Derivaciones perezosas en Programación Lógica con restricciones ecuacionales*

María José Ramis ‡ Javier Piris ‡ María Alpuente ‡

Resumen

El lenguaje CLP($\mathcal{H}/\mathcal{E}$) es una instancia del esquema CLP [17] que resuelve ecuaciones bajo una teoría de la igualdad presentada como una teoría ecuacional de Horn $\mathcal{E}$ [1, 2]. CLP($\mathcal{H}/\mathcal{E}$) aúna el estilo de programación lógica basado en cláusulas de Horn, el paradigma basado en ecuaciones (condicionales) y la programación con restricciones. Como núcleo de su semántica operacional, en [1, 2] se define un procedimiento incremental que semidecide la solubilidad de las ecuaciones en la estructura $\mathcal{H}/\mathcal{E}$ y que está basado en un cálculo de narrowing. La terminación de este cálculo no está garantizada, por el hecho de que el conjunto de $\mathcal{E}$-unificadores de un par de términos es sólo semidecidible. En este trabajo se considera una versión perezosa del lenguaje CLP($\mathcal{H}/\mathcal{E}$) que retrasa el problema de la $\mathcal{E}$-unificación de las ecuaciones hasta haber resuelto completamente el resto de átomos en el objetivo. Para optimizar los algoritmos se propone un conjunto de transformaciones sintácticas que simplifica las restricciones y, en ocasiones, es capaz de detectar su insatisfacibilidad.

Palabras clave: Programación lógica con restricciones, Programación lógica ecuacional, Sistemas de Reescritura, Unificación Universal, Resolución perezosa.

1 Introducción

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 [5, 6, 9, 13, 18]. Recientemente, el paradigma de Programación lógica pura ha sido generalizado a un contexto más general de Programación lógica con restricciones (Constraint Logic Programming, CLP), un esquema genérico para extensiones conservativas a la programación lógica pura [12, 17]. El estilo CLP se muestra especialmente fecundo en el campo de la integración de lenguajes. La integración de la programación lógica y ecuacional en el marco de CLP resulta ser simple y elegante, especialmente por la garantía de una semántica análoga a la de la programación lógica pura. El lenguaje CLP($\mathcal{X}$) resultante de una instanciación del esquema no sólo hereda las propiedades semánticas más atractivas de la programación lógica (semántica declarativa por modelo mínimo, caracterización por punto fijo y semántica operacional equivalente) sino que, además, posee la ventaja de contar con una semántica algebraica que se define directamente sobre la estructura $\mathcal{X}$ con la que se parametriza el esquema. El concepto de estructura especifica el dominio sobre el que se efectúa la computación dando la interpretación semántica a funciones y relaciones y es el elemento clave para la definición semántica algebraica del lenguaje.

El lenguaje CLP($\mathcal{H}/\mathcal{E}$) es una instancia del esquema CLP que usa ecuaciones para expresar restricciones entre símbolos de función. La estructura con la que se parametriza el esquema es el algebra cociente $\mathcal{H}/\mathcal{E}$, i.e. la partición más fina inducida por una teoría de la igualdad $\mathcal{E}$ sobre el Universo de Herbrand $\mathcal{H}$ para el programa. El único símbolo de predicado para las restricciones es el $=$, que se interpreta como igualdad semántica en el dominio.

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 $\mathcal{H}/\mathcal{E}$ este problema es sólo (semi)-decidible. Como núcleo del mecanismo de resolución de restricciones que semidecide la solubilidad de las mismas, en [1, 2] se define un procedimiento de *narrowing* [13, 24], cuyo uso presenta como inconveniente la posibilidad de no terminación de éste cuando la restricción no es satisfacible. Como consecuencia, un

*Este trabajo ha sido parcialmente subvencionado por CICYT *TIC 92-0793* y realizado mientras la primera autora está disfrutando de una beca de Investigación de la Consellería de Cultura, Educació i Ciència de la Generalitat Valenciana.

‡Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46020 Valencia, Spain. *e-mail:* mjrg(jpiris)(maria)@dsic.upv.es

paso de derivación CLP($\mathcal{H}/\mathcal{E}$) puede conducir a computaciones que entran en ciclo sin producir ninguna respuesta y es para evitar esta dificultad por lo que en este trabajo se considera una versión perezosa del lenguaje. La idea de la evaluación perezosa está estrechamente relacionada con la de hacer todo lo que es seguro en cada momento, evitando el análisis de situaciones peligrosas.

La organización del trabajo es como sigue. Con la intención de ofrecer un trabajo autocontenido, en la sección segunda se revisan algunas nociones y resultados básicos sobre sistemas de reescritura condicionales y unificación semántica. También se reformulan algunos conceptos básicos del marco conceptual CLP en términos de transiciones y de configuraciones para facilitar el posterior uso de la aproximación estructural de Plotkin a la definición semántica operacional de CLP($\mathcal{H}/\mathcal{E}$) y se revisa la definición de la sintaxis del lenguaje. En la sección tercera formalizamos, utilizando la aproximación de Plotkin, nuestra propuesta de mecanismo computacional perezoso, que hace uso de un conjunto de transformaciones sintácticas para optimizar los cálculos [10, 23]. En la sección cuarta se presenta un intérprete Prolog para el lenguaje. Finalmente se establecen algunas conclusiones y se incluye, como anexo, un ejemplo de ejecución. Asumimos en el lector un conocimiento básico de la programación lógica [22], programación lógica con restricciones (CLP) [17, 12], ecuaciones y sistemas de reescritura condicionales [20] y unificación semántica [27]. En [4] se estudia la aplicación del lenguaje CLP($\mathcal{H}/\mathcal{E}$) al prototipado automático de aplicaciones de bases de datos.

2 Preliminares

Por Σ, Π y V (posiblemente con subíndices) denotamos, respectivamente, colecciones disjuntas numerables de símbolos de función, símbolos de predicado y símbolos de variable con sus signaturas. $\tau(\Sigma \cup V)$ y $\tau(\Sigma)$ denotan, respectivamente, los conjuntos de términos y términos sin variables (términos básicos) construidos sobre Σ y V. Un (Π, Σ)-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 (Π, Σ)-restricción es un conjunto (posiblemente vacío) de (Π, Σ)-átomos. El símbolo $\tilde{}$ denotará una secuencia finita de símbolos. $\tau(\Sigma)$ suele ser llamado el Universo de Herbrand ($\mathcal{H}$) sobre el alfabeto Σ.

Una Σ-*ecuación* $s = t$ es un par de términos $s, t \in \tau(\Sigma \cup V)$. Una Σ-*teoría* ecuacional de Horn $\mathcal{E}$ consiste en un conjunto finito de cláusulas de Horn ecuacionales, de la forma $e \leftarrow e_1, e_2, \ldots, e_n, n \geq 0$, donde $e, e_1, e_2, \ldots, e_n$ son Σ-ecuaciones. Una teoría ecuacional de Horn $\mathcal{E}$ puede verse como un sistema de reescritura $\mathcal{R}$ donde las reglas son las cabezas y las condiciones son los respectivos cuerpos. Representamos las ocurrencias de un término como secuencias, posiblemente vacías, de naturales; t/u es el subtérmino de t a la ocurrencia u; $t[u \leftarrow r]$ es el término t con el subtérmino a la ocurrencia u reemplazado por r; $\mathcal{V}(t)$ denota el conjunto de variables de un término t [20].

Sea $\mathcal{E}$ una teoría ecuacional de Horn. Se dice que el término s se reescribe al término t, $s \rightarrow_{\mathcal{R}} t$, si existe una (variante de una) cláusula $(l = r \Leftarrow s_1 = t_1, \ldots, s_n = t_n)$ en $\mathcal{E}$, una ocurrencia u de s y una sustitución θ tales que $s/u = l\theta$, $t = s[u \leftarrow r\theta]$ y $\forall i.\ 1 \leq i \leq n.\ \exists \gamma_i$ tal que $s_i\theta \rightarrow^*_{\mathcal{R}} \gamma_i \wedge t_i\theta \rightarrow^*_{\mathcal{R}} \gamma_i$.

Se dice que dos términos s y t son convergentes, $s \downarrow_{\mathcal{R}} t$, si existe un término ω tal que $s \rightarrow^*_{\mathcal{R}} \omega \wedge t \rightarrow^*_{\mathcal{R}} \omega$. La relación $\rightarrow_{\mathcal{R}}$ se dice que es *confluente* si para todo término s_1, s_2, s_3 tales que $s_1 \rightarrow^*_{\mathcal{R}} s_2 \wedge s_1 \rightarrow^*_{\mathcal{R}} s_3$, existe un término s tal que $s_2 \rightarrow^*_{\mathcal{R}} s \wedge s_3 \rightarrow^*_{\mathcal{R}} s$. La relación $\rightarrow_{\mathcal{R}}$ es *noetheriana* si no existe una cadena infinita $s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} s_3 \rightarrow_{\mathcal{R}} \ldots$ de reescrituras. Una teoría ecuacional de Horn se dice que es *canónica* sii la relación $\rightarrow_{\mathcal{R}}$ es confluente y noetheriana.

