Tema AR–4: Resolución proposicional

José A. Alonso Jiménez
José L. Ruiz Reina

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Idea básica de inconsistencia por resolución

- **Propiedades:**
  - **La cláusula vacía es insatisfacible**
  - **Un conjunto de cláusulas S es inconsistente syss la cláusula vacía es consecuencia de S**
- **Decisión de inconsistencia como búsqueda:**
  - **Para determinar si S es inconsistente, generar consecuencias de S hasta que se obtenga la cláusula vacía**

Regla de resolución proposicional

- **Reglas de inferencia**
  - **Modus Ponens:**

```
p -> q          {-p,q}
p               {p}
------          ------
q               {q}
```

  - **Modus Tollens:**

```
p -> q          {-p,q}
-q              {-q}
------          ------
-p              {-p}
```

  - **Encadenamiento:**

```
p -> q          {-p,q}
q -> r          {-q,r}
------          ------
p -> r          {-p,r}
```

- **Regla de resolución proposicional:**

```
{p1,..., r,...,pm}
{q1,...,-r,...,qn}
--------------------
{p1,...,pm,q1,...,qn}
```

Regla de resolución proposicional

- **Def.: Sean C_1 una cláusula, L un literal de C_1 y C_2 una cláusula que contiene el complementario de L. La resolvente de C_1 y C_2 respecto de L es**

$$\text{resolvente}(C_1, C_2, L) = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$$

- **Ejemplo: $\text{resolvente}(\{p, q\}, \{\neg q, r\}, q) = \{p, r\}$**

- **Procedimiento**

```
;;; (resolvente '((- p) q) '((- q) r) 'q)      =>  ((- P) R)
;;; (resolvente '((- p) (- q)) '(q r) '(- q))  =>  ((- P) R)
;;; (resolvente '((- p) q) '((- p) (- q)) 'q)  =>  ((- P))
(defun resolvente (C1 C2 L)
  (union (remove L C1 :test #'equal)
         (remove (complementario L) C2 :test #'equal)
         :test #'equal))
```

Regla de resolución proposicional

- **Resolventes de dos cláusulas:**
  - **Def.: resolventes(C_1, C_2) es el conjunto de las resolventes entre C_1 y C_2**
  - **Ejemplos:**
    resolventes$(\{\neg p, q\}, \{p, \neg q\})$ $= \{\{p, \neg p\}, \{q, \neg q\}\}$
    resolventes$(\{\neg p, q\}, \{p, q\})$ $= \{\{q\}\}$
    resolventes$(\{\neg p, q\}, \{q, r\})$ $= \{\}$
  - **Nota: $\{\} \notin$ resolventes$(\{p, q\}, \{\neg p, \neg q\})$**
  - **Procedimiento**

```
;;; (resolventes '((- p) q) '(p (- q)))
;;; =>  ((Q (- Q)) ((- P) P))
;;; (resolventes '((- p) q) '(q r))
;;; =>  NIL
(defun resolventes (C1 C2)
  (mapcar #'(lambda (L) (resolvente C1 C2 L))
          (intersection C1
                        (mapcar #'complementario C2)
                        :test #'equal)))
```

Regla de resolución proposicional

- **Adecuación de la regla de resolución:**
  - $C \in$ resolventes$(C_1, C_2) \Longrightarrow \{C_1, C_2\} \models C$
- **CNS de inconsistencia**
  - Sean S un conjunto de cláusulas, $C_1, C_2 \in S$ y $C \in$ resolventes(C_1, C_2)
    Entonces S es inconsistente $\Longleftrightarrow S \cup \{C\}$ es inconsistente
- **Resolventes de una cláusula con un conjunto de cláusulas:**
  - resolventes$(\{\neg p, q\}, \{\{p, q\}, \{q, r\}, \{\neg q, s\}\}) = \{\{q\}, \{q, r\}, \{\neg p, s\}\}$

