Verificación formal y eficiencia: un caso de estudio aplicado a la unificación de términos *

J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo y F.-J. Martín-Mateos
http://www.cs.us.es/{~jruiz, ~jalonso, ~mjoseh, ~fmartin}

Departamento de Ciencias de la Computación e Inteligencia Artificial.
Escuela Técnica Superior de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, España

Resumen En este trabajo presentamos un caso de estudio consistente en la aplicación del sistema ACL2 a la definición y verificación formal de un algoritmo de unificación de términos de primer orden, usando estructuras de datos eficientes: los términos se representan mediante *grafos acíclicos dirigidos* y estos grafos se almacenan en un *objeto de hebra simple*, lo que permite actualizaciones destructivas y mejora la eficiencia del algoritmo. En este caso de estudio pretendemos mostrar cómo el uso de objetos de hebra simple en ACL2 hace posible compaginar en un mismo entorno la definición de algoritmos con estructuras de datos eficientes y la verificación formal de los mismos.

1. Introducción

El sistema ACL2 [3,4] es a la vez un lenguaje de programación, una lógica que permite expresar formalmente propiedades de las funciones definidas en el lenguaje de programación, y un sistema de demostración automática que proporciona ayuda para la demostración de teoremas en la lógica. El lenguaje de programación de ACL2 es una extensión de un subconjunto aplicativo de Common Lisp y la lógica de ACL2 es una lógica de primer orden (con igualdad) sin cuantificadores que describe dicho subconjunto de Common Lisp.

El carácter aplicativo del lenguaje de ACL2 permite razonar sobre los algoritmos como funciones en el sentido matemático. A pesar de esto, es posible declarar determinados objetos como *objetos de hebra-simple* (en adelante llamados *stobjs*[1]) y realizar operaciones destructivas sobre los mismos. Mediante una serie de restricciones sintácticas en el uso de los *stobjs*, es posible asegurar que en cada momento sólo es necesaria una sola copia de tal objeto. Con estas restricciones, las operaciones destructivas son consistentes con la semántica aplicativa de ACL2, combinando la posibilidad de realizar implementaciones imperativas eficientes, con el uso de la semántica aplicativa de los lenguajes funcionales para razonar sobre ellas.

* Este trabajo ha sido financiado por el proyecto TIC2000–1368–C03–02 del MCYT
[1] Del inglés *single–threaded objects*

En este trabajo presentamos un caso de estudio en el que usamos ACL2 para implementar y verificar un algoritmo de unificación de términos de primer orden en el que los términos están representados mediante un grafo acíclico dirigido (*dag*[2] de aquí en adelante), y el grafo está almacenado en un *stobj*. La idea principal del algoritmo es que no es necesario crear nuevos términos durante el proceso de unificación, sino simplemente actualizar el grafo destructivamente, lo que mejora la eficiencia. Mostraremos cómo ACL2 permite combinar la ejecución eficiente de algoritmos y verificación formal de sus propiedades.

Aunque no presentaremos una introducción al sistema ACL2, comentaremos las cuestiones más relevantes cuando se necesiten, a lo largo del artículo. Una excelente introducción al sistema es [3]. Para obtener una descripción detallada, consúltese el manual, disponible en [4]. Debido a la falta de espacio, no daremos aquí detalles sobre la implementación o las demostraciones obtenidas. El lector interesado puede consultar todo el desarrollo en [8].

2. Unificación con *dags*

En esta sección introducimos brevemente los conceptos básicos acerca de la unificación de términos de primer orden, proceso fundamental para la mayoría de los metodos existentes de demostración automática. El lector puede encontrar en [1] una descripción completa de la teoría de la unificación.

Una *ecuación* es un par de términos de primer orden, notada como $t_1 \approx t_2$ y un *sistema de ecuaciones* es un conjunto finito de ecuaciones. Una sustitución σ es una *solución* de $t_1 \approx t_2$ si $\sigma(t_1) = \sigma(t_2)$ y es una solución de un sistema de ecuaciones S si es solución de todas las ecuaciones de S. Dadas dos sustituciones σ y δ decimos que σ es más general que δ si existe un sustitución γ tal que $\delta = \gamma \circ \sigma$, donde $\circ$ denota la composición entre funciones. Diremos que una solución de S es una *solución de máxima generalidad* si es más general que cualquier otra solución de S. Un *unificador* de dos términos t_1 y t_2 es una solución del sistema $\{t_1 \approx t_2\}$ y un *unificador de máxima generalidad (umg)* es una solución de máxima generalidad de dicho sistema. Un *algoritmo de unificación* calcula un unificador de máxima generalidad de dos términos dados, o bien indica que no son unificables.