Cada teoría ecuacional de Horn $\mathcal{E}$ genera una relación de congruencia más fina $=_{\mathcal{E}}$ (llamada $\mathcal{E}$-*igualdad*) sobre $\tau(\Sigma \cup V)$ (la menor teoría que contiene todos los pares $s = t$ tales que[1] $\mathcal{E} \models s = t$). Se dice que $\mathcal{E}$ es una presentación o axiomatización de la teoría de la igualdad $=_{\mathcal{E}}$ y, por abuso, se habla a veces de la teoría de la igualdad $\mathcal{E}$ para denotar la teoría axiomatizada por $\mathcal{E}$. Denotaremos como $\mathcal{H}/\mathcal{E}$ la partición más fina $\tau(\Sigma)/=_{\mathcal{E}}$ inducida por $=_{\mathcal{E}}$ sobre el conjunto de términos básicos $\tau(\Sigma)$.

Se dice que dos términos s, t son $\mathcal{E}$-unificables (o $\mathcal{E}$-iguales) sii existe una sustitución σ tal que $s\sigma$ y $t\sigma$ están en la congruencia $=_{\mathcal{E}}$, i.e. tal que $\mathcal{E} \models s\sigma = t\sigma$. La sustitución σ es llamada un $\mathcal{E}$-unificador de s y t. Se llama $\mathcal{E}$-unificación al proceso de resolver una ecuación bajo la teoría ecuacional $\mathcal{E}$. Ya que el proceso es sólo semidecidible, un algoritmo de $\mathcal{E}$-unificación puede verse como un procedimiento de semidecisión para verificar la solubilidad de restricciones ecuacionales sobre el cociente $\mathcal{H}/\mathcal{E}$. Cada instancia de un $\mathcal{E}$-unificador representa una solución sobre este cociente. Las aproximaciones de mayor significación al problema de computar el conjunto de $\mathcal{E}$-unificadores de dos términos son tres: flat SLD resolución [6, 11, 28], conjuntos completos de transformaciones [10, 13, 23] y paramodulación [25] o formas especiales de ésta, tales como superposición [9] o narrowing [15, 24].

[1] En lo que sigue asumimos que las interpretaciones obedecen los axiomas de la igualdad. Consecuentemente, los conceptos de satisfacibilidad y consecuencia lógica están definidos con respecto a estos axiomas

Sea $\mathcal{E}$ una teoría ecuacional de Horn. Un símbolo de función $f \in \Sigma$ se dice que es *irreducible* sii no existe ninguna cláusula $(l = r \leftarrow e_1, e_2, \ldots, e_n.) \in \mathcal{E}$ tal que $l \in V$ ni f ocurre como el símbolo de función más externo en l; en cualquier otro caso f es un símbolo de función definido. En teorías donde se hace la distinción anterior, la signatura Σ queda particionada como $\Sigma = C \biguplus F$, donde C es el conjunto de símbolos de función irreducibles y F es el conjunto de símbolos de función definidos. A continuación revisamos las nociones básicas de CLP [17].

Sea $\Pi = \Pi_C \cup \Pi_B$, con $\Pi_C \cap \Pi_B = \emptyset$. Un (Π, Σ)-*programa* es un conjunto de cláusulas de la forma: $H \leftarrow c\Box.$ o $H \leftarrow c\Box B_1, \ldots, B_n.$, donde c es una (Π_C, Σ)-*restricción* finita (posiblemente vacía), y H (la cabeza) y $B_1, \ldots, B_n$ (el cuerpo), $n \geq 0$, son (Π_B, Σ)-átomos. Un *objetivo* es una cláusula de programa sin cabeza.

Dada una estructura $\mathcal{X}$ [17], se dice que una restricción c es $\mathcal{X}$-*soluble* (en símbolos $\mathcal{X} \models \tilde{\exists} c$) sii existe una $\mathcal{X}$-valuación θ (una valoración sobre el soporte de $\mathcal{X}$) tal que $\mathcal{X} \models c\theta$. Se dice entonces que θ es una $\mathcal{X}$-solución de c. También se dice que c es *satisfacible*.

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: $(P,\Im) \models (\forall)\ (c \Rightarrow G)$ (versión lógica) y $P \models_{\mathcal{X}} (\forall)\ (c \Rightarrow G)$ (versión algebraica), donde P es un programa CLP, $\mathcal{X}$ la estructura y $\Im$ es una teoría que axiomatiza $\mathcal{X}$ [17]. La teoría CLP impone restricciones sobre $\mathcal{X}$, $\Im$ y sus relaciones para establecer equivalencias entre las semánticas.

Sea P un programa CLP($\mathcal{X}$) y $G = \leftarrow c\Box A_1, \ldots, A_n.$ un objetivo. Definimos una configuración como el par $\mathcal{C} = \langle \leftarrow c \,\Box\, A_1, \ldots, A_n \rangle$. Sea $G_0 = \leftarrow c_0 \Box \tilde{B}.$ un objetivo inicial. La configuración $\mathcal{C}_0 = \langle \leftarrow c_0 \,\Box\, \tilde{B} \rangle$ es una configuración inicial. Una configuración terminal $\mathcal{C}$ tiene la forma $\mathcal{C} = \langle \leftarrow c \,\Box\, \quad \rangle$, donde c representa la restricción computada correspondiente a esa derivación.

Un intérprete (ansioso) de un lenguaje CLP($\mathcal{X}$) 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 (ansiosa) puede verse como una secuencia de pasos de reducción de objetivos en la que se acumulan restricciones, posiblemente simplificadas (previa comprobación de su satisfacibilidad). La computación termina con un conjunto satisfacible de restricciones como respuesta[2].

La regla estándard que describe un paso de $(P, \mathcal{X})$-computación se define del siguiente modo [1, 2, 17]:

$$\frac{\begin{array}{c}\exists\ n\ \textit{variantes de cláusulas en}\ P\ (H_j \leftarrow c'_j \Box \tilde{B}_j.,\, j = 1, \ldots, n)\ \wedge \\ \tilde{c} = \{c'_1, c'_2, \ldots, c'_n, A_1 = H_1, \ldots, A_n = H_n\}\ \wedge\ \mathcal{X} \models \tilde{\exists}(c_i \cup \tilde{c})\end{array}}{\langle \leftarrow c_i \Box A_1, \ldots, A_n \rangle \ \rightarrow_{CLP(\mathcal{X})}\ \langle \leftarrow c_i \cup \tilde{c}\ \Box \tilde{B}_1, \ldots, \tilde{B}_n \rangle}$$

Cuando la condición $\mathcal{X} \models \tilde{\exists}(c_i \cup \tilde{c})$ se elimina de la premisa de la regla se obtiene un intérprete perezoso para el lenguaje CLP comparable al que se describe en [12]. En [12] se asume además que un procedimiento de simplificación de restricciones se aplica en cada paso. Este procedimiento es capaz de detectar, en ocasiones, la insatisfacibilidad de la restricción.

3 Semántica Operacional (perezosa) para CLP($\mathcal{H}/\mathcal{E}$)

CLP($\mathcal{H}/\mathcal{E}$) es un lenguaje diseñado para realizar la integración de la programación lógica y ecuacional en el marco de la programación lógica con restricciones. Informalmente, un programa CLP($\mathcal{H}/\mathcal{E}$) consta 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). Formalmente, sea $\Pi_C = \{=\}$, $\Pi = \Pi_C \cup \Pi_B$ y $\Pi_C \cap \Pi_B = \emptyset$. Definimos un (Π, Σ)-*programa* CLP($\mathcal{H}/\mathcal{E}$) como un (Π, Σ)-programa extendido con una Σ-teoría ecuacional de Horn *canónica* $\mathcal{E}$.

Ejemplo **3.1**. *El siguiente conjunto de cláusulas es un programa CLP($\mathcal{H}/\mathcal{E}$).*
$(p_1)\ p(f(Z)) \leftarrow \Box\, q(Z).$ $\quad (p_2)\ q(a).$
$(e_1)\ f(a) = a.$ $\quad (e_2)\ f(d(X)) = f(X).$ $\quad (e_3)\ g(X) = Y \leftarrow f(X) = Y.$

[2] Debido a que las restricciones no se resuelven completamente en cada paso de computación se debe notar que, en el caso particular de la estructura $\mathcal{H}/\mathcal{E}$, esta forma de resolución evita el problema de la posible infinitud del número de resolventes entre una cláusula de programa y una cláusula objetivo que se presenta en las definiciones clásicas de resolución ecuacional (donde, a causa de la posible infinitud del conjunto de $\mathcal{E}$-unificadores de un par de términos, el espacio de búsqueda de la resolución puede no sólo ser infinito en profundidad sino también en amplitud) [13].

