\chi$). Dichos modelos permiten caracterizar completamente el comportamiento operacional del programa desde un punto de vista declarativo y permiten, por tanto, definir diferentes nociones de equivalencia entre programas (ver, al respecto, [Maher 88]). La construcción se basa en una nueva noción de interpretación (conjunto de *átomos restringidos* $c \square p(X_1, X_2, ..., X_n)$ que representan,

implícitamente, un conjunto (posiblemente infinito) de instancias "ground" $p(d_1, d_2, ..., d_n)$ sobre el dominio correspondiente), en una extensión natural del concepto estándar de verdad (una fórmula es verdad en una interpretación extendida $I$ si es verdad en la interpretación $[I]$ que se obtiene tomando las instancias de los átomos de $I$) y en la definición de varios operadores de consecuencia inmediata $T_i$, $i = 1, 2, 3$, cuyos menores puntos fijos sobre el retículo de las nuevas interpretaciones son modelos que corresponden a las diferentes propiedades observables. En particular, existe un operador de consecuencia inmediata, que se define en términos de restricciones resolubles, cuyo menor punto fijo es un modelo no minimal correspondiente a una semántica de *restricción respuesta computada*. Dentro de este marco se obtiene la total equivalencia entre semántica operacional y semántica declarativa.

En este capítulo se describe, en dicho marco, la semántica de los programas CLP(*H/E*). Introduciremos tres semánticas operacionales $SS_i(P, H/E)$, $i = 1, 2, 3$. Cada una de ellas define una noción de "observable" e induce, por tanto, una relación de equivalencia $\approx_i$ entre programas: $P_1 \cup E_1 \approx_i P_2 \cup E_2$ sii $SS_i(P_1, H_1/E_1) = SS_i(P_2, H_2/E_2)$. Quedará de manifiesto que las dos primeras relaciones $\approx_1$ e $\approx_2$ son demasiado débiles para caracterizar el comportamiento de los programas en términos de restricciones respuesta. Las diferentes propiedades (operacionales) observables de los programas (conjunto de éxitos "ground", consecuencias atómicas y restricciones respuesta computadas) tendrán su contrapartida en la semántica declarativa del lenguaje que será capaz, por tanto, de capturar la diferencia entre respuestas que son computadas efectivamente y aquellas que no lo son. Mostraremos que las distintas propiedades corresponden a diferentes modelos, incluyendo la semántica operacional estándar, que coincidirá aún con el modelo mínimo del programa.

En primer lugar presentamos una caracterización por punto fijo de los programas CLP(*H/E*), siguiendo la aproximación que se describe en [Gabbrielli Levi 91a, Gabbrielli Levi 91b].

## 4.2. Semántica por punto fijo.

En programación lógica tradicional, el método del punto fijo asocia al programa una aplicación continua sobre el retículo de las interpretaciones de Herbrand, aplicación que puede verse como un operador de inferencia de consecuencias atómicas "ground". Se demuestra además que el conjunto de éxitos "ground" del programa coincide exactamente con el menor punto fijo de dicha aplicación. Esta técnica resulta de mucha

utilidad para justificar la corrección y completitud de los procedimientos computacionales utilizados por el intérprete.

A continuación revisamos brevemente los principales conceptos y resultados de la aproximación de punto fijo [Apt 90, Apt van Emden 82, van Emden Kowalski 76, Lloyd 87].

### 4.2.1. Principales resultados de la aproximación de punto fijo.

Un *retículo completo* es un conjunto $U$ sobre el que se define una relación de orden $\leq$ reflexiva, antisimétrica y transitiva. Para todo subconjunto $X$ de $U$ existe en $U$ una menor cota superior $lub(X)$ y una mayor cota inferior $glb(X)$ con respecto a la relación $\leq$.

Una aplicación $T$ sobre un retículo completo $U$ es monótona si $X \leq Y$ implica $T(X) \leq T(Y)$ para todo $X, Y \in U$.

Un conjunto $X$ es dirigido si contiene una cota superior para cada uno de sus subconjuntos finitos.

Una aplicación $T$ sobre un retículo completo $U$ es continua si para cada subconjunto dirigido $X$ de $U$ ocurre que $T(lub(X)) = lub(\{T(Y): Y \in X\})$. Cualquier aplicación continua es monótona. El inverso no es cierto.

El teorema de Knaster-Tarski establece que toda función monótona $T$ definida sobre un retículo completo $U$ tiene un menor punto fijo $lfp(T) = glb\{X: T(X) = X\} = glb\{X: T(X) \leq X\}$. El menor punto fijo es computable cuando el operador $T$ lo es. El método de cálculo del mínimo punto fijo como el límite de las potencias ordinales de $T$ da un procedimiento efectivo para decidir si un elemento pertenece a él. Las potencias ordinales de $T$ se definen de la siguiente forma:

$T \uparrow 0 = glb(U)$.
$T \uparrow \alpha = T(T \uparrow (\alpha - 1))$ si $\alpha$ es un sucesor ordinal.
$T \uparrow \alpha = lub(T \uparrow \beta: \beta < \alpha)$ si $\alpha$ es un ordinal límite.

Este definición permite caracterizar el mínimo punto fijo de una aplicación continua $T$ en términos de

$$lfp(T) = T \uparrow \omega$$

donde $\omega$ denota el primer ordinal infinito.

A continuación aplicaremos los resultados de la teoría del punto fijo a la definición semántica declarativa de los programas CLP(*H/E*). Dicha definición se obtiene en varios pasos:

• en primer lugar generalizamos el concepto estándar de base e interpretación de Herbrand a *r-H/E-base* y *r-H/E-interpretación,* que son conjuntos (posiblemente infinitos) de clases de equivalencia de átomos restringidos, mostrando también la relación entre estos conceptos y los que pueden ser obtenidos como una instancia de las definiciones en [Jaffar Lassez 86].

• el conjunto de las *r-H/E-interpretaciones* se organiza como un retículo $(I, \subseteq)$.

• se definen varios operadores de consecuencia inmediata $T_{i_{(P,H/E)}}$, $i = 1, 2, 3,$ asociados a un programa $P \cup E$, mostrando su continuidad sobre el retículo $(I, \subseteq)$. Esto permite la definición de varias semánticas punto fijo $T_{i_{(P,H/E)}} \uparrow \omega$ para el programa.

• se demuestra la equivalencia entre las diferentes semánticas de punto fijo y las distintas caracterizaciones del significado operacional del programa. De esta forma, los operadores de consecuencia inmediata modelan las diferentes propiedades observables del programa.

### 4.2.2. Generalización de los conceptos de base e interpretación de Herbrand.

Las siguientes definiciones instancian las definiciones en [Gabbrielli Levi 91a] al caso de programas CLP(*H/E*).

**Definición 4.2.2.1.** *(H/E-instancias)*

El conjunto de *H/E-instancias* $[c \square p(\tilde{X})]$ de un átomo restringido $c \square p(\tilde{X})$ se define como

$$[c \square p(\tilde{X})] = \{p(\tilde{X})\theta: \ \theta \text{ es una } H/E\text{-solución de } c\}$$

Esta definición puede extenderse a conjuntos de átomos restringidos. Sea $S$ un conjunto de átomos restringidos. Entonces

$$[S] = \bigcup_{A \in S} [A]$$

Para evitar redundancias en la definición de las interpretaciones, vamos a factorizar el conjunto de átomos restringidos respecto a una relación de equivalencia apropiada, que generaliza el concepto clásico de varianza de expresiones (≈). Informalmente, dos átomos restringidos serán equivalentes si contienen la misma información positiva. La relación de equivalencia va a ser definida como la intersección de una nueva relación de orden parcial (entre átomos restringidos) con su inversa.

**Definición 4.2.2.2.** *(preorden $\subseteq$)*

Sea $A$ el conjunto de átomos restringidos para un (Π,Σ)-programa $P \cup E$. Definimos el preorden $\subseteq$ sobre $A$ como sigue

$\tilde{X}$

$$c_1 \square p(\tilde{X}) \subseteq c_2 \square p(\tilde{X}) \quad \text{sii} \quad [c_1 \square p(\tilde{X})] \subseteq [c_2 \square p(\tilde{X})]$$

La relación de equivalencia inducida por el preorden $\subseteq$ sobre el conjunto de átomos restringidos será denotada por $\equiv$

$$c_1 \square p(\tilde{X}) \equiv c_2 \square p(\tilde{X}) \quad \text{sii} \quad c_1 \square p(\tilde{X}) \subseteq c_2 \square p(\tilde{X}) \wedge c_2 \square p(\tilde{X}) \subseteq c_1 \square p(\tilde{X}) \quad \text{sii} \quad [c_1 \square p(\tilde{X})] = [c_2 \square p(\tilde{X})]$$

es decir, dos átomos restringidos son equivalentes sii tienen las mismas *H/E-instancias,* i.e. sii las restricciones $c_1$ y $c_2$ tienen las mismas *H/E-soluciones* para las variables en $\tilde{X}$.

Se debe notar que no existe una caracterización sintáctica de la relación de equivalencia $\equiv$ puesto que no existe una propiedad de *forma normal* para las restricciones sobre *H/E*, i.e. no existe una representación explícita finita de las restricciones en forma resuelta (cfr. §2.2.5). En el caso de que la teoría ecuacional $E$ sea vacía, la relación de equivalencia $\equiv$ equivale a la relación de varianza (≈) entre expresiones. En este caso, $c_1 \square p(\tilde{X}) \equiv c_2 \square p(\tilde{X})$ sii $p(\tilde{X})\theta_1$ y $p(\tilde{X})\theta_2$ son variantes, siendo $\theta_i$ un mgu idempotente de $c_i$, $i = 1, 2$ (ver definición 3.1.7). En este caso sí se disfruta de una propiedad de forma normal, que corresponde a la posibilidad de representar cada ecuación mediante una ecuación en forma resuelta equivalente.

A continuación vamos a generalizar los conceptos estándar de base e interpretación de Herbrand, que son conjuntos (posiblemente infinitos) de átomos "ground", a conjuntos (posiblemente infinitos) de clases de átomos restringidos.

**Definición 4.2.2.3.** *(r-H/E-base)*

Sea $P \cup E$ un (Π,Σ)-programa CLP(*H/E*) y sea $A$ el conjunto de todos los átomos restringidos $c \square p(\tilde{X})$ asociados al programa, donde $c$ es una restricción *H/E-soluble*. Se define la *r-H/E-base* de interpretaciones, $B$, como el cociente del conjunto $A$ con respecto a la relación de equivalencia ≡. La ordenación inducida por $\subseteq$ sobre $B$ seguirá siendo denotada por $\subseteq$. Por simplicidad, *A* representará la clase de equivalencia del átomo restringido *A*.

**Definición 4.2.2.4.** *(r-H/E-interpretación)*

Una *r-H/E-interpretación* es cualquier subconjunto de $B$. El conjunto de todas las *r-H/E-interpretaciones* (el conjunto potencia de $B$) será denotado por $I$.

De acuerdo con la definición de *H/E-instancia,* si $I \in I$ es una *r-H/E-interpretación* entonces $[I] = \{p(\tilde{d}) \in H/E\text{-base}: p(\tilde{d}) \in [A] \wedge A \in I\}$.

**Definición 4.2.2.5.** *(verdad)*

Sea $I$ una *r-H/E-interpretación*. Un átomo restringido $c \square p(\tilde{X})$ es verdad en $I$ sii $[c \square p(\tilde{X})] \subseteq [I]$ lo que denotaremos como $I \models c \square p(\tilde{X})$. Una cláusula $H \leftarrow c \square B_1, B_2, ..., B_n$. es verdad en $I$ sii para cada *H/E-valoración* $\theta$ tal que $\theta$ es una *H/E-solución* de *c*, $\{B_1\theta, B_2\theta, ..., B_n\theta\} \subseteq [I]$ implica que $H\theta \in [I]$.

Informalmente, una cláusula es verdad en una interpretación restringida $I$ sii (cada *H/E-instancia* de la cláusula) es verdad en $[I]$.

**Definición 4.2.2.6.** *(r-H/E-modelo)*

Sea $P \cup E$ un (Π,Σ)-programa CLP(*H/E*). Un *r-H/E-modelo* del programa es cualquier *r-H/E-interpretación* en la que todas las cláusulas de $P$ son verdad.

A continuación mostramos la relación entre los conceptos anteriores y los conceptos de *H/E-base, H/E-interpretación* y *H/E-modelo* que pueden obtenerse instanciando al caso CLP(*H/E*) las definiciones en [Jaffar Lassez 86].

### 4.2.3. Relación con los conceptos CLP estándar.

**Definición 4.2.3.1.** *(H/E-base)*

Sea $\Pi = \Pi_C \cup \Pi_B$, con $\Pi_C \leftrightarrow \Pi_B = \varnothing$. Sea $P \cup E$ un $(\Pi,\Sigma)$-programa CLP(*H/E*). Se define la *H/E-base* de interpretaciones $B_{H/E}$ del programa, como el conjunto

$\{p(X_1, X_2, ..., X_n)\theta:\ p \in \Pi_B$ es *n*-ario $\wedge\ \theta$ es una *H/E-valoración* de las variables $X_1, X_2, ..., X_n\}$.

Informalmente, la *H/E-base* del programa es el producto cruzado de los símbolos de predicado de la signatura por los elementos del dominio de interpretación, *H/E*.

**Ejemplo 4.2.3.2.** Consideremos el programa

$$P=\{p(a)., q(X) \leftarrow f(X) = Y \,\square\, p(Y).\},$$
$$E = \{f(a)=a.\}$$

La partición más fina generada por $E$ sobre el universo de Herbrand $H$ del programa tiene una única clase de equivalencia a la cual pertenecen todos los elementos de $H$ $(a, f(a), f(f(a)), f^n(a), ...)$ y que representaremos por $a$. La *H/E-base* del programa $B_{H/E} = \{p(a), q(a)\}$. La *r-H/E-base* del programa puede representarse como $B = \{Y = a \,\square\, p(Y),\ Y = a \,\square q(Y)\}$. Se debe notar que la restricción $Y = a$ tiene un único *E-mgu* $\{Y/a\}$.

**Definición 4.2.3.3.** *(H/E-interpretación)*

Una *H/E-interpretación* de un programa $P \cup E$ es cualquier subconjunto de la *H/E-base*.

**Definición 4.2.3.4.** *(H/E-modelo)*

Un *H/E-modelo* de un programa $P \cup E$ es cualquier *H/E-interpretación* $I$ tal que para cualquier regla en $P$, $H \leftarrow c \,\square\, B_1, B_2, ..., B_n$, y para cualquier *H/E-valoración* $\theta$ sobre $H$, $c$ y $B_i$, $1 \le i \le n$, tal que $\theta$ es una *H/E-solución* de $c$,

$\{B_1\theta, B_2\theta, ..., B_n\theta\} \subseteq I$ implica que $H\theta \in I$.

Es evidente que la *H/E-base* de interpretaciones de un programa $B_{H/E}$ puede definirse en términos de nuestra *r-H/E-base* como $B_{H/E} = [B]$. Por otra parte, dado que la estructura $H/E$ es *compacta respecto a las soluciones* y no tiene elementos límite, cualquier *H/E-instancia* $p(d_1, d_2, ..., d_n) = p(X_1, X_2, ..., X_n)\theta$ puede representarse mediante un átomo restringido $c \square p(X_1, X_2, ..., X_n)$ donde $\theta$ es la única *H/E-solución* de la restricción $c$. Consecuentemente, cada *H/E-interpretación* puede representarse mediante una *r-H/E-interpretación* equivalente. Las *H/E-interpretaciones* son, en este sentido, un caso particular de *r-H/E-interpretaciones*. Análogamente, los *H/E-modelos* son sólo un caso especial de *r-H/E-modelos*. De hecho, si $M$ es un *H/E-modelo* del programa, existe un *r-H/E-modelo* $N$ tal que $M = [N]$ (cada *H/E-modelo* tiene asociado un *r-H/E-modelo* que contiene la misma información).

Es importante notar que nuestra noción de verdad es consistente con la noción estándar de verdad sobre las *H/E-interpretaciones* (ya que se define sobre *H/E-instancias*)Por otra parte, para cada *H/E-interpretación* $I$ existe un número (posiblemente infinito) de *r-H/E-interpretaciones* $I_1, I_2, ...$ tales que $I = [I_1] = [I_2], ...$ De la definición de verdad se deduce que $I$ es un *H/E-modelo* sii $I_1, I_2, ...$ son *r-H/E-modelos*. Pero, como veremos posteriormente, los diferentes $I_i$ pueden exhibir diferentes propiedades computacionales.

**Definición 4.2.3.5.** *(verdad en los H/E-modelos)*

Sea $P \cup E$ un programa CLP(*H/E*) y sea $G$ un átomo restringido.

$P \models_{H/E} G$ sii $G$ es verdad en todos los *H/E-modelos* de $P$
sii (de acuerdo a la definición 4.2.2.5.),
para cada *H/E-modelo* $M$ de $P$, $[G] \subseteq M$.

Dado que nuestro concepto de verdad es único, la verdad en todos los *H/E-modelos* equivale a la verdad en todos los *r-H/E-modelos*.

En el siguiente apartado se describen varias semánticas que modelan las propiedades computacionales importantes de los programas CLP(*H/E*). Todas ellas merecerán una caracterización por punto fijo, que será presentada en el punto siguiente.

### 4.2.4. Comportamiento observable de los programas CLP(*H/E*).

La semántica operacional de un lenguaje, desde un punto de vista puramente lógico, se define en términos de derivaciones que terminan con éxito. La definición operacional del lenguaje CLP(*H/E*) fue descrita en el capítulo tres de esta tesis por medio de conjuntos de reglas de inferencia que especifican la forma en que se efectúan las derivaciones a partir de un objetivo dado. Sin embargo, desde el punto de vista de la total caracterización de un lenguaje de programación, la semántica operacional debe dar cuenta de cierta información adicional, de ciertas propiedades observables, como el conjunto de respuestas computadas y, en su caso, el conjunto de bloqueos, el fallo finito, el fallo finito "ground", ... o alguna combinación adecuada de ellas. De hecho, un programa dado puede tener diferentes semánticas dependiendo de cuál es la propiedad que se observa.

En este apartado vamos a introducir tres semánticas operacionales $SS_i(P, H/E)$, $i = 1, 2, 3$. Cada una de ellas define una noción de "observable" (*conjunto de éxitos "ground", consecuencias atómicas* y *restricciones respuesta computadas*). En el resto del capítulo, nuestra aproximación a la caracterización operacional del lenguaje será "teoría de modelos".

Sin pérdida de generalidad, consideraremos la relación de transición estándar $\rightarrow_{CLP(H/E)}$ entre objetivos que se obtiene de la relación más general cuando el "constraint solver" se diseña, simplemente, para comprobar la solubilidad de las restricciones.

Asociada a una derivación con éxito $G \rightarrow^{*}_{CLP(H/E)}$ ricción respuesta $c$, la restricción en el último objetivo de la derivación. La primera semántica operacional que introducimos, $SS_1(P, H/E)$, generaliza de forma inmediata la semántica operacional tradicional de la programación lógica.

**Definición 4.2.4.1** *(Semántica de consecuencias atómicas "ground")*

Sea $P \cup E$ un programa CLP(*H/E*). El *conjunto de éxitos "ground"* $SS_1(P, H/E)$ de programa se define como sigue:

$$SS_1(P, H/E) = \{p(\tilde{d}) \in B_{H/E}: \quad (\leftarrow c \ \square p(\tilde{X}).) \rightarrow^{*}_{CLP(H/E)} (\leftarrow c' \ \square.) \wedge$$
$$c \text{ es finita } \wedge \ p(\tilde{d}) = p(\tilde{X})\theta \ \wedge \ \theta \text{ es una } H/E\text{-solución de } c'\}.$$

De esta forma, $SS_1(P, H/E)$ corresponde a la restricción a elementos de la *H/E-base* de la definición de conjunto de éxitos "ground" que se introduce en el esquema de lenguaje de programación lógica definido por [Jaffar et al 84a, Jaffar et al 86a]. En lo que sigue mantendremos este (ab)uso del calificativo "ground", aplicándolo a las expresiones cuyos argumentos toman valor sobre el dominio de computación, *H/E*.

Como veremos posteriormente, esta definición caracteriza todavía el conjunto de consecuencias atómicas "ground" del programa. Sin embargo, la definición de equivalencia entre programas basada en esta semántica es demasiado débil para distinguir entre programas con comportamiento operacional diferente.

**Ejemplo 4.2.4.2.** Consideremos los siguientes programas CLP(*H/E*)

$$W_1 = P_1 \cup E_1 \qquad\qquad W_2 = P_2 \cup E_2$$

$$P_1 = \{\ \ q(X) \leftarrow true \square.\ \}\ \ P_2 = \{q(X) \leftarrow a = X \square.,\ \ q(X) \leftarrow f(f(a)) = X \square.\ \}$$
$$E_1 = E_2 = E = \{\ f(f(a)) = f(a).\ \}$$

En este caso, la partición más fina generada por $E$ sobre el universo de Herbrand $H$ del programa tiene dos clases de equivalencia, que representaremos por $a$ y $f(a)$. A esta última clase pertenecen los elementos $f(a), f(f(a)), f(f(f(a))), f^n(a)$ ...

Es evidente que $W_1 \approx_1 W_2$ ya que

$$SS_1(P_1, H/E) = SS_1(P_2, H/E) = H/E\text{-}base = \{q(a), q(f(a))\}$$

Sin embargo, el objetivo $(\leftarrow true \square q(x))$ computa las restricciones respuesta $a = X$ y $f(f(a)) = X$ sólo en $W_2$. Este ejemplo demuestra, por tanto, que la relación $\approx_1$ es demasiado débil para caracterizar el comportamiento operacional del programa.

La segunda definición de conjunto de éxitos es la que se da en [Jaffar Lassez 86].

**Definición 4.2.4.3.** *(Semántica de consecuencias atómicas)*

Sea $P \cup E$ un programa CLP(*H/E*). El *conjunto de éxitos* $SS_2(P, H/E)$ se define como sigue:

$$SS_2(P, H/E) = \{ c \square p(\tilde{X}) \in B: (\leftarrow c \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c' \square.) \wedge$$
$$c \text{ es finita } \wedge \quad c \square p(\tilde{X}) \equiv c' \square p(\tilde{X}) \}.$$

Como veremos posteriormente, esta definición caracteriza el conjunto de consecuencias atómicas del programa. La correspondiente relación de equivalencia $\approx_2$ es más fuerte que $\approx_1$. De hecho, reconsiderando el ejemplo 4.2.4.2., es evidente que $\neg(W_1 \approx_2 W_2)$ ya que

$$SS_2(P_1, H/E) = \{true \square q(X),\ a = X \square q(X),\ f(f(a)) = X \square q(X) \}$$
$$\neq SS_2(P_2, H/E) = \{a = X \square q(X),\ f(f(a)) = X \square q(X)\}$$

Sin embargo, la definición tampoco caracteriza completamente el comportamiento operacional del programa ya que sigue sin hacer distinción entre programas que computan diferentes restricciones respuesta para un mismo objetivo.

**Ejemplo 4.2.4.4.** Consideremos los siguientes programas CLP(*H/E*)

$$W_1 = P_1 \cup E_1 \qquad\qquad W_2 = P_2 \cup E_2$$

$P_1 = \{ q(X) \leftarrow true \square. \}$
$P_2 = \{ q(X) \leftarrow true \square.,\ q(X) \leftarrow f(f(a)) = X \square. \}$
$E_1 = E_2 = E = \{ f(f(a)) = f(a). \}$

Es evidente que $W_1 \approx_1 W_2$ ya que

$$SS_1(P_1, H/E) = SS_1(P_2, H/E) = H/E\text{-}base = \{q(a), q(f(a))\}$$

Así mismo ocurre que $W_1 \approx_2 W_2$ ya que

$$SS_2(P_1, H/E) = SS_2(P_2, H/E) = \{c \square q(X) \in B: c \text{ es finita}$$
$$\wedge \quad c \text{ es } H/E\text{-soluble } \} = r\text{-}H/E\text{-}base =$$
$$\{true \square q(X),\ a = X \square q(X),\ f(f(a)) = X \square q(X) \}$$

Sin embargo, el objetivo $(\leftarrow true \square q(X))$ computa la restricción respuesta $f(f(a)) = X$ sólo en $W_2$.

Consecuentemente, necesitamos una noción más fuerte de éxito, capaz de capturar mayor información sobre el comportamiento operacional del programa. Introducimos la siguiente definición [Gabbrielli Levi 91a]

**Definición 4.2.4.5.** *(Semántica de restricción respuesta computada)*

Sea $P \cup E$ un programa CLP(*H/E*). La *semántica de restricción respuesta computada* $SS_3(P, H/E)$ se define como sigue:

$$SS_3(P, H/E) = \{ c \ \square p(\tilde{X}) \in B: \ (\leftarrow true \ \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c \ \square.) \}.$$

Veamos un ejemplo que ilustra la definición de la semántica $SS_3(P, H/E)$.

**Ejemplo 4.2.4.6.** *Número de nodos de un árbol binario.*

```
%  SPEC Number_of_Nodes

%  CONSTRUCTORS
%       0          /   0
%       succ       /   1
%       nil        /   0
%       tree       /   3

%  FUNCTIONS
%       +          /   2

%  PREDICATES
%       nn         /   2

%  RELATIONAL CLAUSES
       /* p1 */     nn( nil, 0 ).
       /* p2 */     nn( tree( Left_Tree, Element, Right_Tree ), N )        :-
                        N = Left_Num + Right_Num + succ( 0 )
                        □ nn( Left_Tree, Left_Num ), nn( Right_Tree, Right_Num ).

%  FUNCTIONAL CLAUSES
       /* e1 */     X + 0   →  X.
       /* e2 */     X + succ( Y )   →  succ( X + Y ).

%  ENDSPEC
```

Explicamos ahora brevemente la intención de los símbolos y reglas propuestos. *0, succ, nil* y *tree* son símbolos irreducibles que representan, respectivamente, el cero y el sucesor de los números naturales y los constructores de árboles binarios. + define la suma de naturales. *nn* asocia a un árbol binario su número de nodos.

La semántica $SS_3(P, H/E)$ de este programa *(P, E)* es el siguiente conjunto.

$SS_3(P, H/E) =$
$\{ X = nil,\ Y = 0 \square nn(X, Y),$
$X = tree(LT, E, RT),\ Y = s(0),\ LT = nil,\ RT = nil \square nn(X, Y),$
$X = tree(LT, E, RT),\ Y = s(s(0)),\ LT = nil,\ RT = tree(LT', E', RT'),$
$LT' = nil,\ RT' = nil \square nn(X, Y)\ \ldots\ \}$

La definición de $SS_3(P, H/E)$ entraña el contenido en información más rico, al caracterizar completamente las restricciones respuesta computadas por el programa. Como muestra, reconsideremos de nuevo el ejemplo 4.2.4.4. Es evidente que $\neg(W_1 \approx_{3)} W_2)$ ya que

$$SS_3(P_1, H/E) = \{true \square q(X)\}$$

mientras que

$$SS_3(P_2, H/E) = \{true \square q(X),\ f(f(a)) = X \square q(X).\ \}$$

Se debe notar que, si bien las diferentes definiciones de semántica operacional caracterizan diferentes propiedades computacionales del programa, todas ellas definen, abstractamente, el mismo conjunto de fórmulas. A continuación demostraremos el siguiente resultado, que será utilizado posteriormente: $SS_1(P, H/E) = [SS_2(P, H/E)] = [SS_3(P, H/E)]$. Para ello es necesario introducir un par de lemas técnicos.

**Lema 4.2.4.7.**

Sea $G = (\leftarrow c \square p(\tilde{X}).)$ un objetivo. Si $G$ tiene una derivación con éxito cuya restricción respuesta es $c'$ entonces el objetivo $G' = (\leftarrow c' \square p(\tilde{X}).)$ tiene también una derivación con éxito cuya respuesta es $c'$.

**Demostración.**

Obviamente, $c' = c \cup c^*$ para alguna restricción $c^*$ consistente con $c$. Por compacidad, si $c'$ es consistente cada una de sus partes finitas lo es. Entonces, repitiendo cada paso de la derivación a partir de $G$ añadiendo $c^*$ a cada restricción en la secuencia obtenemos la derivación a partir de $G'$ deseada. $\square$

**Lema 4.2.4.8.** [Gabbrielli Levi 91a]

Sea $G = (\leftarrow c_0 \square p(\tilde{t}).)$ un objetivo. Si $G$ tiene una derivación con éxito cuya restricción respuesta es $c$ entonces el objetivo $G' = (\leftarrow true \square p(\tilde{X}).)$ tiene una derivación con éxito

cuya respuesta es una restricción $c'$ tal que $c = c' \cup c_0 \cup \{\tilde{X} = \tilde{t}\}$. Además, si el objetivo $G' = (\leftarrow true \square p(\tilde{X}).)$ tiene una derivación con éxito cuya restricción respuesta es $c'$ y la restricción $c = c' \cup c_0$ es *H/E-soluble*, entonces $\leftarrow c_0 \square p(\tilde{X}).$ tiene una derivación con éxito cuya restricción respuesta es $c$. □

**Lema 4.2.4.9.**

$$SS_1(P, H/E) = [SS_2(P, H/E)] = [SS_3(P, H/E)].$$

**Demostración.**

- Demostraremos primero que $SS_1(P, H/E) = [SS_2(P, H/E)]$.

*(⊆)* Consideremos un elemento cualquiera $p(\tilde{d}) \in SS_1(P, H/E)$. Por definición de $SS_1(P, H/E)$, para alguna restricción finita $c_0$ existe una derivación

$$(\leftarrow c_0 \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c' \square.)$$
$$\text{tal que } p(\tilde{d}) \in [c' \square p(\tilde{X})].$$

Por el lema 4.2.4.7, está garantizada la existencia de una derivación a partir del objetivo $(\leftarrow c' \square p(\tilde{X}).)$ con restricción respuesta $c'$. Se deduce entonces que $p(\tilde{d}) \in [SS_2(P, H/E)]$. □

*(⊇)* Consideremos un elemento cualquiera $p(\tilde{d}) \in [SS_2(P, H/E)]$. Por definición de $SS_2(P, H/E)$, existirá una derivación $(\leftarrow c_0 \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c' \square.)$ con $c_0$ finita tal que $p(\tilde{d}) \in [c_0 \square p(\tilde{X})] = [c' \square p(\tilde{X})]$. Se deduce entonces que $p(\tilde{d}) \in SS_1(P, H/E)$. □

- Demostraremos ahora que $SS_1(P, H/E) = [SS_3(P, H/E)]$.