Regla de resolución proposicional

- **Procedimiento**

```
;;; (resolventes-clausula-conjunto-1 '((- p) q) '((p q) (p r)((- q) s)))
;;; => ((Q) (Q R) ((- P) S))
(defun resolventes-clausula-conjunto-1 (C S)
  (union-general (mapcar #'(lambda (C2) (resolventes C C2))
                         S)))

;;; (resolventes-clausula-conjunto '((- p) q) '((p q) (p r)((- q) s)))
;;; => ((Q) (Q R) ((- P) S))
(defun resolventes-clausula-conjunto (C S)
  (let ((res ()))
    (loop for L in C do
      (let ((L1 (complementario L)))
        (loop for C1 in S do
          (when (member L1 C1 :test #'equal)
                (push (resolvente C C1 L) res)))))
    (nreverse res)))
```

Decisión de inconsistencia por resolución

- **Procedimiento de inconsistencia por resolución:**
  - **Descripción**

```
* Entrada: S, un conjunto de cláusulas.
* Salida: T, si S es inconsistente; NIL, en caso contrario.
* Procedimiento:
  1. Sea el SOPORTE el conjunto S y USABLES el conjunto vacío.
  2. Mientras que el SOPORTE sea no vacío,
     3. Sea ACTUAL la primera cláusula del SOPORTE,
     4. Añadir ACTUAL al principio de USABLES.
     5. Quitar ACTUAL del SOPORTE.
     6. Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES.
     7. Si una de las cláusulas de NUEVAS es la cláusula vacía, devolver T
        y terminar.
     8. Añadir las NUEVAS al final del SOPORTE.
```

Decisión de inconsistencia por resolución

- **Implementación**

```
(defun es-inconsistente-por-resolucion-1 (S)
  (let ((soporte S)                                                    ;1
        (usables ())
        (actual ())
        (nuevas ()))
    (loop until (null soporte) do                                      ;2
      (setf actual (first soporte))                                    ;3
      (setf usables (adjoin actual usables :test #'igual-conjunto))    ;4
      (setf soporte (rest soporte))                                    ;5
      (setf nuevas (resolventes-clausula-conjunto actual usables))     ;6
      (when (member nil nuevas) (return t))                            ;7
      (setf soporte (n-union soporte nuevas :test #'igual-conjunto)))))  ;8
```

Decisión de inconsistencia por resolución

- **Traza**

```
> (trace resolventes)
> (es-inconsistente-por-resolucion-1
    '(((- p) q) (p) ((- q))))
Soporte: (((- P) Q) (P) ((- Q)))
Usables: NIL
1. Trace: (RESOLVENTES '((- P) Q) '((- P) Q))
1. Trace: RESOLVENTES ==> NIL
Soporte: ((P) ((- Q)))
Usables: (((- P) Q))
1. Trace: (RESOLVENTES '(P) '((- P) Q))
1. Trace: RESOLVENTES ==> ((Q))
1. Trace: (RESOLVENTES '(P) '(P))
1. Trace: RESOLVENTES ==> NIL
Soporte: (((- Q)) (Q))
Usables: (((- P) Q) (P))
1. Trace: (RESOLVENTES '((- Q)) '((- P) Q))
1. Trace: RESOLVENTES ==> (((- P)))
1. Trace: (RESOLVENTES '((- Q)) '(P))
1. Trace: RESOLVENTES ==> NIL
1. Trace: (RESOLVENTES '((- Q)) '((- Q)))
1. Trace: RESOLVENTES ==> NIL
Soporte: ((Q) ((- P)))
Usables: (((- P) Q) (P) ((- Q)))
1. Trace: (RESOLVENTES '(Q) '((- P) Q))
1. Trace: RESOLVENTES ==> NIL
1. Trace: (RESOLVENTES '(Q) '(P))
1. Trace: RESOLVENTES ==> NIL
1. Trace: (RESOLVENTES '(Q) '((- Q)))
1. Trace: RESOLVENTES ==> (NIL)
1. Trace: (RESOLVENTES '(Q) '(Q))
1. Trace: RESOLVENTES ==> NIL
T
```

Demostraciones por resolución

- **Demostración por resolución:**
  - Def.: La sucesión $(C_1, \ldots, C_n)$ es una demostración por resolución de C a partir de S si $C = C_n$ y para todo $i \in \{1, ..., n\}$ se verifica una de las siguientes condiciones:
    * $C_i \in S$;
    * existen $j, k < i$ tales que $C_i \in$ resolventes(C_j, C_k)
  - Def.: C es demostrable por resolución a partir de S si existe una demostración por resolución de C a partir de S
  - Representación: $S \vdash C$
- **Refutación por resolución:**
  - Def.: La sucesión $(C_1, \ldots, C_n)$ es una refutación por resolución de S si es una demostración por resolución de la cláusula vacía a partir de S
  - Def.: S es refutable por resolución si existe una refutación por resolución a partir de S
  - Representación: $S \vdash \{\}$