Es inmediato que un mecanismo computacional impaciente (i.e. ansioso) para CLP($\mathcal{H}/\mathcal{E}$) presenta como inconveniente la posibilidad de entrar en ciclo sin producir ninguna respuesta cuando se intenta probar la solubilidad de la restricción acumulada. El siguiente ejemplo ilustra este hecho.

EJEMPLO **3.2**. *Consideremos,e.g., el programa del ejemplo 3.1 con el objetivo inicial* $G_0 = \leftarrow \Box p(b)$. *La configuración CLP($\mathcal{H}/\mathcal{E}$) inicial es* $C_0 = \langle \leftarrow \Box p(b) \rangle$. *Considerando la cláusula* (p_1), *la nueva restricción* $\tilde{c}_1$ *es* $\tilde{c}_1 = \{\ b = f(Z)\ \}$.

El intérprete ansioso para CLP($\mathcal{H}/\mathcal{E}$) debería decidir la satisfacibilidad de la restricción $\tilde{c}_1$ antes de efectuar el correspondiente paso de computación. El intento de $\mathcal{E}$-unificar la restrición (irresoluble) $\tilde{c}_1$ utilizando la regla (e_1) fallaría al derivar la ecuación (ground) irresoluble $\{b = a\}$. Sin embargo la derivación $b = f(Z) \rightarrow_{\{Z/d(X)\}(e_2)} b = f(X) \rightarrow_{\{X/d(X')\}(e_2)} b = f(X') \rightarrow \ldots$ es infinita por aplicación de la regla (e_2).

3.1 Derivaciones perezosas

En este apartado se formaliza un mecanismo computacional perezoso como semántica operacional del lenguaje CLP($\mathcal{H}/\mathcal{E}$) para evitar el problema de la no posible terminación de un paso de computación CLP($\mathcal{H}/\mathcal{E}$) ansioso cuando la restricción acumulada no es satisfacible. Nuestra noción de *computación perezosa* es similar a la noción de resolución perezosa presentada en [13] y a la noción de reducción de objetivos que se define en [12], donde la solubilidad de las restricciones no se exige en cada paso. Es evidente, no obstante, que una estrategia "demasiado" perezosa detectaría la irresolubilidad de las restricciones "demasiado" tarde, produciendo una explosión inaceptable del árbol de derivaciones CLP. La capacidad para detectar la insatisfacibilidad de las restricciones es muy importante para podar caminos superfluos del árbol y evitar así computaciones innecesarias. En lo que sigue formalizamos la idea de un simplificador (incremental) de restricciones que también analiza la insatisfacibilidad de las mismas. En este contexto incrementalidad significa que no es necesario repetir todo el trabajo realizado en el paso de computación previo cuando se añade incrementalmente una nueva restricción.

DEFINICIÓN **3.1.1** (relación de transición $\longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})}$). *Sea c una restricción. Supongamos que c puede escribirse de la forma* $c_1 \cup c_2$ *(cualquiera de las cuales puede ser vacía) donde* c_1 *es equivalente a un conjunto de ecuaciones en forma resuelta* $\hat{\theta}_1$ *[21]. Entonces llamamos* representación de c *al par* $\langle \hat{\theta}_1, c_2 \rangle$[3]. *En lo que sigue no haremos distinción entre restricciones y su representación. Usaremos el terminal* fail *para representar restricciones irresolubles. La regla que describe un paso de computación perezosa para CLP($\mathcal{H}/\mathcal{E}$) a partir de una configuración* $\langle \leftarrow c_i \ \Box\ A_1, \ldots, A_n \rangle$ *viene dada por:*

$$\frac{\begin{array}{l}\exists\ n\ \textit{variantes de cláusulas en}\ P\ (H_j \leftarrow c'_j \Box \tilde{B}_j.,\ j = 1, \ldots, n)\ \wedge \\ \tilde{c} = \{c'_1, c'_2, \ldots, c'_n, A_1 = H_1, \ldots, A_n = H_n\}\ \wedge \\ c_{i+1} = \mathcal{S}implificación(c_i, \tilde{c})\ \wedge\ c_{i+1} \neq fail\end{array}}{\langle \leftarrow c_i \Box A_1, \ldots, A_n \rangle \ \longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})}\ \langle \leftarrow c_{i+1} \Box \tilde{B}_1, \ldots, \tilde{B}_n \rangle}$$

donde la función $\mathcal{S}implificación(c_i, \tilde{c})$ en la premisa de la regla reduce la restrición $c_i \cup \tilde{c}$ a una forma (más simple) equivalente mediante un conjunto apropiado de transformaciones, que se describirá posteriormente, sin comprobar su solubilidad en la estructura $\mathcal{H}/\mathcal{E}$, i.e. sin estudiar el problema de la $\mathcal{E}$-unificabilidad de las ecuaciones. El efecto de la simplificación es reducir el espacio de búsqueda de las restricciones disminuyendo, consecuentemente, el indeterminismo. La restricción c_{i+1} resultante del proceso de simplificación equivale lógicamente a la restricción $c_i \cup \tilde{c}$ original en el sentido de que tiene las mismas soluciones sobre la estructura $\mathcal{H}/\mathcal{E}$. Si la vuelta de la función es $fail$, la restricción $c_i \cup \tilde{c}$ es irresoluble.

Una configuración terminal $\mathcal{C}$ tiene la forma $\mathcal{C} = \langle \leftarrow c \ \Box\ \rangle$. Si la restricción c es $\mathcal{E}$-unificable entonces representa la *restricción respuesta computada* correspondiente a esa derivación.

La corrección y completitud de este cálculo es consecuencia inmediata de los resultados en [12, 17], que se aplican a cualquier lenguaje que puede formalizarse como una instancia del esquema.

[3] La restricción vacía se representa por $\langle\ \rangle$.

3.2 Conjunto de Transformaciones Sintácticas

En la premisa de la regla que describe un paso de *computación perezosa* $\longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})}$ se evalúa la función $\mathcal{S}implificación(c_i, \tilde{c})$, que reduce la restricción $c_i \cup \tilde{c}$ a una forma (más simple) equivalente. A continuación se describe en forma de reglas de inferencia el conjunto de transformaciones que definen esta función.

DEFINICIÓN **3.2.1** (relación de transición $\rightarrow_{Simpl}$). *Sea θ una sustitución. Sea E un conjunto de ecuaciones. Definimos una $\mathcal{S}impl$-configuración como un par $T = \langle\, \hat{\theta}\,,\, E \,\rangle$[4]. Se define la relación $\rightarrow_{Simpl}$ entre restricciones mediante el siguiente sistema de transición.*
(1) Regla de eliminación de ecuaciones triviales:

$$\frac{x \in V}{\langle\, \hat{\theta}\,,\, (\{x = x\} \cup E) \,\rangle \rightarrow_{Simpl} \langle\, \hat{\theta}\,,\, E \,\rangle}$$

(2) Regla de reordenación:

$$\frac{x \in V \wedge t \notin V}{\langle\, \hat{\theta}\,,\, (\{t = x\} \cup E) \,\rangle \rightarrow_{Simpl} \langle\, \hat{\theta}\,,\, (\{x = t\} \cup E) \,\rangle}$$

(3) Regla de eliminación de variables:

$$\frac{x \in V \wedge data(t) \wedge x \notin \mathcal{V}(t)}{\langle\, \hat{\theta}\,,\, (\{x = t\} \cup E) \,\rangle \rightarrow_{Simpl} \langle\, \theta.\widehat{\{X/t\}}\,,\, E\{X/t\} \,\rangle}$$

(4) Regla de descomposición de términos:

$$\frac{f \in C}{\langle\, \hat{\theta}\,,\, (\{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\} \cup E) \,\rangle \rightarrow_{Simpl} \langle\, \hat{\theta}\,,\, (\{s_1 = t_1\} \cup \ldots \cup \{s_n = t_n\} \cup E) \,\rangle}$$

(5) Regla de reescritura de términos condicional (CTR)[5] *:*

$$\frac{(\lambda = \rho \Leftarrow \tilde{e}) \in \mathcal{E} \wedge e/u = \lambda\sigma \wedge e' = e[u \leftarrow \rho\sigma] \wedge s\sigma \downarrow_{\mathcal{R}} t\sigma \;\; \forall (s = t) \in \tilde{e}}{\langle\, \hat{\theta}\,,\, (e \cup E) \,\rangle \rightarrow_{Simpl} \langle\, \hat{\theta}\,,\, (e' \cup E) \rangle}$$

(6) Reglas de fallo:

$$\frac{f\,,\, g \in C \wedge f \neq g}{\langle\, \hat{\theta}\,,\, (\{f(s_1, s_2, \ldots, s_n) = g(t_1, t_2, \ldots, t_m)\} \cup E) \,\rangle \rightarrow_{Simpl} fail}$$

$$\frac{x \in V \wedge data(t) \wedge x \in \mathcal{V}(t) \wedge t \neq X}{\langle\, \hat{\theta}\,,\, (\{x = t\} \cup E) \,\rangle \rightarrow_{Simpl} fail}$$

siendo $data(t)$ la relación definida como:
$data(t)$ *si* $(t \in C) \vee (t \in V) \vee (t = f(t_1, \ldots t_2) \wedge f \in C \wedge (\forall i,\ 1 \leq i \leq n, data(t_i)))$

DEFINICIÓN **3.2.2** ($\mathcal{S}impl$-configuración terminal T). *Definimos el conjunto de $\mathcal{S}impl$-configuraciones terminales de la siguiente forma:* $\{\, \langle\, \hat{\theta}\,,\, E \,\rangle \,/\, \langle\, \hat{\theta}\,, E \,\rangle \not\rightarrow_{Simpl} \,\} \cup \{fail\}$.

