Formalización del razonamiento ecuacional en una lógica computacional

J.L. Ruiz, J.A. Alonso, M.J. Hidalgo y F.J. Martín *
Dpto. Ciencias de la Computación e Inteligencia Artificial
Facultad de Informática y Estadística, Universidad de Sevilla
E–mail: {jruiz,jalonso,mjoseh,fjesus}@cica.es

Noviembre de 2000

Abstract

Presentamos en este artículo el estado actual de nuestro trabajo sobre la formalización en una lógica computacional (la lógica de ACL2), de diversas propiedades de la lógica ecuacional. Como ejemplo principal, presentamos la formalización y demostración mecánica del teorema de pares críticos de Knuth-Bendix.

Palabras clave: Deducción Automática, Sistemas de Reescritura, ACL2

1 Introduction

Para razonar formalmente en Matemáticas, es necesario disponer de una lógica. Ésta viene dada por un conjunto de símbolos (el lenguaje), una serie de propiedades asumidas como verdaderas (axiomas) y unas reglas para derivar nuevas propiedades o teoremas (reglas de inferencia). Cuando esta lógica está diseñada especialmente para formalizar y razonar sobre propiedades de sistemas de computación, la denominamos lógica computacional. La principal ventaja que tiene el razonar sobre estos sistemas usando una lógica computacional es que el buen funcionamiento de tales sistemas se tiene asegurado si se demuestra formalmente un teorema de corrección de los mismos.

Una de las lógicas computacionales más conocidas es la lógica de Boyer y Moore, un fragmento constructivo de la lógica de primer orden, con igualdad y libre de cuantificadores. El razonamiento usando la lógica de Boyer y Moore puede ser mecanizado, de manera que las pruebas formales son realizadas por un ordenador, guiado por el usuario. Esta mecanización es fundamental en cualquier aplicación de métodos formales al estudio de sistemas computacionales, ya que permite reforzar la confianza en las demostraciones formales llevadas a cabo. El sistema ACL2, sucesor de Nqthm y desarrollado por Moore y Kaufmann en la Universidad de Texas, implementa esta mecanización para la lógica de Boyer y Moore.

Hemos empleado ACL2 para razonar formalmente sobre procesos de decisión en teorías ecuacionales (es decir, teorías cuyos axiomas vienen dados por ecuaciones), enfocando especialmente nuestro trabajo hacia la demostración formal y mecánica de conceptos sobre técnicas de reescritura. De esta manera, usamos una lógica computacional para razonar sobre propiedades de la lógica ecuacional. Esta formalización consiste en expresar los conceptos y propiedades usando la lógica de Boyer y Moore, incluyendo la sucesión de lemas que permiten al demostrador obtener una prueba formal de los teoremas principales.

*Este trabajo está parcialmente finaciado por los proyectos DGES/MEC PB96-0098-C04-04 y PB96-1345

Se han formalizado un buen número de propiedades fundamentales del razonamiento ecuacional, obteniendo una biblioteca de resultados que pueden ser empleados en el futuro en la formalización de otras teorías relacionados. Entre los principales resultados obtenidos se encuentran:

1. Formalización de las propiedades reticulares de los términos de primer orden, incluyendo la corrección de un algoritmo de unificación.

2. Una estudio formal de las relaciones de reducción abstractas, incluyendo una prueba del lema de Newman.

3. Formalización de las teorías ecuacionales y de los sistemas de reescritura de términos, incluyendo una prueba del teorema de pares críticos de Knuth-Bendix.

Creemos que existen dos puntos principales de interés en el trabajo llevado a cabo. Por un lado, y desde un punto de vista teórico, las pruebas formales realizadas en un entorno de demostración automática proporcionan una verificación mecánica de una teoría matemática, además de permitir examinar y comprender sus teoremas con más rigor y claridad. En este caso, además, hemos mostrado cómo una lógica relativamente poco expresiva (de primer orden y sin cuantificadores) puede servir para razonar sobre propiedades no triviales. Por otro lado, existe un componente aplicado en nuestro trabajo: puesto que ACL2 es también un lenguaje de programación, es posible construir una biblioteca de algoritmos eficientes y formalmente verificados que pueden ser usados como componentes "certificados" en sistemas de computación simbólica.