El algoritmo de unificación que hemos implementado está basado en el conjunto de reglas de transformación de Martelli y Montanari, que se presenta en la figura 1. Este conjunto de reglas actúa sobre pares de sistemas de ecuaciones de la forma $S; U$. Intuitivamente, el sistema S se puede ver como un conjunto de pares de términos que se quieren unificar y el sistema U como el unificador[3] parcialmente calculado hasta ese momento (decimos que el par $S; U$ es un *problema de unificación*). El símbolo $\perp$ representa fallo en la unificación. Comenzando en un par de sistemas $S; \emptyset$, estas reglas se aplican (de manera no determinista)

[2] Del inglés *directed acyclic graph*.

[3] Identificaremos un sistema de ecuaciones de la forma $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, donde las x_i son variables, con la sustitución $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

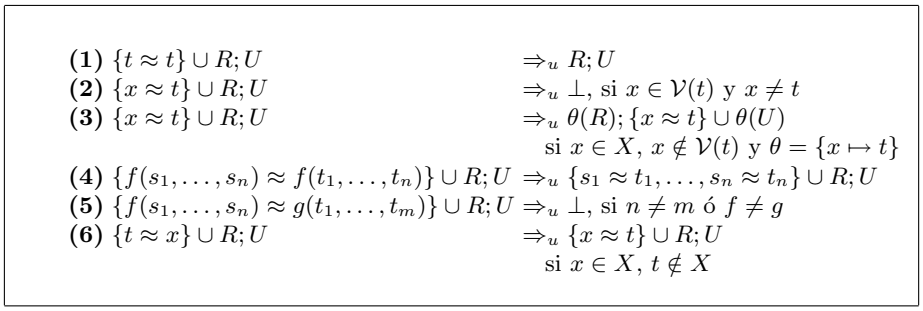

Figura 1. Reglas de transformación de Martelli–Montanari

hasta que se obtiene un par de sistemas de la forma $\emptyset; U$ (ó $\bot$). Es posible probar que la aplicación de estas reglas siempre termina y que S es unificable si y sólo si no se obtiene $\bot$; en ese caso U es un unificador de máxima generalidad (*umg*) de S. Así, se puede diseñar un algoritmo para unificar dos términos t_1 y t_2, escogiendo una estrategia para aplicar las reglas, y comenzando con el par de sistemas $\{t_1 \approx t_2\}; \emptyset$.

En un trabajo anterior [5,6,7] habíamos definido y verificado un algoritmo de unificación basado en estas reglas, como parte de una biblioteca ACL2 con pruebas formales sobre las propiedades reticulares de los términos de primer orden. En esta biblioteca, los términos de primer orden se representan en notación prefija usando listas (excepto las variables, que se representan mediante objetos atómicos). Por ejemplo, $f(x, g(y), h(x))$ se representa por la lista `'(f x (g y) (h x))`. Las sustituciones se representan mediante listas de asociación y los sistemas de ecuaciones mediante listas de pares punteados de términos. En lo que sigue, por *representación prefija* nos referiremos a esta representación de los términos en forma prefija, usando listas.

Si no se usan estructuras de datos eficientes, el algoritmo descrito por las reglas de transformación puede ser ineficiente en algunas situaciones. Obsérvese, por ejemplo, el siguiente problema clásico, que denominaremos U_n:

$$p(x_1, x_2, \ldots, x_n) \approx p(f(x_0, x_0), f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1}))$$

Un *umg* para este ejemplo es $\{x_1 \mapsto f(x_0, x_0), x_2 \mapsto f(f(x_0, x_0), f(x_0, x_0)), \ldots\}$, sustitución que asigna a cada variable x_i un arbol binario completo de profundidad i. Este unificador se obtiene aplicando repetidas veces la regla **(3)** (conocida como regla de *eliminación*). Si se usa la representación prefija, es necesario reconstruir los sistemas de ecuaciones cada vez que se aplica la regla.

La manera habitual de paliar esta fuente de ineficiencia es representar los términos como *dags*. Por ejemplo, en la figura 2 representamos el problema de unificación $f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$. Los nodos se etiquetan con los símbolos de función o variable, y las aristas conectan cada nodo con los *dags* que representan sus subtérminos inmediatos. Por tanto, podemos identificar un término con el nodo raiz del *dag* que lo representa. Nótese también que existe cierta *compartición de estructuras*, al menos para las variables.