Demostraciones por resolución

- **Ej.: Refutación de $\{\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}\}$:**

```
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
5 (2 1) {Q}
7 (4 3) {-Q}
8 (7 5) {}
```

- **Propiedades del cálculo por resolución:**
  - **Adecuación: $S \vdash \{\} \Longrightarrow S$ es inconsistente**
  - **Completitud: S es inconsistente $\Longrightarrow S \vdash \{\}$**
- **Refutabilidad como búsqueda en espacios de estados**

Demostraciones por resolución

- **Búsqueda de refutaciones**

```
> (prueba-1 '(((- p) q) (p) ((- q))))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {P}
  ** 4 (2 1) {Q}
3 NIL {-Q}
  ** 5 (3 1) {-P}
4 (2 1) {Q}
  ** 6 (4 3) {}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
4 (2 1) {Q}
6 (4 3) {}
===== Fin de la prueba =====

T
```

Cláusulas anotadas

- **Cláusulas anotadas:**
  - **Campos: número, padres, cláusula**
  - **Definición**

```
;;; (setf x (crea-clausula-anotada :numero 2 :clausula '((- p) q)))
;;; => 2 NIL {-P,Q}
;;; (ca-numero x)    =>  2
;;; (ca-padres x)    =>  NIL
;;; (ca-clausula x)  =>  ((- P) Q)
(defstruct (clausula-anotada (:constructor crea-clausula-anotada)
                             (:conc-name ca-)
                             (:print-function escribe-clausula-anotada))

  (numero 0)
  padres
  clausula)
```

Cláusulas anotadas

```
(defun escribe-clausula-anotada (clausula-anotada
                                 &optional (canal t) profundidad)
  (format canal "~d ~s ~a" (ca-numero clausula-anotada)
                           (ca-padres clausula-anotada)
                           (clausula->cadena (ca-clausula clausula-anotada))))

;;; (clausula->cadena '(p))           => "{P}"
;;; (clausula->cadena '(p (- q) r))   => "{P,-Q,R}"
(defun clausula->cadena (C)
  (if (null C)
      "{}"
      (format nil "{~a~{,~a~}}"
              (literal->cadena (first C))
              (mapcar #'literal->cadena (rest C)))))

;;; (literal->cadena '(- p))  =>  "-P"
(defun literal->cadena (L)
  (if (es-literal-positivo L)
      (format nil "~a" L)
      (format nil "-~a" (complementario L))))
```

Cláusulas anotadas

- **Resolvente de cláusulas anotadas:**

```
;;; (setf ca1 (crea-clausula-anotada :numero 1 :clausula '((- p) q)))
;;; => 1 NIL {-P,Q}
;;; (setf ca2 (crea-clausula-anotada :numero 2 :clausula '((- q) r)))
;;; => 2 NIL {-Q,R}
;;; (resolvente-ca ca1 ca2 'q)
;;; => 0 (1 2) {-P,R}
(defun resolvente-ca (CA1 CA2 L)
  (crea-clausula-anotada
   :padres (list (ca-numero CA1) (ca-numero CA2))
   :clausula (resolvente (ca-clausula CA1) (ca-clausula CA2) L)))
```

Cláusulas anotadas

- **Resolventes de cláusulas anotadas:**

```
;;; (setf ca1 (crea-clausula-anotada :numero 1 :clausula '((- p) q)))
;;; => 1 NIL {-P,Q}
;;; (setf ca2 (crea-clausula-anotada :numero 2 :clausula '(p (- q))))
;;; => 2 NIL {P,-Q}
;;; (resolventes-ca ca1 ca2)
;;; => (0 (1 2) {Q,-Q} 0 (1 2) {-P,P})
(defun resolventes-ca (CA1 CA2)
  (mapcar #'(lambda (L) (resolvente-ca CA1 CA2 L))
          (intersection (ca-clausula CA1)
                        (mapcar #'complementario (ca-clausula CA2))
                        :test #'equal)))
```