Es conocido que cada regla del cálculo $\rightarrow_{Simpl}$ transforma un conjunto de ecuaciones $\hat{\theta} \cup E$ en otro conjunto $\hat{\theta}' \cup E'$ con el mismo conjunto de $\mathcal{H}/\mathcal{E}$-soluciones [13]. Si el cálculo termina en *fail* el conjunto de ecuaciones original es irresoluble. Claramente, el cálculo $\rightarrow_{Simpl}$ siempre termina para teorías ecuacionales canónicas y es independiente del orden en que hayan sido aplicadas las reglas[6] [13]. Se debe notar la incrementalidad en el proceso de simplificación de las restricciones gracias a la representación adoptada.

[4] Denotamos con $\hat{\theta}$ la representación ecuacional $\{x_1 = t_1, \ldots, x_n = t_n\}$ de la sustitución $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$.
[5] Las variables de la cláusula $(\lambda = \rho \Leftarrow \tilde{e})$ se asumen estandarizadas aparte.
[6] Para la terminación de la regla de reescritura condicional el sistema de reescritura debe ser decreciente en el sentido de [19, 24]. Esta propiedad es más fuerte que la de terminación y asegura la decibilidad de la relación $\rightarrow_{\mathcal{R}}$ para sistemas de reescritura condicionales.

DEFINICIÓN **3.2.3** (Comportamiento del cálculo $\rightarrow_{Simpl}$).
Sea $T_0 = \langle \hat{\theta}\ ,\ (c \cup \tilde{c}\theta) \rangle$. *Definimos el comportamiento de la relación* $\rightarrow_{Simpl}$ *como la función:*

$$\mathcal{S}implificaci\acute{o}n(\langle \hat{\theta}\ ,\ c \rangle, \tilde{c}) = \begin{cases} \langle \hat{\theta}'\ ,\ c' \rangle & si\ T_0 \rightarrow^*_{Simpl} \langle \hat{\theta}'\ ,\ c' \rangle \\ fail & si\ T_0 \rightarrow^*_{Simpl} fail \end{cases}$$

El siguiente ejemplo ilustra cómo el mecanismo anteriormente descrito evita el problema de la posible no terminación de un paso de computación CLP($\mathcal{H}/\mathcal{E}$) ansiosa.

EJEMPLO **3.2.4**. *Consideremos el programa CLP($\mathcal{H}/\mathcal{E}$) definido en el ejemplo 3.1. con el objetivo inicial* $G_0 = \leftarrow \Box p(b)$. *La configuración CLP($\mathcal{H}/\mathcal{E}$) inicial es* $C_0 = \langle \leftarrow \Box p(b) \rangle$. *Considerando la cláusula* (p_1), *la nueva restricción* $\tilde{c}_1$ *es* $\tilde{c}_1 = \{\ b = f(Z)\ \}$.

El intérprete ansioso para CLP($\mathcal{H}/\mathcal{E}$) no es capaz de producir una respuesta para el objetivo, sin embargo, teniendo en cuenta que $\mathcal{S}implificaci\acute{o}n(\langle\rangle, \{b = f(Z)\}) = \langle \emptyset\ ,\ \{b = f(Z)\} \rangle = c_1'$, *el mecanismo computacional perezoso descrito en la sección 3.1 puede probar la transición* $C_0 \longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})} \langle c_1' \Box\ q(Z) \rangle$

Considerando ahora la cláusula (p2), la nueva restricción es $\tilde{c}_2 = \{Z = a\}$. *Teniendo en cuenta que* $\langle \emptyset\ ,\ (\{b = f(Z)\} \cup \{Z = a\}) \rangle \rightarrow_{Simpl\ (3)} \langle \{Z = a\}\ ,\ \{b = f(a)\} \rangle \rightarrow_{Simpl\ (5)} \langle \{Z = a\}\ ,\ \{b = a\} \rangle \rightarrow_{Simpl\ (6^i)} fail$ *se tiene que* $c_2' = \mathcal{S}implificaci\acute{o}n(\langle \emptyset\ ,\ \{b = f(Z)\} \rangle, \{Z = a\}) = fail$ *y, como consecuencia, no es posible probar la correspondiente transición* $\longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})}$. *El objetivo* G_0 *fracasa, por tanto, finitamente.*

Se debe notar también que nuestro concepto de derivación perezosa recupera la capacidad de computar funciones que no posee el intérprete CLP($\mathcal{H}/\mathcal{E}$) ansioso que se obtiene como instancia de la definición estándar presentada en la sección 2.

EJEMPLO **3.2.5**. *Sea el siguiente programa CLP($\mathcal{H}/\mathcal{E}$) que define el número de elementos de una lista:*
(e_1) n_elementos(λ) = 0. *(e_2) n_elementos(X.T) = 1+n_elementos(T).*
Consideremos el objetivo inicial $G_0 = \leftarrow$ *n_elementos(a.b.c) = X* $\Box$. *El intérprete (ansioso) que se describe en la sección 2 comprueba la satisfacibilidad de la restricción en el objetivo* G_0 *y devuelve la propia restricción como respuesta. El intérprete que se describe en esta sección simplifica la restricción n_elementos(a.b.c) = X computando la restricción respuesta X = 3.*

3.3 Optimización del cálculo $\rightarrow_{Simpl}$

A continuación presentamos una optimización del cálculo de simplificación de restricciones que está basado en la idea de convergencia de operadores *(operator joinability)* que se describe en [8]. En lo que sigue extendemos dicha noción al caso de sistemas de reescritura condicionales. En [3] se redefine este concepto en el marco formal de la teoría de la Interpretación Abstracta.

Sea $\mathcal{R}$ un sistema de reescritura de términos sobre la signatura Σ. Construimos un sistema de reescritura $\mathcal{R}'$ derivado de $\mathcal{R}$ de la siguiente forma:

- Para cada $r = (f(t_1, t_2, \ldots, t_n) \rightarrow g(s_1, s_2, \ldots, s_m) \Leftarrow \tilde{e}) \in \mathcal{R}$, se construye otra regla $r' = (f \rightarrow g \Leftarrow out(\tilde{e}))$. Se añade al sistema de reescritura abstracto $\mathcal{R}$' cada instancia $r'\rho$ de r', donde $\rho = \{X_1/f_1, \ldots, X_n/f_n\}$, siendo $\{X_1, \ldots, X_n\}$ las variables de la regla r' y $f_1, \ldots, f_n \in \Sigma$.

- Para cada $r = (f(t_1, t_2, \ldots, t_n) \rightarrow X \Leftarrow \tilde{e}) \in \mathcal{R}$, $X \in V$ se construye la regla $r' = (f \rightarrow X \Leftarrow out(\tilde{e}))$. Se añade al sistema de reescritura abstracto $\mathcal{R}$' cada instancia $r'\rho$ de r', donde $\rho = \{X_1/f_1, \ldots, X_n/f_n\}$, siendo $\{X_1, \ldots, X_n\}$ las variables de la regla r' y $f_1, \ldots, f_n \in \Sigma$, $f_1 \neq f$.

donde $out(\tilde{e}) = \{out(t) = out(s) \mid (t = s) \in \tilde{e}\}$ siendo $out(t)$ el símbolo más externo de t si $t \notin V$ y t en cualquier otro caso.

La noción de convergencia de operadores es una noción decidible que puede ser implementada, e.g., mediante un sencillo proceso que construye el grafo de la relación $\rightarrow_{\mathcal{R}'}$ de acuerdo con las reglas del sistema de reescritura $\mathcal{R}'$. Dos símbolos de función $f, g \in \Sigma$ son convergentes en $\mathcal{R}'$, $f \downarrow_{\mathcal{R}'} g$, si los caminos en el grafo que parten de los vértices correspondientes tienen, al menos, un vértice común [26]. Se debe notar que el conjunto de vértices de cada grafo es finito puesto que el número de reglas en $\mathcal{R}'$ lo es.

PROPOSICIÓN **3.3.1** ([8]). *Una ecuación* $f(\tilde{t}) = g(\tilde{s})$ *es satisfacible en* $\mathcal{R}$ *sólo si* $f \downarrow_{\mathcal{R}'} g$.