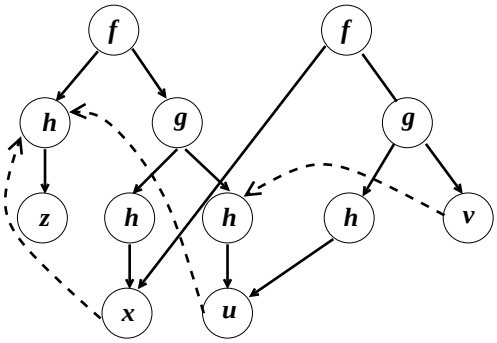

Figura 2. Representación como *dag* de $f(h(z), g(h(x), h(u))) \approx f(x, g(h(u), v))$

Para implementar un algoritmo de unificación con esta representación de los términos, la idea principal es que no es necesario construir nuevos términos, sino simplemente actualizar punteros. En particular, la regla de eliminación se puede implementar mediante la inclusión de un puntero que conecta la variable con el término que se le asigna; de esta manera, no se necesita reconstruir un término cuando se le aplica una sustitución. En la figura 2, estos punteros se representan mediante líneas discontinuas. Para conocer el término asignado a una variable, basta con seguir los punteros recorriendo el árbol en profundidad. En este caso, la sustitución que se representa es $\{x \mapsto h(z), u \mapsto h(z), v \mapsto h(h(z))\}$, que resulta ser un *umg* de $f(h(z), g(h(x), h(u)))$ y $f(x, g(h(u), v))$.

3. Implementación del algoritmo en ACL2

En esta sección describimos una implementación en ACL2 de un algoritmo de unificación en el que los términos se representan en forma de *dags* usando un *stobj*. Este *stobj* se define de la siguiente manera:

```
(defstobj terms-dag
  (dag :type (array t (0)) :resizable t))
```

De esta manera se define un *stobj* llamado `terms-dag` como una estructura que contiene una matriz (llamada `dag`) que puede ser redimensionada dinámicamente y que va a almacenar los nodos del *dag* que representa el problema de unificación. Mediante esta llamada a `defstobj`, se definen automáticamente las funciones `dagi` y `update-dagi`, que permiten, respectivamante, acceder y actualizar las celdas de la matriz del *stobj*. El acceso y actualización se produce en tiempo constante y además la actualización es destructiva. Sin embargo, desde el punto de vista lógico, la matriz se puede ver como una lista, dotada de una semántica aplicativa (es decir, como si cada vez que se actualizara la matriz se creara una nueva). Esto es posible debido a que el uso de un *stobj* en ACL2 tiene ciertas restricciones sintácticas que permiten asegurar que sólo es necesario

mantener una copia del mismo en cada momento. En esencia, esta restricción sintáctica obliga a que la única referencia al *stobj* se haga mediante su nombre (`terms-dag`, en este caso). Véase [2] para más información sobre los *stobjs*.

Cada nodo del grafo es almacenado en una celda de la matriz `dag`. Es decir, un nodo del grafo se puede identificar con el número natural correspondiente al índice de la celda de la matriz. Las celdas almacenan información sobre las etiquetas y vecinos de cada nodo, mediante el siguiente convenio:

- Si el nodo `i` representa una variable no asignada x, entonces la celda `(dagi i terms-dag)` contiene el par punteado $(x$ `.` `t`$)$.
- Si el nodo `i` representa una variable asignada a un término, entonces la celda `(dagi i terms-dag)` contiene el índice n correspondiente al nodo raiz del término al cual la variable está asignada.
- Si el nodo `i` es el nodo raiz de un término no variable $f(t_1, \ldots, t_n)$, entonces `(dagi i terms-dag)` es un par punteado de la forma $(f$ `.` $l)$, donde l es la lista de los índices de los nodos raices correspondientes a $t_1, \ldots, t_n$.

Así, podemos almacenar un problema de unificación en `terms-dag`. Por ejemplo, con el término $equ(f(h(z), g(h(x), h(u))), f(x, g(h(u), v)))$ (esto es, el problema de unificación de la figura 2) las celdas significativas de la matriz `dag` son:

```
((EQU 1 9)
 (F 2 4) (H 3) (Z . T) (G 5 7) (H 6) (X . T) (H 8) (U . T)
 (F 10 11) 6 (G 12 14) (H 13) 8 (V . T))
```

Identificaremos, de manera natural, un índice de la matriz con el término cuyo nodo raiz es el correspondiente a dicho índice. Podemos sacar partido de esta idea en la definición de una función que aplica un paso de transformación de la relación $\Rightarrow_u$ (figura 3). A continuación describimos dicha función, llamada `dag-transform-mm-st`.