*(⊆)* Consideremos un elemento cualquiera $p(\tilde{d}) \in SS_1(P, H/E)$. Por definición de $SS_1(P, H/E)$, para alguna restricción finita $c_0$ existe una derivación

$$(\leftarrow c_0 \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c' \square.)$$
$$\text{tal que } p(\tilde{d}) \in [c' \square p(\tilde{X})].$$

Por el lema 4.2.4.8, está garantizada la existencia de una derivación a partir del objetivo $(\leftarrow true \square p(\tilde{X}).)$ con restricción respuesta $c''$ tal que $c' = c'' \cup c_0$. Dado que $[c' \square p(\tilde{X})] \subseteq [c'' \square p(\tilde{X})]$, se deduce que $p(\tilde{d}) \in [SS_3(P, H/E)]$. □

*(⊆)* Consideremos un elemento cualquiera $p(\tilde{d}) \in [SS_3(P, H/E)]$. Por definición de $SS_3(P, H/E)$, existirá una derivación $(\leftarrow true \square p(\tilde{X}).) \rightarrow^*_{CLP(H/E)} (\leftarrow c' \square.)$ tal que $p(\tilde{d}) \in [c' \square p(\tilde{X})]$. Dado que *true* es una restricción finita, se deduce que $p(\tilde{d}) \in SS_1(P, H/E)$, lo que completa la prueba. □

A continuación establecemos los resultados de corrección y de completitud para $SS_3(P, H/E)$ como instancia de los resultados en [Gabbrielli Levi 91a] para el caso de la estructura *H/E*. Informalmente, el siguiente teorema viene a establecer que la semántica $SS_3(P, H/E)$ de un programa $P \cup E$ (que es un conjunto de átomos retringidos, de la forma $c \square p(\tilde{X})$ puede verse como un conjunto (posiblemente infinito) de cláusulas sin cuerpo (cláusulas "unit"), de la forma $p(\tilde{X}) \leftarrow c \square.$, y que las restricciones respuesta pueden computarse "ejecutando" directamente el objetivo en dicho "programa" $SS_3(P, H/E)$.

**Teorema 4.2.4.10.** *(Corrección y completitud para $SS_3(P, H/E)$)*

Sea $P \cup E$ un programa CLP(*H/E*) y sea $G = \leftarrow c_0 \square A_1, ..., A_n.$ un objetivo. Entonces $G \rightarrow^*_{CLP(H/E)} (\leftarrow c_{ans} \square.)$ sii existen *n* átomos restringidos $(c_i \square B_i) \in SS_3(P, H/E)$, $i = 1, 2, ..., n$, sin variables comunes con $G$ ni con el resto, tales que $(c_0 \wedge c_1 \wedge ... \wedge c_n \wedge A_1 = B_1 \wedge ... \wedge A_n = B_n)$ y $c_{ans}$ tienen las mismas *H/E*-soluciones para las variables en *G*. □

Obviamente, este resultado no se cumple si consideramos la semántica $SS_2(P,H/E)$. A modo de contraejemplo, reconsideremos el programa

$$P = \{ q(X) \leftarrow true \square. \}$$
$$E = \{ f(f(a)) = f(a). \}$$

En este caso,

$$SS_2(P, H/E) = \{true \square q(X),\ a = X \square q(X),\ f(f(a)) = X \square q(X) \}$$

Sin embargo, el objetivo $(\leftarrow true \square q(X))$ no computa, e.g., la restricción respuesta $X = a$.

En lo que sigue vamos a caracterizar por la técnica del punto fijo cada una de las tres propiedades computacionales de los programas CLP(*H/E*) anteriormente descritas.

### 4.2.5. Semántica de punto fijo de los programas CLP(*H/E*)

En programación lógica tradicional, sobre el retículo de las interpretaciones de Herbrand se define una aplicación continua $T_P$ (operador de consecuencia inmediata) asociada a un programa *P*. La definición de $T_P$ es como sigue [van Emden Kowalski 76]:

$T_P(I) = \{ d \in$ *base de Herbrand:* $\exists\ A \leftarrow B_1, B_2, ..., B_n. \in P,$
$\exists$ una *substitución "ground"* $\theta$ tal que
$A\theta = d, \{B_1\theta, B_2\theta, ..., B_n\theta\} \subseteq I \}$

Las potencias ordinales de $T_P$ se definen de la siguiente forma:

$T_P \uparrow 0 = \varnothing.$
$T_P \uparrow(\alpha+1) = T_P(T_P \uparrow \alpha)$ si $\alpha$ es un sucesor ordinal.
$T_P \uparrow \alpha = \bigcup_{\beta < \alpha} (T_P \uparrow \beta)$ si $\alpha$ es un ordinal límite.

Este definición permite caracterizar el menor punto fijo de la aplicación $T_P$ en términos de sus potencias ordinales:

$$lfp(T_P) = T_P \uparrow \omega$$

Por otra parte, el conjunto de éxitos "ground" del programa coincide con el menor punto fijo de la aplicación $T_P$,

$$lfp(T_P) = T_P \uparrow \omega = SS(P)$$

Este resultado estándar se generaliza de forma inmediata al caso CLP(*H/E*). Definimos la siguiente aplicación $T_{1(P, H/E)}(I)$ entre *H/E-interpretaciones:*

**Definición 4.2.5.1.** Sea $P \cup E$ un programa CLP(*H/E*) y sea $I \subseteq [B] = B_{H/E}$.

$T_{1(P, H/E)}(I) = \{ d \in B_{H/E}:\ \exists\ A \leftarrow c \square B_1, B_2, ..., B_n, \in P,$
$\exists$ una *H/E-valoración* $\theta$ tal que
$H/E \models c\theta, H/E \models A\theta = d,$
$\{B_1\theta, B_2\theta, ..., B_n\theta\} \subseteq I \}$

**Definición 4.2.5.2.** *(potencias de T)*

Las potencias ordinales de un operador monótono $T_{i_{(P,\ H/E}}$ ,$i$ = *1, 2, 3,* (las transformaciones $T_2$ y $T_3$ serán introducidas posteriormente) se definen del siguiente modo:

$T_{i_{(P,\ H/E)}} \uparrow 0 = \varnothing$.
$T_{i_{(P,\ H/E)}} \uparrow (\alpha+1) = T_{i_{(P,\ H/E)}}(T_{i_{(P,\ H/E)}} \uparrow \alpha)$ si $\alpha$ es un sucesor ordinal.

$T_{i_{(P,\ H/E)}} \uparrow \alpha = \bigcup_{\beta < \alpha} (T_{i_{(P,\ H/E)}} \uparrow \beta)$ si $\alpha$ es un ordinal límite.

Dado que nuestro lenguaje es una instancia del esquema CLP, las propiedades esperadas para las transformaciones $T_{i_{(P,\ H/E)}}$ se obtienen de forma directa particularizando a la estructura *H/E* los resultados en [Gabbrielli Levi 91a, Jaffar Lassez 86]:

**Lema 4.2.5.3.**

$T_{1_{(P,\ H/E)}}$ es continua (y, por tanto, monótona) sobre el retículo completo $(I_{H/E}, \subseteq)$, donde $I_{H/E} = \{[I]: I \in I\}$ y existe un menor menor punto fijo $lfp(T_{1_{(P,\ H/E)}}) = T_{1_{(P,\ H/E)}} \uparrow \omega$. □

El siguiente teorema proporciona la caracterización por punto fijo de la semántica $SS_1(P, H/E)$. La prueba es estándar [Jaffar Lassez 86].

**Teorema 4.2.5.4**

Sea $P \cup E$ un programa CLP(*H/E*). Entonces $SS_1(P, H/E) =$

$= lfp(T_{1_{(P,\ H/E)}}) = T_{1_{(P,\ H/E)}} \uparrow \omega$. □

Se debe notar que, aunque la semántica operacional que se define en [Jaffar Lassez 86] se corresponde con $SS_2(P, H/E)$, la semántica de punto fijo que allí se presenta caracteriza $SS_1(P, H/E)$.

A continuación introducimos un nuevo operador de consecuencia inmediata que nos permite caracterizar la semántica operacional $SS_3(P, H/E)$, aquella que proporciona el contenido operacional más rico.

**Definición 4.2.5.5.** Sea $P \cup E$ un programa CLP(*H/E*) y sea $I \subseteq B$.

$T_{3_{(P, H/E)}}(I) = \{c \square p(\tilde{X}) \in B:$
$\exists$ una variante de una cláusula en $P$
$p(\tilde{t}) \leftarrow c_0 \square p_1(\tilde{t}_1), p(\tilde{t}_2), ..., p_n(\tilde{t}_n) \in P,$
$\exists\, c_i \square p_i(\tilde{X}) \in I,\ 1 \leq i \leq n,$ sin variables comunes,
$c' = c_0 \cup \{\tilde{X}_1 = \tilde{t}_1, ..., \tilde{X}_n = \tilde{t}_n\} \cup \{c_1, ..., c_n\},$
$c'$ es *H/E-soluble,* finita y $c = c' \cup \{\tilde{X} = \tilde{t}\}\}$.

El siguiente lema nos permite definir una semántica de punto fijo usando el operador $T_{3_{(P, H/E)}}$.

**Lema 4.2.5.6.**

La aplicación $T_{3_{(P, H/E)}}$ es continua sobre el retículo completo $(I, \subseteq)$. Existe un menor punto fijo $lfp(T_{3_{(P, H/E)}}) = T_{3_{(P, H/E)}} \uparrow \omega$. □

El siguiente teorema proporciona la caracterización por punto fijo de la semántica $SS_3(P, H/E)$.

**Teorema 4.2.5.7.** *(equivalencia entre la semántica operacional y la semántica de punto fijo).*

Sea $P \cup E$ un programa CLP(*H/E*). Entonces $SS_3(P, H/E) =$
$= lfp(T_{3_{(P, H/E)}}) = T_{3_{(P, H/E)}} \uparrow \omega$. □

También resulta posible dar una caracterización por punto fijo de la semántica $SS_2(P, H/E)$ modificando adecuadamente la definición de $T_{3_{(P, H/E)}}$. Se debe notar que $SS_2(P, H/E)$ contiene (las clases de equivalencia de) los átomos restringidos que son verdad en todos los *r-H/E-modelos* del programa, mientras que $SS_3(P, H/E)$ contiene sólo aquellos que pueden ser computados por él. Consecuentemente, $SS_2(P, H/E)$ podría definirse equivalentemente añadiendo en cada paso de derivación cualquier restricción $c^*$ consistente con las restricciones *H/E-solubles* ya computadas. Así pues, para capturar $SS_2(P, H/E)$ se debe modificar $T_{3_{(P, H/E)}}$ añadiendo en cada paso todas las restricciones consistentes, lo que conduce a la siguiente definición:

**Definición 4.2.5.8.** Sea $P \cup E$ un programa CLP(*H/E*) y sea $I \subseteq B$.

$T_{2_{(P,\ H/E)}}(I) = \{c \square p(\tilde{X}) \in B:$
$\exists$ una variante de una cláusula en $P$
$p(\tilde{t}) \leftarrow c_0 \square p_1(\tilde{t}_1), p_2(\tilde{t}_2), ..., p_n(\tilde{t}_n) \in P,$
$\exists\, c_i \square p_i(\tilde{X}_i) \in I,\ 1 \leq i \leq n,$ sin variables comunes,
$c' = c_0 \cup \{\tilde{X}_1 = \tilde{t}_1, ..., \tilde{X}_n = \tilde{t}_n\} \cup \{c_1, ..., c_n\} \cup c^*,$
$c'$ es *H/E-soluble,* finita y $c = c' \cup \{\tilde{X} = \tilde{t}\}\}$.

Para $T_{2_{(P,\ H/E)}}$ se cumplen también los resultados habituales, como queda establecido por el siguiente lema.

**Lema 4.2.5.9.**

$T_{2_{(P,\ H/E)}}$ es continua (y, por tanto, monótona) sobre el retículo completo $(I, \subseteq)$ y existe un menor menor punto fijo $lfp(T_{2_{(P,\ H/E)}}) = T_{2_{(P,\ H/E)}} \uparrow \omega$. □

Análogamente a los casos anteriores, el siguiente teorema proporciona una caracterización por punto fijo de la semántica $SS_2(P, H/E)$.

**Teorema 4.2.5.10.**

Sea $P \cup E$ un programa CLP(*H/E*). Entonces $SS_2(P, H/E) =$
$= lfp(T_{2_{(P,\ H/E)}}) = T_{2_{(P,\ H/E)}} \uparrow \omega$. □

El siguiente teorema resume, finalmente, la caracterización por punto fijo de las semánticas operacionales $SS_i(P, H/E)$, $i = 1, 2, 3$.

**Teorema 4.2.5.11.**

Sea $P \cup E$ un programa CLP(*H/E*). Entonces $SS_i(P,\ H/E) = lfp(T_{i_{(P,\ H/E)}}) = T_{i_{(P,\ H/E)}} \uparrow \omega$, $i = 1, 2, 3$. □

En la siguiente sección se establece la relación esperada entre los conceptos de consecuencia lógica bajo la teoría ecuacional $E$, verdad en los *r-H/E-modelos,* las funciones $T_{i_{(P,\ H/E)}} \uparrow \omega$, $i = 1, 2, 3,$ y la relación de transición $\rightarrow_{CLP(H/E)}$.

## 4.3. Semántica por teoría de modelos.

En lo que sigue vamos a ver que las distintas semánticas operacionales introducidas en el apartado anterior son, todas ellas, (*r-H/E-*)modelos del programa. Sin embargo, dichos modelos carecen de una propiedad relevante de los modelos de Herbrand: la intersección de un conjunto de modelos ya no es, necesariamente, un modelo. El siguiente ejemplo pone de manifiesto esta circunstancia.

**Ejemplo 4.3.1.** Consideremos el siguiente programa

$$P=\{\quad q(X) \leftarrow true \;\square.,\; q(X) \leftarrow f(f(a)) = X \;\square.\;\}$$
$$E = \{f(f(a)) = f(a).\}$$

tal como vimos en la sección anterior,

- $SS_1(P, H/E) = \{q(a), q(f(a))\}$ que puede ser representado de forma única mediante la *r-H/E-interpretación* equivalente $SS'_1(P, H/E) = \{a = X \;\square\; q(X),\; f(f(a)) = X \;\square\; q(X)\}$

- $SS_2(P, H/E) = \{true \;\square\; q(X),\; a = X \;\square\; q(X),\; f(f(a)) = X \;\square\; q(X)\}$

- $SS_3(P, H/E) = \{true \;\square\; q(X),\; f(f(a)) = X \;\square\; q(X)\}$

Es inmediato comprobar que las *r-H/E-interpretaciones* $SS'_1(P, H/E)$, $SS_2(P, H/E)$ y $SS_3(P, H/E)$ son, todas ellas, *r-H/E-modelos* del programa. Sin embargo, su intersección es el conjunto $\{f(f(a)) = X \;\square\; q(X)\}$, que no es ya un *r-H/E-modelo* del programa. En consecuencia, no existe un modelo mínimo respecto al orden entre interpretaciones basado en la inclusión de conjuntos.

La ausencia de la propiedad de intersección de modelos se entiende fácilmente teniendo en cuenta la noción de inclusión entre conjuntos no refleja adecuadamente el hecho de que un átomo restringido representa el conjunto de sus instancias sobre el dominio. De hecho, la información contenida en una *r-H/E-interpretación* $I_1$ puede venir recogida también por otra *r-H/E-interpretación* $I_2$ sin que $I_1 \subseteq I_2$. Otro aspecto a considerar está ligado a la capacidad que tienen las *r-H/E-interpretaciones* para modelar restricciones respuesta computadas. Por ejemplo, la *r-H/E-interpretación* $\{true \;\square\; p(\tilde{X}),\; X=a \;\square\; p(\tilde{X})\}$ contiene más información que $\{true \;\square\; p(\tilde{X})\}$ aún en caso de que tuvieran las mismas *H/E-instancias*. Ambas propiedades deben ser consideradas si se desea establecer un orden entre interpretaciones.

### 4.3.1. El retículo de las interpretaciones y la propiedad de intersección de modelos.

De acuerdo con [Falaschi et al 91, Gabbrielli Levi 91a], para superar el problema que se plantea al generalizar el concepto de interpretación, definimos un orden parcial no estricto entre *r-H/E*-interpretaciones que, en el caso particular de *H/E-interpretaciones* [Jaffar Lassez 86], equivale a la inclusión $\subseteq$ entre conjuntos. Este orden será establecido teniendo en cuenta las dos propiedades comentadas en el párrafo anterior. Mostraremos que el conjunto de las *r-H/E*-interpretaciones junto con este orden parcial sigue siendo un retículo completo y que el *glb* de cualquier conjunto de *r-H/E-modelos* es también un modelo. Este resultado garantiza la existencia de un *r-H/E-modelo* mínimo que corresponde al concepto estándar de menor modelo de Herbrand o, equivalentemente, al conjunto de éxitos "ground" del programa. Este modelo es completamente satisfactorio si se está únicamente interesado en los aspectos lógicos del programa. Veremos que las otras semánticas operacionales, las que caracterizan un comportamiento más rico, son igualmente modelos (no minimales) del programa.

**Definición 4.3.1.1.** *(preorden $\leqslant$ entre r-H/E-interpretaciones)*

Definimos el preorden $\leqslant$ sobre $I$ como sigue. Sean $I_1, I_2 \in I$. Entonces

- $I_1 \subseteq_I I_2$ sii $\forall A_1 \in I_1 \; \exists A_2 \in I_2$ tal que $A_1 \subseteq A_2$ y
- $I_1 \leqslant I_2$ sii ($I_1 \subseteq_I I_2$) y ($I_2 \subseteq_I I_1$ implica $I_1 \subseteq I_2$).

El significado intuitivo de las relaciones definidas anteriormente es el siguiente. $I_1 \subseteq_I I_2$ significa que cualquier átomo de la *H/E-base* que está en $[I_1]$ está también en $[I_2]$ ($I_2$ contiene más información positiva). $I_1 \leqslant I_2$ significa que, o bien $I_2$ contiene estrictamente más información positiva que $I_1$ o, si la cantidad de información es la misma, que $I_1$ la expresa con menor número de elementos que $I_2$ ($I_2$ es más redundante).

La definición anterior y el lema siguiente aplican los resultados en [Gabbrielli Levi 91a] al caso CLP(*H/E*).

**Lema 4.3.1.2.**

La relación $\leqslant$ es un preorden sobre $I$. Más aún, ($I$, $\leqslant$) es un retículo completo. $B$ es el elemento "top" y $\varnothing$ es el "bottom". □

**Lema 4.3.1.3.**

Sea $P \cup E$ un programa CLP(*H/E*) y sea $M$ el conjunto de los *r-H/E-modelos* del programa. Entonces *glb(M)* (de acuerdo con la anterior definición de orden $\preceq$) es un *r-H/E-modelo* del programa. □

**Definición 4.3.1.4.**

Sea $P \cup E$ un programa CLP(*H/E*). Su semántica teoría de modelos

$$M_{(P,\ H/E)} = glb\ \{N:\ N \text{ es un } r\text{-}H/E\text{-}modelo \text{ de } P \cup E\ \}.$$

Por otra parte, los resultados en [Jaffar Lassez 86] muestran la existencia de un *H/E-modelo* mínimo, el equivalente al menor modelo de Herbrand de la programación lógica tradicional. El siguiente teorema en [Gabbrielli Levi 91a] muestra que este modelo, representado como una interpretación restringida equivalente, es el mismo que el menor *r-H/E-modelo*.

**Lema 4.3.1.5.**

Sea $P \cup E$ un programa CLP(*H/E*). Su *H/E-modelo* mínimo $M_{(P,\ H/E)}$ es su menor *r-H/E-modelo* mínimo $M_{(P,\ H/E)}$ □

Analogamente al caso de la programación lógica, este modelo caracteriza el conjunto de consecuencias lógicas "ground" del programa y es igual a la semántica de punto fijo "ground" $T_{I_{(P,\ H/E)}} \uparrow \omega$.

**Teorema 4.3.1.6.**

Sea $P \cup E$ un programa CLP(*H/E*). Sea $p(\tilde{d}) \in B_{H/E}$. Entonces

$$P \models_{H/E} p(\tilde{d}) \quad \text{sii} \quad p(\tilde{d}) \in SS_I(P, H/E) \quad \text{sii} \quad p(\tilde{d}) \in T_{I_{(P,\ H/E)}} \uparrow \omega$$
$$\text{sii } p(\tilde{d}) \in M_{(P,\ H/E)} \quad \text{sii} \quad (P,E) \models p(\tilde{d})$$ □

En la sección anterior se han introducido otros modelos del programa. El siguiente lema muestra la relación entre ellos, poniendo de manifiesto el hecho de que $SS_2(P, H/E)$ y $SS_3(P, H/E)$ son modelos no minimales.

**Lema 4.3.1.7.** Sea $P \cup E$ un programa CLP(*H/E*). Entonces $SS_2(P, H/E)$ y $SS_3(P, H/E)$ son *r-H/E-modelos* del programa. Además,

$$SS_1(P, H/E) \leqslant SS_3(P, H/E) \leqslant SS_2(P, H/E) \quad \square$$

**Ejemplo 4.3.1.8.** Consideremos el siguiente programa

$$P = \{q(x) \leftarrow true \ \square. \ \}$$
$$E = \{ f(f(a)) = f(a). \ \}.$$

En este caso,

- $SS_1(P, H/E) = \{q(a), q(f(a)\},$
- $SS_2(P, H/E) = \{true \ \square \ q(X), \ a = X \ \square \ q(X), \ f(f(a)) = X \ \square \ q(X)\},$
- $SS_3(P, H/E) = \{true \ \square \ q(X)\}$

y se cumple, por tanto

$$SS_1(P, H/E) \leqslant SS_3(P, H/E) \leqslant SS_2(P, H/E)$$

Se debe notar, por otra parte, que la relación de orden entre *r-H/E-modelos* no está relacionada directamente con el mayor contenido de información.

Para finalizar, vamos a establecer una correspondencia completa entre la semántica lógica y la semántica algebraica del lenguaje, considerando la teoría $E$ que corresponde, en el sentido de [Jaffar Lassez 86], a la estructura *H/E*. Obtenemos así los esperados resultados de correción y completitud para derivaciones con éxito respecto de la semántica $SS_2(P, H/E)$.

### 4.3.2. Dualidad entre las aproximaciones lógica y algebraica a la semántica por teoría de modelos.

El siguiente teorema, demostrado para el caso CLP genérico en [Jaffar Lassez 86], establece la dualidad entre las aproximaciones lógica y algebraica a la semántica por teoría de modelos de los programas CLP(*H/E*).

**Teorema 4.3.2.1.**

Sea $P \cup E$ un programa CLP($H/E$) y sea $G = c \square p(\tilde{X})$ un átomo restringido. Entonces las siguientes afirmaciones son equivalentes

- $G \in SS_2(P, H/E)$
- $[G] \in SS_1(P, H/E)$ (es decir, $SS_1(P, H/E) = [G: (P,E) \models \tilde{\exists} G]$).
- $\leftarrow G$ tiene una $\rightarrow_{CLP(H/E)}$ derivación con éxito.
- $P \models_{H/E} \tilde{\exists} G$.
- $(P,E) \models \tilde{\exists} G$. □

# 5 La aproximación CLP(*H/E*) al diseño de Bases de Datos

Una de las aproximaciones de mayor interés al desarrollo de aplicaciones de bases de datos aboga por el uso de tecnología de programación lógica [Gallaire et al 78, Gallaire et al 84, Kowalski 81, Lloyd 87]. Este enfoque permite formalizar el concepto de base de datos y ofrece, adicionalmente, un lenguaje único para expresar de manera uniforme datos, programas, vistas, preguntas y restricciones de integridad. La lógica subyacente permite la expresión declarativa de todos estos conceptos a la vez que proporciona el soporte teórico necesario, e.g., para establecer las propiedades de corrección y completitud del proceso de evaluación de preguntas o de un método de simplificación para verificar (de forma eficiente) la satisfacción de restricciones de integridad.

En los capítulos precedentes hemos formalizado el lenguaje CLP(H/E), diseñado para integrar la programación lógica y ecuacional de manera flexible, sencilla y con buen soporte formal, como una instancia del esquema de programación lógica con restricciones especializada en resolver ecuaciones bajo una teoría ecuacional. Como una aplicación interesante de las técnicas propuestas, en este capítulo se estudia la utilidad de las mismas para la validación de requerimientos de aplicaciones de bases de datos. En particular, mostramos que dichas técnicas son capaces de operar con especificaciones ejecutables en dos modos distintos, uno de los cuales envuelve una capacidad inferencial que puede utilizarse para resolver problemas de generación automática de planes [dos Santos et al 82, Furtado Moura 86, Kowalski 81, Nilsson 82, Warren 74].

La organización del capítulo es como sigue. En la sección primera se introduce el problema de la modelización conceptual de aplicaciones de bases de datos, una de las actividades de mayor interés dentro del proceso de desarrollo de la aplicación, cuyo objetivo es la obtención de una descripción (ejecutable) precisa, correcta y completa del sistema objeto. Se revisan también los principales marcos linguísticos utilizados para dicha descripción. Introducimos entonces la idea de describir el esquema conceptual como una teoría formal en una lógica de primer orden con igualdad. En nuestra aproximación, basada en el nuevo paradigma de programación lógico-ecuacional con restricciones, la teoría será presentada como un programa CLP(*H*/*E*). Las definiciones

semánticas declarativa y operacional del lenguaje dan así soporte formal a los aspectos de interpretación y ejecutabilidad de la teoría. La sección segunda discute los requerimientos de un ejemplo guía, que será utilizado en el resto del capítulo. El lenguaje CLP(*H/E*) será utilizado para especificar el esquema conceptual de la base de datos del ejemplo y comentaremos las ventajas obtenidas con nuestra formulación. La sección tercera está dedicada a examinar dos modos distintos (análisis y síntesis) de operar con este tipo de especificaciones. Ambos modos resultan de utilidad para la validación del modelo conceptual del sistema (mediante ejecución simbólica). El segundo, además, se relaciona con la capacidad de sintetizar secuencias alternativas de operaciones que, partiendo de un estado inicial, lleven a la base de datos a un nuevo estado caracterizado por sus propiedades observables (por el conjunto de hechos que pueden considerarse ciertos en él). Este problema es similar a la formulación clásica (en el campo de la inteligencia artificial) del problema de la formación (o generación automática) de planes. La definición de un marco formal unificado hace posible que problemas como éste sean estudiados como un problema único conectando distintas áreas. En este capítulo, por tanto, caracterizamos las técnicas de programación lógico - ecuacional con restricciones como un eslabón importante para establecer este enlace.

## 5.1. Modelización conceptual de aplicaciones de Bases de Datos.

Entre las diferentes actividadades en el campo del diseño de aplicaciones de Bases de Datos, la Modelización Conceptual [Brodie et al 84, Gustafsson et al 82, dos Santos et al 81, Olivé 86] juega un papel fundamental. Constituye la primera fase en el desarrollo de cualquier aplicación de Bases de Datos y su salida, el modelo conceptual, describe cualquier información, restricción o requerimiento referente al sistema, con la intención de representar nuestro conocimiento acerca de una parcela de interés del mundo real. Un modelo conceptual es declarativo e independiente de la estructura interna del sistema.

Una de las líneas de investigación más atractivas en este campo es el estudio de técnicas y herramientas para derivar sistemáticamente un sistema operacional a partir de sus especificaciones (prototipado rápido [Agresti 86, Balzer 85]). Es una tesis ampliamente aceptada que este objetivo puede alcanzarse de forma elegante siguiendo una aproximación formal. La especificación formal, que asumimos ejecutable, puede considerarse como un prototipo (de muy alto nivel) del sistema (cfr. §1.1).

Las diferentes aproximaciones al problema de especificar formalmente el esquema conceptual de una aplicación de bases de datos difieren en su nivel de precisión matemática y en su claridad semántica. Algunas propuestas bien establecidas intentan

combinar ideas desarrolladas en el campo de la especificación algebraica con técnicas consolidadas de bases de datos [Dosh et al 82, Ehrich 86, Ehrich et al 78, Ehrich et al 86, Gogolla 89, Sernadas et al 87]. Por otra parte, el diseño de aplicaciones de bases de datos en lógica, tal y como fue propuesto en [Gallaire et al 84, Kowalski 81, Reiter 84] y posteriormente discutido en [Ceri et al 89, Lloyd 87, Minker 88], soporta la tarea de la modelización conceptual mediante técnicas de programación lógica [Bubenko Olivé 86, Gustafsson et al 82, Olivé 86, Veloso Furtado 85, Weigand 85]. La utilidad de las técnicas de especificación lógico - ecuacional para la validación de los modelos conceptuales de aplicaciones de bases de datos fue discutida en [dos Santos et al 81, Furtado Moura 86, Veloso et al 81, Veloso Furtado 85]. Sin embargo, ninguno de estos trabajos aborda el problema de la ejecutabilidad de tales especificaciones ni, consecuentemente, de la derivación (eficiente) de respuestas a partir de los axiomas de la especificación.

En nuestra aproximación [Alpuente Ramírez 90], los estados de la base de datos se denotan mediante composición de operaciones de actualización que, presentadas por medio de símbolos de función, llevan estados sobre estados. A su vez, la igualdad y otros predicados se utilizan para afirmar propiedades y relaciones sobre un estado. Se muestra cómo la lógica de cláusulas de Horn con igualdad soporta, de manera uniforme, la definición de las restricciones de integridad que expresan las condiciones de aplicabilidad de las operaciones de actualización, el efecto esperado de dichas operaciones y la respuesta a las preguntas. Debido a la incorporación de la relación de igualdad y al uso de variables lógicas, esta aproximación soporta la tarea de la modelización conceptual de una forma que es estrictamente más potente que la que sustentan el estilo lógico o el estilo ecuacional por separado. En este capítulo extendemos dichas técnicas con la introducción de restricciones. En nuestra aproximación, el modelo conceptual del sistema será presentado como un programa CLP(*H/E*). La satisfacción de las restricciones de integridad será verificada mediante técnicas de resolución de restricciones. Mostraremos también cómo la representación intensional (por medio de restricciones) del conjunto de respuestas a un requerimiento de consulta (a un objetivo CLP(*H/E*)) proporciona facilidades adicionales que enriquecen este campo.

