Verificación automática de sistemas de razonamiento*

(aplicación a la enseñanza de la Inteligencia Artificial)

J.L. Ruiz, F.J. Martín, J.A. Alonso, M.J. Hidalgo
Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
{jruiz,fjesus,jalonso,mjoseh}@cica.es

Resumen

En este artículo proponemos el uso de sistemas de demostración automática como herramienta práctica en la enseñanza de las asignaturas de Ciencias de la Computación e Inteligencia Artificial. Esta propuesta se ilustra con un ejemplo: la verificación de la corrección y completitud de un pequeño sistema de demostración automática de fórmulas proposicionales. Para ello usamos el demostrador de Boyer y Moore.

1 Introducción

Uno de los usos principales de los sistemas de deducción y verificación automática de teoremas es el razonamiento sobre propiedades de programas. Es decir, dado un programa, se formalizan las propiedades que se le suponen, en forma de teoremas; la veracidad de esos teoremas se comprueba mediante un sistema de razonamiento automático. El programa se convierte así en un objeto sobre el que razonar. Varios son los objetivos que se consiguen:

- En primer lugar, se *formalizan* las propiedades esperadas de un algoritmo, con lo que queda mucho más clara la estructura lógica del mismo.
- Las pruebas sobre propiedades de programas suelen ser mucho más largas y pesadas que las de resultados matemáticos. Es especialmente conveniente el tratarlas mecánicamente.
- Por otro lado, supone un proceso *efectivo* de asegurar la corrección y otras propiedades de los programas.

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

Esta verificación automática de teoremas puede ser utilizada para la prueba de propiedades de algoritmos dentro del campo de la Inteligencia Artificial, incluso dentro del propio campo del Razonamiento Automático.

En este artículo proponemos la verificación automática como instrumento en la enseñanza de la Inteligencia Artificial, disponiendo así de una herramienta para dar rigor y razonar sobre algunas de sus técnicas más conocidas, ayudando a una comprensión más profunda de la materia.

Como ejemplo, planteamos la verificación automática de la corrección y completitud de un demostrador de fórmulas en lógica proposicional. La verificación se realiza usando el demostrador de Boyer y Moore.

2 El demostrador de Boyer y Moore.

El demostrador de Boyer y Moore, también conocido como **NQTHM**, implementa una lógica computacional, cuya principal característica es la automatización del principio de inducción, que lo hace especialmente adecuado para la demostración de propiedades de programas. Una impresionante lista de teoremas han sido probados y verificados en NQTHM, en el campo de las matemáticas, el *software* y el *hardware*. Una detallada información del sistema se puede encontrar en [**?**] y [**?**].

2.1 La lógica computacional.

Hacemos aquí una breve descripción de la lógica que usa el demostrador de Boyer y Moore (llamada *lógica computacional*) para formalizar y demostrar teoremas. Se trata de un fragmento

de la lógica de primer orden, sin cuantificadores. Su sintaxis y semántica está basada en el lenguaje Lisp. Por tanto, usa una notación prefija para denotar términos. Por ejemplo, escribimos `(and p q)` en lugar de $p \wedge q$.

Incluye la lógica proposicional, variables, símbolos de función y el predicado de igualdad. Además de las conectivas usuales (`AND`, `OR`, `NOT`, `IMPLIES`), se incluye la conectiva `IF` (la función if-then-else).

La lógica incluye inicialmente la definición de los números naturales y de los pares ordenados, como objetos construidos recursivamente:

- Los naturales, obtenidos a partir del objeto base `(ZERO)` y del constructor `ADD1` (la función sucesor). Su función de acceso (destructor) es `SUB1`.
- Los pares ordenados, obtenidos a partir del constructor `CONS`, con los destructores `CAR` y `CDR`.

El usuario puede definir nuevos tipos de datos, dando nuevos objetos base, constructores y destructores. Esto hará que nuevos axiomas, describiendo al nuevo tipo de dato, se añadan a la lógica. La característica principal de estos tipos de datos definidos por el usuario es que pueden estar definidos recursivamente. Esto es especialmente útil, ya que la mayoría de las propiedades de los programas se prueban por inducción en la estructura de los datos que reciben como entrada.