2 El sistema ACL2

Describimos aquí muy brevemente la lógica de ACL2, conocida también como lógica de Boyer y Moore. La mejor introducción a ACL2 es [2]. Instamos al lector interesado a consultar dicho libro para obtener una completa información del sistema.

La lógica de ACL2 es una lógica de primer orden, con igualdad y sin cuantificadores. La sintaxis en la cual se escriben las fórmulas de la lógica es la misma que la del lenguaje Common Lisp (supondremos en lo que sigue que el lector está familiarizado con este lenguaje de programación y con su notación prefija). Las fórmulas no tienen cuantificadores y las variables que aparecen en ellas se consideran universalmente cuantificadas. El lenguaje incluye las conectivas usuales proposicionales, como `and`, `or`, `not` e `implies` y el símbolo de igualdad, `equal`.

La lógica incluye los axiomas usuales de la lógica proposicional y además una serie de axiomas que establecen propiedades sobre funciones y tipos de datos Lisp. Las reglas de inferencias son, entre otras, las usuales del cálculo proposicional, las de la igualdad y la regla de instanciación de variables. La manera de introducir axiomas que definen nuevos símbolos de función es mediante el *principio de definición*: usando el comando `defun` estas nuevas definiciones serán admitidas como axiomas si existe una medida ordinal respecto de la cual cada una de las llamadas recursivas decrece. Por este motivo las funciones definidas son totales, asegurándose así de que nuevas definiciones no provocan inconsistencias. La lógica tiene una defínción constructiva de los ordinales hasta ε_0, en términos de listas y números naturales. Estos ordinales se reconocen mediante el predicado `e0-ordinalp`. El orden usual entre ordinales viene dado por la función `e0-ord-<`. Una regla de inferencia muy importante en esta lógica viene dada mediante el *principio de inducción*, que permite probar una fórmula usando inducción sobre ε_0.

El mecanismo de "encapsulación" (usando el comando `encapsulate`) provee a la lógica de una manera de introducir nuevos símbolos de función mediante axiomas que aseveran ciertas propiedades sobre tales funciones, pero que posiblemente no las definen completamente. Para asegurar la consistencia, se ha de probar previamente que una función concreta (un "testigo") tiene tales propiedades. Dentro del ámbito de un `encapsulate` las propiedades establecidas con el comando `defthm` necesitan ser demostradas para las funciones testigo. Fuera de este ámbito, estos

teoremas se consideran propiedades asumidas. Los símbolos de función parcialmente definidos mediante `encapsulate` se pueden ver como variables de segundo orden, representando a funciones con esas propiedades. Existe una regla de inferencia derivada, llamada de *instanciación funcional* que permite un cierto razonamiento de segundo orden: los teoremas sobre una determinada función pueden ser instanciados para otros símbolos de función si previamente se prueba que tales símbolos representan a funciones con las mismas propiedades.

La relativa debilidad expresiva de la lógica descrita aporta una ventaja desde el punto de vista de la mecanización, ya que es posible automatizar en cierta medida la demostración de teoremas. El demostrador automático que posee ACL2 implementa esta mecanización. No vamos a entrar en detalles sobre el mismo en este artículo, ya que nos centraremos más en la formalización de teorías que en la prueba mecánica de los teoremas. Simplemente decir que el comando `defthm` inicia un intento de demostración de la fórmula que recibe como entrada. En este intento de demostración, el sistema aplica una serie de procedimientos y heurísticas (principalmente simplificación e inducción) con intención de concluir que la fórmula es un teorema. El demostrador es automático en el sentido de que una vez se llama a `defthm`, el usuario no puede interactuar con el sistema. Sin embargo, rara vez el demostrador encuentra la demostración de un resultado no trivial en el primer intento. Por esta razón, el uso del sistema es interactivo en un sentido más profundo. El papel del usuario consiste en guiar al demostrador hacia una prueba preconcebida mediante la demostración de lemas previos que serán usados para descomponer la prueba en pasos más simples.

