Razonamiento Automático en Sistemas de Representación del Conocimiento*

(y su relación con la enseñanza de la Inteligencia Artificial)

F.J. Martín, J.A. Alonso, M.J. Hidalgo, J.L. Ruiz

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
{fjesus,jalonso,mjoseh,jruiz}@cica.es

Resumen

Uno de los objetivos, tanto docentes como investigadores, del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla es el estudio de distintos sistemas de representación del conocimiento. En lo que a docencia se refiere, este estudio se centra en la utilización de dichos sistemas para resolver problemas clásicos de la Inteligencia Artificial. En el plano de la investigación se intentan demostrar distintas propiedades de estos sistemas utilizando herramientas de Razonamiento Automático.

Entre los sistemas de representación del conocimiento más utilizados se encuentran los basados en reglas de producción. Su uso más común ha sido el de desarrollar sistemas expertos, existiendo herramientas específicas para tal fin, como son CLIPS o ART. Aún siendo ésta su principal utilidad, estos sistemas de representación del conocimiento se pueden aplicar a otros problemas en el campo de la Inteligencia Artificial.

En concreto, proponemos aquí el uso de ART y CLIPS para la implementación y enseñanza de tableros semánticos proposicionales y unificación sintáctica como paso previo a la construcción de Sistemas de Razonamiento Automático de primer orden.

1 Sistemas de Producción

Los elementos básicos de los sistemas basados en reglas de producción son los hechos y las reglas. Los hechos sirven fundamentalmente para almacenar información y las reglas indican qué acciones se pueden llevar a cabo ante la presencia de ciertos hechos. Una regla tiene dos partes, el lado izquierdo o precondición, donde se listan los hechos cuya presencia es requerida para que la regla pueda ser aplicada y el lado derecho o postcondición, donde aparece la secuencia de acciones que la regla realiza:

```
(defrule <nombre>
  <precondicion>
  =>
  <postcondicion>)
```

En algunas ocasiones la regla puede requerir la ausencia de hechos de cierto tipo, ésto se indica anteponiendo la particula negativa `not` al hecho en cuestión, o la verificación de ciertas restricciones (`&` y `test`). Las acciones más comunes son la eliminación de hechos (`retract`), la inserción de nuevos hechos (`assert`) y la escritura en pantalla (`printout`).

Decimos que una regla está activa si en el sistema se encuentran todos los hechos cuya presencia es requerida por la misma, no hay ninguno de aquellos cuya ausencia es necesaria y se verifican todas las restricciones sobre los hechos. Una regla se desactiva cuando dejan de cumplirse alguna de las condiciones anteriores.

Para poder trabajar con los valores almacenados en los hechos se utilizan variables. Se distinguen dos tipos de variables, las simples (`?x`) y las múltiples (`$?x`). Mientras que las variables simples sólo pueden almacenar un valor, las múltiples pueden almacenar cualquier cantidad de ellos, incluso ninguno.

*Este trabajo ha sido financiado por la DGES del MEC, proyectos PB96-0098-C04-04 y PB96-1345

El sistema de producción consta de un proceso de inferencia que elige una de las reglas activas, realiza las acciones que en ella se indican y comprueba la activación y desactivación de las reglas. Este proceso se realiza hasta que no quedan reglas activas en el sistema, por tanto es posible construir reglas que hagan que nunca termine. Una de las propiedades de los sistemas de producción es la refracción: una vez que una regla se ejecuta con un conjunto de hechos, nunca más se ejecutará con el mismo conjunto.

En este trabajo utilizamos dos sistemas basados en reglas de producción bastante conocidos, ART y CLIPS. Una introducción más amplia sobre los sistemas basados en reglas de producción en general y sobre CLIPS en particular se puede encontrar en [4].

2 Tableros Semánticos Proposicionales

El método de los tableros semánticos se puede utilizar para comprobar si una fórmula dada es una tautología. En este sentido, es un sistema de refutación: es decir, para demostrar que una fórmula X es una tautología, se comienza a trabajar con $\neg X$ hasta llegar a una contradicción.