Si se quiere que el demostrador verifique propiedades de algoritmos, éstos se tienen que poder definir dentro de la lógica. Esto se consigue mediante el uso de funciones al estilo LISP. Por ejemplo, podríamos definir un algoritmo de concatenación de listas de la siguiente manera:

```
(defn conc (l1 l2)
  (if (not (listp l1))
      l2
    (cons (car l1)
          (conc (cdr l1) l2))))
```

Sólo se admite la definición de una función después de que el demostrador haya sido capaz de probar que es total (*principio de definición*). Esto asegura la consistencia de la lógica. La prueba de que una función es total consiste en ver que sus argumentos decrecen en cada llamada recursiva, respecto de una determinada "medida", que se supone bien fundamentada.

Además de las reglas de inferencia de la lógica proposicional, de la igualdad, y la instanciación, la lógica incluye un *principio de inducción matemática*. Este principio, en esencia, asegura que para demostrar que una propiedad es cierta para un determinado objeto, podemos suponer que es cierta para objetos más pequeños (respecto a cierta "medida" bien fundamentada).

Existe una dualidad entre los principios de definición y de inducción, de manera que argumentos similares a los usados en la prueba de la admisión de la definición de una función, pueden servir para probar por inducción determinadas propiedades de la misma.

2.2 El demostrador.

El usuario puede pedir al sistema que intente la prueba de un determinado teorema expresado con el lenguaje de la lógica computacional. Por ejemplo, para intentar demostrar que la función que concatena listas (que hemos definido anteriormente) es asociativa, escribiriamos la siguiente expresión:

```
(prove-lemma asociatividad-conc (rewrite)
  (equal (conc x (conc y z))
         (conc (conc x y) z)))
```

Esto hace que el sistema aplique una serie de técnicas encaminadas a encontrar una prueba del teorema requerido. Si el sistema lo consigue, es posible reconstruir una prueba del teorema usando los axiomas y esquemas de inferencia de la lógica anteriormente referida. Además, el resultado puede ser usado en posteriores pruebas. Si el sistema falla en el intento, nada podemos asegurar sobre la validez del teorema.

La principal técnica usada es la de simplificación, cuyo componente básico es un sistema de reglas de reescritura, obtenidas a partir de resultados anteriores.

La segunda técnica en importancia consiste en aplicar el principio de inducción. Presenta la dificultad de tener que escoger un esquema de inducción adecuado para la prueba. Una de las principales aportaciones del sistema de B. & M. es la de obtener esos esquemas de manera mecánica, mediante una serie de heurísticas.

Otras técnicas se pueden aplicar también a una fórmula como parte del intento de probar que es teorema: generalización, eliminación de irrelevancias, eliminación de destructores, etc.

2.3 El papel del usuario.

Aunque, en sentido estricto, NQTHM no es interactivo, de alguna manera, el usuario puede influir en la búsqueda de una demostración.

Si un resultado se ha demostrado previamente, éste puede ser usado por el demostrador en la prueba de posteriores resultados. La manera en que el sistema usa estos teoremas previos, es, generalmente, almacenándolos en forma de reglas de simplificación. Por ejemplo, una vez probada la asociatividad de `conc`, podremos usarla en intentos de prueba posteriores para reescribir una expresión de la forma `(conc X (conc Y Z))` como `(conc (conc X Y) Z)`.

Usualmente, el sistema es incapaz de probar un resultado de una cierta complejidad sin ninguna información adicional, aún cuando se trate de un teorema demostrable en la lógica del sistema.

El proceso típico que se sigue en un intento de demostración es el siguiente:

- El usuario conoce una prueba a mano de un resultado y quiere verificarla.
- Formaliza el resultado en la lógica del sistema.
- Guía al demostrador hacia la prueba conocida, demostrando lemas previos.
- Usualmente, en primera instancia, la prueba de un teorema por parte del sistema suele fallar (a no ser que sea muy "simple"). Inspeccionando esta prueba fallida es posible obtener como consecuencia que se necesitan una serie de lemas que han de ser probados previamente.