3 Formalización del teorema de pares críticos

Como un ejemplo relevante del trabajo realizado, vamos a describir en esta sección cómo hemos formalizado en la lógica de ACL2 el teorema de Knuth y Bendix de pares críticos. Supondremos que el lector es conocedor de la teoría de los sistemas de reescritura de términos. Una muy buena introducción a esa materia se encuentra en [1] (las notaciones y definiciones que usemos son las de ese libro). El teorema de pares críticos se enuncia usualmente de la siguiente manera:

Teorema 1 (Knuth y Bendix)*:* Sea R un sistema de reescritura tal que la reducción $\rightarrow_R$ es noetheriana. Entonces $\rightarrow_R$ tiene la propiedad de Church-Rosser si para cualquier par crítico (s,t) de R, las formas normales de s y t son iguales.

Este teorema es un resultado básico para obtener algoritmos de decisión de ciertas teorías ecuacionales (véase [1]). Si queremos razonar formalmente sobre sistemas que automaticen la lógica ecuacional, la prueba de este teorema usando ACL2 es una de las piedras angulares que debe tener nuestro trabajo. Previo a la demostración mecánica, y más importante si cabe, es la formalización de su enunciado en la lógica que usamos para razonar. Se trata de expresar el teorema con los limitados recursos expresivos que pone a nuestra alcance la lógica de ACL2, que como se ha comentado en la sección anterior es un subconjunto, libre de cuantificadores, de la lógica de primer orden.

En lo que sigue, describiremos la formalización del teorema 1 en la lógica de ACL2. Cuando hablemos de "demostración", nos referiremos a "demostración mecánica llevada a cabo usando ACL2". Debido a la falta de espacio, no entraremos en detalle sobre la demostración mecánica, y omitiremos el código de algunas definiciones. Instamos al lector interesado a consultar [3] y la página web `http://www-cs.us.es/~jruiz/acl2-rewr`, donde puede encontrar la secuencia de tallada de definiciones y lemas previos que llevan a la demostración de éste y otros resultados relativos a la lógica ecuacional y la reescritura de términos.

3.1 Teorías ecuacionales

Puesto que la reescritura de términos es una reducción definida en el conjunto de los términos de primer orden, necesitaremos razonar sobre ellos en ACL2. Para ello, hemos escogido una representación prefija de los mismos, usando listas. Por ejemplo, el término $k(x, g(y), h(y, v))$ se representa mediante la lista `'(k x (g y) (h y v))`. Toda lista no vacía se puede ver como un término cuyo `car` es el símbolo de función principal y el `cdr` la lista de sus argumentos. Las variables se representan mediante objetos atómicos, con una excepción: para hacer el renombrado de términos de manera fácil, las listas que tiene como primer elemento el símbolo `var` tambien se consideran variables (por ejemplo, `'(var x 1)`). La función `variable-p` reconoce los objetos que representan variables. Las sustituciones se representan como listas de asociación, y las ecuaciones y reglas de reescritura como pares punteados de términos. La función `instance` implementa la aplicación de una sustitución a un término. Hemos implementado y verificado una serie de algoritmos que actúan sobre términos, entre los que destacamos los algoritmos de subsunción y unificación. En la formalización que describimos a continuación, resultarán de especial interes las funciones que se refieren a la estructura de árbol de un término: `positionp` comprueba si una secuencia de números naturales representa una posición de un término, `occurrence` devuelve el subárbol en una posición dada y `replace-term` lleva a cabo una sustitución de un subtérmino en una posición dada (véase [1] para obtener más detalles).