2.1 Lógica proposicional

Los elementos mínimos de la lógica proposicional son las variables proposicionales, éstas representan afirmaciones elementales que pueden ser ciertas o falsas. Notaremos por P, Q,... a las variables proposicionales. Hay también dos constantes básicas en la lógica, que representan los dos valores de verdad, **cierto**, que lo notaremos como $\top$, y **falso**, que lo notaremos como $\bot$,

Definición. 1 *Una fórmula proposicional atómica es una variable proposicional, $\top$ ó $\bot$.*

Definición. 2 *El conjunto de las fórmulas proposicionales es el menor conjunto $\mathcal{P}$ tal que:*

1. *Si A es una fórmula atómica entonces $A \in \mathcal{P}$.*
2. *Si $X \in \mathcal{P}$ entonces $\neg X \in \mathcal{P}$*
3. *Si $\circ$ es una conectiva proposicional binaria y $X, Y \in \mathcal{P}$ entonces $(X \circ Y) \in \mathcal{P}$*

Las conectivas proposicionales binarias que consideraremos son la disyunción, la conjunción, la implicación y la equivalencia, que notaremos respectivamente como $\vee$, $\wedge$, $\rightarrow$ y $\leftrightarrow$.

La lógica proposicional clásica es bivaluada, los valores de verdad que consideraremos son **cierto** y **falso**, notados respectivamente como **t** y **f**. El valor de verdad de $\top$ es **cierto** y el de $\bot$ es **falso**. El valor de verdad de una fórmula depende del valor de verdad de las variables proposicionales y de las conectivas proposicionales que en ella intervienen. El valor de verdad de las conectivas proposicionales se determina de acuerdo con la tabla 1. Llamaremos valoración a una asignación cualquiera de valores de verdad a las variables proposicionales.

	$\neg$
t	f
f	t

		$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$
t	t	t	t	t	t
t	f	f	t	f	f
f	t	f	t	t	f
f	f	f	f	t	t

Tabla 1: Valor de verdad de las conectivas proposicionales.

Definición. 3 *Dos fórmulas son equivalentes, si tienen el mismo valor de verdad para cualquier valoración que se considere.*

Siguiendo la notación de [3], agrupamos todas las fórmulas proposicionales de la forma $(X \circ Y)$ y $\neg(X \circ Y)$ en dos categorías, aquellas que actúan de manera conjuntiva, a las que llamaremos fórmulas α, y aquellas que actúan de manera disyuntiva, a las que llamaremos fórmulas β. Una fórmula X es α si existen dos fórmulas α_1 y α_2 tales que X es equivalente a $\alpha_1 \wedge \alpha_2$. Una fórmula X es β, si existen dos fórmulas β_1 y β_2 tales que X es equivalente a $\beta_1 \vee \beta_2$. La categoría de cada una de las fórmulas de la forma $(X \circ Y)$ y $\neg(X \circ Y)$ y las componentes α_1, α_2, β_1 y β_2 de las mismas, se pueden ver en la tabla 2

Definición. 4 *Una fórmula X es una tautología si su valor de verdad es **cierto** para cualquier valoración que se considere.*

Conjuntivas		
α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \rightarrow Y)$	X	$\neg Y$

Disyuntivas		
β	β_1	β_2
$X \vee Y$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$X \rightarrow Y$	$\neg X$	Y
$X \leftrightarrow Y$	$\neg X \wedge \neg Y$	$X \wedge Y$
$\neg(X \leftrightarrow Y)$	$\neg X \wedge Y$	$X \wedge \neg Y$

Tabla 2: Fórmulas α y β y sus componentes.

2.2 Tableros semánticos

Un tablero semántico es un árbol donde cada nodo está etiquetado con una fórmula. Cada rama representa la conjunción de todas las fórmulas que en ella aparecen. Un tablero representa la disyunción de las fórmulas asociadas a todas las ramas que posee.