Como resultado final la prueba de un teorema consiste en una serie de lemas que van aportando al sistema el *conocimiento* suficiente como para permitirle completar la prueba del resultado principal.

3 Verificación de un sistema de demostración proposicional.

3.1 Introducción.

Presentamos en esta sección un ejemplo muy ilustrativo del uso de NQTHM en la demostración de propiedades de un algoritmo, basado en un ejemplo presentado en [**?**]. Se trata de definir un algoritmo que decide si una fórmula de la lógica proposicional es tautología. Está basado en la técnica conocida como *Diagramas de Decisión Binaria (BDDs)*, ver [**?**]. La idea principal consiste en transformar la fórmula a otra cuya única conectiva sea *if-then-else* y que sólo tenga átomos en el *test*. Es sobre estas fórmulas *normalizadas* donde se decide la propiedad de ser tautología. Usaremos NQTHM para probar la corrección y completitud de este demostrador de tautologías.

Varias son las características que hacen especialmente interesante el ejemplo planteado desde el punto de vista docente. Cualquier sistema lógico tiene como una de sus componentes básicas la lógica proposicional, y esto la hace uno de los puntos fundamentales en el estudio de los sistemas de razonamiento automático. Además, el algoritmo contiene varios elementos de uso muy común en cualquier sistema de demostración automática: paso a formas normales, codificación de suposiciones y simplificación. Además, el uso de NQTHM hace que sea posible definir el demostrador de tautologías, ejecutarlo y verificar sus propiedades dentro del mismo entorno.

Es necesario hacer hincapié en diferenciar entre la lógica proposicional del demostrador de tautologías y la lógica computacional (propia de NQTHM) que usamos para razonar y demostrar propiedades sobre el demostrador de tautologías. La lógica computacional actúa aquí como una *metalógica* para el estudio de la lógica proposicional. Al usar un demostrador automático para verificar procesos, sistemas o algoritmos que se usan en los demostradores, puede existir alguna confusión por parte del alumno en discernir qué forma parte del demostrador que se usa, y qué del sistema que se verifica.

3.2 Formalización de la lógica proposicional.

Las *fórmulas* de la lógica proposicional son `T` (verdad), `F` (falso), *variables* proposicionales (`p,q,r,s,...`), o expresiones de la forma `(Y F1 F2)`, `(O F1 F2)`, `(NO F1 F2)`, `(IMPLICA F1 F2)`, y `(SI F1 F2 F3)`, siendo `F1, F2, F3` fórmulas. Las variables, `T` y `F` se denominan fórmulas *atómicas*.

Las *conectivas* `Y, O, NO, IMPLICA` tienen el

significado clásico de conjunción, disyunción, negación e implicación respectivamente. La conectiva `SI` se corresponde con la construcción *if-then-else* ya conocida en programación: si el valor de su primer argumento (*test*) es `T`, su valor es el de su segundo argumento (*rama izquierda*), y si es `F`, el valor del tercer argumento (*rama derecha*).

Introducimos esta definición de fórmula proposicional en el sistema con un tipo de dato para cada conectiva. Así, por ejemplo, las fórmulas cuya conectiva principal es `SI` quedan definidas de la siguiente forma:

```
(add-shell SI nil si-expresion
        ((test-si (none-of) zero)
         (rama-izq-si (none-of) zero)
         (rama-der-si (none-of) zero)))
```

Nótese que en la definición especificamos el constructor del dato (la función `SI`), tres funciones para acceder a sus argumentos (`test-si, rama-izq-si, rama-der-si`) y una función para reconocer el tipo (`si-expresion`).

De manera análoga se definen el resto de las conectivas. Esto nos permite expresar las fórmulas proposicionales dentro del sistema, y poder razonar sobre ellas. Por ejemplo, la fórmula $p \rightarrow (q \vee (r \wedge T))$ se representa como `(IMPLICA p (O q (Y r T)))`.

Una *valoración* es una función del conjunto de las variables proposicionales en {`T`,`F`}. Las valoraciones las representaremos como listas de asociación. Por ejemplo, la valoración que asigna `F` a todas las variables excepto a `q` y a `r`, se representa por la lista `((p . F) (q . T) (r . T))`. Nótese que si una variable no aparece en la valoración, se considera implícitamente falsa. Además, en lo que sigue, identificaremos cualquier valor distinto de `F` con `T`.