Dado un conjunto de ecuaciones E, para formalizar la relación $\overset{*}{\leftrightarrow}_E$ en ACL2, nos centramos primero en la relación de reducción $\rightarrow_E$. Aunque se podría representar tal relación simplemente como un predicado binario entre términos, adoptaremos un punto de vista diferente, con objeto de hacer énfasis en el hecho de que se trata de una "reducción": si $t_1 \rightarrow_E t_2$, mas importante que el hecho de que t_1 y t_2 están relacionados, es el hecho de que t_2 se obtiene a partir de t_1 mediante la aplicación a t_1 de una transformación u *operador*. De esta manera, $\rightarrow_E$ se puede ver como una función binaria tal que dado un término y un *operador ecuacional*, devueve otro término, llevando a cabo *un paso de reducción*. Los operadores ecuacionales se representan mediante estructuras Lisp con tres campos: la regla de reescritura (o ecuación) que se aplica, la posición del subtérmino que se reescribe y la sustitución de equiparación que se emplea para ello:

```
(defstructure eq-operator rule pos matching)
```

Con esta representación debemos tener en cuenta que no cualquier operador se puede aplicar a cualquier término: el lado izquierdo (`lhs`) de la regla tiene que subsumir al subtérmino que ocurre en la posición dada. Esta idea viene formalizada por la función `eq-legal`, que comprueba si un operador es aplicable a un término:

```
(defun eq-legal (term op E)
  (let ((pos (pos op)) (rule (rule op)) (sigma (matching op)))
    (and (eq-operator-p op) (member rule E) (positionp pos term)
         (equal (instance (lhs rule) sigma) (occurrence term pos)))))
```

La función `eq-reduce-one-step` aplica un operador ecuacional a un término, sustuituyendo el subtérmino correspondiente por la correspondiente instancia del lado derecho (`rhs`) de la ecuación:

```
(defun eq-reduce-one-step (term op)
  (replace-term
     term (pos op) (instance (rhs (rule op)) (matching op))))
```

Estas dos funciones que se acaban de presentar nos permiten definir la relación $\overset{*}{\leftrightarrow}_E$. Nótese que debido a la naturaleza constructiva de la lógica de ACL2 (no existen cuantificadores existenciales), debemos incluir un argumento con una secuencia de pasos $t_1 = u_0 \leftrightarrow_E u_1 \leftrightarrow_E u_2 \ldots \leftrightarrow_E u_n = t_2$ que justifican la equivalencia de t_1 y t_2. En la figura 1, definimos la función

`(eq-equiv-p t1 t2 p E)`. Esta función comprueba si `p` es una prueba ecuacional que justifica la equivalencia $t_1 \overset{*}{\leftrightarrow}_E t_2$, donde una *prueba ecuacional*, o simplemente una *prueba*[1] es una secuencia de *pasos de prueba* legales. Representamos los pasos de prueba mediante una estructura `r-step` con cuatro campos: `elt1`, `elt2` (los términos que se relacionan), `direct` (el sentido en el que se da el paso, directo o inverso) y `operator` (el operador que se aplica). Un paso de prueba es *legal* (tal y como define `eq-proof-step-p`) si uno de sus elementos se obtiene aplicando el operador (que debe ser aplicable) al otro, en el sentido indicado. Diremos que dos pruebas ecuacionales que justifican la misma equivalencia son *pruebas equivalentes*. Es importante destacar que la función `eq-equiv-p` describe formalmente la lógica ecuacional, ya que $E \models s = t$ si, y sólo si, $s \overset{*}{\leftrightarrow}_E t$.

```
(defstructure r-step direct operator elt1 elt2)

(defun eq-proof-step-p (s E)
  (let ((t1 (elt1 s)) (t2 (elt2 s)) (op (operator s)) (dt (direct s)))
   (and (r-step-p s)
        (implies dt (and (eq-legal t1 op E)
                         (equal (eq-reduce-one-step t1 op) t2)))
          (implies (not dt) (and (eq-legal t2 op E)
                                 (equal (eq-reduce-one-step t2 op) t1))))))

(defun eq-equiv-p (t1 t2 p E)
  (if (endp p) (equal t1 t2)
    (and (eq-proof-step-p (car p) E) (equal t1 (elt1 (car p)))
         (eq-equiv-p (elt2 (car p)) t2 (cdr p) E))))
```