## 5.2. Ejemplo guía.

En una aplicación de bases de datos es habitual considerar dos tipos de operaciones, *actualizaciones* y *preguntas,* y algunas propiedades que la base de datos debe satisfacer en cualquier instante para ser consistente, las *restricciones de integridad.* Las actualizaciones son operaciones que modifican las informaciones contenidas en la

base produciendo una transición de estado. Las preguntas son operaciones de interrogación sobre el contenido de la base de datos en un estado dado. Las restricciones de integridad son aquellas afirmaciones que deben cumplirse en todo momento para asegurar la consistencia de la información almacenada. Las restricciones de integridad pueden ser violadas por la aplicación de una operación de actualización. Se asume que algún tipo de mecanismo para hacer cumplir las restriciones de integridad (e.g. basado en precondiciones) rechazará algunos requerimientos de actualización para mantener la información consistente (e.g. aquellas actualizaciones cuyas precondiciones no se satisfacen).

**Ejemplo 5.2.1.** En [Veloso et al 81] se introduce el ejemplo de la *agencia de empleo* para ilustrar diferentes técnicas de especificación formal de una aplicación de base de datos. A continuación revisamos la especificación (informal) original de este problema.

Asumimos que la base de datos se inicializa a un *estado vacío* (denotado por *nil*). En un estado dado, las personas pueden *solicitar* empleo y las empresas pueden *ofrecer* un número de plazas, *contratar* candidatos y *despedir* empleados. En el sistema se imponen las siguientes restricciones de integridad:

• Una persona puede solicitar empleo una sola vez, adquiriendo entonces la condición de *candidato*. Esta condición se pierde si la persona es contratada por una empresa, en cuyo caso pasa a merecer la consideración de *empleado*. Si el candidato es despedido, recupera su condición de candidato.

• Una empresa puede ofrecer plazas varias veces. El número (positivo) de plazas ofrecidas se acumula para definir el número de vacantes de dicha empresa.

• Sólo es posible contratar a personas con la condición de candidato y sólo por empresas que tienen vacantes.

Las preguntas a las que el sistema debe ser capaz de responder son las siguientes: verificar si una persona es o no un candidato, comprobar la empresa para la que trabaja un empleado y calcular el número de vacantes de una empresa dada.

Con la idea de estructurar los requerimientos anteriores, consideraremos las *actualizaciones* como operaciones que llevan estados sobre estados y las preguntas como operaciones (que serán, probablemente, evaluadas a un valor de verdad) que entregan la respuesta a una consulta acerca del estado.

Desde esta perspectiva, el repertorio de posibles actualizaciones sobre un estado *S* de la base de datos es:

| | |
|---|---|
| *nil* | Inicialización sin parámetros<br>*precondición* ninguna |
| *apply(X,S)* | La persona *X* solicita empleo, pasando a ser un candidato.<br>*precondición* *X* no es candidato ni es un empleado. |
| *offer(Y,N,S)* | La empresa *Y* ofrece *N* plazas. El número de vacantes de la empresa *Y* se incrementa *precondición* $N > 0$. |
| *hire(X,Y,S)* | La empresa *Y* contrata al candidato *X*. El número de vacantes en *Y* se decrementa en *1*. *X* deja de ser candidato para adquirir la condición de empleado.<br>*precondición* *X* es candidato e *Y* tiene vacantes. |
| *fire(X,Y,S)* | La empresa *Y* despide al empleado *X*. El número de vacantes en *Y* se incrementa en *1*. *X* recupera su condición de candidato.<br>*precondición* *X* trabaja para la empresa *Y*. |

A continuación definimos las preguntas con que se preve interrogar a la base de datos

| | |
|---|---|
| *freepos(Y,S)* | devuelve el número de vacantes de la empresa *Y* en el estado *S*. |
| *iscand(X,S)* | verifica si la persona *X* es o no un candidato en el estado *S*. La respuesta es *cierto* si *X* ha solicitado empleo y no ha sido contratado o si, habiendo sido contratado, fue posteriormente despedido. |
| *isemployee(X,S)* | verifica si la persona *X* es un empleado en el estado *S*. La respuesta es *cierto* si *X* ha sido contratado y no ha sido posteriomente despedido. |
| *worksfor(X,Y,S)* | comprueba si la persona *X* trabaja para la empresa *Y* en el estado *S*. La respuesta es *cierto* si *X* ha sido contratado por *Y* y no ha sido posteriomente despedido. |
| *haspos(Y,S)* | verifica si una empresa *Y* tiene vacantes en el estado *S*. |

La base de datos, partiendo del estado inicial, alcanza otros estados como resultado de las actualizaciones. Cada estado *T* de la base de datos puede quedar, por tanto, caracterizado por medio de alguna composición adecuada de operaciones de actualización que lleven a la base de datos desde el estado inicial hasta *T*. Sin embargo, es evidente que diferentes secuencias de actualizaciones pueden denotar un mismo estado del sistema, en lo que respecta a las respuestas que se obtendrán a las preguntas planteadas sobre dicho estado. Por ejemplo, las secuencias

*fire(X,Y,hire(X,Y,offer(Y,1,apply(X,nil))))*

y

*offer(Y,1,apply(X,nil))*

representan un mismo estado de la base de datos. Considerando todas aquellas secuencias que denotan un mismo estado como pertenecientes a una misma clase de equivalencia, resulta obvia la necesidad de disponer de un representante de clase. Seguiremos un

método constructivo de especificación algebraica [Guttag Horning 80, Choquet et al 86] para la especificación del esquema conceptual de la aplicación. La manera de establecer el representante de clase se basa, entonces, en la siguiente idea. Entre las operaciones de actualización distinguiremos un cierto número de operaciones privilegiadas (constructores de estado) asociadas a los eventos (o actualizaciones) que pueden afectar a la vida del sistema. Los constructores se consideran operaciones privilegiadas en el sentido de que son suficientes para representar todos los posibles valores que se requieren para denotar un estado de la base de datos. De esta forma, el estado será denotado por un término bien formado constituído sólo por símbolos constructores. Los operadores no constructores serán definidos por su efecto sobre términos constituídos sólo por constructores. En el caso general, resulta destacable el hecho de que puede haber más de un conjunto de operadores adoptable como conjunto de constructores. Dependiendo del problema, el conjunto elegido puede ser puro (sin relaciones entre las operaciones que lo forman o, equivalentemente, tal que todo par de secuencias distintas formadas con símbolos de conjunto denotan estados distintos) o impuro (con relaciones entre ellos). En este último caso, sería preciso introducir los axiomas que expresan las relaciones entre los operadores constructores (axiomas purificadores). Es habitual escoger como conjunto de constructores el conjunto menos impuro o bien el mínimo (criterios que suelen llevar, muy frecuentemente, al mismo conjunto). Para el ejemplo de la agencia de empleo, podemos elegir como constructores de estado las siguientes operaciones: *nil, apply, offer* y *hire* ya que se asume que una operación de despido *fire* equivale a la cancelación de un contrato (sería comparable a la operación *pop* en un tipo abstracto de datos *pila*). Obviamente, en caso de ampliarse los requerimientos de consulta con la pregunta acerca de si, en el pasado, una persona trabajó para cierta empresa, la operación *fire* sería también tomada como símbolo constructor.

Para describir el efecto de una operación sobre un estado, resulta adecuado describir cómo afecta ésta al correspondiente término canónico que denota dicho estado. La definición será dada, por tanto, por inducción estructural sobre términos constructores. Pero es de utilidad notar que algunas actualizaciones requieren la aplicación previa de otras si han de ser aceptables y productivas (e.g., no es posible contratar a una persona a no ser que ésta sea un candidato y que la empresa tenga vacantes). Así pues, deberemos especificar las (pre-)condiciones para cada actualización, necesarias para garantizar la satisfacción de las restricciones de integridad en cada momento. Las precondiciones aseguran que cada transición desde el estado actual de la base de datos es consistente con las condiciones de integridad de la base y protegen al sistema de entradas erróneas. Los requerimientos de actualización cuyas precondiciones no se satisfagan en un cierto estado serán rechazados.

De acuerdo con la hipótesis habitual de observabilidad [Veloso Furtado 85], el estado puede también caracterizarse por sus propiedades observables, que pueden ser analizadas a través de las respuestas a los requerimientos de consulta. Especificamos las operaciones de interrogación por medio de funciones o relaciones que se definen, al igual que actualizaciones y restricciones de integridad, por inducción sobre la estructura del estado.

## 5.3. La aproximación CLP(*H/E*) a la modelización conceptual de aplicaciones de Bases de Datos.

Antes del problema de especificación en sí, debe tomarse la decisión acerca de qué formalismo emplear. De la discusión anterior resulta evidente que dicho formalismo debe proporcionar, al menos, las siguientes facilidades, que deben ser ofrecidas dentro de un marco formal uniforme:

- igualdad y otras relaciones (para dar soporte a la especificación de las operaciones de actualización y de las respuestas a los requerimientos de consulta).
- distinción entre constructores y funciones definidas (para dar soporte a la metodología constructiva de especificación algebraica anteriormente propuesta).
- axiomas entre constructores (para el rechazo de requerimientos de actualización inaceptables y, por tanto, improductivos)
- mecanismos de resolución y simplificación de restricciones (para verificar y hacer cumplir las restricciones de integridad de la base).
- algún tipo de tratamiento de la negación (que permita definir y deducir la falsedad de una relación, en particular de la igualdad).

Las técnicas de programación ecuacional pueden establecer explícitamente qué secuencias de operaciones tienen el mismo efecto, interpretando los correspondientes términos bien formados sobre la congruencia definida por las ecuaciones. Ya que el modo de computación asociado al paradigma de programación funcional se basa en *evaluación de expresiones*, en un formalismo puramente funcional todo lo que se puede hacer para animar la especificación es:

- verificar si dos términos sin variables $t_1$ y $t_2$ están en una misma clase de equivalencia (intentando derivar la ecuación $t_1 = t_2$ de la especificación o utilizando, e.g., técnicas de reescritura para verificar si los dos términos pueden ser representados por un mismo término irreducible).

- evaluar una pregunta (puramente funcional) reduciéndola a su forma normal.

En un formalismo puramente lógico (sin igualdad) el principal inconveniente es que se pierde la posibilidad de definir funciones. No existe el concepto de función como tal. Los símbolos de función sirven como meros constructores de datos. Todas las funciones, en el sentido convencional, deben ser presentadas como relaciones con un argumento adicional para entregar el resultado. En otras palabras, se define el grafo de la función en vez de la función en sí. Otra limitación es la ausencia de estructura y de modularidad, lo que se convierte en un serio inconveniente cuando la especificación contiene centenares de axiomas. Sin embargo, el uso de la unificación y su resultado, la variable lógica, son la base de algunas características relevantes, como invertibilidad, estructuras de datos parcialmente definidas y no determinismo. La computación se realiza en modo *deducción* de manera que es posible, dado un objetivo, encontrar la sustitución que hace el teorema cierto bajo la teoría especificada. En un formalismo lógico-ecuacional, es posible formular el problema en cualquier estilo, funcional o relacional, existiendo la posibilidad de elegir el formalismo más apropiado para cada aplicación particular y se puede operar con especificaciones ejecutables tanto en modo *evaluación* como en modo *deducción*. Se permite que aparezcan símbolos de función interpretados como argumento de relaciones y que variables cuantificadas existencialmente aparezcan como argumentos de funciones. Esta aproximación, consecuentemente, soporta la tarea de la modelización conceptual de una forma que es estrictamente más potente que la que sustentan el estilo lógico o el ecuacional por separado. En el paradigma de programación con restricciones se reemplaza la unificación por el concepto más general de resolución de restricciones sobre un dominio de computación dado y se incorporan técnicas adicionales para una manipulación efectiva de las mismas. El concepto clásico de *sustitución respuesta computada* se reemplaza por el concepto (mucho más general y compacto) de *restricción respuesta computada* (cfr. §2.2.3.2). Si la integración de los estilos lógico y ecuacional se formaliza dentro de este marco, i.e. en el contexto de CLP(*H*/*E*), tenemos la potencia combinada de los tres paradigmas, capaz de dar soporte a

- la definición de la igualdad y otras relaciones (que se especifican mediante cláusulas de programa)
- la posibilidad de distinguir entre constructores y funciones (que se definen mediante una teoría ecuacional de Horn)
- la posibilidad de definir axiomas entre constructores (presentados también como axiomas de esta teoría)
- la incorporación de mecanismos potentes de resolución y simplificación de restricciones (propuestos en la definición del lenguaje)

- una aproximación (ecuacional) a la negación

En lo que respecta a los diferentes modos de computación, la presencia de símbolos de función interpretados como argumento de los predicados y el uso multimodo de las definiciones de ambos puede ahora ser tratado de manera uniforme mediante las técnicas de resolución de restricciones ecuacionales desarrolladas en los capítulos anteriores. La representación intensional (por medio de restricciones) del conjunto de respuestas a un requerimiento (a un objetivo CLP(*H/E*)) proporciona facilidades adicionales que serán presentadas en el punto siguiente. Se debe notar que, debido a que la aproximación CLP(*H/E*) a la negación es ecuacional, las precondiciones deben ser formalizadas como funciones para hacer posible la definición y deducción de su falsedad. La satisfacción de las restricciones de integridad, consecuentemente, será verificada también haciendo uso de técnicas de resolución de restricciones.

Para ilustrar los argumentos discutidos en el párrafo anterior, a continuación se presenta una especificación CLP*(H/E)* del esquema conceptual de la base de datos de la agencia de empleo. Usaremos un conjunto finito de cláusulas de programa CLP*(H/E)* para definir el efecto de cada actualización y la respuesta a cada consulta sobre el estado. También las precondiciones, alguna clase de axiomas entre constructores y, en general, cualquier detalle sobre el sistema, se especifican de forma análoga.

```
%       SPEC agbd
%       CONSTRUCTORS
%           nil         /   0
%           apply       /   2
%           offer       /   3
%           hire        /   3
%       FUNCTIONS
%           fire        /   3
%           freepos     /   2
%           pcapply     /   2
%           pcoffer     /   3
%           pchire      /   3
%           pcfire      /   3
%       PREDICATES
%           iscand      /   2
%           isemployee  /   2
%           worksfor    /   3
%           haspos /    2
```

```
%        RELATIONAL CLAUSES

%        CLAUSES FOR IS CANDIDATE
iscand(X,apply(Z,T))          :-  X=Z,pcapply(Z,T)  □          .
iscand(X,apply(Z,T))          :-  ¬(X=Z),pcapply(Z,T)          □ iscand(X,T).
iscand(X,offer(W,M,T))        :-  pcoffer(W,M,T) □ iscand(X,T).
iscand(X,hire(Z,W,T))         :-  ¬(X=Z),pchire(Z,W,T)         □ iscand(X,T).

%        CLAUSES FOR IS  EMPLOYEE
isemployee(X,apply(Z,T)) :-  ¬(X=Z),pcapply(Z,T)  □          isemployee(X,T).
isemployee(X,offer(W,M,T)) :-  pcoffer(W,M,T) □ isemployee(X,T).
isemployee(X,hire(Z,W,T)) :-  X=Z,pchire(Z,W,T)  □  .
isemployee(X,hire(Z,W,T)) :-  ¬(X=Z),pchire(Z,W,T)  □          isemployee(X,T).

%        CLAUSES FOR WORKS  FOR
worksfor(X,Y,apply(Z,T))      :-  ¬(X=Z),pcapply(Z,T)  □ worksfor(X,Y,T).
worksfor(X,Y,offer(W,N,T))    :-  pcoffer(W,N,T) □ worksfor(X,Y,T).
worksfor(X,Y,hire(Z,W,T)) :-  X=Z,Y=W,
                                  pchire(Z,W,T)  □          .
worksfor(X,Y,hire(Z,W,T)) :-  ¬(X=Z),pchire(Z,W,T)  □          worksfor(X,Y,T).

%        CLAUSES FOR HAS POSITIONS
haspos(Y,apply(Z,T))          :-  pcapply(Z,T)          □ haspos(Y,T).
haspos(Y,offer(W,N,T))        :-  Y=W,pcoffer(W,N,T)          □ .
haspos(Y,offer(W,N,T))        :-  ¬(Y=W),pcoffer(W,N,T) □ haspos(Y,T).
haspos(Y,hire(Z,W,T))         :-  Y=W,freepos(Y,T)>1,
                                  pchire(Z,W,T)          □ .
haspos(Y,hire(Z,W,T))         :-  ¬(Y=W),pchire(Z,W,T)  □ haspos(Y,T).

%        EQUATIONAL CLAUSES

%        AXIOMS BETWEEN CONSTRUCTORS
apply(Z,T)                    →   T                          :- ¬pcapply(Z,T).
offer(W,M,T)                  →   T                          :- ¬pcoffer(W,M,T).
hire(X,Y,T)                   →   T                          :- ¬pchire(X,Y,T).

%        CLAUSES FOR FIRE
fire(X,Y,apply(Z,T))          →   apply(Z,fire(X,Y,T))       :- pcapply(Z,T),
                                                                pcfire(X,Y,apply(Z,T)).
fire(X,Y,offer(W,M,T))        →   offer(W,M,fire(X,Y,T)) :- pcoffer(W,M,T),
                                                               pcfire(X,Y,offer(W,M,T)).
fire(X,Y,hire(Z,W,T))         →   T                          :- X=Z,W=Y,
                                                                   pchire(Z,W,T),
                                                                pcfire(X,Y,hire(Z,W,T)).
fire(X,Y,hire(Z,W,T))         →   hire(Z,W,fire(X,Y,T))  :- ¬(X=Z),
                                                                   pchire(Z,W,T),
                                                                pcfire(X,Y,hire(Z,W,T)).
fire(X,Y,T)                   →   T                          :- ¬pcfire(X,Y,T).

%        CLAUSES FOR FREE POSTS
freepos(Y,nil)                →   0.
freepos(Y,apply(Z,T))         →   freepos(Y,T)               :- pcapply(Z,T).
freepos(Y,offer(W,N,T))       →   freepos(Y,T)+N             :-  Y=W,pcoffer(W,N,T).
freepos(Y,offer(W,N,T))       →   freepos(Y,T)               :-  ¬(Y=W),
                                                                  pcoffer(W,N,T).
```

| | | | | |
|---|---|---|---|---|
| *freepos(Y,hire(Z,W,T))* | → | *freepos(Y,T)-1* | :- | *Y=W,pchire(Z,W,T).* |
| *freepos(Y,hire(Z,W,T))* | → | *freepos(Y,T)* | :- | *¬(Y=W), pchire(Z,W,T).* |

```
%          CLAUSES FOR PCAPPLY
        pcapply(X,nil).
        ¬pcapply(X,apply(Z,T))                          :-  X=Z,pcapply(Z,T).
        pcapply(X,apply(Z,T))      →  pcapply(X,T)      :-  ¬(X=Z),pcapply(Z,T).
        pcapply(X,offer(W,M,T))    →  pcapply(X,T)      :-  pcoffer(W,M,T).
        ¬pcapply(X,hire(Z,W,T))                         :-  X=Z,pchire(Z,W,T).
        pcapply(X,hire(Z,W,T))     →  pcapply(X,T)      :-  ¬(X=Z),pchire(Z,W,T).

%          CLAUSES FOR PCOFFER
        pcoffer(W,M,T)             :-  M>0.
        ¬pcoffer(W,M,T)            :-  ¬(M>0).

%          CLAUSES FOR PCHIRE
        pchire(X,Y,T)              →  pchire_(X,Y,T)  :-  freepos(Y,T) > 0.
        ¬pchire(X,Y,T)                                :-  ¬freepos(Y,T) > 0.
        ¬pchire_(X,Y,nil).
        pchire_(X,Y,apply(Z,T))                       :-  X=Z,pcapply(Z,T).
        pchire_(X,Y,apply(Z,T))    →  pchire_(X,Y,T)  :-  ¬(X=Z),pcapply(Z,T).
        pchire_(X,Y,offer(W,M,T))  →  pchire_(X,Y,T)  :-  pcoffer(W,M,T).
        ¬pchire_(X,Y,hire(Z,W,T))                     :-  X=Z,pchire_(Z,W,T).
        pchire_(X,Y,hire(Z,W,T))   →  pchire_(X,Y,T)  :-  ¬(X=Z),
                                                          pchire_(Z,W,T).
%          CLAUSES FOR PCFIRE
        ¬pcfire(X,Y,nil).
        ¬pcfire(X,Y,apply(Z,T))                       :-  X=Z,pcapply(Z,T).
        pcfire(X,Y,apply(Z,T))     →  pcfire(X,Y,T)   :-  ¬(X=Z),pcapply(Z,T).
        pcfire(X,Y,offer(W,N,T))   →  pcfire(X,Y,T)   :-  pcoffer(W,N,T).
        pcfire(X,Y,hire(Z,W,T))                       :-  X=Z,Y=W,
                                                          pchire(Z,W,T).
        pcfire(X,Y,hire(Z,W,T))    →  pcfire(X,Y,T)   :-  ¬(X=Z),pchire(Z,W,T).

%       ENDSPEC
```

La teoría ecuacional de Horn en el programa anterior satisface las propiedades de *confluencia* y *terminación*, además de ser *completamente definida*. Es conocido que los axiomas entre constructores no pueden usarse, en general, como reglas de reescritura de términos, como también lo es que las condiciones sintácticas impuestas sobre las cláusulas para garantizar la propiedad de confluencia de la teoría (ver §3.1) dejan de garantizarla, en general, si se permite la incorporación de axiomas entre constructores. Sin embargo, se debe notar que la clase de axiomas entre constructores utilizados en este tipo de especificaciones (axiomas de rechazo de las actualizaciones que no satisfacen las restricciones de integridad, e.g. *apply(Z,T)* $\varnothing$ *T :- ¬pcapply(Z,T)*) conduce a reglas que son, absolutamente, inofensivas ya que son fácilmente orientables y su estructura garantiza que la propiedad de *no ambigüedad* del programa se mantiene.

Por ejemplo, consideremos un objetivo de la forma

*:- iscand(X,apply(Z,T) )*

y el anterior axioma que expresa la no aplicabilidad de *apply(Z,T)*. Dado que las cláusulas en la definición de *iscand* que tienen *apply(Z,T)* como argumento contienen siempre en el cuerpo la restricción *pcapply(Z,T),* los cuerpos de estas cláusulas y el cuerpo del axioma de no aplicabilidad correspondiente son, de forma inmediata, *incompatibles,* i.e., no pueden ser evaluados simultáneamente a *true* [González et al 90].

**Ejemplo 5.3.1.** Para terminar este apartado presentamos un nuevo ejemplo, que resulta casi obligatorio. Este ejemplo, enunciado bajo múltiples variantes, ha sido distinguido como el estándar de gran número de conferencias que tratan el tema de la modelización en bases de datos y sistemas de información. Se trata de la modelización conceptual del sistema de organización de una conferencia internacional que congrega expertos e investigadores trabajando en temas afines para presentar y discutir temas técnicos de interés específico. Como es habitual, se forman dos tipos de comisiones: el comité de programa (encargado de los contenidos técnicos de la conferencia) y el comité de organización (encargado de los aspectos de financiación, publicidad, planificación de actividades, alquiler de locales, alojamiento...). Las dos comisiones deben trabajar conjuntamente ya que utilizan información compartida. Se trata de modelizar el sistema que soporte las actividades en que participan, coordinadamente, ambas comisiones y que, con algunas simplificaciones, formulamos a continuación.

Comité de programa

- redactar la invitación a presentar trabajos
- catalogar y distribuir los trabajos recibidos entre los revisores
- evaluar los trabajos en función de los resultados de la revisión y decidir su aceptación
- atender peticiones de retirada de trabajos

Comité de organización

- distribuir invitaciones de asistencia a la conferencia
- registrar las confirmaciones de asistencia
- atender posibles anulaciones

Para simplificar, supondremos que las preguntas con que se pretende interrogar a este sistema hacen referencia a las siguientes informaciones:

- número de trabajos presentados en la conferencia
- número de participantes en la misma
- trabajos aceptados
- participantes

A continuación presentamos una especificación CLP(*H/E*) que se aproxima a los anteriores requerimientos, en la que las restricciones de integridad impuestas al sistema son las esperadas. La especificación muestra que es posible expresar la no aplicabilidad de una operación de actualización sin recurrir al uso de axiomas entre constructores. Basta para ello definir apropiadamente el resto de axiomas de forma que se ignore la operación cuando no sea admisible (cfr., e.g., las cláusulas que definen la relación *iscand* del ejemplo anterior y la relación *isparticipant* de éste). La ventaja de esta formulación es que, en ella, todos los símbolos constructores son irreducibles. Esto permite aplicar, por tanto, las optimizaciones del algoritmo de "narrowing" descritas en §3.3.1.3 para tales símbolos. Por otra parte, la eliminación de los axiomas purificadores tiene el único efecto de establecer una partición en clases de equivalencia más fina, que distingue entre términos "data" distintos que pertenecían antes a una misma clase. Esta característica, lejos de resultar inadecuada, será conveniente en aquellos sistemas en los que el orden en que se producen los eventos es significativo para la definición de la traza que representa la historia del sistema.

```
%       SPEC Int'l_conf

%       CONSTRUCTORS
%           nil             /   0
%           submit          /   3
%           accept          /   2
%           register   /        2
```

```
%       FUNCTIONS
%           withdraw        /   2
%           cancel          /   2
%           n_papers /      1
%           n_participants  /   1
%           pcsubmit /      3
%           pcaccept /      2
%           pcregister      /   2
%           pcwithdraw      /   2
%           pccancel /      2

%       PREDICATES
%           accepted /      2
%           isparticipant   /   2

%       RELATIONAL CLAUSES

%               CLAUSES FOR ACCEPTED

accepted(N,submit(Q,M,S))   :-      □ accepted(N,S).
accepted(N,accept(M,S))     :-      N=M, pcaccept(M,S) □.
accepted(N,accept(M,S))     :-      ¬(N=M), pcaccept(M,S) □ accepted(N,S).
accepted(N,accept(M,S))     :-      ¬pcaccept(M,S) □ accepted(N,S).
accepted(N,register(P,S))   :-      □ accepted(N,S).

%               CLAUSES FOR  IS PARTICIPANT

isparticipant(P,submit(Q,M,S)) :-           □ isparticipant(P,S).
isparticipant(P,accept(M,S))   :-           □ isparticipant(P,S).
isparticipant(P,register(Q,S)) :-   P=Q, pcregister(Q,S) □.
isparticipant(P,register(Q,S)) :-   ¬(P=Q ), pcregister(Q,S) □ isparticipant(P,S).
isparticipant(P,register(Q,S)) :-   ¬pcregister(Q,S)    □ isparticipant(P,S).

%       EQUATIONAL CLAUSES

%               CLAUSES FOR WITHDRAW

withdraw(N,submit(Q,M,S))   →   submit(Q,M,withdraw(N,S)).
withdraw(N,acceptM,S))      →   S       :-  N = M,
pcaccept(M,S).
withdraw(N,accept(M,S))     →   withdraw(N,S)           :-  ¬(N = M),
                                                            pcaccept(M,S).
withdraw(N,accept(M,S))     →   withdraw(N,S)           :-  ¬pcaccept(M,S).
withdraw(N,register(Q,S))   →   register(Q,withdraw(N,S)).
withdraw(N,S)               →   S                       :-  ¬pcwithdraw(N,S).

%               CLAUSES FOR CANCEL

cancel(P,submit(Q,M,S))     →   submit(Q,M,cancel(P,S)).
cancel(P,accept(M,S))       →   accept(M,cancel(P,S)).
cancel(P,register(Q,S))     →   S                       :-  P= Q,
                                                            pcregister(Q,S).
                                                            pccancel(P,register(Q,S)),
cancel(P,register(Q,S))     →   register(Q,cancel(P,S))  :-  ¬(P = Q),
                                                            pcregister(Q,S).
```

```
cancel(P,register(Q,S))   →  register(Q,cancel(P,S))  :-  ¬pcregister(Q,S).
cancel(P,S)               →  S                        :-  ¬pccancel(P,S).


%                  CLAUSES FOR N_PAPERS

n_papers(nil)             →  0.
n_papers(submit(P,N,S))   →  n_papers(S).
n_papers(accept(N,S))     →  n_papers(S) + 1  :-  pcaccept(N,S).
n_papers(accept(N,S))     →  n_papers(S)      :-  ¬pcaccept(N,S).
n_papers(register(P,S))   →  n_papers(S).

%                  CLAUSES FOR N_PARTICIPANTS

n_participants(nil)                →  0.
n_participants(submit(P,N,S))      →  n_participants(S).
n_participants(accept(N,S))        →  n_participants(S).
n_participants(register(P,S))      →  n_participants(S) + 1   :-  pcregister(P,S).
n_participants(register(P,S))      →  n_participants(S)       :-  ¬pcregister(P,S).

%                  CLAUSES FOR PCSUBMIT

pcsubmit(P,N,nil).
¬pcsubmit(P,N,submit(Q,M,S))                          :-  N = M, pcsubmit(Q,M,S).
pcsubmit(P,N,submit(Q,M,S))  →  pcsubmit(P,N,S)       :-  ¬(N= M),
                                                          pcsubmit(Q,M,S).
pcsubmit(P,N,submit(Q,M,S))  →  pcsubmit(P,N,S)       :-  ¬pcsubmit(Q,M,S).
pcsubmit(P,N,accept(M,S))    →  pcsubmit(P,N,S).
pcsubmit(P,N,register(Q,S))  →  pcsubmit(P,N,S).

%                  CLAUSES FOR PCACCEPT

¬pcaccept(N,nil).
pcaccept(N,submit(Q,M,S))                             :-  N=M,
                                                          pcsubmit(Q,M,S).
pcaccept(N,submit(Q,M,S))    →  pcaccept(N,S)         :-  ¬(N = M),
                                                          pcsubmit(Q,M,S).
pcaccept(N,submit(Q,M,S))    →  pcaccept(N,S)         :-  ¬pcsubmit(Q,M,S).
¬pcaccept(N,accept(M,S))                              :-  N = M,
                                                          pcaccept(M,S).
pcaccept(N,accept(M,S))      →  pcaccept(N,S)         :-  ¬(N = M),
                                                          pcaccept(M,S).
pcaccept(N,accept(M,S))      →  pcaccept(N,S)         :-  ¬pcaccept(M,S).
pcaccept(N,register(Q,S))    →  pcaccept(N,S).

%                  CLAUSES FOR PCREGISTER

pcregister(P,nil).
pcregister(P,submit(Q,M,S))          →  pcregister(P,S).
pcregister(P,accept(M,S))            →  pcregister(P,S).
¬pcregister(P,register(Q,S))                          :-  P= Q,
                                                          pcregister(Q,S).
pcregister(P,register(Q,S))          →  pcregister(P,S)   :-  ¬(P = Q),
                                                              pcregister(Q,S).
pcregister(P,register(Q,S))          →  pcregister(P,S)   :-  ¬pcregister(Q,S).
```

```
%                CLAUSES FOR PCWITHDRAW

¬pcwithdraw(N,nil).
¬pcwithdraw(N,submit(Q,M,S))                                  :- N=M,
                                                                 pcsubmit(Q,M,S).
pcwithdraw(N,submit(Q,M,S))      →  pcwithdraw(N,S)           :- ¬(N = M),
                                                                 pcsubmit(Q,M,S).
pcwithdraw(N,submit(Q,M,S))      →  pcwithdraw(N,S)           :- ¬pcsubmit(Q,M,S).
pcwithdraw(N,acceptM,S))                                      :- N = M,
                                                                 pcaccept(M,S).
pcwithdraw(N,accept(M,S))        →  pcwithdraw(N,S)           :- ¬(N = M),
                                                                 pcaccept(M,S).
pcwithdraw(N,accept(M,S))        →  pcwithdraw(N,S)           :- ¬pcacept(M,S).
pcwithdraw(N,register(Q,S))      →  pcwithdraw(N,S).

%                CLAUSES FOR PCCANCEL

¬pccancel(P,nil).
pccancel(P,submit(Q,M,S))        →  pccancel(P,S).
pccancel(P,accept(M,S))          →  pccancel(P,S).
pccancel(P,register(Q,S))                                     :- P= Q,
                                                                 pcregister(Q,S).
pccancel(P,register(Q,S))        →  pccancel(P,S)             :- ¬(P = Q),
                                                                 pcregister(Q,S).
pccancel(P,register(Q,S))        →  pccancel(P,S)             :- ¬pcregister(Q,S).

%          ENDSPEC
```

## 5.4. Usando CLP(*H/E*) para la verificación de modelos conceptuales.

En esta sección se analiza, en el marco de la programación lógico-ecuacional (concretamente en el contexto de CLP(*H/E*)), la posibilidad de desarrollar herramientas software (de asistencia, e.g., al diseño de bases de datos) provistas de alguna característica experta tal como una cierta capacidad inferencial que permita llegar a sintetizar secuencias alternativas de acciones que, partiendo de un estado inicial, lleven al sistema a un nuevo estado (caracterizado por sus propiedades observables) [dos Santos et al 81, Veloso Furtado 85, Furtado Moura 86]. Esto permitiría, dada una aplicación, interrogar exhaustivamente, y en diversos modos, su esquema conceptual, haciendo uso de las técnicas de ejecución asociadas. A continuación presentamos una caracterización de dos modos interesantes de tratar con especificaciones de modelos conceptuales de aplicaciones de bases de datos en el lenguage CLP(*H/E*). Dichos modos de ejecución serán relacionados formalmente con las técnicas desarrolladas en los capítulos precedentes.

