Razonamiento automático en programación genérica con ACL2: estudio de un caso

I. Medina Bulo[†], J. A. Alonso Jiménez[*], F. Palomo Lozano[†]
{inmaculada.medina, francisco.palomo}@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, programación genérica, algoritmos genéricos, ACL2

Resumen

En este artículo se estudia un caso de formalización y demostración automática en ACL2 relacionado con la programación genérica. Tras una definición axiomática del concepto de orden estricto débil se demuestran sus propiedades esenciales para, a continuación, formalizar las listas ordenadas genéricas junto a sus operaciones de inserción y fusión. Para terminar, se realiza una instanciación de las operaciones genéricas y de sus propiedades.

1 Introducción

El desarrollo de los métodos formales en los últimos treinta años 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».

Dentro de los sistemas de razonamiento automático adecuados para la especificación y verificación formal de sistemas informáticos, uno de ellos, ACL2 (*A Computational Logic for Applicative Common Lisp*), merece especial atención ya que formaliza un subconjunto de un lenguaje de programación real, COMMON LISP. La mayor parte de las ideas presentes en ACL2 [5, 6] provienen de NQTHM, el demostrador de teoremas de *Boyer* y *Moore* [1, 2].

Por otro lado, el paradigma de la *programación genérica* [8] ha madurado a partir de lenguajes como TECTON [7] e incluso se ha mezclado con otros paradigmas, como en C++ que incluso posee una biblioteca estándar, la STL (*Standard Template Library*), de contenedores y algoritmos genéricos. La programación genérica requiere, si cabe, mayor formalización que la convencional ya que un *algoritmo genérico* no es más que una abstracción de muchos otros que resulta de generalizar tipos y operaciones concretas.

[*]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.

Debido a esto, parece necesario para razonar sobre programas genéricos un formalismo lógico muy potente que incluya el concepto de tipo y que sea de orden superior. No obstante, mostraremos cómo es posible salvar estas vicisitudes con una lógica en apariencia más débil como es el caso de ACL2: una lógica de primer orden sin cuantificadores y con igualdad, que contiene únicamente dos *principios de extensión* y que carece de tipos.

Estos principios de extensión 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 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. Este segundo principio juega un importante papel en el desarrollo estructurado de teorías, como se explica en [4].

Se presenta a continuación el estudio de un caso de razonamiento formal sobre programas genéricos en ACL2. En él aparecen (entre otras cosas) diversas funciones definidas de manera análoga a como se hace en COMMON LISP, ya que ACL2 es también un lenguaje de programación aplicativo derivado de él. Puede encontrarse una descripción concisa de ACL2 en [3]. Los ficheros con el ejemplo completo, algo ampliado, y la demostración producida por el sistema están disponibles públicamente en `www-cs.us.es/~fpalomo/listas-ordenadas.html`.

2 Operaciones sobre listas ordenadas genéricas

El ejemplo escogido es una generalización del descrito en [10]. En éste se formalizaba en ACL2 el conjunto de las listas ordenadas de números racionales y dos operaciones sobre dicho conjunto, `insercion` y `fusion`. En este trabajo se elimina el requisito de que los elementos de la lista tengan que ser de tipo «número racional». En realidad, se aprovecha la ausencia de tipos en ACL2 para relajar aún más la definición permitiendo incluso que, potencialmente, las listas no sean homogéneas.

2.1 El problema del orden

Puesto que ahora las operaciones han de ser abstractas, por ser las listas de tipo arbitrario, y no disponemos de tipos, parece que la única solución es introducir una *función de comparación* como parámetro de las operaciones de inserción y fusión, dando lugar así a una lógica de orden superior.

Una solución en ACL2 consiste en utilizar el segundo principio de extensión para introducir un nuevo símbolo de función restringido por axiomas oportunos a cumplir las propiedades que se esperan de una función de comparación. Pero, cabe preguntarse cuáles deben ser dichas propiedades, ya que existen diversas nociones formales que permiten abstraer nuestro concepto informal de «orden». ¿Es más adecuado emplear un orden total que uno parcial? ¿Basta con un preorden? ¿Qué conviene entender por elementos equivalentes?