Dado un árbol finito $\mathcal{T}$, para aplicar las reglas de expansión de tableros se escoge una rama Θ de $\mathcal{T}$ y una ocurrencia en Θ de una fórmula no atómica X. Si X es $\neg\neg Z$ ($\neg\top$ ó $\neg\bot$) aumentamos Θ añadiendo al final un nodo etiquetado con Z ($\bot$ ó $\top$). Si X es una fórmula α, añadimos al final de Θ dos nodos nuevos, uno etiquetado con α_1 y otro con α_2. Si X es una fórmula β, añadimos al nodo final de Θ dos hijos, uno etiquetado con β_1 y otro etiquetado con β_2. Si el árbol resultante lo llamamos $\mathcal{T}^*$, diremos que $\mathcal{T}^*$ se ha obtenido a partir de $\mathcal{T}$ mediante la aplicación de una regla de expansión de tableros.

Definición. 5 *Sea $\{A_1, \ldots, A_n\}$ un conjunto finito de fórmulas proposicionales. El árbol de una única rama cuyos nodos están etiquetados con $A_1, \ldots, A_n$ es un tablero para $\{A_1, \ldots, A_n\}$*

Si $\mathcal{T}$ es un tablero para $\{A_1, \ldots, A_n\}$ y $\mathcal{T}^*$ se obtiene a partir de $\mathcal{T}$ mediante una regla de expansión de tableros, entonces $\mathcal{T}^*$ es un tablero para $\{A_1, \ldots, A_n\}$.

Definición. 6 *Una rama Θ de un tablero se dice cerrada si $\bot$ aparece en Θ o existe una fórmula X tal que X y $\neg X$ aparecen en Θ. Diremos que un tablero está cerrado si todas sus ramas están cerradas.*

Dado un tablero $\mathcal{T}$, si Θ es una rama cerrada de $\mathcal{T}$, puesto que dicha rama representa la conjunción de todas las fórmulas que en ella aparecen, se tendrá que dicha conjunción es **falsa** bajo cualquier valoración. Asimismo, si el tablero $\mathcal{T}$ está cerrado entonces, para cualquier valoración, todas sus ramas son "falsas", y por tanto también lo será la disyunción de estas. Es decir, el propio tablero será "falso".

Definición. 7 *Una prueba por tableros de X es un tablero cerrado para $\{\neg X\}$.*

El método de los tableros, tal como se ha descrito, es correcto (i.e., si hay una prueba por tableros de X entonces X es una tautología), y completo (i.e., si X es una tautología entonces hay una prueba por tableros de X). Sin embargo presenta un problema de implementación: la terminación del proceso. Ya que la selección de ramas y elementos en éstas es no determinista, una misma ocurrencia de una fórmula podría ser seleccionada en varias ocasiones para expandir una rama, lo cual no sólo es redundante sino que puede llevar a un ciclo infinito. Esto se evita eliminando del tablero, una vez procesada, la fórmula seleccionada en cada paso de expansión. Aún queda pendiente el hecho de que la selección de ramas y elementos de éstas es no determinista, que se soluciona normalmente estableciendo un orden entre las ramas y otro entre los elementos de cada rama y realizando la selección siguiendo dichos órdenes.

2.3 Implementación en ART

El objetivo de la implementación en ART es comprobar si una fórmula dada X es una tautología, utilizando el método de los tableros semánticos descrito en la sección anterior.

Dentro del sistema representaremos la fórmula original X como un hecho de la forma `(es-tautologia ?for)` donde `?for` es la expresión prefija de X. Representaremos las conectivas utilizando los símbolos `not`, `and`, `or`, `if` e `iff`, que se asociarán respectivamente a $\neg$, $\wedge$, $\vee$, $\rightarrow$ y $\leftrightarrow$. Las constantes $\top$ y $\bot$ serán representadas como `T` y `F` respectivamente. Cualquier otro símbolo será una representación válida de una variable proposicional.

Cada rama de un tablero será representada dentro de ART como un hecho de la forma

`(rama $?lista)` donde `$?lista` es la lista de fórmulas, escritas en notación prefija, que aparecen en dicha rama.

El primer paso del proceso será el de crear el tablero inicial, un árbol de una única rama y un único elemento en dicha rama, la negación de la fórmula original. De esto se encargará la siguiente regla:

```
(defrule comprueba-tautologia
   ?f1 <- (es-tautologia ?for)
   =>
   (retract ?f1)
   (assert (rama (not ?for))
   (formula-original ?for)))
```

Almacenamos la fórmula original como un hecho de la forma `(formula-original ?for)` donde `?for` es la expresión prefija de dicha fórmula.

