CONTRIBUCIONES CIENTÍFICAS
EN HONOR DE MIRIAN ANDRÉS GÓMEZ
(Laureano Lambán, Ana Romero y Julio Rubio, editores),
Servicio de Publicaciones, Universidad de La Rioja,
Logroño, Spain, 2010.

TOPOLOGÍA SIMPLICIAL EN ACL2

L. LAMBÁN, F. J. MARTÍN–MATEOS Y J. L. RUIZ–REINA

Dedicado a la memoria de Mirian. "Compañera del alma, tan temprano..."

Resumen. En este trabajo se introduce un marco abstracto para la formalización de la Topología Simplicial en el sistema ACL2. Este marco permite expresar en lenguaje de lógica de primer orden propiedades sobre conjuntos simpliciales y sobre complejos de cadenas que habitualmente se enuncian en términos de lógica de orden superior. La formalización en ACL2 lleva a la noción de polinomio simplicial. Se desarrolla en ACL2 una teoría de polinomios simpliciales, que se aplica para obtener demostraciones en dicho sistema de algunos resultados esenciales en Topología Simplicial.

Abstract. In this paper an abstract framework to formalize Simplicial Topology in the ACL2 system is introduced. The aim of the framework is to allow the modeller to express in the language of first-order logic properties on simplicial sets and chain complexes, which usually are stated in higher-order logic. The formalization in ACL2 led us to the concept of *simplicial polynomial*. A theory of simplicial polynomials is then developed in ACL2, and applied to obtain ACL2 proofs of some essential Simplicial Topology results.

1. Introducción

Este trabajo se inscribe en el mismo ámbito en el que Mirian Andrés estaba desarrollando su actividad investigadora: "Demostración automatizada en Topología Algebraica, con aplicación a la corrección de sistemas de Cálculo Simbólico". Como se observa en las referencias utilizadas, sus contribuciones son la base de una buena parte del contenido de este artículo. Aunque aquí se efectúa un cambio de enfoque respecto al modo en que en el proyecto de doctorado de Mirian se estaban abordando estos mismos problemas, es voluntad de los autores presentar esta contribución como la continuación natural de su proyecto de Tesis Doctoral. De hecho, algunos de los resultados que aparecen en este trabajo ya estaban prácticamente alcanzados por Mirian, aunque usando una infraestructura distinta de demostración.

Key words and phrases. Simplicial set, ACL2, automated reasoning.

Este trabajo ha sido financiado parcialmente por el Ministerio de Educación y Ciencia, proyecto coordinado MTM2009-13842.

Kenzo [7] es un sistema software que está dedicado al Cálculo Simbólico en Topología Algebraica. Fue desarrollado en la década de los noventa bajo la dirección de F. Sergeraert, usando como lenguaje de programación Common Lisp. Su utilización ha permitido obtener algunos resultados (invariantes topológicos de espacios; en concreto, cálculos en homología) que todavía no han sido probados por otros medios, ni teóricos ni computacionales. Este hecho hace que el análisis de la corrección de los algoritmos que usa Kenzo se convierta en una tarea de gran interés, que además requiere aportaciones desde diferentes ámbitos de lo que se denominan métodos formales en Informática.

En [10] y [6] se desarrolla y fundamenta una formalización de las estructuras de datos utilizadas en Kenzo (aunque dirigida hacia su antecesor, el sistema EAT [14]). En particular, se obtiene una caracterización en términos categoriales de la implementación en el sistema EAT de dichas estructuras de datos. En [3], [5] y [2] se usan asistentes para la demostración (Isabelle, Coq y ACL2, respectivamente) con el objetivo de garantizar la corrección de ciertos algoritmos fundamentales del sistema Kenzo. A partir de las demostraciones proporcionadas por dichos asistentes es posible generar el correspondiente código asociado (al menos en el caso de Isabelle y Coq), quedando por resolver el difícil problema de estudiar la equivalencia entre dicho "código certificado" y el "código real" de Kenzo. Como alternativa, parece natural usar asistentes de demostración para verificar el propio código de Kenzo. En esta línea, el demostrador ACL2 [8] se erige como un candidato apropiado, ya que es a la vez un lenguaje de programación, una lógica computacional y un demostrador de teoremas. Como lenguaje de programación, ACL2 puede ser entendido como un subconjunto de Common Lisp (extendido con algunas funciones relacionadas con la lógica del sistema), de ahí su adecuación para ser relacionado con Kenzo. Siguiendo esta dirección, en [12] se da una prueba certificada de un fragmento concreto de código presente en Kenzo. No obstante, ACL2 presenta ciertas limitaciones que impiden que esta dirección pueda usarse de forma generalizada sobre la totalidad del código presente en Kenzo. Las restricciones vienen determinadas por el hecho de que la lógica subyacente en el sistema ACL2 es básicamente lógica de primer orden, lo que en términos de lenguaje se traduce en un estilo de programación aplicativa en el que las funciones no son elementos de primera clase (es decir, no pueden ser pasadas como argumentos ni devueltas como valores de otras funciones). Por contra, buena parte de las construcciones presentes en Kenzo, y las propiedades topológicas en las que se basan, se establecen en términos de lógica de orden superior, lo que lleva implícito que en su implementación se hayan usado de forma esencial los mecanismos de programación funcional de orden superior del lenguaje Common Lisp. Para superar este obstáculo, en [1] se propone como estrategia trabajar sobre versiones modificadas, programadas en ACL2, de código real de Kenzo. Esta línea de trabajo conduciría hacia una versión certificada (que es esperable que sea más ineficiente que el sistema real), que permitiría la realización de *testing* de los resultados obtenidos por Kenzo. Con ello se incrementa la fiabilidad del sistema, pero sin abordar de forma directa su corrección.

El presente trabajo sigue la línea que se acaba de describir. Sin separarse esencialmente del objetivo inicial (la prueba de la corrección del sistema Kenzo), el foco se fija en el desarrollo de la infraestructura necesaria para poder desarrollar Topología Algebraica en el sistema ACL2, lo que necesariamente exige superar las limitaciones impuestas por la lógica del demostrador. El trabajo se circunscribe al ámbito de la Topología Simplicial. Se presenta un marco abstracto, inspirado por la definición de los objetos simpliciales en términos de prehaces, que permite desarrollar los conceptos y construcciones característicos de la Topología Simplicial, evitando la necesidad de expresar propiedades que requieran lógica de orden superior. Además, se realiza la correspondiente modelización y representación en ACL2, mostrando su adecuación para el desarrollo de una Topología Simplicial certificada en dicho sistema. Como aplicación, se obtiene una prueba en ACL2 del resultado que establece que el operador borde, definido sobre el complejo de cadenas asociado a un conjunto simplicial, constituye un verdadero morfismo diferencial; es decir, su cuadrado es nulo.

