# Inserción y fusión con listas ordenadas: un ejemplo de razonamiento automático con ACL2

F. Palomo Lozano[†], J. A. Alonso Jiménez[*], I. Medina Bulo[†]
{francisco.palomo, inmaculada.medina}@uca.es[†]
jalonso@cica.es[*]

Departamento de Ciencias de la Computación e Inteligencia Artificial[*]
Universidad de Sevilla
Departamento de Lenguajes y Sistemas Informáticos[†]
Universidad de Cádiz

**Palabras clave:** Lógica Computacional, métodos formales, razonamiento automático, lenguaje de programación aplicativo, ACL2, NQTHM

**Resumen**

En este artículo presentamos un ejercicio de formalización y demostración automática en ACL2, un sistema de razonamiento automático que proviene de NQTHM, el demostrador de teoremas de *Boyer* y *Moore*. Partiendo de la definición de dos operaciones esenciales sobre listas ordenadas, discutimos aspectos de formalización, descubrimiento de propiedades, ejecutabilidad y mejora de la eficiencia computacional.

## 1 Introducción

Ya en 1949, se hizo patente la dificultad de crear programas que se ejecutaran según lo especificado. Fue el propio *Turing* quien, siguiendo las ideas de *von Neumann* y *Goldstine*, planteó la necesidad de afrontar formalmente los problemas de corrección parcial y terminación. Herederos de estas ideas, *McCarthy*, *Dijkstra*, *Floyd* y *Hoare* las desarrollaron durante tres décadas.

El amplio desarrollo de los métodos formales ha dado lugar a una nueva disciplina denominada CAR (*Computer-Aided Reasoning*), que tiene entre sus objetivos el desarrollo de herramientas informáticas que proporcionen asistencia en su empleo. Estas herramientas suelen denominarse genéricamente «sistemas de razonamiento automático» e incluso «demostradores automáticos de teoremas».

Dentro del amplio espectro de sistemas de razonamiento automático, existen varios especialmente adecuados para la especificación y verificación formal de sistemas informáticos. Uno de ellos, ACL2, merece especial atención ya que formaliza un subconjunto de un lenguaje de programación real y con él se han obtenido grandes éxitos en la verificación formal automatizada tanto de sistemas de *hardware* como de *software*, además de en la demostración por computador de conocidos teoremas de la Lógica y de las Matemáticas.

[*]Facultad de Informática y Estadística. Avda. Reina Mercedes, s/n. 41012 Sevilla. España.
[†]Escuela Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz. España.

Se expondrá a continuación una breve descripción de ACL2, de ningún modo exhaustiva, atendiendo a los diferentes puntos de vista desde los que puede observarse. Posteriormente, se propondrá un ejemplo sobre cómo razonar formalmente acerca de las propiedades de los programas en ACL2; éste incluirá un caso de descubrimiento automático. Los ficheros ACL2 con el ejemplo completo, algo ampliado, y la demostración producida por el sistema (unas 50 páginas) están disponibles públicamente en `www-cs.us.es/~fpalomo/listas-ordenadas.html`.

# 2 Las tres visiones de ACL2

ACL2 (*A Computational Logic for Applicative Common Lisp*) [6, 7] es el sucesor de NQTHM, el demostrador de teoremas de *Boyer* y *Moore* [1, 3]. ACL2 ha sido desarrollado durante la última década, principalmente por *Moore* y *Kaufmann*, y continúa en evolución. Intenta hacer realidad, con bastante éxito, el deseo de *McCarthy* de disponer de un sistema capaz de demostrar teoremas sobre programas escritos en LISP (puro).

Puede encontrarse una descripción concisa de ACL2 en [4]. En realidad, para definir qué es ACL2 hay que enfocarlo desde tres puntos de vista distintos:

1. Desde un punto de vista lógico.
2. Desde la perspectiva de los lenguajes de programación.
3. Desde el enfoque de los sistemas de razonamiento automático.