La regla de expansión de tableros que afecta a una rama con una fórmula de la forma $\neg\neg Z$ introduce Z como nueva fórmula en la rama:

```
(defrule rompe-not-not
   ?f1 <- (rama $?ini (not (not ?p)) $?fin)
   =>
   (retract ?f1)
   (assert (rama $?ini ?p $?fin)))
```

Las reglas que afectan a las fórmulas $\neg\top$ y $\neg\bot$ son expresadas de manera similar a la regla anterior.

La regla de expansión de tableros que afecta a una rama con una fórmula α introduce α_1 y α_2 como nuevas fórmulas en la rama. La expresión en ART de esta regla para fórmulas de la forma $X \wedge Y$ es la siguiente:

```
(defrule rompe-and
   ?f1 <- (rama $?ini (and ?p ?q) $?fin)
   =>
   (retract ?f1)
   (assert (rama $?ini ?p ?q $?fin)))
```

El resto de las fórmulas de tipo α tienen asociadas reglas similares a la anterior.

La regla de expansión de tableros que afecta a una rama con una fórmula β introduce dos nuevos hijos etiquetados con β_1 y β_2, esto supone una ramificación del tablero actual y por tanto que el resultado se expresa en ART utilizando dos nuevos hechos, uno por cada rama recién creada. La expresión en ART de esta regla para fórmulas de la forma $X \vee Y$ es la siguiente:

```
(defrule rompe-or
   ?f1 <- (rama $?ini (or ?p ?q) $?fin)
   =>
   (retract ?f1)
   (assert (rama $?ini ?p $?fin)
           (rama $?ini ?q $?fin)))
```

El resto de las fórmulas de tipo β tienen asociadas reglas similares a la anterior.

Una vez que una rama de un tablero se cierra, al aparecer en ella la constante $\bot$ o la pareja de fórmulas X y $\neg X$, no es necesario seguir expandiendo dicha rama, la acción correspondiente en ART será la de eliminar el hecho que representaba dicha rama:

```
(defrule elimina-tablero-1
   ?f1 <- (rama $?ini F $?fin)
   =>
   (retract ?f1))

(defrule elimina-tablero-2
   ?f1 <- (rama $?ini ?p $?cen (not ?p)
               $?fin)
   =>
   (retract ?f1))

(defrule elimina-tablero-3
   ?f1 <- (rama $?ini (not ?p) $?cen ?p
               $?fin)
   =>
   (retract ?f1))
```

Si una vez terminado el proceso de expandir el tablero se han eliminado todos los hechos que representaban las ramas, entonces se puede concluir que la fórmula original es una tautología:

```
(defrule reconoce-tautologia
   (not (rama $?))
   ?f1 <- (formula-original ?for)
   =>
   (retract ?f1)
   (printout t "La formula:" t ?for t
             " es una tautologia." t))
```

3 Unificación

El proceso de unificación es fundamental en cualquier aplicación de la Inteligencia Artificial dentro de la lógica de primer orden o superior. Prueba de ello es la gran cantidad de trabajos que existen sobre dicho tema ([6]). En esencia, se trata de establecer valores para las variables que aparecen en dos términos de la lógica, de forma que al sustituir las ocurrencias de dichas

variables por los valores determinados, se obtenga como resultado dos nuevos términos iguales dentro de la lógica. Nos ocupamos aquí del caso más simple de unificación, la sintáctica en lógica de primer orden, en la que los términos resultado han de ser sintácticamente iguales.

3.1 Términos

En lo sucesivo consideraremos un conjunto de variables V y un conjunto de símbolos de función F. Cada elemento f de F tendrá asociado un número natural al que llamaremos aridad de f. A los elementos de F de aridad 0 los llamaremos constantes.

