# Generalización de fórmulas lógicas y su aplicación al aprendizaje automático

A. Moreno

Report LSI–92–28–R

# Generalización de fórmulas lógicas y su aplicación al aprendizaje automático

Antonio Moreno Ribas

Verano, 1992

Memoria de Licenciatura para obtener el grado de Licenciado en Informática por la Facultat d'Informàtica de Barcelona de la Universitat Politècnica de Catalunya.

Director: Ulises Cortés García

[1]Parcialmente financiado por el proyecto **TIC-90 801/C02. CICYT.**

# Índice

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

## 1.1 Motivación del trabajo realizado

La motivación inicial de este trabajo es desarrollar un **algoritmo de generalización** de expresiones lógicas. Este algoritmo de generalización es uno de los parámetros del algoritmo de **aprendizaje basado en similitudes** (SBL) propuesto por Gustavo Núñez en su tesis doctoral [**NÚÑE91**]. El algoritmo que propone surge de una caracterización no monótona de la inferencia inductiva (el proceso de inducción) y, concretamente, usa la circunscripción de predicados ([**McCA80**]) para formalizar el razonamiento no monótono.

Una vez desarrollado este algoritmo de generalización se vió la posibilidad de usarlo para construir un programa que fuera capaz de estructurar todos los conceptos que generaliza de forma jerárquica, explorando las relaciones de los unos con los otros y organizando los conceptos en un grafo según las relaciones de inclusión existentes entre ellos. Así surgió el **constructor de jerarquías**.

Tanto el algoritmo de generalización como el constructor de jerarquías se han probado en uno de los dominios más clásicos dentro de la Inteligencia Artificial: el del mundo de los bloques definido por Winston [**WINS75**] (aunque ambos algoritmos están escritos de manera que se puedan usar en cualquier dominio que tenga ciertas características que se describen más adelante). Los algoritmos han sido escritos en **Common Lisp**.

## 1.2 Estructura de la memoria

Esta memoria está organizada de la siguiente forma:

- **La generalización dentro del aprendizaje automático**

  En este capítulo se describe el marco teórico donde se sitúa el algoritmo de generalización. Se explica la importancia del **aprendizaje automático** (*Machine Learning*)

dentro de la Inteligencia Artificial, así como los principales paradigmas identificados hasta el momento en este campo. Se hace especial énfasis en el **aprendizaje inductivo**, que es el más relacionado con nuestro trabajo, explicando en qué se basa y los diferentes subtipos en los que se puede dividir, llegando así al **aprendizaje basado en similitudes**. En este punto se describe el algoritmo de aprendizaje propuesto por Núñez **[NÚÑE91]** y se ve como el algoritmo de generalización es uno de los parámetros de este algoritmo de aprendizaje.

- **Algoritmo de generalización**

  Se describe el proceso de implementación del algoritmo de generalización, realizado a base de prototipos de complejidad creciente (desde la versión inicial que sólo trabajaba con un atributo por objeto y en el dominio del mundo de los bloques hasta la versión final que puede trabajar con cualquier número de atributos y en diferentes dominios). Se ilustra la explicación de las diferentes versiones con ejemplos del mundo de los bloques.

  En cada paso se examinan las limitaciones de las diferentes versiones del algoritmo y se intenta paliar alguna de ellas en la versión posterior. Se va aumentando el poder expresivo del lenguaje de representación de escenas (permitiendo cualquier número de atributos, aumentando el número de relaciones conocidas por el programa), se mejora la eficiencia del algoritmo con el uso de una heurística que reduce el número de posibles generalizaciones calculadas por éste y también se modifica para aumentar el número de dominios en el que es aplicable.

- **Comparación con métodos anteriores**

  En este capítulo se describen algunos de los algoritmos de generalización más conocidos desarrollados dentro del campo del aprendizaje a partir de ejemplos, como **[VERE75]**, **[WINS75]**, **[HAYE77]** o **[DIET81]** y se usa el mismo ejemplo que en **[DIET83]** para compararlos con el algoritmo de generalización descrito en el capítulo anterior. También se usan otros ejemplos mostrados por algunos de estos autores en sus artículos, como un ejemplo con bloques de Hayes-Roth y McDermott (**[HAYE78]**) o el conocido ejemplo de los trenes de Michalski en **[MICH80]**.

- **Constructor de jerarquías**

  Se describe el funcionamiento del constructor de jerarquías de conceptos a partir de las descripciones generalizadas que obtenemos con el algoritmo anterior. Básicamente lo que hace es organizar las generalizaciones en un grafo acíclico dirigido que expresa las relaciones de inclusión entre los diversos conceptos. También se modifican las expresiones generalizadas que se le suministran al algoritmo de forma que reflejen la existencia de objetos *simples* dentro de los objetos más *complejos*. También se muestran ejemplos para ilustrar su funcionamiento.

- **Conclusiones. Trabajo futuro**

  Se hace un breve resumen del trabajo realizado y se indican varios puntos en donde éste podría ser extendido o mejorado.

- **Bibliografía**

  Se mencionan los textos usados durante la realización del algoritmo y otros a los que se puede acudir para ampliar algunos de los puntos que aparecen en esta memoria.

- **Apéndice A**

  En este apéndice se puede encontrar descripciones de ejemplos usados por el algoritmo de generalización y resultados de alguna de las pruebas realizadas con el mismo.

- **Apéndice B**

  Para finalizar la memoria se incluye el código **Vax Common Lisp** tanto del algoritmo de generalización como del constructor de jerarquías.

# Capítulo 2

# La generalización dentro del aprendizaje automático

## 2.1 Introducción

Uno de los campos de la Inteligencia Artificial que más interés ha despertado en los últimos años es el del **aprendizaje automático** (**Machine Learning**). Es evidente que una de las características humanas que demuestran de una forma más clara su inteligencia es la capacidad de aprender. Cuando se trabaja con un sistema informático, por el contrario, en muchas ocasiones se pone rápidamente de manifiesto su incapacidad de aprendizaje. Estos sistemas, por lo general, no son capaces de mejorar con la experiencia, obtener conocimiento del dominio sobre el que estén trabajando por medio de experimentos, mejorar sus algoritmos automáticamente, formular nuevas abstracciones, proponer una solución a un problema por analogía con la solución dada previamente a un problema semejante, etc.. Se podría decir que el aprendizaje automático intenta que los ordenadores sean capaces de:

- Adquirir nuevos conocimientos o nuevas habilidades.
- Reorganizar el conocimiento que tienen de forma que puedan usarlo posteriormente para resolver los problemas que se les planteen de la forma más eficiente posible.

Dentro del aprendizaje automático siempre hay *alguien* que proporciona información al algoritmo de aprendizaje (el entorno a través de sensores, un experto en un dominio determinado). Éste ha de modificar los datos que se le ofrecen de forma conveniente, para que luego pueda acceder a ellos cuando los necesite para resolver un problema. Según la transformación sobre los datos que haya de realizar se han identificado diversos paradigmas de aprendizaje. Esta clasificación ha ido evolucionando rápidamente en la última década. Una de las versiones más recientes ([**ZHON92**]) identifica los siguientes paradigmas:

- **Aprendizaje inductivo**

  Este tipo de aprendizaje está relacionado directamente con el trabajo desarrollado en esta tesina, y a él se le dedica especial atención en secciones posteriores de este capítulo. Es el paradigma más estudiado dentro del aprendizaje automático. Trata problemas como inducir la descripción de un concepto a partir de una serie de ejemplos y *contraejemplos* del mismo o determinar una descripción taxonómica (clasificación) de un grupo de objetos (p.e. **[DIET81]**, **[DIET83]**, **[BÉJA92]**).

- **Aprendizaje analógico**

  Este tipo de aprendizaje intenta emular algunas de las capacidades humanas más sorprendentes: poder entender una situación por su parecido con situaciones anteriores conocidas, poder crear y entender metáforas o poder resolver un problema notando su posible *semejanza* con problemas vistos anteriormente y adaptando de forma conveniente la solución que se encontró para esos problemas (p.e. **[WINS82]**, **[CREI88]**).

- **Aprendizaje analítico**

  Los métodos usados en este tipo de aprendizaje intentan formular generalizaciones después de analizar algunas instancias en términos del conocimiento del sistema. En contraste con las técnicas empíricas de aprendizaje, que normalmente son métodos basados en las similitudes, el aprendizaje analítico requiere que se le proporcione al sistema un amplio conocimiento del dominio. Este conocimiento es usado para guiar las cadenas deductivas que se utilizan para resolver nuevos problemas. Por tanto, estos métodos se centran en mejorar la eficiencia del sistema, y no en obtener nuevas descripciones de conceptos como el aprendizaje inductivo (p.e. **[MITC86]**).

- **Aprendizaje genético**

  Los algoritmos genéticos están inspirados en las mutaciones que ocurren durante la reproducción biológica y en la selección natural de Darwin. Los problemas principales que trata de resolver son el descubrimiento de reglas y la asignación de crédito a las mismas. Este último punto consiste en valorar positiva o negativamente las reglas según lo útiles que le sean al sistema. Esta valoración será la que determine qué regla aplicar para resolver un problema determinado (p.e. **[HOLL75]**, **[DAVI87]**).

- **Aprendizaje conexionista**

  Estos sistemas de aprendizaje también se llaman *redes neuronales*, en clara referencia a la neurofisiología humana. Normalmente el aprendizaje consiste en el ajuste de los pesos de las conexiones existentes entre varios elementos (*neuronas*) en una red con una topología fija. Al ajustar estos pesos cambiará la respuesta de la red delante de cualquier entrada, y se dice que el sistema *aprende* si llega a unos pesos de las conexiones que le lleven a dar la salida correcta ante todas (o la mayoría) de las entradas que se le ofrezcan (se pueden encontrar muchas referencias sobre este tipo de aprendizaje en **[MORE92a]**).

## 2.2 Aprendizaje inductivo

El proceso de aprendizaje inductivo consiste en la adquisición de nuevo conocimiento después de realizar inferencia inductiva (inducción) sobre los datos proporcionados por el entorno o por un maestro. Este proceso se puede caracterizar ([**NILS80**], [**MITC82**], [**MICH83**]) como una búsqueda heurística en un espacio de estados, donde:

- Los estados son descripciones simbólicas de mayor o menor generalidad. El estado inicial son los datos de entrada.
- Los operadores son reglas de inferencia, fundamentalmente *reglas de generalización* (pasan de una descripción simbólica a otra más general) y *reglas de especialización* (transforman una descripción en otra más particular).
- El estado final es una aserción inductiva que implica los datos de entrada, satisface el conocimiento de respaldo del problema y maximiza el criterio de preferencia que se aplique para valorar la *calidad* de las descripciones encontradas. Por conocimiento de respaldo (*background knowledge*) se entiende el conocimiento que tiene el programa sobre el problema que está tratando de solucionar.

Las reglas de generalización que se usan son de dos tipos ([**MICH83**]):

- **Reglas de selección**

  Son aquellas reglas en las que todos los descriptores que aparecen en la expresión generalizada ya estaban presentes en las descripciones iniciales del concepto. Las más habituales son:

  - **Supresión de condiciones**
    Consiste en eliminar un elemento dentro de una conjunción, obteniendo de esta forma una expresión más general (a & *b* es más específico que a).
  - **Adición de condiciones**
    Consiste en añadir un elemento dentro de una disjunción (a $\vee$ *b* es más general que a).
  - **Cerrar intervalos**
    Si se tienen dos descripciones de la misma clase que difieren en el valor de un sólo descriptor lineal, se pueden reemplazar por una única descripción en la cual la referencia del descriptor sea el intervalo entre estos dos valores. Por ejemplo, si en una descripción se tiene *valor=2* y en otra *valor=7*, se pueden generalizar a *valor en el intervalo 2..7*.
  - **Cambio de constantes por variables**
    Consiste en substituir alguna de las constantes que aparezcan en la descripción de un concepto por una variable cuantificada universalmente, obteniendo así una expresión más general.

- **Subir el árbol de generalización**

  Un atributo de tipo estructural es aquel cuyo dominio se puede representar de forma jerárquica (árbol de generalización). Si hay varias descripciones en las que un atributo de tipo estructural tiene diferentes valores, se pueden generalizar a una descripción en la que ese atributo tenga como valor el nodo más bajo del árbol de generalización que sea antecesor de esos valores. Por ejemplo, *forma=cuadrado* y *forma=rectángulo* se pueden generalizar a *forma=polígono.*

- **Reglas constructivas**

  Estas reglas generan aserciones inductivas que contienen descriptores que no existían en las descripciones originales. Las más habituales consisten en contar los argumentos de un predicado o generar propiedades en una cadena (objetos al principio, al final, en una posición determinada de la cadena). Por ejemplo, en el mundo de los bloques se pueden generar descripciones en las que aparezca el predicado **(TOP x)** -que indica que **x** no tiene ningún objeto por encima- a partir de descripciones en las que sólo aparezca la relación **ON**, relación binaria que indica que un objeto está sobre otro.

### 2.2.1 Tipos de aprendizaje inductivo

Se pueden distinguir [**MICH83**] dos grandes tipos de aprendizaje inductivo:

- **Aprendizaje a partir de ejemplos**

  También se conoce como *adquisición de conceptos.* Se caracteriza porque hay un *profesor* que proporciona al programa la descripción de algunos objetos, ya clasificados en una o más clases (*conceptos*). La hipótesis que se induce puede ser vista como una regla de reconocimiento del concepto. Esto significa que si un objeto satisface las condiciones de la regla entonces representa al concepto dado.

  Algunos problemas tratados en este tipo de aprendizaje son:

  - Aprender la descripción característica de una clase de objetos, que especifica las propiedades comunes a todos los objetos conocidos de la clase (p.e. [**WINS70**], [**HAYE78**]).
  - Aprender la descripción discriminante de una clase de objetos, que la distingue de un número limitado de clases diferentes (p.e. [**MICH80**]).
  - Inferir reglas de extrapolación a partir de secuencias, capaces de predecir el siguiente elemento de una secuencia dada (p.e. [**DIET79**]).

- **Aprendizaje a partir de la observación**

  También es conocido como *generalización descriptiva.* Su objetivo es determinar una descripción general que caracterize un conjunto de observaciones.

Algunos ejemplos de este tipo de aprendizaje son:

- Formular una teoría que caracterize un conjunto de elementos (p.e. **[LENA83]**).
- Descubrir patrones en datos (p.e. **[LANG83]**).
- Determinar una descripción taxonómica (clasificación) de una colección de objetos (p.e. **[MART91]**, **[BÉJA92]**).

El aprendizaje basado en similitudes se ocupa de la modelización de los procesos que conducen a la adquisición de conocimiento a partir de hechos específicos, y el tema de la *adquisición de conceptos* es uno de los más ampliamente estudiados bajo este paradigma.

## 2.3 Un algoritmo de adquisición de conceptos

En su tesis doctoral Núñez, ([**NÚÑE91**]), propone una nueva aproximación a la formalización de la inducción, caracterizándola como una forma de razonamiento no monótono. La componente no monótona implícita dentro del proceso de inferencia inductiva es clara, ya que un nuevo ejemplo (o contraejemplo) de un concepto puede invalidar una descripción de ese concepto obtenida anteriormente tras generalizar varios ejemplos. Esto se puede ver con el ejemplo presentado en la figura 2.1.

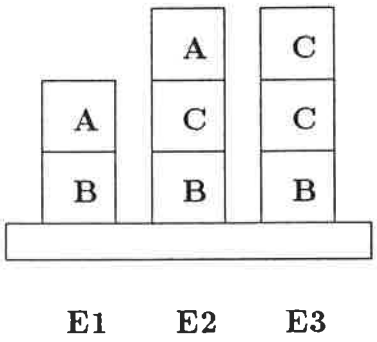



Figura 2.1: Ejemplo para ilustrar la no monotonía de la inducción

Si en un principio sólo se conocen los dos primeros ejemplos del concepto, la descripción del mismo que se obtendría sería *"Hay un cuadrado de color B encima de la mesa y que siempre tiene por encima un cuadrado de color A, que no tiene ningún otro bloque por encima"*. Al considerar el tercer ejemplo de la figura 2.1 la generalización previa ya no sería válida, y la descripción pasaría ser *"Hay un cuadrado de color B encima de la mesa"*, que es lo único que tienen en común los tres ejemplos (aparte de otras consideraciones como que todos los ejemplos son torres de cuadrados o que hay como mínimo dos bloques en cada ejemplo).

La formalización que propone Núñez puede ser usada ([**NÚÑE90**], [**NÚÑE91**]) para obtener una nueva aproximación al problema del aprendizaje inductivo de conceptos (adquisición de conceptos) en presencia de instancias positivas y negativas, caracterizándolo como un problema de inferencia no monótona. Esta aproximación no va a ser detallada aquí, pero la idea básica es maximizar la extensión de la función de reconocimiento $C$ del concepto objetivo, realizando la circunscripción de la base de datos disponible $\Delta$ - incluyendo el conocimiento de respaldo $\Gamma$ - respecto a la negación de $C$ [**McCA80**], para obtener una fórmula $F$ equivalente a $C$. Simbólicamente:

$$CIRC[\Gamma \cup \Delta; \neg C] \models \forall x(F(x) \longleftrightarrow C(x))$$

Mediante esta fórmula $F$ se expresa formalmente la idea de que las únicas instancias negativas del concepto objetivo son las que se deducen de la información existente en $\Gamma \cup \Delta$.

Las instancias positivas y negativas del concepto se expresan como conjunciones de literales básicos y el conocimiento de respaldo se expresa como un conjunto de fórmulas de primer orden. El método que propone Núñez no se basa en una exploración del conjunto de hipótesis compatibles con los datos, sino que construye la solución a partir de la información disponible. Este enfoque permite obtener -para un mismo conjunto de instancias positivas y negativas- funciones de reconocimiento del concepto de distinto nivel de generalidad, en función del conocimiento de respaldo que se considere. Con el formalismo de aprendizaje que presenta se obtiene una visión formal unificada de diversos métodos clásicos de aprendizaje de conceptos, como los de Vere [**VERE75**] y Winston [**WINS70**].

El algoritmo de aprendizaje de conceptos que propone Núñez maneja tres casos, según el conocimiento de respaldo que se considere:

1. El conocimiento de respaldo es vacío.
2. Las similitudes entre las instancias positivas son condiciones necesarias para la caracterización del concepto.
3. Existe un subconjunto de relaciones comunes a todas las instancias positivas, cuya conjunción es una condición necesaria y suficiente para la caracterización del concepto. En este caso las descripciones de las instancias positivas deben tener, como mínimo, una fórmula, $P_e$, que exprese la *posición estable*, o punto de referencia del objeto (u objetos), a partir del cual se construye su descripción [**CORT84**].

El algoritmo de aprendizaje de conceptos que presenta es el siguiente:

- **Entradas:** instancias positivas ($pe_1$, $pe_2$, ..., $pe_n$) y negativas ($ne_1$, $ne_2$, ..., $ne_m$)
- **Parámetros:** conocimiento de respaldo ($\Gamma$), algoritmo de cotejamiento (matching) (**Am**), algoritmo de reducción de fórmulas lógicas (**Arf**)

- **Salida:** Descripción del concepto (D)

1. Aplicar **Am** a $((pe_1, pe_2, ..., pe_n), (ne_1, ne_2, ..., ne_m))$ para obtener las descripciones generalizadas $gpe_i$, $gne_j$ de las instancias positivas y negativas.

2. Si el conocimiento de respaldo es vacío (caso 1), asignar a $F_e$ la fórmula $T$ ($\equiv$ tautología) e ir al paso 4. En caso contrario, ir al paso 3.

3. Aplicar **Am** para determinar una generalización maximal de los ejemplos positivos y asignársela a $F_e$.

   - Si el conocimiento de respaldo es el correspondiente al caso 2, eliminar las instancias negativas que no satisfagan $F_e$. Ir al paso 4.
   - Si el conocimiento de respaldo es el correspondiente al caso 3, eliminar las instancias negativas que satisfagan $F_e$. Para cada instancia negativa restante, determinar el conjunto de literales $L_{j_1}, ..., L_{j_{m_j}}$ de $F_e$ que no aparezcan en su descripción $ne_j$. Aplicar **Arf** a la expresión

     $$\wedge_{j\epsilon J}(\vee_{k=1}^{m_j} L_{j_k})$$

     y asignar a D la fórmula

     $$D \longleftarrow P_e \wedge \textbf{Arf}\ (\wedge_{j\epsilon J}(\vee_{k=1}^{m_j} L_{j_k}))$$

4. Aplicar el algoritmo **Arf** a la fórmula

   $$\neg(\vee_{j\epsilon J}\ gcne_j)$$

   y asignar a D la fórmula

   $$D \longleftarrow F_e \wedge \textbf{Arf}\ (\neg(\vee_{j\epsilon J}\ gcne_j))$$

El algoritmo de generalización que se describe en el siguiente capítulo es una posible implementación del algoritmo de cotejamiento **Am** usado como parámetro dentro de este algoritmo de aprendizaje de conceptos. Este algoritmo de generalización **Am** se usa en dos pasos del algoritmo de aprendizaje:

- En el paso 1 para obtener las *descripciones generalizadas* de las instancias positivas y negativas del concepto objetivo.
- En el paso 3, al que se llega si el conocimiento de respaldo $\Gamma$ no es vacío, donde ha de calcular una *generalización maximal* de los ejemplos positivos.

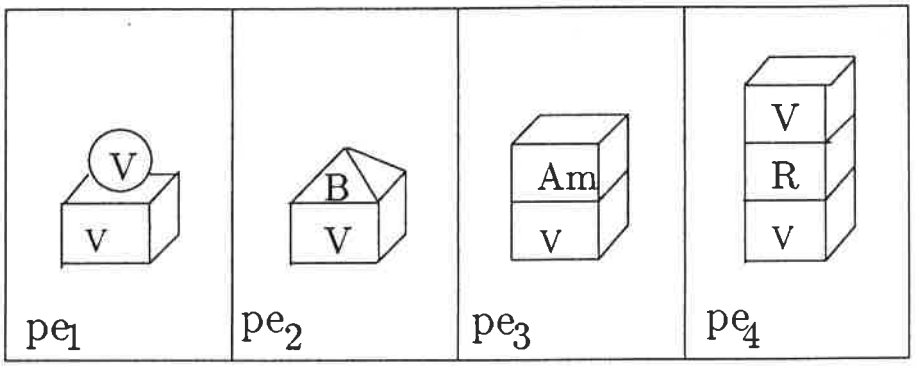



Figura 2.2: Instancias positivas del concepto objetivo

En el algoritmo que se ha desarrollado la obtención de las descripciones generalizadas es un paso intermedio en el cálculo de la generalización maximal de los ejemplos. En el siguiente capítulo se describe extensamente el proceso que se sigue hasta llegar a esta generalización maximal. Para ver qué se espera del algoritmo de generalización se puede seguir el ejemplo que usa Núñez para ilustrar el algoritmo. En la figura 2.2 se pueden ver las instancias positivas del concepto a aprender, y en la figura 2.3 están las instancias negativas.

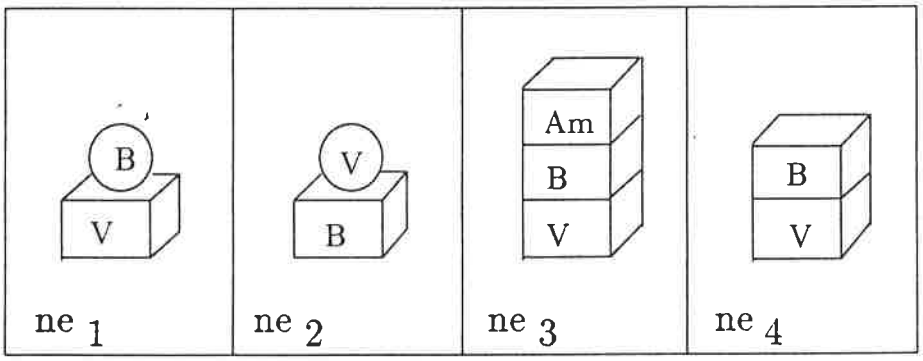



Figura 2.3: Instancias negativas del concepto objetivo

En estas figuras y a lo largo de este ejemplo se usan las siguientes abreviaciones para los predicados que se indican,

Sobre (**S**) Esfera (**E**) Cubo (**C**) Pirámide (**P**)

Amarillo (**Am**) Rojo (**R**) Verde (**V**) Azul (**B**)

Las instancias positivas y negativas se representan de la siguiente manera:

- $pe_1 = \mathbf{S}(o_2, M) \wedge \mathbf{S}(o_1, o_2) \wedge \mathbf{C}(o_2) \wedge \mathbf{V}(o_2) \wedge \mathbf{E}(o_1) \wedge \mathbf{V}(o_1)$
- $pe_2 = \mathbf{S}(o_4, M) \wedge \mathbf{S}(o_3, o_4) \wedge \mathbf{C}(o_4) \wedge \mathbf{V}(o_4) \wedge \mathbf{P}(o_3) \wedge \mathbf{B}(o_3)$
- $pe_3 = \mathbf{S}(o_6, M) \wedge \mathbf{S}(o_5, o_6) \wedge \mathbf{C}(o_6) \wedge \mathbf{V}(o_6) \wedge \mathbf{C}(o_5) \wedge \mathbf{Am}(o_5)$
- $pe_4 = \mathbf{S}(o_8, M) \wedge \mathbf{S}(o_7, o_8) \wedge \mathbf{C}(o_8) \wedge \mathbf{V}(o_8) \wedge \mathbf{S}(o_9, o_7) \wedge \mathbf{C}(o_7) \wedge \mathbf{C}(o_9) \wedge \mathbf{R}(o_7) \wedge \mathbf{V}(o_9)$
- $ne_1 = \mathbf{S}(p_2, M) \wedge \mathbf{S}(p_1, p_2) \wedge \mathbf{E}(p_1) \wedge \mathbf{C}(p_2) \wedge \mathbf{V}(p_2) \wedge \mathbf{B}(p_1)$
- $ne_2 = \mathbf{S}(p_4, M) \wedge \mathbf{S}(p_3, p_4) \wedge \mathbf{E}(p_3) \wedge \mathbf{C}(p_4) \wedge \mathbf{V}(p_3) \wedge \mathbf{B}(p_4)$
- $ne_3 = \mathbf{S}(p_6, M) \wedge \mathbf{S}(p_5, p_6) \wedge \mathbf{S}(p_7, p_5) \wedge \mathbf{C}(p_7) \wedge \mathbf{C}(p_5) \wedge \mathbf{C}(p_6) \wedge \mathbf{Am}(p_7) \wedge \mathbf{V}(p_6) \wedge \mathbf{B}(p_5)$
- $ne_4 = \mathbf{S}(p_9, M) \wedge \mathbf{S}(p_8, p_9) \wedge \mathbf{C}(p_8) \wedge \mathbf{C}(p_9) \wedge \mathbf{V}(p_9) \wedge \mathbf{B}(p_8)$

En el primer paso del algoritmo se obtienen las *descripciones generalizadas* tanto de las instancias positivas como de las negativas. En este ejemplo serían:

- $gpe_1 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{E}(x) \wedge \mathbf{V}(x)$
- $gpe_2 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{P}(x) \wedge \mathbf{B}(x)$
- $gpe_3 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{C}(x) \wedge \mathbf{Am}(x)$
- $gpe_4 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{S}(z, x) \wedge \mathbf{C}(x) \wedge \mathbf{C}(z) \wedge \mathbf{R}(x) \wedge \mathbf{V}(z)$
- $gne_1 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{E}(x) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{B}(x)$
- $gne_2 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{E}(x) \wedge \mathbf{C}(y) \wedge \mathbf{V}(x) \wedge \mathbf{B}(y)$
- $gne_3 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{S}(z, x) \wedge \mathbf{C}(z) \wedge \mathbf{C}(x) \wedge \mathbf{C}(y) \wedge \mathbf{Am}(z) \wedge \mathbf{B}(x) \wedge \mathbf{V}(y)$
- $gne_4 = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(x) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y) \wedge \mathbf{B}(x)$

En el capítulo siguiente se detalla cómo se obtienen estas descripciones generalizadas. Básicamente, el proceso que hay que seguir es identificar los objetos de los diversos ejemplos que tengan características *semejantes* y substituirlos por una misma variable.

En el tercer paso del algoritmo de aprendizaje se calcula la *generalización conjuntiva maximal* de los ejemplos positivos, si el conocimiento de respaldo no es vacío. En este ejemplo el resultado es:

$$F_e = \mathbf{S}(y, M) \wedge \mathbf{S}(x, y) \wedge \mathbf{C}(y) \wedge \mathbf{V}(y)$$

Esta expresión es la conjunción más larga contenida en todas las descripciones generalizadas de las instancias positivas. Se podría parafrasear como *"Hay un cubo verde encima de la mesa que está soportando algún otro objeto"*.

Obviamente, la dificultad estriba en encontrar la substitución de objetos por variables que conduzca a la generalización conjuntiva maximal. Si en la segunda instancia positiva se hubiese substituido el objeto $o_3$ por la variable $y$ y el objeto $o_4$ por la variable $x$, la generalización conjuntiva maximal hubiese sido nula, ya que no habría ningún conjuntando común a todas las descripciones generalizadas.

El algoritmo de generalización que se presenta en el capítulo tres es capaz de construir descripciones teniendo en cuenta las características que al usuario le parezcan importantes. Por ejemplo, al presentar las instancias de la figura 2.2 podría ser relevante la forma de los objetos pero no su color. Las generalizaciones de las instancias positivas que obtiene el algoritmo son:

- **Considerando las características FORMA y COLOR**

  $\mathbf{S}(x, M) \wedge \mathbf{V}(x) \wedge \mathbf{C}(x)$

  *Hay un cubo verde encima de la mesa*

- **Considerando sólo la característica FORMA**

  $\mathbf{S}(x, M) \wedge \mathbf{C}(x)$

  *Hay un cubo encima de la mesa*

- **Considerando sólo la característica COLOR**

  $\mathbf{S}(x, M) \wedge \mathbf{V}(x)$

  *Hay un objeto verde encima de la mesa*

- **Considerando la parte estructural de los ejemplos**

  $\mathbf{S}(x, M) \wedge \mathbf{C}(x) \wedge \mathbf{V}(x) \wedge \mathbf{S}(\mathbf{y}, \mathbf{x})$

  *Hay un cubo verde encima de la mesa que está soportando algún otro objeto*

En este ejemplo tan sencillo al considerar menos atributos lo que se obtiene es simplemente un subconjunto de la descripción calculada considerando más atributos, pero éste no es el caso general, como se verá en ejemplos mostrados en los capítulos 3 y 4. La última generalización es exactamente la obtenida en **[NÚÑE91], p. 121**.

# Capítulo 3

# Descripción del algoritmo de generalización

## 3.1 Introducción

La suposición subyacente en los métodos de adquisición de conceptos es que las similitudes entre las instancias positivas del concepto a aprender contienen información suficiente para la construcción de descripciones características del concepto. Estas descripciones se representan típicamente como conjunciones de las relaciones primitivas utilizadas para expresar dichas instancias. Las descripciones que obtenemos con el algoritmo de cotejamiento desarrollado son las más específicas, es decir, las conjunciones con el mayor número de relaciones presentes en la representación de las instancias del concepto. Estas descripciones son las **generalizaciones conjuntivas maximales** [MITC82] que se quieren obtener en el paso 3 del algoritmo de aprendizaje propuesto en [NÚÑE91], visto anteriormente.

Podemos expresar de un modo formal el concepto de **generalización conjuntiva maximal** de la siguiente forma:

- Sean $E_1$, $E_2$ dos fórmulas de primer orden expresadas en forma normal conjuntiva. Se dice que $E_1$ es más general que $E_2$ si existe una substitución $\sigma$ de términos de $E_2$ por variables tal que el conjunto de cláusulas de $E_1$ es un subconjunto del de $\sigma E_2$ (donde $\sigma E_2$ es la fórmula resultante de aplicar la substitución $\sigma$ a $E_2$). Para que una fórmula E sea una generalización conjuntiva maximal de $n$ fórmulas $E_1$, $E_2$, ..., $E_n$ se debe cumplir que:
  - E es una generalización de $E_i$, $\forall_{i=1}^{n}$.
  - Si F es una generalización de $E_i$, $\forall_{i=1}^{n}$, entonces E no es una generalización de F.