Tomemos como ejemplo la función booleana `<` de ACL2. Sobre el conjunto de los números es un orden estricto total y, sin embargo, no lo es sobre el de los objetos de ACL2. En efecto, un intento de demostrar la siguiente propiedad fracasará:

```
(defthm tricotomica
  (or (< a b) (< b a) (= a b))
  :rule-classes nil)
```

Para comprender el porqué, basta inspeccionar parte de la salida que produce ACL2 en respuesta a la anterior conjetura (aparecen subrayados los puntos clave):

```
 ...
This simplifies, using trivial observations, to

Goal''
(IMPLIES (AND (<= B A) (<= A B))
         (EQUAL A B)).

This forcibly simplifies, using linear arithmetic, to

Goal'''
(IMPLIES (<= A A) (EQUAL A A)).

But simplification reduces this to T, using primitive type reasoning.

q.e.d. (given two forced hypotheses)

Modulo the following two forced goals, that completes the proof of
the input Goal.  See :DOC forcing-round.

[1]Subgoal 2, below, will focus on
(ACL2-NUMBERP A),
 ...

[1]Subgoal 1, below, will focus on
(ACL2-NUMBERP B),
 ...
```

Esto significa que puede demostrarlo, siempre que `a` y `b` sean números, pero no en general. De hecho, se puede comprobar que no se cumple si tomamos `nil` y `t` como valores de `a` y `b`. Aunque pueda parecer irrelevante (ciertamente, la función `<` se emplea para comparar números) no hay que olvidar que ACL2 es una lógica sin tipos de funciones totales y que es importante poder razonar sobre éstas con las mínimas restricciones.

No es difícil ver que el concepto de orden total es demasiado restrictivo para muchas aplicaciones prácticas. Seguiremos así el criterio empleado en la biblioteca STL de C++, donde se incluye entre las precondiciones de los algoritmos genéricos que emplean una función (u operador) de comparación que ésta tenga estructura de *orden estricto débil*. La idea de emplear esta noción de orden en algoritmos genéricos aparece ya en TECTON y de ahí pasa a la STL [9]. Esta elección se ha demostrado adecuada para aplicaciones industriales, como refleja el hecho de que esta biblioteca fuera elegida para formar parte del estándar ANSI/ISO de C++.

2.2 Formalización de los conceptos de orden y equivalencia

Recuérdese que un *orden estricto* (parcial) es un conjunto A con una relación[1] binaria r que cumple las siguientes propiedades:

$$\forall a \in A : \neg r(a,a) \qquad \text{(antirreflexiva)}$$

$$\forall a,b,c \in A : r(a,b) \wedge r(b,c) \implies r(a,c) \qquad \text{(transitiva)}$$

de donde se deduce esta otra:

$$\forall a,b \in A : r(a,b) \implies \neg r(b,a) \qquad \text{(antisimétrica)}$$

Un *orden estricto débil* (en adelante, OED) es un orden estricto que cumple además la siguiente propiedad, que expresa el hecho de que la «incomparabilidad» ha de ser transitiva:

$$\forall a,b,c \in A : \neg r(a,b) \wedge \neg r(b,a) \wedge \neg r(b,c) \wedge \neg r(c,b) \implies \neg r(a,c) \wedge \neg r(c,a)$$

[1] Representaremos a las relaciones mediante funciones booleanas.

Dado un OED $\langle A, r\rangle$, éste induce la siguiente relación binaria:

$$\forall a, b \in A : e_r(a, b) = \neg r(a, b) \wedge \neg r(b, a)$$

que se puede demostrar que es una relación de equivalencia (es decir, reflexiva, simétrica y transitiva), recibiendo por ello el nombre de *equivalencia inducida* por el OED. Nótese que dos elementos son equivalentes si, y sólo si, son incomparables. En función de esta nueva relación, el tercer axioma de los OED queda:

$$\forall a, b, c \in A : e_r(a, b) \wedge e_r(b, c) \implies e_r(a, c) \qquad \text{(transitiva de la equivalencia)}$$