Definición. 8 *El conjunto de términos en F sobre V, notado $\mathcal{T}(F,V)$, es el menor conjunto tal que:*

1. $V \subseteq \mathcal{T}(F,V)$
2. *Si* $t_1,\ldots,t_n \in \mathcal{T}(F,V)$ *y* $f \in F$ *de aridad* n *entonces* $f(t_1,\ldots,t_n) \in \mathcal{T}(F,V)$.

Llamaremos término a cualquier elemento de $\mathcal{T}(F,V)$.

Definición. 9 *El conjunto de variables de un término t, notado $\mathcal{V}(t)$, es el conjunto definido recursivamente como sigue:*

- *Si* $t \in V$, *entonces* $\mathcal{V}(t) = t$.
- *Si* $t = f(t_1,\ldots,t_n)$ *con* $f \in F$ *de aridad* n *y* $t_1,\ldots,t_n \in \mathcal{T}(F,V)$, *entonces* $\mathcal{V}(t) = \mathcal{V}(t_1) \bigcup \ldots \bigcup \mathcal{V}(t_n)$.

Definición. 10 *El conjunto de subtérminos de un término t, notado $Sub(t)$, es el conjunto definido recursivamente como sigue:*

- *Si* $t \in V$, *entonces* $Sub(t) = \{t\}$.
- *Si* $t = f(t_1,\ldots,t_n)$ *con* $f \in F$ *de aridad* n *y* $t_1,\ldots,t_n \in \mathcal{T}(F,V)$, *entonces* $Sub(t) = Sub(t_1) \bigcup \ldots \bigcup Sub(t_n) \bigcup \{t\}$.

Si $s \in Sub(t)$, *decimos que s es subtérmino de t. Si s es subtérmino de t y* $s \neq t$, *decimos que s es subtérmino propio de t.*

Definición. 11 *Una sustitución es una aplicación σ del conjunto de las variables V en el conjunto de los términos $\mathcal{T}(F,V)$. El resultado de aplicar una sustitución σ a un término t es otro término en el que toda* $x \in \mathcal{V}(\sqcup)$, *ha sido sustituida por* $\sigma(x)$. *Notaremos como* $\{x \mapsto s\}$ *a la sustitución que a la variable x le hace corresponder el término s y no afecta a las restantes variables.*

Los términos pueden ser representados de muy diversas formas, dos de las más comunes son la basada en árboles, en la que un término se representa como un árbol, y la basada en grafos acíclicos dirigidos, en este caso los términos se representan como grafos etiquetados, dirigidos y ordenados (Esto sirve para distinguir entre términos como $f(x,y)$ y $f(y,x)$).

Dado $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, un grafo G que represente a t será aquel que verifique:

1. Para todo $x \in \mathcal{V}(t)$, existe en G un único nodo etiquetado por "x" al que llamaremos nodo variable y que será el nodo principal del subtérmino x de t.
2. Para todo $f(t_1,\ldots,t_n)$ subtérmino de t, hay en G un nodo etiquetado por "f" del cual salen exactamente n arcos enumerados hacia los nodos principales de los grafos asociados a $t_1,\ldots,t_n$. A este nodo lo llamamos nodo funcional y es el nodo principal de $f(t_1,\ldots,t_n)$.

Diremos que un grafo acíclico dirigido es minimal si no existe otro grafo acíclico dirigido con menos nodos que represente al mismo término.

3.2 Unificación sintáctica

Un problema de unificación en $\mathcal{T}(F,V)$ es un conjunto finito de pares no ordenados de términos, $\{\{s_1,t_1\},\ldots,\{s_n,t_n\}\}$, que notaremos de la forma $S = \{s_1 =^? t_1,\ldots,s_n =^? t_n\}$, en el que se busca una instancia de las variables de los términos implicados, de modo que al sustituir en los términos iniciales las variables por las instancias encontradas, los resultados sean sintácticamente iguales.

Diremos que dos problemas de unificación, S_1 y S_2, son equivalentes, si poseen el mismo conjunto de soluciones. De esta forma, para encontrar la solución de un problema de unificación, iremos reduciendo el mismo a otro equivalente de más fácil solución. Estas reducciones se realizan mediante la aplicación de un conjunto de

reglas, que preservan el conjunto de soluciones del problema de unificación al que se aplican y que conducen a problemas de unificación más simples (Una discusión más detallada se puede encontrar en [5]).