### 5.4.1. Modo análisis.

En este modo, dado un estado inicial $S$ y una secuencia $t$ de actualizaciones, el problema está en verificar si dicha secuencia es capaz de producir la transición desde el estado $S$ a un estado final $T$ (caracterizado a través de sus propiedades observables) donde ciertas propiedades son ciertas o no y bajo qué asunciones será dicho estado alcanzado. A continuación se presentan algunos ejemplos que ilustran la ejecución de la especificación del ejemplo guía en modo análisis.

Podemos plantear los siguientes objetivos:

• ¿ Bajo qué condiciones se alcanzará un estado en que la empresa *E2* tiene vacantes tras una secuencia de eventos en que la empresa *E1* ofrece 2 plazas y luego la empresa *e* ofrece $N$ plazas?:

```
:- ▯ haspos(E2,offer(e,N,offer(E1,s(s(0)),nil))).

% Respuestas:

   E2 = e, N > 0;
   E2 = E1, ¬ (E2 = e), N > 0;
   E2 = E1, ¬ (N > 0);
   no
```

• ¿ Se alcanzará un estado en el que $p$ es un candidato mediante una secuencia de actualizaciones donde la persona $Z$ solicita empleo, posteriormente lo solicita $Y$ y finalmente $X$ ?

```
:- ▯ iscand(p, apply(X,apply(Y,apply(Z,nil)))).

% Respuestas:

   X = p, pcapply(p,apply(Y,apply(Z,nil)));
   Y = p, ¬ (X = p), pcapply(X,apply(p,apply(Z,nil))),
                        pcapply(p,apply(Z,nil));
Z = p, ¬ (Y = p), ¬ (X = p), pcapply(Y,apply(p,nil)),
                   pcapply(X,apply(Y,apply(p,nil)));
Z = p, ¬ (X = p), ¬ pcapply(Y,apply(p,nil)),
                   pcapply(X,apply(Y,apply(p,nil)));
Y = p, ¬ pcapply(X,apply(p,apply(Z,nil))),
                   pcapply(p,apply(Z,nil));
Z = p, ¬ (Y = p), pcapply(Y,apply(p,nil)),
                   ¬ pcapply(X,apply(Y,apply(p,nil)));
Z = p, ¬ pcapply(Y,apply(p,nil)),
                   ¬ pcapply(X,apply(Y,apply(p,nil)));
   no
```

### 5.4.2. Modo síntesis: formación de planes.

Muchos problemas en diferentes campos pueden formalizarse haciendo uso de un concepto de sistema que viene provisto de posibles acciones capaces de transformar el sistema produciendo una transición de estado. El problema de la formación de planes [Kowalski 79, Nilsson 82, Warren 74] se formula como la discrepancia, en un sistema dado, entre una situación de partida y otra que se pretende alcanzar. La solución al problema requiere la obtención de un plan que consiste, justamente, en la secuencia de acciones capaces de producir la transición requerida. La especificación de un problema de formación de planes incluye la descripción del estado actual (i.e. de la situación que se quiere alterar), la del estado objetivo y la del conjunto de operaciones capaces de transformar un estado. El problema de la generación de planes es un problema ampliamente tratado en disciplinas tales como la inteligencia artificial y podemos encontrar en libros sobre esta materia largas disertaciones que tratan su solución [Kowalski 79, Nilsson 82]. Más recientemente, el problema ha vuelto a merecer la atención en conexión con los intentos de encontrar una solución satisfactoria al problema de la negación constructiva [Chan Wallace 89].

Una consideración importante para el tratamiento de este problema está en la observación de que el estado del sistema, en un cierto momento de la vida del mismo, puede ser representado de dos formas diferentes:

- como el conjunto de hechos que son ciertos en dicho estado.
- como la secuencia de actualizaciones que se han debido producir para que, a partir del estado inicial (vacío), el sistema haya alcanzado la nueva configuración (i.e. para "construir" tal configuración).

La primera de las representaciones (conocida como *orientada a preguntas* [Veloso et al 81]), desemboca de forma directa en el problema de la regla marco o cómo tratar el hecho de que muchas de las propiedades que son ciertas en un estado determinado continúan siéndolo despúes de realizada una acción. Es conocido que en lógica de primer orden es posible definir un tratamiento satisfactorio de este problema, que pasa por introducir un (meta-)axioma marco para establecer que todas las propiedades (*P*) que no son invalidadas por la acción (*A*) continúan siendo ciertas en el nuevo estado (*holds(P,result(A,S)) :- holds(P,S), preserved(A,P)*) [Kowalski 79].

En este apartado abordamos el problema de la formación de planes para aplicaciones de bases de datos en el contexto de CLP(*H/E*) donde, de acuerdo a la

metodología desarrollada en las secciones precedentes, es la segunda aproximación (*orientada a actualizaciones* [Veloso et al 81]) la que se sigue. Veremos que en esta aproximación el problema se resuelve sin incorporar ninguna estrategia especial, explotando simplemente la capacidad inherente a la programación lógica de encontrar, para cada variable cuantificada existencialmente, una respuesta computada que expresa el valor de la misma para hacer la fórmula cierta. Para una comparación entre ambas aproximaciones en relación al problema de la generación automática de planes en un contexto de programación lógica con igualdad, consultar [Ramis 91]. En [Veloso Furtado 85, Furtado Moura 86] se trata también el problema de utilizar los axiomas que definen la base de datos para sintetizar las secuencias de acciones. Sin embargo, su enfoque se basa en la primera aproximación. Su formalización de cambio de estado describe la información borrada y añadida por la acción por medio de construcciones de metanivel como *holds* y *not_holds*. En [Costal 91] se presenta una aproximación *deductiva* [Olivé 86] al problema de la formación de planes, donde el axioma marco está implícito en las reglas de derivación que definen el valor de las informaciones en cada momento. La estrategia se basa en "compilar" la especificación a un modelo de eventos internos que permite deducir, en cada momento, las inserciones y borrados inducidos por la previa ocurrencia o ausencia de eventos externos.

Dado el estado final $T$ de una base de datos, la ejecución de la especificación en modo *síntesis* se entiende como la derivación de secuencias alternativas de actualizaciones que lleven a la base de datos desde un estado inicial $S$ hasta $T$. El estado final será (parcialmente) caracterizado mediante un conjunto de átomos que expresan las propiedades que deben cumplirse en $T$ y las que no. La derivación de las secuencias va a venir guiada por la relación de orden parcial entre símbolos de función establecida, implícitamente, por las precondiciones, que son manipuladas por técnicas de resolución de restricciones. Si un símbolo de función es candidato a participar en la secuencia requerida, sus precondiciones serán añadidas como restricciones al objetivo derivado, previa comprobación de su satisfacibilidad. Si resulta una inconsistencia el intérprete abstracto ensayará automáticamente y de forma no determinista otra cláusula.

Como ejemplo, consideraremos los siguientes objetivos:

• Obtener una secuencia $S$ conduciendo a un estado en el que la persona $p$ trabaja para la empresa $c$ y $c$ tiene vacantes:

```
:- □ worksfor(p,c,S), haspos(c,S).
% Respuestas:
```

```
S = hire(p,c,T), pchire(p,c,T), freepos(c,T)>s(0);
S = apply(Z,hire(p,c,U)), ¬ (p = Z), pcapply(Z,hire(p,c,U)),
       pchire(p,c,U), freepos(c,U) > s(0);
S = offer(c,N,hire(p,c,U)), pchire(p,c,U), N > 0; ...
```

• Obtener una secuencia $S$ conduciendo a un estado en el que la persona $p$ trabaja para la empresa $c$ y la persona $R$ es un candidato:

```
   :- ◻ worksfor(p,c,S), iscand(R,S).

   % Respuestas:

S = hire(p,c,apply(R,U)), pchire(p,c,apply(R,U)), ¬ (R = p),
      pcapply(R,U);
S = apply(R,hire(p,c,U)), pcapply(R,hire(p,c,U)), ¬ (R = p),
      pchire(p,c,U); ...
```

• Obtener una secuencia $S$ representando un estado en el que la persona $p$ es un candidato y el número de eventos en dicha secuencia es $< 4$:

```
   :- ◻ iscand(p,S), ne(S) < s(s(s(s(0)))).

   % Respuestas:

S = apply(p,T), pcapply(p,T);
S = apply(Z,apply(p,T)), ¬ (p = Z), pcapply(p,T),
      pcapply(Z,apply(p,T));
S = apply(Z,offer(E,N,apply(p,T))), ¬ (p = Z), pcapply(p,T),
      pcapply(Z,offer(E,N,apply(p,T)));
S = offer(E,N,apply(p,T)), pcapply(p,T);
S = offer(E,N,apply(Z,apply(p,T))), ¬ (p = Z), pcapply(p,T),
      pcapply(Z,apply(p,T));
S = offer(E,N,offer(F,M,apply(p,T))), pcapply(p,T);...
```

donde la función *ne(S)* se asume definida en la forma obvia [Choquet et al 86].

Se debe notar que la representación intensional, por medio de restricciones, del conjunto de respuestas a un objetivo tiene la ventaja de que, en general, distintas soluciones quedan subsumidas por una misma restricción respuesta. Por ejemplo, la primera restricción respuesta en el primer ejemplo

```
S = hire(p,c,T), pchire(p,c,T), freepos(c,T) > s(0)
```

subsume las secuencias

```
S = hire(p,c,offer(c,N,apply(p,nil))), N > s(0);
S = hire(p,c,apply(p,offer(c,N,nil))), N > s(0);
S = hire(p,c,apply(Q,offer(c,N,apply(p,nil)))), ¬ (Q = p),
       N > s(0);
S = hire(p,c,offer(c,N,apply(Q,apply(p,nil)))), ¬ (Q = p),
       N > s(0);
S = hire(p,c,hire(q,c,offer(c,N,apply(q,apply(p,nil))))),
       N > s(s(0)); ...
```

que, en el caso convencional, se obtendrían, por ejemplo, mediante un procedimiento de "narrowing" completo (incorporando estrategias adecuadas de control, como las propuestas en [Choquet et al 86] para SLOG) como solución al objetivo original

```
:- □ worksfor(p,c,S), haspos(c,S).
```

o, equivalentemente, por resolución del objetivo:

```
:- □ S = hire(p,c,T), pchire(p,c,T), freepos(c,T) > s(0).
```

Finalmente comentar que en el diseño de un sistema lógico ecuacional basado en las técnicas que presentamos (y específicamente concebido para soportar tareas de modelización conceptual) será conveniente incorporar una interfaz (*procesador de información*) que, a más alto nivel, se ocupe, e.g., de gestionar la ocurrencia de los eventos (comprobando, para cada requerimiento de actualización, el cumplimiento de las restricciones de integridad de la base) y de simplificar y ocultar las informaciones relativas a la satisfacción de las correspondientes precondiciones cuando éstas se cumplan.

# Conclusiones

En esta tesis se propone la idea de formalizar la integración de la programación lógica y ecuacional en un contexto más general de programación lógica con restricciones y se desarrolla dicha idea desde el establecimiento de los fundamentos teóricos hasta aspectos relacionados con el desarrollo y uso de un nuevo lenguaje de programación. La principal aportación del trabajo es la definición, en forma simple y rigurosa, de las semánticas declarativa y operacional del lenguaje, relacionadas por los esperados resultados formales que establecen la equivalencia entre ambas. Estos resultados extienden a un contexto lógico con igualdad las propiedades semánticas fundamentales de los programas lógicos tradicionales y establecen la plena validez de los mecanismos computacionales propuestos. La definición semántica declarativa del lenguaje se presenta en el estilo estándar: teoría de modelos y punto fijo. En nuestra aproximación, definimos modelos capaces de capturar el comportamiento operacional de los programas y de expresar, por tanto, diferentes propiedades observables de los mismos, como el conjunto de consecuencias lógicas atómicas o el conjunto de las respuestas calculadas para un objetivo dado. Desarrollamos, adicionalmente, diversas estrategias de optimización de los mecanismos operacionales propuestos para el lenguaje. Dichas estrategias reducen significativamente el espacio de búsqueda y, consecuentemente, el coste de verificar la satisfacibilidad de las restricciones. La principal contribución del trabajo, en este punto, es la definición de varias estrategias para obtener incrementalidad y simplificación en el proceso de cómputo, acompañadas de un estudio de sus propiedades formales. Presentamos una primera optimización que permite reutilizar sustituciones ya ensayadas para guiar, de forma heurística, la búsqueda. Formalizamos también una estrategia que simplifica las restricciones de manera incremental. Presentamos, por último, una estrategia que construye incrementalmente el árbol de búsqueda para las restricciones. Como una aplicación interesante de las diferentes técnicas desarrolladas caracterizamos, finalmente, una familia de aplicaciones a cuyas necesidades dichas técnicas dan respuesta. En nuestra propuesta, el esquema conceptual de la aplicación se describe como una teoría formal que se presenta como un programa en el lenguaje CLP($H$/$E$). Las definiciones semánticas declarativa y operacional del lenguaje dan así soporte formal a los aspectos de ejecutabilidad e interpretación de la teoría a la vez que proporcionan el soporte necesario para formalizar diferentes modos de entender la ejecución de estos programas.

# Conclusiones

Las líneas de trabajo abiertas por esta tesis son varias. Una de las más interesantes se basa en extender nuestra propuesta para incorporar desigualdades en el lenguaje de las restricciones (y en la teoría ecuacional), estudiando la posibilidad de interpretar la negación de manera constructiva e intentando formalizar estrategias para obtener incrementalidad y simplificación en este nuevo marco. Este tema es argumento de una tesis doctoral estrechamente relacionada con la nuestra y actualmente en desarrollo en nuestro grupo. Otra línea de interés se centra en la definición de estrategias para el aplazamiento del test de la solubilidad basadas en el uso de técnicas de propagación de soluciones. Este tema también será objeto de un próximo trabajo de investigación dentro del grupo.

# Referencias Bibliográficas.


**[Abramski Hankin 87] S. Abramski and C. Hankin**, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, Chichester, UK, 1987.

**[Agresti 86] W. Agresti**, editor. *New Paradigms for Software Development*. IEEE Computer Society Press, 1986.

**[Aiba et al 88] A. Aiba, K. Sakai, Y. Sato and D.J. Hawley.** Constraint Logic Programming Language CAL. In *Proc. Int'l Conf. on Fifth Generation Computer Systems,* pages 263-276. Ohmsha Publishers, Institute for new generation COmputer Technology ICOT, Tokyo, 1988.

**[Aït-Kaci Lincoln 88] H. Aït-Kaci and P. Lincoln.** LIFE, a Natural Language for Natural Language. MCC Technical Report ACA-ST-074-88, Microelectronics and Computer Technology Corporation, Austin, TX, 1988.

**[Aït-Kaci Nasr 86] H. Aït-Kaci and R. Nasr.** LOGIN: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:187-215, 1986.

**[Aït-Kaci Nasr 89] H. Aït-Kaci and R. Nasr.** Integrating Logic and Functional Programming. *LISP and Symbolic Computation*, 2(1):51-89, 1989.

**[Aït-Kaci Podelski 90] H. Aït-Kaci and A. Podelski.** The meaning of LIFE. PRL Research Report, Digital Equipment Corporation, Paris Research Laboratory, France, 1990. forthcoming.

**[Aït-Kaci et al 87] H. Aït-Kaci, P. Lincoln and R. Nasr.** Le Fun: Logic, equations and Functions. In *Proc. 1987 IEEE Int'l Symp. on Logic Programming SLP'87*, pages 17-23. IEEE Computer Society Press, 1987.

**[Albert Puget 91] P. Albert and J.F. Puget.** The language PECOS. In *Proc 11th Int'l Workshop on Expert Systems and their applications,* Avignon, May 1991.

**[Alpuente Falaschi 91] M. Alpuente and M. Falaschi.** Narrowing as an Incremental Constraint Satisfaction Algorithm. In *Proc. Third Int'l Symp. on Programming Language Implementation and Logic Programming PLILP'91,* Passau, Aug 1991. volume 528 of *Lecture Notes in Computer Science*, pages 111-122, Springer-Verlag, Berlin, 1991.

**[Alpuente Ramírez 90] M. Alpuente and M. J. Ramírez.** Rapid Prototyping of Database applications in the logic + equational EUROPA environment. In *Proc. 7th Int'l Conf. on Expert Systems, Theory and Applications,* pages 62-66, Los Angeles, Ca., Dec 1990.

**[Alpuente Ramírez 91] M. Alpuente and M. J. Ramírez.** An Equational Constraint Logic approach to Database Design. In *Actas I Jornadas sobre Programación Declarativa*, pages 260-281, Torremolinos, Oct 1991. submitted to *Database and Expert Systems Applications, DEXA'92*.

**[Alpuente et al 91] M. Alpuente, M. Falaschi and G. Levi.** Incremental Constraint Satisfaction for Equational Logic Programming. Technical Report TR-20-91, Dipartimento di Informatica, Università di Pisa, Oct 1991. To appear in *Theoretical Computer Science*. extended abstract in *Proc. Int'l Logic Programming Symp. ILPS'91 Workshop on Defeasible Reasoning and Constraint Solving*, pages 47-75, San Diego, Ca., Oct. 1991.

**[Apt van Emden 82] K. Apt and M.H. van Emden.** Contributions to the Theory of Logic Programming. *Journal of the ACM,* 29(3):841-862, 1982.

**[Apt 90] K.R. Apt.** Introduction to Logic Programming. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science*, volume B, pages 493-574. North-Holland, Amsterdam, 1990.

**[Bachmair et al 86] L. Bachmair, N. Dershowitz and J. Hsiang.** Orderings for equational proofs. In *Proc. First Symp. on Logic in Computer Science,* pages 346-357, Cambridge, Mass. IEEE Computer Society Pres, 1986.

**[Balzer 85] K. Balzer.** A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering,* volume SE-II (2):1257-1268, 1985.

**[Bandes 84] R.G. Bandes.** Constraining-Unification and the Programming Language Unicorn. In *Proc. 11th ACM Symp. on Principles of Programming Languages POPL'84,* Salt Lake City, 1984. Also in [DeGroot Lindstrom 86], pages 411-440.

**[Barbutti et al 84] R. Barbutti, M. Bellia, G. Levi and M. Martelli.** On the integration of logic programming and functional programming. In *Proc. 1984 IEEE Int'l Symp. on Logic Programming SLP'84,* pages 160-166. IEEE Computer Society Press, 1984.

**[Barbutti et al 86] R. Barbutti, M. Bellia, G. Levi and M. Martelli.** LEAF: A language which integrates Logic, Equations And Functions. In [DeGroot Lindstrom 86], pages 201-238.

**[Barendregt 84] H.P. Barendregt.** The Lambda-Calculus: Its Syntax and Semantics. revised edition. North-Holland, Amsterdam, 1984.

**[Barendregt 90] H.P. Barendregt.** Functional Programming and Lambda Calculus. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science*, volume B, pages 321-363. North-Holland, Amsterdam, 1990.

**[Beierle Pletat 87] C. Beierle and U. Pletat.** On the integration of Equality, Sorts and Logic Programming. In *Proc. Österreichische AI-Tagung ÖGAI,* IFB 151, pages 133-144, 1987.

**[Bellia Levi 86] M. Bellia and G. Levi.** The relation between logic and functional languages: a survey. *Journal of Logic Programming,* 3:217-236, 1986.

**[Bellia et al 87] M. Bellia, P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi.** A two-level approach to logic plus functional programming integration. In *Proc. Parallel Arquitectures and Languages Europe Conference PARLE'87,* volume 258 of *Lecture Notes in Computer Science,* pages 374-393. Springer-Verlag, Berlin, 1987.

**[Bergstra Klop 86] J.A. Bergstra and J.W. Klop.** Conditional Rewrite Rules: confluence and termination. *Journal of Computer and System Science*, 32:323-362, 1986.

**[Beringer Porcher 89] H. Beringer and F. Porcher.** A Relevant Scheme for Prolog Extensions: CLP (Conceptual Theory). In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming ICLP'89,* pages 131-148. The MIT Press, Cambridge, Mass., 1989.

**[Bert Echahed 86] D. Bert and R. Echahed.** Design and Implementation of a Generic and Functional programming Language. In *Proc ESOP'86,* volume 213 of *Lecture Notes in Computer science,* pages 119-132. Springer-Verlag, Berlin, 1986.

**[Bockmayr 86] A. Bockmayr.** Conditional rewriting and narrowing as a theoretical framework for logic-functional programming: a survey. Technical Report 10/86, Institut für Informatik, Universität Karlshruhe, 1986.

**[Borning 81] A. Borning.** The programming language aspects of Thing-Lab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems TOPLAS,* 3(4):252-387, 1981.

**[Borning et al 88] A. Borning, M. Maher, A. Martindale and M. Wilson.** Constraint Hierarchies and Logic Programming. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming ICLP'89,* pages 149-164. The MIT Press, Cambridge, Mass., 1989.

**[Bosco Giovannetti 86] P. Bosco and E. Giovannetti.** IDEAL: An Ideal DEductive Applicative Language. In *Proc. 1986 IEEE Int'l Symp. on Logic Programming SLP'86,* pages 89-94. IEEE Computer Society Press, 1986.

**[Bosco et al 87] P. Bosco, E. Giovannetti and C. Moiso.** Refined strategies for semantic unification. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'87,* volume 250 of *Lecture Notes in Computer Science*, pages 276-290. Springer-Verlag, Berlin, 1987.

**[Bosco et al 88] P. Bosco, E. Giovannetti and C. Moiso.** Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3-23. North-Holland, Amsterdam, 1988.

**[Bosco et al 91] P. Bosco, C. Cecchi, E. Giovannetti, C. Moiso and C. Palamidessi.** Using resolution for a sound and efficient implementation of logic and functional programming. In J. de Baker, editor, *Languages for Parallel Arquitectures: design, semantics, implementation and models,* pages 167-222. John Wiley and Sons, 1991.

**[Botella et al 88] P. Botella, X. Burgués and X. Franch.** Combining the imperative and the equational programming paradigms in the Excalibur system. In *Proc. 1st Int'l Workshop on Software Engineering and its Applications,* Tolouse, France, 1988.

**[Brand 75] D. Brand.** Proving theorems with the modification method. SIAM Journal of Computing, 4:412-430, 1975.

**[Bratko 86] I. Bratko.** PROLOG Programming for Artificial Intelligence. Addison Wesley. Reading, Mass., 1986.

**[Brodie et al 84] M.L. Brodie, J. Mylopoulous and J. Schmidt**, editors. *On Conceptual Modelling*. Springer-Verlag, New York, 1984.

**[Bruynooghe Pereira 84] M. Bruynooghe and L.M. Pereira.** Deduction revision by intelligent backtracking. In [Campbell 84], pages 194-215.

**[Bruynooghe 88] M. Bruynooghe.** Abstract Interpretation, 1988. *5th ICLP Tutorial Notes.*

**[Bubenko Olivé 86] J. Bubenko and A. Olivé.** Dynamic or Temporal modelling? An illustrative comparison. *SYSLAB Working paper* 117, Univ. of Stockholm, Sweden, 1986.

**[Buchberger 87] B. Buchberger.** History and Basic Features of the Critical-Pair/Completion Procedure. *Journal of Symbolic Computation,* 31(1&2):3-38, 1987. Preliminary version in *Proc. First Int'l Conf. on Rewriting Techniques and Applications RTA'85,* volume 202 of *Lecture Notes in Computer Science,* pages 1-45. Springer-Verlag, Berlin, 1985.

**[Bürckert 87] H-J. Bürckert.** Lazy E-unification - a method to delay alternetive solutions. Internal Report 87 R 34, Université de Nancy, France, 1987.

**[Bürckert 88] H-J. Bürckert.** Solving Disequations in Equational Theories. In *9th Int'l Conf. on Automated Deduction,* volume 310 of *Lecture Notes in Computer Science,* pages 517-527. Springer-Verlag, Berlin, 1988.

**[Bürckert et al 87] H-J. Bürckert, A. Herold and M. Schmidt-Schauβ.** On equational theories, unification and decidability. In *Proc. Second Int'l Conf. on Rewriting Techniques and Applications RTA'87,* volume 256 of *Lecture Notes in Computer Science,* pages 204-215. Springer-Verlag, Berlin, 1987.

**[Burstall Goguen 77] R. Burstall and J.A. Goguen.** Putting theories together to make specifications. In *Proc. Fifth Int'l Joint Conf. on Artificial Intelligence,* pages 1045-1058. Department of Computer Science, Carnegie-Mellon University, 1977.

**[Burstall Goguen 80] R. Burstall and J.A. Goguen.** The semantics of CLEAR: A specification language. In D. Bjørner, editor, *Proc. Copenhaguen Winter School on Abstract Software Specification,* volume 86 of *Lecture Notes in Computer Science,* pages 282-332. Springer-Verlag, Berlin, 1980.

**[Burstall et al 80] R. Burstall, D. MacQueen and D. Sanella.** HOPE: an experimental applicative language. In *Proc. First Int'l LISP Conf.,* pages 136-143. Stanford University, 1980.

**[Büttner et al 90] W. Büttner, K. Stenfeld, R. Schmid, H.-A. Schneider and E. Tidén.** Symbolic Constraint Handling Through Unification in Finite Algebras. In *Proc. Applicable Algebra in Engineering, Communication and Computing AAECC'90,* 1:97-118. Springer-Verlag, Berlin, 1990.

**[Campbell 84] J. Campbell**, editor. *Implementations of Prolog*. Ellis Horwood, Chichester, UK, 1984.

**[Ceri et al 89] S. Ceri, G. Gottlob and L. Tanca.** Logic Programming and Databases. Springer-Verlag, Berlin, 1989.

**[Clark 78] K.L. Clark.** Negation as Failure. In [Gallaire et al 78], pages 293-322.

**[Clark Gregory 86] K.L. Clark and S. Gregory.** PARLOG: A parallel logic programming language. *ACM Transactions on Programming Languages and Systems,* 8(1):1-49, 1986.

**[Clark Tärnlund 82] K.L. Clark and S.A. Tärnlund**, editors**.** *Logic Programm*ing. Academic Press, NY, 1982.

**[Cohen 86] S. Cohen.** The APPLOG language. In [DeGroot Lindstrom 86], pages 239-278.

**[Cohen 88] J. Cohen.** A view of the origins and development of Prolog. *Communications of the ACM,* 3(1):26-36, 1988.

**[Cohen 90] J. Cohen.** Constraint Logic Programming Languages. *Communications of the ACM*, 33(7):54-67, 1990.

**[Coleman et al 88] D. Coleman, C. Dollin, R. Gallimore, P. Arnold, and T. Rush.** An Introduction to the Axis specification language. Technical Report, HP Labs., Bristol, UK, 1988.