Este artículo se organiza de la siguiente forma. Se dedica la siguiente sección a presentar algunas nociones y construcciones básicas de Topología Simplicial, que serán objeto de estudio en el resto del trabajo. En la sección 3 se plantea un modelo formal que resulta adecuado para representar en el sistema ACL2 dichos conceptos y construcciones de carácter topológico. La sección 4 constituye una breve presentación del sistema ACL2. En la sección 5 se crea la infraestructura necesaria en ACL2 para poder representar esos objetos de carácter simplicial, dando lugar a la noción de polinomio simplicial. Se desarrolla la correspondiente teoría como instancia de una teoría más general, la de combinaciones lineales sobre un monoide. En la sección 6 se aplica dicha infraestructura para obtener una prueba certificada de la propiedad $d^2 = 0$, donde d es el operador borde. El trabajo concluye con un apartado de conclusiones y trabajo futuro.

2. Topología Simplicial: conjuntos simpliciales

La esencia de la Topología Simplicial [13] radica en la sustitución de los espacios topológicos por modelos combinatoriales de los mismos, en los que se puedan estudiar más fácilmente las propiedades topológicas y realizar cálculos de los invariantes algebraicos; por ejemplo, cálculo de los grupos de homología y de homotopía. Los modelos combinatoriales más simples son los *complejos simpliciales*. Constituyen una abstracción de los *espacios topológicos triangulados*, los cuales forman una clase suficientemente grande en la que desarrollar las nociones y técnicas de la topología general y algebraica. Sin embargo, para la realización de cálculos resulta más apropiada la noción de *conjunto simplicial*. En este trabajo se va a usar una definición de conjunto simplicial más abstracta que la que habitualmente aparece en la literatura [13], presentando los conjuntos simpliciales como un caso particular de prehaces sobre una categoría.

Para llegar a la noción de conjunto simplicial, se considera la *categoría simplicial*, habitualmente denotada por Δ^*. Sus objetos están parametrizados por los números naturales $n \in \mathbb{N}$, concretamente son la colección $\{\mathbf{n}; n \geq 0\}$, donde $\mathbf{n}$

representa el conjunto $\mathbf{n} = \{0, 1, \ldots, n\}$. Sus morfismos son las aplicaciones conjuntistas crecientes.

Entre los morfismos de Δ^* hay dos familias particularmente importantes. Las aplicaciones que pertenecen a la primera de ellas se denominan *morfismos cara* y las de la segunda *morfismos de degeneración*. Los morfismos cara son la familia $\{\delta_i^n \colon \mathbf{n} \to \mathbf{n+1}; 0 \leq i \leq n+1\}$, siendo $\delta_i^n(j) = j$ si $j < i$ y $\delta_i^n(j) = j+1$ si $j \geq i$. Por su parte, los morfismos de degeneración son la familia $\{\sigma_i^n \colon \mathbf{n} \to \mathbf{n\text{-}1}; 0 \leq i \leq n-1\}$, siendo $\sigma_i^n(j) = j$ si $j \leq i$ y $\sigma_i^n(j) = j-1$ si $j > i$.

Entre los morfismos anteriores se cumplen las siguientes igualdades (en la notación se prescinde deliberadamente del objeto $\mathbf{n}$ que juega el papel de origen del morfismo):

(1)	$\delta_j \delta_i = \delta_{i+1} \delta_j$	si $i \geq j$
(2)	$\sigma_j \sigma_i = \sigma_i \sigma_{j+1}$	si $i \leq j$
(3)	$\sigma_j \delta_i = \delta_i \sigma_{j-1}$	si $i < j$
(4)	$\sigma_j \delta_i = id$	si $i = j$ ó $i = j+1$
(5)	$\sigma_j \delta_i = \delta_{i-1} \sigma_j$	si $i > j+1$

Caras y degeneraciones generan todos los morfismos de Δ^*, es decir, cualquier aplicación creciente se puede describir como composición de aplicaciones de estos dos tipos. Además, dicha factorización es única sujeta a ciertas restricciones. Así, todo morfismo $\mu \colon \mathbf{n} \to \mathbf{m}$ en Δ^*, se expresa de forma única como:

$$\mu = \delta_{j_s} \ldots \delta_{j_1} \sigma_{i_t} \ldots \sigma_{i_1}, \text{ con } 0 \leq i_t < \ldots < i_1 \text{ y } 0 \leq j_1 < \ldots < j_s$$

Esta propiedad es un caso particular de factorización monomorfismo-epimorfismo, y es la clave para el modelo simplicial que se va a desarrollar en este trabajo.

Los conjuntos simpliciales se definen como prehaces sobre Δ^* ([4] es una referencia sobre Teoría de Categorías suficiente para seguir este trabajo). Así, un *conjunto simplicial* K es un funtor contravariante $K \colon \Delta^* \to Set$, donde Set denota la categoría de conjuntos (sus objetos son los conjuntos y sus morfismos son las funciones conjuntistas). Puede pensarse entonces que un conjunto simplicial es una interpretación conjuntista de la propia categoría simplicial. Aplicando la propiedad de factorización de morfismos en Δ^* que se acaba de referir, se concluye que un conjunto simplicial K viene determinado de la siguiente forma: Un conjunto graduado $\{K_n\}_{n \in \mathbb{N}}$. Cada elemento $x \in K_n$ se denomina *n-símplice*, y se indica diciendo que x tiene *dimensión n*. En cada dimensión, hay dos familias de aplicaciones distinguidas, $\{\partial_i^n \colon K_n \to K_{n-1}; 0 < n \text{ y } 0 \leq i \leq n\}$, que se denominan *operadores cara*, y $\{\eta_i^n \colon K_n \to K_{n+1}; 0 \leq i \leq n\}$, denominados *operadores de degeneración*.

De esta forma, identificamos el conjunto simplicial (un funtor) con la imagen de Δ^* que proyecta sobre Set. Los conjuntos K_n constituyen la realización en el conjunto simplicial K de los objetos de la categoría simplicial. Los operadores cara y de degeneración constituyen la interpretación en K de los morfismos cara y degeneración, respectivamente. Desde este punto de vista, las identidades presentadas

para la categoría simplicial se entienden ahora como propiedades que deben cumplir los operadores cara y de degeneración de todo conjunto simplicial, dando lugar a la siguiente relación de igualdades (se debe tener en cuenta la contravarianza y que se prescinde de indicar la dimensión en los operadores):

$$\begin{array}{lll}
(6) & \partial_i\partial_j = \partial_j\partial_{i+1} & \text{si } i \geq j \\
(7) & \eta_i\eta_j = \eta_{j+1}\eta_i & \text{si } i \leq j \\
(8) & \partial_i\eta_j = \eta_{j-1}\partial_i & \text{si } i < j \\
(9) & \partial_i\eta_j = id & \text{si } i = j \text{ ó } i = j+1 \\
(10) & \partial_i\eta_j = \eta_j\partial_{i-1} & \text{si } i > j+1
\end{array}$$

Las igualdades anteriores son llamadas *identidades simpliciales* y, como se observa, emanan directamente de la propia categoría simplicial. Interpretadas como reglas (leídas de izquierda a derecha) de un sistema de reescritura cuyos términos son las composiciones de caras y de degeneraciones (coherentes respecto a la dimensión), dan lugar a un sistema confluyente. La forma normal de una composición de caras y de degeneraciones se obtiene al trasladar a los conjuntos simpliciales la propiedad de factorización única que se verifica en Δ^*. Así, todo endomorfismo de un conjunto simplicial que venga definido como composición de caras y de degeneraciones, admite una expresión canónica del tipo:

$$\eta_{i_1} \ldots \eta_{i_p} \partial_{j_1} \ldots \partial_{j_q}, \text{ con } 0 \leq i_p < \ldots < i_1 \text{ y } 0 \leq j_1 < \ldots < j_q$$

De aquí en adelante, se llamará *operador simplicial* a cualquier secuencia válida (respetando la dimensión) de operadores cara y de degeneración. Con ello, todo operador simplicial puede ser expresado en forma canónica. Dicha forma canónica será denominada *término simplicial*.