Cláusulas anotadas

- **Resolventes de una cláusula anotada con un conjunto de cláusulas**

```
;;; (setf ca1 (crea-clausula-anotada :numero 1 :clausula '((- p) q)))
;;; (setf ca2 (crea-clausula-anotada :numero 2 :clausula '(p q)))
;;; (setf ca3 (crea-clausula-anotada :numero 3 :clausula '(p r)))
;;; (setf ca4 (crea-clausula-anotada :numero 4 :clausula '((- q) s)))
;;; (resolventes-ca-conjunto-1 ca1 (list ca2 ca3 ca4))
;;; => (0 (1 2) {Q} 0 (1 3) {Q,R} 0 (1 4) {-P,S})
(defun resolventes-ca-conjunto-1 (CA S)
  (union-general (mapcar #'(lambda (CA2) (resolventes-ca CA CA2))
                         S)))

(defun resolventes-ca-conjunto (CA S)
  (let ((res ()))
    (loop for L in (ca-clausula CA) do
      (let ((L1 (complementario L)))
        (loop for CA1 in S do
          (when (member L1 (ca-clausula CA1) :test #'equal)
                (push (resolvente-ca CA CA1 L) res)))))
    (nreverse res)))
```

Búsqueda de prueba por saturación

- **Ejemplo**

```
> (prueba-1 '(((- p) q) (p) ((- q))))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {P}
  ** 4 (2 1) {Q}
3 NIL {-Q}
  ** 5 (3 1) {-P}
4 (2 1) {Q}
  ** 6 (4 3) {}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {P}
3 NIL {-Q}
4 (2 1) {Q}
6 (4 3) {}
===== Fin de la prueba =====

T
```

Búsqueda de prueba por saturación

- **Descripción**

```
* Entrada: S, un conjunto de cláusulas.
* Salida: Una refutación de S, si S es inconsistente;
          NIL, en caso contrario.
* Procedimiento:
  1. Sea el SOPORTE el conjunto S y USABLES el conjunto vacío.
  2. Mientras que el SOPORTE sea no vacío y no se tenga una refutación,
     3. Sea ACTUAL la primera cláusula del SOPORTE,
     4. Añadir ACTUAL al principio de USABLES.
     5. Quitar ACTUAL del SOPORTE.
     6. Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES.
     7. Para cada cláusula C de NUEVAS,
        8. Añadir C al final del SOPORTE.
        9. Si C es la cláusula vacía, escribir la refutación y terminar.
```

Búsqueda de prueba por saturación

- **Procedimiento**

```
(defun prueba-1 (S)
  (let ((soporte ()) (usables ()) (actual ())  (nuevas ()) (probada nil))
    (setf *contador* 0)
    (setf soporte (inicia-soporte S))                                          ;1
    (loop until (null soporte) do                                              ;2
      (when probada (return t))                                                ;2
      (setf actual (first soporte))                                            ;3
      (format t "~&~a" actual)
      (setf usables (adjoin actual usables :test #'igual-clausula-anotada))   ;4
      (setf soporte (rest soporte))                                            ;5
      (setf nuevas (resolventes-ca-conjunto actual usables))                   ;6
      (loop for CA in nuevas do                                                ;7
        (numera CA)
        (format t "~&  ** ~a" CA)
        (setf soporte (n-union soporte (list CA)                               ;8
                              :test #'igual-clausula-anotada))
        (when (null (ca-clausula CA))                                          ;9
              (escribe-prueba (list CA) usables)
              (return (setf probada t)))))))
```

Búsqueda de prueba por saturación

```
(defun inicia-soporte (S)
  (let ((soporte (mapcar #'(lambda (C)
                             (crea-clausula-anotada :numero (incf *contador*)
                                                    :clausula C))
                         S)))
    (format t "~%========== Soporte ==========")
    (loop for C in soporte do (print C))
    (format t "~%====== Fin del soporte ======~%~%")
    soporte))

(defun igual-clausula-anotada (CA1 CA2)
  (igual-conjunto (ca-clausula CA1)
                  (ca-clausula CA2)))

(defun numera (ca)
  (setf (ca-numero CA) (incf *contador*)))
```