## 2.1 ACL2 es una Lógica Computacional

ACL2 es una lógica de primer orden sin cuantificadores y con igualdad[1]. Su *sintaxis* coincide con la del lenguaje de programación COMMON LISP. Esto quiere decir que un término de la lógica es una constante, un símbolo de variable o la aplicación de un símbolo de función (o de una $\lambda$-expresión) $n$-ario a $n$ términos. Formalmente, en ACL2 no existen símbolos de predicado, aunque a las funciones booleanas se las denomine informalmente «predicados».

Entre sus *axiomas* se incluyen los de la lógica proposicional con igualdad y los necesarios para trabajar cómodamente con los tipos de datos de uso más común: números (enteros, racionales y complejos racionales)[2], caracteres, cadenas de caracteres, símbolos y listas.

Las *reglas de inferencia* son las del cálculo proposicional con igualdad junto con instanciación de variables, inducción e instanciación funcional. Todas permiten deducir nuevos teoremas a partir de otros ya conocidos[3]. La *regla de instanciación de variables* permite sustituir variables en un teorema por términos para obtener otro teorema. La *regla de inducción* permite reducir la demostración de un teorema a un conjunto finito de casos empleando una forma muy potente de inducción matemática sobre los $\varepsilon_0$-ordinales. La *regla de instanciación funcional* permite sustituir funciones en un teorema por otras para obtener un nuevo teorema, siempre que las antiguas fueran introducidas por el principio de encapsulado que se explica a continuación y las nuevas cumplan sus axiomas de restricción.

La lógica incluye también dos *principios de extensión*: definición y encapsulado. Ambos permiten añadir nuevos símbolos de función y axiomas a la lógica preservando su consistencia. El *principio de definición* es esencial, ya que permite introducir nuevos símbolos de función y asociarles (axiomáticamente) una definición; para preservar la consistencia, el sistema únicamente

[1] O, recíprocamente, una lógica proposicional con igualdad extendida con símbolos de variable y función.

[2] Una reciente extensión de ACL2, denominada ACL2 (r), amplía la axiomatización para incluir a los reales.

[3] Por supuesto que, inicialmente, los únicos teoremas conocidos son los axiomas.

admite una función bajo este principio si puede demostrar su terminación bajo ciertas condiciones. El *principio de encapsulado* permite introducir nuevos símbolos de función restringidos por axiomas a tener ciertas propiedades; para preservar la consistencia, el sistema exige que se proporcionen «testigos» de la existencia de tales funciones. La instanciación funcional es una regla de inferencia derivada de él. El papel que juega el principio de encapsulado en el desarrollo estructurado de teorías se explica en [5].

La ausencia de cuantificación hace que el carácter de la lógica sea fundamentalmente constructivo: más que enunciar el hecho de que un cierto objeto existe, se ha de exponer una función computable que construya un objeto con las propiedades deseadas.

Otro aspecto interesante es la ausencia de tipos[4] y el carácter total de la lógica. Las funciones introducidas bajo el principio de definición han de ser funciones recursivas totales. Aunque esto último constituya una limitación teórica, en la práctica, la mayoría de las funciones parciales que solemos definir pueden ser generalizadas convenientemente para obtener funciones totales equivalentes bajo sus dominios de definición. Posteriormente un mecanismo extra-lógico basado en *protecciones* permite declarar el dominio que se pretende que dichas funciones tengan en ejecución.

## 2.2 ACL2 es un lenguaje de programación aplicativo

Toda función admitida bajo el principio de definición de ACL2 es una función de COMMON LISP. El recíproco no es cierto, ya que para razonar en la lógica ACL2 sobre una función su ejecución debe sólo depender de sus parámetros reales[5]: a iguales parámetros, igual resultado. Por otro lado, los lenguajes de programación convencionales permiten «definir» funciones cuya terminación no está garantizada.

Esto hace que sea posible pensar en ACL2 como en un *lenguaje de programación aplicativo*. Más explícitamente, puede considerarse que es un subconjunto de COMMON LISP libre de efectos colaterales.

## 2.3 ACL2 es un sistema de razonamiento automático

Cuando se suministra un posible teorema a ACL2, o se extiende la lógica mediante uno de los principios de extensión siendo necesario comprobar que se cumplen ciertas condiciones, ACL2 se comporta como un demostrador de teoremas.