De esta definición se deduce que para obtener el conjunto de todas las descripciones conjuntivas maximales (lo que equivale a todas las maneras posibles de extraer similitudes

de los ejemplos) se han de explorar todas las formas posibles de cotejamiento de estructuras (**pattern matching**) entre los argumentos de cada literal común a las instancias positivas, lo que genera serios problemas de complejidad. En el algoritmo de generalización que se ha desarrollado se trata este problema de la complejidad, y se introducen en algunos puntos heurísticas que, a pesar de que no aseguren encontrar la mejor generalización (la más específica) en todos los casos, en la mayoría de ellos reducirán considerablemente el esfuerzo computacional sin que se elimine la generalización que se está buscando.

## 3.2 Evolución del algoritmo

El algoritmo de generalización se desarrolló a base de prototipos, empezando con versiones que hicieran tareas relativamente sencillas hasta obtener un algoritmo complejo. Este algoritmo (al igual que el constructor de jerarquías) se implementó en **Common Lisp**, usando átomos para representar objetos, listas de propiedades para almacenar los valores de los atributos de los objetos y listas para representar las relaciones entre objetos.

Un ejemplo de cómo se representaría un concepto del mundo de los bloques es el siguiente:

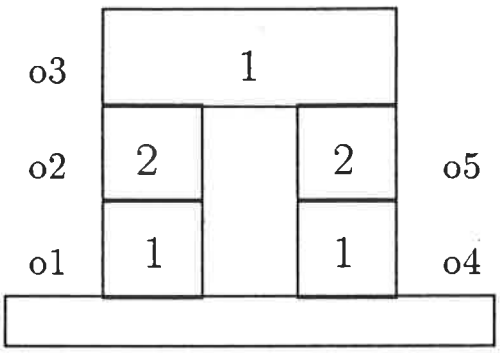



Figura 3.1: Arco con 5 objetos

- Representación del ejemplo de la figura 3.1
  - **Relaciones: ((on o1 mesa) (on o2 o1) (left* o1 o4) (left* o2 o5) (on o3 o2) (on o3 o5) (on o5 o4) (on o4 mesa))**
  - **Atributos**
    - **o1 ((color a) (peso 1) (forma cuad))**
    - **o2 ((color b) (peso 2) (forma cuad))**
    - **o3 ((color a) (peso 1) (forma rect))**
    - **o4 ((color b) (peso 2) (forma cuad))**
    - **o5 ((color a) (peso 1) (forma cuad))**

### 3.2.1 Primera versión

Las primeras versiones del algoritmo generalizador estaban diseñadas explícitamente para el dominio del **mundo de los bloques**, dominio clásico en la Inteligencia Artificial **[WINS75]**. La primera versión sólo trataba torres de objetos (los objetos de una escena sólo podían estar relacionados mediante la relación **ON**, para señalar que un objeto está justo encima de otro), y los objetos sólo podían tener un atributo (p.e. color).

El proceso que sigue este algoritmo para llegar a la expresión generalizada del concepto es el siguiente:

1. Creación de todas las posibilidades de **substitución de objetos por variables** (usando la misma variable para objetos que tuvieran el mismo valor en el atributo considerado).
2. **Cálculo del factor común** (expresión generalizada) asociado a cada una de estas posibilidades.
3. **Selección de la mejor solución** (la más específica, o sea, el factor común con más elementos).

La fase 1 es la más importante. Se basa en la **Hipótesis de Generalidad**, que se enuncia en **[NÚÑE91]** de la forma siguiente:

- *Las constantes específicas en que son instanciadas las relaciones utilizadas para describir los ejemplos no son importantes para la caracterización del concepto, lo que importa son las relaciones que satisfacen dichas constantes*

Esta hipótesis equivale a suponer que los diferentes símbolos de constante que aparecen en la descripción de cada ejemplo del concepto pueden ser substituidos por variables distintas en cualquier algoritmo de cotejamiento de patrones que se use. Pero, evidentemente, este algoritmo de cotejamiento de patrones ha de evitar la introducción innecesaria de variables y extraer de forma conveniente las características comunes de los ejemplos. Esto se hace substituyendo por variables diferentes sólo los objetos que tengan valores diferentes en el atributo que se considera, y usando la misma variable con aquellos objetos que tengan el mismo valor en el atributo.

Un aspecto a resaltar de esta primera versión es que en la expresión generalizada pueden aparecer 3 relaciones (**ON** - un objeto justamente encima de otro -, **SOBRE** - un objeto por encima, quizás sin tocarse, de otro - y **TOP** - un objeto que no tiene ningún otro por encima -) , mientras que en la representación de los conceptos sólo se usaba la relación **ON**.

Este mecanismo de **substitución de constantes por variables** según el valor del atributo hace que se pueda llegar a varias generalizaciones distintas a partir de los mismos ejemplos, incluso en esta versión donde sólo se consideran torres y un atributo. Se puede comprobar este hecho en los siguientes ejemplos.

- **Ejemplo 1**

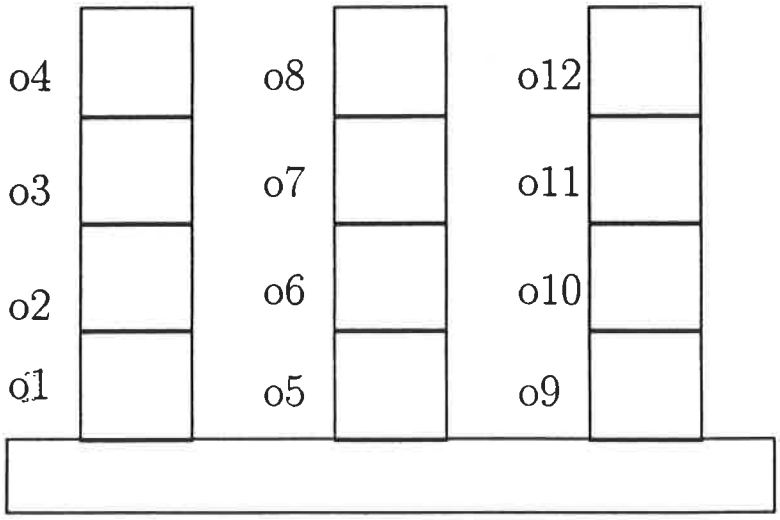



Figura 3.2: Primer ejemplo de la primera versión

Representación de los ejemplos:

- **Ej. 1: ((on o1 mesa) (on o2 o1) (on o3 o2) (on o4 o3))**
- **Ej. 2: ((on o5 mesa) (on o6 o5) (on o7 o6) (on o8 o7))**
- **Ej. 3: ((on o9 mesa) (on o10 o9) (on o11 o10) (on o12 o11))**

Valor del atributo color:

- **Color a**: o1, o3, o5, o6, o9, o12
- **Color b**: o2, o4, o7, o8, o10, o11

Combinaciones después de asignar variables:

```
((((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c1
  ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X4 X2) (ON X3 X4)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c2
  ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
  ((ON X1 MESA) (ON X2 X1) (ON X4 X2) (ON X3 X4)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c3
  ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
  ((ON X1 MESA) (ON X4 X1) (ON X2 X4) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c4
  ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
  ((ON X1 MESA) (ON X4 X1) (ON X2 X4) (ON X3 X2)))
```

```
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c5
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X1 MESA) (ON X2 X1) (ON X4 X2) (ON X3 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c6
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X1 MESA) (ON X2 X1) (ON X4 X2) (ON X3 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c7
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X1 MESA) (ON X4 X1) (ON X2 X4) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c8
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X1 MESA) (ON X4 X1) (ON X2 X4) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c9
 ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
 ((ON X3 MESA) (ON X2 X3) (ON X4 X2) (ON X1 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c10
 ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
 ((ON X3 MESA) (ON X2 X3) (ON X4 X2) (ON X1 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c11
 ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
 ((ON X3 MESA) (ON X4 X3) (ON X2 X4) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c12
 ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
 ((ON X3 MESA) (ON X4 X3) (ON X2 X4) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c13
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X3 MESA) (ON X2 X3) (ON X4 X2) (ON X1 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c14
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X3 MESA) (ON X2 X3) (ON X4 X2) (ON X1 X4)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c15
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X3 MESA) (ON X4 X3) (ON X2 X4) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c16
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X3 MESA) (ON X4 X3) (ON X2 X4) (ON X1 X2))))
```

A partir de estas posibilidades de substitución de objetos por variables se obtienen las siguientes generalizaciones (escritas en el mismo orden que la posibilidad de la que provienen):

```
(((ON X1 MESA) (SOBRE X2 X1))  ;;; g1
 ((ON X1 MESA) (SOBRE X2 X1))  ;;; g1
```

```
((ON X1 MESA) (SOBRE X2 X1)) ;;; g1
((ON X1 MESA) (SOBRE X2 X1)) ;;; g1
((SOBRE X1 MESA) (ON X2 X1)) ;;; g2
((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
((SOBRE X4 X3)) ;;; g4
((SOBRE X4 X3)) ;;; g4
((SOBRE X4 X3)) ;;; g4
((ON X4 X3)) ;;; g5
((SOBRE X1 MESA) (SOBRE X4 X3)) ;;; g6
((SOBRE X1 MESA) (SOBRE X4 X3)) ;;; g6
((SOBRE X1 MESA) (SOBRE X4 X3)) ;;; g6
((SOBRE X1 MESA) (SOBRE X4 X3))) ;;; g6
```

Valores del atributo color en las variables:

```
X1: color a
X2: color b
X3: color a
X4: color b
```

Como se puede ver, en este caso existen 16 posibilidades de substitución de objetos por variables, que dan lugar a 6 generalizaciones distintas.

Por ejemplo, la posibilidad de substitución número 5 es:

```
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X1 MESA) (ON X2 X1) (ON X4 X2) (ON X3 X4)))
```

Esta asociación de objetos de diferentes instancias a través de las variables se puede representar gráficamente tal como se ve en la figura 3.3 en la página siguiente.

El algoritmo, en esta primera versión, reconoce el patrón mostrado en la figura con línea contínua (**x2** siempre está justo encima de **x1**, que es un objeto por encima de la mesa). Las relaciones establecidas con las variables **x3** y **x4** (mostradas en la figura con líneas discontínuas) no le permiten encontrar ningún patrón más que sea común a las tres instancias del concepto objetivo.

Las seis generalizaciones obtenidas se pueden parafrasear de la siguiente forma:

1. **((ON X1 MESA) (SOBRE X2 X1))** aparece 4 veces
   Hay un objeto de **color a** encima de la mesa que tiene en alguna posición por encima suyo un objeto de **color b**.

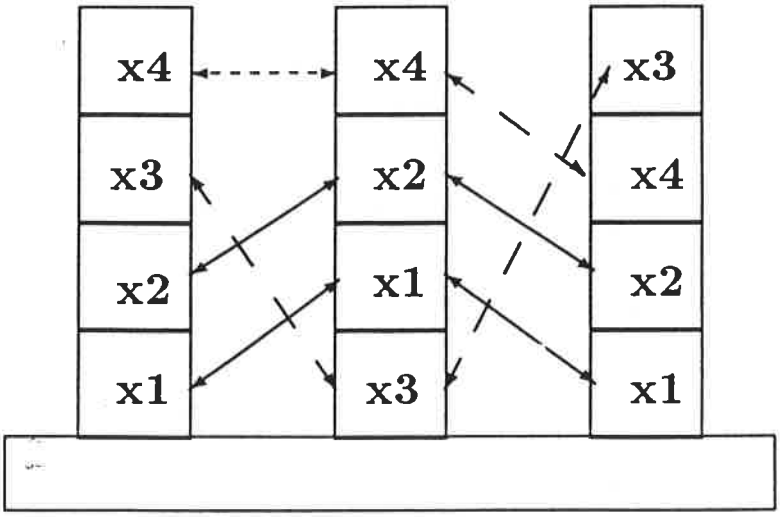



Figura 3.3: Asociación de objetos en una substitución

2. **((SOBRE X1 MESA) (ON X2 X1))** aparece 1 vez
   Hay un objeto de **color a** en algún punto por encima de la mesa que tiene un objeto de **color b** justo encima de él.
3. **((SOBRE X1 MESA) (SOBRE X2 X1))** aparece 3 veces
   Hay un objeto de **color a** en algún punto por encima de la mesa, y tiene un objeto de **color b** en algún punto por encima suyo.
4. **((SOBRE X4 X3))** aparece 3 veces
   Hay un objeto de **color b** en algún punto por encima de un objeto de **color a**.
5. **((ON X4 X3))** aparece 1 vez
   Hay un objeto de **color b** justo encima de un objeto de **color a**.
6. **((SOBRE X1 MESA) (SOBRE X4 X3))** aparece 4 veces
   Hay un objeto de **color a** por encima de la mesa, y tambien hay un objeto de **color b** que está por encima de otro objeto de **color a**.

- **Ejemplo 2**

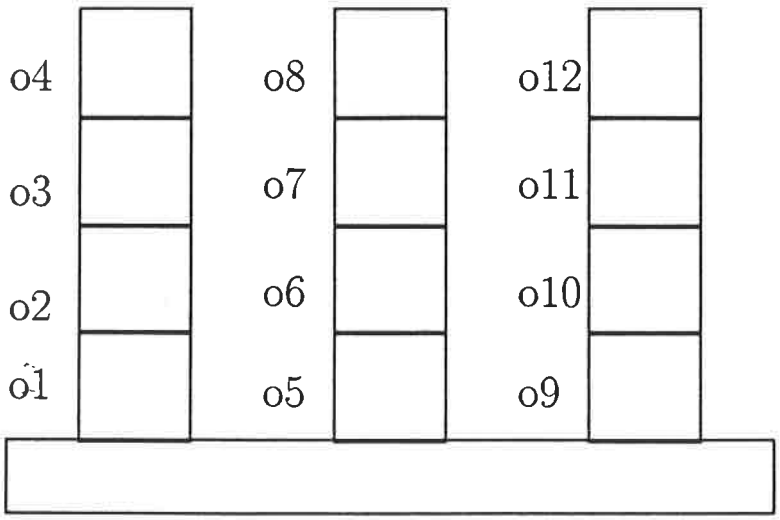



Figura 3.4: Segundo ejemplo de la primera versión

Representación de los ejemplos:

- **Ej. 1: ((on o1 mesa) (on o2 o1) (on o3 o2) (on o4 o3))**
- **Ej. 2: ((on o5 mesa) (on o6 o5) (on o7 o6) (on o8 o7))**
- **Ej. 3: ((on o9 mesa) (on o10 o9) (on o11 o10) (on o12 o11))**

Valor del atributo color:

- **Color a**: o1, o3, o5, o6, o10, o11
- **Color b**: o2, o4, o7, o8, o9, o12

Combinaciones después de asignar variables:

```
((((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c1
  ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
  ((ON X2 MESA) (ON X1 X2) (ON X3 X1) (ON X4 X3)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c2
  ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
  ((ON X2 MESA) (ON X1 X2) (ON X3 X1) (ON X4 X3)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c3
  ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
  ((ON X4 MESA) (ON X1 X4) (ON X3 X1) (ON X2 X3)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c4
  ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
  ((ON X4 MESA) (ON X1 X4) (ON X3 X1) (ON X2 X3)))
```

```
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c5
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X2 MESA) (ON X1 X2) (ON X3 X1) (ON X4 X3)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c6
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X2 MESA) (ON X1 X2) (ON X3 X1) (ON X4 X3)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c7
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X4 MESA) (ON X1 X4) (ON X3 X1) (ON X2 X3)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c8
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X4 MESA) (ON X1 X4) (ON X3 X1) (ON X2 X3)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c9
 ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
 ((ON X2 MESA) (ON X3 X2) (ON X1 X3) (ON X4 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c10
 ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
 ((ON X2 MESA) (ON X3 X2) (ON X1 X3) (ON X4 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c11
 ((ON X1 MESA) (ON X3 X1) (ON X2 X3) (ON X4 X2))
 ((ON X4 MESA) (ON X3 X4) (ON X1 X3) (ON X2 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c12
 ((ON X1 MESA) (ON X3 X1) (ON X4 X3) (ON X2 X4))
 ((ON X4 MESA) (ON X3 X4) (ON X1 X3) (ON X2 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c13
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X2 MESA) (ON X3 X2) (ON X1 X3) (ON X4 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c14
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X2 MESA) (ON X3 X2) (ON X1 X3) (ON X4 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c15
 ((ON X3 MESA) (ON X1 X3) (ON X2 X1) (ON X4 X2))
 ((ON X4 MESA) (ON X3 X4) (ON X1 X3) (ON X2 X1)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON X4 X3))  ;;; c16
 ((ON X3 MESA) (ON X1 X3) (ON X4 X1) (ON X2 X4))
 ((ON X4 MESA) (ON X3 X4) (ON X1 X3) (ON X2 X1))))
```

Generalización obtenida para cada posibilidad:

```
(((SOBRE X1 MESA) (SOBRE X4 X3) (TOP X4))  ;;; g1
 ((SOBRE X1 MESA) (ON X4 X3)) ;;; g2
 ((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
 ((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
```

```
((SOBRE X1 MESA) (SOBRE X4 X3) (TOP X4))  ;;; g1
((SOBRE X1 MESA) (SOBRE X4 X3)) ;;; g4
((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
((SOBRE X1 MESA) (SOBRE X2 X1)) ;;; g3
((SOBRE X4 X3) (TOP X4)) ;;; g5
((SOBRE X4 X3)) ;;; g6
((SOBRE X2 X1)) ;;; g6
((SOBRE X2 X1)) ;;; g6
((SOBRE X1 MESA) (SOBRE X4 X3) (TOP X4)) ;;; g1
((SOBRE X1 MESA) (SOBRE X4 X3)) ;;; g4
((SOBRE X1 MESA) (ON X2 X1))  ;;; g7
((SOBRE X1 MESA) (SOBRE X2 X1))) ;;; g3
```

Valores del atributo color en las variables:

```
X1: color a
X2: color b
X3: color a
X4: color b
```

En este segundo ejemplo se obtienen 7 generalizaciones diferentes de entre las 16 posibilidades existentes, y son las siguientes:

1. **((SOBRE X1 MESA) (SOBRE X4 X3) (TOP X4))** aparece 3 veces
   Hay un objeto de **color a** en algún punto por encima de la mesa, y además hay un objeto de **color b** por encima de otro objeto de **color a**, y este objeto de **color b** no tiene ningún otro objeto por encima.
2. **((SOBRE X1 MESA) (ON X4 X3))** aparece 1 vez
   Hay un objeto de **color a** en algún lugar por encima de la mesa, y hay otro objeto de **color a** que tiene justo encima un objeto de **color b**.
3. **((SOBRE X1 MESA) (SOBRE X2 X1))** aparece 5 veces
   Hay un objeto de **color a** por encima de la mesa, y hay otro objeto de **color b** que está por encima de este objeto de **color a**.
4. **((SOBRE X1 MESA) (SOBRE X4 X3))** aparece 2 veces
   Hay un objeto de **color a** por encima de la mesa, y hay otro objeto de **color a** que tiene en algún lugar por encima suyo un objeto de **color b**.
5. **((SOBRE X4 X3) (TOP X4))** aparece 1 vez
   Hay un objeto de **color a** que tiene por encima un objeto de **color b**, que no está cubierto por ningún otro objeto.

6. **((SOBRE X4 X3))** aparece 1 vez
   **((SOBRE X2 X1))** aparece 2 veces
   Hay un objeto de **color b** por encima de un objeto de **color a**.
7. **((SOBRE X1 MESA) (ON X2 X1))** aparece 1 vez
   Hay un objeto de **color a** por encima de la mesa, y tiene justo encima un objeto de **color b**.

Obviamente esta versión tiene toda una serie de limitaciones bastante importantes:

1. Sólo sirve para un dominio determinado (bloques), y además este dominio está reducido (sólo existe la relación **ON**, es decir, sólo se pueden definir torres de objetos).
2. Sólo se considera 1 atributo por objeto.
3. No es muy eficiente: calcula el factor común para todas las posibilidades de substitución de objetos por variables.

### 3.2.2 Segunda versión

En la segunda versión se eliminó la segunda de las limitaciones que se acaban de mencionar, ya que se generalizó el algoritmo para que pudiera tratar un número cualquiera de atributos, **aumentando así la expresividad del lenguaje de representación de escenas.**

Este cambio de uno a varios atributos es importante, porque obviamente la generalización obtenida a partir de una serie de ejemplos depende de los atributos que se están considerando para hacerla (ya que si varían los atributos considerados, varía la asociación objetos / variables que se hace en el paso 1).

Concretamente lo que hace el programa es buscar la generalización usando primero todos los atributos; la muestra al usuario y si a éste no le satisface, busca la generalización considerando todos los atributos menos uno (los tiene en una lista sin ningún orden especial; simplemente deja de tener en cuenta el que estaba en primer lugar de esta lista). Se repite este proceso hasta llegar a alguna generalización que le parezca adecuada al usuario o hasta que se acaben los atributos (el generalizador no puede trabajar sin atributos, ya que su fase más importante - la substitución de objetos por variables - depende de los valores de los atributos de los objetos que se están considerando en ese momento). Evidentemente, cuando se considera un número mayor de atributos se están poniendo más condiciones para poder substituir dos objetos distintos por la misma variable, y el número de posibilidades de substitución de constantes por variables que encontrará el algoritmo será siempre menor o igual al que se tendrá cuando se considera un atributo menos.

Una manera más adecuada de tener en cuenta los atributos probablemente hubiera sido mostrar al usuario todos los atributos y preguntarle cuáles quiere usar en una generalización dada, o tener alguna manera de medir la importancia de los atributos para ordenarlos e ir eliminando los menos importantes.

Vamos a ver con un ejemplo cómo varían las generalizaciones obtenidas al considerar atributos diferentes.

- **Ejemplo 3**

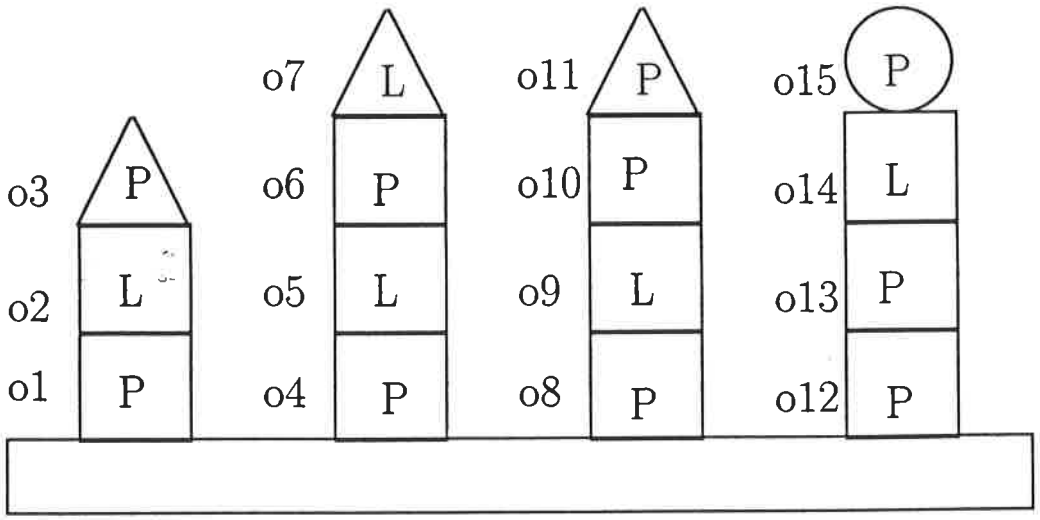



Figura 3.5: Ejemplo de la segunda versión

Representación de los ejemplos:

- **Ej. 1: ((on o1 mesa) (on o2 o1) (on o3 o2))**
- **Ej. 2: ((on o4 mesa) (on o5 o4) (on o6 o5) (on o7 o6))**
- **Ej. 3: ((on o8 mesa) (on o9 o8) (on o10 o9) (on o11 o10))**
- **Ej. 4: ((on o12 mesa) (on o13 o12) (on o14 o13) (on o15 o14))**

Valores de los atributos (forma, peso y color):

```
o1: cuadrado, pesado, a
o2: cuadrado, ligero, b
o3: triangulo, pesado, a
o4: cuadrado, pesado, a
o5: cuadrado, ligero, b
o6: cuadrado, pesado, b
o7: triangulo, ligero, a
o8: cuadrado, pesado, a
o9: cuadrado, ligero, b
o10: cuadrado, pesado, a
o11: triangulo, pesado, a
o12: cuadrado, pesado, a
o13: cuadrado, pesado, b
o14: cuadrado, ligero, b
o15: circulo, pesado, a
```

Vamos a ver las generalizaciones que se obtienen considerando diferentes conjuntos de atributos.

1. **Considerando los atributos PESO, FORMA y COLOR**

   Combinaciones después de asignar variables:

```
((((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c1
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON O7 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON O15 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c2
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON O7 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON O15 X2))))
```

   Generalización obtenida para cada posibilidad:

```
(((ON X1 MESA) (SOBRE X2 X1)) ;;; g1
 NIL)
```

   Valores de los atributos en las variables:

```
X1: peso pesado, forma cuadrado, color a
X2: peso ligero, forma cuadrado, color b
X3: peso pesado, forma triangulo, color a
```

Considerando los tres atributos, sólo hay dos posibilidades diferentes de substituir objetos por variables, que a su vez sólo dan lugar a una generalización diferente de NIL, que se puede interpretar así:

(a) **((ON X1 MESA) (SOBRE X2 X1))**
Hay un objeto **cuadrado**, **pesado** y de **color a** justo encima de la mesa, y en algún punto por encima de él hay un objeto **cuadrado**, **ligero** y de **color b**.

Como se puede ver, al realizar las substituciones de objetos por variables hay algunos objetos que quedan sin substituir, como o6, o7, o10, o13 y o15 en la primera posibilidad. Esto sucede porque empezamos substituyendo cada objeto del primer ejemplo del concepto por una variable, y luego se asocia cada variable con un objeto de cada uno de los otros ejemplos que tenga los mismos valores en los atributos. Por tanto, no se usa nunca la misma variable para dos objetos diferentes de un mismo ejemplo. Una mejora que se ha incluído para reducir este proceso de substitución es ordenar los ejemplos de forma creciente según el número de objetos que aparecen en ellos; por lo tanto, sólo se genera un número de variables igual al número de objetos del primer ejemplo, que es el más pequeño. Esto se puede hacer sin que afecte negativamente al resultado de la generalización porque ésta sólo puede incluir relaciones que aparezcan en todos los ejemplos y, por tanto, el ejemplo más pequeño limita de forma natural la máxima longitud posible de las generalizaciones que se puede obtener. Otra razón posible para que un objeto que no está en el primer ejemplo se quede sin substituir es, obviamente, que no coincida en los valores de los atributos con ninguno de los objetos del primer ejemplo (por ejemplo o7 en este caso).

2. **Considerando los atributos FORMA y COLOR**

Combinaciones después de asignar variables:

```
((((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c1
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON O15 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c2
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON O15 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c3
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON O15 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c4
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
```

```
 ((ON X1 MESA) (ON 013 X1) (ON X2 013) (ON 015 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c5
 ((ON X1 MESA) (ON X2 X1) (ON 06 X2) (ON X3 06))
 ((ON 08 MESA) (ON X2 08) (ON X1 X2) (ON X3 X1))
 ((ON X1 MESA) (ON X2 X1) (ON 014 X2) (ON 015 014)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c6
 ((ON X1 MESA) (ON 05 X1) (ON X2 05) (ON X3 X2))
 ((ON 08 MESA) (ON X2 08) (ON X1 X2) (ON X3 X1))
 ((ON X1 MESA) (ON X2 X1) (ON 014 X2) (ON 015 014)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c7
 ((ON X1 MESA) (ON X2 X1) (ON 06 X2) (ON X3 06))
 ((ON 08 MESA) (ON X2 08) (ON X1 X2) (ON X3 X1))
 ((ON X1 MESA) (ON 013 X1) (ON X2 013) (ON 015 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))                    ;;; c8
 ((ON X1 MESA) (ON 05 X1) (ON X2 05) (ON X3 X2))
 ((ON 08 MESA) (ON X2 08) (ON X1 X2) (ON X3 X1))
 ((ON X1 MESA) (ON 013 X1) (ON X2 013) (ON 015 X2))))
```

Generalización obtenida para cada posibilidad:

```
(((ON X1 MESA) (ON X2 X1))  ;;; g1
 ((ON X1 MESA) (SOBRE X2 X1)) ;;; g2
 ((ON X1 MESA) (SOBRE X2 X1)) ;;; g2
 ((ON X1 MESA) (SOBRE X2 X1)) ;;; g2
 NIL
 NIL
 NIL
 NIL)
```

Valores de los atributos en las variables:

```
X1: forma cuadrado, color a
X2: forma cuadrado, color b
X3: forma triangulo, color a
```

Al pasar a considerar sólo los atributos `forma` y `color` se pasa a tener 8 posibilidades de substitución de objetos por variables (lógicamente más que en el caso anterior porque sólo se imponen 2 condiciones para cada objeto, cuando antes se usaban 3). De las generalizaciones que obtiene (una para cada posibilidad) vemos que hay 4 que no llevan a ninguna conclusión y que sólo hay 2 válidas, que son las siguientes:

(a) **((ON X1 MESA) (ON X2 X1))** aparece 1 vez
Hay un objeto **cuadrado** y de **color a** encima de la mesa que tiene justo encima un objeto **cuadrado** de **color b**.

(b) **((ON X1 MESA) (SOBRE X2 X1))** aparece 3 veces
Hay un objeto **cuadrado** y de **color a** encima de la mesa y tiene en algún punto por encima suyo un objeto **cuadrado** de **color b**.

3. **Considerando el atributo COLOR**

En este caso hay 96 combinaciones después de asignar variables, que pueden verse en el apéndice A, junto con la generalización asociada a cada posibilidad.

Al considerar sólo el atributo `color`, las posibilidades de substitución de objetos por variables aumentan de forma considerable, ya que en los 12 objetos del ejemplo sólo hay 2 colores diferentes. Es importante resaltar que en 80 de las 96 posibilidades no se puede llegar a ninguna generalización. De las 16 posibilidades restantes se obtienen 8 generalizaciones distintas, que son las siguientes:

(a) **((ON X1 MESA) (ON X2 X1) (SOBRE X3 X2))** aparece 1 vez
Hay un objeto de **color a** encima de la mesa. Este objeto tiene justo encima suyo un objeto de **color b**, que tiene en algún punto encima suyo otro objeto de **color a**.

(b) **((ON X1 MESA) (ON X2 X1) (SOBRE X3 X2) (TOP X3))** aparece 1 vez
Hay un objeto de **color a** encima de la mesa. Este objeto tiene justo encima suyo un objeto de **color b**, que tiene en algún punto encima suyo otro objeto de **color a**. Este último objeto no está cubierto por ningún otro.

(c) **((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2))** aparece 2 veces
Hay un objeto de **color a** encima de la mesa. En algún punto por encima suyo tiene un objeto de **color b**, que a su vez tiene en alguna parte por encima suyo un objeto de **color a**.

(d) **((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2) (TOP X3))** aparece 3 veces
Hay un objeto de **color a** encima de la mesa. En algún punto por encima suyo tiene un objeto de **color b**, que a su vez tiene en alguna parte por encima suyo un objeto de **color a**, que no está cubierto por ningun otro.

(e) **((ON X1 MESA) (SOBRE X2 X1) (ON X3 X2))** aparece 1 vez
Hay un objeto de **color a** encima de la mesa. En algún punto por encima suyo tiene un objeto de **color b**, que a su vez tiene justo encima suyo un objeto de **color a**.