Además del *stobj*, esta función recibe como entrada un sistema `S` (no vacío) de ecuaciones pendientes de unificar, y una sustitución `U` parcialmente calculada. Dependiendo de la forma de la primera ecuación de `S`, se aplica una de las reglas de $\Rightarrow_u$. La clave del procedimiento es que `S` y `U` *sólo contienen punteros* a los correspondientes términos almacenados en `terms-dag`. En particular, `S` es una lista de pares de índices, y `U` es una lista de pares de la forma $(x$ `.` $n)$, donde x es un símbolo de variable y n es el índice del nodo raiz del término asignado a x. La función devuelve un multivalor que consta de los siguientes componentes, obtenidos como resultado de la aplicación de un paso de transformación de $\Rightarrow_u$: el sistema resultante de ecuaciones pendientes de unificar, la sustitución resultante, un valor booleano (si se obtiene $\perp$, este valor es `nil`) y el *stobj* (posiblemente modificado) `terms-dag`. El *stobj* `terms-dag` se actualiza sólamente después de aplicar la regla de eliminación, haciendo que una variable no asignada hasta el momento quede asignada a un término.

Usando esta función que aplica un paso de las reglas transformación, podemos definir una función que calcula el unificador de máxima generalidad de

```
(defun dag-transform-mm-st (S U terms-dag)
  (declare (xargs :stobjs terms-dag))
  (let* ((ecu (car S))
         (t1 (dag-deref-st (car ecu) terms-dag))
         (t2 (dag-deref-st (cdr ecu) terms-dag))
         (R (cdr S))
         (p1 (dagi t1 terms-dag))
         (p2 (dagi t2 terms-dag)))
    (cond
     ((= t1 t2) (mv R U t terms-dag))
     ((dag-variable-p p1)
      (if (occur-check-st t t1 t2 terms-dag)
          (mv nil nil nil terms-dag)
        (let ((terms-dag (update-dagi t1 t2 terms-dag)))
          (mv R (cons (cons (dag-symbol p1) t2) U) t terms-dag))))
     ((dag-variable-p p2)
      (mv (cons (cons t2 t1) R) U t terms-dag))
     ((not (eql (dag-symbol p1)
                (dag-symbol p2)))
      (mv nil nil nil terms-dag))
     (t (mv-let (pair-args bool)
                (pair-args (dag-args p1) (dag-args p2))
                (if bool
                    (mv (append pair-args R) U t terms-dag)
                  (mv nil nil nil terms-dag)))))))
```

Figura 3. Definición de un paso de transformación

dos términos. Dicha función, que llamamos `dag-mgu`, recibe como entrada dos términos (en representación prefija), almacena ambos términos como *dags* en `terms-dag` (redimensionando previamente la matriz `dag` al tamaño necesario) y aplica iterativamente la función `dag-transform-mm-st` hasta que se detecta que los términos no son unificables o bien hasta que no quedan más ecuaciones por unificar. En este último caso, se devuelve la sustitución (en representación prefija) que se obtiene siguiendo los índices de las variables instanciadas. En la página web puede consultarse el resto de las definiciones utilizadas.

Lo que sigue son dos ejemplos obtenidos con `dag-mgu`. Nótese que la función devuelve dos valores: el primero es un booleano indicando si los términos son o no unificables y el segundo el *umg*, en el caso de que lo sean.

```
ACL2 >(dag-mgu '(f (h z) (g (h x) (h u))) '(f x (g (h u) v)))
(T ((V H (H Z)) (U H Z) (X H Z)))
```

```
ACL2 >(dag-mgu '(f y x) '(f (k x) y))
(NIL NIL)
```

Es interesante destacar que las restricciones sintácticas impuestas en ACL2 sobre las funciones que usan *stobjs* se cumplen de manera natural en este caso. Asímismo, el hecho de que el lenguaje de ACL2 sea un subconjunto de Common Lisp hace posible compilar y ejecutar las funciones que definen el algoritmo en cualquier Common Lisp (si previamente se han cargado las funciones ACL2 necesarias).