Como tal, emplea una serie de *técnicas de prueba* en su búsqueda de una demostración del supuesto teorema. Cada técnica de prueba puede verse como un proceso que recibe una fórmula como entrada y produce un conjunto de fórmulas como resultado: la fórmula de entrada es un teorema *si* cada una de las resultantes lo es.

Por supuesto, puede que un proceso no sea aplicable a una fórmula. En tal caso, el proceso no falla, sino que produce un conjunto que contiene únicamente a dicha fórmula. Por otro lado, si un proceso es capaz de demostrar que una determinada fórmula es un teorema, devuelve el conjunto vacío.

Cuando el usuario proporciona una conjetura al sistema, su fórmula se convierte en el objetivo de la demostración y pasa secuencialmente por cada uno de los procesos hasta que uno de ellos sea aplicable. Éste produce un conjunto de subobjetivos que reemplazan al objetivo original y todo comienza de nuevo. Esto continúa hasta que no quedan subobjetivos pendientes de demostrar o hasta que se cumplen determinadas condiciones de terminación que indican que el sistema

[4] Sin embargo, un primitivo sistema de deducción de tipos está presente internamente en el sistema.

[5] De lo contrario, su definición no bastaría: sería necesario introducir el concepto de «estado global».

no puede completar la demostración de la conjetura (incluso, a veces, aparece un ciclo); en este último caso la conjetura inicial puede o no ser un teorema. Como se observa, el razonamiento es regresivo. Describamos, brevemente, cada proceso.

El proceso de *simplificación* incluye procedimientos de decisión para la lógica proposicional, la igualdad y la *aritmética lineal*. También se encarga de la reescritura de términos y de las meta-funciones [2], que son simplificadores que el usuario define y demuestra su corrección en el sistema. Es el único proceso que puede devolver un conjunto vacío de fórmulas demostrando así que su fórmula de entrada es un teorema. El sistema de reescritura de términos juega un papel fundamental: los axiomas, definiciones y teoremas se almacenan como reglas de reescritura.

El proceso de *eliminación de destructores* permite sustituir variables afectadas por operaciones destructoras por un término formado por una operación constructora y nuevas variables. Con esto se pueden eliminar operaciones destructoras para obtener fórmulas más sencillas.

Los tres procesos siguientes son la *fertilización cruzada*, que controla heurísticamente cuándo se deben utilizar y descartar hipótesis de igualdad, la *generalización*, que permite reemplazar términos que no son variables por nuevas variables y la *eliminación de irrelevancia*, que intenta descartar aquellas hipótesis de una conjetura que no influyen en su veracidad. Estos tres procesos son potencialmente peligrosos, ya que al descartar una hipótesis o generalizar un término se obtiene una conjetura más general que puede no ser un teorema, aunque, si lo es, la original también lo será. Su cometido es preparar la fórmula para una posible inducción ya que, a veces, es necesario generalizar una fórmula para poder demostrarla por inducción.

Por último, el proceso de *inducción* intenta encontrar un esquema de inducción apropiado para demostrar la conjetura por inducción. Para ello, se centra en aquellos términos de la conjetura correspondientes a funciones recursivas donde aparecen variables ocupando posiciones de parámetros formales que decrecen en la recursión. Éstos sugieren distintos esquemas de inducción; heurísticamente el sistema decide cuáles no son adecuados y de entre los restantes selecciona un único candidato (quizás una fusión de varios). Si ninguno es adecuado el proceso falla y la conjetura no puede ser demostrada.