Búsqueda de prueba por saturación

```
(defun escribe-prueba (nuevas usables)
  (format t "~%~%========== Prueba ==========~%")
  (loop for CA in (clausulas-de-la-prueba nuevas usables) do
    (format t "~&~a" CA))
  (format t "~%===== Fin de la prueba =====~%~%"))

(defun clausulas-de-la-prueba (nuevas usables)
  (ordena-prueba
   (elimina-repeticiones
    (clausulas-de-la-prueba-aux nuevas usables))))

(defun clausulas-de-la-prueba-aux (nuevas usables)
  (let ((final (find-if #'(lambda (CA) (null (ca-clausula CA))) nuevas)))
    (cons final
          (antecesores final usables))))
```

Búsqueda de prueba por saturación

```
(defun antecesores (CA usables)
  (if (null (ca-padres CA)) ()
      (let* ((numero-de-padres (ca-padres CA))
             (numero-de-padre-1 (first numero-de-padres))
             (numero-de-padre-2 (second numero-de-padres))
             (padre-1 (find-if #'(lambda (CA)
                                   (= numero-de-padre-1 (ca-numero CA)))
                               usables))
             (padre-2 (find-if #'(lambda (CA)
                                   (= numero-de-padre-2 (ca-numero CA)))
                               usables)))
        (append (list padre-1 padre-2)
                (antecesores padre-1 usables)
                (antecesores padre-2 usables)))))

(defun elimina-repeticiones (S)
  (remove-duplicates S :test #'igual-clausula-anotada))

(defun ordena-prueba (S)
  (sort S #'< :key #'ca-numero))
```

Búsqueda de prueba con eliminaciones

- **Ejemplo de búsqueda para $S = \{\{p, q\}, \{\neg p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}\}$**
  - **Por saturación**

```
> (prueba-1 '((p q)((- p) q)(p (- q))((- p)(- q))))

========== Soporte ==========
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
====== Fin del soporte ======

1 NIL {P,Q}
2 NIL {-P,Q}
  ** 5 (2 1) {Q}
3 NIL {P,-Q}
  ** 6 (3 2) {-Q,Q}
  ** 7 (3 2) {P,-P}
  ** 8 (3 1) {P}
4 NIL {-P,-Q}
  ** 9 (4 3) {-Q}
  ** 10 (4 1) {-Q,Q}
  ** 11 (4 2) {-P}
  ** 12 (4 1) {-P,P}
5 (2 1) {Q}
  ** 13 (5 4) {-P}
  ** 14 (5 3) {P}
```

Búsqueda de prueba con eliminaciones

```
6 (3 2) {-Q,Q}
  ** 15 (6 6) {Q,-Q}
  ** 16 (6 5) {Q}
  ** 17 (6 2) {Q,-P}
  ** 18 (6 1) {Q,P}
  ** 19 (6 6) {-Q,Q}
  ** 20 (6 4) {-Q,-P}
  ** 21 (6 3) {-Q,P}
7 (3 2) {P,-P}
  ** 22 (7 7) {-P,P}
  ** 23 (7 4) {-P,-Q}
  ** 24 (7 2) {-P,Q}
  ** 25 (7 7) {P,-P}
  ** 26 (7 3) {P,-Q}
  ** 27 (7 1) {P,Q}
8 (3 1) {P}
  ** 28 (8 7) {P}
  ** 29 (8 4) {-Q}
  ** 30 (8 2) {Q}
9 (4 3) {-Q}
  ** 31 (9 6) {-Q}
  ** 32 (9 5) {}

========== Prueba ==========
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
5 (2 1) {Q}
9 (4 3) {-Q}
32 (9 5) {}
===== Fin de la prueba =====

T
```

Búsqueda de prueba con eliminaciones

- **Por saturación con eliminaciones**

```
> (prueba-2 '((p q)((- p) q)(p (- q))((- p)(- q))))

========== Soporte ==========
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
====== Fin del soporte ======

1 NIL {P,Q}
2 NIL {-P,Q}
  ** 5 (2 1) {Q}
3 NIL {P,-Q}
  ** 6 (3 1) {P}
4 NIL {-P,-Q}
  ** 7 (4 3) {-Q}
8 (7 5) {}

========== Prueba ==========
1 NIL {P,Q}
2 NIL {-P,Q}
3 NIL {P,-Q}
4 NIL {-P,-Q}
5 (2 1) {Q}
7 (4 3) {-Q}
8 (7 5) {}
===== Fin de la prueba =====

T
```