El objeto de la Topología Algebraica es asociar a los espacios topológicos una serie de invariantes algebraicos que permiten estudiar y caracterizar algunas de sus propiedades geométricas. Para el cálculo de los invariantes algebraicos de un conjunto simplicial el primer paso necesario es la construcción de su *complejo de cadenas asociado*. La forma de realizarlo consiste en la obtención, en cada dimensión n, del grupo abeliano libre generado por los n-símplices del conjunto simplicial de partida. Traducido al lenguaje de los términos simpliciales, algunos operadores topológicos definidos sobre el complejo de cadenas asociado a un conjunto simplicial, pueden ser presentados como combinaciones lineales (sumas formales) con coeficientes enteros de términos simpliciales. Dichas combinaciones serán denominadas *polinomios simpliciales*. Es, por ejemplo, el caso del propio morfismo diferencial, el operador borde, que se define como el polinomio $d_n = \partial_0 - \partial_1 + \ldots + (-1)^n\partial_n$.

El punto de vista que se adopta en este trabajo es que las nociones y propiedades topológicas de carácter simplicial, las cuales constituyen una parte fundamental del sistema Kenzo, pueden formularse usando únicamente operadores y polinomios simpliciales, con lo que definiciones, construcciones y demostraciones admiten una expresión en el lenguaje de los términos simpliciales, usando las identidades como reglas de reescritura.

3. Formalización del modelo simplicial

En esta sección se justifica la corrección del modelo con el que se va a trabajar en el sistema ACL2 para desarrollar partes de la Topología Simplicial. No se pretende mostrar todos los detalles sobre la formalización matemática que subyace, sino explicar las simplificaciones realizadas sobre el modelo matemático real y las razones por las que dichas simplificaciones pueden ser llevadas a cabo. El objetivo es poder obtener pruebas en ACL2 de propiedades relacionadas con construcciones propias de la Topología Simplicial que están presentes en Kenzo. Para centrar el problema, se van a tratar propiedades que responden a enunciados del tipo: *Para cualquier conjunto simplicial K, para cualquier dimensión n (mayor que una dada) y para cualquier símplice $x \in K_n$ se cumple $T(x) = T'(x)$*, donde T y T' son combinaciones lineales (sumas formales) de operadores simpliciales. O formalmente:

$$\forall K\ \forall n\ \forall x \in K_n, T(x) = T'(x)$$

El modelo adoptado en [2] para abordar pruebas en ACL2 de propiedades de ese tipo, va en la dirección de prescindir del primero de los tres cuantificadores universales que aparecen en dichos enunciados. Para ello, se usa la existencia de un conjunto simplicial universal, que en la literatura se denota habitualmente por Δ. Aunque admite otras descripciones (la que se da justo a continuación difiere levemente de la usada en [2]), se puede considerar que los n-símplices en Δ son listas crecientes de números naturales de longitud $n+1$. Respecto de los operadores, la cara i-ésima de un símplice es la lista que resulta de eliminar el natural que ocupa la posición i (comenzando los índices en 0), mientras que la degeneración i-ésima de un símplice es la lista que resulta de duplicar el natural que ocupa la posición i. Su carácter universal como conjunto simplicial se traduce en el hecho de que cualquier propiedad válida en Δ, y que sólo dependa de las identidades simpliciales, es necesariamente válida en cualquier otro conjunto simplicial (Δ posee el menor número de identificaciones entre símplices, sólo posee las inducidas por las identidades). Por tanto, es suficiente realizar pruebas en Δ de los enunciados del tipo planteado. En el lenguaje de la Teoría de Categorías, este resultado interpreta el hecho de que Δ es un cierto colímite de los prehaces representables de la categoría simplicial, lo que permite extender a Δ los morfismos que representan los símplices de los conjuntos simpliciales.

La siguiente simplificación que se plantea es la no utilización de elementos, lo que lleva implícito no usar la evaluación de los operadores simpliciales. Así, el sistema de reescritura definido por las identidades simpliciales va a ser la herramienta de la que emanen todos los patrones de prueba. En el presente trabajo todavía se lleva a cabo una tercera simplificación del modelo matemático de partida, prescindir de la dimensión. Aunque la notación utilizada puede llevar a engaño, cada una de las identidades simpliciales no es una única regla, sino que representa a toda una familia numerable de reglas (una en cada dimensión). Por ejemplo, la primera de las reglas, "$\partial_i\partial_j = \partial_j\partial_{i+1}$", representa al conjunto:

$$\partial_i^{n-1}\partial_j^n = \partial_j^{n-1}\partial_{i+1}^n \text{ con } 0 \leq j \leq i < n.$$

Así, la validez de una prueba basada en reescritura deberá incorporar la justificación de su propia corrección respecto a la dimensión. En principio, lo natural es incorporar una noción de tamaño de una prueba, es decir, una dimensión a partir de la cual la prueba formal define una verdadera demostración sobre el modelo matemático de partida. Sobre esta base se desarrolla una infraestructura en ACL2 que resulta adecuada para la representación de nociones y construcciones de tipo simplicial y permite la realización de demostraciones que certifiquen la corrección de enunciados referidos a ellas.

4. El sistema ACL2

ACL2 [9] es simultáneamente un lenguaje de programación, una lógica computacional y un demostrador de teoremas. Este sistema constituye un entorno en el que se pueden definir y ejecutar algoritmos, se pueden especificar formalmente sus propiedades, y se pueden demostrar éstas últimas con la ayuda de un sistema de demostración automatizado.

Como lenguaje de programación, ACL2 es una extensión de un subconjunto aplicativo de Common Lisp, sin efectos secundarios, de primer orden y sin variables globales. Por otro lado, las primitivas básicas ACL2 se comportan exactamente de la misma manera que en Common Lisp. De esta forma, cualquier programa ACL2 puede ser ejecutado en cualquier entorno Common Lisp estándar extendido con un pequeño conjunto de funciones.

La lógica de ACL2 es una lógica de primer orden, en la que las fórmulas se escriben en notación prefija y en la que las variables de las fórmulas están cuantificadas universalmente de forma implícita. Las conectivas proposicionales vienen definidas por las funciones `implies`, `and`, `or` y `not`, y la función `equal` define la igualdad. Entre los axiomas de la lógica se encuentran los habituales de la lógica proposicional, de la igualdad y otros que describen el comportamiento de las primitivas básicas Common Lisp soportadas por ACL2. Sus reglas de inferencia incluyen la instanciación de variables de una fórmula y un principio de demostración de teoremas por inducción. Las definiciones de funciones introducidas por el usuario (mediante `defun`) se consideran axiomas en la lógica (lo que se conoce como *principio de definición*). Por ese motivo, y para mantener la consistencia, antes de que una definición sea aceptada debe ser demostrado que termina para todo dato de entrada.