El cálculo $\rightarrow_{Simpl}$ puede reforzarse haciendo uso del resultado anterior con la introducción de la siguiente regla:

$$\frac{f \ \not\downarrow_{\mathcal{R}'} \ g}{\langle \, \hat{\theta} \ , \ (\{f(s_1, s_2, \ldots, s_n) = g(t_1, t_2, \ldots, t_m)\} \cup E) \, \rangle \ \rightarrow_{Simpl} \ fail}$$

EJEMPLO **3.3**. *Sea $(P, \mathcal{E})$ un programa CLP($\mathcal{H}/\mathcal{E}$) definido por las siguientes reglas:*
(p_1) p(b(Z)) $\leftarrow$ $\Box$ q(Z). *(p_2) q(0).* *(e_1) a(f(X)) = c(X).* *(e_2) b(f(X)) = d(X).*

Sea $G_0 = \leftarrow \ \Box$ p(a(X)) el objetivo inicial. La configuración inicial es $C_0 = \langle \leftarrow \Box$ p(a(X)) $\rangle$. Utilizando la regla (p_1) del programa se tiene la restricción acumulada $\tilde{c} = \{$ a(X) = b(Z) $\}$. La inconsistencia de $\tilde{c}$ sólo puede reconocerse tras detectar la no convergencia de los operadores más externos a y b, i.e. $a \not\downarrow_{\mathcal{R}} b$, respecto a las reglas del sistema derivado $\mathcal{R}' = \{ \ a \rightarrow_{\mathcal{R}'} c, \ b \rightarrow_{\mathcal{R}'} d \ \}$. El cálculo $\rightarrow_{Simpl}$ original no detecta esta inconsistencia dado que ninguna de las reglas de transformación es aplicable a $\tilde{c}$.

4 Un intérprete Prolog (perezoso) para CLP($\mathcal{H}/\mathcal{E}$)

En esta sección se presenta un intérprete Prolog (perezoso) para programas CLP($\mathcal{H}/\mathcal{E}$). El intérprete ha sido implementado en BIM-Prolog e instalado sobre estaciones SUN-3/80.

El intérprete se define, esencialmente, mediante cuatro predicados: `solve/3`, `ics/3`, `simplify/5` e `inwing/3`. El primero de estos predicados conduce las transiciones del nivel CLP($\mathcal{H}/\mathcal{E}$) del intérprete. Los otros dos realizan el segundo nivel del sistema, simplificando (incrementalmente) las restricciones y detectando, en su caso, su insatisfacibilidad. El predicado `inwing/3` define un procedimiento de narrowing condicional básico que se puede utilizar para comprobar la satisfacibilidad de las restricciones en cada configuración terminal.

4.1 El nivel $\rightarrow_{LCLP}$

En la definición del modelo operacional del lenguaje se adoptó la regla de computación estándar del modelo CLP, en la que todos los átomos existentes en el objetivo se seleccionan simultáneamente [16]. Esta regla es, canónicamente, justa pero poco compatible con una interpretación eficiente. 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 es, por tanto, una aproximación al modelo operacional del lenguaje.

En virtud de la propiedad de independencia de regla de computación podemos redefinir la relación de transición $\longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})}$ entre configuraciones de la siguiente forma:

$$\frac{\begin{array}{c}(H \leftarrow c' \Box \tilde{B}.) \in P \ \wedge \ \tilde{c} = \{c', A_i = H\} \ \wedge \\ c' = \mathcal{S}implificaci\acute{o}n(c, \tilde{c}) \ \wedge \ c' \neq fail\end{array}}{\langle \leftarrow c \Box A_1, \ldots, A_n \rangle \ \longrightarrow_{LCLP(\mathcal{H}/\mathcal{E})} \ \langle \leftarrow c' \Box A_1, \ldots, A_{i-1}, \tilde{B}, A_{i+1}, \ldots, A_n \rangle}$$

donde se asume presente una regla de computación que selecciona, como subobjetivo a reducir, el átomo A_i, $1 \leq i \leq n$, y una regla de búsqueda que determina qué cláusula de programa utilizar. En la implementación se utilizan las reglas estándar *left-to-right* y *top-to-bottom* en profundidad de Prolog.

Las cláusulas de programa se almacenan como cláusulas unit: `clause_(Head,Body,ConstrBody)` correspondientes a una regla del programa `Head:- ConstrBody` $\Box$ `Body.`, donde `ConstrBody` y `Body` son, respectivamente, la parte ecuacional y relacional del cuerpo de la cláusula.

Los argumentos del predicado `solve/3` son: la lista de átomos a resolver, el estado de partida del simplificador de restricciones y el estado con que termina.

```
solve([],ICS_state,ICS_state).
solve([Goal|Restgoal],PrevICS_state,NewICS_state):-
    solve(Goal,PrevICS_state,TempICS_state),
solve(Restgoal,TempICS_state,NewICS_state).
solve(Goal,PrevICS_state,NewICS_state):-
    flatgoal(Goal,Flatgoal,Listeqs), clause_(Flatgoal,Body,ConstrBody),
    append(Listeqs,ConstrBody,TildeConstr),
```

```
ics(PrevICS_state,TildeConstr,TempICS_state),
    solve(Body,TempICS_state,NewICS_state).
```

El procedimiento `flatgoal/3` realiza un aplanamiento del subobjetivo `Goal`, sustituyendo cada uno de sus argumentos por una nueva variable y añadiendo la ecuación que expresa la igualdad entre ambos a la lista de restricciones `Listeqs`.

```
flat_goal(Goal,Goal,[]).
flat_goal(Goal,Flat_goal,[Eq|Rest_eq]):-
    Eq=..[=-,Arg_goal,New_vble], Goal=..[Pred_goalArg_goal|Rest_arg_goal],
    Partial_goal=..[Pred_goal|Rest_arg_goal],
    flat_goal(Partial_goal,Flat_part_goal,Rest_eq),
    Flat_part_goal=..[Pred_goal|Args_flat_part_goal],
    Args_flat_goal=[New_vble|Args_flat_part_goal],
    Flat_goal=..[Pred_goal|Args_flat_goal].
```

El núcleo del procedimiento `solve/3` es el predicado `ics/3`, que define el segundo nivel del sistema.

4.2 El simplificador de restricciones $\rightarrow_{TRANS}$

Los argumentos de `ics/3` son: el estado de partida, la nueva restricción $\tilde{c}$ a acumular y el estado resultante que incorpora la restricción (simplificada) acumulada.

```
ics(PrevICS_state,Tildeconstr,NewICS_state):-
    PrevICS_state=(Constr,Hat_theta), NewICS_state=(Constr_prima,Hat_theta_prima),
    apply_sust(Tildeconstr,Hat_theta,Tilde_hat_constr),
    append(Constr,Tilde_hat_constr,Tempconstr),
    simplify(Hat_theta,Tempconstr,Hat_theta_prima,Constr_prima,Fail), Fail.
```

El predicado `simplify/5` es el núcleo del simplificador. `simplify/5` implementa las reglas del sistema $\rightarrow_{TRANS}$ descritas en 3.2. El quinto argumento del predicado `simplify/5` informa de la detección de insatisfacibilidad.

```
simplify(Prev_theta,Prevconstr,New_theta,NewConstr,Fail):-
    apply_rules(Prev_theta,Prevconstr,New_theta,NewConstr,Fail),
    (Fail, update(solv(yes)); update(solv(no))), !.
```