# 3 Operaciones en listas ordenadas

## 3.1 Definiciones y su admisión

La siguiente función de ACL2 permite comprobar si un objeto es una lista (propia) ordenada de números racionales.

```
(defun ordenadap (l)
  (cond ((atom l)
         (null l))
        ((atom (rest l))
         (and (rationalp (first l)) (null (rest l))))
        (t
         (and (rationalp (first l)) (rationalp (first (rest l)))
              (<= (first l) (first (rest l))) (ordenadap (rest l))))))
```

Las siguientes funciones permiten, respectivamente, insertar un elemento en la posición adecuada de una lista ordenada y fundir dos listas ordenadas.

```
(defun insercion (e l)
  (declare (xargs :guard (and (rationalp e) (ordenadap l))))
  (cond ((endp l)
         (list e))
        ((<= e (first l))
         (cons e l))
        (t
         (cons (first l) (insercion e (rest l))))))
```

```
(defun fusion (l1 l2)
  (declare (xargs :guard (and (ordenadap l1) (ordenadap l2))))
  (cond ((endp l1)
         l2)
        (t
         (insercion (first l1) (fusion (rest l1) l2)))))
```

Nótese que `fusion` es bastante ineficiente, ya que el número de comparaciones que realiza pertenece a $O(n^2)$, siendo $n$ la longitud de `l1`. No obstante, esto carece de importancia cuando el interés inmediato radica en demostrar propiedades de la función más que en su uso computacional. Existe un compromiso entre facilidad de demostración y eficiencia computacional. Dados dos algoritmos equivalentes, a menudo es más difícil demostrar propiedades generales sobre el más eficiente; esto es así porque en él suelen subyacer las ideas más ingeniosas. Volveremos sobre este punto más adelante.

Estas tres funciones son admisibles bajo el principio de definición empleando el mismo argumento: todas poseen una única rama recursiva que reduce la longitud de su primer parámetro. Por ejemplo, al intentar definir `ordenadap` el demostrador responde:

```
The admission of ORDENADAP is trivial, using the relation E0-ORD-<
(which is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT L).  We observe that the type of ORDENADAP is
described by the theorem (OR (EQUAL (ORDENADAP L) T) (EQUAL (ORDENADAP L) NIL)).
We used primitive type reasoning.
```

La función booleana `e0-ord-<` representa el orden estándar entre los $\varepsilon_0$-ordinales[6] (que son reconocidos por `e0-ordinalp`) y `acl2-count` es una norma sobre los objetos de ACL2. Cuando la longitud de una lista decrece, su `acl2-count` también. El sistema conoce (como meta-teorema) que `e0-ord-<` está bien fundada o, equivalentemente, que es *noetheriana*. Nótese cómo ACL2 deduce que el resultado de la función ha de ser `t` o `nil` y recuerda este hecho como información sobre el tipo de la función, aunque formalmente la lógica no tenga tipos.

## 3.2 Propiedades y su demostración automática

Podemos demostrar sin dificultad que las funciones de inserción y fusión producen listas ordenadas (es sólo una parte de su especificación).

```
(defthm insercion-ordenada
  (implies (and (rationalp e) (ordenadap l))
           (ordenadap (insercion e l)))
  :rule-classes :type-prescription)

(defthm fusion-ordenada
  (implies (and (ordenadap l1) (ordenadap l2))
           (ordenadap (fusion l1 l2)))
  :rule-classes :type-prescription)
```

Una propiedad interesante (y útil) consiste en reconocer que el orden de las inserciones en una lista ordenada es irrelevante:

```
(defthm insercion-insercion
  (implies (and (rationalp e1) (rationalp e2) (ordenadap l))
           (equal (insercion e1 (insercion e2 l))
                  (insercion e2 (insercion e1 l)))))
```

Con estas propiedades ya es posible para el sistema encontrar automáticamente una demostración de una propiedad bastante más compleja: la fusión de listas ordenadas es asociativa.

[6] Éste coincide con el orden habitual $\langle \mathbb{N}, < \rangle$ cuando los ordinales implicados son inferiores a $\omega$.

```
(defthm fusion-asociativa
  (implies (and (ordenadap l1) (ordenadap l2) (ordenadap l3))
           (equal (fusion (fusion l1 l2) l3) (fusion l1 (fusion l2 l3)))))
```

Si intentamos que demuestre la conmutatividad directamente, fallará.

```
(defthm fusion-conmutativa
  (implies (and (ordenadap l1) (ordenadap l2))
           (equal (fusion l1 l2) (fusion l2 l1)))))
```