El *principio de encapsulado* (mediante `encapsulate`) es otra manera, alternativa al principio de definición, de introducir nuevas funciones en la lógica. En este caso, una función no se introduce definiéndola completamente, sino a través de una serie de propiedades que se asumen sobre la misma. Para mantener la consistencia, se debe asegurar que al menos existe una *función testigo* que verifica dichas propiedades. Dentro de un encapsulado, estas propiedades se demuestran para el testigo, pero fuera del mismo actúan como propiedades generales asumidas sobre la función que se introduce en el encapsulado. Una regla de inferencia derivada, la *instanciación funcional*, permite simular el razonamiento de segundo orden en ACL2: una vez demostrado un teorema sobre una función introducida mediante encapsulado, es posible instanciar ese resultado sustituyendo dicha función por

otra, siempre que se demuestre que ésta última cumple las mismas propiedades que se asumieron para la función sustituida.

Como demostrador automático, ACL2 es un asistente para la demostración de teoremas en la lógica. Básicamente, el usuario introduce una fórmula en el mismo (mediante `defthm`) y esto inicia un intento de demostración. Si el intento concluye con éxito, el teorema es almacenado y se puede usar en la demostración de nuevos teoremas. Las principales técnicas que el demostrador utiliza para demostrar un teorema son simplificación por reescritura e inducción. Las reglas de simplificación se obtienen a partir de las definiciones de funciones y de los teoremas previos demostrados por el usuario.

Aunque el demostrador de ACL2 es automático (en el sentido de que una vez comienza un intento de demostración, no hay interacción con el usuario), en realidad se trata de un demostrador interactivo en cierto sentido. Usualmente, el primer intento de demostración de un resultado no trivial es fallido, y es tarea del usuario el encontrar un conjunto de definiciones y lemas que hagan finalmente que el intento de demostración termine con éxito. Estos lemas suelen sugerirse a partir de una demostración a mano ya conocida o de la inspección de un intento de prueba fallido. A su vez, la demostración de estos lemas puede requerir también este tipo de interacción, y por tanto necesitar de nuevos lemas. Como consecuencia, el producto final de una formalización suele ser un conjunto de definiciones y lemas que proporcionan las suficientes herramientas de simplificación para que los teoremas finales puedan demostrarse. Esta manera de interactuar con el sistema para obtener una demostración es la que los autores del sistema llaman "El Método".

En lo que sigue, se presenta una teoría de los términos y polinomios simpliciales en ACL2. Es decir, las definiciones que formalizan estos conceptos, y los teoremas demostrados sobre ellos. Para obtener todas las demostraciones en ACL2, se ha interactuado con el sistema siguiendo "El Método". No se incluyen detalles técnicos sobre esta interacción, ya que la exposición estará más centrada en las cuestiones relativas a la formalización.

5. Formalización de polinomios simpliciales en ACL2

En esta sección se presentará la formalización en ACL2 de la noción de polinomio simplicial y se demostrará que el conjunto de polinomios simpliciales con las operaciones de suma y composición tiene estructura de anillo. La presentación de la formalización realizada se hará de forma progresiva. En primer lugar se introducen las definiciones y propiedades del monoide de los términos simpliciales. Después de esto, y antes de definir los polinomios simpliciales, se define una teoría general y abstracta sobre el conjunto de las combinaciones lineales de los elementos de un monoide, con las operaciones de suma y la extensión a combinaciones lineales del operador del monoide. De esta forma, en segundo lugar se presenta la formalización de esta situación más general y la demostración de que el resultado tiene estructura de anillo. Finalmente, el anillo de los polinomios simpliciales se obtiene como una instancia funcional de la formalización del anillo de las combinaciones lineales.

Debido a las limitaciones de espacio, aquí sólo aparecerán las definiciones y propiedades más relevantes de nuestra formalización. El contenido completo de la misma se puede consultar en la página Web
`http://www.glc.us.es/fmartin/acl2/simplicialtopology`

5.1. **Términos simpliciales.** La caracterización de los términos simpliciales vista en la sección 2, permite la siguiente definición. Un término simplicial en ACL2 es una lista formada por una lista estrictamente decreciente de números naturales, representando una secuencia de operadores de degeneración, y una lista estrictamente creciente de números naturales, representando una secuencia de operadores cara. Para formalizar este concepto en ACL2 se utilizan tres funciones: `ld-p`, que comprueba que su argumento es una lista estrictamente creciente de números naturales; `ln-p`, que comprueba que su argumento es una lista estrictamente decreciente de números naturales; y `st-p`, que comprueba que su argumento es un término simplicial tal y como se acaba de describir:

```
(defun ld-p (ld)
  (cond ((endp ld) (equal ld nil))
        ((endp (cdr ld)) (and (equal (cdr ld) nil)
                              (natp (car ld))))
        (t (and (natp (car ld))
                (natp (cadr ld))
                (< (car ld) (cadr ld))
                (ld-p (cdr ld))))))

(defun ln-p (ln)
  (cond ((endp ln) (equal ln nil))
        ((endp (cdr ln)) (and (equal (cdr ln) nil)
                              (natp (car ln))))
        (t (and (natp (car ln))
                (natp (cadr ln))
                (> (car ln) (cadr ln))
                (ln-p (cdr ln))))))

(defun st-p (st)
  (and (consp st)
       (consp (cdr st))
       (equal (cddr st) nil)
       (ln-p (first st))
       (ld-p (second st))))
```

La *composición* de términos simpliciales se define como la composición de los operadores que los componen: $\langle [i_1, ..., i_{p_1}], [j_1, ..., j_{q_1}] \rangle \circ \langle [i'_1, ..., i'_{p_2}], [j'_1, ..., j'_{q_2}] \rangle = \eta_{i_1} \dots \eta_{i_{p_1}} \partial_{j_1} \dots \partial_{j_{q_1}} \eta_{i'_1} \dots \eta_{i'_{p_2}} \partial_{j'_1} \dots \partial_{j'_{q_2}}$, que aplicando las identidades simpliciales puede ser transformado de nuevo en un término simplicial. Por ejemplo, la composición de los términos simpliciales $\langle [2,1], [3] \rangle$ y $\langle [1], [2,4] \rangle$ es $\eta_2\eta_1\partial_3\eta_1\partial_2\partial_4$, y aplicando repetidamente las identidades simpliciales obtenemos $\eta_2\eta_1\eta_1\partial_2\partial_2\partial_4$, $\eta_2\eta_2\eta_1\partial_2\partial_2\partial_4$, $\eta_3\eta_2\eta_1\partial_2\partial_2\partial_4$ y finalmente $\eta_3\eta_2\eta_1\partial_2\partial_3\partial_4$, que es el término simplicial $\langle [3,2,1], [2,3,4] \rangle$. Como se demostrará, el conjunto de los términos simpliciales con esta operación de composición tiene estructura de monoide.