Búsqueda de prueba con eliminaciones

- **Cláusulas tautológicas**
  - **Def.: C es tautología $\Longleftrightarrow$ C contiene un literal y su complementario**
  - **Ejemplos:**
    $\{p, q, \neg p\}$ es tautología
    $\{p, q, \neg r\}$ no es tautología
  - **Procedimiento**

```
;;; (es-tautologia '(p q (- p)))  =>  T
;;; (es-tautologia '(p q (- r)))  =>  NIL
;;; (es-tautologia '())           =>  NIL
(defun es-tautologia (C)
  (if (some #'(lambda (L) (member (complementario L)
                                  C
                                  :test #'equal))
            C)
      t))
```

  - **C es tautología $\Longleftrightarrow$ C es válida**
  - **Si $C \in S$ y C es tautología, entonces**
    S es inconsistente $\Longleftrightarrow$ $S - \{C\}$ es inconsistente

Búsqueda de prueba con eliminaciones

- **Cláusulas unitarias:**
  - **Def.:** C **es unitaria** $\Longleftrightarrow$ C **tiene sólo un literal.**
  - **Procedimiento**

```
;;; (es-unitaria '(p))       =>  T
;;; (es-unitaria '((- p)))   =>  T
;;; (es-unitaria '(p q))     =>  NIL
(defun es-unitaria (C)
  (= (length C) 1))
```

Búsqueda de prueba con eliminaciones

- **Búsqueda de prueba por saturación con eliminaciones**
  - **Descripción**

```
* Entrada: S, un conjunto de cláusulas.
* Salida: Una refutación de S, si S es inconsistente; NIL, en caso contrario.
* Procedimiento:
  1. Sea el SOPORTE el conjunto S y USABLES el conjunto vacío.
  2. Mientras que el SOPORTE sea no vacío y no se tenga una refutación,
     3. Sea ACTUAL la primera cláusula del SOPORTE,
     4. Añadir ACTUAL al principio de USABLES.
     5. Quitar ACTUAL del SOPORTE.
     6. Sea NUEVAS las resolventes de ACTUAL con las cláusulas de USABLES,
        que no sean tautologías ni estén en las USABLES o en el SOPORTE.
     7. Para cada cláusula C de NUEVAS,
        8.  Añadir C al final del SOPORTE.
        9.  Si C es la cláusula vacía, escribir la refutación y terminar.
        10. Si C es unitaria y su complementaria está en las USABLES o en el
            SOPORTE, escribir la refutación y terminar.
```

Búsqueda de prueba con eliminaciones

```
(defun prueba-2 (S)
  (let ((soporte ()) (usables ()) (actual ()) (nuevas ()) (probada nil))
    (setf *contador* 0)
    (setf soporte (inicia-soporte S))                                  ;1
    (loop until (null soporte) do                                      ;2
      (when probada (return t))                                        ;2
      (setf actual (first soporte))                                    ;3
      (format t "~&~a" actual)
      (setf usables (cons actual usables))                             ;4
      (setf soporte (rest soporte))                                    ;5
      (setf nuevas (nuevas-resolventes actual usables soporte))        ;6
      (loop for CA in nuevas do                                        ;7
        (numera CA) (format t "~&  ** ~a" CA)
        (setf soporte (append soporte (list CA)))                      ;8
        (cond ((null (ca-clausula CA))                                 ;9
               (escribe-prueba (list CA) usables)
               (return (setf probada t)))
              ((and (es-unitaria (ca-clausula CA))                     ;10
                    (complementaria CA usables soporte))
               (procesa-unitaria CA usables soporte)
               (return (setf probada t)))))))))
```