Por ejemplo, el conjunto de las cadenas de caracteres con la relación «tener menor longitud que» es un OED, estando sus clases de equivalencia formadas por cadenas de igual longitud. Otro ejemplo de OED es el conjunto de los objetos de ACL2 con la función booleana `ACL2::<`, aunque sus clases de equivalencia tienen una estructura más compleja:

```
(defthm caracterizacion-clases-equivalencia-<
  (iff (and (not (< a b)) (not (< b a)))
       (or (equal a b)
           (and (or (not (acl2-numberp a)) (equal a 0))
                (or (not (acl2-numberp b)) (equal b 0)))))
  :rule-classes nil)
```

A continuación realizaremos una formalización general de los OED en ACL2. Por conveniencia notacional empleamos los símbolos de función `<` e `=` para representar la relación del OED y su equivalencia inducida; también introducimos la siguiente macro:

```
(defmacro <= (a b)
  `(not (< ,b ,a)))
```

Para que no haya conflicto entre estos símbolos y los ya disponibles en ACL2 los definimos en un paquete de nombre `OED`. Una vez hecho esto, extendemos la lógica mediante encapsulado añadiendo únicamente un símbolo de función binario `<` restringido por los tres axiomas de los OED. Como testigo de la existencia de tal función empleamos `ACL2::<`, es decir, el orden estricto ya definido[2] sobre los objetos del lenguaje.

```
(encapsulate
  ((< (a b) t))

  (local
    (defun < (a b)
      (LISP::< a b)))

  (defthm antirreflexiva
    (not (< a a)))

  (defthm transitiva
    (implies (and (< a b) (< b c))
             (< a c)))

  (defthm incomparabilidad-transitiva
    (implies (and (not (< a b)) (not (< b a)) (not (< b c)) (not (< c b)))
             (and (not (< a c)) (not (< c a))))))
```

Ahora es posible introducir la equivalencia inducida por el OED extendiendo la lógica mediante el principio de definición para que incluya el símbolo de función binaria `=`. ACL2 demuestra que es realmente una equivalencia sin dificultad.

[2] Nótese que esto demuestra precisamente nuestra afirmación previa de que `ACL2::<` es un ejemplo de OED.

```
(defun = (a b)
  (and (not (< a b)) (not (< b a))))

(defequiv =)
```

La propiedad de antisimetría se deduce de la antirreflexividad y de la transitividad, aunque, en este caso, ACL2 requiere ayuda sobre la forma de emplear la transitividad.

```
(defthm antisimetrica
  (implies (< a b)
           (not (< b a)))
  :hints (("Goal" :use (:instance transitiva (c a)))))
```

También se cumple una propiedad de tricotomía respecto a la equivalencia inducida. El corolario expuesto es apropiado para su empleo como regla de reescritura, cosa que no ocurre con la formulación original del teorema.

```
(defthm tricotomica
  (or (< a b) (< b a) (= a b))
  :rule-classes
  ((:rewrite :corollary
             (implies (not (or (< a b) (= a b)))
                      (< b a)))))
```

A continuación se presentan tres propiedades de transitividad que resultan muy útiles. La primera se demuestra tras una adecuada descomposición por casos; el sistema es capaz de reducir las otras dos a ésta.

```
(defthm <=-<-transitiva
  (implies (and (<= a b) (< b c))
           (< a c))
  :hints (("Goal" :cases ((< a b) (< c a)))))

(defthm <-<=-transitiva
  (implies (and (< a b) (<= b c))
           (< a c)))

(defthm <=-transitiva
  (implies (and (<= a b) (<= b c))
           (<= a c)))
```

Empleándolas es posible demostrar esta importante propiedad de los OED: el orden se preserva bajo la equivalencia inducida.

```
(defcong = iff (< a b) 1)
(defcong = iff (< a b) 2)
```

2.3 Operaciones genéricas, equivalencia inducida y congruencias

Las operaciones genéricas con las que trataremos no son más que abstracciones de las que aparecen en [10]. Simplemente se abstrae el tipo y el orden en cada una de las funciones definidas.