La definición en ACL2 del operador de composición de términos simpliciales es bastante compleja, dado que hay que aplicar las identidades simpliciales para mantener el resultado en forma normal. Para ello se procede como se describe a continuación: la composición de los términos simpliciales $\langle \eta_{s_1}, \partial_{s_1} \rangle$ y $\langle \eta_{s_2}, \partial_{s_2} \rangle$ es $\eta_{s_1}\partial_{s_1}\eta_{s_2}\partial_{s_2}$; se aplican primero las identidades simpliciales (8), (9) y (10) para

componer $\partial_{s_1}\eta_{s_2}$ y obtener así una lista estrictamente decreciente de operadores de degeneración η_{s_3} y una lista estrictamente creciente de operadores cara ∂_{s_3} tales que $\partial_{s_1}\eta_{s_2} = \eta_{s_3}\partial_{s_3}$; a continuación se aplica la identidad simplicial (6) para componer $\partial_{s_3}\partial_{s_2}$ obteniendo una lista estrictamente creciente de operadores cara ∂_{s_4}; finalmente basta aplicar la identidad simplicial (7) para componer $\eta_{s_1}\eta_{s_3}$ obteniendo una lista estrictamente decreciente de operadores cara η_{s_4}. El resultado de la composición de los términos simpliciales iniciales es $\langle \eta_{s_4}, \partial_{s_4} \rangle$.

Las funciones `ln-cmp-d-ln` y `ln-cmp-ld-ln` implementan la primera parte del proceso de composición de una lista estrictamente creciente de operadores cara (∂_{s_1} en la descripción del proceso) con una lista estrictamente decreciente de operadores de degeneración (η_{s_2} en la descripción del proceso), aplicando las identidades simpliciales (8), (9) y (10) para obtener como resultado la lista estrictamente decreciente de operadores de degeneración, primera componente del resultado de la composición (η_{s_3} en la descripción del proceso).

```
(defun ln-cmp-d-ln (d ln)
  (cond ((endp ln) nil)
        ((< d (car ln)) (cons (1- (car ln)) (ln-cmp-d-ln d (cdr ln))))
        ((> d (1+ (car ln))) (cons (car ln)
                                   (ln-cmp-d-ln (1- d) (cdr ln))))
        (t (cdr ln))))

(defun ln-cmp-ld-ln (ld ln)
  (cond ((endp ld) ln)
        (t (ln-cmp-d-ln (car ld) (ln-cmp-ld-ln (cdr ld) ln)))))
```

Las funciones `ld-cmp-d-ln` y `ld-cmp-ld-ln` implementan la segunda parte del proceso de composición de una lista estrictamente creciente de operadores cara (∂_{s_1} en la descripción del proceso) con una lista estrictamente decreciente de operadores de degeneración (η_{s_2} en la descripción del proceso), aplicando las identidades simpliciales (8), (9) y (10) para obtener como resultado la lista estrictamente creciente de operadores cara, segunda componente del resultado de la composición (∂_{s_3} en la descripción del proceso).

```
(defun ld-cmp-d-ln (d ln)
  (cond ((endp ln) (list d))
        ((< d (car ln)) (ld-cmp-d-ln d (cdr ln)))
        ((> d (1+ (car ln))) (ld-cmp-d-ln (1- d) (cdr ln)))
        (t nil)))

(defun ld-cmp-ld-ln (ld ln)
  (cond ((endp ld) nil)
        (t (cmp-ld-ld (ld-cmp-d-ln (car ld) (ln-cmp-ld-ln (cdr ld) ln))
                      (ld-cmp-ld-ln (cdr ld) ln)))))
```

Las funciones `cmp-d-ld` y `cmp-ld-ld` implementan el proceso de composición de listas estrictamente crecientes de operadores cara (∂_{s_3} y ∂_{s_2} en la descripción del proceso), aplicando la identidad simplicial (6) para obtener como resultado una lista estrictamente creciente de operadores cara:

```
(defun cmp-d-ld (d ld)
  (cond ((endp ld) (list d))
        ((<= (car ld) d) (cons (car ld) (cmp-d-ld (1+ d) (cdr ld))))
        (t (cons d ld))))

(defun cmp-ld-ld (ld1 ld2)
  (cond ((endp ld1) ld2)
        (t (cmp-d-ld (car ld1) (cmp-ld-ld (cdr ld1) ld2)))))
```

Las funciones `cmp-n-ln` y `cmp-ln-ln` implementan el proceso de composición de listas estrictamente decrecientes de operadores de degeneración (η_{s_1} y η_{s_3} en la descripción del proceso), aplicando la identidad simplicial (7) para obtener como resultado una lista estrictamente decreciente de operadores de degeneración:

```
(defun cmp-n-ln (n ln)
  (cond ((endp ln) (list n))
        ((<= n (car ln)) (cons (1+ (car ln)) (cmp-n-ln n (cdr ln))))
        (t (cons n ln))))

(defun cmp-ln-ln (ln1 ln2)
  (cond ((endp ln1) ln2)
        (t (cmp-n-ln (car ln1) (cmp-ln-ln (cdr ln1) ln2)))))
```

Finalmente la composición de términos simpliciales se implementa con la función `cmp-st-st`, que aplica las funciones anteriores de la forma descrita anteriormente para obtener como resultado un nuevo término simplicial:

```
(defun cmp-st-st (st1 st2)
  (list (cmp-ln-ln (first st1) (ln-cmp-ld-ln (second st1) (first st2)))
        (cmp-ld-ld (ld-cmp-ld-ln (second st1) (first st2)) (second st2))))
```

Puesto que la composición de términos simpliciales se realiza aplicando las identidades simpliciales para mantener la forma normal, el resultado es a su vez un término simplicial. Es decir, la composición es una operación interna en el conjunto de los términos simpliciales. La demostración de esta propiedad supone probar que las funciones auxiliares anteriores devuelven como resultado una lista de operadores con el orden correcto, es decir, estrictamente creciente para los operadores cara y estrictamente decreciente para los operadores de degeneración.

```
(defthm st-p-cmp-st-st
  (implies (and (st-p t1)
                (st-p t2))
           (st-p (cmp-st-st t1 t2))))
```

El elemento unidad del monoide de los términos simpliciales es la composición vacía de operadores cara y degeneración. En nuestra formalización este elemento se construye con la función `st-id` y sus propiedades se demuestran fácilmente:

```
(defun st-id ()
  (list nil nil))

(defthm st-id-st-p
  (st-p (st-id)))

(defthm cmp-st-st-identity
  (implies (st-p t1)
           (and (equal (cmp-st-st t1 (st-id)) t1)
                (equal (cmp-st-st (st-id) t1) t1))))
```

La composición de términos simpliciales es asociativa, como establece el siguiente teorema:

```
(defthm cmp-st-st-associative
  (implies (and (st-p t1)
                (st-p t2)
                (st-p t3))
           (equal (cmp-st-st (cmp-st-st t1 t2) t3)
                  (cmp-st-st t1 (cmp-st-st t2 t3)))))
```

Esta propiedad resulta bastante difícil de demostrar, dado que la definición de la operación de composición se basa en 8 funciones auxiliares. En concreto, han sido

necesarios un total de 19 lemas auxiliares y 5 esquemas de inducción explícitos, no descubiertos por el demostrador.