Figura 1: Definición de los conceptos de prueba ecuacional y teoría ecuacional

La propiedad de Church-Rosser y la confluencia local se pueden redefinir atendiendo a la forma de una prueba ecuacional, como veremos en la subsección 3.3. Así, definimos (definiciones que omitimos), funciones que reconocen formas particulares de pruebas ecuacionales: `local-peak-p` reconoce pruebas de la forma $s \leftarrow_E u \rightarrow_E t$ y `steps-valley` reconoce pruebas de la forma $s \overset{*}{\rightarrow}_E u \overset{*}{\leftarrow}_E t$.

Los sistemas de reescritura se definen en [1] como un caso especial de conjuntos de ecuaciones: el lado izquierdo de las ecuaciones no debe ser una variable y debe ser un término que contenga a las variables del lado derecho. Definimos una función `rewrite-system` (omitida aquí) que describe este concepto. Sin embargo, la formalización que hemos presentado en esta subsección no necesita asumir que el conjunto de ecueciones deba ser un sistema de reescritura (puede ser cualquier conjunto de ecuaciones).

3.2 Termination and reduction orderings

Con objeto de formalizar las propiedades relativas a la terminación de sistemas de reescritura, usaremos *órdenes de reducción*; es decir, órdenes bien fundamentados cerrados bajo instanciación (*estables*) y bajo sustitución de subtérminos (*compatibles*). Usamos la siguiente caracterización: un sistema de reescritura R termina si, y sólo si, existe un orden de reducción $\succ$ tal que $l \succ r$ para todo $l \rightarrow r \in R$ ([1]).

La lógica de ACL2 viene dotada de una definición restringida de buena fundamentación, basada en el siguiente (meta) teorema: una relación sobre un conjunto A está bien fundamentada si, y sólo si, existe una *función de inmersión* $F : A \rightarrow Ord$ tal que $x < y \Rightarrow F(x) < F(y)$, donde Ord es la clase de todos los ordinales. En ACL2, una vez que se demuestra que una relación binaria

[1] No confundir con las pruebas realizadas usando ACL2

satisface tal condición, podría usarse en el test de admisión de una función recursiva. Puesto que tan sólo los ordinales hasta ε_0 están representados en la lógica de ACL2, esto impone una limitación en el tipo ordinal maximal de las relaciones bien fundamentadas que se pueden representar como tales en ACL2. En consecuencia, nuestra formalización ha de sufrir de la misma restricción (en cualquier caso, debemos decir que la misma demostración podría llevarse a cabo si tratara de ordinales mayores, ya que, excepto su buena fundamentación, ninguna otra propiedad particular de ε_0 se ha necesitado).

En la figura 2, se usa el mecanismo `encapsulate` para definir parcialmente la función `red<`, asumiendo que se trata de un orden de reducción general (usamos puntos suspensivos para omitir detalles técnicos, como el resto del artículo). La función `(noetherian-red< TRS)` se define de manera que comprueba si `red<` es un orden de reducción que justifica la terminación del sistema de reescritura `TRS`. Nótese que la función `fn-red<` es una inmersión en los ordinales, que justifica la buena fundamentación de `red<`. Esta propiedad se almacena en el sistema como una regla `:well-founded-relation`, lo cual permitiría usar `red<` en el test de admisibilidad de una función recursiva.

```
(encapsulate
  ((red< (t1 t2) booleanp) (fn-red< (term) e0-ordinalp))
   ....
  (defthm red<-well-founded-relation
    (and (e0-ordinalp (fn-red< t1))
         (implies (red< t1 t2) (e0-ord-< (fn-red< t1) (fn-red< t2))))
         :rule-classes :well-founded-relation)

  (defthm red<-stable
    (implies (red< t1 t2) (red< (instance t1 sigma) (instance t2 sigma))))

  (defthm red<-compatible
    (implies (and (positionp pos term) (red< t1 t2))
          (red< (replace-term term pos t1) (replace-term term pos t2))))

  (defthm red<-transitive
    (implies (and (red< x y) (red< y z)) (red< x z))))
;;; --------
(defun noetherian-red< (TRS)
  (if (endp TRS) t
    (let ((rule (car TRS)))
         (and (red< (rhs rule) (lhs rule)) (noetherian-red< (cdr TRS))))))
```