El predicado `apply_rules/5` aplica las reglas de transformación sintácticas a un conjunto de restricciones.

```
apply_rules(Prev_theta,Prevconstr,New_theta,NewConstr,Fail):-
    apply_rules_(Prev_theta,Prevconstr,New_theta,NewConstr_,Fail,[],_),
    (Fail,take_off(NewConstr_,NewConstr);true).

apply_rules_(Prev_theta,[],Prev_theta,Rest_,true,L,Rest):-
    append(Rest,L,Rest_), !.
apply_rules_(Prev_theta,Constr,N_theta,R_constr,Fail,L,NL):-
    trans_rule(Prev_theta,Constr,(Rule_used,N_Constr),New_theta,Fail_,L,NL_),
    (Fail, \+(Rule_used=none),
    apply_rules_(New_theta,N_Constr,R_constr,N_theta,Fail,NL_,NL);
    Fail=Fail_), !.

trans_rule(Theta,Constr,(1,NConstr),Theta,true,L,L):-
    Constr=[=-(Left,Right)|RestC], Left==Right, NConstr=RestC, !.
trans_rule(Theta,Constr,(2,NConstr),Theta,true,L,L):-
    Constr=[=-(Left,Right)|RestC], nonvar(Left), var(Right),
    NConstr=[=-(Right,Left)|RestC], !.
trans_rule(Theta,Constr,(3,NConstr),New_theta,true,L,NL__
    Constr=[=-(Left,Right)|RestC], var(Left), data(Right),
```

```
    takevbles(Right,Var_Right), \+(in(Left,Var_Right)), difieren(L,Left,Right,E2,NL_),
    apply(Left,Right,RestC,R), append(R,E2,NConstr),
    append(NL_,[=-(Left,Right)],NL__),
    compose_sust(Theta,[=-(Left,Right)],New_theta), !.
trans_rule(Theta,Constr,(4,NConstr),Theta,true,L,L):-
    Constr=[=-(Left,Right)|RestC], nonvar(Left), nonvar(Right),
    functor(Left,F1,A1), functor(Right,F2,A2),
    irreducibles(F1,A1), irreducibles(F2,A2)
    F1=F2, A1=A2, Left=..[F1|Args1], Right=..[F2|Args2],
    joinargs(Args1,Args2,Listarest), append(Listarest,RestC,NConstr), !.
trans_rule(Theta,Constr,(5,NConstr),Theta,true,L,L):-
    Constr=[=-(Left,Right)|RestC], normal_form(Left,Right,LeftN,RightN),
    (\+(Left=LeftN) ; \+(Right=RightN)),
    append([=-(LeftN,RightN)],RestC,NConstr), !.
trans_rule(Theta,Constr,(5,NConstr),Theta,true,L,NL):-
    Constr=[=-(Left,Right)|RestC],
    normal_form(Left,Right,LeftN,RightN),
    LeftN==Left, RightN==Right,
    (var(Right), append(L,[=$(Right,Left)],NL);
    append(L,[=$(Left,Right)],NL)), !.
trans_rule(_,Constr,('6i',unsatisfiable_constraint),_,fail,L,L):-
    Constr=[=-(Left,Right)|Rest], nonvar(Left), nonvar(Right),
    functor(Left,F1,A1), functor(Right,F2,A2),
    irreducibles(F1,A1), irreducibles(F2,A2)
    (F1\==F2; A1\==A2), !.
trans_rule(_,Constr,('6ii',unsatisfiable_constraint),_,fail,L,L):-
    Constr=[=-(Left,Right)|Rest], var(Left), data(Right),
    takevbles(Right,Var_Right), in(Left,Var_Right), !.
trans_rule(Hat_theta,Constraint,(none,Constraint),Hat_theta,true,L,L):- !.

difieren(L,Left,Right,E2,NL):- take_eqs(L,NLL,E),
    aplicaciones(E,Left,Right,E2,New), apply(Left,Right,NLL,N),
    append(N,New,NL), !.

aplicaciones([],Left,Right,[],[]):- !.
aplicaciones([E1|Rest],Left,Right,E,Rest2):- apply(Left,Right,[E1],E2),
    [E1]\==E2, aplicaciones(Rest,Left,Right,R,Rest2),
    append(E2,R,E), !.
aplicaciones([E1|Rest],Left,Right,R,Rest2):- E1=..[=-,L,R],
    aplicaciones(Rest,Left,Right,R,R2),
    append([=$(L,R)],R2,Rest2], !.

take_eqs([=$(L_,R)|R__],L,[=-(L_,R)|R_]):- take_eqs(R__,L,R_).
take_eqs([E|R],[E|L],R_]):- take_eqs(R,L,R_), !.
take_eqs([],[],[]).

take_off([],[]).
take_off([=$(L,R)|R_],[=-(L,R)|RR]):- take_off(R_,RR), !.
take_off([E|R],[E|R_]):- take_off(R,R_), !.
```

El predicado `apply_sust/3` aplica un conjunto de sustituciones a un término.

```
apply_sust(G,[],G):- !.
apply_sust(G,[=-(L,R)|T],G_):-
    sust(L,R,G,GG), apply_sust(GG,T,G_), !.

sust(L,R,T,R):- L==T, !.
sust(L,R,T,T):- (var(T);ground(T)), !.
sust(L,R,T,T_):- T=..[F|A], sust_a(L,R,A,As), T_=..[F|As], !.
```

```
sust_a(_,_,[],[]).
sust_a(L,R,[A|RA],[As|RAs]):-
    sust(L,R,A,As), sust_a(L,R,RA,RAs), !.
```

El predicado compose_sust/3 realiza la composición de dos sustituciones.

```
compose_sust(T,[],T):- !.
compose_sust([],S,S):- !.
compose_sust(T,S,NT):- comp1(T,S,T1), extract_var(T,VT),
    comp2(VT,S,NS), append(T1,NS,NT), !.

comp1([],_,[]):- !.
comp1([H|T],[=-(V1,T1)|Rest],[HS|R]):-
    sust(V1,T1,H,Hs), V1\==TS, !, comp1(Rest,T,R), !.

comp2(_,[],[]):- !.
comp2(VT,[=-(V,T)|R],NS):- in(V,VT), !, comp2(VT,R,NS), !
comp2(XT,[X|R],[X|NS]):- comp2(VT,R,NS), !.
```

El predicado extract_var/2 extrae las variables de los pares de una sustitución.

```
extract_var([],[]):- !.
extract_var([=-(X,_)|Rest],[X|R]):- extract_var(Rest,R), !.
```

El predicado normal_form/4 reduce los argumentos de una ecuación a su forma canónica.

```
normal_form(T,S,TN,SN):-rewrite(T,TN),rewrite(S,SN),!.

rewrite(TC,TC):- write(TC),nl, var(TC),!.
rewrite(TC,TR):- TC=..[F|Arg],rewrite_arg(Arg,Narg),
    TC1=..[F|Narg],rewrite_term(TC1,TR).

rewrite_term(TC,TR):- TC=..[F|Args_m],functor(TC,F,N_A),
    functor(TC_prima,F,N_A),rwr(TC_prima,R,_,T),
    TC_prima=..[F|Args_prima], test(Args_m,Args_prima),
    Args_prima = Args_m, list_est(T,L),call_condition(L),rewrite(R,TR).
rewrite_term(T,T).

rewrite_arg([],[]).
rewrite_arg([A1|Arg],[AR|ArgR]):-rewrite(A1,AR),rewrite_arg(Arg,ArgR).

call_condition([=-(I,D)|Rc]):-!,normal_form(I,D,N,N), call_condition(Rc),!.
call_condition(_).

test([],[]):-!.
test([X|Rx],[Y|Ry]):-test_term(X,Y),test(Rx,Ry),!.
test_term(X,Y):-nonvar(X),nonvar(Y),X=..[F|Ax],Y=..[F|Ay],test(Ax,Ay),!.
test_term(X,Y):-var(Y).

list_est([],''):- !.
list_est([X|[]],X):- !.
list_est([H|T],E):- list_est(T,E1), E=(H,E1), !.
```

4.3 El algoritmo de narrowing condicional

Las cláusulas de la teoría ecuacional están almacenadas como cláusulas Prolog con la siguiente forma `=-(Left,Right):- Condition`. Siguiendo las ideas de [6] 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.

occ(T,0) :- var(T), !.
occ(T,[1|Ann]):- T=..[F|ARG], occ1(ARG,Ann).
occ1([],[]).
occ1([TH|TT],[A|Ann]):- occ(TH,A), occ1(TT,Ann).
```

El procedimiento `inwing/2` implementa un procedimiento de narrowing condicional con regla de computación *left-to-right* y regla de búsqueda *top-to-bottom* en profundidad.

```
=-(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(T,0,T):- !.
narred(TC,[1|Oarg],TR):- TC=..[F|Arg], narrarg(Arg,Oarg,Narg),
    TC1=..[F|Narg], narrterm(TC1,TR).