4. Verificación formal del algoritmo

Presentamos a continuación los principales teoremas que hemos probado acerca de la función `dag-mgu`, estableciendo formalmente que dicha función calcula el unificador de máxima generalidad de dos términos si y sólo si los dos términos son unificables:

```
(defthm dag-mgu-completeness
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma)
                       (instance t2 sigma)))
           (first (dag-mgu t1 t2))))

(defthm dag-mgu-soundness
  (implies (and (term-p t1) (term-p t2)
                (first (dag-mgu t1 t2)))
           (equal (instance t1 (second (dag-mgu t1 t2)))
                  (instance t2 (second (dag-mgu t1 t2))))))

(defthm dag-mgu-most-general-solution
  (implies (and (term-p t1) (term-p t2)
                (equal (instance t1 sigma)
                       (instance t2 sigma)))
           (subs-subst (second (dag-mgu t1 t2)) sigma)))
```

La función `instance` define la aplicación de una sustitución a un término y el predicado `subs-subst` define la relación de generalidad entre sustituciones. El predicado `term-p` reconoce aquellos objetos ACL2 que representan términos de primer orden en notación prefija. Nótese que la teoría básica que nos permite expresar estas propiedades está construida sobre la representación de los términos en notación prefija; del mismo modo la salida y entrada del algoritmo representa a los términos en notación prefija, aún cuando el proceso interno del algoritmo se hace sobre los términos representados como *dags*.

El primer teorema, `dag-mgu-completeness`, expresa que el algoritmo devuelve `t` (como primer valor) si los dos términos son unificables. El teorema `dag-mgu-soundness` expresa que en ese caso además devuelve (como segundo

valor) un unificador de los términos que recibe como entrada. Por último, el teorema `dag-mgu-most-general-solution` establece que dicha sustitución es más general que cualquier otro unificador de los dos términos.

5. Comentario sobre la prueba

En esta sección comentamos brévemente las cuestiones que consideramos más destacables del trabajo de verificación que ha conducido finalmente a la prueba de las propiedades anteriores usando el demostrador automático de ACL2.

5.1. La metodología seguida

El demostrador automático del sistema ACL2 puede ser visto como un asistente que ayuda al usuario a demostrar propiedades (en la lógica de ACL2) acerca de las funciones definidas. Las principales técnicas de demostración empleadas por el demostrador son *simplificación* (principalmente reescritura) e *inducción*; posiblemente una de las claves del éxito de ACL2 es la generación automática de esquemas de inducción que permiten demostrar propiedades de las funciones definidas, aplicando el principio de inducción de la lógica.

Aunque el demostrador no es interactivo (una vez que se llama al comando `defthm` no es posible interactuar con el sistema), en un sentido más amplio puede verse como interactivo. Habitualmente el usuario guía al demostrador hacia una prueba preconcebida (en la mayoría de los casos, una prueba informal hecha a mano), mediante la demostración previa de determinados lemas auxiliares que el sistema usa como reglas de reescritura durante el proceso de simplificación. En algunos casos, algunos lemas adicionales necesarios se deducen después de inspeccionar una prueba fallida de otros lemas. Esta metodología, habitual para la mayoría de usuarios de ACL2, es la que hemos seguido para guiar al demostrador hacia la demostración de los teoremas anteriores.

5.2. Listas *versus* matrices

La principal ventaja en el razonamiento acerca de funciones que usan *stobjs*, es que desde el punto de vista lógico, una matriz de un *stobj* no es más que una lista. Así, las funciones de acceso y modificación de una matriz de un *stobj* tienen, en la lógica de ACL2, las mismas propiedades que las funciones aplicativas `nth` y `update-nth` que, respectivamente, acceden y modifican los contenidos de una lista.

Es por este motivo que gran parte del razonamiento sobre los grafos acíclicos dirigidos lo podemos hacer como si el grafo estuviera almacenado en una lista, donde los elementos de la lista se corresponden con los contenidos de la matriz del *stobj*. Aparte de simplificar la formulación de los teoremas, esto permite definir determinadas funciones sobre los contenidos del *stobj* siguiendo el estilo habitual de definición de funciones recursivas sobre listas. Usualmente, es más fácil la demostración de propiedades de funciones definidas de esta manera que sus análogas definidas haciendo recursión en el índice de una matriz.

5.3. Refinamiento paso a paso

La mayoría de las propiedades esenciales del algoritmo de unificación se demuestran más fácilmente si consideramos los términos con la representación prefija. De hecho, previamente a la verificación de `dag-mgu` hemos verificado un algoritmo de unificación basado en el conjunto de reglas de transformación de Martelli–Montanari actuando sobre la representación prefija [7]. Es posible trasladar todas las propiedades de este algoritmo al definido por `dag-mgu`, que implementa las reglas de transformación actuando sobre la representación en forma de *dag*: para ello basta demostrar que todo paso de transformación dado sobre la representación en forma de *dag* se corresponde con un paso de transformación dado sobre la representación en forma prefija.