Figura 2: Un orden de reducción

3.3 El teorema de pares críticos

Podemos usar `encapsulate` para definir (parcialmente) un sistema de reescritura `(RC)`, asumiendo que tiene las propiedades descritas por las hipótesis del teorema de pares críticos: `(RC)` termina (justificado por `red<`) y todo par crítico que se obtenga a partir de dos reglas de `(RC)` tiene una forma normal común. Véase la figura 3. En esta formalización los conceptos de forma normal y de par crítico los describen formalmente las funciones `RC-normal-form` y `cp-r`, respectivamente.

La función `RC-normal-form` se define de manera que calcula la forma normal de un término respecto del sistema de reescritura `(RC)`. Se trata de aplicar iterativamente la función `r-reduce` hasta que se llega a un término irreducible. La función `(r-reduce term TRS)`, cuya definición omitimos aquí, lleva a cabo un paso de reescritura, siempre que sea posible. Recorre las posiciones

de `term` en busca de un subtérmino que esté subsumido por el lado izquierdo de una regla del sistema de reescritura `TRS`. Cuando tal subtérmino se encuentra, se sustituye por la instancia correspondiente del lado derecho de la regla. Si no se encuentra, entonces `r-reduce` devuelve `nil` (y por tanto `term` es irreducible). Tales propiedades de `r-reduce` han de ser demostradas formalmente. Nótese que para ello, se necesita también disponer de un algoritmo de subsunción formalmente verificado. Además, hemos de demostrar un teorema adicional sobre `r-reduce`: si `(r-reduce term (RC))` no devuelve `nil`, entonces devuelve un término `term` que es menor respecto a la relación bien fundamentada `red<`. Véase la página web para obtener más detalles.

La función `(cp-r l1 r1 pos l2 r2)` calcula el par crítico (si existe) determinado por las reglas `l1`→`r1` y `l2`→`r2` en la posición `pos` de `l1`. Antes de calcular el par crítico, la función renombra las reglas para conseguir que no tengan variables en común. Nótese que para razonar adecuadamente sobre esta función, necesitamos un algoritmo de unificación formalmente verificado y además demostrar propiedades sobre el renombrado de variables.

```
(encapsulate
 ((RC () terminating-rewrite-system-with-common-n-f-critical-pairs))
 ...
 (defthm RC-rewrite-system (rewrite-system (RC)))

 (defthm RC-noetherian-red< (noetherian-red< (RC)))

 (defun RC-normal-form (term)
   (declare (xargs :measure term :well-founded-relation red<))
   (let ((red (r-reduce term (RC))))
     (if red (RC-normal-form (unpack red)) term)))

 (defthm RC-common-n-f-critical-pairs
   (implies (and (member (cons l1 r1) (RC)) (member (cons l2 r2) (RC))
                 (positionp pos l1)
                 (not (variable-p (occurrence l1 pos))))
            (let ((cp-r (cp-r l1 r1 pos l2 r2)))
              (implies cp-r
                       (equal (RC-normal-form (lhs cp-r))
                              (RC-normal-form (rhs cp-r))))))))
```

Figura 3: Hipótesis del teorema de pares críticos

Una vez asumidas la s propiedades de las figuras 2 y 3, para concluir el teorema de pares críticos debemos demostrar que `(RC)` tiene la propiedad de Church-Rosser: en la terminología de pruebas ecuacionales, esto significa que para cualquier prueba en `(RC)`, *existe* una prueba valle equivalente. Como ya se ha dicho en 2, la lógica de ACL2 es una lógica sin cuantificador existencial. Por tanto debemos *definir* una función `RC-transform-eq-proof` y demostrar formalmente que, dada una prueba ecuacional `p` que justifica la equivalencia $t_1 \overset{*}{\leftrightarrow}_{\texttt{(RC)}} t_2$, entonces `(RC-transform-eq-proof p)` es una prueba valle equivalente. Se trata por tanto de una prueba *constructiva*. Este es el teorema principal que hemos demostrado usabdo ACL2:

```
(defthm kb-critical-pair-theorem
   (implies (eq-equiv-p t1 t2 p (RC))
            (and (eq-equiv-p t1 t2 (RC-transform-eq-proof p) (RC))
                 (steps-valley (RC-transform-eq-proof p)))))
```