En resumen, las propiedades anteriores establecen que el conjunto de los términos simpliciales, caracterizados por la función `st-p`, es un monoide con respecto a la operación de composición implementada por la función `cmp-st-st`.

5.2. **El anillo de las combinaciones lineales.** Dado un monoide $(T, \circ, \mathbf{1})$, sea P el conjunto de las combinaciones lineales con coeficientes enteros de los elementos de T. Los elementos de P están formados por "sumandos" (o "monomios"), pares formados por un coeficiente entero no nulo y un elemento de T. Denotaremos este conjunto como M. En el conjunto P se pueden definir una operación de suma $+$ que agrupe los elementos de M cuyos elementos de T asociados sean iguales, sumando sus coeficientes. La combinación lineal vacía, $\mathbf{0}$, actúa como elemento neutro con respecto a $+$. Se puede definir también una operación de composición entre combinaciones lineales como una extensión natural de la operación $\circ$ cumpliendo la propiedad distributiva con respecto a $+$. En esta situación se tiene que $(P, +, \circ, \mathbf{0}, \mathbf{1})$ es un anillo.

Este resultado se puede formalizar en el sistema ACL2 como una teoría genérica [11] en la que a partir de un conjunto de funciones y propiedades que caracterizan de forma general y abstracta un monoide, se definen las funciones y se demuestran las propiedades que aseguran que el conjunto de las combinaciones lineales es un anillo. Una vez hecho esto, esta teoría genérica se puede instanciar para casos particulares de monoides, como es el caso del monoide de los términos simpliciales.

Un elemento de P se representa fácilmente en ACL2 como una lista de pares formados por un coeficiente entero no nulo y un elemento del monoide T. Para evitar que un mismo elemento de P tenga varias representaciones distintas en ACL2, se exigirá que la lista que lo representa esté ordenada con respecto a un orden total definido sobre el conjunto T, lo cual supone una restricción adicional sobre dicho monoide (lo que llamaremos *monoide ordenado*).

De esta forma, el punto de partida de la teoría genérica del anillo de las combinaciones lineales es la introducción, mediante un encapsulado, de una serie de funciones que definen un monoide ordenado genérico: una función `ter-p` que caracteriza los elementos de T, una función `cmp-ter-ter` que implementa la operación $\circ$ del monoide, una función `ter-id` que construye el elemento unidad del monoide, y un orden total `ter-<` entre los elementos de T. Las únicas propiedades asumidas sobre estas funciones son las propias de monoide ordenado:

```
(encapsulate
  ...

  (defthm ter-p-cmp-ter-ter
    (implies (and (ter-p t1)
                  (ter-p t2))
             (ter-p (cmp-ter-ter t1 t2))))

  (defthm cmp-ter-ter-associative
    (implies (and (ter-p t1)
                  (ter-p t2)
                  (ter-p t3))
             (equal (cmp-ter-ter (cmp-ter-ter t1 t2) t3)
                    (cmp-ter-ter t1 (cmp-ter-ter t2 t3))))))
```

```
(defthm ter-id-ter-p
  (ter-p (ter-id)))

(defthm cmp-ter-ter-identity
  (implies (ter-p t1)
           (and (equal (cmp-ter-ter t1 (ter-id)) t1)
                (equal (cmp-ter-ter (ter-id) t1) t1))))

(defthm ter-<-irreflexive
  (not (ter-< t1 t1)))

(defthm ter-<-transitive
  (implies (and (ter-< t1 t2)
                (ter-< t2 t3))
           (ter-< t1 t3)))

(defthm ter-<-total
  (implies (and (not (ter-< t1 t2))
                (not (ter-< t2 t1)))
           (equal t1 t2)))
```

A partir de aquí se definen en ACL2 las caracterizaciones de los elementos de M y P. Los elementos de M se representan como una lista formada por dos elementos, un número entero no nulo y un elemento de T; la función `mon-p` caracteriza dicho conjunto. La relación de orden definida en T se extiende de forma natural al conjunto M mediante la función `mon-<`. Finalmente los elementos de P son representados como una lista ordenada de elementos de M; la función `pol-p` caracteriza dicho conjunto:

```
(defun mon-p (m)
  (and (consp m)
       (integerp (car m))
       (not (equal (car m) 0))
       (ter-p (cdr m))))

(defun mon-< (m1 m2)
  (ter-< (cdr m1) (cdr m2)))

(defun pol-p (p)
  (cond ((endp p) (equal p nil))
        ((endp (cdr p))
         (and (mon-p (car p))
              (equal (cdr p) nil)))
        (t (and (mon-p (car p))
                (mon-< (first p) (second p))
                (pol-p (cdr p))))))
```

La suma de elementos de P se define como la aplicación recursiva del proceso de suma de un elemento de M a un elemento de P. Dado que la representación de los elementos de P es mediante listas ordenadas, la suma de un elemento de M con un elemento de P tiene en cuenta esta característica para mantener el resultado ordenado:

```
(defun add-mon-pol (m p)
  (cond ((endp p) (list m))
        ((mon-< m (car p))
         (cons m p))
        ((mon-< (car p) m)
         (cons (car p) (add-mon-pol m (cdr p))))
        (t (if (zerop (+ (car m) (car (car p))))
               (cdr p)
               (cons (cons (+ (car m) (car (car p))) (cdr m))
                     (cdr p))))))
```

```
(defun add-pol-pol (p1 p2)
  (cond ((endp p1) p2)
        (t (add-mon-pol (car p1) (add-pol-pol (cdr p1) p2)))))
```

El elemento neutro de la suma es la lista vacía, y el inverso de una combinación lineal se obtiene cambiando el signo de los coeficientes de los sumandos que la componen:

```
(defun add-pol-pol-id ()
  nil)

(defun neg-mon (m)
  (cons (- (car m)) (cdr m)))

(defun inverse-add-pol-pol (p)
  (if (endp p)
      p
      (cons (neg-mon (car p))
            (inverse-add-pol-pol (cdr p)))))
```

La operación $\circ$ sobre T se extiende a M multiplicando los coeficientes enteros y componiendo los elementos de T. La función `cmp-mon-mon` implementa esta operación en ACL2:

```
(defun cmp-mon-mon (m1 m2)
  (cons (* (car m1) (car m2))
        (cmp-ter-ter (cdr m1) (cdr m2))))
```

La composición de elementos de P se define como la aplicación recursiva del proceso de composición de un elemento de M con un elemento de P y la suma de los resultados. La función `cmp-mon-pol` implementa la composición de un elemento de M con un elemento de P y la función `cmp-pol-pol` implementa la composición de dos elementos de P:

```
(defun cmp-mon-pol (m p)
  (cond ((endp p) nil)
        (t (add-mon-pol (cmp-mon-mon m (car p))
                        (cmp-mon-pol m (cdr p))))))

(defun cmp-pol-pol (p1 p2)
  (cond ((endp p1) nil)
        (t (add-pol-pol (cmp-mon-pol (car p1) p2)
                        (cmp-pol-pol (cdr p1) p2)))))
```

Finalmente, el elemento unidad de la composición de elementos de P es la lista formada por un único sumando formado por el coeficiente 1 y el elemento unidad de la composición definida en T:

```
(defun mon-id ()
  (cons 1 (ter-id)))

(defun cmp-pol-pol-id ()
  (list (mon-id)))
```

A partir de estas definiciones se han demostrado las propiedades que aseguran que el conjunto de las combinaciones lineales, caracterizadas por la función `pol-p`, es un anillo con respecto a la operación de suma implementada por la función `add-pol-pol` y la operación de composición implementada por la función `cmp-pol-pol`.