**[Colmerauer 82] A. Colmerauer.** Prolog and Infinite Trees. In [Clark Tärnlund 82], pages 231-251.

**[Colmerauer 84] A. Colmerauer.** Equations and Inequations on Finite and Infinite Trees. In *Proc. Int'l Conf. on Fifth Generation Computer Systems,* pages 85-99. ICOT, Tokyo, 1984.

**[Colmerauer 87] A. Colmerauer.** Opening the Prolog III Universe. *Byte Magazine*, special issue on Logic Programming, pages 177-182, Aug. 1987.

**[Colmerauer 90] A. Colmerauer.** An Introduction to Prolog III. *Communications of the ACM,* 33(7):69-90, 1990.

**[Comon Lescanne 89] H. Comon and P. Lescanne.** Equational problems and Disunification. *Journal of Symbolic Computation,* 7:371-425, 1989.

**[Costal 91] D. Costal.** An approach to Validation of Deductive Conceptual Models. In *Proc. 2nd Int'l Workshop on the Deductive Approach to Information Systems and Databases,* pages 50-72, Aiguablava (Catalonia), 1991.

**[Cox Pietrzykowski 85] P.T. Cox and T. Pietrzykowski.** Surface Deduction: a Uniform Mechanism for Logic Programming. In *Proc. 1985 IEEE Int'l Symp. on Logic programming SLP'85,* pages 220-227. IEEE Computer Society Press, 1985.

**[Chan Wallace 89] D. Chan and M. Wallace.** An Experiment with Programming Using Pure Negation. *In Proc. AISB'89,* Sussex, 1989.

**[Chang Lee 73] C.L. Chang and C.T. Lee.** Symbolic Logic and Mechanical Theorem Proving. Academic Press, NY, 1973.

**[Cheng et al 90] M. Cheng, M. van Emden and B. Richards.** On Warren's Method for Functional Programming in Logic. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming ICLP'90,* pages 546-560. The MIT Press, Cambridge, Mass., 1990.

**[Cheong Fribourg 91] P.H. Cheong and L. Fribourg.** Efficient Integration of Simplification into PROLOG. In *Proc. Third Int'l Symp. on Programming Language Implementation and Logic Programming PLILP'91,* Passau, Aug 1991. volume 528 of *Lecture Notes in Computer Science*, pages 359-370, Springer-Verlag, Berlin, 1991.

**[Choppy 87] C. Choppy.** Formal specification, prototyping and integration tests. Technical Report No. 377, LRI, Orsay, France, 1987.

**[Choppy Kaplan 90] C. Choppy and S. Kaplan.** Mixing abstract and concrete modules: specification, development and prototyping. In *Proc. 12th Int'l Conf. on Software Engineering,* Nice, France, 1990.

**[Choquet et al 86] N. Choquet, L. Fribourg and A. Mauboussin.** Runnable Protocol Specifications using the Logic Interpreter SLOG. In M. Díaz, editor, *IFIP'86,* pages 149-168. North-Holland, Amsterdam, 1986.

**[Darlington et al 86] J. Darlington, A.J. Field and H. Pull.** The unification of Functional and Logic Languages. In [DeGroot Lindstrom 86], pages 37-72.

**[Darlington et al 91] J. Darlington, Y.K. Guo and H. Pull.** Constraints unify functional and logic programming. Technical Report, Department of Computing, Imperial College, London, 1991.

**[DeGroot Lindstrom 86] D. DeGroot and G. Lindstrom**, editors. *Logic programming, Functions, Relations and Equations.* Prentice Hall, Englewood Cliffs, NJ, 1986.

**[Deransart 83] P. Deransart.** Derivation de programmes Prolog a partir de specifications algebriques. Technical Report, INRIA Laboria, France, 1983

**[Dershowitz 91] N. Dershowitz.** Canonical sets of Horn clauses. In J. Leach, B. Monien and M. R. Artalejo, editors, *Proc. 18th Int'l Colloquium on Automata, Languages and Programming ICALP'91,* volume 510 of *Lecture Notes in Computer Science.* pages 267-278, Springer-Verlag, Berlin, 1991.

**[Dershowitz Josephson 84] N. Dershowitz and A. Josephson.** Logic Programming by Completion. In *Proc. 2nd Int'l Conf. on Logic Programming, Uppsala, Sweden,* pages 313-320, 1984.

[**Dershowitz Jouannaud 90] N. Dershowitz and J.P. Jouannaud.** Rewrite Systems. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science*, volume B, pages 243-320. North-Holland, Amsterdam, 1990.

**[Dershowitz Plaisted 85] N. Dershowitz and A. Plaisted.** Logic Programming cum Applicative Programming. In *Proc. 1985 IEEE Int'l Symp. on Logic Programming SLP'85,* pages 54-66. IEEE Computer Society Press, 1985.

**[Dershowitz Plaisted 88]** **N. Dershowitz and A. Plaisted.** Equational Programming. *Machine Intelligence, 11,* pages 21-56. J.E. Hayes. D. Michie and J. Richards, editors, Clarendon Press, 1988.

**[Dershowitz Sivakumar 88]** **N. Dershowitz and G. Sivakumar.** Solving Goals in Equational Languages. In [Kaplan Jouannaud], pages 45-55.

**[Dershowitz et al 91]** **N. Dershowitz, S. Kaplan and D.A. Plaisted.** Rewrite, Rewrite, Rewrite, Rewrite, Rewrite, ....*Theoretical Computer Science*. 83:71-96, 1991.

**[Dincbas van Hentenryck 87]** **M. Dincbas and P. van Hentenryck.** Extended Unification Algorithms for the Integration of Functional Programming into Logic Programming. *Journal of Logic Programming*, 4:197-227, 1987.

**[Dosh et al 82]** **W. Dosh, G. Mascari and M. Wirsing.** On the Algebraic Specification of Databases. In *Proc. 8th Conf. on Very Large Data Bases VLDB ' 82,* Mexico City, 1982.

**[Echahed 88]** **R. Echahed.** On completeness of Narrowing Strategies. In *Proc. 13th Colloquium on Trees in Algebra and Programming CAAP'88,* volume 299 of *Lecture Notes in Computer Science,* pages 89-101. Springer-Verlag, Berlin, 1988.

**[Ehrich 86]** **H.D. Ehrich.** Key Extensions of Abstract Data Types, Final Algebras and Database Semantics. In D. Pitt et al, editors, *Proc. Workshop on Category Theory and Computer Programming*, pages 412-433. Springer-Verlag, Berlin, 1986.

**[Ehrich et al 78]** **H.D. Ehrich, H.J. Kreowski and H. Weber.** Algebraic Specification Scheme for Database Systems. In *Proc. 4th Conf. on Very Large Data Bases VLDB'78*, Berlin, 1978.

**[Ehrich et al 86]** **H.D. Ehrich, K. Drosten and M. Gogolla.** Towards an Algebraic Semantics for Database Specification, Knowledge and Data. In R. Meersan and D. Sernadas, editors, *Proc. IFIP WG 2.6 Working Conference on Database Semantics*, Albufeira, Portugal, 1986. North-Holland, Amsterdam, 1986.

**[Ehrig Mahr 85]** **H. Ehrig and B. Mahr.** Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, volume 6. Springer-Verlag, Berlin, 1985.

**[Ehrig Mahr 89]** **H. Ehrig and B. Mahr.** Fundamentals of Algebraic Specification 2: Module Specifications and Constraints. EATCS Monographs on Theoretical Computer Science, volume 21. Springer-Verlag, Berlin, 1989.

**[Ehrig Orejas 89]** **H. Ehrig and F. Orejas.** On recent Trends in Algebraic Specifications. Invited paper for the*8th Int'l Colloquium on Automata, Languages and Programming ICALP'89, Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1989.

**[Elcock 89l** **E.W. Elcock.** PROLOG: subsumption of the equality axioms by the homogeneous form. *Journal of Logic Programming,* 6(1&2):45-56, 1989.

**[Elcock 90l** **E.W. Elcock.** ABSYS: the first logic programming language - a retrospective and a commentary. *Journal of Logic Programming,* 9:1-7, 1990.

**[van Emden Kowalski 76]** **M. van Emden and R. Kowalski.** The semantics of predicate logic as a programming language. *Journal of the ACM,* 23(4):733-742, 1976.

**[van Emden Lloyd 84]** **M. van Emden and J.W. Lloyd.** A logical reconstruction of Prolog II. *Journal of Logic Programming,* 1(2):143-150, 1984.

**[van Emden Maibaum 81] M. van Emden and T. Maibaum.** Equations compared with clauses for specification of Abstract Data Types. In H. Gallier, J. Minker and J. Nicolas, editors, *Advances in Data Base Theory,* volume 1, pages 159-193. Plenum Press, NY, 1981.

**[van Emden Yukawa 87] M. van Emden and K. Yukawa.** Logic Programming with Equations. *Journal of Logic Programming,* 4:265-288, 1987.

**[Fages 84] F. Fages.** Associative-Commutative Unification. In *Proc. 7th Int'l Conf. on Automated Deduction,* volume 170 of *Lecture Notes in Computer Science,* pages 194-208. Springer-Verlag, Berlin, 1984. revised version in the *Journal of Symbolic Computation*, 3(3):257-275, 1987.

**[Fages Huet 83] F. Fages and G. Huet.** Unification and Matching in Equational Theories. In *Proc. Proc. 8th Colloquium on Trees in Algebra and Programming CAAP'83,* volume 159 of *Lecture Notes in Computer Science,* pages 205-220. Springer-Verlag, Berlin, 1983.

**[Falaschi et al 88] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi.** A new declarative semantics for logic languages. In R.A. Kowalski and K.A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming ICLP'88,* pages 993-1005. The MIT Press, Cambridge, Mass., 1988.

**[Falaschi et al 89] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi.** Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*. 69(3):289-318, 1989.

**[Falaschi et al 91] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi.** A Model-Theoretic reconstruction of the Operational Semantics of Logic programs. Technical Report TR 32/89, Dipartimento di Informatica, Università di Pisa, 1989. To appear in Information and Computation, 1991.

**[Fariñas 86] L. Fariñas del Cerro.** MOLOG: A system which extends PROLOG with modal logic. *New Generation Computing,* 4(1):35-50, 1986.

**[Fay 79] M. Fay.** First Order Unification in Equational Theories. In *Proc. 4th Int'l Conf. on Automated Deduction,* pages 161-167, Austin, TX, 1979.

**[Freeman-Benson et al 90] B.N. Freeman-Benson, J. Maloney and A. Borning.** An Incremental Constraint Solver. *Communications of the ACM,* 33(1):54-63, 1990.

**[Fribourg 84a] L. Fribourg.** A narrowing procedure for theories with constructors. In *Proc. 7th Int'l Conf. on Automated Deduction,* volume 170 of *Lecture Notes in Computer Science,* pages 259-301. Springer-Verlag, Berlin, 1984.

**[Fribourg 84b] L. Fribourg.** Oriented Equational Clauses as a Programming Language. *Journal of Logic Programming,* 1(2):165-177, 1984. Also in *Proc. 3rd Int'l Colloquium on Automata, Languages and Programming ICALP'84,* volume 172 of *Lecture Notes in Computer Science,* pages 162-173. Springer-Verlag, Berlin, 1984.

**[Fribourg 85a] L. Fribourg.** Handling Function Definitions Through Innermost Superposition and Rewriting. Technical Report 84-69, Institut de Programation, Université de Paris 7, France, 1984. Also in *Proc. First Int'l Conf. on Rewriting Techniques and Applications RTA'85,* volume 202 of *Lecture Notes in Computer Science,* pages 325-344. Springer-Verlag, Berlin, 1985.

**[Fribourg 85b] L. Fribourg.** Slog: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. 1985 IEEE Int'l Symp. on Logic Programming SLP'85*, pages 172-184. IEEE Computer Society Press, 1985.

**[Futatsugi et al 85] K. Futatsugi, J. A. Goguen, J.P. Jouannaud and J. Meseguer.** Principles of OBJ2. In B.K. Reid, editor, *Proc. 12th ACM Symp. on Principles of Programming Languages POLP'85,* pages 52-66. ACM, 1985.

**[Furbach Hölldobler 86] U. Furbach and S. Hölldobler.** Modelling the combination of Functional and Logic programming Languages. *Journal of Symbolic Computation,* 2:123-138, 1986.

**[Furbach et al 89] U. Furbach, S. Hölldobler and J. Schreiber.** Horn equality theories and paramodulation. *Journal of Automated Reasoning*, 5:309-337, 1989.

**[Furtado Moura 86] A. Furtado and C. Moura.** Expert Helpers to data-based Information Systems. In *Expert databases systems*. Benjamin-Cummings, 1986.

**[Furukawa 87] K. Furukawa.** Fifth Generation Computer Project: Current Research, Activity and Future Plans. In *Proc .Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'87,* volume 250 of *Lecture Notes in Computer Science,* pages 23-38. Springer-Verlag, Berlin, 1987.

**[Furukawa 91] K. Furukawa.** The Fifth Generation Computer Project: Towards Large-scale Knowledge Information Processing. In *Proc. Int'l Logic Programming Symp. ILPS'91.* The MIT Press, Cambridge, Mass., 1991.

**[Gabbrielli Levi 90] M. Gabbrielli and G. Levi.** Unfolding and fixpoint semantics of Concurrent Constraint Logic Programming. In H. Kirchner and W. Wechler, editors, *Proc. 2nd. Int'l Conf. on Algebraic and Logic Programming,* volume 463 of *Lecture Notes in Computer Science,* pages 204-216. Springer-Verlag, Berlin, 1990.

**[Gabbrielli Levi 91a] M. Gabbrielli and G. Levi.** Modeling answer constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming ICLP'91.* The MIT Press, Cambridge, Mass., 1991.

**[Gabbrielli Levi 91b] M. Gabbrielli and G. Levi.** On the semantics of Logic Programs. In J. Leach, B. Monien and M. R. Artalejo, editors, *Proc. 18th Int'l Colloquium on Automata, Languages and Programming ICALP'91,* volume 510 of *Lecture Notes in Computer Science.* pages 1-19, Springer-Verlag, Berlin, 1991.

**[Gallaire et al 78] H. Gallaire, J. Minker and J.M. Nicolas.** Logic and Databases. Plenum Press, NY, 1978.

**[Gallaire et al 84] H. Gallaire, J. Minker and J.M. Nicolas.** Advances in Database Theory. Plenum Press, NY, 1984.

**[Gallier 86] J.H. Gallier.** Logic for Computer Science. Harper & Row, NY, 1986.

**[Gallier Raatz 86] J.H. Gallier and S. Raatz.** SLD-resolution methods for Horn Clauses with Equality based on E-unification. In *Proc. 1986 IEEE Int'l Symp. on Logic Programming SLP'86,* pages 168-179. IEEE Computer Society Press, 1986.

**[Gallier Raatz 89] J.H. Gallier and S. Raatz.** Extending SLD resolution to equational Horn clauses using E-unification. *Journal of Logic Programming,* 6(1):3-44, 1989.

**[Gallier Snyder 87] J.H. Gallier and W. Snyder.** A general complete E-unification procedure. In *Proc. Second Int'l Conf. on Rewriting Techniques and Applications RTA'87*, volume 256 of *Lecture Notes in Computer Science*, pages 129-203. Springer-Verlag, Berlin, 1987. see also *Complete Sets of Transformations for General E-unification. Theoretical Computer Science,* 67:203-260, 1989.

**[Geser Hussmann 86] A. Geser and H. Hussmann.** Experiences with the RAP system- a specification interpreter combining term rewriting and resolution. In *Proc. ESOP'86,* volume 213 of *Lecture Notes in Computer science,* pages 339-350. Springer-Verlag, Berlin, 1986.

**[Geser et al 88] A. Geser, H. Hussmann, A. Mück.** A compiler for a class of conditional rewrite systems. In [Kaplan Jouannaud 88], pages 84-90.

**[Giovannetti Moiso 86] E. Giovannetti and C. Moiso.** A completeness result for E-unification algorithms based on Conditional Narrowing. In M. Boscarol, L. Carlucci and G. Levi, editors, *Foundations of Logic and Functional Programming,* volume 306 of *Lecture Notes in Computer Science*, pages 157-167. Springer-Verlag, Berlin, 1986.

**[Giovannetti et al 91] E. Giovannetti, G. Levi, C. Moiso and C. Palamidessi.** Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Science*, 42, 1991.

**[Gogolla 89] M. Gogolla.** Algebraization and Integrity constraints for an Extended Entity-Relationship Approach. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'89,* volume 351 of *Lecture Notes in Computer Science*, pages 259-273. Springer-Verlag, Berlin, 1989.

**[Goguen 80] J.A. Goguen.** How to prove Algebraic Inductive Hypothesis without Induction. In *Proc. 5th Int'l Conf. on Automated Deduction,* pages 356-373, volume 87 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

**[Goguen 86] J.A. Goguen.** One, none, a hundred thousand specification languages. In H.-J. Kugler, editor, *Proc. IFIP'86,* pages 995-1004. North-Holland, Amsterdam, 1986.

**[Goguen Burstall 84] J.A. Goguen and R. Burstall.** Introducing Institutions. In E. Clarke and D. Kozen, editors, *Proc. Logics of Programming Workshop,* volume 164 of *Lecture Notes in Computer Science,* pages 221-256. Springer-Verlag, Berlin, 1984.

**[Goguen Meseguer 82] J.A. Goguen and J. Meseguer.** Rapid prototyping in the OBJ Executable Specification Language. In Proc. ACM SIGSOFT Rapid Prototyping Workshop. *ACM Software Engineering Notes*, 7(5):75-84, 1982.

**[Goguen Meseguer 85] J.A. Goguen and J. Meseguer.** Semantics of Computation: Initial Algebra Semantics and Programming Language Paradigms. Technical Report, SRI International, Menlo Park, Ca., 1986.

**[Goguen Meseguer 86] J.A. Goguen and J. Meseguer.** Eqlog: equality, types and generic modules for logic programming. In [DeGroot Lindstrom 86], pages 295-364. An earlier version appears in the *Journal of Logic programming,* 1(2):179-210, 1984.

**[Goguen Meseguer 87] J.A. Goguen and J. Meseguer.** Models and Equality for Logic programming. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'87,* volume 250 of *Lecture Notes in Computer Science,* pages 1-22. Springer-Verlag, Berlin, 1987.

**[Goguen Meseguer 88] J.A. Goguen and J. Meseguer.** Software for the Rewrite Rule Machine. In *Proc. Int'l Conf. on Fifth Generation Computer Systems,* pages 628-637. Ohmsha Publishers, ICOT, Tokyo, 1988.

**[Goguen Tardo 79] J.A. Goguen and J. Tardo.** An Introduction to OBJ: A language for writing and Testing Software Specifications. In *Specifications of Reliable Software,* pages 170-189. IEEE Computer Society Press, 1979. reprinted in N. Gehani and A.D. McGettrick, *Software Specification Tecniques,* pages 391-420. Addison Wesley. Reading, Mass., 1985.

**[Goguen Winkler 88] J.A. Goguen and T. Winkler.** Introducing OBJ3. Tecnical Report SRI-CSL-88-9, SRI International, Menlo Park, Ca., 1988.

**[Goguen et al 77] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright.** Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM,* 24(1):68-95, 1977.

**[Goguen et al 78] J.A. Goguen, J.W. Thatcher and E.G. Wagner.** An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, pages 80-149. Prentice Hall, Englewood Cliffs, NJ, 1978.

**[González et al 90] J.C. González, M.T. Hortalá and M. Rodríguez-Artalejo.** A Functional Logic Language with Higher Order Logic Variables. Technical Report DIA 90/6, Oct 1990.

**[Guo et al 90] Y. Guo, H. Lock, J. Darlington and R. Dietrich.** A classification for the integration of functional and logic languages. Technical Report, Department of Computer Science, Imperial College and GMD Karlshruhe, 1990. deliverable for the ESPRIT Basic Research Action No. 3147.

**[Gustafsson et al 82] M. Gustafsson, T. Karlsson and J. Bubenko.** A declarative approach to Conceptual Information Modelling. In T.W. Olle et al., editors, *Information Systems Design methodologies: A comparative review*, pages 93-142. North-Holland, Amsterdam, 1982.

**[Guttag Horning 80] J.V. Guttag and J. Horning.** Formal specification as a Design Tool. In *Proc. 7th ACM Symp. on Principles of Programming Languages POPL'80*, pages 251-261. ACM, 1980.

**[Hao Chabrier 90] J.K. Hao and J.-J. Chabrier.** A modular Arquitecture for Constraint Logic Programming. *Proc. 3rd Int'l Workshop on Software Engineering and its Applications*. Tolouse, France, 1990.

**[Hanus 90a] M. Hanus.** Compiling Logic Programs with Equality. In *Proc. 2nd Int'l Symp. on Programming Language Implementation and Logic Programming PLILP'90,* Linköping, Aug 1990. volume 456 of *Lecture Notes in Computer Science*, pages 387-401, Springer-Verlag, Berlin, 1990.

**[Hanus 90b] M. Hanus.** Logic Programs with Equational Type Specifications. In H. Kirchner and W. Wechler, editors, *Proc. 2nd. Int'l Conf. on Algebraic and Logic Programming,* volume 463 of *Lecture Notes in Computer Science,* pages 70-85. Springer-Verlag, Berlin, 1990.

**[Henderson 86] P. Henderson.** Functional Programming, formal specification and rapid prototyping. *IEEE Transactions on Software Engineering,* 12(2):241-250, 1986.

**[van Hentenryck 89] P. van Hentenryck.** Constraint Satisfaction in Logic Programming. Logic Progamming Series, The MIT Press, Cambridge, Mass., 1989.

**[van Hentenryck 90] P. van Hentenryck.** Incremental Constraint Satisfaction in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming ICLP'90,* pages 189-202. The MIT Press, Cambridge, Mass., 1990.

**[van Hentenryck 91] P. van Hentenryck and Y. Deville.** Operational Semantics of Constraint Logic Programming over Finite Domains. In *Proc. Third Int'l Symp. on Programming Language Implementation and Logic Programming PLILP'91,* Passau, Aug 1991. volume 528 of *Lecture Notes in Computer Science*. pages 395-406, Springer-Verlag, Berlin, 1991.

**[Herbrand 71] J. Herbrand.** Logical writings. In W.D. Goldfarb, editor, D. Reidel Publishing Co., Dordrecht, 1971.

**[Hermenegildo 86] M.V. Hermenegildo.** An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In E.Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming ICLP'86,* volume 225 of *Lecture Notes in Computer Science,* pages 25-40. Springer-Verlag, Berlin, 1986.

**[Herold 86] A. Herold.** Narrowing Techniques Applied to Idempotent Unification. SEKI Report SR-86-16, FB Informatik, Universität KaisersLautern, Germany, 1986.

**[Hill 74] R. Hill.** LUSH-resolution and its completeness. DCI Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

**[Hoddinot Elcock 86] P. Hoddinot and E.W. Elcock.** PROLOG: subsumption of Equality axioms by the homogeneous form. In *Proc. 1986 IEEE Int'l Symp. on Logic Programming SLP'86*, pages 115-126. IEEE Computer Society Press, 1986.

**[Höhfeld Smolka 88] M. Höhfeld and G. Smolka.** Definite Relations over Constraint Languages. LILOG Report 53, IWBS, IBM Deutschland GmbH, Stuttgart, Germany, 1988. to appear in the *Journal of Logic Programming*.

**[Hoffmann O'Donnell 82] C.M. Hoffmann and M.J. O'Donnell.** Programming with Equations. *ACM Transactions on Programming Languages and Systems TOPLAS,* 1(4):83-112, 1982.

**[Hogger 84] C.J. Hogger.** Introduction to Logic Programming. Academic Press, NY, 1984.

**[Hölldobler 87] S. Hölldobler.** Equational Logic Programming. In *Proc. 1987 IEEE Int'l Symp. on Logic Programming SLP´87*, pages 335-346. IEEE Computer Society Press, 1987.

**[Hölldobler 88] S. Hölldobler.** From Paramodulation to Narrowing. In R.A. Kowalski and K.A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming ICLP'88,* pages 327-342. The MIT Press, Cambridge, Mass., 1988.

**[Hölldobler 89] S. Hölldobler.** Foundations of Equational Logic Programming, volume 353 of *Lecture Notes in Artificial Intelligence,* subseries of Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1989.

**[Hölldobler 90] S. Hölldobler.** Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75:85-110, 1990.

**[Hudak 89] P. Hudak.** Conception, Evolution and Application of Functional Programming Languages, *ACM Computing Surveys,* 21(3):359-411, 1989.

**[Huet 77] G. Huet.** Deduction and Computation. In *Proc. 18th IEEE Symp. on Foundations of Computer Science,* pages 30-45. IEEE Computer Society Press, 1977.

**[Huet 80] G. Huet.** Confluent reductions: Abstract Properties and Applications to Term Rewriting. *Journal of the ACM,* 27:797-821, 1980.

**[Huet Hullot 82] G. Huet and J.M. Hullot.** Proofs by induction in Equational Theories with constructors. *Journal of Computer and System Science,* 25(2):239-266, 1982.

**[Huet Levy 79] G. Huet and J.J. Levy.** Computing in nonambiguous Linear Rewriting Systems. Technical Report 359, INRIA Laboria, France, 1979.

**[Huet Oppen 80] G. Huet and D. C. Oppen.** Equations and Rewrite Rules: A Survey. In R. V. Book, editor, *Formal Languages, Perspectives and Open problems,* pages 349-405. Academic Press, NY, 1980.

**[Hullot 80] J.M. Hullot.** Canonical Forms and Unification. In *Proc. 5h Int'l Conf. on Automated Deduction,* volume 87 of *Lecture Notes in Computer Science*, pages 318-334. Springer-Verlag, Berlin, 1980.

**[Hussmann 86] H. Hussmann.** Unification in conditional-equational theories. Technical Report MIP-8502, Fakultät für Mathematik und Informatik, Universität Passau, 1986. Also in *Proc. European Conf. on Computer ALgebra EUROCAL'85,* volume 204 of *Lecture Notes in Computer Science,* pages 543-553. Springer-Verlag, Berlin, 1986.

**[Jaffar Lassez 86] J. Jaffar and J.-L. Lassez.** Constraint Logic Programming. Technical Report, Department of Computer Science, Monash University, 1986.

**[Jaffar Lassez 87] J. Jaffar and J.-L. Lassez.** Constraint Logic Programming. In P*roc. Fourteenth Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages POPL'87*, pages 111-119. ACM, 1987.

**[Jaffar Michaylov 87] J. Jaffar and S. Michaylov.** Methodology and Implementation of a CLP System. In J.-L. Lassez, editor, *Proc. Fourth Int'l Conf. on Logic Programming ICLP'87,* Melbourne, Australia, pages 196-218. The MIT Press, Cambridge, Mass., 1987.

**[Jaffar Stuckey 86a] J. Jaffar and P.J. Stuckey.** Logic Program Semantics for Programming with Equations. In E.Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming ICLP'86,* volume 225 of *Lecture Notes in Computer Science,* pages 313-326. Springer-Verlag, Berlin, 1986.

**[Jaffar Stuckey 86b] J. Jaffar and P.J. Stuckey.** Semantics of infinite tree logic programming. *Theoretical computer Science,* 46:141-158, 1986.

**[Jaffar et al 84a] J. Jaffar, J.-L. Lassez and M. Maher.** A theory of complete logic programs with equality. *Journal of Logic Programming*, 3:211-223, 1984.

**[Jaffar et al 84b] J. Jaffar, J.-L. Lassez and M. Maher.** A logical foundation for Prolog II Technical Report 44, Department of Computer Science, Monash University, 1984.

**[Jaffar et al 86a] J. Jaffar, J.-L. Lassez and M. Maher.** Logic programming language scheme. En [DeGroot Lindstrom 86], pages 441-468.

**[Jaffar et al 86b] J. Jaffar, J.-L. Lassez and M. Maher.** PROLOG II as an instance of the logic programming language scheme. In M. Wirsing, editor, *Proc. of the IFIP Conf. on Formal Description of Programming Concepts*. North-Holland, Amsterdam, 1986.

**[Jaffar et al 86c] J. Jaffar, J.-L. Lassez and M. Maher.** Some issues and trends in the semantics of logic programming. In E.Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming ICLP'86,* volume 225 of *Lecture Notes in Computer Science,* pages 223-241. Springer-Verlag, Berlin, 1986.

**[Jaffar et al 88] J. Jaffar, S. Michaylov, P.J. Stuckey and H.C. Yap.** The CLP(←) language and system. Technical Report, IBM T.J. Watson Research Center, NY, 1988.

**[Josephson Dershowitz 86] A. Josephson and N. Dershowitz.** An implementation of narrowing: the RITE way. In *Proc. 1986 IEEE Int'l Symp. on Logic Programming SLP'86*, pages 187-197. IEEE Computer Society Press, 1986. revised version in the *Journal of Logic Programming,* 6(1&2):57-77, 1989.

**[Jouannaud Kirchner 86] J. Jouannaud and C. Kirchner.** Completion of a set of rules modulo a set of equations. SIAM Journal of Computing, 15(4):1155-1194, 1986. Preliminary version in *Proc. 11th ACM Symp. on Principles of Programming Languages POPL'84,* Salt Lake City, ACM 1984.

**[Jouannaud Kounalis 89] J. Jouannaud and E. Kounalis.** Automatic proofs in equational theories without constructors. *Information and Computation,* 82(1):1-30, 1989.

**[Jouannaud Lescanne 87] J. Jouannaud and P. Lescanne.** Rewriting Systems, *TSI,* 6(3):181-199, 1987. translation from french "La réécriture",*TSI,* 5(6):433-452, 1986.

**[Kaplan 84] S. Kaplan.** Conditional Rewrite Rules. *Theoretical Computer Science*, 33:175-193, 1984.

**[Kaplan 86] S. Kaplan.** Fair conditional term rewriting systems: unification, termination and confluence. In H.-J. Kreowski, editor, *Recent Trends in Data Type Specification*, volume 116 of Informatik-Fachberichte, pages 136-155. Springer-Verlag, Berlin, 1986.

**[Kaplan 87] S. Kaplan.** A compiler for term rewriting systems. In *Proc. Second Int'l Conf. on Rewriting Techniques and Applications RTA'87,* volume 256 of *Lecture Notes in Computer Science,* pages 25-41. Springer-Verlag, Berlin, 1987.