Las valoraciones se extienden de manera natural al conjunto de las fórmulas, sin más que interpretar el significado de cada conectiva. Esto se traduce en la siguiente definición:

```
(defn valor (x v)
  (cond
    ((y-expresion x)
     (if (valor (y-izq x) v)
         (valor (y-der x) v)  f))
     ((o-expresion x)
      (if (valor (o-izq x) v)
          t (valor (o-der x) v)))
     ((no-expresion x)
      (if (valor (no-comp x) v) f  t))
     ((implica-expresion x)
      (if (valor (implica-antecedente x) v)
          (valor (implica-consecuente x) v)
          t))
     ((si-expresion x)
      (if (valor (test-si x) v)
          (valor (rama-izq-si x) v)
          (valor (rama-der-si x) v)))
     (t (asignacion x v))))
```

Nótese que se usa una función auxiliar, `asignacion`, que devuelve el valor de una variable respecto de una valoración, y cuya definición omitimos.

Recordemos que el sistema no admite la definición de una función sin antes probar que la función es total. En este caso, el sistema lo prueba sin dificultad, ya que las llamadas recursivas que tiene la función `valor` son todas sobre subfórmulas de la fórmula de entrada.

Desde el punto de vista docente, es importante destacar que todas las definiciones, como ésta de la función `valor`, son *implementadas*. Esto nos permite movernos en el plano teórico y práctico al mismo tiempo.

3.3 El demostrador de tautologías.

El algoritmo de decisión de tautologías lo estructuramos en tres partes bien diferenciadas:

1. Traducción de la fórmula a una equivalente que sólo contenga la conectiva `SI` (fórmulas `SI`-*puras*).

2. *Normalización* de la fórmula `SI`-pura. Es decir, transformación en una equivalente que sólo tenga fórmulas atómicas en el primer argumento de cada una de las conectivas `SI` (denominadas *Diagramas de Decisión Binaria ó BDDs*).

3. Aplicación de un algoritmo que decide si un BDD es una tautología.

Por fórmulas *equivalentes* entendemos aquellas cuyos valores son iguales respecto de cualquier valoración. Una *tautología* es una fórmula cuyo valor es `T` respecto de cualquier valoración.

Estos tres pasos se implementan dentro del sistema con tres funciones. Nótese que, posiblemente, un algoritmo más eficiente se consiga intercalando los tres procesos descritos. Sin embargo, es preferible plantearlos de manera secuencial, ya que la prueba de sus propiedades resulta así mucho más clara.

El primer paso lo realiza la función `transforma-en-si`, cuya definición omitimos. Es una sencilla recursión en la estructura de la fórmula, transformando cada conectiva, como se indica en la tabla **??**.

Conectiva	**Traducción**
`(Y p q)`	`(SI p q F)`
`(O p q)`	`(SI p T q)`
`(NO p)`	`(SI p F T)`
`(IMPLICA p q)`	`(SI p q T)`

Tabla 1: Traducción a SI-fórmula.

El segundo paso queda implementado por la función `normaliza`, que puede verse como la aplicación de la siguiente regla de reescritura a todos los niveles de la fórmula:

```
(SI (SI P Q R) U V) -->
    (SI P (SI Q U V) (SI R U V))
```

```
(defn normaliza (x)
 (if (si-expresion x)
     (if (si-expresion (test-si x))
         (normaliza
          (si (test-si (test-si x))
              (si (rama-izq-si (test-si x))
                  (rama-izq-si x)
                  (rama-der-si x))
              (si (rama-der-si (test-si x))
                  (rama-izq-si x)
                  (rama-der-si x))))
        (si (test-si x)
           (normaliza (rama-izq-si x))
           (normaliza (rama-der-si x))))
    x)
```

Esta función presenta un problema no trivial para su admisión. El sistema no descubre inmediatamente que la función termina para cualquier entrada. La razón es que la fórmula aumenta de tamaño cada vez que se reescribe (aún cuando desciende el tamaño de los tests). El usuario ha de hacer que el demostrador pruebe una serie de lemas para que sea posible su admisión.