Para abstraer el tipo se elimina el requisito de que los elementos sean números racionales y para abstraer el orden se sustituye `ACL2::<` por `OED::<`. Aunque emplearemos los mismos nombres, no existe confusión, pues se crean en un nuevo paquete de nombre `LOG`. Por ejemplo, la siguiente función permite comprobar si un objeto es una *lista ordenada genérica* (LOG):

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

Exactamente lo mismo ocurre con `insercion`, que permite insertar un elemento en la posición adecuada de una LOG, sólo cambia la relación de orden. La función `fusion` ni siquiera varía sintácticamente, ya que se define a partir de la inserción.

La equivalencia inducida por un OED induce a su vez una equivalencia sobre el conjunto de las listas genéricas. De nuevo, por comodidad notacional, utilizaremos el símbolo = para esta equivalencia. Recuérdese que no hay confusión, este = pertenece al paquete `LOG`.

```
(defun = (l1 l2)
  (cond ((or (endp l1) (endp l2))
         (equal l1 l2))
        (t
         (and (OED::= (first l1) (first l2))
              (= (rest l1) (rest l2))))))

(defequiv =)
```

Como se observa, en realidad, el resultado es más general: se obtiene una equivalencia sobre los objetos de ACL2 (las funciones de la lógica son totales). En particular, de recibir listas, éstas ni siquiera tienen que ser propias. Es interesante cómo ACL2 consigue demostrar automáticamente las siguientes congruencias con las operaciones:

```
(defcong OED::= = (insercion e l) 1)
(defcong = = (insercion e l) 2)

(defcong = = (fusion l1 l2) 1)
(defcong = = (fusion l1 l2) 2)
```

2.4 Propiedades de las operaciones genéricas

Podemos demostrar sin dificultad propiedades de las operaciones abstractas sobre LOG análogas a las demostradas en [10]. Para ello, sustituimos la igualdad sintáctica `equal` por =, y eliminamos las restricciones sobre los tipos de los elementos. Por ejemplo:

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

(defthm insercion-insercion
  (implies (ordenadap l)
           (= (insercion e1 (insercion e2 l))
              (insercion e2 (insercion e1 l)))))
```

Todas las propiedades expuestas en dicho trabajo han sido generalizadas en este sentido y demostradas automáticamente por ACL2. En aras de la brevedad, no las incluiremos. El lector interesado puede consultar los detalles en los ficheros correspondientes.

En general, se han encontrado demostraciones muy similares en ambos casos. Una diferencia fundamental estriba en que ahora se emplean los axiomas y las propiedades demostradas sobre los OED en lugar del procedimiento de decisión para la aritmética lineal, que ya no es aplicable. La otra gran diferencia radica en que la reescritura por congruencia sustituye al procedimiento de decisión para la igualdad en contextos donde `OED::=` y `LOG::=` reemplazan a `equal`.

3 Instanciación de operaciones genéricas y de sus propiedades

Los algoritmos genéricos no son directamente ejecutables, ya que son abstractos. Para obtener instancias ejecutables de estos algoritmos es necesario primero instanciar sus operaciones abstractas. Así, en un paquete de nombre `INS-OED` definimos primero instancias de `OED::<` y `OED::=`. Como ejemplo, representaremos el OED que forman los objetos de `ACL2` con `ACL2::<`.

```
(defun < (a b)
  (ACL2::< a b))

(defun = (a b)
  (and (not (< a b)) (not (< b a))))
```

A continuación, deberíamos demostrar que las instancias cumplen las propiedades de un OED[3], pero, en este caso, no es necesario, ya que el procedimiento de decisión para la aritmética lineal incorporado en ACL2 sirve de sustituto.

Finalmente hay que sustituir el orden abstracto `OED::<` por el concreto `INS-OED::<` en cada función genérica para obtener una función ejecutable. Para seguir con los mismos nombres, creamos un nuevo paquete, `INS-LOG`. Por ejemplo, `insercion` queda:

```
(defun insercion (e l)
  (cond ((endp l)
         (list e))
        ((INS-OED::<= e (first l))
         (cons e l))
        (t
         (cons (first l) (insercion e (rest l))))))
```

Las propiedades relevantes de los algoritmos instanciados pueden demostrarse por instanciación funcional de las propiedades genéricas correspondientes. Por ejemplo:

```
(defthm fusion-conmutativa
  (implies (and (ordenadap l1)
                (ordenadap l2))
           (= (fusion l1 l2) (fusion l2 l1)))
  :hints (("Goal"
           :use (:functional-instance
                 (:instance LOG::fusion-conmutativa (LOG::l1 l1) (LOG::l2 l2))
                 (LOG::ordenadap ordenadap)
                 (OED::< INS-OED::<)
                 (LOG::= =)
                 (OED::= INS-OED::=)
                 (LOG::fusion fusion)
                 (LOG::insercion insercion)))))
```

La instanciación de variables de `LOG::fusion-conmutativa` es un detalle técnico que permite adaptar los nombres de las variables al paquete actual antes de la instanciación funcional.

4 Conclusiones y trabajo futuro

Se ha mostrado cómo una lógica aparentemente débil como ACL2, que es una restricción de la lógica de primer orden con igualdad que carece de cuantificadores, tipos y funciones parciales puede emplearse para razonar sobre programas genéricos. Esto se debe, fundamentalmente, a la potencia del segundo principio de extensión de esta lógica que permite la introducción de símbolos de función restringidos por axiomas.

Este mecanismo (y su regla de inferencia asociada, la instanciación funcional) se asemeja a alguno de los disponibles de manera primitiva en otros formalismos lógicos de orden superior. Esto nos permite «simular», aunque de forma muy tosca, la aparición de funciones como parámetros, que permite considerar a las funciones como objetos de primer orden, algo fundamental en programación genérica.

Una vez puesto de manifiesto el importante papel que juegan los OED en la formalización de algoritmos genéricos que emplean órdenes y equivalencias, se ha presentado una posible formalización en ACL2 que se ha demostrado adecuada para la generalización de un trabajo previo sobre

[3]De hecho, ya se hizo (si bien, con carácter temporal) al presentarla como testigo ante los axiomas de un OED.

listas ordenadas homogéneas de un tipo concreto. El hecho de haber encontrado demostraciones muy similares en ambos trabajos es un indicio de la bondad de la formalización realizada.

Como subproducto de esta formalización se ha obtenido una demostración prácticamente automática de que `ACL2::<` es, precisamente, un OED sobre el conjunto de los objetos de ACL2. Este OED concreto se ha empleado para ilustrar cómo se obtienen instancias ejecutables de los algoritmos genéricos y se demuestran sus propiedades mediante instanciación funcional.

Ya que ACL2 no se diseñó expresamente para dar apoyo al razonamiento automático dentro del paradigma de la programación genérica, faltan herramientas que eviten, por ejemplo, la tarea de instanciar un algoritmo genérico sustituyendo los símbolos que representan a operaciones abstractas por los correspondientes a operaciones concretas. La propia instanciación funcional de las propiedades relevantes es un proceso tedioso.

Del mismo modo, aunque en ACL2 existe explícitamente el concepto abstracto de equivalencia y de congruencia, sólo se incluyen relaciones de orden apropiadas para manipular números (`ACL2::<`) y ε_0-ordinales (`ACL2::e0-ord-<`). Creemos que la formalización de los OED presentada es un buen punto de partida para poder razonar con órdenes abstractos en el sistema, pero somos conscientes de que no puede compararse con un procedimiento de decisión tan potente como el que incorpora ACL2 para la aritmética lineal.

Referencias

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

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

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

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

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

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

[7] *Kapur, D.*; *Musser, D. R.* & *Stepanov, A. A. Tecton: A Language for Manipulating Generic Objects.* LNCS, 134. Springer-Verlag. 1982.

[8] *Musser, D. R.* & *Stepanov, A. A. Generic Programming.* LNCS, 358. Springer-Verlag. 1989.

[9] *Musser, D. R. Tecton Description of STL Container and Iterator Concepts.* Universität Tübingen. 1998.

[10] *Palomo Lozano, F.*; *Alonso Jiménez, J. A.* & *Medina Bulo, I. Inserción y fusión con listas ordenadas: un ejemplo de razonamiento automático con ACL2.* Pendiente de aceptación. IV Jornadas Científicas Andaluzas en Tecnologías de la Información. 2000.