narrterm(T,T).
narrterm(TC,TR):- rwr(TC,R,OR,T), call(T), narred(R,OR,T).
narrarg([],[].[]).
narrarg([A1|Arg],[O1|Oarg],[AR|ArggR]):-
    narred(A1,O1,AR), narrarg(Arg,Oarg,ArgR).
```

5 Conclusiones

En este trabajo hemos formalizado un mecanismo computacional perezoso como semántica operacional del lenguaje CLP($\mathcal{H}/\mathcal{E}$) que retrasa el problema de la $\mathcal{E}$-unificación de las restricciones hasta alcanzar un configuración terminal. Nuestra idea de computación perezosa es semejante a la noción de resolución perezosa de [14] y a la de reducción de objetivos que se define en [12]. Este modo de computación evita el problema de la posible no terminación de un paso de computación estándar [16] motivado por la indecibilidad del problema de la satisfacibilidad ecuacional. El intérprete propuesto en este trabajo es capaz de calcular todas las respuestas que puede obtener el intérprete impaciente. Como ventaja adicional se consigue un progreso parcial, en cuanto a simplificación de la restricción respuesta computada, en cada paso de computación CLP. Esto es posible gracias a la incorporación de un conjunto de transformaciones sintácticas que, basadas en algunas reglas clásicas para el tratamiento de la unificación en teorias ecuacionales [7, 10, 14, 23] son capaces, en ocasiones, de detectar la insatisfacibilidad de la restricción.

Agradecimientos

Agradecemos profundamente la labor de nuestros revisores, cuyas valiosas indicaciones nos han permitido mejorar la presentación de este trabajo.

Referencias bibliográficas

[1] M. Alpuente and M. Falaschi. Incremental Constraint Satisfaction for Equational Logic Programming. In J. Maluszynski and M. Wirsing, editors, *Proc. of PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 111–122. Springer-Verlag, Berlin, 1991.

[2] M. Alpuente, M. Falaschi, and G. Levi. Incremental Constraint Satisfaction for Equational Logic Programming. Theoretical Computer Science. to appear. Extended abstract in *Proc. ILPS'91 Workshop on Defeasible Reasoning and Constraint Solving*, pages 47-75. San Diego, Ca., Oct. 1991.

[3] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Inconsistency for Incremental Equational Logic Programming. In *Proc. of PLILP'92*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[4] M. Alpuente and M.J. Ramírez. An Equational Constraint Logic Approach to Conceptual Modelling. In *Proc. of DEXA'92*, pages 26-31. Springer-Verlag, Wien New York, 1992.

[5] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, 3:217–236, 1986.

[6] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[7] H.J. Bürckert. Lazy E-unification - a method to delay alternative solutions. Technical report, Université de Nancy, France, 1987.

[8] N. Dershowitz and J. Sivakumar. Solving Goals in Equational Languages. In S. Kaplan and J.Joaunnaud, editors, *Proc. First Int'l Workshop on Conditional Term Rewriting*, volume 308 of *Lecture Notes in Computer Science*, pages 45–55. Springer-Verlag, Berlin, 1987.

[9] L. Fribourg. Slog: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, 1985.

[10] J.H. Gallier and W. Snyder. A general complete E-unification procedure. In P. Lescanne, editor, *Rewriting Techniques and Applications*, volume 256 of *Lecture Notes in Computer Science*, pages 216–227. Springer-Verlag, Berlin, 1987.

[11] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42, 1991.

[12] M. Höhfeld and G. Smolka. Definite relations over constraint languages. Technical report, IBM Deutschland GmbH, Stuttgart, 1988.

[13] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1989.

[14] S. Hölldobler. Conditional equational theories and complete sets of transformations. *Theoretical Computer Science*, 75:85–110, 1990.

[15] H. Hussman. Unification in conditional-equational theories. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1986.

[16] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, 1986.

[17] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[18] J. Jaffar, J.-L. Lassez, and M.J. Maher. A logic programming language scheme. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 441–468. Prentice Hall, Englewood Cliffs, NJ, 1986.

[19] 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.

[20] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, I. Oxford University Press, 1991.

[21] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Ca., 1988.

[22] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, Berlin, 1987. second edition.

[23] A. Martelli, C. Moiso, and G.F. Rossi. An algorithm for unification in equational theories. In *Proc. IEEE Symp. on Logic Programming*, pages 180–186. IEEE, 1986.

[24] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing. To appear in Proc. Third Int'l Conf. on Algebraic and Logic Programming, 1992.

[25] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[26] M.J. Ramis, J. Piris, and M. Alpuente. Derivaciones perezosas en programación lógica con restricciones ecuacionales. In *Proc. XVIII Conferencia Latinoaméricana de Informática PANEL'92*, pages 1008-1015. Las Palmas de Gran Canaria, 1992

[27] J.H. Siekmann. Universal unification. In *7th Int'l Conf. on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42. Springer-Verlag, Berlin, 1984.

[28] H. Tamaki. Semantics of a Logic Programming Language with a Reducibility Predicate. In *Proc. First IEEE Int'l Symp. on Logic Programming*, pages 259–264. IEEE, 1984.

[29] 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 309-421. IEEE Computer Society Press, 1981.

A Anexo

EJEMPLO **A.1**. `knapsack 0/1` *[2]*

$$
\begin{array}{lllll}
\mathcal{P} = & \{ & (p_1) & knapsack(M,L,W) & \leftarrow addweight(M) = W \ \Box \ sublist(M,L). \\
 & & (p_2) & sublist([\,],Z). & \\
 & & (p_3) & sublist([X \mid Y],[X \mid Z]) & \leftarrow \ \Box \ sublist(Y,Z). \\
 & & (p_4) & sublist(Y,[X \mid Z]) & \leftarrow \ \Box \ sublist(Y,Z). \ \} \\
\mathcal{E} = & \{ & (e_1) & 0 + y = y. & \\
 & & (e_2) & s(x) + y = s(x+y). & \\
 & & (e_3) & weight(a) = s(s(s(0))). & \\
 & & (e_4) & weight(b) = s(s(0)). & \\
 & & (e_5) & weight(c) = s(s(0)). & \\
 & & (e_6) & addweight([\,]) = 0. & \\
 & & (e_7) & addweight([X \mid Y]) = weight(X) + addweight(Y). \ \} &
\end{array}
$$

En la siguiente tabla se compara el comportamiento de:

- un intérprete *eager* (no incremental), que utiliza un procedimiento de narrowing condicional básico como núcleo del *constraint solver* que (semi-)decide la solubilidad de las restricciones, y
- varios intérpretes *lazy*, que retrasan el problema de la $\mathcal{E}$-unificabilidad de las ecuaciones hasta haber resuelto completamente el resto de átomos en el objetivo.

Las abreviaturas empleadas en la tabla son las siguientes:

E	*intérprete eager*, sin optimizaciones.
MM	*intérprete lazy*, con reglas Martelli-Montanari + reescritura condicional (CTR)
MM + Dz	MM + regla *non-joinability* Dershowitz [3].
MM + Dz + aN	MM + Dz + regla *abstract Narrowing* [3].
iEA	*intérprete lazy incremental*, con iEA como solver [3].

Los intérpretes MM + Dz + aN e iEA incorporan una regla de *narrowing abstracto* optimizado, en el que cada nodo del árbol se simplifica antes de un paso de narrowing (detectándose, posiblemente, su insatisfacibilidad). Para la simplificación se utiliza un conjunto $\rightarrow_{Simpl}$ de reglas formado por las reglas de Martelli-Montanari, reescritura condicional (CTR) y la regla de *non-joinability* de Dershowitz. Para los *benchmarks* eliminamos a veces la regla CTR para potenciar la mejora producida por la incrementalidad en la construcción del árbol.

Los tiempos de la tabla se dan en segundos sobre una SUN 3/80 workstation con 16 Mbytes de RAM.

$\wedge$ **Tabla de Tiempos I**

	E		MM		MM + Dz	
Goal	answer	CPU	answer	CPU	answer	CPU
g_1	no	$\boxed{\infty}$	no	$\boxed{0.48}$	no	$\boxed{0.58}$
g_2[7]	no	0.41	no	∞	no	0.42
g_3[8]	no	∞	Y + s(0) = 0	0.18	Y + s(0) = 0	0.30
			no	∞	no	∞
g_4	M = [a]	1.72	M = [a]	2.00	M = [a]	2.48
g_5	$\boxed{X = addw([a,b])}$	2.30	$\boxed{X = 5}$	2.86	$\boxed{X = 5}$	2.40
g_6[7]	no	$\boxed{3.20}$	no	$\boxed{\infty}$	no	$\boxed{0.96}$
g_5, g_2	no	33.52	no	∞	no	17.72
g_5, g_6	no	16.32	no	∞	no	20.98

A Tabla de Tiempos I (cont.)

	MM + Dz + aN				iEA			
	CTR $\notin \to_{Simpl}$		CTR $\in \to_{Simpl}$		CTR $\notin \to_{Simpl}$		CTR $\in \to_{Simpl}$	
Goal	answer	CPU	answer	CPU	answer	CPU	answer	CPU
g_1	no	$\boxed{32.12}$	no	0.50	no	$\boxed{30.72}$	no	0.46
g_2	no	3.10	no	0.42	no	3.22	no	0.36
g_3[8]	no	3.22	no	$\boxed{0.94}$	no	3.50	no	1.08
g_4	M = [a]	3.24	M = [a]	2.82	M = [a]	3.76	M = [a]	3.56
g_5	X = addw([a,b])	2.52	X = 5	2.58	X = addw([a,b])	2.62	X = 5	2.46
g_6	no	1.42	no	1.00	no	1.47	no	1.16
g_5, g_2	no	53.22	no	$\boxed{18.82}$	no	50.98	no	$\boxed{15.40}$
g_5, g_6	no	25.72	no	21.48	no	22.38	no	19.04

Goals:

$g_1 : knapsack([a\|M], [a,b,c], s(0))$	$c_{0_1} : s(0) = addweight([a\|M])$	Unsolvable
$g_2 : knapsack([\,], [a,b,c], weight(Y))$	$c_{0_2} : weight(Y) = addweight([\,])$	Unsolvable
$g_3 : knapsack([\,], [a,b,c], Y + s(0))$	$c_{0_3} : Y + s(0) = addweight([\,])$	Unsolvable
$g_4 : knapsack(M, [a,b,c], s(s(s(0))))$	$c_{0_4} : s(s(s(0))) = addweight(M)$	Solvable
$g_5 : knapsack([a,b], [a,b,c], X)$	$c_{0_5} : X = addweight([a,b])$	Solvable
$g_6 : knapsack([b], [\,], s(s(0)))$	$c_{0_6} : s(s(0)) = addweight([b])$	Solvable

EJEMPLO **A.2**. *[3]*

$$
\begin{array}{llcll}
\mathcal{P} = \{ & & & & \\
p_1) & p(X) & \Leftarrow & h(f(X)) = Y, f(Z) = c(W) & \Box. \\
p_2) & q(X) & \Leftarrow & f(g(X)) = c(0), h(f(Y)) = s(0) & \Box. \\
p_3) & qq(X) & \Leftarrow & c(f(X)) = f(c(Y)) & \Box. \\
p_4) & m(X) & \Leftarrow & f(X) = c(g(h(Y))) & \Box. \}
\end{array}
$$

$$
\begin{array}{llcl}
\mathcal{R} = \{ & & & \\
r_1) & h(0) & \to & s(0). \\
r_2) & h(c(0)) & \to & 0. \\
r_3) & h(f(X)) & \to & X \Leftarrow g(X) = X. \\
r_4) & f(0) & \to & c(0). \\
r_5) & f(c(X)) & \to & c(f(X)). \\
r_6) & g(0) & \to & 0.\}
\end{array}
\qquad
\begin{array}{llcl}
\mathcal{R}_{\mathcal{A}} = \{ & & & \\
r_{1_{\mathcal{A}}}) & h(0) & \to & s(0). \\
r_{2_{\mathcal{A}}}) & h(c(0)) & \to & 0. \\
r_{3_{\mathcal{A}}}) & h(f(X)) & \to & X \Leftarrow g(X) = X. \\
r_{4_{\mathcal{A}}}) & f(0) & \to & c(0). \\
r_{5_{\mathcal{A}}}) & f(c(X)) & \to & c(\bot). \\
r_{6_{\mathcal{A}}}) & g(0) & \to & 0. \}
\end{array}
$$

Tabla de Tiempos II

		E	MM + Dz[8]	MM + Dz + aN		iEA	
Goal	answer			CTR $\notin \to_{Simpl}$	CTR $\in \to_{Simpl}$	CTR $\notin \to_{Simpl}$	CTR $\in \to_{Simpl}$
$p(X)$	Y=h(f(X))	0.38	0.26	0.86	0.76	0.94	0.88
$q(X)$	no	∞	∞	12.70	5.08	12.00	5.18
$qq(X)$	c(f(X))=f(c(Y))	0.38		0.32		0.38	
	f(X)=f(Y)		0.22		0.28		0.40
$m(X), q(Z)$	no	∞	∞	3343.52	351.78	3306.94	339.74
$m(X), qq(Z)$	f(X)=c(g(h(Y))) c(f(Z))=f(c(X1))	0.78		$\boxed{5.50}$		$\boxed{2.38}$	
	f(X)=c(g(h(Y))) f(Z)=f(X1)		0.54		$\boxed{12.50}$		$\boxed{2.48}$
$m(X), p(Z)$	f(X)=c(g(h(Y))) X1=h(f(Z))	0.80	0.58	$\boxed{46.32}$	$\boxed{40.42}$	$\boxed{3.20}$	$\boxed{2.99}$

[7] El intérprete perezoso MM entra en un camino de derivación CLP infinita por aplicación de la cláusula de programa (p_4) mientras el intérprete impaciente, basado en narrowing, detecta la irresolubilidad de la restricción y tiene un espacio de búsqueda finito.

[8] Los intérpretes perezosos MM y MM + Dz computan una restricción simplificada (irresoluble) en tiempo finito. Sin embargo el procedimiento de narrowing, que se utiliza al final de cada derivación para (semi-)decidir qué respuestas computadas son correctas, entra en ciclo sin reconocer su insatisfacibilidad. Los intérpretes MM + Dz + aN e iEA reconocen el fallo en un tiempo finito.

Ejemplo **A.3**. `Agencia de Empleo` *[4]*

% SPEC agbd

% IRREDUCIBLES

%	a	/ 0
%	b	/ 0
%	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[9]

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).
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).
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).
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).

[9] El tratamiento de la negación en CLP($\mathcal{H}/\mathcal{E}$) es ecuacional. La información negativa se incorpora en forma de reglas de reescritura que se utilizan para reducir las restricciones a $false$. Las expresiones $\neg P$ en las reglas siguientes debe entenderse, por tanto, como la versión azucarada de $P = false$.