(f) **((SOBRE X3 X2) (TOP X3))** aparece 4 veces
Hay un objeto de **color a** en alguna parte por encima de un objeto de **color b**, y este objeto de **color a** no tiene ningún otro objeto encima.

(g) **((SOBRE X3 X2))** aparece 3 veces
Hay un objeto de **color a** en alguna parte por encima de un objeto de **color b**.

(h) **((ON X3 X2))** aparece 1 vez

Hay un objeto de **color a** justo encima de un objeto de **color b**.

Como se ha comentado, nos interesa la generalización más específica, que será la que contenga más literales. En este caso sería la segunda o la cuarta generalización, que tienen 4 literales cada una.

### 3.2.3 Tercera versión

En la tercera versión se redujo la ineficiencia en tiempo y espacio que significaba el tener que encontrar la generalización asociada a cada una de las posibilidades de substitución de objetos por variables, para ver cual de ellas era la máxima. Un claro ejemplo de la necesidad de limitar el cálculo de generalizaciones es el que se acaba de ver en la segunda versión, cuando al considerar sólo el atributo `color` se tenían 96 posibilidades de substitución de objetos por variables y 80 de ellas no llevaban a ninguna generalización útil. Para ello se modificó el algoritmo, que pasó a tener las siguientes fases:

1. Creación de todas las posibilidades de substitución de objetos por variables, relacionando con la misma variable objetos que tuvieran los mismos valores en los atributos que se están considerando en ese momento (como en las dos primeras versiones del algoritmo).
2. Hacer una valoración de las posibilidades, que estime cuales son las que pueden conducir a generalizaciones interesantes y cuales no vale la pena considerar.
3. Cálculo del factor común de la posibilidad (o posibilidades) que haya obtenido una valoración más alta en el paso anterior.
4. Selección de la mejor generalización, es decir, el factor común máximo de los encontrados en el paso 3.

La valoración de las posibilidades que se hace en el paso 2 se efectúa a partir de la frecuencia de aparición de las relaciones en los ejemplos que forman cada posibilidad; lo que se quiere hacer aquí es construir una heurística que sea fácil de calcular y que permita estimar qué posibilidades son las más prometedoras, para no tener que buscar la generalización asociada a cada una de ellas. Lo que hace el algoritmo es aplicar la siguiente fórmula:

$$val(posib) = \sum_{c:clausula(posib)} (numero\ de\ apariciones\ de\ c\ en\ posib)^2$$

Por tanto, se da una valoración más alta a aquellas posibilidades donde haya cláusulas que aparecen varias veces en los ejemplos. Eso será un signo bastante claro de que se están encontrando patrones de variables iguales en los ejemplos y, por tanto, de que se está ante

una substitución de objetos por variables que detecta regularidades en los ejemplos del concepto.

Esta heurística es muy fácil de calcular, y hace que se gane mucha eficiencia temporal al pasar de buscar la generalización asociada a cada posibilidad a buscar la generalización sólo de la posibilidad o posibilidades teóricamente más prometedoras. No hay ningún resultado teórico que permita decir que al aplicar este sesgo siempre se va a encontrar la mejor generalización, pero la intuición indica que esta heurística debería funcionar bien en un porcentaje importante de las ocasiones que se pueden presentar.

En un segundo paso se modificó ligeramente esta heurística, de forma que sólo se calculara para aquellas posibilidades que tuvieran un número máximo de cláusulas que se repitieran en todos los ejemplos. Dicho de otro modo, si se está valorando las posibilidades y la mejor hasta ahora tenía 4 cláusulas que aparecían en todos los ejemplos del concepto, no se calculará la valoración de ninguna posibilidad que no tenga al menos 4 cláusulas que aparezcan en todos los ejemplos. De esta forma se reduce el número de veces en que se calcula la valoración, obteniendo si cabe más eficiencia.

Otra pequeña modificación incluída en esta versión fue añadir la relación **TOP** en la descripción de los ejemplos del concepto (lo que no se puede hacer es incluir la relación **SOBRE**, ya que ésta se obtiene a partir de la generalización de **ON** directamente). En versiones posteriores del algoritmo se volvió a eliminar la relación **TOP**, porque es claramente calculable (si interesa) a partir de las relaciones **ON** (se cumple **(TOP X)** si no existe ninguna variable **Y** tal que se verifique **(ON Y X)**).

Se puede usar el mismo ejemplo visto en la versión 2 para mostrar la utilidad de esta función heurística para estimar qué posibilidades de substitución de objetos por variables vale la pena generalizar y cuáles no llevarán (presumiblemente) a ninguna parte. Al aplicar esta función se obtienen los siguientes resultados (la numeración de las posibilidades y las generalizaciones se ha hecho siguiendo el mismo orden en que se han mostrado antes):

1. **Al considerar los atributos PESO, FORMA y COLOR**

| Posibilidades | Valoración | Generalización |
|---|---|---|
| 1 | 39 | g1 |
| 2 | 29 | nil |

Tabla 3.1: Resultados con los atributos peso, forma y color

En este caso la heurística funciona bien, ya que valora más la posibilidad que conducía a una generalización valida que aquella de la que no se podía extraer ninguna generalización.

2. **Al considerar los atributos FORMA y COLOR**

Recuérdese que en este caso había 16 posibilidades de substitución, de las que se obtenían 2 generalizaciones posibles. Había 4 posibilidades en las que no se obtenía

| Posibilidades | Valoración | Generalización |
|---|---|---|
| 1 | 49 | g1 |
| 2 | 45 | g2 |
| 3 | 43 | g2 |
| 4 | 41 | g2 |
| 5 | 37 | nil |
| 6 | 35 | nil |
| 7,8 | 33 | nil |

Tabla 3.2: Resultado de la valoración de las posibilidades

ningún resultado positivo al generalizar. El resultado de las valoraciones está en la tabla 3.2. En este caso g1 y g2 tenían la misma longitud (2 cláusulas), y cualquiera de las 2 hubiera podido ser escogida por el algoritmo como más específica. Notar que las 4 posibilidades de las que no se puede obtener ninguna generalización han quedado en las últimas posiciones usando esta heurística, que por tanto ha funcionado como se esperaba.

3. **Al considerar el atributo COLOR**

   Al considerar sólo un atributo se tenían 96 posibilidades, 80 de las cuales no eran de utilidad alguna. De las 16 restantes se podía llegar a 8 generalizaciones diferentes, donde las más largas eran g2 y g4 (4 cláusulas). El resultado que se obtiene al aplicar la heurística a las 96 posibilidades se puede ver en la tabla 3.3 en la página siguiente.

   Como se puede observar en esta tabla, el resultado que obtenemos al usar la heurística no podría ser mejor. Las 80 posibilidades que no eran de utilidad tienen las valoraciones más pequeñas, y entre las 16 que tienen una generalización posible quedan en primer lugar las que tienen como generalización g2 y g4, que son las más específicas y, por tanto, las que se están buscando (quizá sea una regla, y no una simple heurística ...).

### 3.2.4 Cuarta versión

En la cuarta versión del algoritmo de generalización no se modificó el proceso de generalización. Lo que se hizo fue añadir más relaciones en el lenguaje de descripción de escenas. Concretamente, en esta versión se tratan las relaciones **ON** (un objeto justo encima de otro), **TOP** (un objeto que no tiene ningún otro encima), **LEFT** (un objeto justo a la izquierda de otro) y **LEFT*** (un objeto a la izquierda de otro, posiblemente sin tocarse). Las generalizaciones obtenidas podían contener todas estas relaciones y además la relación **SOBRE**, que es la generalización de **ON**. Este aumento de relaciones tratadas hizo que se hubieran de incluir, en el paso 3 del algoritmo (el que calcula la generalización a partir de una de las posibilidades de substitución de constantes por variables encontrada en el paso 1 y que se haya considerado *prometedora* tras la valoración realizada en el paso 2), funciones

| Posibilidades | Valoración | Generalización |
|---|---|---|
| 2 | 55 | g2 |
| 1,4,6,7,8 | 51 | g1(1) g4(4,6,7) g5(8) |
| 3,5 | 49 | g3 |
| 18,24,40 | 43 | g6(18,24) g8(40) |
| 20,22 | 41 | g6 |
| 34,36,38 | 39 | g7 |
| 10,12,17,23,33,39,50,54,89 | 37 | nil |
| 9,11,13,14,15,16,19,21,35, 37,49,51,52,53,55,56 | 35 | nil |
| 73,79,91,93,95 | 33 | nil |
| 28,32,41,45,46,48,57,58, 63,64,70,72,75,77,80,81, 83,84,88,90,96 | 31 | nil |
| 25,26,27,29,30,31,42,43,44, 47,59,60,61,62,65,66,67,68, 69,71,82,85,86,87,92,94 | 29 | nil |
| 74,76,78 | 27 | nil |

Tabla 3.3: Valoración de las 96 posibilidades

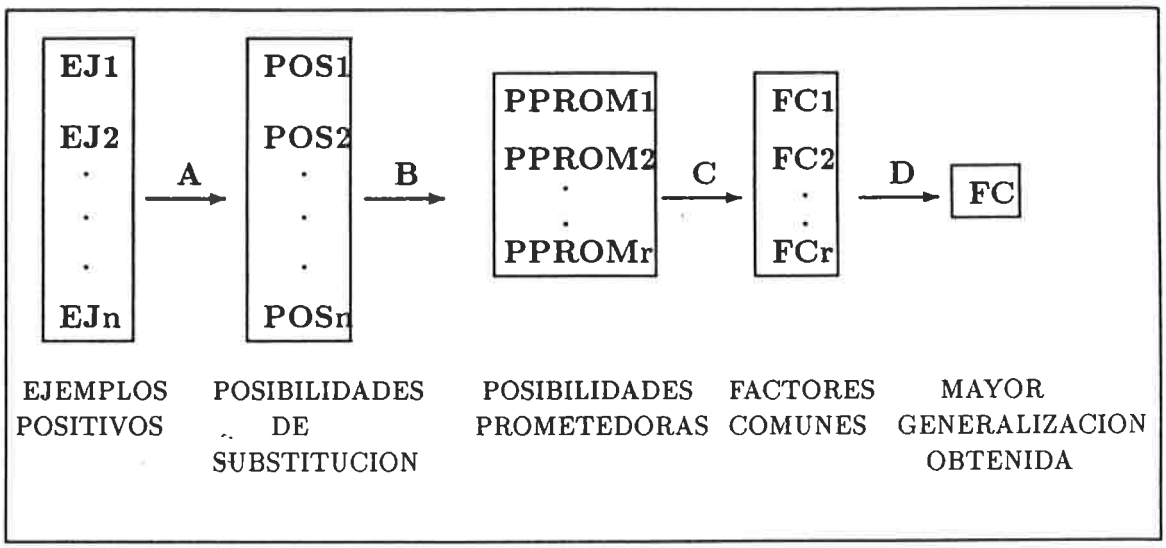
que manejaran específicamente estas relaciones. Estas funciones son, por lo tanto, las que tienen el conocimiento específico del dominio de los bloques. Son las que *saben*, p.e. que la relación **SOBRE** es una generalización de la relación **ON**, que la relación **LEFT*** es transitiva o que la relación **TOP** se cumple cuando no hay ningún bloque encima de un objeto determinado. Estas funciones son las que hacen que las 4 versiones mencionadas hasta ahora del generalizador sean dependientes del dominio. Se puede ver claramente este hecho en el código de la función principal de la tercera fase en esta cuarta versión:

```
;;; Tratar: coge las clausulas del primer ejemplo y deja,
;;; quita o modifica elementos al compararlas con el resto
;;; de ejemplos.

(defun tratar (primero ejemplo)
   (setq l_claus nil)
   (dolist (claus primero)
     (setq funcion (car claus))
     (setq op1 (cadr claus))
     (setq op2 (caddr claus))
     (cond
        ( (equal funcion 'on)
             (if (existe_predicado claus ejemplo)
```

```
                    (setq l_claus (append l_claus (list claus)))
                    (if (existe_sucesion_on op1 op2 ejemplo)
                        (progn
                          (setq cla (cdr claus))
                          (setq cla (cons 'sobre cla))
                          (setq l_claus (append l_claus (list cla)))
                        )))
        )
        ( (equal funcion 'sobre)
            (if (existe_sucesion_on op1 op2 ejemplo)
                    (setq l_claus (append l_claus (list claus))))
        )
        ( (equal funcion 'top)
            (if (existe_predicado claus ejemplo)
                    (setq l_claus (append l_claus (list claus))))
        )
        ( (equal funcion 'left)
            (if (existe_predicado claus ejemplo)
                    (setq l_claus (append l_claus (list claus)))
                     (if (existe_sucesion_left op1 op2 ejemplo)
                       (progn
                        (setq cla (cdr cla))
                        (setq cla (cons 'left* cla))
                        (setq l_claus (append l_claus (list cla))))
                       )
            )
        )
        ( (equal funcion 'left*)
            (if (existe_sucesion_left_t* op1 op2 ejemplo)
                    (setq l_claus (append l_claus (list claus))))
        )
        (t (print "Error: funcion inesperada")
        )
      )
   )
   l_claus
)
```

Las funciones auxiliares sirven para saber, por ejemplo, si existe una sucesión de relaciones **LEFT** o **LEFT*** entre dos objetos determinados, o qué objetos hay a la izquierda de un objeto dado, o si un objeto determinado tiene algún otro objeto por encima, etc.




**A- Substitucion de objetos por variables segun los valores de los atributos**

**B- Valoracion (heuristica) y seleccion de las posibilidades mas prometedoras**

**C- Calculo del factor comun asociado a cada posibilidad prometedora**

**D- Seleccion del factor comun mas largo**

Figura 3.6: Proceso de la versión 4

Las 4 fases de las que consta el proceso de generalización al que se ha llegado en esta cuarta versión se resumen en la figura 3.6.

### 3.2.5 Quinta versión

Una de las deficiencias usuales entre los métodos de aprendizaje **por ejemplos** es que son dependientes del dominio y, en muchos casos, difícilmente generalizables para poder ser usados en dominios diferentes de aquel para el que se diseñó en un principio el algoritmo. Hasta la versión 4 el generalizador sólo trataba con ejemplos del mundo de los bloques, con unas relaciones determinadas **ON, LEFT, ...** que tenían unas propiedades que estaban implícitas en las funciones que trataban estas relaciones, como se ha descrito en la sección anterior (esto también pasaba en sistemas como [**WINS75**] o [**BUCH78**]). En esta última versión del generalizador se estudió qué se tenía que modificar en el algoritmo para que fuera capaz de generalizar ejemplos de dominios diferentes al mundo de los bloques.

Si se examinan las fases previas, se observa lo siguiente:

- **Fase 1**

  En la fase de substitución de objetos por variables no importa el dominio de aplicación. Lo único que se hace es manipular los objetos de los ejemplos, mirar qué valores tienen en los atributos que se están considerando y substituir con la misma variable objetos que tengan los mismos valores en estos atributos.

- **Fase 2**

  Al valorar las posibilidades sólo hay que aplicar la heurística y retener aquellas que den un valor más alto. Con esta función heurística tampoco se tiene en cuenta el dominio: sólo importa la frecuencia de las relaciones dentro de los ejemplos.

- **Fase 3**

  Al calcular el factor común de cada una de las posibilidades seleccionadas en la fase anterior sí se tiene en cuenta el dominio, porque se han de conocer las propiedades de las relaciones para poder llevar a cabo la generalización de cada posibilidad.

- **Fase 4**

  Se selecciona la generalización con más elementos de las encontradas en la fase anterior; tampoco depende del dominio en absoluto.

Por tanto sólo hay que modificar la fase de construcción del factor común asociado a cada posibilidad de manera que el algoritmo se pueda usar en otros dominios. Obviamente, para construir la generalización asociada a cada posibilidad se necesita conocer el comportamiento de las relaciones involucradas. Por ejemplo, si se tiene en una instancia **ON(x,y)** y en otra instancia **ON(x,z)** & **ON(z,y)**, no es posible obtener **SOBRE(x,y)** si no se sabe que la relación **SOBRE** es una generalización de la relación **ON** y que además es transitiva, es decir:

**SOBRE(x,y)** & **SOBRE(y,z)** $\longrightarrow$ **SOBRE(x,z)**.

Si se añade la información referente al comportamiento de las relaciones de forma procedural (implícita en algunas de las funciones del algoritmo) como hasta ahora, no se conseguirá que el método sea independiente del dominio sin tener que escribir funciones específicas para cada relación que intervenga en el mismo. Esta posibilidad queda claramente descartada. Por tanto, sólo queda otra opción: expresar el comportamiento de las relaciones de forma declarativa, formulando de una manera explícita los axiomas que muestran sus propiedades. Por ejemplo, para el mundo de los bloques han de tenerse en cuenta hechos como los siguientes:

- $\forall x, y$ **(ON x y)** $\longrightarrow$ **(SOBRE x y)**
- $\forall x, y, z$ **(SOBRE x y)** & **(SOBRE y z)** $\longrightarrow$ **(SOBRE x z)**

- $\forall x, y$ **(LEFT x y)** $\longrightarrow$ **(LEFT* x y)**
- $\forall x, y, z$ **(LEFT* x y)** & **(LEFT* y z)** $\longrightarrow$ **(LEFT* x z)**

El problema que se plantea ahora es que el proceso de generalización se puede complicar hasta límites insospechados si se ofrece al usuario un mecanismo para expresar los axiomas que sea lo suficientemente potente para describir cualquier dominio. Por ejemplo, se puede imaginar la complejidad que significaría permitir conjunciones cualesquiera en lógica de primer orden, como

$\forall x, y, z, w$ **(R x y)** & **(R x z)** & **(R y w)** & **R(w z)** $\longrightarrow$ **R(x w)**.

Para reducir esta complejidad se decidió permitir sólo dominios cuyas relaciones se pudieran describir usando dos tipos de axiomas:

1. **Axiomas reflexivos**

   Estos axiomas, que también llamaremos *axiomas de substitución*, son los que indican que una relación es una generalización directa de otra, como

   $\forall x, y$ **(ON x y)** $\longrightarrow$ **(SOBRE x y)**.

   Estos axiomas, además, permiten que en la generalización obtenida a partir de una serie de ejemplos aparezcan relaciones que no estaban presentes en el lenguaje en que éstos se describen. Ya se vió en una de las versiones anteriores que la relación **SOBRE** no estaba presente en la descripción de los ejemplos pero sí podía aparecer en la generalización final.

2. **Axiomas transitivos**

   Se usan para mostrar que una relación cumple la propiedad transitiva, como por ejemplo

   $\forall x, y, z$ **(LEFT* x y)** & **(LEFT* y z)** $\longrightarrow$ **(LEFT* x z)**.

Estos dos tipos de axiomas se almacenan para el uso del algoritmo en dos variables globales **AX-SUBST** y **AX-TRANSIT**. **AX-SUBST** es una lista de parejas *(a b)*, indicando que la relación *b* es una generalización de la relación a, y **AX-SUBST** es una lista de las relaciones transitivas.

Una vez se tiene en forma declarativa este conocimiento sobre los axiomas, se ha de pensar cómo utilizarlo para obtener las generalizaciones. Se modificó el algoritmo, que pasó a tener las siguientes fases:

1. Creación de todas las posibilidades de substitución de objetos por variables, relacionando con la misma variable objetos que tuvieran los mismos valores en los atributos que se estuviesen considerando en ese momento (igual que en las anteriores versiones del algoritmo).

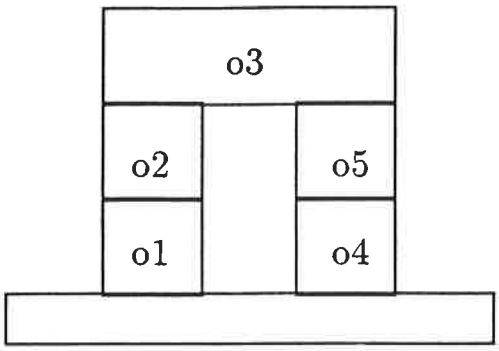



Figura 3.7: Otro arco con 5 objetos

2. Hacer una valoración de las posibilidades, que estime cuales son las que pueden llevar a generalizaciones interesantes y cuales no vale la pena considerar.
3. *Extensión* de las posibilidades prometedoras de acuerdo con los axiomas del dominio.
4. Cálculo del factor común de las posibilidades *extendidas*.
5. Extracción de las posibles redundancias que pudiera haber en los factores comunes.
6. Selección de la mejor generalización, o sea, el factor común más largo.

La fase de *extensión* de las posibilidades prometedoras consiste en hacer explícita toda la información implícita en la descripción de los ejemplos. Por ejemplo, al extender la descripción del arco de la figura 3.7 tendríamos:

- **Descripción:**

  **((on o1 mesa) (on o2 o1) (left* o1 o4) (left* o2 o5) (on o3 o2) (on o3 o5) (on o5 o4) (on o4 mesa))**

- **Información sobre el dominio:**

  - **AX-SUBST: ((ON SOBRE) (LEFT LEFT*))**
  - **AX-TRANSIT: (SOBRE LEFT*)**

- **Descripción extendida:**

  **((on o1 mesa) (sobre o1 mesa) (on o2 o1) (sobre o2 o1) (sobre o2 mesa) (left* o1 o4) (left* o2 o5) (on o3 o2) (sobre o3 o2) (sobre o3 o1) (sobre o3 mesa) (on o3 o5) (sobre o3 o5) (on o5 o4) (sobre o5 o4) (sobre o3 o4) (on o4 mesa) (sobre o4 mesa) (sobre o5 mesa))**

Concretamente, para llegar a estas descripciones extendidas se sigue el siguiente proceso:

1. Aplicar los axiomas substitutivos a la descripción.

2. Aplicar repetidamente los axiomas transitivos hasta que se llegue a un punto en que no se añaden más cláusulas a la descripción.

Al introducir la información sobre los axiomas de forma explícita se ha conseguido que la fase de cálculo de la expresión generalizada asociada a una de las posibilidades de substitución de objetos por variables sea trivial. Al extender la descripción de los ejemplos mostrando toda la información que tenía implícita, **el factor común estará formado por aquellas relaciones que aparezcan en todos los ejemplos de la posibilidad extendida que estemos tratando.** Por tanto, esta fase es absolutamente independiente del dominio de aplicación, ya que lo único que tiene que hacer es examinar las cláusulas y contar cuantas veces aparecen.

El quinto punto en este proceso (extracción de las redundancias) es necesario porque al haber realizado la extensión se puede llegar a expresiones generalizadas tales como:

**(ON x1 mesa) (SOBRE x1 mesa) (ON x2 x1) (SOBRE x2 x1)**

**(SOBRE x2 mesa)**

que es equivalente a una mucho más sencilla:

**(ON x1 mesa) (ON x2 x1).**

Esta extracción de las redundancias se puede entender como una aplicación *hacia atrás* de los axiomas. Este uso de los axiomas para obtener un algoritmo independiente del dominio tiene puntos similares con otros métodos ([**DIET83**]):

- Vere ([**VERE75**]) usa una técnica de activación para extraer las relaciones relevantes de una base de datos y las añade a los ejemplos de la entrada antes de generalizarlos.

- En el programa INDUCE de Michalski [**MICH80**] se puede incorporar conocimiento específico del dominio en el programa definiendo los tipos y dominios de los descriptores, especificando las estructuras de estos dominios, especificando reglas de producción simples (para definir restricciones del dominio sobre combinaciones legales de variables) y proporcionando **reglas de inducción constructivas**. Este último punto parece similar al mecanismo incorporado en la versión 5.

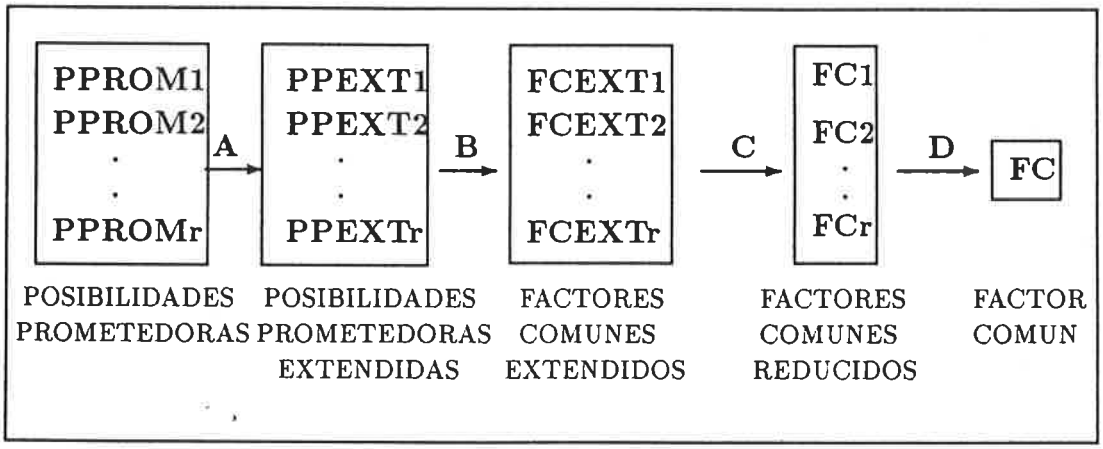
El estado final del proceso se puede seguir en el siguiente pseudo algoritmo:

**Función Generalizador (ejemplos) devuelve factor-común**

```
posibs=substitución-de-objetos-por-variables(ejemplos)
pproms=valorar-y-seleccionar-posibilidades-prometedoras(posibs)
ppexts=extender-posibilidades-prometedoras(pproms,ax-subst,ax-transit)
fcexts=obtener-factores-comunes(ppexts)
fcreds=reducir-factores-comunes(fcexts,ax-subst,ax-transit)
fc=seleccionar-resultado-más-específico(fcreds)
devolver fc
```

**ffunción**

Las fases que cambian en la versión 5 respecto a la anterior quedan reflejadas en el esquema de la figura 3.8.




**A- Extension de las posibilidades prometedoras con ayuda de los axiomas del dominio**

**B- Calculo de los factores comunes extendidos**

**C- Simplificacion de los factores comunes extendidos (proceso inverso al paso A)**

**D- Seleccion del factor comun mas largo**

Figura 3.8: Cambios en la versión 5

# Capítulo 4

# Comparación con trabajos previos

## 4.1 Introducción

Para comparar el algoritmo de generalización con trabajos previos similares se hará referencia a varios artículos y un estudio comparativo realizado en **[DIET83]**. En él se contrastan los métodos de Buchanan **[BUCH78]**, Hayes-Roth **[HAYE77]**, Vere **[VERE75]**, Winston **[WINS75]** y Dietterich y Michalski **[DIET81]**. Este trabajo tiene en cuenta aspectos como :

- **Aplicación**

  Estudiar si el método es de aplicación general (usable en diversos dominios) o si, por el contrario, sólo puede utilizarse en un dominio concreto.

- **Lenguaje usado**

  Qué representación del conocimiento se ha usado dentro del algoritmo para manipular objetos, operadores, etc..

- **Reglas usadas**

  Qué reglas de generalización o especialización se tienen en consideración en el algoritmo.

- **Eficiencia**

  Efectuar diversas medidas de la computación requerida por el algoritmo.

- **Extensibilidad**

  Estudiar si el algoritmo es fácilmente modificable para ser usado en varios dominios, si es robusto ante posible *ruido* en los datos que se le presentan, si tiene incorporado conocimiento sobre el dominio, etc..

## 4.2 Otros métodos de aprendizaje inductivo

En el estudio realizado por Dietterich y Michalski en **[DIET83]** se comparan diversos métodos de aprendizaje inductivo a partir de ejemplos usando un caso concreto, que es el de la siguiente figura :

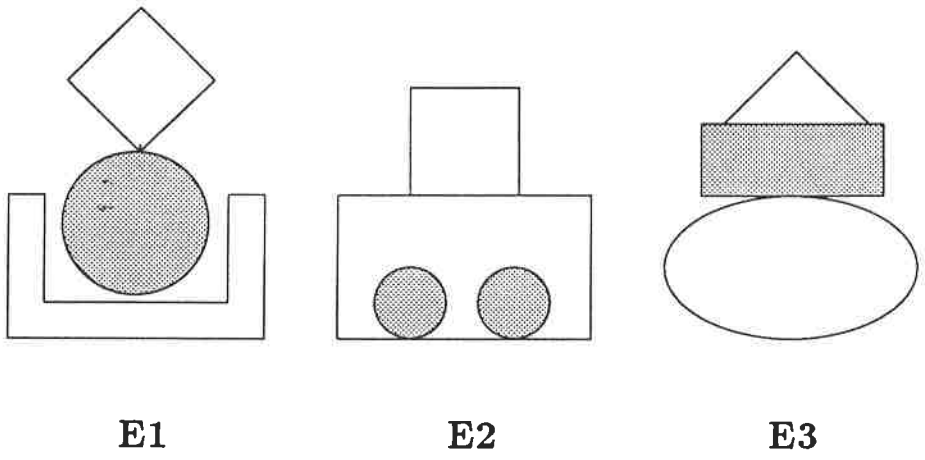



Figura 4.1: Ejemplo usado para la comparación de métodos

Lo que se intenta es determinar un conjunto de generalizaciones conjuntivas lo más específicas posibles a partir de los ejemplos. No se ofrece ningún ejemplo negativo. En las siguientes secciones vamos a describir qué resultados se obtienen con el algoritmo de generalización (**Am**) desarrollado en este trabajo y qué resultados obtienen algunos de los métodos mencionados en dicho artículo.

### 4.2.1 Resultados con el generalizador

La descripción de los objetos del ejemplo que se le suministran al generalizador se puede encontrar en el apéndice A.

En la descripción de los objetos intervienen 3 atributos:

- **Forma**

  Puede tener los valores *caja*, *círculo*, *rombo*, *rectángulo*, *cuadrado*, *elipse* y *triángulo*.

- **Tamaño**

  Puede ser *grande*, *medio* o *pequeño*.

- **Textura**

  Este atributo distingue los objetos sombreados de los que no lo están (*lisos*).

La generalización que se obtiene depende, como se ha explicado en el capítulo anterior, de los atributos que se consideren en cada momento. Los resultados obtenidos fueron los siguientes :

- **Atributos forma, tamaño y textura**

  Sólo se encuentra una posibilidad de substitución de objetos por variables (el programa siempre muestra al menos la substitución en la que ha cambiado todos los objetos del primer ejemplo por variables), y no se encuentra ninguna generalización adecuada a partir de ella.

- **Atributos tamaño y textura**

  También sólo se encuentra una posibilidad de substitución de objetos por variables, que es

```
((((ON x1 table) (ON x2 x1) (ON x3 x2))
  ((ON o4 table) (ON o5 table) (ON x1 table) (INSIDE o4 x1)
   (INSIDE o5 x1) (ON x3 x1))
  ((ON x1 table) (ON x2 x1) (ON o10 x2)))) .
```

  El algoritmo de generalización obtiene el siguiente resultado :

  - **Generalización obtenida:** **((ON x1 table)).**
  - **Valores de los atributos:** x1 **textura** *liso*, **tamaño** *grande.*
  - **Interpretación del resultado:** *Justo encima de la mesa hay un objeto grande que no está sombreado.*

- **Atributos forma y tamaño**

  Aquí pasa lo mismo que al considerar los 3 atributos: no se llega a ninguna generalización informativa.

- **Atributos forma y textura**

  Tampoco se llega a ninguna generalización (en todos estos casos el programa simplemente devuelve nil).

- **Atributo tamaño**

  Sólo hay una posibilidad de substitución de objetos por variables, y de ella se obtiene el siguiente resultado :

  - **Generalización obtenida:** **((ON x1 table) (ON x2 x1)).**
  - **Valores de los atributos:** x1 **tamaño** *grande* ; x2 **tamaño** *medio.*

- **Interpretación del resultado**: *Justo encima de la mesa hay un objeto grande que tiene justo encima un objeto de tamaño medio.*

- **Atributo forma**

  El algoritmo de generalización no es capaz de encontrar ninguna generalización de los ejemplos presentados teniendo en cuenta sólo este atributo.