No incluimos aquí la definición de `RC-transform-eq-proof` debido a la falta de espacio. Usa una función auxiliar importante: `RC-transform-eq-peak` transforma la primera subprueba de `p`

que sea pico local en una subprueba valle equivalente. De esta manera,`RC-transform-eq-proof` se define como la aplicación iterativa de `RC-transform-eq-peak` hasta que la prueba obtenida no tiene picos locales (y por tanto se trata de una prueba valle).

La definición de las dos funciones anteriormente mencionadas constituyen la parte más laboriosa de la prueba. En el caso de la función `RC-transform-eq-proof`, el problema principal es que la prueba de su terminación no es fácil. Obsérvese que al reemplazar un pico local por una prueba valle equivalente, el tamaño de la prueba (contado como el número de elementos que intervienen en ella) aumenta en general. Para probar su terminación hemos tenido que utilizar un orden de multiconjuntos bien fundamentado, demostrando previamente la buena fundamentación de la extensión a multiconjuntos de un orden bien fundamentado (un resultado interesante por sí mismo). Instamos al lector interesado en la prueba a consultar la página web referida al principio de esta sección.

4 Conclusiones

Hemos mostrado cómo los métodos formales pueden ser aplicados para razonar formalmente sobre propiedades de sistemas de computación simbólica y demostradores automáticos. En concreto, los autores estamos empleando el sistema ACL2 para formalizar conceptos relativos teorías ecuacionales y procedimientos decisión de las mismas. Es decir, usamos ACL2 como meta-lógica para razonar sobre otra lógica objeto: la lógica ecuacional.

El interés aplicar métodos formales a la construcción de sistemas de computación no es sólo teórico. Los métodos formales son una herramienta muy útil para asegurar el correcto funcionamiento de tales sistemas: una vez se ha especificado formalmente el comportamiento deseado de un sistema, este especificación puede ser probada como un teorema que se deduce a partir de las definiciones de las funciones que implementan dicho sistema. ACL2 ofrece un entorno en el que se combinan ejecución y razonamiento formal: ya que el lenguaje de ACL2 es un subconjunto de Common Lisp, las definiciones de una función pueden ser considerados como programas ejecutables en cualquier Common Lisp y como axiomas de una teoría lógica, a partir de los cuales podemos deducir teoremas (mecánicamente) que nos aseguren un comportamiento acorde con la especificación.

Como consecuencia de este trabajo, se ha obtenido una biblioteca de algoritmos básicos de manipulación de términos (como por ejemplo, subsunción, unificación, cálculo de pares críticos y cálculo de formas normales) definidos en Common Lisp y cuya corrección haya sido formalmente verificada usando ACL2. De esta manera, se proporcionaría una biblioteca "certificada" de funciones Common Lisp que se podrían integrar con otras funciones en la construcción de sistemas de computación simbólica y razonamiento automático. Parece natural que el trabajo futuro deba estar encaminado a mejorar la eficiencia de tales algoritmos conservando la certificación de los mismos. A más largo plazo, nuestro objetivo es definir en Common Lisp un algoritmo de completación eficiente y formalmente verificado.

Referencias

[1] Baader, F. y Nipkow, T. *Term rewriting and all that.* Cambridge University Press, 1998.

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

[3] Ruiz-Reina, J., Alonso, J., Hidalgo, M., y Martín, F. Formalizing rewriting in the ACL2 theorem prover. *AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation*), por aparecer en LNCS, Springer Verlag, 2000.