- Las operaciones son internas en el conjunto caracterizado por `pol-p`:

```
(defthm add-pol-pol-id-pol-p
  (pol-p (add-pol-pol-id)))

(defthm inverse-add-pol-pol-pol-p
  (implies (pol-p p)
           (pol-p (inverse-add-pol-pol p))))

(defthm pol-p-add-pol-pol
  (implies (and (pol-p p1)
                (pol-p p2))
           (pol-p (add-pol-pol p1 p2))))

(defthm cmp-pol-pol-id-pol-p
  (pol-p (cmp-pol-pol-id)))

(defthm pol-p-cmp-pol-pol
  (implies (and (pol-p p1)
                (pol-p p2))
           (pol-p (cmp-pol-pol p1 p2))))
```

- La operación `add-pol-pol` es conmutativa y asociativa:

```
(defthm add-pol-pol-commutative
  (implies (and (pol-p p1)
                (pol-p p2))
           (equal (add-pol-pol p1 p2)
                  (add-pol-pol p2 p1))))

(defthm add-pol-pol-associative
  (implies (and (pol-p p1)
                (pol-p p2)
                (pol-p p3))
           (equal (add-pol-pol (add-pol-pol p1 p2) p3)
                  (add-pol-pol p1 (add-pol-pol p2 p3)))))
```

- El elemento `add-pol-pol-id` es el elemento neutro con respecto a `add-pol-pol`:

```
(defthm add-pol-pol-id-identity
  (implies (pol-p p)
           (equal (add-pol-pol p (add-pol-pol-id))
                  p)))
```

- La función `inverse-add-pol-pol` produce el inverso, con respecto a `add-pol-pol`, de una combinación lineal:

```
(defthm inverse-add-pol-pol-inverse
  (implies (pol-p p)
           (equal (add-pol-pol p (inverse-add-pol-pol p))
                  (add-pol-pol-id))))
```

- El elemento `cmp-pol-pol-id` es el elemento unidad con respecto a `cmp-pol-pol`:

```
(defthm cmp-pol-pol-id-left-identity
  (implies (pol-p p)
           (equal (cmp-pol-pol (cmp-pol-pol-id) p)
                  p)))

(defthm cmp-pol-pol-id-right-identity
  (implies (pol-p p)
           (equal (cmp-pol-pol p (cmp-pol-pol-id))
                  p)))
```

- La operación `cmp-pol-pol` es asociativa:

```
(defthm cmp-pol-pol-associative
  (implies (and (pol-p p1)
```

```
                (pol-p p2)
                (pol-p p3))
           (equal (cmp-pol-pol (cmp-pol-pol p1 p2) p3)
                  (cmp-pol-pol p1 (cmp-pol-pol p2 p3)))))
```

- La operación `cmp-pol-pol` es distributiva con respecto a `add-pol-pol`, a izquierda y a derecha:

```
(defthm cmp-pol-pol-add-pol-pol-distributive-l
  (implies (and (pol-p p1)
                (pol-p p2)
                (pol-p p3))
           (equal (cmp-pol-pol (add-pol-pol p1 p2) p3)
                  (add-pol-pol (cmp-pol-pol p1 p3)
                               (cmp-pol-pol p2 p3)))))

(defthm cmp-pol-pol-add-pol-pol-distributive-r
  (implies (and (pol-p p1)
                (pol-p p2)
                (pol-p p3))
           (equal (cmp-pol-pol p1 (add-pol-pol p2 p3))
                  (add-pol-pol (cmp-pol-pol p1 p2)
                               (cmp-pol-pol p1 p3)))))
```

Además de estas propiedades se han demostrado otras que facilitan el proceso de reescritura usado por el demostrador ACL2. De esta forma la teoría genérica sobre combinaciones lineales proporciona un total de 28 propiedades.

5.3. **Polinomios simpliciales.** A continuación, la formalización de los polinomios simpliciales se obtiene como una instancia de la teoría genérica de combinaciones lineales presentada en la subsección anterior. Para ello, en dicha teoría genérica se utiliza la macro `def-generic-theory`, que recibe como argumentos una cadena *nombre* identificando la teoría genérica, un bloque de *eventos genéricos* (en este caso, funciones y propiedades que caracterizan un monoide genérico con un orden total) y un conjunto de *eventos exportados* (en este caso, funciones y propiedades que formalizan la estructura genérica de anillo de las combinaciones lineales). El efecto de esta macro es el de definir la macro `definstance-`*nombre* que de forma automática construye instancias concretas de todos los eventos exportados a partir de una instancia de los eventos genéricos. Para más detalles sobre este proceso, se puede consultar [11].

Para poder instanciar la teoría genérica de las combinaciones lineales y obtener el caso concreto de los polinomios simpliciales, se usarán las funciones definidas en la subsección 5.1, que definían una estructura de monoide en el conjunto de términos simpliciales. Además se necesita definir un orden total sobre el conjunto de los términos simpliciales. Para ello se considera el orden lexicográfico, tal y como define la función `st-<`:

```
(defun st-< (t1 t2)
  (and (lexorder t1 t2)
       (not (equal t1 t2))))

(defthm st-<-irreflexive
  (not (st-< t1 t1)))

(defthm st-<-transitive
  (implies (and (st-< t1 t2)
                (st-< t2 t3))
           (st-< t1 t3)))
```

```
(defthm st-<-total
  (implies (and (not (st-< t1 t2))
                (not (st-< t2 t1)))
           (equal t1 t2)))
```

Finalmente, la formalización del anillo de los polinomios simpliciales se obtiene con la siguiente instrucción, en la que se establece la correspondencia entre los elementos genéricos que caracterizan un monoide ordenado y las funciones concretas que formalizan el monoide ordenado de los términos simpliciales:

```
(definstance-*polynomials*
  ((ter-p st-p)
   (cmp-ter-ter cmp-st-st)
   (ter-< st-<)
   (ter-id st-id))
  :name-subst (("MON" "SM")
               ("POL" "SP")))
```

El argumento adicional `:name-subst` proporciona una forma de renombrar los eventos exportados de la teoría genérica para construir los correspondientes en la teoría instanciada, reemplazando las ocurrencias de `mon` y `pol` por `sm` y `sp` respectivamente. De esta forma, los monomios simpliciales están caracterizados por la función `sm-p`, los polinomios simpliciales por la función `sp-p`, las operaciones de suma y composición de un monomio simplicial con un polinomio simplicial están implementadas por las funciones `add-sm-sp` y `cmp-sm-sp`; y las operaciones de suma y composición de polinomios simpliciales por las funciones `add-sp-sp` y `cmp-sp-sp`.