Un conjunto de reglas con esas propiedades es el siguiente:

BORRA:

$$S \cup \{t =^? t\} \Longrightarrow S$$

FALLA:

$$S \cup \{f(t_1, \ldots, t_m) =^? g(u_1, \ldots, u_n)\} \Longrightarrow \{F\}$$
si $f \neq g$ ó $f = g$ y $m \neq n$

DESCOMPONE:

$$S \cup \{f(t_1, \ldots, t_n) =^? f(u_1, \ldots, u_n)\} \Longrightarrow$$
$$\Longrightarrow S \cup \{t_1 =^? u_1, \ldots, t_n =^? u_n\}$$

ELIMINA:

$$S \cup \{x =^? t\} \Longrightarrow \{x \mapsto t\}(S) \cup \{x =^? t\}$$
si $x \in \mathcal{V}(S) - \mathcal{V}(t)$ y $(t \in X \rightarrow t \in \mathcal{V}(S))$

CHEQUEA:

$$S \cup \{x =^? t\} \Longrightarrow \{F\} \text{ si } x \in \mathcal{V}(t) \text{ y } x \neq t$$

De este conjunto de reglas, CHEQUEA detecta la existencia de ciclos dentro de los términos unificados. Dentro de un algoritmo de unificación en concreto, este proceso se puede aplazar para el final, cuando ninguna de las otras reglas sea aplicable.

En [8] se analizan con mayor profundidad este conjunto de reglas como algoritmo de unificación y distintas implementaciones del mismo.

3.3 Implementación en CLIPS

Para representar un término en CLIPS utilizaremos un hecho por cada uno de los subtérminos que éste tenga. Para mantener la estructura de grafo acíclico dirigido minimal no se repetirán hechos para subtérminos idénticos. Todos los subtérminos estarán identificados mediante un índice. Usaremos un hecho de la forma `(termino ?n ?x)` para representar los nodos variables, donde `?n` es el índice del subtérmino y `?x` la variable. Para representar los nodos funcionales utilizaremos hechos de la forma `(termino ?n ?f $?l)` donde `?n` es el índice del subtérmino, `?f` es el símbolo de función del subtérmino y `$?l` es la lista de índices de los subtérminos argumentos.

Con objeto de almacenar el valor por el que se ha de sustituir una variable y distinguir éstas de las constantes (a partir de un hecho de la forma `(termino ?n ?x)` no se puede determinar si `?x` es una variable o una constante), utilizamos hechos de la forma `(variable ?x $?v)`, donde `?x` es la variable y `$?v` es el valor asociado. Elegimos una variable múltiple para éste último, pues puede ser la lista vacía (si no se ha asignado valor a `?x`) o un índice de un término.

Para mantener la estructura de grafo acíclico dirigido minimal, necesitaremos un par de reglas que modifiquen la estructura cuando se detecten subtérminos iguales. La primera regla detecta la existencia de subtérminos iguales de índices `?i` y `?j`, donde el segundo índice es mayor que el primero:

```
(defrule subterminos-iguales
  (termino ?i ?s $?lst)
  ?h <- (termino ?j&:(> ?j ?i) ?s $?lst)
  =>
  (retract ?h)
  (assert (cambia ?j ?i)))
```

La segunda regla cambia todas las referencias al término de mayor índice por referencias al término de menor índice:

```
(defrule modifica-estrutura
  (cambia ?x ?y)
  ?h <- (termino ?i ?s $?ini ?x $?fin)
  =>
  (retract ?h)
  (assert (termino ?i ?s $?ini ?y $?fin)))
```

Consideremos el siguiente problema de unificación $S = \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$. Para representarlo en CLIPS usaremos un conjunto de hechos, uno por cada par $s_i =^? t_i$, de la forma `(unifica ?n ?m)`, donde `?n` es el índice asociado al término `s_i` y `?m` es el índice asociado al término `t_i`.