Búsqueda de prueba con eliminaciones

```
(defun nuevas-resolventes (actual usables soporte)
  (let ((usables-y-soporte (append usables soporte)))
    (loop for CA in (resolventes-ca-conjunto actual usables)
      when (and (not (es-tautologia (ca-clausula CA)))
                (not (member CA usables-y-soporte
                             :test #'igual-clausula-anotada)))
      collect CA)))

(defun complementaria (CA usables soporte)
  (let ((L (complementario (first (ca-clausula CA)))))
    (find-if #'(lambda (CA1)
                 (and (es-unitaria (ca-clausula CA1))
                      (equal L (first (ca-clausula CA1)))))
             (union usables soporte))))
```

Búsqueda de prueba con eliminaciones

```
(defun procesa-unitaria (CA usables soporte)
  (let ((CA1 (complementaria CA usables soporte))
        (CA2 ()))
    (when CA1
          (setf CA2 (first (resolventes-ca CA CA1)))
          (numera CA2)
          (format t "~&~a" CA2)
          (escribe-prueba (list CA2)
                          (cons CA (cons CA1 usables))))))

(defun prueba (S)
  (prueba-2 S))
```

- **Propiedades de los procedimientos de prueba:**
  - **Terminación**
  - **Corrección**

Prueba de validez mediante resolución

- **Reducción de validez a refutación por resolución**
  - **Propiedad:** $\models F \Longleftrightarrow$ **Cláusulas**$(\neg F) \vdash \{\}$
  - **Procedimiento**

```
(defun prueba-validez (F)
  (prueba (clausulas (negacion F))))
```

  - **Ejemplo:** $\models (p \rightarrow q) \vee (q \rightarrow p)$

```
> (prueba-validez '((p -> q) / (q -> p)))

========== Soporte ==========
1 NIL {P}
2 NIL {-Q}
3 NIL {Q}
4 NIL {-P}
====== Fin del soporte ======

1 NIL {P}
2 NIL {-Q}
3 NIL {Q}
  ** 5 (3 2) {}

========== Prueba ==========
2 NIL {-Q}
3 NIL {Q}
5 (3 2) {}
===== Fin de la prueba =====

T
```

Consecuencia mediante resolución

- **Reducción de consecuencia a refutación por resolución**
  - Sea S un conjunto de fórmulas y F una fórmula. Entonces
    $S \models F \Longleftrightarrow \text{Cláusulas}(S \cup \{\neg F\}) \vdash \{\}$
  - **Procedimiento**

```
(defun prueba-consecuencia (S F)
  (prueba (clausulas-conjunto
            (append S
                    (list (negacion F))))))
```

Consecuencia mediante resolución

- **Ejemplo 1:** $\{p \to q, q \to r\} \models p \to r$

```
> (prueba-consecuencia '((p -> q) (q -> r))
                       '(p -> r))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {P}
4 NIL {-R}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {-Q,R}
  ** 5 (2 1) {R,-P}
3 NIL {P}
  ** 6 (3 1) {Q}
4 NIL {-R}
  ** 7 (4 2) {-Q}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {-Q,R}
3 NIL {P}
4 NIL {-R}
6 (3 1) {Q}
7 (4 2) {-Q}
8 (7 6) {}
===== Fin de la prueba =====

T
```

Consecuencia mediante resolución

- **Ejemplo 2:** $\{p\} \not\models p \wedge q$

```
> (prueba-consecuencia '(p)
                       '(p & q))

========== Soporte ==========
1 NIL {P}
2 NIL {-P,-Q}
====== Fin del soporte ======

1 NIL {P}
2 NIL {-P,-Q}
  ** 3 (2 1) {-Q}
3 (2 1) {-Q}
NIL
```

Consecuencia mediante resolución

- **Ejemplo 3:**
  $\{p \to q, m \to p \vee q\} \models m \to q$

```
> (prueba-consecuencia '((p  -> q) (m -> (p / q)))
                       '(m -> q))

========== Soporte ==========
1 NIL {-P,Q}
2 NIL {-M,P,Q}
3 NIL {M}
4 NIL {-Q}
====== Fin del soporte ======

1 NIL {-P,Q}
2 NIL {-M,P,Q}
  ** 5 (2 1) {-M,Q}
3 NIL {M}
  ** 6 (3 2) {P,Q}
4 NIL {-Q}
  ** 7 (4 1) {-P}
  ** 8 (4 2) {-M,P}
5 (2 1) {-M,Q}
  ** 9 (5 4) {-M}

========== Prueba ==========
1 NIL {-P,Q}
2 NIL {-M,P,Q}
3 NIL {M}
4 NIL {-Q}
5 (2 1) {-M,Q}
9 (5 4) {-M}
10 (9 3) {}
===== Fin de la prueba =====

T
```