**[Kaplan Jouannaud 88] S. Kaplan and J. Jouannaud**, editors. *Proc. First Int'l Workshop on Conditional Term Rewriting,* Orsay, France,1987, volume 308 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

**[Kirchner 84] C. Kirchner.** A New Equational Unification Method: A generalization of Martelli-Montanari's Algorithm. In *Proc. 7th. Int'l Conf. on Automated Deduction,* volume 170 of *Lecture Notes in Computer Science,* pages 224-247. Springer-Verlag, Berlin, 1984.

**[Kirchner Lescanne 87] C. Kirchner and P. Lescanne.** Solving Disequations. In *Proc. Second Symp. on Logic in Computer Science,* pages 347-352. IEEE Computer Society Press, 1987.

**[Kirchner et al 88] C. Kirchner, H. Kirchner and J. Meseguer.** Operational Semantics of OBJ3. In *Proc. 9th Int'l Colloquium on Automata, Languages and Programming ICALP'88,* volume 241 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1988.

**[Kirchner et al 91] C. Kirchner, H. Kirchner and M. Rusinowitz.** Deduction with Symbolic Constraints. Tecnical Report, Centre de Recherche en Informatique de Nancy CRIN, France, 1991. Preliminary version in *Revue Française d'Intelligence Artificielle,* 4(3):9-52, 1990.

**[Knight 89] K. Knight.** Unification: A Multidisciplinary Survey. *ACM Computing Surveys,* 21(1):93-124, 1989.

**[Knuth Bendix 70] D. Knuth and P. Bendix.** Simple Word Problems in Universal Algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra,* pages 263-297. Pergamon Press, 1970.

**[Komorowski 82] H.J. Komorowski.** QLOG-The programming environment for Prolog in LISP. In [Clark Tärnlund 82], pages 315-322.

**[Kornfeld 86] W.A. Kornfeld.** Equality for Prolog. In [DeGroot Lindstrom 86], pages 279-294. Preliminary version in *Proc. 8th. Int'l Joint Conf. on Artificial Intelligence IJCAI'83,* pages 514-519, 1983.

**[Kounalis Rusinowitz 88] E. Kounalis and M. Rusinowitz.** On Word Problems in Horn Theories. In [Kaplan Jouannaud 88], pages 144-160.

**[Kowalski 70] R. Kowalski.** The Case for Using Equality Axioms in Automated Deduction. In M. Laudet et al., editors, *Proc. Symp. on Automatic Demonstration,* volume 125 of *Lecture Notes in Mathematics,* pages 112-127. Springer-Verlag, Berlin 1970.

**[Kowalski 74] R. Kowalski.** Predicate Logic as a Programming Language. In *Proc. IFIP'74,* pages 569-574. North-Holland, Amsterdam, 1974.

**[Kowalski 79] R. Kowalski.** Logic for Problem Solving. North-Holland, Amsterdam, 1979.

**[Kowalski 81] R. Kowalski.** Logic as a Database Language. In *Proc. Advanced Seminar on Theoretical Issues of Databases,* Cetrano, Italy, 1981.

**[Kowalski Kuehner 71] R. Kowalski and D. Kuehner.** Linear resolution with selection function. *Artificial Intelligence,* 2:227-260, 1971.

**[Krischer Bockmayr 91] S. Krischer and A. Bockmayr.** Detecting Redundant Narrowing Derivations by the LSE-SL Reducibility Test. In *Proc Forth Int'l Conf. on Rewriting Techniques and Applications RTA'91*. Springer-Verlag, Berlin, 1991. to appear.

**[Lankford 75] D.S. Lankford.** Canonical Inference. Technical Report ATP-32, Department of Mathematics, Southwestern University, Georgetown, TX, 1975.

**[Lassez 87] C. Lassez.** Constraint Logic programming. *Byte Magazine*, special issue on Logic Programming, pages 171-176, Aug.1987.

**[Lassez McAloon 88] J.-L. Lassez and K. McAloon.** Applications of a Canonical Form for Generalized Linear Constraints. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 703-710. Ohmsha Publishers, ICOT, Tokyo, 1988.

**[Lassez McAloon 89] J.-L. Lassez and K. McAloon.** Independence of Negative Constraints. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'89,* volume 351 of *Lecture Notes in Computer Science*, pages 19-27. Springer-Verlag, Berlin, 1989.

**[Lassez et al 88] J.-L. Lassez, M.J. Maher and K. Marriot.** Unification Revisited. In [Minker 88], pages 587-625.

**[Leconte Lelong 91] M. Leconte and V. Lelong.** Programation par contraintes. Génération CHARME. In *Proc 11th Int'l Workshop on Expert Systems and their applications,* Avignon, May 1991.

**[Leler 87] W. Leler.** Constraint Programming Languages. Addison Wesley. Reading, Mass., 1987.

**[Levi 86] G. Levi.** New research directions in logic specification languages. In H.-J. Kugler, editor, *Proc. IFIP'86,* pages 1005-1008. North-Holland, Amsterdam, 1986.

**[Levi et al 87] G. Levi, C. Palamidessi, P. Bosco, E. Giovanetti and C. Moiso.** A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proc. 1987 IEEE Int'l Symp. on Logic Programming SLP'87,* pages 318-327. IEEE Computer Society Press, 1987.

**[Lloyd 87] J.W. Lloyd.** Foundations of logic programming. Springer-Verlag, Berlin, 1987. second edition.

**[Maher 88] M. Maher.** Equivalences of Logic Programs. In [Minker 88], pages 627-658.

**[Maher 91] M. Maher.** Personal communication. Mar 1991. unpublished.

**[Maher Stuckey 89] M. Maher and P.J.Stukey.** Expanding Query Power in Constraint Logic Programming Languages. Technical Report RC 15063, IBM T. J. Watson Research Center, NY, 1989. Also in *Proc. of the North-American Conf. on Logic Programming NACLP'89,* Ohio, 1989. The MIT Press, Cambridge, Mass., 1989.

**[Mahr Makowsky 84] B. Mahr and J. Makowsky.** Characterizing specification languages which admit initial semantics. *Theoretical Computer Science,* 31:49-54, 1984.

**[Maier Warren 87] D. Maier and D.S. Warren.** Computing with Logic. Benjamin Cummings, 1987.

**[Makowsky 85] J.A. Makowsky.** Why Horn Formulas Matter in Computer Science: Initial Structures and Generic Examples. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development TAPSOFT'85,* volume 185 of *Lecture Notes in Computer Science*, pages 374-387. Springer-Verlag, Berlin, 1985.

**[Manes Arbib 86] E.G. Manes and M.A. Arbib.** Algebraic Approaches to Program Semantics. AKM Series in Theoretical Computer Science. Springer-Verlag, Berlin, 1986.

**[Manna Shamir 77] Z. Manna and A. Shamir.** The Optimal Approach to recursive Programs. *Communications of the ACM,* 20:824-831, 1977.

**[Marriot Sondergaard 89] K. Marriot and H. Sondergaard.** Abstract Interpretation, 1989. *1989 SLP Tutorial Notes.*

**[Martelli Montanari 82] A. Martelli and U. Montanari.** An efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems TOPLAS,* 4(2):258-282, 1982.

**[Martelli et al 86] A. Martelli, C. Moiso and G.F. Rossi.** An algorithm for unification in equational theories. In *Proc. 1986 IEEE Int'l Symp. on Logic Programming SLP'86*, pages 180-186. IEEE Computer Society Press, 1986.

**[Mellish Hardy 84] C. Mellish and S. Hardy.** Integrating PROLOG in the POPLOG environement. In [Campbell 84], pages 147-162.

**[Meseguer 89] J. Meseguer.** General Logics. In H.D. Ebbinghaus et al., editors, *Proc. Granada Logic Colloquium'87* pages 275-329. North-Holland, Amsterdam, 1989.

**[Meseguer 91] J. Meseguer.** Conditional Rewriting Logic as a Unified Model of Concurrency. Technical Report SRI-CSL-91-05, SRI International, Menlo Park, Ca., 1991. to appear in *Theoretical Computer Science.*

**[Meseguer Goguen 85] J. Meseguer and J.A. Goguen.** Initiality, Induction and Computability. In J. Reynolds and M. Nivat, editors, *Algebraic Methods in Semantics,* pages 459-541. Cambridge University Press, Cambridge, 1985.

**[Middeldorp 91a] A. Middeldorp.** Personal communication. Jul 1991. unpublished.

**[Middeldorp 91b] A. Middeldorp.** A survey on Term Rewriting Systems. Technical Report, Centre voor Wiskunde en Informatica CWI, Amsterdam, 1991. in preparation.

**[Milner et al 90] R. Milner, M. Tofte and R. Harper.** The definition of Standard ML. The MIT Press, Cambridge, Mass., 1990.

**[Miller Nadathur 86] D. A. Miller and G. Nadathur.** Higher order Logic Programming. In E.Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming ICLP'86,* volume 225 of *Lecture Notes in Computer Science,* pages 448-462. Springer-Verlag, Berlin, 1986.

**[Minker 88] J. Minker**, editor. Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, Los Altos, Ca., 1988.

**[Mishra 84] P. Mishra.** Towards a theory of types in Prolog. In *Proc. 1984 IEEE Int'l Symp. on Logic Programming SLP'84*, pages 289-298. IEEE Computer Society Press, 1984.

**[Moreno 89] J.J. Moreno.** Babel: Diseño, semántica e implementación de un lenguaje que integra la programación funcional y lógica. Ph. D. Thesis, Facultad de Informática de Madrid, Spain, 1989.

**[Moreno Rodríguez 88] J.J. Moreno and M. Rodriguez-Artalejo.** BABEL: A Functional and Logic Programming Language based on a constructor discipline and narrowing. In I. Grabowski, P. Lescanne and W. Wechler, editors, *Algebraic and Logic Programming*, volume 343 of *Lecture Notes in Computer Science*, pages 223-232. Springer-Verlag, Berlin, 1988.

**[Moreno Rodríguez 89] J.J. Moreno and M. Rodriguez-Artalejo.** Logic Programming with Functions and Predicates: The Language BABEL. Technical Report DIA/89/3, Departamento de Informática y Automática, UCM, Madrid, 1989. to appear in the *Journal of Logic Programming*,

**[Moreno et al 90] J.J. Moreno, H. Kuchen, R. Loogen and M. Rodríguez-Artalejo.** Lazy Narrowing in a graph machine. In H. Kirchner and W. Wechler, editors, *Proc. 2nd. Int'l Conf. on Algebraic and Logic*

*Programming,* volume 463 of *Lecture Notes in Computer Science,* pages 298-317. Springer-Verlag, Berlin, 1990.

**[Mosses 90] P.D. Mosses.** Denotational Semantics. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science*, volume B, pages 575-631. North-Holland, Amsterdam, 1990.

**[Mycroft O'Keefe 84] A. Mycroft and R.A. O'Keefe.** A Polymorphic Type System for Prolog. *Artificial Intelligence,* 23:295-307, 1984.

**[Naish 86] L. Naish.** Negation and Control in Prolog, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1986.

**[Nilsson 82] N. Nilsson.** Principles of Artificial Intelligence. Springer-Verlag, Berlin, 1982.

**[Nutt et al 89] W. Nutt, P. Réty and G. Smolka.** Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295-317, 1989.

**[O'Donnell 77] M. O'Donnell.** Computing in systems described by equations, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.

**[O'Donnell 85] M. O'Donnell.** Equational Logic as a Programming Language. The MIT Press, Cambridge, Mass., 1985.

**[Olivé 86]** A. Olivé. A comparison of the Operational and Deductive approaches to Conceptual Systems Modelling. In H.-J. Kugler, editor, *Proc. IFIP'86*, pages 91-96. North-Holland, Amsterdam, 1986.

**[Padawitz 87] P. Padawitz.** Strategy-Controlled Reduction and Narrowing. In *Proc. Second Int'l Conf. on Rewriting Techniques and Applications RTA'87,* volume 256 of *Lecture Notes in Computer Science,* pages 242-255. Springer-Verlag, Berlin, 1987.

**[Padawitz 88] P. Padawitz.** Computing in Horn Clause Theories. EATCS Monographs on Theoretical Computer Science, volume 16. Springer-Verlag, Berlin, 1988.

**[Palamidessi 88] C. Palamidessi.** Strutture ad Ordinamenti Parziali nella semantica dichiarativa dei linguaggi logici. Ph. D. Thesis, Università de Pisa, Italy, 1988.

**[Paterson Wegman 78] M. Paterson and M. Wegman.** Linear Unification. *Journal of Computer and System Science,* 16(2):158-167, 1978.

**[Paul 84] E. Paul.** A New Interpretation of the Resolution Principle. In *Proc. 7th Int'l Conf. on Automated Deduction,* volume 170 of *Lecture Notes in Computer Science,* pages 333-355. Springer-Verlag, Berlin, 1984.

**[Paul 86] E. Paul.** On solving the equality problem in theories defined by Horn Clauses. *Theoretical Computer Science,* 44:127-153, 1986.

**[Peyton-Jones 87] S. Peyton-Jones.** The Implementation of Functional Programming Languages. Prentice Hall, Englewood Cliffs, NJ, 1987.

**[Plotkin 72] G.D. Plotkin.** Building in Equational Theories. *Machine Intelligence,* 7, pages 73-90. Metzer and Michie, editors, 1972.

**[Plotkin 81] G.D. Plotkin.** A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

**[Pull 90] H. M. Pull.** Equation Solving in lazy Functional Languages. Ph. D. Thesis, Imperial College, London, 1990.

**[Ramis 91] M.J. Ramis.** El modo generador en el prototipado automático de bases de datos en el entorno lógico-ecuacional EUROPA. Proyecto fin de carrera. Facultad de Informática, Universidad Politécnica de Valencia, Jul 1991.

**[Reddy 85] U.S. Reddy.** Narrowing as the Operational Semantics of Functional Languages. In *Proc. 1985 IEEE Int'l Symp. on Logic Programming SLP'85*, pages 138-151. IEEE Computer Society Press, 1985.

**[Reddy 86] U.S. Reddy.** On the relationship between logic and functional languages. In [DeGroot Lindstrom 86], pages 3-36.

**[Reddy 87] U.S. Reddy.** Functional Logic Languages, Part I. In J.H. Fasel and R.M. Keller, editors, *Proc. of a Workshop on Graph Reduction,* volume 279 of *Lecture Notes in Computer Science,* pages 401-425. Springer-Verlag, Berlin, 1987.

**[Reichel 87] H. Reichel.** Initial Computability, Algebraic Specifications and Partial Algebras. Clarendon Press, Oxford, 1988.

**[Reiter 84] R. Reiter.** Towards a logical reconstruction of Relational Database Theory. In [Brodie et al 84], pages 191-238.

**[Réty 87] P. Réty.** Improving Basic Narrowing Techniques and Commutation Properties. Technical Report No. 681, Centre de Recherche en Informatique de Nancy CRIN, Franc e, 1987.

**[Réty et al 85] P. Réty, C. Kirchner, H. Kirchner and P. Lescanne.** NARROWER: A new algorithm for unification and its applications to logic programming. In *Proc. First Int'l Conf. on Rewriting Tecniques and Applications RTA'85,* volume 202 of *Lecture Notes in Computer Science,* pages 141-157. Springer-Verlag, Berlin, 1985.

**[Robinson 65] J.A. Robinson.** A machine-oriented logic based on the resolution principle. *Journal of the ACM,* 12(1):23-41, 1965.

**[Robinson 68] J.A. Robinson.** The generalized resolution principle, *Machine Intelligence, 3,* pages 77-94. D. Michie, editor, American Elsevier, 1968.

**[Robinson Sibert 82] J.A. Robinson and E.E. Sibert.** LOGLISP: An alternative to PROLOG. *Machine Intelligence, 10,* pages 399-419. J.E. Hayes, D. Michie and Y.H. Yao, editors, John Wiley and Sons, 1982.

**[Robinson Wos 69] J.A. Robinson and L. Wos.** Paramodulation and Theorem Proving in First order Theories with Equality. *Machine Intelligence, 4,* pages 135-150. Metzer and Michie, editors, Edinburgh University Press, 1969.

**[Rusinowitz 87] M. Rusinowitz.** *Demonstration Automatique par des Tecniques de Réécriture*. Ph. D. Thesis, Centre de Recherche en Informatique de Nancy, CRIN, France, 1987. Intereditions, Paris, 1989.

**[dos Santos et al 81] C.S. dos Santos, T.S.E. Maibaum and A.L. Furtado.** Conceptual Modelling of Database Operations. *Journal of Computer and Information Science*, 10(5):299-314. Plenum Press, NY, 1981.

**[Saraswat 89] V. Saraswat.** *Concurrent Constraint Programming Languages*. Ph. D. Thesis, Carnegie-Mellon University, USA, 1989. to be published as an ACM doctoral dissertation, The MIT Press, Cambridge, Mass., 1991.

**[Saraswat Rinard 90] V. Saraswat and M. Rinard.** Concurrent Constraint Programming. In *Proc 17th ACM Symp. on Principles of Programming Languages POPL'90.* ACM, 1990.

**[Sato Sakurai 84] M. Sato and T. Sakurai.** QUTE: A Functional Language based on Unification. In *Proc. Int'l Conf. on Fifth Generation Computer Systems,* pages 157-165. ICOT, Tokyo, 1984. Also in [DeGroot Lindstrom 86], pages 131-155.

**[Scott 82] D.S. Scott.** Domains for Denotational Semantics. In *Proc. 3rd Int'l Colloquium on Automata, Languages and Programming ICALP'82,* volume 140 of *Lecture Notes in Computer Science,* pages 577-613, Springer-Verlag, Berlin, 1982.

**[Sernadas et al 87] A. Sernadas, C. Sernadas and H.D. Ehrich.** Object-Oriented Specification of Databases: an algebraic approach. In D.M. Stocker and W. Kent, editors, *Proc. 13th Int'l Conf. on Very Large Data Bases VLDB'87*, pages 107-116. Morgan Kaufmann, Los Altos, Ca., 1987.

**[Shapiro 89] E.Y. Shapiro.** The family of Concurrent Logic Programming Languages. *ACM Computing Surveys,* 21:413-510, 1989.

**[Sheperdson 88] J.C. Sheperdson.** Negation in Logic programming. In [Minker 88], pages 19-87.

**[Shin et al 87] D.W. Shin, J.H. Nang, S. Han and S.R. Maeng.** A Functional Logic Language based on Canonical Unification. In *Proc. 1987 IEEE Int'l Symp. on Logic Programming SLP'87*, pages 328-334. IEEE Computer Society Press, 1987.

**[Siekmann 84] J.H. Siekmann.** Universal Unification. In *Proc. 7th Int'l Conf. on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 1-42. Springer-Verlag, Berlin, 1984.

**[Siekmann 89] J.H. Siekmann.** Unification Theory. *Journal of Symbolic Computation*, 7:207-274, 1989.

**[Siekmann Szabo 82] J.H. Siekmann and P. Szabo.** Universal Unification and a classification of Equational Theories. In *Proc. 6th Int'l Conf. on Automated Deduction,* volume 138 of *Lecture Notes in Computer Science,* pages 369-389. Springer-Verlag, Berlin, 1982.

**[Slagle 74] J.R. Slagle.** Automated Theorem-Proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM,* 21:622-642, 1974.

**[Smolka 86] G. Smolka.** FRESH: A Higher-Order Language based on Unification. In [DeGroot Lindstrom 86], pages 469-524.

**[Smolka 88] G. Smolka.** TEL version 0.9: Report and User Manual. SEKI Report ST-87-11, FB Informatik, Universität KaisersLautern, Germany, 1988.

**[Smolka 89] G. Smolka.** Logic Programming over Polymorphically Order-Sorted Types. Ph. D. Thesis, Universität KaisersLautern, FB Informatik, Germany, 1989.

**[Steele Sussman 79] G.L. Steele and G.J. Sussman.** Constraints. In *Proc. APL'78, ACM SIGPLAN STAPL APL Quote Quad,* 9(4):208-225, 1979.

**[Stefik 84] M. Stefik.** Planning with Constraints (MOLGEN: Part1), *Artificial Intelligence,* 16:111-140, 1984.

**[Sterling Shapiro 86] L. Sterling and E. Shapiro.** The Art of Prolog. The MIT Press, Cambridge, Mass., 1986.

**[Stuckey 91a] P.J. Stuckey.** Quantifier Elimination for Structures with Uninterpreted Functors. Technical Report, Department of Computer Science, University of Melbourne, 1991.

**[Stuckey 91b] P.J. Stuckey.** Constructive Negation and Constraint Logic Programming. Technical Report, Department of Computer Science, University of Melbourne, 1991. in preparation.

**[Subrahmanyam You 84] P.A. Subrahmanyam and J-H. You.** Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. In *Proc. 1984 IEEE Int'l Symp. on Logic Programming SLP'84,* pages 144-153. IEEE Computer Society Press, 1984.

**[Subrahmanyam You 86] P.A. Subrahmanyam and J-H. You.** FUNLOG: A Computational Model Integrating Logic and Functional Programming. In [DeGroot Lindstrom 86], pages 157-198.

**[Tamaki 84] H. Tamaki.** Semantics of a Logic Programming Language with a Reducibility Predicate. In *Proc. 1984 IEEE Int'l Symp. on Logic Programming SLP'84,* pages 259-264. IEEE Computer Society Press, 1984.

**[Turner 85] D. Turner.** Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures,* volume 201 of *Lecture Notes in Computer Science*, pages 1-16. Springer-Verlag, Berlin, 1985.

**[Veloso Furtado 85] P. Veloso and A. Furtado.** Towards simpler and yet complete formal specifications. In A. Sernadas, J. Bubenko and A. Olivé, editors, *Information systems: Theoretical and Formal aspects*. North-Holland, Amsterdam, 1985.

**[Veloso et al 81] P. Veloso, J. Castilho and A. Furtado.** Systematic Derivation of Complementary Specifications. In *Proc. 7th Int'l Conf. on Very Large Data Bases VLDB'81,* Cannes, pages 409-421. IEEE Computer Society Press, 1981.

**[Voda 88] P. Voda.** The constraint language Trilogy: Semantics and Computations. Technical Report, Complete Logic Systems, North Vancouver, Canada, 1988.

**[Walinsky 89] C. Walinsky.** CLP(∑*): Constraint Logic Programming with Regular Sets. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming ICLP'89*, pages 181-196. The MIT Press, Cambridge, Mass., 1989.

**[Warren 74] D.H. Warren.** Warplan: A system for Generating Plans. memo 76, University of Edinburgh, 1974.

**[Weigand 85] H. Weigand.** Conceptual Models in Prolog. In *Proc. Database Systems. (DS1)*, pages 7-11. Hasselt, Belgium, 1985.

**[Wirsing 90] M. Wirsing.** Algebraic Specification. In J. van Leeuwen and J. Managing, editors, *Handbook of Theoretical Computer Science*, volume B, pages 675-788. North-Holland, Amsterdam, 1990.

**[Wolfram 86] D.A. Wolfram.** Intractable unifiability problems and backtracking. In E.Y. Shapiro, editor, *Proc. Third Int'l Conf. on Logic Programming ICLP'86,* volume 225 of *Lecture Notes in Computer Science,* pages 107-121. Springer-Verlag, Berlin, 1986.

**[Yamamoto 87] A. Yamamoto.** A Theoretical Combination of SLD-Resolution and Narrowing. In J.-L. Lassez, editor, *Proc. Fourth Int'l Conf. on Logic Programming ICLP'87*, Melbourne, Australia, pages 470-487. The MIT Press, Cambridge, Mass., 1987.

**[You 89] J.H. You.** Enumerating Outer Narrowing Derivations for Constructor-Based Term Rewriting Systems. *Journal of Symbolic Computation,* 7:319-341, 1989.

**[You Subrahmanyam 86] J.H. You and P.A. Subrahmanyam.** A class of confluent term rewriting systems and unification, *Journal of Automated Reasoning,* 2:391-418, 1986.

**[Zachary 88] J.L. Zachary.** A Pragmatic Approach to Equational Logic Programmming. In R.A. Kowalski and K.A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming ICLP'88,* pages 295-310. The MIT Press, Cambridge, Mass., 1988.

**[Zhang Kapur 88] H. Zhang and D. Kapur.** First-order theorem proving using conditional equations. In *Proc. 9th Int'l Conf. on Automated Deduction,* volume 310 of *Lecture Notes in Computer Science,* pages 1-20. Springer-Verlag, Berlin, 1988.

**[Zilles et al 82] S.N. Zilles, P. Lucas and J.W. Thatcher.** A look at algebraic specifications. IBM Research Report, Yorktown Heights, NY, 1982.

# Anexo 1 CLP(*H/E*) como instancia del esquema de Höhfeld-Smolka

En [Höhfeld Smolka 88, Smolka 89] se propone una noción muy general de lenguaje con restricciones junto con un método para combinar de manera jerárquica un lenguaje de programación "a la Prolog" con un lenguaje de restricciones base. Como se comentó en el capítulo dos de esta tesis, el esquema CLP de Höhfeld y Smolka refina y generaliza el de Jaffar y Lassez al relajar algunos requerimientos técnicos sobre el lenguaje de las restricciones y abstraer la sintaxis de las mismas (cfr. §2.2.4). En este anexo presentamos una definición general de lenguaje lógico con restricciones simbólicas y mostramos cómo nuestra construcción puede ser definida de manera equivalente en este marco, instanciando la definición general para la clase de restricciones en la que estamos interesados.

## 1. El lenguaje de restricciones general.

Un lenguaje de restricciones [Smolka 89] es una tupla

$$<V, \Phi, \nu, I>$$

donde:

- $V$ es un conjunto infinito numerable de variables.
- $\Phi$ es un conjunto de fórmulas (llamadas restricciones).
- $\nu$ es una función computable $\nu: \Phi \rightarrow P(V)$ que asigna a cada restricción $c \in \Phi$ un conjunto (finito) $\nu(c)$ de variables (llamadas variables restringidas por $c$ y que corresponden, generalmente, a las variables libres de la fórmula).
- $I$ es una familia no vacía de interpretaciones. Cada interpretación $I \in I$ está constituida por un conjunto no vacío $D_I$ (llamado dominio) y una aplicación *solución* $\| \ \|^I$ que asocia a cada restricción $c \in \Phi$ el conjunto $\|c\|^I$ de sus soluciones en $I$, que es un subconjunto del conjunto de valoraciones $V \rightarrow D_I$. Generalmente, y también en nuestro caso específico, una valoración $\alpha$ es solución de $c$ bajo la interpretación $I$ si $c$ es verdad en $I$ una vez que se ha dado valor a sus variables libres mediante $\alpha$.

De esta forma, el lenguaje de restricciones no está sujeto, a priori, a ninguna sintaxis concreta. En el siguiente apartado particularizamos la definición de lenguaje de restricciones general para definir un lenguaje simbólico.

## 2. El lenguaje de restricciones simbólico.

Dados dos conjuntos $\Sigma$ y $\Pi_C$ que contienen, respectivamente, una colección numerable de símbolos de función y símbolos de predicado, definimos un lenguaje de restricciones símbólico $LS(\Sigma,\Pi_C)$ [Kirchner et al 91] como una tupla

$$<V, \Phi, \nu, I>$$

donde:

- $V$ es un conjunto infinito numerable de variables.
- $\Phi$ es un conjunto de restricciones construidas sobre los términos de primer orden del conjunto $\tau(\Sigma \cup V)$ y los símbolos de predicado en $\Pi_C$ y que, además, es cerrado bajo conjunción.
- $\nu$ es la función $\nu: \Phi \rightarrow P(V)$ que asocia a cada restricción $c \in \Phi$ el conjunto $\nu(c)$ de sus variables restringidas.
- $I$ es una familia no vacía de interpretaciones. Cada interpretación $I \in I$ está constituida por el dominio $D_I$, una interpretación $f_I: D_I^n \rightarrow D_I$ para cada símbolo de función $n$-ario $f \in \Sigma$ y una interpretación $p_I \cup D_I^n$ para cada símbolo de predicado $n$-ario $p \in \Pi_C$. La aplicación *solución* $\| \|^I$, que asocia a cada restricción $c = p(t_1,t_2, ...,t_n)$ el conjunto $\|c\|^I$ de sus soluciones en $I$, se define como:

$$\|c\|^I = \{\alpha: V \rightarrow D_I \mid <\bar{\alpha}(t_1), \bar{\alpha}(t_2), ..., \bar{\alpha}(t_n)> \in p_I\}$$

siendo $\bar{\alpha}$ la función que extiende homomórficamente $\alpha$ para el caso de términos y que asumimos definida en la forma habitual [Kirchner et al 91].

Dada una $\Sigma$*–teoría* ecuacional de Horn $E$, el anterior lenguaje de restricciones simbólico admite como instancia el lenguaje $L = LS(\Sigma,\{=\})$ cuyas fórmulas son conjunciones de ecuaciones entre términos de primer orden $\in \tau(\Sigma \cup V)$, con el cociente $H/E$ del conjunto de términos "ground" $H = \tau(\Sigma)$ módulo la igualdad generada por $E$

como dominio de interpretación y el símbolo de predicado = interpretado como la igualdad $\equiv_E$ en dicho dominio. En estas interpretaciones (que, en esencia, son conjuntos cocientes de términos) la aplicación solución asocia a cada restricción un conjunto de sustituciones.

### 3. La extensión relacional y el lenguaje CLP(*H/E*).

El lenguaje $L$ puede ser extendido de manera conservativa incorporando un conjunto $R = \Pi_B$ de símbolos relacionales y las conectivas de conjunción e implicación lógicas. La sintaxis del lenguaje $R(L)$ de cláusulas relacionales queda definida por:

- el mismo conjunto infinito numerable $V$ de variables.
- el conjunto $R(\Phi)$ de fórmulas $\rho$, que incluye:
  - todas las $L$-restricciones, i.e., todas las fórmulas $c \in \Phi$.
  - todos los átomos relacionales $r(t_1, t_2, ..., t_n)$, siendo $r \in \Pi_B$ un símbolo de predicado de aridad $n$ y $t_1, t_2, ..., t_n$, $n \geq 0$, términos y que es cerrado bajo las conectivas de conjunción $(\wedge)$ e implicación $(\leftarrow)$ lógicas.
- la función $\nu: R(\Phi) \rightarrow P(V)$ que extiende de manera homomórfica la función de igual nombre definida sobre $\Phi$ para asignar a cualquier fórmula $\rho \in R(L)$ el conjunto de variables restringidas por $\rho$.
- la familia de *H/E-interpretaciones* admisibles sobre el cociente *H/E*. Cada interpretación $I$ en esta familia puede entenderse como un conjunto de relaciones $r_I \subseteq (H/E)^n$ para cada $r \in \Pi_B$.