```
%    EQUATIONAL CLAUSES

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).

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).

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).

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).

pcoffer(W,M,T)                                  ← M>0.
¬pcoffer(W,M,T)                                 ← ¬(M>0).

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).

¬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
```

Tabla de Tiempos III

		E	MM + Dz + aN		iEA	
Goal	answer		CTR $\notin \rightarrow_{Simpl}$	CTR $\in \rightarrow_{Simpl}$	CTR $\notin \rightarrow_{Simpl}$	CTR $\in \rightarrow_{Simpl}$
g_1	no	1.20	0.72	0.68	0.74	0.74
g_2	no	2.72	6.84	3.12	6.68	3.20
g_3	X=a	0.26	1.22	0.52	1.40	0.78
	no	3.80	7.08	5.06	7.38	5.80
g_4	X=a	0.26	1.22	0.54	1.42	0.78
	no	4.10	6.72	5.30	7.08	5.80
g_5	X=a	0.36	1.60	0.58	1.58	0.90
	X=b	1.28	6.84	3.38	8.42	4.62
	no	5.04	18.16	13.18	21.62	13.94
g_6	X=a	0.42	1.42	0.64	1.60	0.94
	Y=a X $\neq$ a	1.38	7.74	4.32	8.22	4.78
	no	11.40	19.34	14.44	21.58	14.82
g_4, g_2	no	oveflow	$\boxed{19.36}$	$\boxed{11.06}$	$\boxed{19.02}$	$\boxed{10.62}$

Goals:

g_1 : $iscand(a, nil)$
g_2 : $iscand(a, apply(b, nil))$
g_3 : $iscand(X, apply(a, nil))$
g_4 : $iscand(a, apply(X, nil))$
g_5 : $iscand(X, apply(a, apply(b, nil)))$
g_6 : $iscand(a, apply(X, apply(Y, nil)))$

Conclusiones:

- *Comparación Eager / Lazy*
  - Los intérpretes lazy computan todas las respuestas que puede obtener el intérprete eager.
  - Las reglas de simplificación y fallo de los intérpretes lazy son capaces, en ocasiones, de detectar la insatisfacibilidad de la restricción evitando el posible problema de la no terminación de un paso de derivación eager (ver tabla I, goal g_1, $\infty \div 0.48$).
  - Como ventaja adicional, se consigue un progreso parcial, en cuanto a simplificación de la restricción respuesta computada, en cada paso de derivación CLP. Se recupera la capacidad de computar funciones propia de todo lenguaje ecuacional (ver tabla I, goal g_5, $X = addweight([a, b]) \div X = 5$).
  - El retraso del test de solubilidad puede, sin embargo, producir un espacio de búsqueda (para la resolución lazy CLP) infinito, cuando el espacio de la resolución eager CLP era finito (ver tabla I, goal g_6, $3.20 \div \infty$).

- *Comparación* MM / MM + Dz / MM + Dz + aN
  - No podemos establecer un resultado general para la mejora en eficiencia producida por la incorporación de la regla de narrowing abstracto ya que depende fuertemente de los ejemplos seleccionados.
    - Si las reglas de MM ya reconocían el fallo, simplemente no hay sobrecarga en tiempo (ver tabla I, goal g_1, $0.48 \div 0.58$).
    - En el caso en que sólo la regla de narrowing abstracto sea capaz de detectar el fallo, se consigue una mejora decisiva en los tiempos (ver tabla I, goal g_3, $\infty \div 0.94$).
    - En caso de no detectar fallo, el resultado variará para cada programa en particular, dependiendo de la relación entre el tiempo invertido en intentar la detección del fallo, y el comportamiento del procedimiento de narrowing que analizara la misma restricción. En el caso general, la sobrecarga no es muy grande, mientras puede evitar una derivación infinita y/o el riesgo de computar respuestas incorrectas.

- *Comparación* MM + Dz + aN / iEA
  - La construcción incremental del árbol de búsqueda puede ser una buena optimización. Ha conseguido mejorar significativamente los tiempos (ver tabla II, goal $(m(X), p(Z))$, $46.32 \div 3.20$ y $40.42 \div 2.99$). La bondad de la técnica depende del ratio entre la reducción del espacio de búsqueda debida a la poda y la sobrecarga debida a la computación de la función $merge$. La estrategia resulta efectiva en los casos en que las restricciones no se demuestran irresolubles con las reglas de MM + Dz y los grafos de narrowing abstracto presentan suficientes ramas de fallo a la izquierda de la primera rama de éxito. Para este caso, la exploración incremental del grafo de narrowing abstracto para las restricciones acumuladas tiene un coste menor que el que se requiere para explorar el grafo completo en cada paso de computación. La mejora se aprecia tanto más cuantas menos variables comparten las restricciones. Cuando el narrowing es *normalizante* el efecto puede resultar inapreciable.