Más específicamente, si notamos como UPL al conjunto de problemas de unificación representados en forma prefija, y como UPL_d al conjunto de problemas de unificación representados mediante grafos acíclicos dirigidos, la clave está en demostrar que el siguiente diagrama es conmutativo:

$$\begin{array}{ccc} UPL & \stackrel{\Rightarrow_u}{\longrightarrow} & UPL \\ dt \uparrow & & dt \uparrow \\ UPL_d & \stackrel{\Rightarrow_{u,d}}{\longrightarrow} & UPL_d \end{array}$$

Aquí dt es una función tal que dado un problema de unificación representado mediante un grafo acíclico dirigido, obtiene el correspondiente problema de unificación en representación prefija. Además, $\Rightarrow_{u,d}$ denota la transformación de Martelli–Montanari actuando sobre la representación en forma de *dag* (tal y como se define con la función `dag-transform-mm-st`). Nótese que la demostración de la conmutatividad del diagrama anterior significa lo siguiente:

- Probar que cuando se dan las condiciones necesarias para la aplicación de una regla a un problema de unificación representado en forma de *dag*, entonces se tienen las condiciones necesarias para que la misma regla se pueda aplicar al correspondiente problema de unificación representado en forma prefija.
- En ese caso, el problema de unificación obtenido aplicando el paso de transformación a la representación prefija, es el mismo que el problema de unificación en forma prefija correspondiente al problema de unificación obtenido aplicando el paso de transformación a la representación en forma de *dag*.

Una vez demostrado esto en ACL2, es inmediato trasladar las propiedades previamente verificadas acerca del algoritmo de unificación que actúa sobre la representación prefija a propiedades análogas acerca del algoritmo `dag-mgu`, que actúa sobre los *dags*. Este metodología se conoce como *refinamiento paso a paso* y nos permite separar claramente la lógica del proceso, de las propiedades de las estructuras de datos manejadas por un algoritmo particular.

5.4. Restricciones que aseguran la terminación

La lógica de ACL2 es una lógica de funciones totales. Esto quiere decir que el principio de definición de ACL2 sólo admite nuevas definiciones de funciones recursivas previa demostración de que terminan para cualquier dato de entrada. Por ello, se requiere la demostración de que las llamadas recursivas se producen sobre objetos que son menores que los argumentos de entrada respecto de determinada medida bien fundamentada. De esta manera, se asegura que la consistencia de la lógica se mantiene después de introducir una nueva definición [3].

En este caso, puede que algunas de las funciones que implementan el algoritmo de unificación no terminen en algunos casos. Por ejemplo, si se quiere asegurar que la aplicación iterativa de las reglas de Martelli–Montanari termina, entonces es necesario comprobar que el grafo correspondiente al problema de unificación inicial es acíclico. Lo que sigue es la definición de la función que aplica exhaustivamente las reglas de transformación a un problema de unificación dado[4].

```
(defun dag-solve-system-st (S U bool terms-dag)
  (if (well-formed-upl-st S U terms-dag)
      (if (or (not bool) (endp S))
          (mv S U bool terms-dag)
        (mv-let (S1 U1 bool1 terms-dag)
                (dag-transform-mm-st S U terms-dag)
                (dag-solve-system-st S1 U1 bool1 terms-dag)))
    (mv nil nil nil terms-dag)))
```

La terminación de esta función sólo está asegurada cuando, entre otras condiciones, el grafo almacenado en `terms-dag` es acíclico y las variables están compartidas (lo cual está garantizado cuando se almacenan términos de primer orden, pero no en general). Por tanto, es necesario definir una función `well--formed-upl-st` que comprueba estas condiciones de "buena formación". El hecho de tener que razonar sobre estas condiciones de buena formación, y en particular sobre grafos acíclicos, ha hecho necesario desarrollar una biblioteca que verifica formalmente una serie de propiedades sobre *dags* (en [8] puede consultar más detalles).

Además de la función que aplica iterativamente las reglas de transformación, hay otras funciones que necesitan comprobaciones explícitas que aseguren la terminación. Por ejemplo, el test de ocurrencia de una variable en un término (en la regla de eliminación) también necesita que el grafo almacenado en `terms-dag` sea acíclico. Es claro que la introducción de esas condiciones en las definiciones recursivas, aunque imprescindibles para la verificación de las mismas, hacen que el tiempo de ejecución se haga inaceptable desde el punto de vista práctico. Por ejemplo, la comprobación de no existencia de ciclos es costosa en tiempo y según

[4] Las transformaciones se aplican hasta que se detecta que el problema no tiene solución o hasta que no quedan más ecuaciones pendientes de unificar.

la definición anterior tendría que realizarse *en cada llamada recursiva*. En la sección 6 comentamos cómo solventar parcialmente este problema.