- **Atributo textura**

  Este es el caso donde existen más posibilidades de substitución de objetos por variables (concretamente 8). Este hecho es debido a que hay un aumento de las posibilidades de combinación porque se considera un atributo que sólo tiene 2 valores posibles. El resultado ofrecido por el generalizador es :

  - **Generalización obtenida**: **((ON x1 table) (ABOVE x2 table) (ABOVE x3 x1))**.
  - **Valores de los atributos**: x1 **textura** *liso* ; x2 **textura** *sombreado* ; x3 **textura** *liso*.
  - **Interpretación del resultado**: *Justo encima de la mesa hay un objeto que no está sombreado. En algún lugar por encima de este objeto hay otro objeto que tampoco está sombreado. Además, en alguna parte por encima de la mesa hay un objeto sombreado.*

Se puede apreciar que el algoritmo de generalización no es capaz de encontrar ninguna característica común a todos los ejemplos en todos los casos donde ha de considerar el atributo **forma**. Este hecho no es casual, y tiene una explicación sencilla. Al haber tan sólo 10 objetos y tener este atributo 7 valores diferentes es muy difícil que el generalizador pueda substituir objetos de ejemplos diferentes por la misma variable (recuérdese que para hacerlo han de tener los mismos valores en todos los atributos que se estén considerando). Este hecho es el que marca la diferencia más importante entre el algoritmo desarrollado en este trabajo y otros métodos previos: *se da una importancia fundamental a los atributos, y se obliga al algoritmo a que tenga en cuenta los atributos que le interesen al usuario en cada momento.* Este enfoque es, a mi parecer, mucho más parecido a la aproximación humana del mismo problema. Hay muchas ocasiones en que se nos presentan ejemplos de un concepto para que determinemos qué vemos en común en todos ellos, y a veces no observamos detalles aparentemente evidentes hasta que nos comentan: *"Fíjate en tal aspecto de los ejemplos"* y, entonces, de una forma repentina la mayoría de las veces, nos damos cuenta de aquel aspecto que se nos había pasado por alto hasta ese momento.

De todas formas -como se explica más adelante en este capítulo- después de esta reflexión y a la vista de resultados de otros algoritmos se pensó en la posibilidad de aplicar una filosofía que no tuviera tan en cuenta los atributos (más parecida al resto de métodos) sin cambiar el algoritmo y se consiguió (en cierta medida) con un pequeño *truco* que se explicará posteriormente.

### 4.2.2 Método Winston

El método que se va a comentar a continuación fue desarrollado por Winston en la década de los 70 ([**WINS70**], [**WINS75**]), y es la base para toda una rama del aprendizaje automático conocida como **SBL** (aprendizaje basado en similitudes). Una versión resumida de [**WINS75**] se puede encontrar en [**MORE92a**], donde también se comentan otros tipos de aprendizaje. El aprendizaje por similitudes tiene por objetivo que la máquina aprenda la descripción de un concepto determinado después de haber visto una serie de ejemplos y contraejemplos del concepto. Se trata, por tanto, de un aprendizaje supervisado, guiado por un maestro que va mostrando a la máquina estos ejemplos y contraejemplos en el orden y forma más convenientes.

El programa de Winston trabajaba en el dominio de objetos triédricos como los bloques, esferas, pirámides y objetos sencillos en general (el dominio de juguete de los bloques, ya clásico en la Inteligencia Artificial). El primer problema que enfocó Winston fue el de cómo representar las escenas, y decidió usar redes semánticas, argumentando que son lo suficientemente sencillas y flexibles como para poder representar las escenas de forma adecuada. En estas redes semánticas cada objeto se representa en relación con otros objetos y conceptos conocidos por el programa (p.e. bloque). Se representan de la misma forma las relaciones entre objetos que las propiedades de los objetos. Como ejemplo, se puede ver en la figura 4.2 cómo se representaría la primera instancia del concepto de la figura 4.1.

Para el tipo de aprendizaje que quería hacer Winston (a base de ejemplos y cuasiejemplos) le era imprescindible poder comparar las descripciones de dos escenas y ver cuales eran las diferencias entre ellas. Para hacer eso estableció un proceso de cotejamiento entre las redes que representan las escenas, para ver qué partes de cada red se corresponden entre sí. Un proceso explora las dos redes descriptivas y decide qué nodos de ellas se *corresponden* mejor, en el sentido de tener la misma función en las dos redes respectivas. Estos dos nodos se dice que están unidos (**linked**) el uno con el otro.

Una vez el proceso de cotejamiento ha examinado las dos redes y se han establecido los pares de nodos ligados, se procede a la descripción de las semejanzas en las redes. Eso se hace con otra red semántica, que describe las partes de las redes que se corresponden. Esta red se llama *esqueleto* porque es la base para el resto de la descripción de la comparación. Cada pareja de nodos ligada contribuye con un nodo al esqueleto, que es básicamente una copia de la estructura que está duplicada en las redes comparadas.

La descripción completa de la comparación tiene 2 partes: el esqueleto y un segundo grupo de nodos llamados **notas de comparación**. Hay varios tipos de notas de comparación, que indican cosas como intersecciones (dos nodos de un par ligado que comparten el valor de una determinada propiedad), punteros suplementarios, modificaciones de punteros,etc. (ver [**WINS75**] o [**DIET83**]).

La fase de generalización consiste simplemente en aplicar un tratamiento concreto a cada nota de comparación dependiendo de su tipo y de si es un ejemplo positivo o un cuasiejemplo. En la tesis de Winston ([**WINS70**]) se incluye una tabla con el compor-

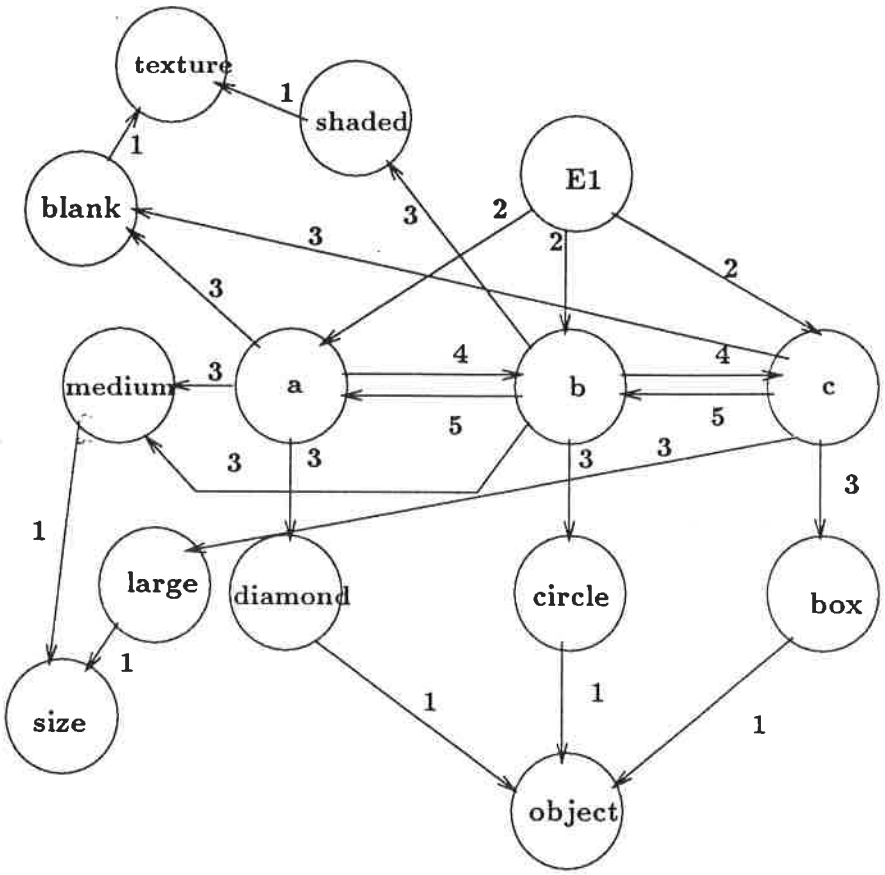



Figura 4.2: Descripción a *la Winston* del primer ejemplo

tamiento en cada caso.

En el método de Winston la generalización a la que se llega depende del orden en que se le presentan los ejemplos. En nuestro generalizador no, la única heurística que se sigue es poner en primer lugar el ejemplo con menos objetos para minimizar las substituciones de objetos por variables. En las figuras 4.3 y 4.4 se pueden ver dos de las generalizaciones a las que llega el programa de Winston.

La primera generalización se puede parafrasear de la siguiente forma : *Hay un polígono de tamaño medio y que no está sombreado sobre otro objeto que tiene tamaño y textura. Hay también otro objeto con tamaño y textura.* La segunda generalización es equivalente a decir: *Hay un objeto grande y que no está sombreado.*

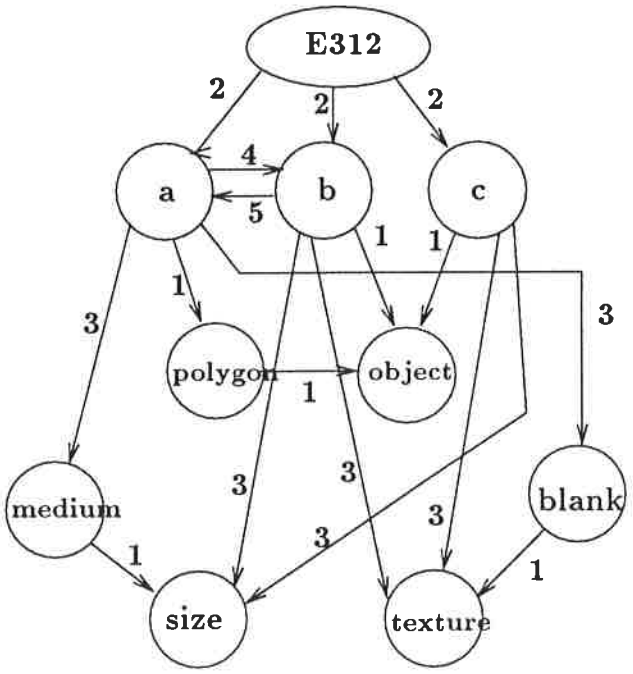



Figura 4.3: Generalización a *la Winston*

Se puede apreciar que en esta segunda generalización se mencionan valores específicos de los atributos *forma* y *textura*. Este hecho hace que esta segunda generalización sea muy similar a la encontrada por nuestro algoritmo cuando tiene en cuenta estos dos atributos. La única diferencia consiste en la referencia a la mesa que hace nuestro generalizador, que no aparece en el método de Winston ya que éste no utiliza ningún punto de referencia.

Lo que no puede hacer el generalizador desarrollado en este trabajo es llegar a algún resultado similar al de la primera generalización de Winston. Hechos como *"sobre otro objeto que tiene tamaño y textura"* no pueden ser tenidos en cuenta debido a que en la generalización que se calcula las variables siempre tienen unos valores determinados en los atributos que se estén considerando, y el resto de atributos no importan. Por supuesto, se podría mencionar en cualquier generalización los valores de los atributos y añadir que el resto de atributos pueden tener cualquier valor, y entonces se llegaría a expresiones como la de Winston. Por ejemplo, la generalización a la que se ha llegado con el atributo *tamaño* se podría describir como: *Hay un objeto grande sobre la mesa, que tiene una forma y una textura. Sobre él hay un objeto de tamaño mediano, que también tiene una forma y una*

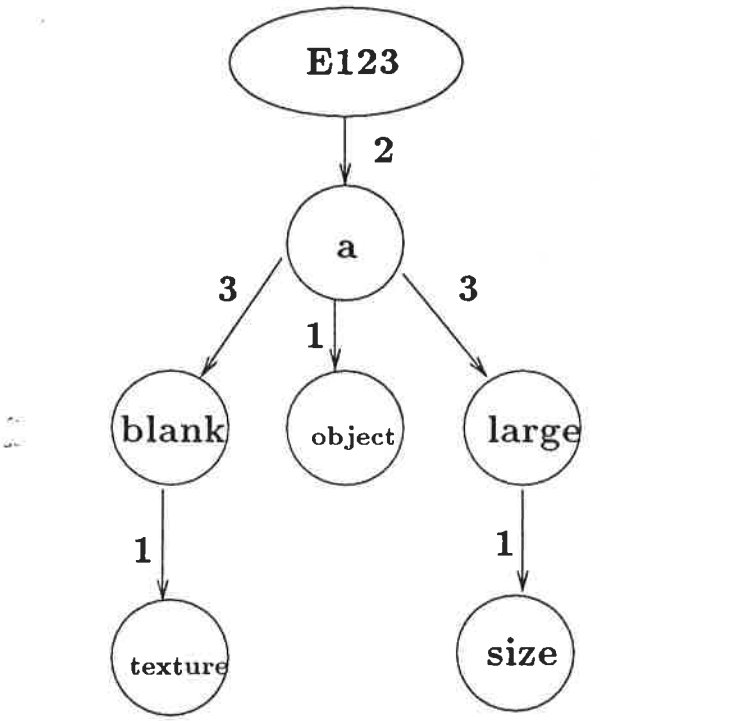



Figura 4.4: Otra generalización *a la Winston*

*textura.*

Otro aspecto interesante de la primera generalización encontrada con el método de Winston es que dice que *hay un* **polígono** *de tamaño medio (...).* La palabra *polígono* aparece porque una de las reglas de generalización empleadas por Winston es la regla de subida por el árbol de generalización (**climbing generalization tree rule, [MICH83]**). En este caso específico la jerarquía de generalización se refiere a los valores que puede tener el atributo forma, y sólo tendría un nivel, como se puede ver en la figura 4.5 .

Así, si dos de los nodos en un par ligado tienen valores diferentes en un atributo, estos valores específicos serán substituídos en la generalización por el valor más bajo en el árbol de generalización que sea antecesor de ambos valores. En este caso, dados dos valores diferentes cualesquiera en el atributo forma siempre serían substituídos por el único antecesor que tienen, que es *polígono.*

Hay varios métodos que utilizan esta técnica (p.e. **[DIET81]**) y, al ver estos resultados, se intentó pensar algún método que permitiera aplicar esta regla en el generalizador sin

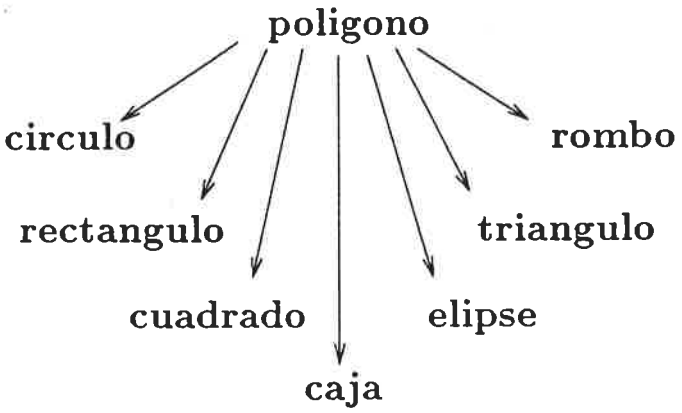



Figura 4.5: Árbol de generalización del atributo *forma*

tener que cambiar el código de forma sensible. Se puede hacer aplicando una pequeña trampa, que es la siguiente: si se dejan los atributos como tales, no se podrá conseguir el efecto pretendido; en la descripción de las escenas sólo se pueden expresar dos cosas: relaciones entre objetos (a través de predicados) y atributos de los objetos. Por lo tanto, no queda más remedio que pasar la información de atributos a relaciones, escribiendo cosas como **(GRANDE o2)** o **(VERDE o4)**, como las que usa Núñez para describir las escenas. Obviamente estos predicados de un argumento no son estrictamente relaciones, ya que no relacionan dos objetos diferentes, sino que explican la particularidad de un objeto. Si simplemente se hace este cambio no cambiará la situación, porque si en dos ejemplos diferentes se tiene **(CIRCULO o1)** y **(CUADRADO o4)** no se pueden generalizar a **(POLIGONO x2)** de ninguna forma. El *truco* consiste en pensar lo siguiente: se dispone de un mecanismo que permite introducir en la generalización predicados que no estaban en la representación inicial de los ejemplos, que son los axiomas de substitución. Estos permitían obtener cosas como **(SOBRE x z)** a partir de **(ON x z)** y **((ON x y) (ON y z))**. Un árbol de generalización como el de la figura 4.5 se puede describir con axiomas como los siguientes:

- $\forall x$ **(CUADRADO x)** $\longrightarrow$ **(POLIGONO x)**
- $\forall x$ **(RECTANGULO x)** $\longrightarrow$ **(POLIGONO x)**
- $\forall x$ **(ROMBO x)** $\longrightarrow$ **(POLIGONO x)**

Si se incluyen estos axiomas en la información conocida (*background knowledge*) por el algoritmo de generalización, si en un ejemplo se tiene **(CUADRADO x1)** y en otro **(RECTANGULO x1)**, en la fase de extensión de las posibilidades pasarán a **((CUADRADO x1) (POLIGONO x1))** y **((RECTANGULO x1) (POLIGONO x1))** y, por lo tanto, se encontrará el factor común **(POLIGONO x1)**, que es lo que se quería.

Un pequeño problema de esta aproximación es que el programa necesita por lo menos un atributo en el que basar su generalización, y no hay ninguno si se aplica este criterio de paso de atributo a relación a todos ellos. Se puede solucionar fácilmente si se introduce un atributo *fantasma* con el mismo valor en todos los objetos y se pide al generalizador que haga la generalización teniendo en cuenta sólo ese atributo. Esto significa que en la fase de creación de posibilidades de substitución podrá emparejar cualquier objeto de un ejemplo con cualquier objeto de cualquier otro ejemplo, lo que dará lugar a un número de combinaciones muy grande. Estas adaptaciones del lenguaje son necesarias porque la genericidad deseada tiene un precio; en este caso sólo adaptar el lenguaje de representación.

En el ejemplo concreto que se está tratando, si se aplican estas modificaciones a la descripción de las escenas (en los 3 atributos se puede ver una jerarquía expresable con un árbol de un nivel), queda como se puede ver en el apéndice A.

Algunas de las generalizaciones encontradas por el programa entre las 144 posibilidades de substitución de objetos por variables son las siguientes (para mostrar estos resultados se ha eliminado la parte de la descripción que se refiere a atributos transformados en predicados de un argumento) :

1. **((ON x1 table) (ABOVE x2 table) (ABOVE x3 x1))**

   - **Atributos de las variables :**
     - x1: tamaño **grande**, textura **liso**, forma *polígono*
     - x2: tamaño *tamaño*, textura **sombreado**, forma *polígono*
     - x3: tamaño *tamaño*, textura **liso**, forma *polígono*
   - **Interpretación :**

     *Hay un polígono grande y no sombreado encima de la mesa. En algún punto encima suyo hay un polígono no sombreado que tiene un cierto tamaño. También hay en algún punto encima de la mesa un polígono sombreado que tiene un tamaño.*

2. **((ON x1 table) (ABOVE x2 table) (ABOVE x3 table))**

   - **Atributos de las variables :**
     - x1: tamaño **grande**, textura **liso**, forma *polígono*
     - x2: tamaño *tamaño*, textura **sombreado**, forma *polígono*
     - x3: tamaño *tamaño*, textura *textura*, forma *polígono*
   - **Interpretación :**

     *Hay un polígono grande y no sombreado encima de la mesa. En algún punto sobre la mesa hay un polígono que tiene un cierto tamaño y textura. También hay en algún punto encima de la mesa hay otro polígono que tiene un tamaño y una textura.*

3. **((ON x1 table) (ON x2 x1) (ABOVE x3 table))**

   - **Atributos de las variables :**
     - x1: tamaño **grande**, textura **liso**, forma *polígono*
     - x2: tamaño **medio**, textura *textura*, forma *polígono*
     - x3: tamaño *tamaño*, textura *textura*, forma *polígono*
   - **Interpretación :**
     *Hay un polígono grande y no sombreado encima de la mesa. Justo encima suyo hay un polígono de tamaño medio que tiene una cierta textura. También hay en algún punto encima de la mesa un polígono que tiene un cierto tamaño y textura.*

4. **((ABOVE x1 table) (ABOVE x2 table) (ABOVE x3 table))**

   - **Atributos de las variables :**
     - x1: tamaño *tamaño*, textura **liso**, forma *polígono*
     - x2: tamaño *tamaño*, textura **sombreado**, forma *polígono*
     - x3: tamaño *tamaño*, textura **liso**, forma *polígono*
   - **Interpretación :**
     *En alguna parte por encima de la mesa hay dos polígonos no sombreados y un polígono sombreado. Todos estos objetos tienen un cierto tamaño (que no tiene porqué coincidir entre ellos, por supuesto).*

En la tabla de la página siguiente (en la cual hay datos obtenidos en **[DIET83]**) se pueden encontrar resumidas las principales diferencias entre el método de Winston y el desarrollado en este trabajo. Se tienen en cuenta aspectos tales como el dominio de aplicación del método, el lenguaje usado para representar los ejemplos y los conceptos, qué operadores se puede llegar a usar en la descripción de los conceptos, qué reglas de generalización se han usado (ver **[MICH83]** para una descripción extensiva de las reglas de generalización más habitualmente empleadas), etc.. Asimismo, se tratan aspectos de extensibilidad, tales como inclusión de formas disjuntivas, inmunidad al ruido en los datos, la posibilidad de dar al programa conocimiento del dominio y si se permite inducción constructiva (que en la generalización aparezcan atributos que no están presentes en los ejemplos iniciales).

Más adelante se explica porque se afirma que nuestro método podría ampliarse de forma que obtuviese generalizaciones conjuntivas.

| **Aspecto** | **Winston** | **Am** |
|---|---|---|
| Dominio | Mundo bloques | Expresable con axiomas substitutivos y transitivos |
| Lenguaje | Redes semánticas | Propiedades y relaciones binarias |
| Conceptos sintácticos | Nodos y uniones | Átomos, listas y listas de propiedades |
| Operadores | AND, excepción | AND |
| Reglas de generalización | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización *especial* |
| **Extensibilidad** | | |
| Formas disjuntivas | No | Sí |
| Inmunidad al ruido | Muy baja | Nula |
| Conocimiento del dominio | Incorporado al programa | Declarado en axiomas |
| Inducción constructiva | Limitada | Con axiomas substitutivos |

Tabla 4.1: Comparación con el método de Winston

### 4.2.3 Método Hayes-Roth

En el trabajo de Hayes-Roth y McDermott ([**HAYE77**], [**HAYE78**]) sobre aprendizaje inductivo se intenta encontrar las generalizaciones conjuntivas más específicas (en su nomenclatura **maximal abştractions** o **interference matches**) a partir de un conjunto de instancias positivas. La estructura que usan para representar tanto estas instancias como las generalizaciones la llaman **parameterized structural representations** (PSRs). Los dos primeros ejemplos que se usan en este capítulo se describirían de la siguiente forma :

```
E1: {{caja:a}{circulo:b}{rombo:c}
     {liso:a}{sombreado:b}{liso:c}
     {grande:a}{medio:b}{medio:c}
     {sobre:b,debajo:a}{sobre:c,debajo:b}}

E2: {{rectangulo:d}{circulo:e}{circulo:f}{cuadrado:g}
     {grande:d}{pequeno:e}{pequeno:f}{medio:g}
     {liso:d}{sombreado:e}{sombreado:f}{liso:g}
     {sobre:g,debajo:d}{fuera:d,dentro:e}{fuera:d,dentro:f}}
```

Cada uno de los componentes de esta representación es un **case frame**, compuesto de **case labels** (*pequeño, círculo*) y de parámetros (*a, b*). Se asume que todos los *case frames*

están conectados de forma conjuntiva. La generalización se hace de la siguiente manera: el primer conjunto de generalizaciones conjuntivas, $G_1$, se inicializa con el primer ejemplo de la entrada. Dado un nuevo ejemplo y el conjunto de generalizaciones obtenido en el paso i-ésimo $G_i$, $G_{i+1}$ se obtiene haciendo un cotejamiento parcial (**interference match**) entre cada elemento de $G_i$ y el ejemplo de entrenamiento actual. Este cotejamiento intenta encontrar la asociación uno-a-uno más larga de parámetros y *case frames*.

Esto se hace en 2 pasos :

1. Hacer el cotejamiento de los *case frames E1* y *E2* de todas las formas posibles para obtener un conjunto $M$, cada elemento del cual será un *case frame* y una lista de correspondencias entre parámetros que permite hacer *matching* con ambos *case frames*.

   El conjunto $M$ que se obtendría a partir de los 2 primeros ejemplos descritos anteriormente sería el siguiente :

```
M = {{circulo:((b/e)(b/f))},
     {liso:((a/d)(a/g)(c/d)(c/g))},
     {sombreado:((b/e)(b/f))}
     {grande:((a/d))}
     {medio: ((b/g)(c/g))}
     {sobre,debajo: ((b/g a/d)(c/g b/d))}}
```

2. Seleccionar un subconjunto de las correspondencias entre parámetros de $M$ de tal forma que todos los parámetros se puedan vincular de forma consistente. Esta selección se hace con un recorrido en anchura del espacio de posibles vinculaciones, podando los nodos que no sean prometedores.

Una vinculación *consistente* significa, por ejemplo, no vincular un mismo parámetro de una instancia con varios parámetros de otra instancia. Un trozo del grafo que se obtendría a partir de este conjunto $M$ se puede ver en la figura 4.6. Cada número de ese grafo representa uno de los nodos que se generarían en el proceso de generalización. El nodo 18 es una vinculación que lleva a una generalización conjuntiva. Concretamente vincula $a$ a $d$ (para obtener *v1*), $b$ a $e$ (para obtener v2) y $c$ a $g$ (para obtener v3), produciendo la conjunción :

```
{{circulo:v2}
 {liso:v1}{sombreado:v2}{liso:v3}
 {grande:v1}{medio:v3}}.
```

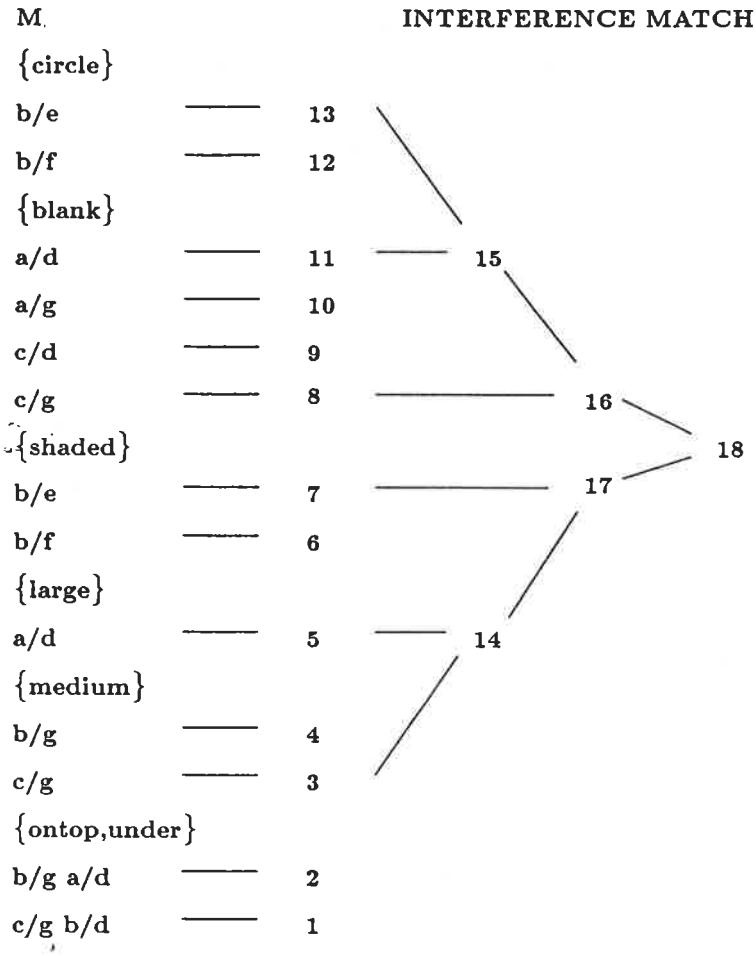



Figura 4.6: Parte del grafo de posibles vinculaciones

Esta conjunción se interpretaría como *Hay un círculo sombreado, un objeto grande no sombreado y un objeto mediano que tampoco está sombreado.*

El algoritmo de Hayes-Roth encuentra las siguientes generalizaciones a partir de los 3 ejemplos considerados en este capítulo :

- **{{sobre:v1,debajo:v2}{medio:v1}{liso:v1}}**
  - *Hay un objeto no sombreado de tamaño medio encima de algo*
- **{{sobre:v1,debajo:v2}{medio:v1}{grande:v2}{liso:v2}}**
  - *Hay un objeto de tamaño medio encima de un objeto grande que no está sombreado*
- **{{medio:v1}{liso:v1}{grande:v3}{liso:v3}{sombreado:v2}}**

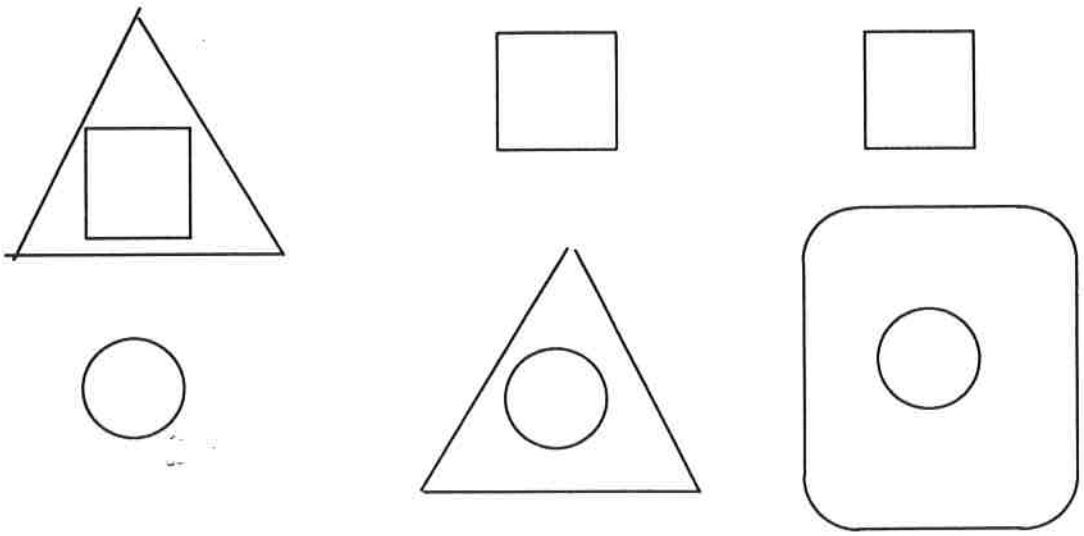



Figura 4.7: Ejemplo de McDermott

- *Hay un objeto no sombreado de tamaño medio, un objeto grande no sombreado y un objeto sombreado*

Como se puede apreciar, estas generalizaciones son muy similares a las que se obtienen con nuestro método (p.e. la segunda de ellas es casi idéntica a una de las que se encontró cuando se pasaron las propiedades a relaciones unarias).

También se puede usar el ejemplo que utilizan Hayes-Roth y McDermott en **[HAYE78]** para tener otra comparación. El ejemplo que utilizan se puede ver en la figura 4.7. La generalización que obtiene McDermott con su método es la siguiente :

- {{ **mismo!tamaño:v1 , mismo!tamaño:v2** }}, { **encima:v1 , debajo:v2** }, { **cuadrado:v1** }, { **círculo:v2** }, { **pequeño:v1** }, { **pequeño:v2** }, { **grande:v3** }}

  - *Hay tres objetos, incluyendo un círculo pequeño y un cuadrado pequeño. El cuadrado está por encima del círculo. El tercer objeto es grande.*

Los resultados obtenidos con nuestro algoritmo son los siguientes :

- **Con los atributos TAMAÑO y FORMA**

  - Existe sólo 1 substitución de objetos por variables.
  - Generalización: **((ENCIMA x1 x3))**.