Veamos primero las reglas de unificación sin considerar la detección de ciclos (este proceso se puede aplazar hasta que ninguna de las otras reglas sea aplicable). Cada una de las reglas de unificación vistas en la subsección anterior se transforma en una o dos reglas dentro del sistema CLIPS.

La regla BORRA se traslada de manera inmediata a CLIPS:

```
(defrule borra
  ?h <- (unifica ?i ?i)
  =>
  (retract ?h))
```

La regla FALLA se ha de dividir en dos dentro de CLIPS, la primera de ellas detecta la intención de unificar dos términos funcionales con distinto símbolo funcional e indica que la unificación no es posible:

```
(defrule falla-simbolo
  ?h <- (unifica ?i ?j&~?i)
  (termino ?i ?s $?)
  (termino ?j ?t&~?s $?)
  (not (variable ?s $?))
  (not (variable ?t $?))
  =>
  (retract *)
  (printout t "Unificacion imposible" t))
```

La segunda regla CLIPS asociada a FALLA detecta la intención de unificar dos términos funcionales con el mismo símbolo funcional pero distinta aridad e indica que la unificación no es posible:

```
(defrule falla-lista
  ?h <- (unifica ?i ?j&~?i)
  (termino ?i ?s $?lst1)
  (termino ?j ?s $?lst2)
  (test (not (= (length$ $?lst1)
                (length$ $?lst2))))
  =>
  (retract *)
  (printout t "Unificacion imposible" t))
```

La regla DESCOMPONE genera nuevos problemas de unificación para cada uno de los argumentos de los dos términos funcionales a unificar. Dado que los términos iniciales se consideran unificados, se cambian todas las referencias al de mayor índice por referencias al de menor índice y se actualizan las estructuras de grafo acíclico dirigido minimal:

```
(defrule descompone
  ?h <- (unifica ?i ?j&:(> ?j ?i))
  (termino ?i ?s $?lst1)
  (termino ?j ?s $?lst2)
  (test (= (length$ $?lst1)
           (length$ $?lst2)))
  =>
  (retract ?h)
  (assert (cambia ?j ?i))
  (loop-for-count
   (?indice 1 (length$ ?lst1))
   (assert (unifica (nth$ ?indice ?lst1)
                    (nth$ ?indice ?lst2)))))
```

La última acción de la regla anterior es un bucle sobre los argumentos de los términos originales, que introduce los subproblemas de unificación que se derivan.

Finalmente la regla ELIMINA asigna un valor a una variable, cambiando todas las referencias al índice asociado al término variable por el índice asociado al otro término. Esta regla difiere un poco de la presentada en la subsección 2.2 ya que en este momento no se comprueba la existencia de ciclos. El valor asignado a la variable es almacenado en un hecho de la forma `(variable ?s ?j)` donde `?s` es la variable en cuestión y `?j` es el índice del término por el que se ha de sustituir:

```
(defrule elimina
  ?h <- (unifica ?i ?j&~?i)
  (termino ?i ?s)
  ?k <- (variable ?s)
  (termino ?j ?t $?lst)
  =>
  (retract ?h ?k)
  (assert (variable ?s ?j))
  (assert (cambia ?i ?j)))
```

Esta regla refleja el caso en el que el problema de unificación es $x =^? t$. Para problemas de unificación con la forma $t =^? x$ habría que construir otra regla distinta.

La inclusión de los hechos `(unifica ?i ?j)` y `(variable ?s ?i)`, donde `?i` y `?j` son índices de subtérminos, hace necesario considerar reglas que modifiquen dichos valores si se ha producido algún cambio en la estructura de los términos, estas reglas son similares a `modifica-estrutura`.

Una vez llegado a este punto se comprueba la posible existencia de ciclos entre los términos asignados a las variables originales del problema, se trata de la regla CHEQUEA. Para detectar ciclos en los términos se utiliza el concepto de subtérmino propio: existirá un ciclo si, y sólo si, podemos encontrar un término que sea subtérmino propio de sí mismo. Para ello necesitaremos tres reglas.