5.5. El esfuerzo de prueba

La tabla siguiente muestra algunos datos cuantitativos sobre el esfuerzo de prueba. En concreto el número de definiciones y de teoremas necesitados durante las distintas fases del trabajo de verificación.[5].

Fase	Definiciones	Teoremas
Algoritmo de unificación (repr. prefija)	24	81
Grafos acíclicos	37	95
Conmutatividad del diagrama	39	66
Almacenamiento de los términos en el grafo	34	208
Algoritmo usando stobjs	43	76
Total	177	703

El número de teoremas y definiciones necesarias da una cierta idea de la complejidad de las teorías formalizadas, así como de la *granularidad* de las pruebas obtenidas y del grado de automatización de las mismas. También es destacable la gran cantidad de teoremas necesarios para verificar la función que almacena los términos que se reciben como entrada en un grafo acíclico dirigido, necesaria en la fase inicial del algoritmo.

6. Ejecución del algoritmo

Como ya se ha señalado en la subsección 5.4, las condiciones necesarias para asegurar la terminación de algunas de las funciones que definen el algoritmo de unificación sobre *dags* hacen que, en la práctica, la ejecución de estas funciones sea demasiado costosa. Para solventar este inconveniente, ejecutamos las funciones sin las comprobaciones de "buena formación".

En ACL2, existe la posibilidad de definir funciones en *modo programa*. En contraste con las definidas en *modo lógico* (como las que hemos visto hasta ahora), el sistema no exige que se demuestre la terminación de la función para cualquier dato de entrada, pero en contrapartida, la definición no entra a formar parte de los axiomas de la lógica y por tanto no se puede razonar sobre ella, aunque sí se puede ejecutar. Nuestra idea es definir un algoritmo análogo a `dag-mgu` pero en modo programa, simplemente quitando las condiciones de "buena formación" en todas las funciones que las llevan. Por ejemplo, la siguiente función sería la análoga, en modo programa, de la función `dag-solve-system-st` definida anteriormente:

[5] Nótese que no hemos incluido en la tabla datos sobre la teoría básica acerca de los términos de primer orden.

```
(defun dag-solve-system-program (S U bool terms-dag)
  (declare (xargs :stobj terms-dag :mode :program))
  (if (or (not bool) (endp S))
      (mv S U bool terms-dag)
    (mv-let (S1 U1 bool1 terms-dag)
            (dag-transform-mm-program S U terms-dag)
            (dag-solve-system-program S1 U1 bool1 terms-dag))))
```

Para asegurarnos de que las funciones en modo programa calculan los mismos valores que las funciones en modo lógico sobre datos "bien formados", hemos demostrado teoremas que aseguran que la recursión de las funciones en modo lógico es cerrada respecto a aquellos objetos bien formados (es decir, que la condición de buena formación es un invariante del proceso). Por ejemplo, el siguiente teorema expresa que la aplicación de reglas de transformación preserva la condición de "buena formación":

```
(defthm well-formed-upl-st-preserved-by-dag-transform-mm-st
  (implies (and (well-formed-upl-st S U terms-dag)
                (consp S))
           (mv-let (S1 U1 bool1 terms-dag)
                   (dag-transform-mm-st S U terms-dag)
                   (well-formed-upl-st S1 U1 terms-dag))))
```

Una vez que probamos que la recursión es cerrada respecto a las condiciones de buena formación, parece justificado usar las funciones en modo programa para ejecutar el algoritmo, en lugar de las funciones en modo lógico.

La siguiente tabla comparativa muestra datos de tiempo (en segundos) y espacio (en kilobytes) necesarios para resolver el problema U_n descrito en la sección 2, para distintos valores de n, usando una definición aplicativa (con representación prefija) del algoritmo, y la definición con *stobjs* que hemos presentado aquí[6]:

	Aplicativo		Dag/Stobj		Cuadrático	
n	Tiempo	Espacio	Tiempo	Espacio	Tiempo	Espacio
10	0.004	51	0.007	1.8	0.001	9.1
15	0.14	1541	0.27	2.73	0.002	10.1
20	8.83	49160	8.81	3.59	0.002	11.8
21	21.2	98313	17.61	3.76	0.002	12
1000	-	-	-	-	4.08	214
5000	-	-	-	-	120	945

Como se puede observar, los datos de tiempo son similares en la versión prefija y en la versión que usa *stobjs*. Sin embargo, en cuanto a consumo de espacio de memoria el rendimiento de éste último es considerablemente mejor que el