  - Valores de los atributos :
    * x1: forma **cuadrado**, tamaño **pequeño**
    * x3: forma **círculo**, tamaño **pequeño**
  - *Hay un círculo pequeño por encima de un cuadrado pequeño.*

- **Con el atributo FORMA**
  - Existen 2 substituciones posibles de objetos por variables.
  - Generalización: **((ENCIMA x1 x3))**.
  - Valores de los atributos :
    * x1: forma **cuadrado**
    * x3: forma **círculo**
  - *Hay un círculo por encima de un cuadrado.*

- **Con el atributo TAMAÑO**
  - Existen 4 substituciones posibles de objetos por variables.
  - Generalización 1: **((ENCIMA x1 x3))**.
  - Valores de los atributos :
    * x1: tamaño **pequeño**
    * x3: tamaño **pequeño**
  - *Hay un objeto pequeño por encima de otro objeto pequeño.*
  - Generalización 2: **((IN x1 x2))**.
  - Valores de los atributos :
    * x1: tamaño **pequeño**
    * x2: tamaño **grande**
  - *Hay un objeto pequeño dentro de un objeto grande.*

- **Sin atributos**
  - Existen 36 substituciones posibles de objetos por variables.
  - Generalización: **((ENCIMA x1 x3))**.
  - Valores de los atributos :
    * x1: forma **cuadrado**, tamaño **pequeño**
    * x2: tamaño **grande**
    * x3: forma **círculo**, tamaño **pequeño**

- *Hay un círculo pequeño por encima de un cuadrado pequeño. También hay un objeto grande.*

Se puede apreciar que esta última generalización (la encontrada cuando se hace la pequeña *trampa* de pasar los atributos a relaciones unarias y poner un atributo fantasma) es exactamente igual a la encontrada en **[HAYE78]**.

Un aspecto interesante es que con el algoritmo objeto de esta tesina se podrían conseguir descripciones disyuntivas de conceptos. Como se puede llegar a obtener varias generalizaciones a partir de un conjunto de atributos dados, podría devolverse como resultado una disyunción de todas las generalizaciones (o las más *interesantes*) que se encuentren. Por ejemplo, considerando el atributo **tamaño** se puede decir que la generalización encontrada es *"Hay un objeto pequeño dentro de otro objeto pequeño"* o *"Hay un objeto pequeño dentro de un objeto grande"*. Para lograr esto se debería añadir una función que recibiera las generalizaciones obtenidas y las simplificase en una disyunción.

Otro punto a notar es que ni el algoritmo de Hayes-Roth ni el nuestro pueden conseguir la generalización conjuntiva más grande posible debido a la forma en que representamos y tratamos los ejemplos. En este caso sería *Hay tres objetos, incluyendo un círculo pequeño y un cuadrado pequeño. El cuadrado está por encima del círculo. El tercer objeto es grande,* **y contiene uno de los objetos pequeños**. En términos generales se puede decir que nuestro método se asemeja bastante al de Hayes-Roth. En especial, existe una similitud importante entre la primera fase de nuestro algoritmo (creación de todas las posibilidades de substitución de objetos por variables teniendo en cuenta el valor de los atributos que se estén considerando) y el primer paso en el **interference match** de Hayes-Roth.

Para acabar, se puede resumir la comparación del método de Hayes-Roth y McDermott con el nuestro en la tabla 4.2, siguiendo los mismos criterios que con el anterior de Winston.

| **Aspecto** | **Hayes-Roth** | **Am** |
|---|---|---|
| Dominio | General | Expresable con axiomas substitutivos y transitivos |
| Lenguaje | PSR | Propiedades y relaciones binarias |
| Conceptos sintácticos | Case frames, case labels, parámetros | Átomos, listas y listas de propiedades |
| Operadores | AND | AND |
| Reglas de generalización | Eliminar condición<br>Constantes a variables | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización *especial* |
| **Extensibilidad** | | |
| Formas disjuntivas | No | Sí |
| Inmunidad al ruido | Baja | Nula |
| Conocimiento del dominio | No | Declarado en axiomas |
| Inducción constructiva | No | Con axiomas substitutivos |

Tabla 4.2: Comparación con el método de Hayes-Roth y McDermott

### 4.2.4 Método Vere

En su trabajo sobre aprendizaje inductivo ([**VERE75**]), Vere también intenta encontrar las generalizaciones conjuntivas más específicas (en su terminología **maximal conjunctive generalizations** o **maximal unifying generalizations**) de un conjunto de instancias positivas de un concepto. Cada ejemplo se representa como una conjunción de **literales**, donde cada literal es una lista de constantes (**términos**) entre paréntesis. Por ejemplo, las tres instancias que se usan a lo largo de este capítulo se representarían de la siguiente manera:

```
EJ1: (caja a)(circulo b)(rombo c)
     (grande a)(medio b)(medio c)
     (liso a)(sombreado b)(liso c)
     (sobre b a)(sobre c b)

EJ2: (circulo d)(circulo e)(rectangulo f)(cuadrado g)
     (pequeno d)(pequeno e)(grande f)(medio g)
     (sombreado d)(sombreado e)(liso f)(liso g)
     (sobre g f)(dentro d f)(dentro e f)

EJ3: (elipse h)(rectangulo i)(triangulo j)
     (grande h)(medio i)(pequeno j)
```

```
(liso h)(sombreado i)(liso j)
(sobre i h)(sobre j i)
```

Aunque se parezca a la manera de representar los ejemplos de Hayes-Roth con *case frames* es bastante diferente, porque Vere trata todos los símbolos de igual manera. No le da semántica alguna a esta representación, no distingue entre nombres de propiedades (p.e. **grande**) y objetos concretos como **a** o **g**. Este hecho llevará a una serie de problemas que se comentarán posteriormente.

El algoritmo que utiliza para generalizar un par de ejemplos es el siguiente :

1. Creación del conjunto *MP*, que contiene todos los pares de literales que hagan **matching**. Dos literales hacen *matching* si tienen el mismo número de constantes y al menos un término común en la misma posición. En nuestro ejemplo, si se consideran las 2 primeras instancias y se sigue este proceso el conjunto resultante sería :

```
MP={((circulo b),(circulo d))
    ((circulo b),(circulo e))
    ((grande a),(grande f))
    ((medio b),(medio g))
    ((medio c),(medio g))
    ((liso a),(liso f))
    ((liso a),(liso g))
    ((sombreado b),(sombreado d))
    ((sombreado b),(sombreado e))
    ((liso c),(liso f))
    ((liso c),(liso g))
    ((sobre b a),(sobre g f))
    ((sobre c b),(sobre g f))}
```

2. Selección de todos los posibles subconjuntos de *MP* de forma que ningún literal de un ejemplo esté emparejado con más de un literal en otro ejemplo. Cada uno de estos subconjuntos formará una generalización de los ejemplos iniciales al final del proceso. En **[DIET83]** ya se hace notar que este paso puede ser muy costoso, ya que el espacio de posibles subconjuntos de *MP* es muy grande (exponencial con el número de elementos). Con el conjunto *MP* que se acaba de mostrar existen cientos de subconjuntos posibles que cumplen la propiedad pedida.

   En este segundo paso se puede llegar a expresiones que no se podían alcanzar ni en nuestro método ni en ninguno de los métodos que se han comentado hasta ahora. Por ejemplo, algunos de los subconjuntos posibles a partir del conjunto *MP* visto anteriormente serían :

```
S1 = {((medio b)(medio g))
      ((liso a)(liso g))}

S2 = {((liso a)(liso f))
      ((liso c)(liso f))}
```

Como se puede apreciar, se está *ligando*, de alguna manera, dos objetos de un ejemplo con un sólo objeto del segundo ejemplo (**a** y **b** con **g** en el primer caso y **a** y **c** con **f** en el segundo). Como se verá al final del proceso, este hecho hace que en las generalizaciones obtenidas por Vere haya vinculaciones de variables del tipo **many-to-one**, y no **one-to-one** como habíamos visto hasta ahora. En la opinión de Dietterich y Michalski (y de la mayoría de los científicos que se dedican al aprendizaje inductivo), *normalmente este tipo de generalizaciones no tienen sentido, y su generación incontrolada es computacionalmente costosa.*

3. Cada subconjunto de los obtenidos en el paso 2 se extiende añadiéndole nuevos pares de literales. Un nuevo par $p$ se añade a un subconjunto $S$ de $MP$ si cada literal de $p$ está relacionado con algún otro par $q$ de $S$ por una constante común en la misma posición. Por ejemplo, si en un subconjunto $S$ tenemos el par *((cuadrado b),(cuadrado d))*, se podría añadir el par *((sobre a b),(dentro e d))* porque el tercer elemento de *(sobre a b)* es el segundo de *(cuadrado b)* y el tercer elemento de *(dentro e d)* es el segundo elemento de *(cuadrado d)*.

   Si en el segundo paso el espacio de posibles subconjuntos era grande, en este tercer paso todavía lo es más. En **[DIET83]** se comenta que en ninguno de los trabajos publicados por Vere (**[VERE75]**, **[VERE77]**, **[VERE78]**, **[VERE80]**) se describe claramente cómo se efectúan los pasos 2 y 3 de este algoritmo de generalización, pero no debe ser con una búsqueda exhaustiva porque sería muy ineficiente.

4. El conjunto resultante de pares se convierte en una conjunción de literales, uniendo cada par para que forme un literal. Los términos que no hacen matching se transforman en nuevos términos, que pueden ser vistos formalmente como variables. Por ejemplo *((círculo a),(círculo c))* pasaría a ser *(círculo v1)*.

   Esta forma de crear literales y el hecho de no distinguir predicados de constantes hará que se puedan generar cosas extrañas. Antes se ha visto que, en la fase de extensión de los pares de literales, se podían añadir cosas como *((sobre a b),(dentro e d))*. Al hacer ahora la generalización de este par para formar un literal, se obtendría un literal como *(v1 v2 v3)*, que no tiene demasiado sentido tampoco.

En el ejemplo de la figura 4.1 que se está considerando para ir comentado los diversos métodos Vere obtiene muchas generalizaciones, algunas de las cuales son las siguientes :

- **( SOBRE v1 v2 ) ( MEDIO v1 ) ( GRANDE v2 ) ( LISO v2 ) ( LISO v3 ) ( SOMBREADO v4 ) ( v5 v4 )**

  *Hay un objeto de tamaño medio sobre un objeto grande no sombreado. Otro objeto no está sombreado. Hay un objeto sombreado.*

- **( SOBRE v1 v2 ) ( LISO v1 ) ( MEDIO v1 ) ( v9 v1 ) ( v5 v3 v4 ) ( SOMBREADO v3 ) ( v7 v3 ) ( v6 v3 ) ( LISO v4 ) ( GRANDE v4 ) ( v8 v4 )**

  *Hay un objeto no sombreado de tamaño medio sobre otro objeto. Hay dos objetos relacionados de alguna forma tal que uno es sombreado y el otro es grande y no está sombreado.*

- **( SOBRE v1 v2 ) ( MEDIO v1 ) ( LISO v2 ) ( GRANDE v2 ) ( v5 v2 ) ( SOMBREADO v3 ) ( v7 v3 ) ( LISO v4 ) ( v6 v4 )**

  *Hay un objeto de tamaño medio sobre un objeto grande no sombreado. Hay un objeto sombreado y hay un objeto no sombreado.*

Como se puede apreciar a partir de estos resultados, la aparición de literales como *(v5 v4)* o de hasta 7 variables diferentes como en la última generalización lleva a bastante confusión, pero si se eliminan los literales *vacíos* (los que sólo contienen variables), las generalizaciones obtenidas son muy similares a las obtenidas con nuestro generalizador o las de Hayes-Roth.

En la página siguiente se puede encontrar un resumen de las principales diferencias entre nuestra aproximación y la de Vere.

### 4.2.5 Método Michalski-Dietterich

En esta sección se va a comentar el método de determinación de las generalizaciones conjuntivas más específicas descrito por Michalski y Dietterich en **[DIET83]**. Ellos describen los ejemplos de entrada en el lenguaje $VL_{21}$, que es una extensión de la lógica de predicados de primer orden. Cada ejemplo es una conjunción de **selectores**, que normalmente contienen un descriptor de predicados (con variables como argumentos) y una lista de los valores que el predicado puede tener. Otra forma de los selectores son predicados n-arios entre corchetes, que se interpretan de la forma habitual. Los tres ejemplos de la figura 4.1 se representarían de la siguiente forma :

**E1: ∃ v1,v2,v3 [ tamaño (v1) = grande ] [ tamaño (v2) = medio ] [ tamaño (v3) = medio ] [ forma (v1) = caja ] [ forma (v2) = círculo ] [ forma (v3) = rombo ] [ textura (v1) = liso ] [ textura (v2) = sombreado ] [ textura (v3) = liso ] [ sobre (v2,v1) ] [ sobre (v3,v2) ]**

**E2: ∃ v4,v5,v6,v7 [ tamaño (v4) = pequeño ] [ tamaño (v5) = pequeño ] [ tamaño (v6) = grande ] [ tamaño (v7) = medio ] [ forma (v4) = círculo ] [**

| **Aspecto** | **Vere** | **Am** |
|---|---|---|
| Dominio | General | Expresable con axiomas substitutivos y transitivos |
| Lenguaje | Predicados de primer orden sin cuantificadores | Propiedades y relaciones binarias |
| Conceptos sintácticos | Literales, constantes | Átomos, listas y listas de propiedades |
| Operadores | AND | AND |
| Reglas de generalización | Eliminar condición<br>Constantes a variables | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización *especial* |
| **Extensibilidad** | | |
| Formas disjuntivas | Sí | Sí |
| Inmunidad al ruido | Probablemente buena | Nula |
| Conocimiento del dominio | Sí | Declarado en axiomas |
| Inducción constructiva | No | Con axiomas substitutivos |

Tabla 4.3: Comparación con el método de Vere

**forma (v5) = círculo ] [ forma (v6) = rectángulo ] [ forma (v7) = cuadrado ] [ textura (v4) = sombreado ] [ textura (v5) = sombreado ] [ textura (v6) = liso ] [ textura (v7) = liso ] [ dentro (v4,v6) ] [ dentro (v5,v6) ] [ sobre (v7,v6) ]**

**E3: ∃ v8.v9.v10 [ tamaño (v8) = grande ] [ tamaño (v9) = medio ] [ tamaño (v10) = pequeño ] [ forma (v8) = elipse ] [ forma (v9) = rectángulo ] [ forma (v10) = triángulo ] [ textura (v8) = liso ] [ textura (v9) = sombreado ] [ textura (v10) = liso ] [ sobre (v9,v8) ] [ sobre (v10,v9) ]**

En este método se tratan de forma diferente los descriptores unarios (o **descriptores de atributos**) y los no unarios (o **descriptores estructurales**). La idea es primero buscar generalizaciones plausibles en el espacio estructural, y después buscar en el espacio de atributos para llenar los detalles de estas generalizaciones. Básicamente lo hacen así para reducir el espacio de búsqueda de las generalizaciones, al tener en cuenta al principio tan sólo el aspecto estructural de los ejemplos. Esta aproximación es parecida a la seguida en nuestro algoritmo de generalización en el sentido de que también se obtiene la generalización en la parte estructural; la diferencia es el uso de los atributos para guiar la parte de **matching** al ir buscando la generalización.

La parte de obtención de las generalizaciones en el método que se está examinando funciona de la siguiente forma [**DIET83**]. El algoritmo hace una búsqueda del tipo *beam search* ([**RUBI77**]) en el espacio estructural. Esta búsqueda es una forma de buscar primero el mejor (*best-first search*) en la cual se mantiene un conjunto de las mejores descripciones candidatas que se hayan obtenido hasta el momento.

Primero se eliminan todos los descriptores unarios de los ejemplos, quedándose de esta manera sólo con la parte estructural de los mismos. Se escoge un ejemplo de forma aleatoria y se toma como $B_0$, el conjunto inicial de generalizaciones. En cada paso, primero se eliminan de $B_i$ las generalizaciones *menos prometedoras*. El criterio para evaluar las generalizaciones lo puede dar el usuario, y el programa también tiene algunos criterios incorporados, como maximizar el número de ejemplos cubiertos por una generalización o maximizar el número de selectores en una generalización, por ejemplo.

Después se comprueba si alguna de las generalizaciones de $B_i$ cubre todos los ejemplos. Si es así, se pasan de $B_i$ al conjunto C, donde se almacenan las generalizaciones conjuntivas candidatas.

Finalmente, $B_i$ se generaliza a $B_{i+1}$ cogiendo cada elemento de $B_i$ y generalizándolo de todas las maneras posibles eliminando un selector. La búsqueda finaliza cuando el conjunto C llega a un tamaño determinado. El conjunto C contiene generalizaciones conjuntivas de los ejemplos de la entrada, algunas de las cuales son el máximo de específicas.

Una vez se ha construído el conjunto de generalizaciones candidatas, cada una de ellas se ha de completar encontrando valores para sus descriptores de atributos. Cada generalización se usa para definir un espacio de atributos en el que se hace una *beam search* similar a la realizada en el espacio estructural.

Entre todas las generalizaciones conjuntivas producidas por la primera fase del algoritmo puede haber algunas que no sean lo más específicas posibles. En **[DIET83]** se afirma que en la mayoría de los casos estas generalizaciones se vuelven el máximo de específicas cuando se llenan los atributos en la segunda fase del algoritmo.

Algunas de las generalizaciones obtenidas por este método usando los ejemplos de la figura 4.1 son las siguientes :

- **∃ v1,v2 [ sobre (v1,v2) ] [ tamaño (v1) = medio ] [ forma (v1) = polígono ] [ textura (v1) = liso ] [ tamaño (v2) = medio ∨ grande ] [ forma (v2) = rectángulo ∨ círculo ]**

  *Existen 2 objetos en cada ejemplo tal que uno es un polígono de tamaño medio no sombreado que está encima del otro, que es un círculo o un rectángulo de tamaño medio o grande.*

- **∃ v1,v2 [ sobre (v1,v2) ] [ tamaño (v1) = medio ] [ forma (v1) = círculo ∨ cuadrado ∨ rectángulo ] [ tamaño (v2) = grande ] [ forma (v2) = caja ∨ rectángulo ∨ elipse ] [ textura (v2) = liso ]**

  *Existen dos objetos tales que uno de ellos es un círculo, rectángulo o cuadrado de tamaño medio que está sobre el otro, que es una caja, rectàngulo o elipse grande y no sombreado.*

- **∃ v1,v2 [ sobre (v1,v2) ] [ tamaño (v1) = medio ] [ forma (v1) = polígono ] [ tamaño (v2) = medio ∨ grande ] [ forma (v2) = rectángulo ∨ elipse ∨ círculo ]**

*Existen 2 objetos tales que uno de ellos es un polígono de tamaño medio que está sobre el otro, un rectángulo, elipse o círculo de tamaño medio o grande.*

- **∃ v1 [ tamaño (v1) = pequeño ∨ medio ] [ forma (v1) = círculo ∨ rectángulo ] [ textura (v1) = sombreado ]**

  *Existe un objeto, que es un círculo o rectángulo, sombreado y de tamaño medio o pequeño.*

Salta a la vista rápidamente que la principal diferencia de las generalizaciones obtenidas con este método respecto a las de otros métodos reside en las descripciones disjuntivas que obtiene (p.e. objetos que son *rectángulos, elipses o círculos*). En mi opinión puede haber casos en que sean interesantes estos tipos de descripciones, pero en este ejemplo tan sencillo ya se puede ver que produce generalizaciones con interpretaciones un tanto artificiales y difíciles de seguir (p.e. *un rectángulo, elipse o círculo de tamaño medio o grande*).

También se han introducido algunas reglas de inducción constructivas en el sistema, con las que se pueden obtener generalizaciones más informativas, tales como la siguiente :

- **[ número de v's = 3,4 ] [ número de v's con textura liso = 2 ]**

  **∃ v1,v2 [ top-most (v1) ] [ sobre (v1,v2) ] [ tamaño (v1) = medio ] [ forma (v1) = polígono ] [ textura (v1) = clear ] [ tamaño (v2) = medio,grande ] [ forma (v2) = círculo,rectángulo ]**

  *Hay 3 ó 4 objetos en cada ejemplo. De ellos exactamente dos no son sombreados. El objeto en posición más elevada es un polígono claro de tamaño medio, y está sobre un círculo o rectángulo de tamaño grande o medio.*

En la tabla de la página siguiente se puede ver una comparación de las principales características de nuestro método y el que acabamos de describir.

| **Aspecto** | **Michalski** | **Am** |
|---|---|---|
| Dominio | General | Expresable con axiomas substitutivos y transitivos |
| Lenguaje | Predicados de primer orden ampliados, $VL_{21}$ | Propiedades y relaciones binarias |
| Conceptos sintácticos | Selectores, variables, descriptores | Átomos, listas y listas de propiedades |
| Operadores | AND,OR,OR interno | AND |
| Reglas de generalización | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización<br>Cerrar intervalos<br>Generalización por OR interno | Eliminar condición<br>Constantes a variables<br>Subir árbol de generalización *especial* |
| **Extensibilidad** | | |
| Formas disjuntivas | Sí | Sí |
| Inmunidad al ruido | Muy buena | Nula |
| Conocimiento del dominio | Sí | Declarado en axiomas |
| Inducción constructiva | Algunas reglas generales | Con axiomas substitutivos |

Tabla 4.4: Comparación con el método de Michalski

Para cerrar este capítulo, se puede ver qué resultados obtiene Michalski en un ejemplo que muestra en **[MICH80]**. En uno de los ejemplos de este artículo intenta encontrar una descripción de una serie de trenes, formados por una serie de vagones de los cuales interesan las siguientes características :

- **Longitud**: hay vagones cortos y largos.
- **Forma**: forma que tiene el vagón (puede ser una elipse, un rectángulo abierto, un rectángulo cerrado, etc.).
- **Forma de la carga**: un vagón puede transportar círculos, triángulos, cuadrados, etc..
- **Número de partes**: número de unidades de carga que lleva cada vagón. Puede ser 1, 2 ó 3.
- **Número de ruedas**: cada vagón tiene 2 ó 3 ruedas.

En la siguiente figura se pueden ver los trenes que usa Michalski en uno de sus ejemplos en **[MICH80]**.

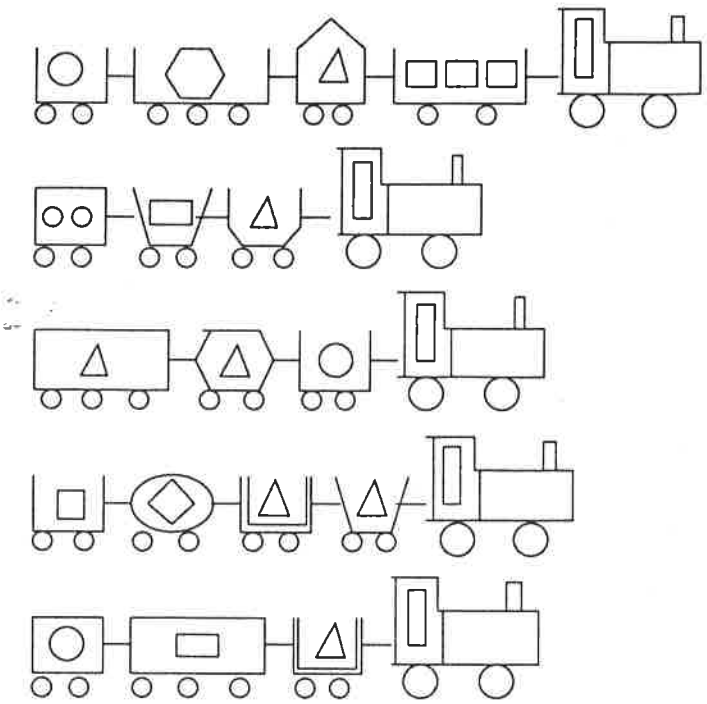


Figura 4.8: Ejemplo de los trenes de Michalski

Las dos descripciones de este tipo de trenes que obtiene Michalski son las siguientes :

- $\exists$ $car_1$ [ **longitud** ($car_1$) = **corto** ] [ **forma** ($car_1$) = **techo cerrado** ]

  *Hay un vagón que es corto y tiene el techo cerrado.*

- $\exists$ $car_1$, $car_2$, $load_1$, $load_2$ [ **delante** ($car_1$,$car_2$) ] [ **lleva** ($car_1$,$load_1$) ] [ **lleva** ($car_2$,$load_2$) ] [ **forma-carga** ($load_1$) = **triángulo** ] [ **forma-carga** ($load_2$) = **polígono** ]

  *Hay un vagón que lleva un triángulo, y el vagón detrás suyo lleva un polígono.*

En el apéndice A se puede ver cómo se codificarían estos ejemplos para dárselos al algoritmo de generalización. Algunos de los resultados que obtiene nuestro generalizador son las siguientes :

- **Con los atributos longitud, forma de la carga, número de partes y número de ruedas**

  - **((PORDET x2 MAQUINA))**

- **Atributos de las variables**

```
x2   longitud: corto
     forma de la carga: triangulo
     numero de partes: 1
     numero de ruedas: 2
```

- *Hay un vagón corto con 2 ruedas que lleva 1 triángulo*

• **Con los atributos longitud, número de partes y número de ruedas**

- **((PORDET x2 MAQUINA)(PORDET x4 x2))**
- **Atributos de las variables**

```
x2   longitud: corto
     numero de partes: 1
     numero de ruedas: 2
x4   longitud: corto
     numero de partes: 1
     numero de ruedas: 2
```

- *Hay un vagón corto con 2 ruedas que lleva 1 sola carga. En algún punto detrás de este vagón hay otro vagón corto con 2 ruedas que también lleva 1 sola carga.*

• **Con el atributo número de partes**

- **((PORDET x2 MAQUINA)(DETRAS x3 x2))**
- **Atributos de las variables**

```
x2   numero de partes: 1
x3   numero de partes: 1
```

- *En algún punto por detrás de la máquina hay un vagón que lleva 1 sola carga. Justo detrás de este vagón hay otro vagón que también lleva 1 sola carga.*

• **Con el atributo número de ruedas**

- **((DETRAS x1 MAQUINA)(PORDET x2 x1))**
- **Atributos de las variables**

```
x1   numero de ruedas: 2
x2   numero de ruedas: 2
```

- *Justo detrás de la máquina hay un vagón con 2 ruedas. En algún punto detrás de este vagón hay otro vagón que también tiene 2 ruedas.*

Si se quiere aplicar la regla de subida por el árbol de generalización al atributo forma de la carga, se sigue el proceso que se ha explicado antes de pasar de atributo a predicado unario y especificar el árbol de generalización con los axiomas substitutivos. Después de este proceso, algunos de los resultados que se obtienen son los siguientes :

- **Con los atributos longitud, número de ruedas y número de partes**
  - **((PORDET x2 MAQUINA) (PORDET x4 MAQUINA) (LLETRIA x2) (LLEPOLI x4))**
  - **Atributos de las variables**

```
x2   longitud: corto
     numero de ruedas: 2
     numero de partes: 1
x4   longitud: corto
     numero de ruedas: 2
     numero de partes: 1
```

  - *En alguna parte detrás de la máquina hay un vagón corto con 2 ruedas que lleva 1 triángulo. También hay otro vagón corto con 2 ruedas que lleva 1 polígono.*

- **Con el atributo número de partes**
  - **((DETRAS x3 x2) (PORDET x2 MAQUINA) (LLETRIA x2) (LLEPOLI x3))**
  - **Atributos de las variables**

```
x2   numero de partes: 1
x3   numero de partes: 1
```

  - *En alguna parte detrás de la máquina hay un vagón que lleva 1 triángulo. Justo detrás de este vagón hay otro vagón que lleva 1 polígono.*

Esta última generalización es exactamente igual a una de las que presenta Michalski en su artículo. Por último, se puede ver qué resultado se obtiene si se pasan todos los atributos a predicados unarios y se crea un atributo *fantasma* para poder aplicarle el algoritmo (en este caso hay 5184 combinaciones posibles de matching entre los diversos objetos) :

- **((DETRAS x1 MAQUINA) (DETRAS x2 x1) (DETRAS x3 x2) (LLEPOLI x1) (LLEPOLI x2) (LLEPOLI x3))**

  *Justo detrás de la máquina hay 3 vagones, y todos ellos transportan polígonos.*

### 4.2.6 Comentarios finales

En general todos los métodos examinados dan resultados bastante parecidos. Todos ellos encuentran en algún punto la dificultad de tener que averiguar qué objetos de cada ejemplo están relacionados entre sí. Ese es, obviamente, el punto clave en cualquier algoritmo de adquisición de conceptos (**concept acquisition**), que ha de calcular la descripción de un concepto a partir de las semejanzas entre los ejemplos que se le presentan.

Winston asume que las redes semánticas que tiene que comparar serán muy similares y, por tanto, el algoritmo de cotejamiento no tendrá que enfrentarse con múltiples posibilidades. Esta idea procede del uso de **cuasiejemplos** (*near-misses*), que se diferencian en tan sólo pequeños detalles de los ejemplos positivos del concepto a aprender. Si Winston admitiese contraejemplos cualesquiera el algoritmo de comparación de redes semánticas se encontraría con bastantes problemas.

En el primer paso del algoritmo de Hayes-Roth se calculan todas las formas de correspondencia posibles a partir de los **case frames** que definen los ejemplos del concepto (que incluyen tanto relaciones entre objetos como propiedades de los mismos), pudiendo generarse por lo tanto múltiples combinaciones.

En el algoritmo de Vere se hace algo similar, ya que en el primer paso se construye un conjunto con todos los pares de **literales** que compartan un **término** en la misma posición. Aquí ya puede haber muchas posibilidades, pero en el siguiente paso normalmente se incrementa este número, ya que se estudian todos los posibles subconjuntos del conjunto de pares de literales. Los literales también engloban tanto relaciones entre objetos como propiedades.

En el método de Michalski el aspecto más interesante es que se busca la generalización en el espacio estructural, sin tener en cuenta los atributos en una primera fase. Eso hace que se reduzca el número de posibilidades respecto a los métodos anteriores.

El algoritmo de generalización de esta tesina también empieza generando todas las posibles combinaciones entre objetos de diversos ejemplos, pero ya se reduce (en la mayoría de los casos de una forma importante) el número de combinaciones generadas al exigir que los objetos que sean substituidos por la misma variable tengan los mismos valores en todos los atributos que se estén considerando. Esta obligación hace que el algoritmo de generalización no tenga ninguna protección contra el ruido en los datos, ya que un solo fallo en el valor de un atributo en un objeto de un ejemplo puede hacer que toda la generalización no se descubra por no haber susbstituido aquel objeto por la variable adecuada. En este tipo de aprendizaje supervisado el ruido debe ser eliminado por el profesor. En otro caso el sistema *aprenderá* con la información suministrada el *mejor* de los conceptos aprendibles.

# Capítulo 5

# Constructor de jerarquías

## 5.1 Introducción

Como ya se vió en capítulos anteriores, la motivación principal del algoritmo de generalización es desarrollar una de las partes del algoritmo de aprendizaje de conceptos definido por Núñez en [NÚÑE91]. En este capítulo se presenta otra posible aplicación de las generalizaciones obtenidas con ese algoritmo: organizarlas de forma jerárquica de manera que puedan ser usadas por otros programas a los que les interesara tener esa información estructurada.

## 5.2 Construcción de jerarquías

Con el algoritmo se podía obtener la descripción de un concepto a partir de una serie de ejemplos positivos del mismo. Esta descripción siempre se hace en base a bloques básicos. En este sentido se puede decir que sólo se trabaja con objetos *simples*.