Al inspeccionar la prueba fallida se observa la utilidad del siguiente teorema. Operacionalmente, su regla de reescritura asociada permite «empujar» una inserción dentro del segundo parámetro de una fusión subsecuente en el contexto de una fórmula que contenga sus hipótesis.

```
(defthm insercion-fusion-2
  (implies (and (rationalp e) (ordenadap l1) (ordenadap l2))
           (equal (insercion e (fusion l1 l2))
                  (fusion l1 (insercion e l2)))))
```

Tras demostrarlo automáticamente, ACL2 completa con éxito la prueba de la conmutatividad. Un hecho realmente notable es que el demostrador es capaz de *descubrir* propiedades interesantes *por sí mismo*. En efecto, basta explorar la demostración que produce de `fusion-conmutativa` para descubrir entre sus subobjetivos el siguiente:

```
Subgoal *1/1'''
(IMPLIES (ORDENADAP L2)
         (EQUAL L2 (FUSION L2 NIL))).
```

No obstante, carece de inteligencia para discernir cuándo una propiedad es importante y merece un nombre (por ser potencialmente reutilizable) y cuándo no. En este caso, reconocida por el usuario la importancia de esta propiedad, que expresa el hecho de que la lista vacía es neutro por la derecha de la operación de fusión, se le suministra al sistema para que la demuestre:

```
(defthm fusion-neutro
  (implies (ordenadap l)
           (equal (fusion l nil) l)))
```

ACL2 responde lo siguiente:

```
But simplification reduces this to T, using the :type-prescription
rule ORDENADAP, the :executable-counterparts of CONSP and ORDENADAP,
the :definition FUSION, the :rewrite rule FUSION-CONMUTATIVA and primitive
type reasoning.

Q.E.D.
```

Esto es, ha encontrado una demostración realmente corta: aplica la conmutatividad de la fusión, con lo que el problema se reduce a demostrar que `nil` es neutro por la izquierda, y expande la definición de fusión, que hace patente su veracidad.

## 3.3 Verificación de protecciones

La cláusula `declare` que aparece en las definiciones de `insercion` y `fusion`, indica una *protección* a la ejecución de la función. ACL2 nos permite verificar que una función que reciba valores en el dominio de su protección no atentará contra la protección de ninguna de las funciones que emplea, pero no utiliza este resultado a la hora de razonar sobre ella.

Una función de ACL2 que tenga su protección verificada se ejecutará, sin otros problemas que los derivados de la finitud de los recursos computacionales, en cualquier sistema COMMON LISP, siempre que se aplique a valores que se ajusten a su protección. Además, cualquier intento de ejecutar una función tal dentro de ACL2 será abortado si se aplica a valores fuera del dominio de su protección.

# 4 Razonamiento por composición

Puesto que la fusión propuesta en 3.1 distaba de ser eficiente, proponemos una mejora evidente que reduce su complejidad temporal a $\Theta(n)$ operaciones de comparación.

```
(defun fusion* (l1 l2)
  (declare (xargs :guard (and (ordenadap l1) (ordenadap l2))
                  :measure (+ (len l1) (len l2))))
  (cond ((endp l1)
         l2)
        ((endp l2)
         l1)
        ((<= (first l1) (first l2))
         (insercion (first l1) (fusion* (rest l1) l2)))
        (t
         (insercion (first l2) (fusion* l1 (rest l2))))))
```

Para que esta función sea admitida ha sido necesario suministrar una medida con la que demostrar el decrecimiento estricto en cada llamada recursiva, ya que su parada no es obvia para ACL2. Basta proponer la suma de ambas longitudes para obtener una medida ordinal que decrece estrictamente en cada llamada recursiva. El demostrador genera y demuestra el siguiente teorema de medida, tras lo que admite la función:

```
For the admission of FUSION* we will use the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (+ (LEN L1) (LEN L2)).  The non-trivial part of the
measure conjecture is

Goal
(AND (E0-ORDINALP (+ (LEN L1) (LEN L2)))
     (IMPLIES (AND (NOT (ENDP L1))
                   (NOT (ENDP L2))
                   (< (CAR L2) (CAR L1)))
              (E0-ORD-< (+ (LEN L1) (LEN (CDR L2)))
                        (+ (LEN L1) (LEN L2))))
     (IMPLIES (AND (NOT (ENDP L1))
                   (NOT (ENDP L2))
                   (<= (CAR L1) (CAR L2)))
              (E0-ORD-< (+ (LEN (CDR L1)) (LEN L2))
                        (+ (LEN L1) (LEN L2))))).
```