Es importante destacar que la llamada a la instrucción anterior va a generar, de manera completamente automática, las definiciones de las funciones anteriores, junto con demostraciones, por instanciación funcional, de las propiedades que formalizan el hecho de que el conjunto de los polinomios simpliciales es un anillo respecto a la suma y composición.

6. Una aplicación del anillo de los polinomios simpliciales

En esta sección se usa la infraestructura desarrollada en la sección anterior, para obtener una prueba en ACL2 de una propiedad que es esencial en Topología Simplicial. Para computar los grupos de homología de un conjunto simplicial K, el primer paso necesario consiste en la construcción del complejo de cadenas asociado a K, que denotaremos $C_*(K)$. En cada dimensión n, el correspondiente grupo $C_n(K)$ es el grupo abeliano libre generado por el conjunto K_n. El operador diferencial sobre este complejo se define en cada dimensión, como la función $d_n\colon C_n(K) \longrightarrow C_{n-1}(K)$ tal que $d_n(x) = \sum_{i=0}^{n}(-1)^i\partial_i(x)$, cuando $x \in K_n$, extendiéndose de manera lineal a las combinaciones lineales de generadores.

A continuación formalizamos en ACL2 una demostración de que $d \equiv \{d_n\}_{n\geq 0}$ define un verdadero morfismo diferencial; es decir, para cada $n \geq 0$, se cumple $d_n d_{n+1} = 0$. Para ello, representamos las funciones d_n en ACL2 como polinomios simpliciales. Así, primero definimos la función auxiliar `d-n` que construye el polinomio $(-1)^n\partial_n$, donde `cmp-scalar-sp` multiplica un escalar por un polinomio simplicial y `di` construye el polinomio simplicial ∂_n. A partir de esta, y usando la

función `add-sp-sp` que suma dos polinomio simpliciales, se define por recursión la función `d` que construye en ACL2 el polinomio simplicial que representa a d_n:

```
(defun d-n (n)
  (cmp-scalar-sp (if (evenp n) 1 -1) (di n)))

(defun d (n)
  (cond ((zp n) (di 0))
        (t (add-sp-sp (d-n n) (d (1- n))))))
```

El siguiente teorema demuestra que la composición de `(d (1+ n))` con `(d n)` devuelve el polinomio nulo:

```
(defthm cmp-d-d=0
  (implies (and (natp n)
                (< 0 n))
           (equal (cmp-sp-sp (d n) (d (1+ n)))
                  (add-pol-pol-id))))
```

Este resultado se demuestra usando las propiedades de anillo de los polinomios simpliciales, junto con las específicas de la composición de términos simpliciales.

7. Conclusiones y futuras líneas de trabajo

En este trabajo se ha propuesto un modelo que permite desarrollar partes de la Topología Simplicial en el sistema ACL2. Dicho modelo está inspirado en la construcción de los conjuntos simpliciales como prehaces sobre una categoría y conduce a las nociones de término simplicial y polinomio simplicial. Permite trabajar con objetos de la Topología Simplicial de un modo puramente formal, trasladando definiciones, construcciones y propiedades topológicas a operaciones y propiedades en un anillo de polinomios. En el trabajo se desarrolla la correspondiente teoría de los polinomios simpliciales en ACL2 y como aplicación, se muestra el uso de dicha infraestructura para probar una sencilla propiedad del ámbito de la Topología Simplicial.

Ya que esta línea de trabajo proviene del interés en tratar la corrección del sistema Kenzo, una parte del trabajo futuro debe ir encaminado a aprovechar la infraestructura aquí presentada para la prueba de resultados que impliquen la corrección de algunas construcciones presentes en Kenzo. Un primer reto puede ser establecer la corrección de la reducción explícita entre un complejo de cadenas y su normalizado. Desde el punto de vista de la formalización, otra dirección en la investigación futura consistirá en plantear posibles ampliaciones del modelo, pretendiendo abarcar un mayor espectro de construcciones topológicas.

Referencias

[1] M. Andrés, L. Lambán, J. Rubio. Executing in Common Lisp, Proving in ACL2. *Lecture Notes in Artificial Intelligence, Calculemus 2007* **4573**, 1–12, 2007.

[2] M. Andrés, L. Lambán, J. Rubio, J. L. Ruiz-Reina. Formalizing Simplicial Topology in ACL2. En *ACL2 Workshop 2007*, pp. 34–39. Austin (USA), 2007.

[3] J. Aransay, C. Ballarin, J. Rubio. A Mechanized Proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning* **40**, 271–292, 2008.

[4] M. Barr, B. Wells. *Category Theory for Computing Science*. Prentice Hall, 1995.

[5] C. Domínguez. Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. *Lecture Notes in Artificial Intelligence, Calculemus 2008* **5144**, 270–284, 2008.

[6] C. Domínguez, L. Lambán, J. Rubio. Object Oriented Institutions to Specify Symbolic Computation Systems. *Rairo - Theoretical Informatics and Applications* **41**, 191–214, 2007.
[7] X. Dousson, F. Sergeraert, Y. Siret. *The Kenzo Program*. Institut Fourier, 1999. `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`.
[8] M. Kaufmann, P. Manolios, J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
[9] M. Kaufmann, J. S. Moore. *ACL2 Home Page*. `http://www.cs.utexas.edu/users/moore/acl2`.
[10] L. Lambán, V. Pascual, J. Rubio. An Object-Oriented Interpretation of the EAT System. *Applicable Algebra in Engineering, Communication and Computing* **14**, 187–215, 2003.
[11] F. J. Martín–Mateos, J. A. Alonso, M. J. Hidalgo, J. L. Ruiz–Reina. A generic instantiation tool and a case study: A generic multiset theory. 3rd International Workshop on the ACL2 Theorem Prover and Its Applications, 2002.
[12] F. J. Martín–Mateos, J. L. Ruiz-Reina, L. Lambán, J. Rubio. Verificación y eficiencia para el cálculo simbólico: un caso de estudio. En *PROLE 2009*; Lucio, Moreno, Peña (eds.), pp. 7–14. San Sebastian, 2009.
[13] J. P. May. *Simplicial Objects in Algebraic Topology*. Van Nostrand, 1967.
[14] J. Rubio, F. Sergeraert, Y. Siret. *EAT: Symbolic Software for Effective Homology Computation*. Institut Fourier, 1997. `http://www-fourier.ujf-grenoble.fr/~sergerar/`.

Dept. de Matemáticas y Computación, Universidad de La Rioja, Edificio Vives, Luis de Ulloa s/n. 26004 Logroño, Spain
Correo electrónico: `lalamban@unirioja.es`

Grupo de Lógica Computacional, Dept. de Ciencias de la Computación e Inteligencia Artificial, Universidad de Sevilla, E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
Correo electrónico: `fjesus@us.es`

Grupo de Lógica Computacional, Dept. de Ciencias de la Computación e Inteligencia Artificial, Universidad de Sevilla, E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
Correo electrónico: `jruiz@us.es`