Por último se ha de definir un función que decida si un BDD es tautología:

```
(defn es-tautologia-normalizada (x val)
 (if (si-expresion x)
     (if (asignada (test-si x) val)
         (if (asignacion (test-si x) val)
             (es-tautologia-normalizada
                   (rama-izq-si x) val)
             (es-tautologia-normalizada
                   (rama-der-si x) val))
        (and (es-tautologia-normalizada
                   (rama-izq-si x)
                   (suponer-que-es-verdad
                      (test-si x) val))
             (es-tautologia-normalizada
                   (rama-der-si x)
                   (suponer-que-es-falso
                      (test-si x) val))))
    (asignacion x val)))
```

La funciones `suponer-que-es-verdad` y `suponer-que-es-falso` reciben una variable y una valoración y devuelven la valoración ampliada con la asignación a la variable del valor `T` o `F`, respectivamente.

Nótese que un BDD puede verse como un árbol binario, en el que por cada conectiva `SI` se tiene un nodo (con el test) y los subárboles de la rama izquierda y derecha respectivamente. La función definida recorre este arbol, optando por la rama izquierda o derecha dependiendo del valor `T` o `F` del test. Si en un nodo aparece una variable que no tiene asignado ningún valor, se recorren los subárboles izquierdo y derecho, asumiendo, respectivamente, que la variable vale `T` o `F`. Estas *suposiciones* sobre el valor de las variables se representan como valoraciones y aparecen como segundo argumento de la función. Un BDD es tautología si todas sus ramas toman el valor `T`.

El algoritmo de decisión de tautologías consiste en la composición de los tres algoritmos vistos:

```
(defn es-tautologia (x)
  (es-tautologia-normalizada
      (normaliza
        (transforma-en-si x)) nil))
```

En este momento, el demostrador de tautologías está definido, y estas definiciones son axiomas dentro de la lógica del demostrador. Esto nos permite disponer de él como un *objeto sobre el que razonar*.

Pero además, se trata de una definición ejecutable. Es decir, disponemos de un entorno en el que poder evaluar llamadas a la función. Por ejemplo:

```
>(r-loop)
...
*(setq f1 (Y (IMPLICA 'p 'r)
             (IMPLICA 'p 's)))
*(setq f2
   (IMPLICA (Y (Y (IMPLICA 'P 'R)
                  (IMPLICA 'P 'S))
               (Y (IMPLICA 'Q 'R)
                  (IMPLICA 'Q 'S)))
        (IMPLICA (O 'P 'Q) (Y 'R 'S))))
*(setq v1 (list (cons 'p T) (cons 'q F)
              (cons 'r T) (cons 's F)))
*(es-tautologia f1)
 F    ;;; f1 NO es tautologia
*(valor f1 v1)
 F    ;;; contraejemplo
*(es-tautologia f2)
 T    ;;; f2 SI es tautologia.
*(valor f2 v1)
 T    ;;; ejemplo
```

3.4 Formalización de la corrección y completitud.

Una vez definido el algoritmo, se trata de probar algunas de sus propiedades. Es decir, `es-tautologia` es un símbolo de función dentro de la lógica de NQTHM, con un determinado significado, y por tanto podemos expresar, mediante fórmulas de la lógica del sistema, distintas propiedades del algoritmo, propiedades que a su vez probaremos con NQTHM.

La primera propiedad que vamos a verificar es la de ser *correcto*. Es decir, si `es-tautologia` aplicada a una fórmula X devuelve `T`, entonces el valor de X respecto a cualquier valoración es `T`. Esto se traduce en el siguiente comando al demostrador:

```
(prove-lemma correccion ()
  (implies (es-tautologia x)
           (valor x val)))
```

El demostrar esta propiedad no es suficiente para asegurar que `es-tautologia` se comporta como es esperado. Una función que siempre devuelva `T` también es correcta en este sentido. Se necesita, además, un resultado de completitud:

```
(prove-lemma completitud ()
  (implies (not (es-tautologia x))
           (equal (valor x (falsifica x))
                  f)))
```