La aplicación solución $\| \|^I$, que asocia a cada fórmula $\rho$ el conjunto $\|\rho\|^I$ de sus soluciones en $I$, puede definirse inductivamente como sigue:

$$\|r(t_1,t_2, ...,t_n)\|^I = \{\alpha: V \rightarrow H/E \mid <\bar{\alpha}(t_1), \bar{\alpha}(t_2), ..., \bar{\alpha}(t_n)> \in r_I\}$$
$$\|\rho_1 \wedge \rho_2\|^I = \|\rho_1\|^I \leftrightarrow \|\rho_2\|^I$$
$$\|\rho_1 \leftarrow \rho_2\|^I = (\{\alpha: V \rightarrow H/E\} - \|\rho_2\|^I) \cup \|\rho_1\|^I$$

El lenguaje CLP(*H/E*) se obtiene, finalmente, considerando la restricción del lenguaje $R(L)$ que acabamos de definir al caso de cláusulas relacionales definidas, que son de la forma

$$H \leftarrow c \square B_1, \ldots, B_n$$

donde $c$ es una $L$-restricción y $H$ (la cabeza) y $B_1, \ldots, B_n$ (el cuerpo), $n \geq 0$, son átomos relacionales en $R(L)$. El símbolo $\square$ se interpreta, como es habitual, como la conectiva lógica de conjunción.

Un resolvente es una fórmula de la forma:

$$c \square B_1, \ldots, B_n$$

Dados los conjuntos $\Pi_C = \{=\}$, $\Pi = \Pi_C \cup \Pi_B$ y $\Pi_C \leftrightarrow \Pi_B = \varnothing$, definimos un $(\Pi,\Sigma)$-programa CLP$(H/E)$ como una $\Sigma$-teoría ecuacional de Horn $E$ junto con un conjunto finito $P$ de cláusulas relacionales definidas de acuerdo con lo anterior.

## 4. Las semánticas declarativa y operacional y los resultados de corrección y completitud.

Dado un conjunto finito $P$ de cláusulas de programa, un $H/E$-modelo de $P$ puede definirse como una *H/E-interpretación* tal que cada valoración es solución de todas las fórmulas en $P$. A partir de los resultados del esquema, es inmediato definir un $H/E$-modelo mínimo de $P$ mediante una construcción "punto fijo". Dicho modelo $M_{(P,H/E)}$ se construye como el límite

$$M_{(P,H/E)} = \bigcup_{i \geq 0} A_i$$

de una secuencia $A_0 \subseteq A_1 \subseteq \ldots$ de *H/E-interpretaciones* que se definen de la siguiente forma:

$$r^{A0} = \varnothing$$

$$r^{Ai+1} = \{ < \bar{\alpha}(t_1), \bar{\alpha}(t_2), \ldots, \bar{\alpha}(t_n) > \mid \alpha \in \|\rho\|^{Ai} \wedge (r(t_1,t_2,\ldots,t_n) \leftarrow \rho) \in P \}$$

$$r^{M(P,H/E)} = \bigcup_{i \geq 0} r^{Ai}$$

La regla de reducción de resolventes definida en el esquema proporciona la definición abstracta de un intérprete correcto y completo para programas CLP(*H/E*). La reducción de un resolvente $R$ de la forma

$$R = c_i \square A_1, \ldots, A_n$$

mediante una cláusula de programa $(H \leftarrow c' \square \tilde{B}) \in P$ es el nuevo resolvente $R'$ de la forma

$$R' = c_i \cup \{c', A_i = H\} \square A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, \tilde{B}$$

Las propiedades de corrección y completitud de esta regla son inmediatas y pueden entenderse como una reformulación de algunos resultados presentados para la semántica $SS_2(P, H/E)$ en el capítulo cuarto: para cualquier *H/E-modelo* $M$ y cualquier valoración que hace $R'$ cierto, también $R$ es cierto, i.e., $\|R'\|^I \cup \|R\|^I$. También la propiedad de completitud se hereda del esquema: si $\alpha \in \|R\|^{M(P,H/E)}$ es una solución de la fórmula $R$ en $M_{(P,H/E)}$, entonces existe una secuencia de reducciones a partir de $R$ tal que el último resolvente de la secuencia es una restricción $c$ y $\alpha \in \|c\|^{M(P,H/E)}$. Se debe notar que aunque el modelo mínimo $M_{(P,H/E)}$ representa canónicamente todos los *H/E-modelos* del programa, no es posible obtener, a partir de este esquema, los resultados que relacionan la verdad en estos modelos y la verdad en todos los *E*-modelos del programa, resultado que sí fue obtenido (cfr. §4.3.2) formalizando nuestro lenguaje como una instancia del esquema CLP de Jaffar y Lassez.

# Anexo 2 Un intérprete Prolog para CLP(*H/E*)

En este anexo presentamos un intérprete Prolog para programas CLP(*H*/*E*) puros. El intérprete ha sido implementado en BIM-Prolog e instalado sobre estaciones de trabajo SUN 3/80. Se trata de una versión prototipo del sistema, diseñada en las primeras etapas de desarrollo del lenguaje con el objeto de convertirse en banco de pruebas tanto de programas (e.g. de las especificaciones que se describen en el capítulo cinco) como de cuestiones relacionadas con los mecanismos operacionales del lenguaje.

La arquitectura del prototipo es modular y se ajusta al modelo operacional descrito en el capítulo tres. Se han desarrollado tres módulos principales e independientes que se corresponden, respectivamente, con los tres niveles en que se organiza el sistema:

- el nivel CLP,
- el "constraint solver" incremental,
- el procedimiento de "narrowing" condicional que se utiliza como núcleo de éste.

El sistema se completa con un módulo de salida (que convierte las restricciones desde su representación interna a un formato de impresión apropiado), una biblioteca de cláusulas de uso general y un módulo de utilidades. Gracias a la modularidad, cada parte del sistema puede ser extendida sin afectar (significativamente) al resto.

El intérprete se define, esencialmente, mediante tres predicados: solve/3, ics/3 e inwing/2 que implementan, respectivamente, los tres niveles mencionados. El primero de ellos hace la función de máquina de inferencia, conduciendo las transiciones del nivel CLP(H/E) del intérprete. El segundo simplifica y resuelve (incrementalmente) las restricciones, pasándolas a un procedimiento de "narrowing" condicional "innermost" que se define por medio del tercer predicado. Esta estructura garantiza la necesaria independencia entre inferencias lógicas y "constraint solving".

Antes de presentar la implementación concreta, que incluímos como última sección de este anexo, vamos a comentar los aspectos más significativos de los módulos principales. Esta exposición trata de ser lo más comprensible posible, por lo cual hemos

obviado algunos detalles de implementación que sí se recogen en el código final (que se ofrece completo y comentado, para facilitar su lectura). En esta presentación sólo discutiremos aquellos aspectos que consideramos más importantes. Para la descripción utilizaremos, en la medida de lo posible, una sintaxis Prolog estándar y libre de facilidades "impuras".

## 1. El nivel CLP.

En la definición operacional de un lenguaje lógico hay dos aspectos no deterministas que deben ser decididos para la obtención de las secuencias de derivación: la regla de computación (o regla de selección de átomos, que determina cuál es el próximo subobjetivo a resolver) y la estrategia de búsqueda (que determina qué cláusula de programa ha de ser utilizada). En la definición del modelo operacional del lenguaje CLP(*H/E*), presentada en §3.3.2, se adoptó la regla de computación estándar del modelo CLP, en la que todos los átomos en el objetivo se seleccionan simultáneamente [Jaffar Lassez 86]. Esta regla es, canónicamente, justa pero poco compatible con una implementación eficiente. Aunque nuestra motivación para desarrollar este intérprete no ha sido obtener una implementación eficiente del lenguaje, la adopción de Prolog como lenguaje de implementación ha impulsado a utilizar sus mecanismos (en particular el de "backtracking") para programar cómodamente la búsqueda. La implementación que se describe en este capítulo sólo es, por tanto, una aproximación al modelo operacional del lenguaje.

En virtud de la conocida propiedad de independencia de la regla de computación, podemos redefinir la relación de transición $\rightarrowtail_{CLP(H/E)}$ entre configuraciones de la siguiente forma:

$$\frac{(H \leftarrow c' \ \square \ \tilde{B}.) \in P \ \wedge \ \tilde{c} = \{c', A_i = H\} \ \wedge \ s_i \overset{\tilde{c}}{\rightarrow} s_{i+1}}{< \leftarrow s_i \ \Diamond \ A_1, \ldots, A_n > \rightarrowtail_{CLP(H/E)} < \leftarrow s_{i+1} \Diamond A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n, \tilde{B} >}$$

donde se asume presente una regla de computación que selecciona, como subobjetivo a reducir, el átomo $A_i$, $1 \le i \le n$, y una regla de búsqueda que determina la cláusula de programa a utilizar. En nuestra implementación hemos adoptado la regla de computación "left_to_right" y la estrategia de búsqueda "top_to_bottom" en profundidad de Prolog. Estas estrategias son ampliamente utilizadas en la mayoría de sistemas de programación lógica existentes (e.g. Prolog II, Prolog III o CLP($\Re$)), aunque una consecuencia conocida de esta decisión es la pérdida de completitud del intérprete ya que éste es

incapaz de encontrar ninguna respuesta que se encuentre, en el árbol de búsqueda, a la derecha de la primera rama infinita.

Para reducir el espacio de búsqueda hemos adaptado una optimización que se aplica en programación lógica convencional y que se generaliza de forma inmediata a nuestro contexto. Dicha optimización explota el hecho de que sólo es necesario ensayar aquellas cláusulas cuya cabeza unifica con el átomo a ser reducido. Para implementar el nivel superior del intérprete generalizamos también la idea que permite definir un intérprete Prolog para lenguajes CLP en sólo tres cláusulas de programa [Cohen 90]. Nuestra extensión aporta la necesaria incrementalidad al proceso de computación y se desarrolla de acuerdo a las técnicas presentadas en el capítulo tres de esta tesis.

A continuación definimos el "procedimiento" solve/3 que implementa, incorporando las ideas antes comentadas, el nivel superior del intérprete CLP(*H/E*). Siguiendo las ideas de [Cohen 90], las cláusulas de programa se almacenan como cláusulas "unit":

```
clause__(Head, Body, Constr_Body).
```

correspondientes a una regla de programa CLP(*H/E*):

```
Head :- {Constr_Body} □ Body.
```

donde *Constraints* y *Body* son, respectivamente, la parte ecuacional y relacional del cuerpo de la cláusula.

De acuerdo con la definición de la relación de transición $\rightarrow_{CLP(H/E)}$, los argumentos del predicado solve/3 son:

i) la lista de átomos a resolver.
ii) el estado de partida del "constraint solver".
iii) el nuevo estado del "constraint solver".

Las reglas que definen este predicado son:

```
solve([],ICS_state,ICS_state).  /* terminal configuration */

solve([Goal|Rest_Goal],Prev_ICS_state,New_ICS_state):-
    solve(Goal,Prev_ICS_state,Temp_ICS_state),
```

```
    solve(Rest_Goal,Temp_ICS_state,New_ICS_state).

solve(Goal,Prev_ICS_state,New_ICS_state):-
    flat_Goal(Goal,Flat_Goal,List_Eqs),
    clause__(Flat_Goal,Body,Constr_Body),
    append(List_Eqs,Constr_Body,Tilde_Constr),
    ics(Prev_ICS_state,Tilde_Constr,Temp_ICS_state),
    solve(Body,Temp_ICS_state,New_ICS_state).
```

El procedimiento flat_Goal/3 realiza un aplanamiento del subobjetivo *Goal*, sustituyendo cada uno de sus argumentos por una variable nueva y añadiendo la ecuación que expresa la igualdad (=-) entre ambos a la lista de restricciones *List_Eqs*. El resultado de este proceso es la lista *List_Eqs* y el átomo aplanado *Flat_Goal*. Cuando *Flat_Goal* se resuelve con una de las cláusulas del programa, las ecuaciones en *List_Eqs* se instancian y sus elementos pasan automáticamente a representar las restricciones de igualdad entre los argumentos de *Goal* y los argumentos de la cabeza de la cláusula utilizada.

El núcleo del procedimiento solve/3 es el predicado ics/3, que define el segundo nivel del sistema. Su cometido es simplificar y verificar (incrementalmente) la satisfacibilidad de las restricciones. En la siguiente sección comentamos los aspectos más significativos de su implementación.

## 2. El "constraint solver" incremental.

Tal como se estableció en el capítulo tres, en el contexto de CLP(*H/E*) es posible diseñar un "constraint solver" incremental reutilizando la solución $\theta$ a la restricción $c_i$ encontrada en el paso anterior para comprobar la satisfacibilidad de la nueva restricción $c_i \cup \tilde{c}$. Los estados *s[c]* de este "constraint solver" han de contener esta información adicional. De acuerdo con lo establecido en la sección §3.3, dicho estado se representa como un par $s[c] = < \theta, c >$, donde $c$ es la restricción (satisfacible) acumulada y $\theta$ es la solución que estableció la satisfacibilidad de $c$. La relación de transición que define este "constraint solver" se expresa en sólo dos reglas.

(1) Regla de éxito

$$\frac{\text{existe un } E\text{-unificador } \theta' \text{ de } \tilde{c}\theta}{< \theta, c > \xrightarrow{\tilde{c}} < \theta\theta', c \cup \tilde{c} >}$$

(2) Regla de reinicio

$$\frac{\textit{no existe E-unificador de } \tilde{c}\theta \ \wedge \textit{ existe un E-unificador } \theta' \textit{ de } c \cup \tilde{c}}{< \theta, c > \ \xrightarrow{\tilde{c}} < \theta', c \cup \tilde{c} >}$$

Con objeto de ofrecer un código fuente sencillo, que pueda servir como base a diferentes extensiones del lenguaje tanto como al desarrollo de otros intérpretes, el diseño de nuestro "constraint solver" incremental reproduce fielmente este cálculo. En la codificación, sin embargo, ha resultado mucho más sencillo trabajar con dos versiones de la restricción $c$, una de ellas instanciada con la sustitución $\theta$ que determinó la satisfacibilidad de $c$, para representar los *ICS_estados*.

El "constraint solver" incremental se ha implementado por medio del procedimiento ics/3, que comprueba la satisfacibilidad de la nueva restricción acumulada, a la vez que la simplifica. Los argumentos de ics/3 son: el estado $s_i$ de partida del "constraint solver", la nueva restricción $\tilde{c}$ a acumular y el nuevo estado $s_{i+1}$ a devolver. Cada estado del constraint solver es el par formado por la restricción acumulada $c$ y la restricción instanciada con la solución $\theta$ encontrada por narrowing en el paso previo. El predicado se define mediante la siguiente regla:

```
ics(Prev_state,Tilde_Constr,New_state):-
   Prev_state = (Prev_Constr,Prev_Theta),
   New_state  = (New_Constr,New_Theta),
   append(Tilde_Constr,Prev_Constr,Temp_Constr),
   simplify(Temp_Constr,New_Constr),

   ((try_reuse(Prev_Constr,Prev_Theta,   /* regla de éxito */
        New_Constr);
     incremental(yes),!);

   (solvable(New_Constr)),              /* regla de reinicio */

   clause(new_sol(New_Theta),True),nl,!.
```

El predicado try_reuse/3 implementa la primera de las reglas que definen el "constraint solver" (regla de éxito). En ella se intenta resolver la nueva restricción instanciada con la sustitución que probó la satisfacibilidad de la restricción previa. Este predicado termina siempre con fallo, para desinstanciar la restricción. Pero en caso de ser satisfacible, se añade a la base de datos el indicador incremental(yes) e ics/3 termina con éxito.

```
try_reuse(Prev_Constr,Prev_Theta,Simpl_Constr):-
```

```
   (Prev_Constr = Prev_Theta,/* se fuerza la instanciación */
    solvable(Simpl_Constr),
    update(incremental(yes)),!,fail);
   (\+(Prev_Constr = Prev_Theta),fail).

try_reuse(_,_,_):- update(incremental(no)),!,fail.
```

El predicado solvable/1 es, en ambas reglas (éxito y reinicio), quien determina la satisfacibilidad de la restricción correspondiente, lanzando dicha restricción a unificación semántica. El predicado termina con éxito cuando se encuentra una solución. Para olvidar las instanciaciones producidas en el proceso de resolución de la restricción, se utiliza la misma estrategia que en el procedimiento anterior, por lo que se define el predicado auxiliar solvable_1, que siempre falla.

```
solvable(Temp_Constr):-
   (solvable_1(Temp_Constr);solv(yes)).

solvable_1(Temp_Constr):-
        conv_list_to_conjunction(Temp_Constr,Conj_Eqs),
        call(Conj_Eqs),
        update(solv(yes)),
        update(new_sol(Temp_Constr)),!,fail.

solvable_1(_):- update(solv(no)),!,fail.
```

Se debe notar que la salida *fallo* de todos los algoritmos se ha implementado como fallo de Prolog. Por ejemplo, si no se encuentra solución para la restricción, el procedimiento ics/3 falla y el procedimiento solve/3 ensayará otra cláusula del programa.

El predicado simplify/2 en la regla que define ics/3 implementa (obviamente, sólo para símbolos irreducibles) un algoritmo de *forma resuelta* clásico [Lassez et al 88]. Este algoritmo simplifica el conjunto de ecuaciones por aplicación de un conjunto de reglas. Estas simplificaciones se hacen imprescindibles si se quiere ofrecer una traza de ejecución de los programas que resulte fácil de seguir. La acción tomada por el algoritmo queda determinada por la forma de la ecuación. El algoritmo termina cuando no se puede aplicar ninguna regla o cuando se detecta el fallo.

1. $t = X$ se reemplaza por $X = t$. /* si $t$ no es una variable */
2. $X = X$ se elimina la ecuación.
3. $X = t$ se sustituye $X$ por $t$ en el resto de ecuaciones.
   /* si $X$ no ocurre en $t$ y $t$ es data*/

4. $f(t_1, t_2,... t_n) = f(s_1, s_2,... s_n)$ /* si $f$ es símbolo irreducible */
   se reemplaza por las ecuaciones $t_1 = s_1,... t_n = s_n$.
5. $f(t_1, t_2,... t_n) = g(s_1, s_2,... s_m)$ /*$f \neq$ g símbolos irreducibles */
   parar con fallo.

En todos los predicados anteriores se han eliminado los fragmentos de código que, en cada paso de computación, producen la salida, en un formato agradable, de los objetivos, de las restricciones (original, simplificada e instanciada) y de las soluciones encontradas por narrowing cuando se trabaja en modo traza. No obstante, en el listado del intérprete es fácil reconocer las extensiones (y optimizaciones) efectuadas sobre el código con respecto a lo comentado en este apartado.

## 3. El algoritmo de "narrowing" condicional.

Para implementar el procedimiento de unificación semántica, hemos extendido al caso condicional el algoritmo de "narrowing" selección "innermost" presentado en [Bosco et al 87, Bosco et al 88]. Para su implementación, las cláusulas de la teoría ecuacional se almacenan como cláusulas Prolog:

```
#-(Left,Right) :- Condition.
```

correspondientes a la ecuación condicional

```
Left = Right ← Condition.
```

donde *Left* y *Right* son, respectivamente, la parte izquierda y derecha de la cabeza de la ecuación y *Condition* es el cuerpo.

Siguiendo las ideas en [Bosco et al 87, Bosco et al 88], se ha programado un procedimiento init/0 que computa el conjunto de ocurrencias no variables de las partes derechas de las cabezas de las cláusulas de la teoría ecuacional.

```
init :- clause(#-(L,R),C),
        occ(R,OR),assertz(rwr(L,R,OR,C)),fail.
init.
```

El predicado occ/2 devuelve el conjunto de ocurrencias no variables de un término como una lista anidada de ceros y unos (correspondiente a la estructura del

término). Cada 0 representa una posición variable e indica que el correspondiente término no ha de ser reducido. Cada 1 indica un posible redex para narrowing.

```
occ(T,0) :-
   var(T),!.
occ(T,[1|Ann]) :-
   T=..[F|ARG],
   occ1(ARG,Ann).
```

El predicado occ1/2 se encarga de hacer la llamada para cada uno de los argumentos del término:

```
occ1([],[]).
occ1([TH|TT],[A|Ann]) :- occ(TH,A),occ1(TT,Ann).
```

Por ejemplo:

```
?- occ(menos(vacantes(X,Y),succ(cero)),Occ).
    Occ = [1,[1,0,0],[1,[1]]]
```

El procedimiento inwing/2 implementa el "innermost conditional selection narrowing" con regla de computación "left_to_right" y regla de búsqueda "top_to_bottom" en profundidad (i.e. las mismas estrategias que las utilizadas para realizar el nivel superior del intérprete). Cuando se lanza un sistema de ecuaciones a narrowing (por medio del metapredicado call/1 con una conjunción de átomos con functor `=-` como argumento), cada una de las ecuaciones se resuelve mediante la siguiente regla.

```
=-(T,S) :- inwing(T,S).

inwing(T,S):- occ(T,OT),
              occ(S,OS),!,
              narred(T,OT,TR),
              narred(S,OS,SR),TR=SR.
```

El predicado narred/3 ensaya todas las derivaciones por "narrowing" para su primer argumento sobre las ocurrencias marcadas por el segundo. En el último entrega el correspondiente término reducido.

```
narred(T,0,T) :- !.          % 0 indica que T no ha de ser reducido
narred(TC,[1|Oarg],TR) :-    % antes de reducir un término han de
   TC=..[F|Arg],                        % reducirse sus argumentos
```

```
   narrarg(Arg,Oarg,Narg),
   TC1=..[F|Narg],            % ahora  TC1 es un término "innermost"
   narrterm(TC1,TR).


narrterm(T,T).                          % paso de "narrowing" nulo
narrterm(TC,TR) :-             % regla   de   narrowing   condicional
   rwr(TC,R,OR,C),call(C),
   narred(R,OR,TR).


narrarg([],[],[]).
narrarg([A1|Arg],[O1|Oarg],[AR|ArgR]) :-
                          narred(A1,O1,AR),
                          narrarg(Arg,Oarg,ArgR).
```

## 4. Implementación realizada.

El intérprete presentado en los apartados anteriores viene acompañado de un entorno de desarrollo superpuesto a BIM-Prolog y realizado aprovechando ciertas facilidades de éste. Este entorno ofrece como utilidades más significativas la generación de reglas para el tratamiento ecuacional de la negación (que el usuario puede manipular para definirse su propia negación de forma segura) y la posibilidad de trabajar en modo traza, tanto para las derivaciones CLP(*H/E*) como para las derivaciones por "narrowing". Para las derivaciones por narrowing se muestra la ecuación condicional seleccionada, la ecuación resultante de cada paso de narrowing y cada éxito del algoritmo. Para las del nivel CLP, en cada paso de computación se muestra (siempre con un formato fácilmente legible) el objetivo actual, la nueva restricción acumulada, la restricción simplificada, la restricción instanciada con la sustitución previa y la nueva solución encontrada por narrowing. Como ilustración, incluimos un fragmento de la traza de ejecución para un objetivo que corresponde al ejemplo guía introducido en la sección §5.2.

```
?- new(clpagbd).

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Irreducible function symbols: [(0,0),(s,1),(a,0),
      (b,0),(nil,0),(apply,2),(offer,3),(hire,3)]
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Yes
?- clp.


=-=-=-=-=-=          CLP(H/E)  Mode        =-=-=-=-=-=

Enter Goal:        (utility. to exit)
```

```
  ATOMS ?  iscand(a,apply(X,apply(Y,nil))).

  CONSTRAINT (C0) ?.

        =-=-=-=-=-=-=-=-=-=-=-=-=-=
NEW GOAL: C0 [] iscand(a,apply(X,apply(Y,nil)))


******  NEW CONSTRAINT C1  ******
  a = X1
  apply(X,apply(Y,nil)) = apply(X2,X3)
  X1 = X2
  pcapply(X2,X3) = true
*********************************
 simplified constraint:
  X = a
  pcapply(a,apply(Y,nil)) = true


** reusing previous solution.. **
 instantiated constraint:
  X = a
  pcapply(a,apply(Y,nil)) = true

         there is one solution:
          Y = b
          X = a
narrowing time: 0.179993
        _________________________


        _________________________
        |        SOLUTION       |
        -------------------------

  X = a
  pcapply(a,apply(Y,nil)) = true

total cpu time: 0.359994
more (y/n) ? y.


        =-=-=-=-=-=-=-=-=-=-=-=-=-=
NEW GOAL: C0 [] iscand(a,apply(X,apply(Y,nil)))

******  NEW CONSTRAINT C2  ******
  a = X4
  apply(X,apply(Y,nil)) = apply(X5,X6)
  ~ (X4 = X5)
  pcapply(X5,X6) = true
*********************************
 simplified constraint:
  X = X5
  ~ (a = X5)
  pcapply(X5,apply(Y,nil)) = true


** reusing previous solution.. **
 instantiated constraint:
  X = X5
  ~ (a = X5)
  pcapply(X5,apply(Y,nil)) = true
```

```
         there is one solution:
          Y = a
          X5 = b
          X = b
narrowing time: 0.179993
        ____________________________


        =-=-=-=-=-=-=-=-=-=-=-=-=-=
NEW GOAL: C2 [] iscand(a,apply(Y,nil))

******  NEW CONSTRAINT C3  ******
  a = X7
  apply(Y,nil) = apply(X8,X9)
  X7 = X8
  pcapply(X8,X9) = true
  X = X5
  ~ (a = X5)
  pcapply(X5,apply(Y,nil)) = true
*******************************
 simplified constraint:
  Y = a
  X = X5
  ~ (a = X5)
  pcapply(X5,apply(Y,nil)) = true
  pcapply(a,nil) = true


** reusing previous solution.. **
 instantiated constraint:
  a = a
  b = b
  ~ (a = b)
  pcapply(b,apply(a,nil)) = true
  pcapply(a,nil) = true

         there is one solution:
          * empty substitution *
narrowing time: 0.160034
        ____________________________


        _________________________
        |        SOLUTION       |
        -------------------------

  Y = a
  ~ (a = X)
  pcapply(X,apply(a,nil)) = true

total cpu time: 1.43001
more (y/n) ? n.
```

Dado que las variables de CLP(*H/E*) son tratadas como variables Prolog y que incluso las ligaduras de las variables son manejadas por éste, para el funcionamiento en modo traza ha sido necesario desarrollar procedimientos específicos capaces de mantener y manipular estructuras de la forma (name = _Number), en las que se asocian el nombre

con que se conoce externamente una variable y su representación interna. Todas las facilidades relacionadas con este tipo de tratamiento de las variables están contenidas en el módulo de salida, que resulta enormemente útil en la práctica. El módulo incluye también las reglas que permiten simplificar la restricción final computada como respuesta de una derivación proyectando respecto a las variables del objetivo inicial, y utilizando el menor número posible de restricciones, el espacio de soluciones definido por dicha restrición.

Otras utilidades adicionales ofrecidas por el entorno son la consulta de programas, el borrado de la base de reglas y el listado del programa en uso.

El funcionamiento del sistema ha sido estudiado sobre numerosos ejemplos. Las pruebas realizadas dan resultados que son, en cuanto a tiempo y memoria, mejorables. Sin embargo, el sistema es capaz de tratar satisfactoriamente con todos los ejemplos planteados[23] y los tiempos conseguidos son comparables a los que se obtienen con intépretes de otros lenguajes lógico - ecuacionales, como RAP [Geser Hussmann 86] o EUROPA [Alpuente Ramírez 90].

El sistema que se presenta constituye el núcleo inicial de un laboratorio experimental en el que se pretende ensayar diferentes extensiones y optimizaciones del lenguaje y en cuyo desarrollo participará un grupo de trabajo mayor. A continuación incluimos el listado completo del código, en el estado actual de implementación, convenientemente documentado para facilitar la comprensión de las pequeñas modificaciones efectuadas con respecto a la descripción simplificada ofrecida en esta presentación.