El problema está en cómo demostrar que esta función tiene las mismas propiedades que la anterior. Una demostración directa de las propiedades es complicada por lo que resulta más adecuado razonar «por composición», reduciendo el problema a demostrar que ambas funciones hacen *exactamente* lo mismo, es decir, que son equivalentes.

```
(defthm fusion*<->fusion
  (implies (and (ordenadap l1) (ordenadap l2))
           (equal (fusion* l1 l2) (fusion l1 l2)))
  :hints (("Goal" :induct (fusion* l1 l2))))
```

La demostración de este teorema de equivalencia no es fácil y ACL2 no escoge automáticamente el esquema de inducción más apropiado, por lo que se le sugiere que emplee un esquema con la estructura de la función `fusion*`. Con esto se consigue que, al introducir la conjetura, la demostración comience de la siguiente forma:

```
Name the formula above *1.

We have been told to use induction.  One induction scheme is suggested
by the induction hint.

We will induct according to a scheme suggested by (FUSION* L1 L2).
If we let  (:P L1 L2) denote *1 above then the induction scheme we'll
```

```
use is
(AND (IMPLIES (AND (NOT (ENDP L1))
                   (NOT (ENDP L2))
                   (< (CAR L2) (CAR L1))
                   (:P L1 (CDR L2)))
              (:P L1 L2))
     (IMPLIES (AND (NOT (ENDP L1))
                   (NOT (ENDP L2))
                   (<= (CAR L1) (CAR L2))
                   (:P (CDR L1) L2))
              (:P L1 L2))
     (IMPLIES (AND (NOT (ENDP L1)) (ENDP L2))
              (:P L1 L2))
     (IMPLIES (ENDP L1) (:P L1 L2))).
```

Una vez demostrado el teorema, la regla de reescritura que genera es capaz de transformar (`fusion*` $x$ $y$) en (`fusion` $x$ $y$) si (`ordenadap` $x$) y (`ordenadap` $y$) aparecen disponibles entre las hipótesis. Así, por ejemplo, demostrar que `fusion*` es conmutativa (para listas ordenadas) se reduce al hecho, previamente demostrado, de que `fusion` lo es.

# 5 Conclusiones y trabajo futuro

Este sencillo ejemplo (básicamente, se ha demostrado que el conjunto de las listas ordenadas de números racionales es un monoide conmutativo respecto de la operación de fusión) muestra cómo ACL2, una lógica en apariencia no muy potente, es adecuada para razonar sobre las propiedades de programas escritos en un lenguaje de programación real.

Es de nuestro interés estudiar bajo qué condiciones estas operaciones pueden extenderse para representar algoritmos genéricos que trabajen con elementos arbitrarios (y no sólo con números racionales) de manera que los esquemas de las demostraciones varíen lo menos posible de los encontrados para el caso particular presentado aquí.

# Referencias


[1] *Boyer, R. S.* & *Moore, J S. A Computational Logic.* Academic Press. 1978.

[2] *Boyer, R. S.* & *Moore, J S. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.* En *The Correctness Problem in Computer Science* (eds: *R. S. Boyer* y *J S. Moore*). Academic Press. 1981.

[3] *Boyer, R. S.* & *Moore, J S. A Computational Logic Handbook.* Academic Press. 2ª ed. 1998.

[4] *Kaufmann, M.* & *Moore, J S. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp.* IEEE Transactions on Software Engineering. 1997.

[5] *Kaufmann, M.* & *Moore, J S. Structured Theory Development for a Mechanized Logic.* Enviado para su publicación. 1999.

[6] *Kaufmann, M.*; *Manolios, P.* & *Moore, J S. Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers. 2000.

[7] *Kaufmann, M.*; *Manolios, P.* & *Moore, J S.* (eds.) *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers. 2000.