Obsérvese que la fórmula anterior se podría expresar como "si la función `es-tautologia` devuelve `F` al actuar sobre una fórmula `x`, entonces *existe* una valoración tal que el valor de la fórmula respecto de tal valoración es `F`". El problema es que la lógica computacional no tiene cuantificadores: todas las fórmulas se suponen implícitamente cuantificadas universalmente y no hay cuantificador existencial. Solución: aplicar el método de *skolemización*, que consiste en sustituir cada variable existencial por una función: en este caso, la función `falsifica`.

Pero si hemos de razonar con `falsifica`, hemos de definirla dentro de NQTHM: una función que actuando sobre una fórmula devuelva una valoración que la haga falsa o `F` si esto no es posible (es decir, si es tautología).

Esta ausencia del cuantificador existencial es, a veces, una ventaja: obliga a que las pruebas sean totalmente *constructivas*. Si se afirma la existencia de algo, ha de ser definido.

Los dos teoremas anteriores (el de corrección y el de completitud), no se demuestran en primera instancia. Como ya se ha dicho, es necesaria una cierta interacción con el sistema y tener una estrategia de prueba para buscar y demostrar en NQTHM los lemas auxiliares necesarios que hacen posible la prueba. En los siguientes apartados se describe el proceso en líneas generales. Una versión más amplia con la demostración completa está disponible en [**?**].

3.5 Prueba de la corrección.

Paso 1: Para probar la corrección, en primer lugar se prueba que el algoritmo actuando sobre expresiones `SI` puras normalizadas (es decir, BDDs), es correcto:

```
(prove-lemma es-taut-norm-correcto
 (rewrite)
 (implies
   (and (expresion-si-normalizada x)
        (si-pura x)
        (es-tautologia-normalizada x val))
  (valor x (append val val1))))
```

Este resultado lo prueba el sistema previa definición de los predicados `si-pura`, y `expresion-si-normalizada`. Nuevamente el primer intento de la prueba falla. Sin embargo, de la inspección de la prueba fallida se deduce la necesidad de disponer de una serie de lemas adicionales, con cuya ayuda ya se obtiene una demostración.

Paso 2: El proceso de normalización obtiene fórmulas normalizadas (BDDs).

```
(prove-lemma normaliza-normaliza
  (rewrite)
  (expresion-si-normalizada
            (normaliza x)))

(prove-lemma transforma-en-si-pura
  (rewrite)
  (si-pura (transforma-en-si x)))

(prove-lemma si-pura-normaliza
  (rewrite)
  (equal (si-pura (normaliza x))
         (si-pura x)))
```

Paso 3: La normalización obtiene una fórmula equivalente a la original.

```
(prove-lemma transforma-en-si-correcto
  (rewrite)
  (equal
   (valor (transforma-en-si x) val)
   (valor x val)))

(prove-lemma normaliza-es-correcto
  (rewrite)
  (equal (valor (normaliza x) val)
         (valor x val)))
```

Necesitamos un lema más para ensamblar los resultados anteriores:

```
(prove-lemma lema-puente-1
 (rewrite)
 (implies
   (and
    (es-tautologia-normalizada z val1)
    (expresion-si-normalizada z)
    (si-pura z)
    (equal (valor x val)
           (valor z (append val1 val))))
   (valor x val)))
```

Con la ayuda de este último lema, el sistema prueba directamente el teorema de corrección.

3.6 Prueba de la completitud.

La principal dificultad a la hora de probar la completitud es la definición de la función `falsifica`. Se trata de definir una función que recibiendo una fórmula como entrada devuelva una valoración que la haga falsa (o `F` si esto no es posible). Se va a transformar la fórmula a forma normal y definir una función que actúa sobre BDDs.

```
(defn falsifica-normalizada (x val)
 (if (si-expresion x)
     (if (asignada (test-si x) val)
         (if (asignacion (test-si x) val)
             (falsifica-normalizada
                   (rama-izq-si x) val)
           (falsifica-normalizada
                  (rama-der-si x) val))
       (if (falsifica-normalizada
            (rama-izq-si x)
            (suponer-que-es-verdad
                     (test-si x) val))
           (falsifica-normalizada
            (rama-izq-si x)
            (suponer-que-es-verdad
                     (test-si x) val))
         (falsifica-normalizada
          (rama-der-si x)
          (suponer-que-es-falso
                     (test-si x) val))))
   (if (asignada x val)
       (if (asignacion x val) f val)
     (cons (cons x f) val))))
```