Por ejemplo, supóngase que el algoritmo conoce el concepto de **min-arco** descrito en la siguiente figura :

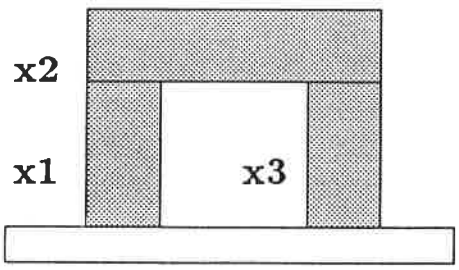



Figura 5.1: Min-arco

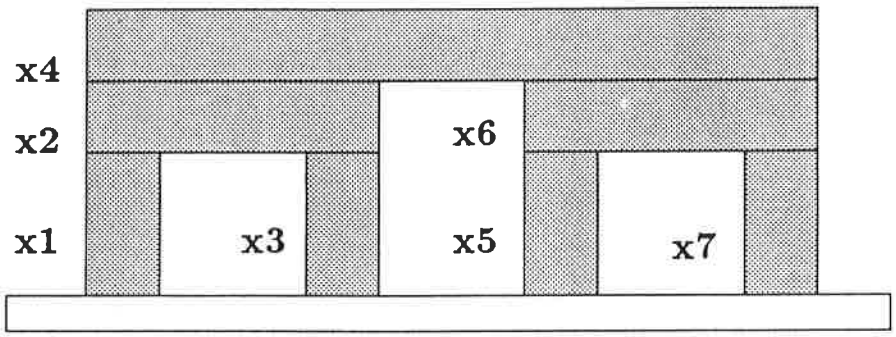



Figura 5.2: Superarco

Cuando el generalizador describe el concepto de **superarco** de la figura 5.2 lo hace en función de bloques básicos, obteniendo por lo tanto una descripción bastante menos compacta y comprensible que si estuviera en términos de **min-arcos**.

La idea del constructor de jerarquías es describir los conceptos más *complejos* en función de conceptos más sencillos conocidos previamente. Esta dependencia de unos conceptos con otros se podría expresar de forma jerárquica, por ejemplo con un grafo acíclico dirigido como se puede ver en la figura 5.3 .

Se puede almacenar la información asociada a cada nodo del grafo en la lista de propiedades de un átomo. Esta información incluye :

- Nombre del concepto
- Conceptos que aparecen en la descripción del concepto (antecesores en el grafo).
- Conceptos en la descripción de los cuales aparece el concepto (sucesores en el grafo).
- Número de objetos presentes en la descripción del concepto.
- Atributos tenidos en cuenta para obtener la descripción.
- Descripción del concepto.

La idea inicial era, simplemente, descubrir estos conceptos elementales dentro de los conceptos más complejos, hecho que permitiría una reducción del conocimiento a almacenar y una mejor comprensión del mismo (p.e. es mucho más fácil entender un **superarco** formulado en términos de **arcos** que en términos de relaciones entre bloques simples). Se consideraron 2 posibilidades para hacerlo :

- Buscar los conceptos ya conocidos en los ejemplos positivos del concepto a aprender, modificar estos ejemplos y entonces aplicarles un algoritmo de generalización adecuado.

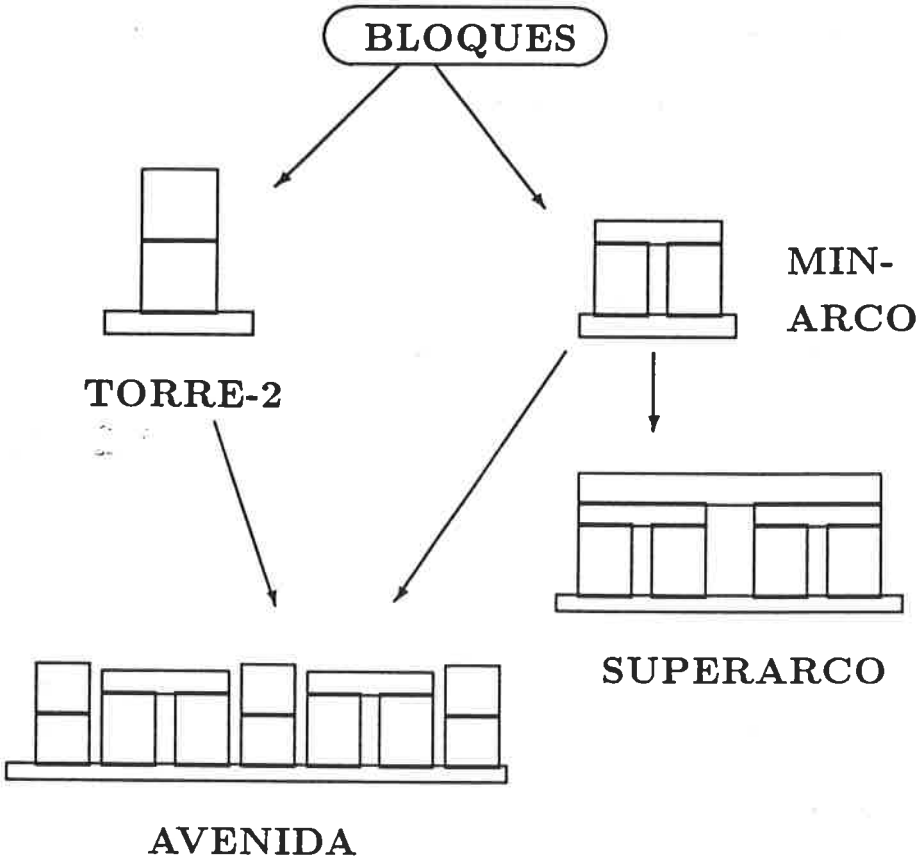



Figura 5.3: Posible jerarquía de conceptos

- Aplicar el algoritmo de generalización a los ejemplos positivos del concepto, tal y como se ha hecho hasta ahora, y buscar los conceptos simples en la expresión generalizada resultante.

Se escogió esta segunda aproximación por 2 motivos :

- No se ha de modificar el algoritmo generalizador realizado previamente.
- Sólo se han de buscar los conceptos conocidos en la expresión generalizada (1 vez), y no en todos los ejemplos positivos disponibles del concepto que tengamos ($n$ veces).

Una vez decidido este enfoque, a primera vista, existen dos maneras de buscar los conceptos sencillos dentro de los nuevos conceptos :

- **Ascendente**

  Se buscan primero los conceptos más complicados conocidos (aquellos que tengan más objetos, por ejemplo) en la expresión generalizada. Si falla la búsqueda se

pasa a buscar conceptos conocidos más sencillos, y si se encuentran se reescribe la expresión generalizada en términos del concepto conocido y se buscan los conceptos más sencillos en la parte restante de la generalización.

Este enfoque tiene la ventaja de agrupar muchos objetos al principio si encuentra los conceptos más complicados dentro de la generalización. Las desventajas principales son que:

1. Es más costoso encontrar los conceptos grandes que los pequeños dentro de la generalización y,
2. Hay que buscar conceptos conocidos mientras haya alguna parte de la generalización que no ha sido agrupada bajo ningún otro concepto.

En el ejemplo anterior significaría buscar primero si hay **avenidas** en el nuevo concepto, después mirar si hay **superarcos**, etc. hasta llegar a la raíz de la jerarquía.

- **Descendente**

  Aquí se empieza buscando los conceptos sencillos y se avanza hacia los más complejos, substituyendo en la descripción en curso los objetos simples por los complejos cuando se vayan encontrando.

  Las ventajas de hacerlo así es que es menos costoso encontrar conceptos sencillos que complejos, que se pueden encontrar los conceptos complejos ayudándose de los lugares donde se han encontrado previamente los simples y que se puede podar la búsqueda cuando no se encuentre un concepto determinado. Siguiendo el ejemplo anterior, si en el nuevo concepto no hay **min-arcos** no hace falta buscar **avenidas** o **superarcos**, y si hay **min-arcos** su posición ayudará a ver si hay **avenidas** o **superarcos** más fácilmente que si se hubieran de buscar sin ninguna ayuda.

  La desventaja es que una substitución no es definitiva hasta el final del proceso. Por ejemplo, si se encuentran en la descripción del nuevo concepto 2 **min-arcos** no se sabe si son simplemente **min-arcos** o bien se han encontrado porque forman parte de un **superarco** que se encontrará después.

Finalmente se implementó el algoritmo siguiendo esta segunda aproximación, a causa de todas las ventajas que ofrece con respecto a la primera opción.

## 5.3 Algoritmo

El constructor de jerarquías tiene como entrada la expresión generalizada que describe un concepto y ha de reformular esta expresión teniendo en cuenta los conceptos que ya conozca, comenzando por los conceptos simples hasta llegar a los más complejos. Las posibilidades que se le pueden presentar en un momento dado de su ejecución son las siguientes :

1. **Expresión simple y concepto simple**

   En este caso el algoritmo ha de investigar si aparece un concepto simple (sólo compuesto por bloques, sin que intervengan conceptos más pequeños) en una expresión generalizada en la cual todavía no se ha substituido ninguna de sus partes por otro concepto (está toda ella, por tanto, expresada en función de relaciones entre bloques). Este sería el primer caso que se presentaría siempre.

   En este punto se inicia un proceso de búsqueda si hay más objetos en la expresión generalizada que en el concepto que se va a buscar (obviamente, si hay menos objetos en la generalización no es posible encontrar nunca el concepto que se está buscando). Otra condición que se impone para buscar posibles apariciones del concepto en la generalización es que el conjunto de atributos tenido en cuenta al obtener la generalización incluya el conjunto de atributos con los que se definió el concepto a buscar.

   El proceso de búsqueda se realiza de forma exhaustiva, y su objetivo es ver si existe una substitución de variables por variables que aplicada a la descripción del concepto dé como resultado un conjunto de relaciones que esten incluídas en la expresión generalizada. Esta substitución ha de tener en cuenta, obviamente, los valores de los atributos de las variables.

2. **Expresión compleja y concepto simple**

   En este caso se trata de buscar un concepto que está descrito sólo con bloques en una expresión generalizada en la que ya se habían encontrado (y, por tanto, substituido) algún otro concepto simple. Por ejemplo buscar **arcos** en una expresión donde ya se había encontrado **torre-2s**.

   El proceso que se sigue en este caso es aplicar la misma búsqueda que en el caso anterior pero usando la parte de la expresión generalizada que no se haya substituído ya.

3. **Expresión simple y concepto complejo**

   Este caso no puede darse en nuestro algoritmo, porque si no se ha encontrado conceptos simples dentro de la expresión (por eso sigue siendo una expresión generalizada simple) seguro que tampoco aparecerán los conceptos más complejos. En nuestro ejemplo, si no se ha encontrado ni **torre-2s** ni **arcos** no se buscarán ni **superarcos** ni **avenidas** (ningún sucesor de **torre-2** o de **arco**).

4. **Expresión compleja y concepto complejo**

   En este caso es cuando se ha tenido en cuenta la desventaja mencionada antes en la aproximación descendente. En este punto es donde se ha de averiguar si las apariciones de conceptos ya detectadas son válidas por sí mismas o son simplemente parte de conceptos mayores que todavía no se han buscado. Aquí también se hace una búsqueda exhaustiva similar a la que se tenía en los dos primeros casos tratados.

Concretando un poco más, el proceso de búsqueda para ver si un concepto está presente o no en la descripción del nuevo concepto que se tiene hasta el momento es el siguiente (donde *c1* es el concepto conocido buscado, y *c2* es la descripción actual del nuevo concepto):

1. Si *c2* tiene menos objetos que c1 no es necesario seguir ya que es imposible que *c2* esté contenido en *c1*.

2. Para cada variable de *c1*, se examinan qué variables de *c2* tienen los mismos valores en los atributos considerados. Si hay alguna variable de *c1* que no tiene ninguna variable de *c2* con los mismos valores de los atributos no hace falta seguir, ya que eso significa que *c1* no puede aparecer en *c2*. Esto implica que el conjunto de atributos usado para obtener la generalización de *c1* ha de ser un subconjunto del conjunto de atributos usados en la generalización de *c2*.

3. Comprobar para cada substitución de las variables de *c1* por las variables con los mismos valores en los atributos en *c2* si las relaciones obtenidas están incluídas o no en la descripción de *c2*. Cada substitución que satisfaga esta condición estará indicando la presencia de una instancia de *c1* en *c2*. De todas formas, si hay varias substituciones que satisfagan la condición pero que tengan variables en común, eso significa que *c1* no aparece *correctamente* en *c2*, y por tanto esas substituciones deben ser rechazadas (un ejemplo de este caso se ve en la próxima sección).

4. Si *c1* aparece una o más veces en *c2*, reescribir la descripción de *c2* en términos de *c1*.

Después de haber considerado todos los conceptos conocidos previamente, el programa integra el nuevo concepto en su posición en la jerarquía, como sucesor de todos aquellos conceptos que estén incluídos en él.

## 5.4 Ejemplo

Vamos a ilustrar el funcionamiento del constructor de jerarquías con el mismo ejemplo que en [**MORE92b**]:

1. El algoritmo de generalización le da al programa la descripción de una torre con dos bloques (después de haber visto unas cuantas instancias positivas del mismo), que se llamará **torre-2** :

   **Torre-2:**

   **((ON x1 mesa) (ON x2 x1))**

   Suponiendo que el constructor de jerarquías no conociera ningún concepto en aquel momento, no tiene que buscar nada y simplemente pondrá este concepto como sucesor

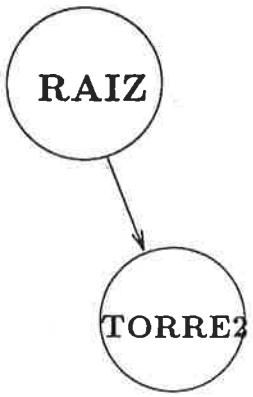



Figura 5.4: Jerarquía después de incluir TORRE-2

del nodo raíz de la jerarquía, que es fijo. Por tanto, la jerarquía en este punto tendría el siguiente aspecto :

2. El algoritmo de generalización obtiene la siguiente descripción después de examinar arcos con 3 bloques (que se llamará **min-arco**) :

   **MIN-ARCO:**

   **((ON x1 mesa) (ON x2 x1) (LEFT* x1 x3) (ON x2 x3) (ON x3 mesa))**

   El constructor de jerarquías intenta ver si encuentra **torre-2s** dentro de esta descripción (se asume en todo el ejemplo que no hay ningún problema con los valores de los atributos). La búsqueda falla por un detalle que ya se mencionó: el algoritmo se da cuenta de que **((ON x1 mesa) (ON x2 x1))** y **((ON x3 mesa) (ON x2 x3))** están definiendo dos apariciones de **torre-2** en **min-arco**, pero el programa también se da cuenta de que x3 aparece en ambas. El programa está diseñado de tal manera que si encuentra la misma variable varias veces cuando busca un concepto conocido en la nueva descripción asume que el viejo concepto no está presente en el nuevo. Se hace así porque sino tendría que seleccionar una de las **torre-2s** para escribir la nueva descripción de **min-arco**. Por tanto, lo único que puede hacer es poner **min-arco** como sucesor de la raíz en la jerarquía, que pasa a ser la de la figura 5.5 (ver en la página siguiente).

3. Se proporciona al constructor de jerarquías la descripción de un arco como el de la figura 3.1 :

   **ARCO:**

   **((ON x1 mesa) (LEFT* x1 x4) (ON x2 x1) (LEFT* x2 x5) (ON x3 x2) (ON x3 x5) (ON x5 x4) (ON x4 mesa)).**

   El algoritmo empieza buscando apariciones de **torre-2s** en esta descripción. Encuentra dos, definidas por las variables **(x1, x2)** y **(x4, x5)** y, por tanto, reescribe la definición de **arco** de la siguiente forma :

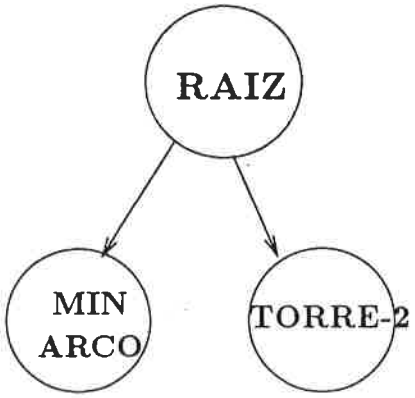



Figura 5.5: Jerarquía después de incluir MIN-ARCO

**ARCO:**

**((ES t1 TORRE-2) (ES t2 TORRE-2) (LEFT* t1 t2) (ON t1 mesa) (ON x3 t1) (ON t2 mesa) (ON x3 t2))**

En este punto el programa intentaría encontrar **min-arcos** en esta descripción, pero no encontraría ninguno porque, en su estado actual, no generaliza de bloques a objetos, **min-arco** está definido en términos de objetos, y **arco** ha sido reescrito en términos de **torre-2s**.

Se puede anotar que el programa no hubiera encontrado **min-arcos** en **arco** aunque los hubiera buscado antes que las **torre-2s**. Las variables x2, x3 y x5 de **arco** no definen un **min-arco** porque no están sobre la mesa. La mesa actúa como un punto fijo que es muy útil como referencia para todos los objetos de la escena. La importancia de tener un punto fijo en la descripción de las escenas fue puesta de manifiesto por Cortés en [**CORT84**], ya que evita la obtención de descripciones demasiado generales.

La jerarquía tendría en este punto el aspecto de la figura 5.6 .

4. Para acabar este ejemplo del constructor de jerarquías vamos a ver qué sucede cuando le damos el concepto de la figura 5.7, que se denomina **avenida**. La descripción inicial sería la siguiente :

   **AVENIDA:**

   **((ON x1 mesa) (ON x2 x1) (LEFT* x1 x3) (LEFT* x2 x4) (ON x3 mesa) (ON x4 x3) (LEFT* x3 x7) (LEFT* x4 x6) (ON x5 x4) (ON x5 x6) (ON x6 x7) (LEFT* x6 x9) (LEFT* x7 x8) (ON x7 mesa) (ON x8 mesa) (ON x9 x8) (LEFT* x8 x10) (LEFT* x9 x11) (ON x10 mesa) (ON x11 x10) (LEFT* x10 x14) (LEFT* x11 x13) (ON x12 x11) (ON x12 x13) (ON x13 x14) (LEFT* x13 x16) (ON x14 mesa) (LEFT* x14 x15) (ON x15 mesa) (ON x16 x15))**

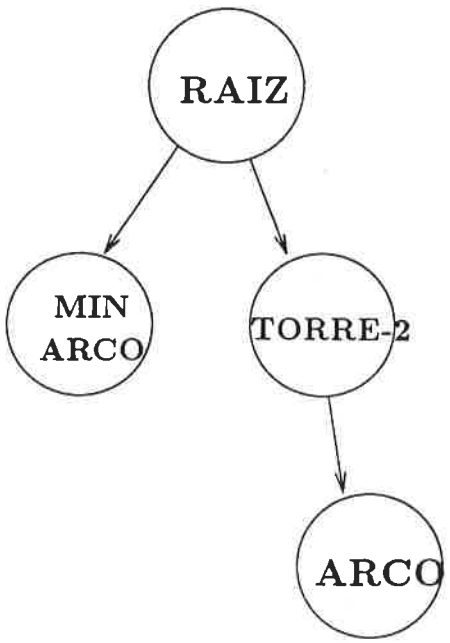



Figura 5.6: Jerarquía después de incluir ARCO

El programa empieza buscando **torre-2s** en este concepto, y descubre 7 de ellas, definidas por las variables **(x1, x2), (x3, x4), (x7, x6), (x8, x9), (x10, x11), (x14, x13), (x15, x16)**. Por lo tanto, reescribe la definición de avenida de la siguiente manera :

**AVENIDA:**

**((ES t1 TORRE-2) (ON t1 mesa) (ES t2 TORRE-2) (ON t2 mesa) (LEFT* t1 t2) (ES t3 TORRE-2) (ON t3 mesa) (ON x5 t2) (ON x5 t3) (LEFT* t2 t3) (ES t4 TORRE-2) (ON t4 mesa) (LEFT* t3 t4) (ES t5 TORRE-2) (ON t5 mesa) (LEFT* t4 t5) (ES t6 TORRE-2) (ON t6 mesa) (LEFT* t5 t6) (ON x12 t5) (ON x12 t6) (ES t7 TORRE-2) (ON t7 mesa) (LEFT* t6 t7))**

Aquí el programa empieza a buscar **min-arcos** en esta descripción, pero no puede encontrar ninguno por la misma razón por la que no podía encontrarlos en **arco**. El programa sabe que si ha encontrado **torre-2s** en la descripción del nuevo concepto ha de buscar sus sucesores también, en este caso **arcos**. Busca los **arcos** con la ayuda de los sitios donde ha encontrado **torre-2s** y encuentra 2 de ellas (con las **torre-2s** *t2* y *t3* y el objeto *x5* y también con las **torre-2s** *t5* y *t6* con el objeto *x12*), reescribiendo de nuevo la descripción de **avenida** :

**AVENIDA:**

**((ES t1 TORRE-2) (ON t1 mesa) (ES t8 ARCO) (ON t8 mesa) (LEFT* t1 t8) (ES t4 TORRE-2) (LEFT* t8 t4) (ON t4 mesa) (ES t9 ARCO) (ON**

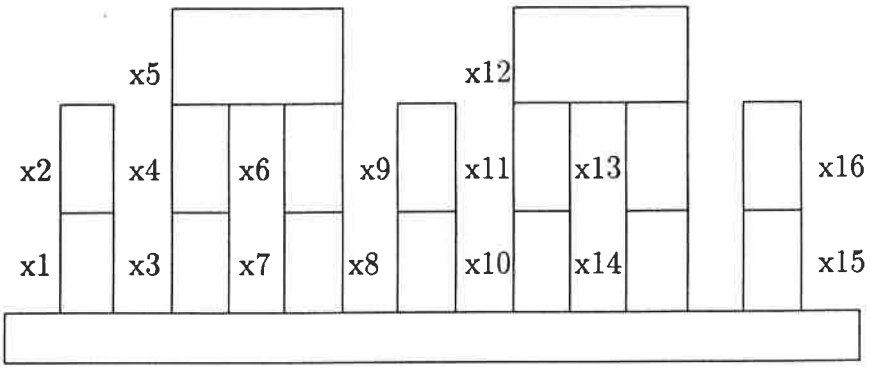



Figura 5.7: Avenida

**t9 mesa) (LEFT* t4 t9) (ES t7 TORRE-2) (ON t7 mesa) (LEFT* t9 t7)).**

Por lo tanto, el concepto **AVENIDA** se ha de situar en el grafo como sucesor de **TORRE-2** y de **ARCO**, obteniendo el resultado que se puede ver en la figura 5.8.

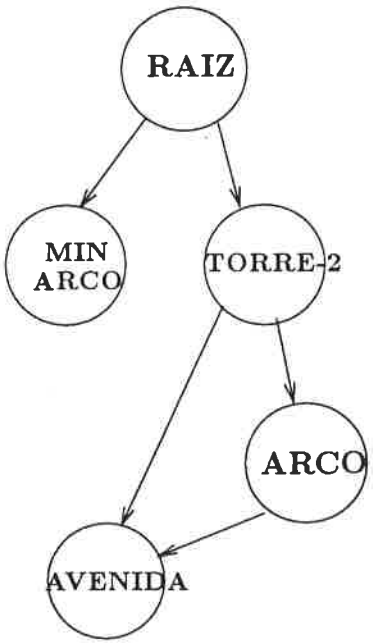



Figura 5.8: Jerarquía después de incluir AVENIDA

# Capítulo 6

# Conclusiones y trabajo futuro

## 6.1 Conclusiones

Las principales aportaciones de este trabajo son las siguientes:

- El desarrollo del algoritmo de generalización pone de manifiesto que el algoritmo teórico de aprendizaje basado en similitudes propuesto por Núñez en [**NÚÑE91**] es implementable en la práctica, ya que éste es su parámetro principal. En algún punto se han introducido heurísticas para controlar la eficiencia en las partes más costosas, como sólo generar un número de variables igual al número de objetos del ejemplo más pequeño o la función que valora si una posibilidad de substitución de constantes por variables dará lugar a una generalización interesante o no.

- El constructor de jerarquías muestra que las generalizaciones obtenidas se pueden usar en otros contextos, como el generar una base de datos estructurada (almacenando la información en frames, por ejemplo) donde se incluyan todos los conceptos conocidos así como las relaciones de interdependencia entre los mismos.

- El método empleado para obtener las generalizaciones es en algunos aspectos bastante diferente de los métodos clásicos ([**WINS75**], [**VERE75**], [**HAYE77**], [**DIET81**]). La principal diferencia radica en la importancia que se le otorga a los atributos, que guían el proceso de construcción de las generalizaciones usando la parte estructural de las instancias del concepto.

- El hecho de que el proceso de generalización dependa de los atributos considerados hace que se puedan llegar a resultados bastante diferentes usando las mismas instancias de un concepto. Este aspecto acerca el proceso de generalización al comportamiento humano delante de ejemplos más claramente que cualquiera de los otros métodos desarrollados hasta el momento. Además, los resultados que pedía Núñez en su tesis son sólo un pequeño subconjunto de los que puede obtener el algoritmo de generalización desarrollado en este trabajo.

## 6.2 Trabajo futuro

Algunos de los puntos en los que el trabajo desarrollado podría ser mejorado o extendido son los siguientes:

- Se podría pasar a implementar todo el algoritmo de aprendizaje propuesto por Núñez en **[NÚÑE91]**. La parte más importante que faltaría por hacer es el algoritmo de simplificación de fórmulas lógicas (**Arf**) que tiene como parámetro.
- Los algoritmos desarrollados se podrían mejorar en los siguientes puntos:
  - El algoritmo de generalización todavía crea todas las posibilidades de substitución de objetos por variables, a pesar de que no busca la generalización asociada a cada posibilidad. Es posible que se pueda encontrar alguna manera más hábil de evitar la generación de todas las posibilidades completamente. Este hecho sería interesante de cara a usar menos memoria y también porque las pruebas han puesto de manifiesto que la mayoría de las posibilidades no son útiles en la fase de generalización.
  - Se han de hacer más pruebas con dominios diferentes al mundo de los bloques. Valdría la pena estudiar si la limitación de uso del generalizador (a dominios cuyos predicados se puedan expresar con axiomas de substitución y transitivos) es muy grande o hay realmente muchos dominios interesantes que sean caracterizables de esta forma. También se podría incluir alguna forma más potente para expresar los axiomas que rigen el comportamiento de las relaciones.
  - Puede ser interesante probar funciones heurísticas diferentes para seleccionar las posibilidades que se obtienen en el primer paso del generalizador.
  - Se podría incorporar un tratamiento más adecuado de los atributos, preguntando al usuario cuales quiere considerar o teniendo alguna forma de saber qué atributos son los más importantes. Hasta el momento están en una lista sin orden ninguno. Existen trabajos **[BELA91]** que muestran la importancia de considerar la relevancia de los atributos y su impacto al reducir los espacios de búsqueda.
  - El constructor de jerarquías se debería generalizar para que tratase de forma uniforme tanto los objetos *simples* como los *complejos*. También se podría pensar alguna manera más eficiente de comprobar si un concepto forma parte de otro concepto o no.
  - El constructor de jerarquías podría estar conectado con un sistema de representación del conocimiento potente que le permitiera generar la jerarquía de conceptos de forma que este conocimiento fuera utilizable de forma sencilla y eficiente por otro programas.

# Bibliografía


[BÉJA92] Béjar, J., Cortés, U., "**LINNEO+: Herramienta para la adquisición de conocimiento y generación de reglas de clasificación en dominios poco estructurados**", *Proceedings del III Congreso Iberoamericano de Inteligencia Artificial IBERAMIA-92, La Habana (Cuba)*, pp. 471-482.

[BELA91] Belanche,Ll., "**To be or nought to be: una qüestió irrellevant?**", tesis de licenciatura, FIB, UPC, 1991.

[BUCH78] Buchanan, B., Feigenbaum, E., "**DENDRAL and META-DENDRAL: their applications dimension**", *Artificial Intelligence*, Vol.11, pp. 5-24, 1978.

[CORT84] Cortés, U., "**Esquema multinivel para la adquisición, representación y tratamiento interactivo del conocimiento en escenarios 2D y 3D**", tesis doctoral, FIB, UPC, 1984.

[CREI88] Creiner, R. et alt., "**Analogica**", Arman Prieditis Ed., Morgan Kaufmann, 1988.

[DAVI87] Davis,L. (Ed.), "**Genetic algorithms and simulated annealing**", Pitman, 1987.

[DIET79] Dietterich, T., "**The methodology of knowledge layers for inducing descriptions of sequentially ordered events**", Tesis de Licenciatura, University of Illinois, Urbana, Octubre 1979.

[DIET81] Dietterich, T., Michalski, R., "**Inductive Learning of Structural Descriptions**", *Artificial Intelligence*, Vol. 16, 1981.

[DIET83] Dietterich, T., Michalski, R.,"**A comparative review of selected methods for learning from examples**" , *Michalski, R., Carbonell, J., Mitchell, T.(Eds.) Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, 1983.

[HAYE77] Hayes-Roth, F. ,McDermott J., "**Knowledge acquisition from structural descriptions**" , *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, IJCAI, Cambridge, Mass. , pp. 356-362, 1977.

[HAYE78] Hayes-Roth, F., McDermott J., "**An Interference Matching Technique for Inducing Abstractions**" , *Communications of the ACM*, Vol. 21, No. 5, Mayo 1978.

[HOLL75] Holland, "**Adaptation in Natural and Artificial Systems**", University of Michigan Press, 1975.

[LANG83] Langley, P., Simon, H., "**Rediscovering chemistry with the BACON system**", *Michalski, R., Carbonell, J., Mitchell, T. (Eds.) Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, 1983.

[LENA83] Lenat, D., "**The role of heuristics in learning by discovery: three case studies**", *Michalski, R., Carbonell, J., Mitchell, T. (Eds.) Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, 1983.

[MART91] Martín, M., "**LINNEO: Eina per l'ajut en la construcció de bases de coneixements en dominis poc estructurats**", Tesis de Licenciatura, FIB, UPC, 1991.

[McCA80] McCarthy, J., "**Circumscription - A form of non-monotonic reasoning**", *Artificial Intelligence* 13, 1980, pp. 27-39.

[MICH80] Michalski, R., "**Pattern recognition as rule-guided inference**", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, No. 4, pp. 349-361, 1980.

[MICH83] Michalski, R., "**A theory and methodology of inductive learning**", *Michalski, R., Carbonell, J., Mitchell, T. (Eds.) Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann, 1983.

[MITC82] Mitchell, T.M., "**Generalization as search**" , *Artificial Intelligence* 18 (2), 1982, pp. 203-226.

[MITC86] Mitchell, T.M., Keller R.M., Kedar-Cabelli, S.T., "**Explanation-based generalization: a unifying view**", *Machine Learning* 1(1), p. 47, 1986.

[MORE92a] Moreno, A., López, J., Pérez, J. , "**Aprenentatge i representació del coneixement**", *trabajo para el Curso de doctorado sobre Adquisición y Representación del Conocimiento*, LSI, UPC, junio 1992 (comunicación).

[MORE92b] Moreno, A., "**A generaliser applied to SBL**", *Proceedings of the Fifth International Symposium on Knowledge Engineering*, Sevilla, 1992 (por aparecer).

[NILS80] Nilsson, N., "**Principles of Artificial Intelligence**", Tioga Publishing Co., 1980.

[NÚÑE90] Núñez, G., Cortés, U., "**Aprendizaje SBL usando inferencia no monótona**", *Reporte de Investigación LSI-RT-90-02*, FIB, UPC, 1990.

[NÚÑE91] Núñez, G., "**Caracterización no monótona de la inferencia inductiva y su aplicación al aprendizaje basado en similitudes (SBL)**", Tesis doctoral, FIB, UPC, 1991.

[RUBI77] Rubin, S., Reddy, R., "**The locus mode of search and its use in image interpretation**", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, IJCAI, pp. 590-595, 1977.

[VERE75] Vere, S., "**Induction of concepts in the predicate calculus**", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, IJCAI, Tbilisi, USSR, pp.281-287, 1975.