[6] Los datos están tomados usando un procesador Pentium III a 800 Mhz con 256Mb de RAM.

de la versión aplicativa. En particular, su complejidad en espacio es lineal en el tamaño de los términos de entrada, mientras que la versión aplicativa tiene complejidad exponencial. Lamentablemente, el algoritmo presentado aquí sigue siendo exponencial en tiempo[7], ya que algunas operaciones, como por ejemplo el test de ocurrencia de la regla de eliminación, puede que necesiten recorrer un término de tamaño exponencial. Hay que decir, sin embargo, que este algoritmo de unificación es el más comunmente usado en la práctica, puesto que en la mayoría de los casos no se dá este comportamiento exponencial. En cualquier caso, es posible realizar determinadas modificaciones técnicas al algoritmo para reducir su complejidad y hacerlo cuadrático en tiempo. En la tabla hemos incluido también los datos de tiempo para este algoritmo cuadrático, que hemos implementado en modo programa [8], aunque que por el momento aún no ha sido verificado formalmente. Obsérvese cómo con la versión cuadrática, es posible resolver, en un tiempo razonable, el problema U_n para $n = 5000$.

7. Conclusiones y trabajo futuro

Hemos presentado un caso de estudio en el que se pretende mostrar cómo el sistema ACL2 se puede usar para compaginar, en el mismo entorno, definición de algoritmos con estructuras de datos eficientes y verificación formal. En concreto, hemos usado *stobjs* para implementar y verificar un algoritmo de unificación en el que los términos están representados mediante grafos acíclicos dirigidos.

Como conclusión positiva, hemos visto que es posible verificar formalmente algoritmos eficientes. Como contrapartida, los datos sobre el esfuerzo de verificación que se presentan en la tabla de la sección 5.5 muestran cómo la mayor complejidad del algoritmo definido hace que el esfuerzo de verificación sea también mayor. En un trabajo de verificación previo [7], habíamos verificado una versión aplicativa del algoritmo, usando notación prefija. En aquel trabajo de verificación se necesitaron 19 definiciones y 129 teoremas, lo que contrasta con las 177 definiciones y 703 teoremas necesitados en este trabajo. En cualquier caso, este esfuerzo de verificación adicional ha dado como resultado el desarrollo de una serie de bibliotecas que pueden ser reusadas en otras formalizaciones (como por ejemplo, la teoría de grafos desarrollada).

Otra cuestión importante ha sido la dificultad que ha planteado la necesidad de definir condiciones que aseguren la terminación de las funciones. Aunque que la ejecución de las funciones en modo programa solventa parcialmente el problema, pensamos que sería necesario un mayor soporte del sistema para salvar este tipo de situaciones.

En cuanto, al trabajo futuro, como se apuntó al final de la sección anterior, estamos intentando verificar el algoritmo de unificación cuadrático que ya se ha definido en modo programa. Las modificaciones técnicas que son necesarias para obtener la versión cuadrática del algoritmo están realizadas en un estilo

[7] En concreto, con el problema U_n alcanza dicha complejidad. Si se resuelve el problema U_n cambiando de orden las ecuaciones, el tiempo hubiera sido lineal, aunque la versión aplicativa seguiría siendo exponencial.

imperativo de programación, lo que contrasta con el caracter funcional de la lógica de ACL2 y hace más complicada su verificación formal.

Referencias

1. Baader, F. y Snyder, W. Unification theory. *Handbook of Automated Reasoning*, Elvesier Science Publishers, 2001.
2. Boyer R.S. y Moore J S. Single-threaded objects in ACL2. En URL: `www.cs.utexas.edu/users/moore/publications/acl2-papers.html#Foundations`.
3. Kaufmann, M., Manolios, P., y Moore, J S. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
4. Kaufmann, M. y Moore, J S. `http://www.cs.utexas.edu/users/moore/acl2/` ACL2 Version 2.6, 2002.
5. Ruiz-Reina, J.L. Una teoría computacional acerca de la lógica ecuacional *Tesis doctoral*, Universidad de Sevilla, 2001.
6. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., y Martín, F.J. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover En *Joint Conference on Declarative Programming*, L'Aquila (Italia), 1999.
7. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., y Martín, F.J. A theory about first–order terms in ACL2 En *Third ACL2 Workshop*, Grenoble, 2002.
8. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., y Martín, F.J. Un algoritmo de unificación en ACL2 usando *dags*. URL: `www.cs.us.es/~jruiz/-unificacion-dag`, 2002.