Se prueba (con algún lema auxiliar) que cuando la función anterior obtiene una valoración, ésta falsifica a la fórmula (que es un BDD):

```
(prove-lemma falsifica-norm-falsifica
 (rewrite)
 (implies (and
           (expresion-si-normalizada x)
           (si-pura x)
           (falsifica-normalizada x val))
   (equal
    (valor x (falsifica-normalizada x val))
    f)))
```

Es de destacar la analogía con la función `es-tautologia-normalizada`. Esta analogía la aprovecha NQTHM en la prueba del siguiente lema:

```
(prove-lemma relacion-taut-fals
  (rewrite)
  (implies
     (and
      (expresion-si-normalizada x)
      (not (es-tautologia-normalizada x val))
      val)
    (falsifica-normalizada x val)))
```

Es decir, si `x` es un BDD y el comprobador de tautologías devuelve falso, entonces `falsifica-normalizada` devuelve una valoración (que por el lema anterior a éste, sabemos que falsifica a la fórmula).

Ya podemos definir `falsifica`:

```
(defn falsifica (x)
  (falsifica-normalizada
     (normaliza
        (transforma-en-si x)) nil))
```

Por último, un lema que agrupe los resultados anteriores:

```
(prove-lemma lema-puente-2 (rewrite)
 (implies
  (and
    (falsifica-normalizada y val)
    (si-pura y)
    (expresion-si-normalizada y)
    (equal
       (valor x
         (falsifica-normalizada y val))
       (valor y
         (falsifica-normalizada y val))))
  (equal
    (valor x
      (falsifica-normalizada y val)) F)))
```

Con la ayuda de este último lema, el sistema prueba directamente el anterior teorema de completitud.

4 Conclusiones

Hemos presentado en esta comunicación la definición de un pequeño sistema de demostración de tautologías proposicionales, verificado por un demostrador automático, y lo hemos propuesto como ejemplo del uso del Razonamiento Automático en la enseñanza de la Inteligencia Artificial.

Varias son las ventajas de esta aproximación. En primer lugar cabe destacar el aspecto práctico. Los algoritmos se definen en un entorno en el que se pueden ejecutar estas definiciones. Además, este mismo entorno es a su vez un sistema de razonamiento automático, donde el alumno puede probar propiedades de las definiciones que ha implementado y observar cómo las técnicas que está aprendiendo funcionan en un sistema real.

En la mayoría de los casos, la formulación de propiedades en la lógica de estos sistemas exige una mayor formalización y rigor, lo que conduce a una comprensión más profunda de los procesos involucrados. En el caso de NQTHM, las pruebas que el sistema encuentra son, además, explicadas con todo detalle en lenguaje natural.

Otro aspecto a destacar es el papel del usuario en la búsqueda de una prueba de algún resultado no trivial: es necesario buscar una serie de lemas previos que aumentan el conocimiento que se tiene de los procesos verificados. Nuevamente, esto redunda en beneficio de una mejor comprensión de la materia estudiada.

El principal inconveniente al verificar mecánicamente teoremas y propiedades no triviales es que exige un trabajo considerablemente mayor que en pruebas hechas a mano. Un motivo para que ocurra esto es el tener que demostrar una serie de lemas básicos que uno asume conocidos en las pruebas a mano. Esto puede solventarse con el uso de librerías de propiedades ya demostradas sobre diversos objetos de uso común en computación.

Una de las líneas actuales de investigación en el Departamento de Ciencias de la Computación de la Universidad de Sevilla es la verificación automática de sistemas de razonamiento, construyendo así una librería de resultados útiles para la verificación de procesos y técnicas usuales en el campo del Razonamiento Automático.