[VERE77] Vere, S., "**Induction of relational productions in the presence of background information**", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, IJCAI, Cambridge, Mass. , pp.349-355, 1977.

[VERE78] Vere, S., "**Inductive learning of relational productions**", *Pattern-Directed Inference Systems*, Waterman, D. y Hayes-Roth, F. (Eds.), Academic Press, New York, 1978.

[VERE80] Vere, S., "**Multilevel counterfactuals for generalizations of relational concepts and productions**", *Artificial Intelligence*, Vol. 14, No. 2, pp. 138-164, Septiembre 1980.

[WINS70] Winston, P.H., "**Learning structural descriptions from examples**", *Technical Report AI-TR-231*, MIT, Cambridge, Mass., Septiembre 1970.

[WINS75] Winston, P.H., "**Learning structural descriptions from examples**", en Winston, P.H. (Ed.) *The psychology of computer vision*, McGraw-Hill, New York, 1975.

[WINS82] Winston, P.H., "**Learning new principles from precedents and exercises**", *Artificial Intelligence*, No. 19, pp. 321-350, 1982.

[ZHON92] Zhongzhi, S., "**Principles of Machine Learning**", International Academic Publishers, 1992.

# Apéndice A

# Ejemplos y resultados

## A.1 Combinaciones después de asignar variables en el ejemplo de la figura 3.4 al considerar el atributo COLOR

```
((((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
```

```
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
```

```
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
```

```
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
  ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
  ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
```

```
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X1 MESA) (ON X2 X1) (ON O14 X2) (ON X3 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X1 MESA) (ON O13 X1) (ON X2 O13) (ON X3 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
 ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
```

```
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
```

```
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X1 MESA) (ON X2 X1) (ON X3 X2) (ON O11 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O10 X2) (ON X3 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
```

```
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X3 MESA) (ON X2 X3) (ON X1 X2) (ON O11 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X1 X2) (ON X3 X1))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
```

```
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON X2 X1) (ON O6 X2) (ON X3 O6))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X1 MESA) (ON O5 X1) (ON X2 O5) (ON X3 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
 ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
(((ON X1 MESA) (ON X2 X1) (ON X3 X2))
 ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
 ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
```

```
  ((ON X3 MESA) (ON X2 X3) (ON O14 X2) (ON X1 O14)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O6 X2) (ON X1 O6))
  ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON X3 MESA) (ON X2 X3) (ON O10 X2) (ON X1 O10))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2)))
 (((ON X1 MESA) (ON X2 X1) (ON X3 X2))
  ((ON X3 MESA) (ON O5 X3) (ON X2 O5) (ON X1 X2))
  ((ON O8 MESA) (ON X2 O8) (ON X3 X2) (ON X1 X3))
  ((ON X3 MESA) (ON O13 X3) (ON X2 O13) (ON X1 X2))))
```

Generalización obtenida para cada posibilidad:

```
(((ON X1 MESA) (ON X2 X1) (SOBRE X3 X2))
 ((ON X1 MESA) (ON X2 X1) (SOBRE X3 X2) (TOP X3))
 ((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2))
 ((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2) (TOP X3))
 ((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2))
 ((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2) (TOP X3))
 ((ON X1 MESA) (SOBRE X2 X1) (ON X3 X2))
 ((ON X1 MESA) (SOBRE X2 X1) (SOBRE X3 X2) (TOP X3))
  NIL NIL NIL NIL NIL NIL NIL NIL NIL
 ((SOBRE X3 X2) (TOP X3))
  NIL
 ((SOBRE X3 X2) (TOP X3))
  NIL
 ((SOBRE X3 X2) (TOP X3))
  NIL
 ((SOBRE X3 X2) (TOP X3))
  NIL NIL NIL NIL NIL NIL NIL NIL NIL
 ((SOBRE X3 X2))
  NIL
 ((SOBRE X3 X2))
  NIL
 ((SOBRE X3 X2))
  NIL
```

```
((ON X3 X2))
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL NIL NIL)
```

Valores del atributo en las variables :

```
X1: color a
X2: color b
X3: color a
```

## A.2 Descripción del ejemplo de la figura 4.1

```
;;; Ejemplo de Dietterich-Michalski en Machine Learning I

(setf (get 'o1 'shape) 'box)
(setf (get 'o1 'size)  'large)
(setf (get 'o1 'texture)  'blank)

(setf (get 'o2 'shape) 'circle)
(setf (get 'o2 'size)  'medium)
(setf (get 'o2 'texture)  'shaded)

(setf (get 'o3 'shape) 'diamond)
(setf (get 'o3 'size)  'medium)
(setf (get 'o3 'texture)  'blank)

(setf (get 'o4 'shape) 'circle)
(setf (get 'o4 'size)  'small)
(setf (get 'o4 'texture)  'shaded)

(setf (get 'o5 'shape) 'circle)
(setf (get 'o5 'size)  'small)
(setf (get 'o5 'texture)  'shaded)

(setf (get 'o6 'shape) 'rectangle)
(setf (get 'o6 'size)  'large)
(setf (get 'o6 'texture)  'blank)

(setf (get 'o7 'shape) 'square)
(setf (get 'o7 'size)  'medium)
```

```
(setf (get 'o7 'texture)  'blank)

(setf (get 'o8 'shape) 'ellipse)
(setf (get 'o8 'size)  'large)
(setf (get 'o8 'texture)  'blank)

(setf (get 'o9 'shape) 'rectangle)
(setf (get 'o9 'size)  'medium)
(setf (get 'o9 'texture)  'shaded)

(setf (get 'o10 'shape) 'triangle)
(setf (get 'o10 'size)  'small)
(setf (get 'o10 'texture)  'blank)

(setq ej1 '((on o1 table) (on o2 o1) (on o3 o2)))

(setq ej2 '((on o4 table) (on o5 table) (on o6 table) (inside o4 o6)
    (inside o5 o6) (on o7 o6)))

(setq ej3 '((on o8 table) (on o9 o8) (on o10 o9)))

;;; Variables para el tratamiento de las relaciones

(setq ax_subst '((on above)))
(setq ax_transit '(above inside))
```

## A.3 Descripción modificada del ejemplo de la figura 4.1

```
;;; Ejemplo de Dietterich-Michalski en Machine Learning I
;;; Version sin atributos

(setf (get 'o1 'shape) 'box)
(setf (get 'o1 'size)  'large)
(setf (get 'o1 'texture)  'blank)
(setf (get 'o1 'dummy) 'yy)

(setf (get 'o2 'shape) 'circle)
(setf (get 'o2 'size)  'medium)
(setf (get 'o2 'texture)  'shaded)
```

```
(setf (get 'o2 'dummy) 'yy)

(setf (get 'o3 'shape) 'diamond)
(setf (get 'o3 'size)  'medium)
(setf (get 'o3 'texture)  'blank)
(setf (get 'o3 'dummy) 'yy)

(setf (get 'o4 'shape) 'circle)
(setf (get 'o4 'size)  'small)
(setf (get 'o4 'texture)  'shaded)
(setf (get 'o4 'dummy) 'yy)

(setf (get 'o5 'shape) 'circle)
(setf (get 'o5 'size)  'small)
(setf (get 'o5 'texture)  'shaded)
(setf (get 'o5 'dummy) 'yy)

(setf (get 'o6 'shape) 'rectangle)
(setf (get 'o6 'size)  'large)
(setf (get 'o6 'texture)  'blank)
(setf (get 'o6 'dummy) 'yy)

(setf (get 'o7 'shape) 'square)
(setf (get 'o7 'size)  'medium)
(setf (get 'o7 'texture)  'blank)
(setf (get 'o7 'dummy) 'yy)

(setf (get 'o8 'shape) 'ellipse)
(setf (get 'o8 'size)  'large)
(setf (get 'o8 'texture)  'blank)
(setf (get 'o8 'dummy) 'yy)

(setf (get 'o9 'shape) 'rectangle)
(setf (get 'o9 'size)  'medium)
(setf (get 'o9 'texture)  'shaded)
(setf (get 'o9 'dummy) 'yy)

(setf (get 'o10 'shape) 'triangle)
(setf (get 'o10 'size)  'small)
(setf (get 'o10 'texture)  'blank)
(setf (get 'o10 'dummy) 'yy)

(setq ej1 '((on o1 table) (on o2 o1) (on o3 o2) (fu o1) (ci o2) (ro o3)
```

```
            (li o1) (ra o2) (li o3) (gr o1) (me o2) (me o3)))

(setq ej2 '((on o4 table) (on o5 table) (on o6 table) (inside o4 o6)
            (inside o5 o6) (on o7 o6) (ci o4) (ci o5) (re o6) (cu o7)
            (ra o4) (ra o5) (li o6) (li o7) (pe o4) (pe o5)
            (gr o6) (me o7)))

(setq ej3 '((on o8 table) (on o9 o8) (on o10 o9) (el o8) (re o9) (tr o10)
            (li o8) (ra o9) (li o10) (gr o8) (me o9) (pe o10)))

;;; Para tratar los axiomas

 (setq ax_subst '((on above) (fu poli) (ci poli) (ro poli) (re poli)
                  (cu poli) (el poli) (tr poli) (li color) (ra color)
                  (gr tam) (pe tam) (me tam)))

 (setq ax_transit '(above inside))
```

## A.4 Descripción del ejemplo de los trenes

```
;;; Ejemplo de los trenes de Michalski

(setf (get 'o1 'longitud) 'largo)
(setf (get 'o1 'forma)  'rectabierto)
(setf (get 'o1 'fcarga)  'cuadrado)
(setf (get 'o1 'npartes) '3)
(setf (get 'o1 'nruedas) '2)


(setf (get 'o2 'longitud) 'corto)
(setf (get 'o2 'forma)  'polcerrado)
(setf (get 'o2 'fcarga)  'triangulo)
(setf (get 'o2 'npartes) '1)
(setf (get 'o2 'nruedas) '2)


(setf (get 'o3 'longitud) 'largo)
(setf (get 'o3 'forma)  'rectabierto)
(setf (get 'o3 'fcarga)  'hexagono)
```

```
(setf (get 'o3 'npartes) '1)
(setf (get 'o3 'nruedas) '3)


(setf (get 'o4 'longitud) 'corto)
(setf (get 'o4 'forma)  'rectabierto)
(setf (get 'o4 'fcarga)  'circulo)
(setf (get 'o4 'npartes) '1)
(setf (get 'o4 'nruedas) '2)


(setf (get 'o5 'longitud) 'corto)
(setf (get 'o5 'forma)  'polabierto)
(setf (get 'o5 'fcarga)  'triangulo)
(setf (get 'o5 'npartes) '1)
(setf (get 'o5 'nruedas) '2)


(setf (get 'o6 'longitud) 'corto)
(setf (get 'o6 'forma)  'vasoabierto)
(setf (get 'o6 'fcarga)  'rectangulo)
(setf (get 'o6 'npartes) '1)
(setf (get 'o6 'nruedas) '2)


(setf (get 'o7 'longitud) 'corto)
(setf (get 'o7 'forma)  'rectcerrado)
(setf (get 'o7 'fcarga)  'circulo)
(setf (get 'o7 'npartes) '2)
(setf (get 'o7 'nruedas) '2)


(setf (get 'o8 'longitud) 'corto)
(setf (get 'o8 'forma)  'rectabierto)
(setf (get 'o8 'fcarga)  'circulo)
(setf (get 'o8 'npartes) '1)
(setf (get 'o8 'nruedas) '2)


(setf (get 'o9 'longitud) 'corto)
(setf (get 'o9 'forma)  'vasocerrado)
(setf (get 'o9 'fcarga)  'triangulo)
(setf (get 'o9 'npartes) '1)
```

```
(setf (get 'o9 'nruedas) '2)


(setf (get 'o10 'longitud) 'largo)
(setf (get 'o10 'forma)  'rectcerrado)
(setf (get 'o10 'fcarga)  'triangulo)
(setf (get 'o10 'npartes) '1)
(setf (get 'o10 'nruedas) '3)


(setf (get 'o11 'longitud) 'corto)
(setf (get 'o11 'forma)  'vasoabierto)
(setf (get 'o11 'fcarga)  'triangulo)
(setf (get 'o11 'npartes) '1)
(setf (get 'o11 'nruedas) '2)


(setf (get 'o12 'longitud) 'corto)
(setf (get 'o12 'forma)  'doblefondo)
(setf (get 'o12 'fcarga)  'triangulo)
(setf (get 'o12 'npartes) '1)
(setf (get 'o12 'nruedas) '2)


(setf (get 'o13 'longitud) 'corto)
(setf (get 'o13 'forma)  'elipse)
(setf (get 'o13 'fcarga)  'rombo)
(setf (get 'o13 'npartes) '1)
(setf (get 'o13 'nruedas) '2)


(setf (get 'o14 'longitud) 'corto)
(setf (get 'o14 'forma)  'rectabierto)
(setf (get 'o14 'fcarga)  'cuadrado)
(setf (get 'o14 'npartes) '1)
(setf (get 'o14 'nruedas) '2)


(setf (get 'o15 'longitud) 'corto)
(setf (get 'o15 'forma)  'doblefondo)
(setf (get 'o15 'fcarga)  'triangulo)
(setf (get 'o15 'npartes) '1)
(setf (get 'o15 'nruedas) '2)
```

```
(setf (get 'o16 'longitud) 'largo)
(setf (get 'o16 'forma)  'rectcerrado)
(setf (get 'o16 'fcarga)  'rectangulo)
(setf (get 'o16 'npartes) '1)
(setf (get 'o16 'nruedas) '3)


(setf (get 'o17 'longitud) 'corto)
(setf (get 'o17 'forma)  'rectcerrado)
(setf (get 'o17 'fcarga)  'circulo)
(setf (get 'o17 'npartes) '1)
(setf (get 'o17 'nruedas) '2)


(setq ej1 '((detras o1 maquina) (detras o2 o1) (detras o3 o2)
(detras o4 o3)))
(setq ej2 '((detras o5 maquina) (detras o6 o5) (detras o7 o6)))
(setq ej3 '((detras o8 maquina) (detras o9 o8) (detras o10 o9)))
(setq ej4 '((detras o11 maquina) (detras o12 o11) (detras o13 o12)
(detras o14 o13)))
(setq ej5 '((detras o15 maquina) (detras o16 o15) (detras o17 o16)))


 (setq ax_subst '((detras pordet)))
 (setq ax_transit '(pordet))
```