[23] En problemas de síntesis, y debido a la adopción de la estrategia en profundidad, el intérprete no es capaz, por sí mismo, de generar todas las secuencias que son solución al problema, aunque el modelo operacional del lenguaje sí proporciona todas las respuestas. Sin embargo, es inmediato adaptar, e.g., las técnicas presentadas en [Choquet et al 86] para el rechazo del objetivo en base a la longitud de la secuencia sintetizada.

```
/****************************************
*****************
CLP(H/E)
ics.pro
(incremental constraint solver)
************************
****************************************
***********/



/*********************************************************************
            * ics/5  combina y verifica (incrementalmente) la
                 satisfacibilidad de las restricciones.
Los argumentos son: el estado de partida del constraint solver, la
nueva restricción (a acumular), el nuevo estado (a devolver), el
objetivo actual y la lista de variables del objetivo inicial. Cada
estado del constraint solver es una 4-tupla formada por: la restricción
acumulada, la restricción instanciada con la solución (encontrada por
narrowing) que demostró su satisfacibilidad, un número natural (que
identifica la restricción) y una lista de pares (name = _Number) donde
name es un nombre de variable y _Number es la variable Prolog asociada.
Las dos últimas componentes del estado (y los dos últimos argumentos de
ics/5) se utilizan para dar un formato agradable a la salida (que
facilita el seguimiento de la traza cuando se trabaja en modo
trace_clp) .
*********************************************************************
/


ics(Prev_state,Tilde_Constr,New_state,Goal,V):-
  Prev_state = (Prev_Constr,Prev_Theta,Prev_num,Prev_list),
  New_state  = (New_Constr,New_Theta,New_num,New_list),
  append(Tilde_Constr,Prev_Constr,Temp_Constr),

                                              /* código para la traza */
  inc_constraint_num(New_num),
  new_list(Temp_Constr,Prev_list,New_list),
  pretty_print_goal_et_constr(Goal,Temp_Constr,New_list,
       Prev_num,New_num),
  simplify(Temp_Constr,New_Constr,V),
  pretty_print_constr(' simplified constraint:',New_Constr,New_list),

                                                    /* success rule */

 ((message_nl('** reusing previous solution.. **'),
    try_reuse(Prev_Constr,Prev_Theta,New_Constr,New_list);
    incremental(yes),!);

                                          /* starting from scratch rule */

   (message_nl('** starting from scratch.. **'),
    solvable(New_Constr,New_list))),

  clause(new_sol(New_Theta),True),!.



/*********************************************************************
                                *
                      inc_constraint_num/1 incrementa el
```

```
       contador de restricciones utilizado para construir la traza.
**********************************************************************
                                    /



inc_constraint_num(New_Nc):-
        retract(constraint_counter(Nc)),
        New_Nc is Nc +1,
        assert(constraint_counter(New_Nc)),!.


constraint_counter(0).



/**********************************************************************
*
 try_reuse/4 instancia la nueva restricción acumulada con la
substitución previa e intenta probar su satisfacibilidad. En caso de
éxito, se añade a la base de datos el indicador incremental(yes).
Termina siempre con fallo, para deshacer las instanciaciones
efectuadas.
**********************************************************************
/


try_reuse(Prev_Constr,Prev_Theta,New_Constr,New_l):-
   (Prev_Constr = Prev_Theta,
    pretty_print_constr(' instantiated constraint:',New_Constr,New_l),
    solvable(New_Constr,New_l),
    update(incremental(yes)),
    !,fail);
   (\+(Prev_Constr = Prev_Theta),
       message_tab('The system is NOT solvable.'),
       message_tab_nl('_______________________________'),
       fail).

try_reuse(_,_,_,_):-
   update(incremental(no)),
   !,fail.


incremental(no).


/**********************************************************************
*
       solvable/2 determina la satisfacibilidad de una restricción.
Utiliza un predicado auxiliar solvable_1/2 que siempre falla (para
deshacer las instanciaciones). Si la restricción es resoluble,
solvable_1/2 añade a la base de datos el indicador sov(yes) y
solvable/2 termina con éxito.
**********************************************************************
/


solvable(Temp_Constr,List_Vbles):-
   (solvable_1(Temp_Constr,List_Vbles);
    solv(yes)).

solv(no).
```

```
/*********************************************************************
*
   solvable_1/2 analiza la satisfacibilidad de la restricción y falla.
Convierte una copia de ésta (representada como lista) en una conjunción
(Conj_Eqs), que se pasa al procedimiento de narrowing condicional. En
caso de éxito, se añade a la base de datos el indicador solv(yes).El
procedimiento pretty_print_sol/2 muestra (si está activo el modo traza
clp) la sustitución obtenida por narrowing.
*********************************************************************
/

solvable_1(Temp_Constr,List_Vbles):-
         assert(constraint_(Temp_Constr)),
         clause(constraint_(Temp_Constr_2),True),
         retract(constraint_(Temp_Constr_3)),
         assign_name(List_Vbles),
         conv_l_to_c(Temp_Constr_3,Conj_Eqs),
         time(call(Conj_Eqs),T_narr),
         update(solv(yes)),update(new_sol(Temp_Constr_3)),
         ((flag(yes),
           do_pretty(
            Temp_Constr_2,               /* Restricción desinstanciada */
            Temp_Constr,   /* con variables instanciadas a sus nombres */
            Temp_Constr_3,                /* instanciada por narrowing */
            [],Pretty),
            pretty_print_sol(T_narr,Pretty));
         flag(no)),!,fail.



/*********************************************************************
*
      Si no hay solución, se actualiza el indicador de solubilidad a
    solv(no). Si está activo el modo trace_clp, se informa del fallo.
                      Se termina siempre con fallo.
*********************************************************************
/


solvable_1(_,_):- update(solv(no)),
                  message_tab('The system is NOT solvable.'),
                  message_tab_nl('_________________________________'),!,fail.



/*********************************************************************
*
                  simplify/3 simplifica la restricción.
solved_form/3 simplifica las ecuaciones aplicando un algoritmo de
"forma                                                  resuelta" clásico.
re_ordering/2 reordena las restricciones para que aquellas que reducen
   el espacio de búsqueda sean procesadas primero.
*********************************************************************
/


simplify(Temp_Constr,Simpl_Constr,V):- take_vbles(V,Only_vars),
         solved_form(Temp_Constr,R_Constr,Only_vars),
         re_ordering(R_Constr,Simpl_Constr).
```

```
/*********************************************************************
          *solved_form/3: simplifica un sistema de ecuaciones
                      de acuerdo a cuatro reglas:
       1. Reemplaza ecuaciones t = X por X = t si t no es una
          variable o si siéndolo, X es del objetivo inicial y t no.
       2. Elimina ecuaciones triviales X = X.
       3. Elimina ecuaciones X = t (con t un término irreducible),
            después de sustituir X por t en el resto de ecuaciones.
       4. Reemplaza ecuaciones de la forma c(t1,...,tn) = c(s1,...,sn)
            con c un símbolo irreducible por t1 = s1,...,tn = sn.
       5. c(t1,...,tn) = d(s1,...,sn) con c \= d, para con fallo.
*********************************************************************
/


solved_form(Constraint,R_Constraint,List_Vbles):-
      vars_to_left(Constraint,L_Constraint,List_Vbles),     /* regla 1 */
      reduce_(L_Constraint,[],R_Constraint,List_Vbles).
                                                    /* reglas 2, 3 y 4 */



/*********************************************************************
*
                          regla 1. vars_to_left/3
*********************************************************************
/

/*********************************************************************
                                  *
                  1.1.  Ecuaciones X = Y pasan a Y = X
             si X no es del objetivo original e Y sí lo es.
*********************************************************************
/


vars_to_left([],[],_).

vars_to_left([=-(Left,Right)|Rest_Constraint],
             [=-(Right,Left)|Rest],List_Vbles):-
      var(Right),var(Left),
      \+(is_in(Left,List_Vbles)),is_in(Right,List_Vbles),
      vars_to_left(Rest_Constraint,Rest,List_Vbles),!.



/*********************************************************************
                                  *
       1.2. Ecuaciones t = X pasan a X = t si t no es una variable.
*********************************************************************
/

vars_to_left([=-(Left,Right)|Rest_Constraint],
             [=-(Right,Left)|Rest],List_Vbles):-
      var(Right),nonvar(Left),
      vars_to_left(Rest_Constraint,Rest,List_Vbles),!.



/*********************************************************************
                                  *
                 1.3  El resto de ecuaciones no cambia.
```

```
*********************************************************************
/


vars_to_left([=-(Left,Right)|Rest_Constraint],
             [=-(Left,Right)|Rest],List_Vbles):-
      vars_to_left(Rest_Constraint,Rest,List_Vbles).



/*********************************************************************
                                 *
     reglas 2, 3 y 4. Para obtener traza de estas simplificaciones,
                        eliminar los comentarios.
*********************************************************************
/



/*
reduce_([=-(Left,Right)|Rest_Constraint],Acc,
            Temp_Constraint,S_vars):-
      write([=-(Left,Right)|Rest_Constraint]),nl,fail.
*/



/*********************************************************************
             *regla 2.  Elimina ecuaciones triviales X = X.
*********************************************************************
/


reduce_([=-(Left,Right)|Rest_Constraint],Acc,Temp_Constraint,S_vars):-
      Left==Right,
      update(counter_rev(0)),
      reduce_(Rest_Constraint,[=-(Left,Right)|Rest_Constraint],
            Temp_Constraint,S_vars),!.



/*********************************************************************
              *regla 3. Aplica substituciones X/t (t data).
*********************************************************************
/


reduce_([=-(Left,Right)|Rest_Constraint],Acc,Constraint,S_vars):-
      var(Left),data(Right),\+(is_in(Left,S_vars)),
      call(Left = Right),
      vars_to_left(Rest_Constraint,New_syst,S_vars),
      update(counter_rev(0)),
      reduce_(New_syst,[=-(Left,Right)|Rest_Constraint],
            Constraint,S_vars),!.



/*********************************************************************
               *reglas 4 y 5.  Simplificación estructural.
*********************************************************************
/
```

```
reduce_([=-(Left,Right)|Rest_Constraint],Acc,Constraint,S_vars):-
      nonvar(Left),nonvar(Right),
      functor(Left,F1,A1),functor(Right,F2,A2),
      irreducibles(L),member((F1,A1),L),member((F2,A2),L),
      ((A1 = A2,Left =.. [F|Args1],Right =.. [F|Args2],
      join_args(Args1,Args2,L_eq),
      vars_to_left(L_eq,New,S_vars),
      update(counter_rev(0)),
      append(New,Rest_Constraint,New_join),
      reduce_(New_join,[=-(Left,Right)|Rest_Constraint],
            Constraint,S_vars),!);(!,fail)).


join_args([],[],[]).

join_args([A1|R_args1],[A2|R_args2],[=-(A1,A2)|R]):-
      join_args(R_args1,R_args2,R).



/*********************************************************************
*counter_rev/1 mantiene la cuenta de ecuaciones que han sido examinadas
por las reglas y no han sido afectadas por ellas. Cuando este número es
igual a uno se han intentado sin éxito todas las reglas.
*********************************************************************
/


counter_rev(0).



/*********************************************************************
                              *test de parada.
*********************************************************************
/


reduce_([=-(Left,Right)|Rest_Constraint],Acc,Temp_Constraint,S_vars):-
      \+([=-(Left,Right)|Rest_Constraint] == Acc),
      append(Rest_Constraint,[=-(Left,Right)],Other_Temp_Constraint),
      ((counter_rev(0),
        length([=-(Left,Right)|Rest_Constraint],Num_eqs),
        update(counter_rev(Num_eqs)),
        reduce_(Other_Temp_Constraint,[=-(Left,Right)|Rest_Constraint],
             Temp_Constraint,S_vars),!
       );
       (counter_rev(N),New_N is N - 1,
        (((New_N>1;New_N=0),
       update(counter_rev(New_N)),
       reduce_(Other_Temp_Constraint,[=-(Left,Right)|Rest_Constraint]
            ,Temp_Constraint,S_vars),!
      );
      (update(counter_rev(0)),
       [=-(Left,Right)|Rest_Constraint]=Temp_Constraint,!
      )))).

reduce_([],_,[],_).



/*********************************************************************
                                    *
```

```
        re_ordering/2 reordena ecuaciones dejando en primer lugar
    las igualdades entre términos data y luego las desigualdades ~ (X =
                                    Y).
**********************************************************************
/


re_ordering(L_i,L_o):-
         re_ordering_(L_i,L_eqs2,Rest_Constraints),
         append(L_eqs,L_eqs2,L),
         append(L,Rest_Constraints,L_o).



re_ordering_([],[],[],[]):-!.

re_ordering_([=-(=-(L,R),F)|Rest],Eqs_data,Eqs2,
         [=-(=-(L,R),F)|Rest_New]):-
     F==false,
     re_ordering_(Rest,Eqs_data,Eqs2,Rest_New),!.

re_ordering_([=-(L,R)|Rest],Eqs_data,Eqs2,New):-
     (R==false;R==true),
     re_ordering_(Rest,Eqs_data,Eqs2,Temp),
     append(Temp,[=-(L,R)],New),!.

re_ordering_([=-(L,R)|Rest],[=-(L,R)|Rest_Eqs_data],Eqs2,New):-
     data(L),data(R),
     re_ordering_(Rest,Rest_Eqs_data,Eqs2,New),!.

re_ordering_([=-(L,R)|Rest],Eqs_data,[=-(L,R)|Rest_Eqs2],New):-
     re_ordering_(Rest,Eqs_data,Rest_Eqs2,New),!.

re_ordering_([C|Rest],Eqs,Eqs2,New):-
     re_ordering_(Rest,Eqs,Eqs2,Temp),
     append(Temp,[C],New),!.
```

```
/****************************************
                ****************
                    CLP(H/E)
                  inwing.pro
          (innermost conditional selection narrowing)
                  ************************
              ****************************************
                        ***********/



/*********************************************************************
                    *init/0 computa el conjunto de
          ocurrencias no variables de las rhs's de las reglas.
*********************************************************************
                                  /


init:-  clause(#-(L,R),Condition),
        occ(R,OR),
        assertz(rwr(L,R,OR,Condition)),
        fail.

init.



/*********************************************************************
*
                  occ/2 entrega dicho conjunto como una
   lista de ceros y unos (correspondiente a la estructura del término).
*********************************************************************
/


occ(T,0):- var(T),!.                    /* 0  indica  posición  variable */

occ(T,[1|Ann]):-  T=..[F|ARG],          /* 1 indica posición no variable */
                  occ1(ARG,Ann).



/*********************************************************************
                     *occ1/2 realiza el cálculo para
                   la lista de argumentos del termino.
*********************************************************************
/


occ1([],[]).

occ1([TH|TT],[A|Ann]):- occ(TH,A),
                        occ1(TT,Ann).



/*********************************************************************
               * ~/1 tratamiento ecuacional de la negación.
*********************************************************************
/

~(P):- =- (P,false).
```

```
/*********************************************************************
*
           inwing/2 Innermost Conditional Selection Narrowing.
                   Regla de computación left-to-right,
              regla de selección top-to-bottom depth-first.
*********************************************************************
/


=-(T,S):- inwing(T,S).

inwing(T,S):-  occ(T,OT),occ(S,OS),!,
               narred(T,OT,TR),
               narred(S,OS,SR),
               TR=SR.



/*********************************************************************
*
       narred/3 ensaya todas las derivaciones por narrowing para su
     primer argumento sobre las ocurrencias marcadas por el segundo.
          El término reducido se entrega en el tercer argumento.
*********************************************************************
/


narred(T,0,T):-!.     /* 0 indica que el término no ha de ser reducido */

narred(TC,[1|Oarg],TR):-
     TC=..[F|Arg],              /* antes de reducir un término se deben */
     narrarg(Arg,Oarg,Narg), /*       reducir sus argumentos.        */
     TC1=..[F|Narg],            /*   Ahora TC1 es un término innermost. */
     narrterm(TC1,TR).



narrterm(T,T).                          /*    paso de narrowing nulo   */

narrterm(TC,TR):-                       /*  regla de nwing condicional */
     rwr(TC,R,OR,Condition),
     call(Condition),
     narred(R,OR,TR).



narrarg([],[],[]).

narrarg([A1|Arg],[O1|Oarg],[AR|ArgR]):-
     narred(A1,O1,AR),
     narrarg(Arg,Oarg,ArgR).
```

```
            /****************************************
                         ****************
                              CLP(H/E)
                            output.pro
                          (módulo de salida)
                       *************************
                  ****************************************
                            ***********/



/*********************************************************************
*
new_list/3 asocia un nombre a las variables de una lista de ecuaciones
(1º arg) que no ocurren en una lista de pares (name = _Number) (2º
arg). Devuelve (3º arg) la lista original extendida con los nuevos
pares.
*********************************************************************
/


new_list([],Vbles_der,Vbles_der).

new_list([Eq|Rest],L_Vs,Vbles_der):-  new_name(Eq,L_Vs,L_tmp),
                                      new_list(Rest,L_tmp,Vbles_der).



/*********************************************************************
*
         new_name/3 idem para una de las ecuaciones de la lista.
Se utiliza el predicado predefinido numbervars/4, que genera strings
terminados en números correlativos, y se los asocia como nombre a las
variables de la ecuación.
*********************************************************************
/
/*********************************************************************
*
                          numbervars(T,C,M,S).
   T  :  término cuyas variables se pretende nombrar.
   C  :  número a partir del que se empieza numerar.
   M  :  número final.
   S  :  prefijo del string que se asignará como nombre a las
         variables.
*********************************************************************
/


new_name(Eq,List_Vbles,New_List):-
      update(aux_name(Eq)),
      vclause(aux_name(Eq),Body_true,Vbles_Eq),
      new_name_1(Vbles_Eq,[],New_List,List_Vbles).



new_name_1([],Acc,New_List,List_Vbles):-
      append(List_Vbles,Acc,New_List).
```

```
new_name_1([(Num = Vble_Eq)|Rest_List],Acc,New_List,List_Vbles):-
      ((already_named(Vble_Eq,List_Vbles),Temp=Acc,!);
       (counter_vars(C),
        M is C+1,
        update(counter_vars(M)),
        numbervars(Aux,C,M,'X'),
        Temp = [(Aux = Vble_Eq)|Acc])),
      new_name_1(Rest_List,Temp,New_List,List_Vbles).


already_named(X,[(X1 = X2)|_]):-X==X2,!.

already_named(X,[_|R]):- already_named(X,R),!.


/*********************************************************************
*
       assign_name/1 forma una lista de ecuaciones bonitas dando,
     de acuerdo a los pares (name = _Number) de la lista List_Vbles,
                       un nombre a las variables.
*********************************************************************
/


assign_name(List_Vbles):-  remove_nonvar(List_Vbles,New_List),
                           conv_l_to_c(New_List,Conj),
                           call(Conj),!.


/*********************************************************************
*
              remove_nonvar/2 elimina de una lista de pares
        (name = _Number) aquellos en que _Number está instanciada.
           Se deja sólo un par para una variable _Number dada.
*********************************************************************
/


remove_nonvar([],[]).

remove_nonvar([(V = N)|R],R2):-
   nonvar(N),
   remove_nonvar(R,R2),!.

remove_nonvar([(V=N)|R],[(V=N)|R2]):-
   var(N),remove_1(N,R,R1),
   remove_nonvar(R1,R2),!.



remove_1(_,[],[]).

remove_1(N,[(V_ = N_)|R],R2):-
   N==N_,
   remove_1(N,R,R2),!.

remove_1(N,[(V_ = N_)|R],[(V_ = N_)|R2]):-
   \+(N==N_),
   remove_1(N,R,R2),!.
```

```
/*********************************************************************
*
        pretty_print_goal_et_constr/5 muestra el objetivo actual
            y la lista de restricciones acumulada resultante.
Para facilitar el seguimiento de la traza, el predicado assign_name/1
asigna nombres, consistentes con los asignados en pasos de derivación
previos (lista Vbles_der), a las variables del objetivo y de la
restricción. La primera regla de pretty_print_constraint/5 termina con
fallo, para deshacer las instanciaciones de las variables a sus
nombres.
*********************************************************************
/


pretty_print_goal_et_constr(Goal,Temp_Constr,Vbles_der,L_s_c,N_c):-
  ((flag(yes),
      tab,write('=-=-=-=-=-=-=-=-=-=-=-=-='),
      assign_name(Vbles_der),nl,
      write('NEW GOAL: '),write('C'),write(L_s_c),write(' [] '),
      write(Goal),nl,nl,
      headline(N_c),
      show_eqs(Temp_Constr),
      write('********************************'),nl);
   flag(no)),fail.


pretty_print_goal_et_constr(_,_,_,_,_).



/*********************************************************************
*
             headline/1 saca cabecera para nueva restricción.
*********************************************************************
/


headline(N_c):-  write('******  NEW CONSTRAINT C'),write(N_c),
                 name(N_c,T),length(T,N),M is 3 - N,spaces(M),
                 write('******'),nl,!.



/*********************************************************************
*
        pretty_print_constr/3 muestra la restricción, asignando un
                          nombre a las variables.
*********************************************************************
/


pretty_print_constr(Message,Simpl_Constr,New_list):-
       ((flag(yes),write(Message),nl,
         assign_name(New_list),
         show_eqs(Simpl_Constr),nl);
        flag(no)),fail.

pretty_print_constr(_,_,_).
```

```
/*********************************************************************
*
show_eqs/1 escribe las ecuaciones de la lista en la forma L = R y en
columna. Las ecuaciones =-(Q,false) se escriben, azucaradas, en la
forma ~ Q.
*********************************************************************
/


show_eqs([]).

show_eqs([=-(=-(Left,Right),F)|Rest]):- F == false,
      spaces(2),write('~ ('),write(Left),
      write(' = '),write(Right),write(')'),nl,show_eqs(Rest),!.

show_eqs([=-(Q,F)|Rest]):-  F == false,
      spaces(2),write('~ '),write(Q),nl,show_eqs(Rest),!.

show_eqs([=-(Left,Right)|Rest]):-
      spaces(2),write(Left),write(' = '),
      write(Right),nl,show_eqs(Rest),!.

/*********************************************************************
*
  do_pretty/5(S1,S2,S3,S,NS):
            S1:   sistema desinstanciado.
            S2:   sistema bonito (con las variables instanciadas a sus
                        nombres).
            S3:   sistema instanciado por narrowing.
            S:    parámetro acumulador
            NS:   lista en que se devuelven los pares (name = value)
                        (substitución bonita).
Dada una solución de narrowing, devuelve la correspondiente
substitución (en forma Pretty) en la lista NS. Para ello recorre S1, S2
y S3 en paralelo. Cuando en S1 encuentra una variable mete en S, si no
lo estaba ya, el nombre de la variable (que está en S2) y su valor (de
S3).
*********************************************************************
/


do_pretty([],[],[],Pretty_su,Pretty_su).

do_pretty([=-(L1,R1)|Rest_S1],
           [=-(L2,R2)|Rest_S2],
           [=-(L3,R3)|Rest_S3],
           Pretty_su,New_Pretty_su):-
   pretty_var(L1,L2,L3,Pretty_su,T_su),
   pretty_var(R1,R2,R3,T_su,T2_su),
   do_pretty(Rest_S1,Rest_S2,Rest_S3,T2_su,New_Pretty_su).



pretty_var(P1,P2,P3,Pretty_su,[(P2 = P3)|Pretty_su]):-
   var(P1),
   \+(already(P2,Pretty_su)),!.

pretty_var(P1,P2,P3,Pretty_su,Pretty_su):-
```

```
   var(P1),
   already(P2,Pretty_su),!.

pretty_var(P1,P2,P3,Pretty_su,Pretty_su):-
   functor(P1,F,0),!.

pretty_var(P1,P2,P3,Pretty_su,New_Pretty_su):-
   P1=..[F1|Args_F1],
   P2=..[F2|Args_F2],
   P3=..[F3|Args_F3],
   pretty_args(Args_F1,Args_F2,Args_F3,Pretty_su,New_Pretty_su).


pretty_args([],[],[],Pretty_su,Pretty_su).

pretty_args([Arg1|Args_F1],
        [Arg2|Args_F2],
        [Arg3|Args_F3],
        Pretty_su,
        New_Pretty_su):-
  pretty_var(Arg1,Arg2,Arg3,Pretty_su,Temp_Pretty_su),
  pretty_args(Args_F1,Args_F2,Args_F3,Temp_Pretty_su,New_Pretty_su).


already(X,[(X = Y)|_]):-!.

already(X,[_|R]):- already(X,R),!.



/*********************************************************************
*
    pretty_print_sol/2 muestra la substitución obtenida por narrowing
*********************************************************************
/


pretty_print_sol(T_n,Pretty):-
         tab,write(' there is one solution:'),nl,
         (\+(Pretty = []),pretty_out_subst(Pretty);
           tab,write('  * empty substitution *'),nl),
         write('narrowing time: '),write(T_n),nl,
         tab,write('_____________________________'),nl,nl,nl.


pretty_out_subst([]).

pretty_out_subst([H|T]):-
      spaces(10),write(H),nl,
      pretty_out_subst(T).



/*********************************************************************
     *pretty_answ_constr/3 simplifica y muestra la restricción final.
*********************************************************************
                                    /


pretty_answ_constr(Answer_Constraint,Vbles_der,V):-
```

```
      tab,write('_________________________'),nl,
      tab,write('|        SOLUTION       |'),nl,
      tab,write('-------------------------'),nl,nl,
      last_red(Answer_Constraint,Pretty_Constraint,V),
      assign_name(Vbles_der),
      show_eqs(Pretty_Constraint),nl,!.


/*********************************************************************
                  *last_red/3 última simplificación.
         Se restringe la restricción computada a las ecuaciones
  necesarias para establecer el valor de las variables del Goal inicial.
*********************************************************************
/


last_red(Answer_Constraint,Simpl_Constraint,V):-
      take_vbles(V,L_vars),
      introduce_vars_goal(Answer_Constraint,L_vars),
      select_eqs(L_vars,Answer_Constraint,Simpl_Constraint_),
      remove_triv(Simpl_Constraint_,Simpl_Constraint).



/*********************************************************************
                        *introduce_vars_goal/2.
Para cada ecuación de la forma X = t, con X una variable del objetivo
inicial, se sustituye X por t en todas sus ocurrencias en el resto de
ecuaciones.
*********************************************************************
/


introduce_vars_goal([],_).

introduce_vars_goal([=-(Left,Right)|Rest],L):-
      ((var(Left),var(Right),
        \+(is_in(Right,L)),is_in(Left,L),
        call(Left = Right));
        true),
      introduce_vars_goal(Rest,L),!.



/*********************************************************************
   *select_eqs/3 selecciona de una lista de ecuaciones las necesarias
              para definir el valor de ciertas variables.
*********************************************************************
/


select_eqs([],L_e,[]):-!.

select_eqs(L_V,L_e,N_l_e):-
      select_eqs_1(L_V,L_e,N_l),
      take_vbles(N_l,N_L_V),
      (\+(L_V == N_L_V),
       select_eqs_1(N_L_V,L_e,N_l_e);
      N_l = N_l_e),!.
```

```
/*********************************************************************
*
          select_eqs_1/3 selecciona de una lista de ecuaciones
           aquellas que contienen variables de una lista dada.
*********************************************************************
/


select_eqs_1(L_V,[],[]):-!.

select_eqs_1(L_V,[=-(L,R)|Rest],[=-(L,R)|Rest_eqs]):-
      take_vbles(L,List_vbles_L),
      take_vbles(R,List_vbles_R),
      append(List_vbles_L,List_vbles_R,List_vbles_eq),
      is_some(L_V,List_vbles_eq),
      select_eqs_1(L_V,Rest,Rest_eqs),!.

select_eqs_1(L_V, [=-(L,R)|Rest],Rest_eqs):-
      select_eqs_1(L_V,Rest,Rest_eqs),!.



/*********************************************************************
*
                              remove_triv/2
                    Elimina ecuaciones triviales t = t.
*********************************************************************
/


remove_triv([],[]).

remove_triv([=-(L,R)|Rest],Out):-  L == R,
                                   remove_triv(Rest,Out),!.

remove_triv([P|Rest],[P|Out]):- remove_triv(Rest,Out).



/*********************************************************************
*
             clear_screen/0    -    tab/0    -    lines_/1.
*********************************************************************
/


clear_screen:-put(12).    /*lines_(45).*/

tab:-spaces(8).

lines_(N):-assert(g(1)),  repeat,
                            retract(g(K)),nl,M is K+1,assert(g(M)),
                          M=N,abolish(g,1),!.



/*********************************************************************
*
  message_nl(tab)/1 muestra mensaje (si está activo el modo trace_clp).
```

```
**************************************************************************
/

message_nl(X):-       ((flag(yes),nl,write(X),nl);flag(no)).

message_tab(X):-      ((flag(yes),tab,write(X),nl);flag(no)).

message_tab_nl(X):-   ((flag(yes),tab,write(X),nl,nl);flag(no)).
```

```
/****************************************
                ****************
                    CLP(H/E)
                  library.pro
            (biblioteca cláusulas generales)
               ************************
          ****************************************
                      ***********/



/*********************************************************************
*
              append/3    -    length/2    -    member/2.
*********************************************************************
/

append([],L,L).
append([A|L1],L2,[A|L3]):-append(L1,L2,L3).

length([],0).
length([C|T],N):-length(T,M),N is M + 1.

member(Elem,List):-append(_,[Elem|_],List).


/*********************************************************************
*
        is_some/2 determina si algún elemento de la primera lista
                          ocurre en la segunda.
*********************************************************************
/

is_some([Var|Rest], List):-
      is_in(Var,List);
      is_some(Rest,List).

is_in(Var,[Vble|Rest]):-  Var==Vble,!.
is_in(Var,[Vble|Rest]):-  is_in(Var,Rest).


/*********************************************************************
*
       take_vbles/2 construye la lista de variables de un término.
*********************************************************************
/

take_vbles(T,[T]):-    var(T),!.

take_vbles(T,[]):-     functor(T,F,0),!.

take_vbles(T,List):-   T=..[F|Args_F],
                       take_vbles_args(Args_F,List).

take_vbles_args([],[]).
take_vbles_args([Arg_i|Args],List):-
          take_vbles(Arg_i,List_Arg_i),
          take_vbles_args(Args,List_Args),
```

```
          append(List_Arg_i,List_Args,List).


/*********************************************************************
        *data/1  tiene éxito si su argumento es un término data.
*********************************************************************
/


data(X):-  var(X),!.

data(X):-  (obtain_domain(L),
             member((X,0),L);
            atomic(X)),!.                              /* atoms, numbers */

data(X):-  X=..[F|F_args],
           functor(X,F,A),
           irreducibles(L),
           member((F,A),L),
           data_arg(F_args).

data_arg([]).
data_arg([Arg|Rest_F_args]):-
           data(Arg),!,
           data_arg(Rest_F_args).


/*********************************************************************
*
       conv_l_to_c/2 convierte una lista en conjunción de átomos.
                          conv_c_to_l/2 viceversa.
*********************************************************************
/

conv_l_to_c([],true).
conv_l_to_c([C|T],(C,R)):-  conv_l_to_c(T,R).


conv_c_to_l([C|T],(C,R)):-  conv_c_to_l(T,R),!.
conv_c_to_l([C],C).


/*********************************************************************
*
           desugaring/2 pasa ecuaciones de la forma azucarada
          ~(Q) a la forma ecuacional =-(Q,false) (y viceversa).
*********************************************************************
/

desugaring([],[]).

desugaring(L_eq,L_neg):-    nonvar(L_eq),
                            L_eq = [=-(Q,F)|Rest_List],
                            (F == false,L_neg = [~(Q)|L]);
                             L_neg = [=-(Q,F)|L]),
                            desugaring(Rest_List,L),!.

desugaring(L_eq,L_neg):-    nonvar(L_neg),
                            L_neg = [~(=-(X,Y))|L],
                            L_eq = [=-(=-(X,Y),false)|Rest_List],
                            desugaring(Rest_List,L),!.
```

```
desugaring(L_eq,L_neg):-    nonvar(L_neg),
                            L_neg = [~(Q)|L],
                            L_eq = [=-(Q,false)|Rest_List],
                            desugaring(Rest_List,L),!.

desugaring([=-(I,D)|Rest_List],[=-(I,D)|L]):-
      desugaring(Rest_List,L).
```