Problema de animales

```
> (prueba-consecuencia
   '(((tiene_pelos / da_leche) -> es_mamifero)
     ((es_mamifero & (tiene_pezugnas / rumia)) -> es_ungulado)
     ((es_ungulado & tiene_cuello_largo) -> es_jirafa)
     ((es_ungulado & tiene_rayas_negras) -> es_cebra)
     (tiene_pelos & (tiene_pezugnas & tiene_rayas_negras)))
   'es_cebra)

========== Soporte ==========
1 NIL {-TIENE_PELOS,ES_MAMIFERO}
2 NIL {-DA_LECHE,ES_MAMIFERO}
3 NIL {-ES_MAMIFERO,-TIENE_PEZUGNAS,ES_UNGULADO}
4 NIL {-ES_MAMIFERO,-RUMIA,ES_UNGULADO}
5 NIL {-ES_UNGULADO,-TIENE_CUELLO_LARGO,ES_JIRAFA}
6 NIL {-ES_UNGULADO,-TIENE_RAYAS_NEGRAS,ES_CEBRA}
7 NIL {TIENE_PELOS}
8 NIL {TIENE_PEZUGNAS}
9 NIL {TIENE_RAYAS_NEGRAS}
10 NIL {-ES_CEBRA}
====== Fin del soporte ======
```

Problema de animales

```
1 NIL {-TIENE_PELOS,ES_MAMIFERO}
2 NIL {-DA_LECHE,ES_MAMIFERO}
3 NIL {-ES_MAMIFERO,-TIENE_PEZUGNAS,ES_UNGULADO}
  ** 11 (3 1) {-TIENE_PEZUGNAS,ES_UNGULADO,-TIENE_PELOS}
  ** 12 (3 2) {-TIENE_PEZUGNAS,ES_UNGULADO,-DA_LECHE}
....

21 (9 6) {-ES_UNGULADO,ES_CEBRA}
  ** 46 (21 10) {-ES_UNGULADO}

========== Prueba ==========
1 NIL {-TIENE_PELOS,ES_MAMIFERO}
3 NIL {-ES_MAMIFERO,-TIENE_PEZUGNAS,ES_UNGULADO}
6 NIL {-ES_UNGULADO,-TIENE_RAYAS_NEGRAS,ES_CEBRA}
7 NIL {TIENE_PELOS}
8 NIL {TIENE_PEZUGNAS}
9 NIL {TIENE_RAYAS_NEGRAS}
10 NIL {-ES_CEBRA}
19 (7 1) {ES_MAMIFERO}
20 (8 3) {-ES_MAMIFERO,ES_UNGULADO}
21 (9 6) {-ES_UNGULADO,ES_CEBRA}
45 (20 19) {ES_UNGULADO}
46 (21 10) {-ES_UNGULADO}
47 (46 45) {}
===== Fin de la prueba =====

T
```

Referencias

- **Chang, C–L y Lee, R. C–T. *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, 1973)**
  - **Cap. 5: "The resolution principle"**
- **Genesereth, M.R. y Nilsson, N.J. *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann, 1987)**
  - **Cap. 4 "Resolution"**
- **Lucas, P. y Gaag, L.v.d. *Principles of Expert Systems* (Addison–Wesley, 1991).**
  - **Cap. 2 "Logic and resolution"**
- **Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994)**
  - **Cap. 5.4.3 "Resolución en lógica proposicional"**

Referencias

- Russell, S. y Norvig, P. *Inteligencia artificial (un enfoque moderno)* (Prentice–Hall, 1996)
  - Cap. 9.5 "Resolución: un procedimiento completo de inferencia"
- Thayse, A. y otros *Aproche logique de l'Intelligence Artificielle. (Vol 1: de la logique classique à la programmation logique).* (Dunod, 1988)
  - Cap. 1.1 "Calcul des propositions"