# Apéndice B

# Código del generalizador y del constructor de jerarquías

```
;;; Prototipo version 6 - Octubre 1991

(defvar lisnoms)
(defvar listats)
(defvar objetos)
(defvar variables)
(defvar props-atribs)
(defvar concepto)
(defvar ax_subst)
(defvar ax_transit)
(defvar atributos)
(defvar vars)

;;; Cambio basico de la version 6
;;; - Posibilidad de creacion de una 'jerarquia' de objetos,donde
;;; se pueden definir objetos 'complejos' que contengan objetos
;;; mas 'simples' definidos anteriormente.
;;; Idea : -aplicar el factorizador a los ejemplos del nuevo concepto.
;;;        -intentar buscar apariciones de conceptos ya conocidos
;;;         en la descripcion general del nuevo concepto.
;;;        -introducir el nuevo concepto en la jerarquia de conceptos,
;;;         colocandolo en el lugar adecuado.

;;; Tambien elimino variables globales innecesarias.

(setq lisnoms '(t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16
```

```
t17 t18 t19 t20 t21 t22 t23 t24 t25))
(setq listats '(t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16
t17 t18 t19 t20 t21 t22 t23 t24 t25))

;;; Definicion inicial del grafo

(setf (get 'raiz 'padres) '())
(setf (get 'raiz 'hijos) '())
(setf (get 'raiz 'nobjetos) 0)
(setf (get 'raiz 'descripcion) '())
(setf (get 'raiz 'atributos) '())

;;; Funciones necesarias en la version 6.

(defun busqueda (fc atrs nodo nomatrs)
(print " Fase 1: construccion de posibilidades")
   (let ((atr1 nil)
         (atr2 nil)
         (var1 nil)
         (var2 nil)
         (res nil)
         (l nil)
         (parar nil)
         (llista nil)
         (nl nil)
        )
        (setq atr1 (coge-atrs (cadar (get nodo 'atributos))))
        (setq atr2 nomatrs)
        (if (not (incluidos atr2 atr1))
         (progn
          (print "En el nuevo concepto no puede aparecer el concepto:")
          (print nodo)
          (print " por incompatibilidad de atributos.")
         )
           (progn
                (setq var1 (coge-vars (get nodo 'atributos)))
                (setq var2 (busca-vars fc))
                (setq res nil)
                (dolist (var var1)
                 (if (not parar)
                  (progn
                  (setq l nil)
                  (dolist (v var2)
```

```
            (if (coinciden atr2 var v nodo atrs)
                        (setq l (append l (list v)))
                        ()
            )
          )
          (setq res (append res (list l)))
          (if (eq l nil)
           (progn
            (print "En el nuevo concepto no aparece ningun:")
            (print nodo)
            (setq parar t)
           )
            ()
           )
           )
          )
         )
(if (not parar)
      (progn
        (print "Ahora empieza la segunda fase : combinaciones ")
          (dolist (elem res)
            (setq llista (combi llista elem))
          )
        (print "Ahora empieza la fase 2 bis : eliminar repes")
          (setq llista (elimi llista))
        (print "Ahora empieza la fase 3 : substituciones")
        (setq res (fase3 llista (get nodo 'descripcion) var1))
        (print "Ahora empieza la fase 4 : comprobaciones")
        (dolist (r res)
         (if (todos-estan r (car (extender (list fc))))
          (progn
           (print "El concepto ")
           (print nodo)
           (print " aparece en el nuevo concepto en las vars")
           (print (car llista))
           (setq nl (append nl (list (car llista))))
          )
          ()
         )
         (setq llista (cdr llista))
        )
      )
      ()
```

```
          )
         )
        )
   )
)

;;; Todos-estan : mira si todos los elementos de una lista estan
;;; contenidos en otra lista.

(defun todos-estan (l1 l2)
   (if (null l1)
        t
        (and (pertenece (car l1) l2)
             (todos-estan (cdr l1) l2)
        )
   )
)

;;; Sust : hace una substitucion

(defun sust (pos descr varias)
   (if (null pos)
        descr
        (sust (cdr pos)
              (sustituir descr (car varias) (car pos))
              (cdr varias)
        )
   )
)

;;; Fase3 : hace la fase de sustitucion

(defun fase3 (posibs descr varias)
   (let ((ltp nil))
        (dolist (pos posibs)
          (setq ltp (append ltp
                            (list (sust pos descr varias))
                    )
          )
        )
        ltp
    )
)
```

```
;;; Elimi : elimina listas con repetidos

(defun elimi (llist)
   (if (null llist)
        nil
        (if (eliminable (car llist))
               (elimi (cdr llist))
               (cons (car llist) (elimi (cdr llist)))
        )
   )
)

;;; Eliminable : funcion auxiliar usada por elimi.

(defun eliminable (l)
   (if (null l)
        nil
        (if (pertenece (car l) (cdr l))
               t
               (eliminable (cdr l))
        )
   )
)

;;; Combi : construye combinaciones

(defun combi (l1 l2)
   (if (null l1)
        l2
        (let ((l ()))
         (dolist (el1 l1)
                (dolist (el2 l2)
                   (if (atom el1)
                        (setq l (append l (list (list el1 el2))))
                        (setq l (append l (list (append el1 (list el2)))))
                   )
                )
         )
         l
        )
   )
)
```

```
;;; Coinciden : mira si los valores de los atributos de 2
;;; variables coinciden.

(defun coinciden (latribs v1 v2 nodo atrs)
   (if (null latribs)
        t
        (and (equal (prop (car latribs)
                          (cadr (assoc v1 (get nodo 'atributos)))
                    )
                    (prop (car latribs)
                          (cadr (assoc v2 atrs))
                    )
             )
             (coinciden (cdr latribs) v1 v2 nodo atrs)
        )
   )
)

;;; Prop : devuelve el valor de la propiedad en la lista

(defun prop (atr l)
   (if (null l)
        nil
        (if (equal atr (car l))
                (cadr l)
                (prop atr (cddr l))
        )
   )
)

;;; Coge-atrs : coge los atributos de la propiedad atributos

(defun coge-atrs (lista)
   (if (null lista)
        nil
        (cons (car lista) (coge-atrs (cddr lista)))
   )
)

;;; Busca-vars : da las variables usadas en el fc.
;;; Asume que toda variable aparece alguna vez como primer argumento en
;;; alguna clausula.
```

```
(defun busca-vars (fc)
   (if (null fc)
        nil
            (unio (list (cadar fc))
                  (busca-vars (cdr fc))
            )
    )
)

;;; Coge-vars : da las variables de la lista de atributos.

(defun coge-vars (atrs)
   (if (null atrs)
        nil
        (cons (caar atrs)
              (coge-vars (cdr atrs))
        )
    )
)

;;; Desc : da los nodos descendientes del nodo dado.

(defun desc (nodo)
   (let ((l ()))
    (setq l (get nodo 'hijos))
    (if (null l)
        nil
        (unio l (union-desc l))
    )
   )
)

;;; Union-desc : funcion auxiliar de desc.

(defun union-desc (lli)
   (if (null lli)
        nil
        (unio (get (car lli) 'hijos)
              (union-desc (cdr lli))
        )
   )
)
```

```
;;; Funcion auxiliar : mira si todos los elementos del cto.
;;; estan en la lista.

(defun incluidos (cto lista)
   (if (null cto)
       t
       (and (member (car cto) lista)
            (incluidos (cdr cto) lista)
       )
   )
)

;;; Colocar : coloca el concepto en la jerarquia,con los padres dados.

(defun colocar (concepto fc atrs padres)
    (setf (get concepto 'padres) padres)
    (setf (get concepto 'hijos) '())
    (setf (get concepto 'nobjetos) (length (busca-vars fc)))
    (setf (get concepto 'descripcion) fc)
    (setf (get concepto 'atributos) atrs)
    (dolist (elem padres)
      (setf (get elem 'hijos) (append (get elem 'hijos) (list concepto)))
    )
)

;;; Los atributos de los objetos se ponen en la lista de propiedades.
;;; En la primera version solo tratabamos una propiedad (color),pero
;;; en la version 2 ya se generalizo a un numero variable de atributos.
;;; Concretamente, en el ejemplo de esa version tratabamos los atributos
;;; color,forma y peso.

;;; Los ejemplos estan en ficheros aparte.

;;; Las relaciones binarias se ponen explicitamente en los ejemplos.
;;; Hasta la version 3 solo tratabamos las relaciones ON y TOP.En la
;;; version 4 ya tratamos las relaciones ON,LEFT,LEFT* y TOP, que son las
;;; que parecen suficientes para definir cualquier escena en 2D (incluso
;;; sobraria TOP).

;;; ****************************************************************
;;; ****************************************************************
```

```
;;; Organizacion en fases desde la version 5.
;;; 1- Substitucion de objetos por variables en los ejemplos positivos
;;;    para obtener la lista de posibilidades de matching.
;;; 2- Valoracion y seleccion de las posibilidades de matching mas
;;;    prometedoras segun un cierto criterio.
;;; 3- Extension de las posibilidades prometedoras con ayuda de los
;;;    axiomas del dominio.
;;; 4- Extraccion del factor comun de estas posibilidades expandidas.
;;; 5- Reduccion de los factores comunes obtenidos (deshacer la fase 3)
;;; 6- Seleccionar el mejor factor comun (el mas largo).

;;; ************************************************************************
;;; ************************************************************************

;;; Primera fase : Paso del conjunto de ejemplos al conjunto de
;;; posibilidades de matching (sustituyendo objetos por variables
;;; segun el valor de los atributos que consideramos;asi quedan
;;; asociados en cada posibilidad los objetos que hayan sido
;;; substituidos por la misma variable).

;;; Desarrollar : 'expande' los ejemplos a partir de su nombre.

(defun desarrollar (lista_nombres)
   (if (null lista_nombres)
       nil
       (append (list (eval (car lista_nombres)))
               (desarrollar (cdr lista_nombres))
       )
   )
)

;;; Unio : hace la union de los elementos de 2 listas quitando repetidos.
;;; En lista1 no hay repetidos.Es diferente de la funcion union
;;; (por ejemplo, (unio '(a b c) '(a a b))  -> (A B C)  y
;;;                  (union '(a b c) '(a a b)) -> (B A A C)    ).

(defun unio (lista1 lista2)
   (if (null lista2) lista1
       (if (member (car lista2) lista1)
           (unio lista1 (cdr lista2))
            (unio (append lista1 (list (car lista2)))
                  (cdr lista2)
           )
```

```
        )
    )
)

;;; Objetop : mira si un determinado atomo es un objeto (oXX) o no.

(defun objetop (atomo)
   (if (member atomo objetos)
       t
       nil
    )
)

;;; Obj_claus : devuelve los objetos presentes en una clausula.

(defun obj_claus (clausula)
  (let ( (primer ())
         (segon ())
         (res ())
       )
   (setq primer (cadr clausula))
   (setq segon (caddr clausula))
   (if (objetop primer) (setq res (list primer)))
   (if (objetop segon) (setq res (append res (list segon))))
   res
  )
)

;;; Dar_obj : devuelve los objetos presentes en un ejemplo.

(defun dar_obj (ejemplo)
   (if (null ejemplo)
       nil
       (unio  (obj_claus (car ejemplo))
              (dar_obj (cdr ejemplo))
       )
    )
)

;;; Sustituir_clausula : sustituye las apariciones de un objeto por una
;;; variable en una clausula determinada.

(defun sustituir_clausula (lista objeto variable)
```

```
   (if (null lista)
     nil
     (if (eq (car lista) objeto)
       (cons variable (sustituir_clausula (cdr lista) objeto variable))
       (cons (car lista) (sustituir_clausula (cdr lista) objeto variable))
     )
   )
)

;;; Sustituir : sustituye las apariciones de un objeto por una variable en
;;; un ejemplo determinado.

(defun sustituir (ejemplo objeto variable)
   (if (null ejemplo)
     nil
     (append (list (sustituir_clausula (car ejemplo) objeto variable))
             (sustituir (cdr ejemplo) objeto variable)
     )
   )
)

;;; Sumatorio : funcion que anyade a s3 la combinacion de todos
;;; los elementos de s1 con el de s2.

(defun sumatorio (s1 s2 s3)
   (if (null s1)
        s3
       (let ( (temp ())
            )
          (setq temp (car s1))
           (setq temp (append temp (list s2)))
          (setq s3 (append s3 (list temp)))
          (sumatorio (cdr s1) s2 s3)
       )
   )
)

;;; Dar_valores : construye la lista con los valores de los atributos de
;;; la lista de atributos del objeto en cuestion.

(defun dar_valores (objeto lista_atribs)
   (if (null lista_atribs)
        nil
```

```
        (cons (get objeto (car lista_atribs))
              (dar_valores objeto (cdr lista_atribs))
        )
   )
)

;;; Cumple_obj : mira si el objeto tiene los valores pedidos
;;; en los atributos.

(defun cumple_obj (objeto l_atributos l_valores)
   (if (null l_atributos)
        t
        (and (eq (get objeto (car l_atributos)) (car l_valores))
             (cumple_obj objeto (cdr l_atributos) (cdr l_valores))
        )
   )
)

;;; Filtro_objetos : filtra los objetos de la lista que tengan los mismos
;;; valores en los atributos que consideramos.

(defun filtro_objetos (l_obs l_atrs l_vals)
   (if (null l_obs)
        nil
        (let ( (obt ())
             )
          (setq obt (car l_obs))
          (if (cumple_obj obt l_atrs l_vals)
               (cons obt (filtro_objetos (cdr l_obs) l_atrs l_vals))
               (filtro_objetos (cdr l_obs) l_atrs l_vals)
          )
        )
   )
)

;;; Nueva_fase : funcion que aplica una fase.Consiste en cambiar un objeto
;;; por una variable en el primer ejemplo
;;; y construir todas las combinaciones
;;; posibles con el resto de objetos que tengan los mismos valores en los
;;; atributos en el resto de ejemplos.

(defun nueva_fase (l_l_ej l_atrs)
  (let ( (nueva_lista ())
```

```
          (var ())
          (prim_ej ())
          (l_obj ())
          (obj ())
          (valores ())
          (subs1 ())
          (subl ())
          (posible ())
          (subs2 ())
          (flag ())
        )
    (setq var (car variables))
    (setq variables (cdr variables))
    (setq flag nil)
    (dolist (lista_ejemplos l_l_ej)
         (setq prim_ej (car lista_ejemplos))
         (setq l_obj (dar_obj prim_ej))
         (setq obj (car l_obj))
         (setq valores (dar_valores obj l_atrs))
         (setq subs1 (list (list (sustituir prim_ej obj var))))
         (dolist (ej_rest_ej (cdr lista_ejemplos))
             (setq subl nil)
             (setq l_obj (dar_obj ej_rest_ej))
             (setq posible (filtro_objetos l_obj l_atrs valores))
             (if posible
                 (progn
                    (dolist (objeto posible)
(if (not flag)
    (progn
         (setq props-atribs (append props-atribs
                                    (list (list var (symbol-plist obj)))
                            )
         )
         (setq flag t)
    )
    ()
)

                       (setq subs2 (sustituir ej_rest_ej objeto var))
                       (setq subl (sumatorio subs1 subs2 subl))
                    )
                )
                (setq subl (sumatorio subs1 ej_rest_ej subl))
```

```
            )
            (setq subs1 subl)
        )
        (setq nueva_lista (append nueva_lista subs1))
    )
    nueva_lista
  )
)

;;; Aplicar_fases : funcion que va aplicando fases mientras queden objetos
;;; que se hayan de pasar a variables en el primer ejemplo.

(defun aplicar_fases (lis_lis l_atrib)
  (let ( (ej_aux ())
         (obj_aux ())
       )
    (setq ej_aux (caar lis_lis))
    (setq obj_aux (dar_obj ej_aux))
    (if  (null obj_aux)
         lis_lis
         (aplicar_fases (nueva_fase lis_lis l_atrib) l_atrib)
    )
  )
)

;;; ***************************************************************
;;; ***************************************************************

;;; Segunda fase : valoracion y seleccion de las posibilidades mas
;;; prometedoras.

;;; Ya_esta : mira si elem es el car de alguno de los pares de l_parells.

(defun ya_esta (elem l_parells)
   (if (null l_parells)
        nil
        (or (equal (caar l_parells) elem)
            (ya_esta elem (cdr l_parells))
        )
   )
)

;;; Increm : incrementa el cdr del par cuyo car sea claus.
```

```
(defun increm (claus lista)
   (if (equal (caar lista) claus)
        (append (list (cons (caar lista) (+ 1 (cdar lista))))
                (cdr lista)
        )
        (append (list (car lista))
                (increm claus (cdr lista))
        )
   )
)

;;; Mirar_frecuencias : crea una lista donde devuelve las relaciones que
;;; existen en cada posibilidad y su frecuencia de aparicion.

(defun mirar_frecuencias (posib)
  (let ( (llr ())
       )
   (dolist (ejp posib)
     (dolist (clsla ejp)
       (if (ya_esta clsla llr)
          (setq llr (increm clsla llr))
          (setq llr (append llr (list (cons clsla 1))))
       )
     )
   )
   llr
 )
)

;;; Mirar_repeticiones : mira cuantas clausulas existen con frecuencia
;;; igual al numero de ejemplos.

(defun mirar_repeticiones (fre)
   (if (null fre)
        0
        (if (= (cdar fre) (length concepto))
                (+ 1 (mirar_repeticiones (cdr fre)))
                (mirar_repeticiones (cdr fre))
        )
   )
)
```

```
;;; Estimacion : da la estimacion de cuan buena es una substitucion de
;;; objetos por variables segun la frecuencia de aparicion de clausulas.

(defun estimacion (li_frec)
   (if (null li_frec)
        0
        (+ (* (cdar li_frec) (cdar li_frec))
           (estimacion (cdr li_frec))
        )
   )
)

;;; Funcion principal de esta segunda fase.

(defun seleccionar (lispos)
  (let ( (posi_mej ())
         (valo_maxi 0)
         (repet_max 0)
         (num_repet 0)
         (frec 0)
         (valoracion 0)
       )
       (dolist (possi lispos)
          (setq frec (mirar_frecuencias possi))
          (setq num_repet (mirar_repeticiones frec))
          (if (> num_repet repet_max)
               (progn
                   (setq repet_max num_repet)
                  (setq valoracion (estimacion frec))
                  (setq valo_maxi valoracion)
                  (setq posi_mej (list possi))
               )
               (if (= num_repet repet_max)
                  (progn
                     (setq valoracion (estimacion frec))
                       (if (= valoracion valo_maxi)
                       (setq posi_mej (append posi_mej (list possi)))
                        (if (> valoracion valo_maxi)
                         (progn
                             (setq valo_maxi valoracion)
                             (setq posi_mej (list possi))
                         )
                        )
```

```
                    )
                     )
                  )
            )
          )
posi_mej
  )
)

;;; ************************************************************************
;;; ************************************************************************

;;; Fase 3: Extension de las posibilidades de acuerdo con los axiomas
;;; del dominio que estemos tratando.Solo se consideran axiomas de
;;; tipo reflexivo ( ej: (on x y) -> (sobre x y) )o transitivo
;;; ( ej: (sobre x y) and (sobre y z) -> (sobre x z) ).

;;; In_una_ref_ej : incluye el efecto de un axioma reflexivo
;;; sobre un ejemplo.

(defun in_una_ref_ej (ejem axi)
  (let ( (pred_ini ())
         (pred_fin ())
         (nova_llista ())
       )
   (setq pred_ini (car axi))
   (setq pred_fin (cadr axi))
   (setq nova_llista ejem)
   (dolist (cla ejem)
      (if (equal (car cla) pred_ini)
                (setq nova_llista (append nova_llista
                                          (list (cons pred_fin (cdr cla)))
                                  )
                )
      )
   )
   nova_llista
  )
)

;;; Existe_pred:mira si existe un predicado de cierta forma.

(defun existe_pred (ejem axi op2)
```

```
  (let ( (resul ())
       )
   (if (null ejem)
        (setq resul nil)
        (if (and (equal (caar ejem) axi)
                 (equal (caddar ejem) op2)
            )
            (setq resul (cadar ejem))
            (setq resul (existe_pred (cdr ejem) axi op2))
        )
   )
   resul
  )
)

;;; Pertenece : mira si un objeto pertenece a una lista.

(defun pertenece (obj lis)
   (if (null lis)
        nil
        (or (equal obj (car lis))
            (pertenece obj (cdr lis))
        )
   )
)

;;; Repetir_fases : funcion con el bucle necesario para la aplicacion de
;;; los axiomas transitivos.

(defun repetir_fases (ejem ult_con axi)
  (let ( (c_incl ())
         (nova_llista ())
         (op1 ())
         (nova_cla ())
       )
   (if (null ult_con)
       ejem
       (progn
          (setq c_incl nil)
          (setq nova_llista ejem)
          (dolist (clau ult_con)
               (if (equal (car clau) axi)
                  (progn
```

```
                    (setq op1 (existe_pred ejem axi (cadr clau)))
                    (if (not (null op1))
                       (progn
                        (setq nova_cla (list axi op1 (caddr clau)))
                        (if (and (not (pertenece nova_cla ejem))
                                 (not (pertenece nova_cla c_incl))
                            )
                         (setq c_incl (append c_incl (list nova_cla)))
                        )
                       )
                     )
                  )
               )
            )
                  (setq nova_llista (append nova_llista c_incl))
                  (repetir_fases nova_llista c_incl axi)
            )
     )
     )
)

;;; In_una_tra_ej : aplica un axioma transitivos a un ejemplo.

(defun in_una_tra_ej (ejem axi)
   (repetir_fases ejem ejem axi)
)

;;; Incl_ref_ej : incluye el efecto de todos los axiomas reflexivos sobre
;;; un ejemplo.

(defun incl_ref_ej (eje ll_ax_su)
   (if (null ll_ax_su)
        eje
        (incl_ref_ej (in_una_ref_ej eje (car ll_ax_su))
                     (cdr ll_ax_su)
        )
   )
)

;;; Incl_tra_ej : aplica los axiomas transitivos a un ejemplo.

(defun incl_tra_ej (eje ll_ax_tr)
   (if (null ll_ax_tr)
```

```
        eje
        (incl_tra_ej (in_una_tra_ej eje (car ll_ax_tr))
                     (cdr ll_ax_tr)
        )
    )
)

;;; Incl_refl : incluye el efecto de todos los axiomas reflexivos sobre
;;; una posibilidad.

(defun incl_refl (posiprom axi_sub)
   (if (null posiprom)
       nil
       (cons (incl_ref_ej (car posiprom) axi_sub)
             (incl_refl (cdr posiprom) axi_sub)
       )
   )
)

;;; Incl_tran : incluye el efecto de todos los axiomas transitivos sobre
;;; las posibilidades prometedoras.

(defun incl_tran (posiprom axi_tra)
   (if (null posiprom)
       nil
       (cons (incl_tra_ej (car posiprom) axi_tra)
             (incl_tran (cdr posiprom) axi_tra)
       )
   )
)

;;; Extender : aplica todos los axiomas a una posibilidad prometedora.

(defun extender (posprom)
   (setq posprom (incl_refl posprom ax_subst))
   (setq posprom (incl_tran posprom ax_transit))
   posprom
)

;;; Extension : aplica todos los axiomas a todas las posibilidades
;;; prometedoras.

;;; Nota : los axiomas deben de ser de tal forma que aplicando primero los
```

```
;;; reflexivos (1 vez) y luego los transitivos (n veces) obtengamos la
;;; extension de las clausulas.( Implica esto una gran restriccion??? ).

(defun extension (lposprom)
   ( if (null lposprom)
        nil
        (append (list (extender (car lposprom)))
                (extension (cdr lposprom))
        )
   )
)

;;; ***************************************************************************
;;; ***************************************************************************

;;; Fase 4 : matching de las posibilidades prometedoras extendidas.Como
;;; ya estan extendidas es un paso trivial,solo se han de ver que
;;; claausulas aparecen en todos los ejemplos.

;;; Filtrar : se queda con las clausulas que aparecen en todos los
;;; ejemplos.

(defun filtrar (lista num)
   (if (null lista)
        nil
        (if (equal num (cdar lista))
                (cons (caar lista) (filtrar (cdr lista) num))
                (filtrar (cdr lista) num)
        )
   )
)

;;; Matching : mira las frecuencias de aparicion de las clausulas y
;;; se queda con las que aparezcan en todos los ejemplos.

(defun matching (lposib)
  (let ( (nuevo_res ())
         (fr 0)
        )
    (dolist (pos lposib)
        (setq fr (mirar_frecuencias pos))
        (setq fr (filtrar fr (length concepto)))
        (setq nuevo_res (append nuevo_res (list fr)))
```

```
   )
   nuevo_res
  )
)

;;; ************************************************************************
;;; ************************************************************************

;;; Fase 5 : reduccion de los factores comunes obtenidos,para deshacer
;;; el efecto de la extension de la fase 3.

;;; Buscar_pos : busca unas posibilidades para quitar ax. trans.

(defun buscar_pos (tfc cap f2)
 (let ((auxi nil)
       )
   (if (null tfc)
        nil
        (progn
          (setq auxi (existe_pred (list (car tfc)) cap f2))
          (if (null auxi)
                (buscar_pos (cdr tfc) cap f2)
                (cons auxi (buscar_pos (cdr tfc) cap f2))
          )
        )
   )
  )
)

 ;;; Comprobar : comprueba transitividades

 (defun comprobar (lista cap f1 tfc)
    (if (null lista)
         nil
         (if (pertenece (list cap f1 (car lista)) tfc)
                t
                (comprobar (cdr lista) cap f1 tfc)
         )
    )
 )

 ;;; Elim_tran : elimina la aplicacion de los axiomas transitivos.
```

```
(defun elim_tran (ufc tfc)
  (let ( (cl ())
         (cap ())
         (f1 ())
         (f2 ())
         (xx ())
       )
    (if (null ufc)
        nil
        (progn
          (setq cl (car ufc))
          (setq cap (car cl))
          (if (member cap ax_transit)
                (progn
                  (setq f1 (cadr cl))
                  (setq f2 (caddr cl))
                  (setq xx (buscar_pos tfc cap f2))
                  (if (null xx)
                    (cons cl (elim_tran (cdr ufc) tfc))
                    (if (comprobar xx cap f1 tfc)
                        (elim_tran (cdr ufc) tfc)
                        (cons cl (elim_tran (cdr ufc) tfc))
                    )
                  )
                )
                (cons cl (elim_tran (cdr ufc) tfc))
          )
        )
    )
  )
)

;;; Busca_par : mira si nom corresponde a alguno de los segundos elementos
;;; de una lista de pares.

(defun busca_par (nom pares)
   (if (null pares)
       nil
       (if (equal nom (cadar pares))
               (caar pares)
               (busca_par nom (cdr pares))
       )
    )
```

```
)

;;; Mira si la clausula viene de una axioma reflexivo

(defun clau_refl (clausula axiomes)
   (busca_par (car clausula) axiomes)
)

;;; Elim_refl : elimina el efecto de los axiomas reflexivos
;;; en un factor comun.

(defun elim_refl (ufc tfc)
  (let ( (cl ())
         (cl2 ())
       )
   (if (null ufc)
       nil
       (progn
         (setq cl (car ufc))
          (setq cl2 (clau_refl cl ax_subst))
         (if (null cl2)
               (cons cl (elim_refl (cdr ufc) tfc))
               (if (pertenece (cons cl2 (cdr cl)) tfc)
                       (elim_refl (cdr ufc) tfc)
                       (cons cl (elim_refl (cdr ufc) tfc))
               )
         )
       )
   )
  )
)

;;; Compresion : comprime los factores comunes expandidos.Es como una
;;; aplicacion inversa de los axiomas.

(defun compresion (lfcs)
  (let ( (fc ())
         (nfc ())
       )
   (if (null lfcs)
       nil
       (progn
          (setq fc (car lfcs))
```

```
            (setq nfc (elim_tran fc fc))
             (setq nfc (elim_refl nfc nfc))
            (append (list nfc) (compresion (cdr lfcs)))
          )
     )
    )
)

;;; ***************************************************************
;;; ***************************************************************

;;; Fase 6 : busqueda del 'mejor' factor comun (nos quedamos con el
;;; mas largo).

;;; Mejor_sol : busca la 'mejor' solucion de las halladas (la mas larga).

(defun mejor_sol (soluciones)
  (let ((mej ())
        (lmax 0)
        (long 0)
       )
   (dolist (soluc soluciones)
      (setq long (length soluc))
      (if (> long lmax)
            (progn
              (setq mej soluc)
              (setq lmax long)
            )
            nil
      )
   )
   mej
  )
)

;;; ***************************************************************
;;; ***************************************************************

;;; Funciones auxiliares diversas.

;;; Existe : mira si var aparece en el ejemplo.

(defun existe (var ejemplo)
```

```
   (if (null ejemplo)
        nil
        (or (equal var (cadr (car ejemplo)))
            (equal var (caddr (car ejemplo)))
            (existe var (cdr ejemplo))
        )
   )
)

;;; Quitar : quita elem de l (supone que solo aparece 1 vez).

(defun quitar (l elem)
   (if (null l)
        nil
        (if (equal (car l) elem)
                (cdr l)
                (cons (car l) (quitar (cdr l) elem))
        )
   )
)

;;; Resta : hace l1 -l2.

(defun resta (l1 l2)
   (if (null l2)
        l1
        (if (member (car l2) l1)
                (resta (quitar l1 (car l2)) (cdr l2))
                (resta l1 (cdr l2))
        )
   )
)

;;; Restaesp : resta con funcion pertenece.

(defun restaesp (l1 l2)
   (if (null l1)
        nil
        (if (pertenece (car l1) l2)
                (restaesp (cdr l1) l2)
                (cons (car l1) (restaesp (cdr l1) l2))
        )
   )
```

```
)

;;; ************************************************

;;; Simplep : para ver si una descripcion es simple o compleja

(defun simplep (descri)
   (if (null descri)
        t
        (if (equal (caar descri) 'ES)
               nil
               (simplep (cdr descri))
        )
   )
)

;;; Existe_solapamiento : mira si hay repeticiones en la lista dada

(defun existe_solapamiento (lls)
        (if (null lls)
                nil
                (if (algun_repe (car lls) (cdr lls))
                        t
                        (existe_solapamiento (cdr lls))
                )
        )
)

;;; Algun_repe : mira si algun elem. de l aparece en lls

(defun algun_repe (l lls)
        (if (null l)
                nil
                (if (aparece (car l) lls)
                    t
                    (algun_repe (cdr l) lls)
                )
        )
)

;;; Aparece : mira si un elemento esta en una lista de listas

(defun aparece (el lls)
```

```
   (if (null lls)
        nil
        (if (member el (car lls))
            t
            (aparece el (cdr lls))
        )
   )
)

;;; Poner-nombres : busca nombres (ts) para objetos

(defun poner-nombres (ctoss noms)
   (if (null ctoss)
        nil
        (progn
         (setq lisnoms (cdr lisnoms))
         (cons (list (car noms) (car ctoss))
               (poner-nombres (cdr ctoss) (cdr noms))
         )
        )
   )
)

;;; Incluir : incluye un nuevo objeto complejo en la descripcion

(defun incluir (llax de nc)
  (dolist (el llax)
        (setq de (cons (list 'ES (car el) nc) de))
  )
  de
)

;;; Buscar-t : mira si aparece una variable en la lista

(defun buscar-t (var lis)
   (if (null lis)
        nil
        (if (member var (cadar lis))
                (caar lis)
                (buscar-t var (cdr lis))
        )
   )
)
```

```
;;; Susti: substituye los predicados antiguos por el nuevo objeto

(defun susti (csubs nomconc desc)
  (let ((daux ())
        (laux ())
        (ti ())
        (tj ())
       )
   (setq laux (poner-nombres csubs lisnoms))
   (setq daux nil)
   (dolist (el desc)
     (if (not (null (cddr el)))
        (progn
           (setq ti (buscar-t (cadr el) laux))
           (setq tj (buscar-t (caddr el) laux))
           (if (null ti)
                (if (null tj)
                        (setq daux (append daux (list el)))
                        (setq daux (append daux
                                (list (list (car el) (cadr el) tj))))
                )
                (if (null tj)
                   (setq daux (append daux
                              (list (list (car el) ti (caddr el)))))
                   (if (and (not (eq ti tj))
                            (not (pertenece (list (car el) ti tj) daux))
                       )
                       (setq daux (append daux
                           (list (list (car el) ti tj))))
                   )
                )
            )
        )
        (progn
          (setq ti (buscar-t (cadr el) laux))
          (if (null ti)
                (setq daux (append daux (list el)))
                (setq daux (append daux
                                (list (list (car el) ti))))
          )
        )
    )
```

```
   )
   (setq daux (incluir laux daux nomconc))
   daux
 )
)

;;; Quitar-objs: deja predicados con variables

(defun quitar-objs (des atrs)
   (let ((l nil)
         (aux nil)
        )
        (setq aux (busca-vars des))
        (dolist (el atrs)
           (if (member (car el) aux)
                (setq l (append l (list el)))
           )
        )
        l
   )
)

;;; Elimina-ts: quita predicados con ts

(defun elimina-ts (descr)
   (if (null descr)
        nil
        (if (existen-ts (car descr))
                (elimina-ts (cdr descr))
                (cons (car descr) (elimina-ts (cdr descr)))
        )
   )
)

;;; Existen-ts: mira si hay objetos complejos en las clausulas

(defun existen-ts (claus)
   (if (null claus)
        nil
        (if (member (car claus) listats)
                t
                (existen-ts (cdr claus))
        )
```

```
   )
)

;;; Anyadir: crea ED necesaria para proceso posterior

(defun anyadir (l ti n)
   (if (null l)
        (list (list n (list ti)))
        (if (eq (caar l) n)
                (cons (list n (append (cadar l) (list ti))) (cdr l))
                (cons (car l) (anyadir (cdr l) ti n))
        )
   )
)

;;; Existe_posibilidad: mira asociaciones para ver si puede haber
;;; algun objeto complejo

(defun existe_posibilidad (l1 l2)
   (if (null l1)
        t
        (if (> (length (cadar l1)) (length (cadar l2)))
                nil
                (existe_posibilidad (cdr l1) (cdr l2))
        )
   )
)

;;; Aplana : obtiene lista de listas con los elems de l

(defun aplana (l)
   (if (null l)
        nil
        (cons (list (car l)) (aplana (cdr l)))
   )
)

;;; Perm: busca permutaciones

(defun perm (l n)
   (if (= n 1)
        (aplana l)
        (aumenta l (perm l (- n 1)))
```

```
   )
)

;;; Aumenta : funcion auxiliar de perm

(defun aumenta (l lls)
  (let  ((sol ())
         )
   (dolist (l1 lls)
        (dolist (el2 l)
           (if (not (member el2 l1))
               (setq sol (append (list (append l1 (list el2))) sol))
           )
        )
   )
   sol
  )
)

;;; Tmu : quita ultimo elem

(defun tmu (l)
   (if (null (cdr l))
        nil
        (cons (car l) (tmu (cdr l)))
   )
)

;;; Ult : obtiene ultimo elem

(defun ult (l)
   (if (null (cdr l))
        (car l)
        (ult (cdr l))
   )
)

;;; Supcomb: genera combinaciones de la subfase 4

(defun supcomb (l lls)
   (if (null (cdr lls))
        (let ((res ())
              (nom ())
```

```
            (lli ())
           )
           (setq nom (caar lls))
           (setq lli (cadar lls))
           (if (null l)
              (dolist (el lli)
                (setq res (append res (list (list (list nom el)))))
              )
              (dolist (el1 l)
                 (dolist (el2 lli)
                      (setq res (append res (list
                              (append el1 (list (list nom el2))))))
                 )
              )
            )
        res
        )
        (supcomb (supcomb l (tmu lls)) (list (ult lls)))
  )
)

;;; Susel: busca pareja de el en l2

(defun susel (el l1 l2)
   (if (eq el (car l1))
        (car l2)
        (susel el (cdr l1) (cdr l2))
   )
)

;;; Susparel: auxiliar de susparcl

(defun susparel (el l_a_conc elem)
   (if (null l_a_conc)
        el
        (if (member el (cadar l_a_conc))
            (susel el (cadar l_a_conc) (cadar elem))
            (susparel el (cdr l_a_conc) (cdr elem))
        )
   )
)

;;; Susparcl: auxiliar de suspar
```

```
(defun susparcl (cla l_a_conc elem)
   (let ((ax1 ())
         )
    (dolist (el cla)
        (setq ax1 (append ax1 (list (susparel el l_a_conc elem))))
    )
    ax1
   )
)

;;; Suspar: función principal de la subfase 5

(defun suspar (conc l_a_conc elem)
   (let ((ax ())
         )
    (dolist (cla conc)
        (setq ax (append ax (list (susparcl cla l_a_conc elem))))
    )
    ax
   )
)

;;; Susvar : como suspar pero con variables

(defun susvar (conc vs el)
   (let ((ax ())
         )
     (dolist (cla conc)
        (setq ax (append ax (list (susvarcl cla vs el))))
     )
     ax
    )
)

;;; Susvarcl : auxiliar de susvar

(defun susvarcl (cla vs el)
   (let ((ax ())
         )
     (dolist (elem cla)
        (setq ax (append ax (list (svar elem vs el))))
     )
```

```
     ax
    )
)

;;; Svar : auciliar de susvarcl

(defun svar (elem vs el)
   (if (null vs)
        elem
        (if (eq elem (car vs))
          (car el)
          (svar elem (cdr vs) (cdr el))
        )
   )
)

;;; Unio-llistes : une listas con formato especial

(defun unio-llistes (l)
   (if (null l)
        nil
        (unio (cadar l) (unio-llistes (cdr l)))
   )
)

;;; Busca-padres: mira quienes son los padres del nuevo concepto,
;;; examinando los predicados ES.

(defun busca-padres (desc)
   (if (null desc)
        nil
        (if (eq (caar desc) 'es)
                (unio (list (caddar desc)) (busca-padres (cdr desc)))
                (busca-padres (cdr desc))
        )
    )
)

;;; Esta-mesa-p : mira si esta la mesa en la descripcion
;;; Deberia generalizarse a punto-fijo para cualquier dominio

(defun esta-mesa-p (desc)
   (if (null desc)
```

```
        nil
        (if (eq (caddar desc) 'mesa)
                t
                (esta-mesa-p (cdr desc))
        )
   )
)

;;; Quita-repes: quita los repetidos de una lista

(defun quita-repes (l)
   (if (null l)
        nil
        (if (pertenece (car l) (cdr l))
                (quita-repes (cdr l))
                (cons (car l) (quita-repes (cdr l)))
        )
   )
)

;;; Alguno-esta: mira si algun elem de l1 esta en l2

(defun alguno-esta (l1 l2)
   (if (null l1)
        nil
        (if (pertenece (car l1) l2)
           t
           (alguno-esta (cdr l1) l2)
        )
   )
)

;;; Combi-posible : hay combinacion posible si no hay apariciones
;;; de elems de l1 en l2

(defun combi-posible (l1 l2)
   (if (null l1)
        t
        (if (alguno-esta (car l1) l2)
         nil
         (combi-posible (cdr l1) l2)
        )
   )
```

```
)

;;; ************************************************************
;;; ************************************************************

;;; Match : funcion principal.Primero buscamos todas las
;;; sustituciones posibles de objetos por variables teniendo en
;;; cuenta todas las propiedades,luego (si hace falta) con una
;;; propiedad menos,etcetera.En la version 3 se incorporo la funcion
;;; de valoracion segun la frecuencia de aparicion de las relaciones
;;; (sugerencia de Ulises) para buscar el factor comun solo en los
;;; casos donde esta valoracion sea maxima y el numero de clausulas con
;;; aparicion en todos los ejemplos sea tambien maxima (sugerencia
;;; de Gustavo).
;;; En la version 5 se incorpora el tratamiento uniforme y explicito
;;; de los axiomas del dominio.

(defun match ()
  (let ( (l_l_ej_inic ())
         (fin ())
         (atrs ())
         (result ())
         (res ())
         (posi_mej ())
         (aux2 ())
         (descripcion_actual ())
         (lista-todos-nodos ())
         (padres ())
         (resaux ())
        )
   (print "Escribe el nombre del concepto o bien FIN para acabar")
   (setq fin (read))
   (if (not (equal fin 'fin))
        (progn
         (load fin)
         (setq l_l_ej_inic (list (desarrollar concepto)))
         (setq atrs atributos)
        )
        ()
   )
   (do () ((or (equal fin 'fin) (null atrs)) result)
        (print "Consideramos los siguientes atributos : ")
        (print atrs)
```

```
        (setq props-atribs nil)
        (setq lisnoms listats)
        (setq res (aplicar_fases l_l_ej_inic atrs))
        (setq posi_mej (seleccionar res))
        (setq res (extension posi_mej))
        (setq res (matching res))
        (setq res (compresion res))
        (setq res (mejor_sol res))
        (print "La mejor descripcion encontrada es : ")
        (print res)
        (print "Valores de los atributos de las variables :")
        (print props-atribs)
        (setq aux2 atrs)
        (setq variables vars)
        (print "concepto : ")
        (print fin)
        (print "FC : ")
        (print res)
        (print "atributos : ")
        (print props-atribs)
        (print "numero de objetos : ")
        (print (length (busca-vars res)))
        (print "Colocar esta descripcion en el grafo (t/nil) ?")
        (if (read)
         (progn

;;;;; Aqui comienza el recorrido autentico para colocar el
;;;;; concepto en el grafo de forma adecuada.

          (setq descripcion_actual res)
          (print " La descripcion del concepto es")
          (print descripcion_actual)
          (setq lista-todos-nodos (desc 'raiz))
          (do () ((null lista-todos-nodos))
           (if (simplep descripcion_actual)
               (if (simplep (get (car lista-todos-nodos) 'descripcion))
                       (progn
                         (print "Caso 1 : conc. y desc. simples")
                         (setq resaux
                             (busqueda descripcion_actual props-atribs
                             (car lista-todos-nodos) aux2)
                         )
                         (if resaux
```

```
                    (if (existe_solapamiento resaux)
                      (print " Solapamiento !!! ")
                      (progn
                       (setq descripcion_actual
                            (susti resaux
                                  (car lista-todos-nodos)
                                   descripcion_actual)
                       )
                       (setq padres
                            (cons (car lista-todos-nodos)
                                   padres
                            )
                       )
                      )
                    )
                )
              )
              (progn
                (print "Caso 2 : conc. complejo desc. simple")
                (print "NO puede darse este caso !!!!!!!!!!!")
                (setq resaux nil)
              )
)
(if (simplep (get (car lista-todos-nodos) 'descripcion))
          (let ((daux ())
               )
            (print "Caso 3 : conc simple desc compleja")
            (setq daux (elimina-ts descripcion_actual))
            (if (null daux)
                  (setq resaux nil)
                  (progn
                    (setq resaux
                      (busqueda descripcion_actual
                         props-atribs
                         (car lista-todos-nodos) aux2)
                    )
                    (if (not (null resaux))
                      (if (existe_solapamiento resaux)
                          (progn
                             (print "Solapamiento")
                             (setq resaux nil)
                          )
                          (progn
```

```
                                (setq descripcion_actual
                                     (susti resaux
                                       (car lista-todos-nodos)
                                       descripcion_actual
                                     )
                                )
                                (setq padres (cons
                                  (car lista-todos-nodos)
                                  padres)
                                )
                            )
                        )
                    )
                )
            )
        )
        (let ((lista_aux_conc ())
              (lista_aux_desc ())
              (concepto ())
              (vs ())
              (listapos ())
              (lvs ())
              (laux ())
              (flag ())
              (susti ())
             )
          (print "Caso 4 : conc y desc complejas")
          (setq concepto (get (car lista-todos-nodos)
                              'descripcion))
          (print "Subfase1 : creacion listas objetos")
          (dolist
              (claus concepto)
             (if (eq (car claus) 'es)
                (setq lista_aux_conc
                  (anyadir lista_aux_conc (cadr claus)
                           (caddr claus)
                  )
                )
             )
          )
          (dolist (elem lista_aux_conc)
                (setq lista_aux_desc
                  (append lista_aux_desc
```

```
                    (list (list (car elem) '()))
                )
            )
        )
        (dolist (claus descripcion_actual)
            (if (eq (car claus) 'es)
                (setq lista_aux_desc
                    (anyadir lista_aux_desc
                        (cadr claus) (caddr claus))
                )
            )
        )
        (print "Fin  de la subfase1")
        (print lista_aux_conc)
        (print lista_aux_desc)
        (print "Subfase2 : comprobar longitudes")
    (if (existe_posibilidad lista_aux_conc
                            lista_aux_desc)
            (progn
             (print "Puede ser,quizas,quien sabe...")
             (print "Subfase3 : estudio variables")
             (setq vs (resta (busca-vars concepto)
                             listats))
             (dolist (v vs)
               (setq lvs (resta
                    (busca-vars descripcion_actual)
                    listats))
               (setq laux nil)
               (setq flag nil)
               (dolist (var lvs)
                 (if (not flag)
                    (if (coinciden aux2 v var
                            (car lista-todos-nodos)
                            props-atribs)
                        (setq laux (cons var laux))
                    )
                 )
               )
               (setq listapos (append listapos
                            (list laux)))
               (if flag
                (progn
                 (print "Variables niegan posib.")
```

```
                                        (setq resaux nil)
                                       )
                                       (progn
                                        (if (null (cdr listapos))
                                         (setq susti
                                            (aplana (car listapos)))
                                         (dolist (el listapos)
                                            (setq susti (combi susti el))
                                         )
                                        )
                                        (setq susti (elimi susti))
(if (null susti)
  (print " Nada que hacer por las variables")
  (let
        ((co ())
         (ko ())
         (n ())
         (con1 ())
         (con2 ())
         (ll ())
         (deaux ())
        )
    (setq resaux nil)
    (print "Empieza subfase4 : generacion combinaciones")
    (dolist (el1 lista_aux_conc)
        (setq n (length (cadr el1)))
        (dolist (el2 lista_aux_desc)
          (setq ko (car el2))
          (setq co (append co (list (list ko (perm (cadr el2) n)))))
        )
     )
     (setq co (supcomb nil co))
     (print "Empieza la subfase 5 : la definitiva")
     (dolist (elem co)
      (if (combi-posible resaux (unio-llistes elem))
       (progn
        (setq con1 concepto)
        (setq con1 (suspar con1 lista_aux_conc elem))
        (dolist (elem2 susti)
           (setq con2 con1)
           (setq con2 (susvar con2 vs elem2))
           (if (todos-estan con2 descripcion_actual)
             (progn
```

```
(print "BINGO : hemos encontrado algo")
(print "El concepto buscado esta en la descripcion en :")
(print con2)
(print "variables que intervienen")
(print elem2)
(print elem)
(setq ll (unio-llistes elem))
(setq ll (unio ll elem2))
(print ll)
(setq resaux (append resaux ( list ll )))
(setq descripcion_actual
      (restaesp descripcion_actual con2))
(setq deaux descripcion_actual)
(setq descripcion_actual nil)
(dolist (cla deaux)
  (if (pertenece (cadr cla) ll)
       (setq descripcion_actual (append descripcion_actual
        (list (list (car cla) (car lisnoms) (caddr cla)))))
      (if (pertenece (caddr cla) ll)
       (setq descripcion_actual (append descripcion_actual
        (list (list (car cla) (cadr cla) (car lisnoms)))))
       (setq descripcion_actual (append descripcion_actual
                                        (list cla)))
      )
  )
)
(setq descripcion_actual
      (append (list (list 'es
                          (car lisnoms)
                          (car lista-todos-nodos)))
              descripcion_actual
      )
 )
 (if (esta-mesa-p con2) ;; dependiente del dominio...
       (setq descripcion_actual
              (append (list (list 'on (car lisnoms) 'mesa))
              descripcion_actual)
       )
 )
(setq descripcion_actual (quita-repes descripcion_actual))
(setq lisnoms (cdr lisnoms))
(print "La descripcion actual es : ")
(print descripcion_actual)
```

```
                )
            )
        )
      )
      (print "Combinacion eliminada sin tratar,por repeticion vars")
    )
  )
     (if resaux
        (progn
          (print "combinaciones encontradas ....")
          (print resaux)
          (print "La descripcion final es :")
          (print descripcion_actual)
        )
        (print "Nada,no ha habido suerte")
     )
  )
)
                                )
                               )
                              )
                             )
                            (progn
                             (print "Las longitudes dicen que no puede ser")
                             (setq resaux nil)
                            )
                          )
                         )
                    )
        )
        (print "El resultado de la busqueda ha sido ...")
        (print resaux)
        (print " La descripcion actual es ...")
        (print descripcion_actual)
        (setq padres (busca-padres descripcion_actual))
                (if resaux
                    (setq lista-todos-nodos (cdr lista-todos-nodos))
                  (setq lista-todos-nodos
                        (resta
                           (cdr lista-todos-nodos)
                           (desc (car lista-todos-nodos))
                        )
                  )
```

```
                )
            ) ;; acaba el do
          (setq props-atribs (quitar-objs descripcion_actual props-atribs))
          (if (null padres)
                (colocar fin descripcion_actual props-atribs '(RAIZ))
                (colocar fin descripcion_actual props-atribs padres)
          )
            (dolist (el atrs)
                (dolist (o objetos)
                  (remprop o el)
                )
            )
            (setq atrs nil)
          )
          ()
        )
(dolist (obcs objetos)
  (remprop obcs (car atrs))
)
(setq atrs (cdr atrs))
   )
  (if (not (equal fin 'fin)) (match) () )
  )
)
```