La primera de ellas establece que todos los argumentos de un término funcional son subtérminos propios de dicho término:

```
(defrule argumentos-subterminos-propios
  (termino ?i ? $? ?j $?)
  =>
  (assert (sub-propio ?j ?i)))
```

La segunda regla establece el carácter transitivo de la relación "ser subtérmino propio":

```
(defrule subtermino-propio-transitiva
  (sub-propio ?j ?i)
  (sub-propio ?k ?j)
  =>
  (assert (sub-propio ?k ?i)))
```

Finalmente, si un término es subtérmino propio de sí mismo habremos encontrado un ciclo:

```
(defrule detecta-ciclos
  (sub-propio ?i ?i)
  =>
  (printout t
      "CICLO: Unificacion imposible" t)
  (retract *))
```

4 Conclusiones

El uso de los sistemas de producción como herramientas de programación es novedoso y existe poca literatura al respecto ([7] y [2]). Como argumentos en favor de su uso destacamos la facilidad con que se pueden codificar gran cantidad de algoritmos, de los cuales presentamos dos ejemplos. Esta facilidad no sólo se debe al hecho de que estos algoritmos se expresen como la aplicación de una serie de reglas, sino a que cada una de las reglas que se utiliza para su implementación recoge un aspecto particular del algoritmo que, aislado del resto, resulta mucho más comprensible, como ocurre en algoritmos de carácter recursivo.

Otro argumento en favor de los sistemas de producción es el carácter no determinista que parece dirigir la selección de la regla que se ha de aplicar en cada momento. Por supuesto se sigue un criterio que puede ser modificado por el usuario, pero no es necesario tenerlo en cuenta a la hora de realizar un programa.

Este "carácter" no determinista de los sistemas de producción da lugar a dos de las principales dificultades de la programación en los mismos. En primer lugar, hay que evitar que la base de conocimiento haga que el sistema se pierda en bucles, en los que se ejecutan siempre las mismas reglas. En segundo lugar, algunas veces es necesario ejecutar las reglas agrupadas en varios conjuntos, de manera que no se mezclen en la ejecución reglas de un conjunto con reglas de otro. Para solventar estas dificultades es necesario utilizar otras funcionalidades de estos sistemas que no hemos citado aquí, como son el uso de prioridades y la descomposición del programa en módulos.

Hemos presentado aquí implementaciones en sistemas basados en reglas de producción de los tableros semánticos proposicionales y de un algoritmo de unificación sintáctica, resultados de gran importancia dentro del campo de la Inteligencia Artificial. Esto es punto de partida para realizar otras aplicaciones dentro de este campo. Una posibilidad es llegar a construir herramientas de razonamiento automático dentro de sistemas de producción como ART o CLIPS.

El sistema CLIPS se utiliza en varias asignaturas impartidas por el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. Con él se han obtenido soluciones a una gran cantidad de problemas de búsqueda en espacios de estado, implementado algoritmos recursivos y basados en reglas y desarrollado juegos con heurísticas. Más información se puede encontrar en [1].

Referencias

[1] José A. Alonso, Delia Balbotín, Francisco J. Martín, y José L. Ruiz. *Inteligencia artificial II (curso 97-98). Universidad de Sevilla.* `http://www-cs.us.es/~jalonso/ia2`.

[2] Lee Brownston. *Programming expert systems in OPS5 : an introduction to rule-based programming.* Addison-Wesley, Reading, Mass., 1986.

[3] Melvin Fitting. *First-Order Logic and Automated Theorem Proving otra cosa.* Springer-Verlag New York Inc., 1996.

[4] Joseph Giarratano y Gary Riley. *Expert Systems. Principles and Programming.* PWS Publishing Company, 1993.

[5] J.-P. Jouannaud y Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. *Jean-Louis Lassez & Gordon Plotkin (Eds.), Computational Logic: Essays in Honor of Alan Robinson, The MIT Press, 1991*, 1991.

[6] Claude Kirchner (ed.). *Unification.* Academic Press Inc., 1990.

[7] Thaddeus J. Kowalski y Leon S. Levy. *Rule-based programming.* Kluwer Academic Publishers, 1996.

[8] Francisco J. Martín. Problemas de unificación: Unificación sintáctica y e-unificación. Tesina de licenciatura, Universidad de Sevilla, 1995.