Demostración asistida por ordenador*

por

Jesús María Aransay Azofra y César Domínguez Pérez

Introducción

Actualmente los matemáticos intentan demostrar teoremas de una gran complejidad. Un ejemplo es la demostración del último teorema de Fermat, que fue presentada por Andrew Wiles en 1993. Previo a su publicación un revisor encontró un error, que fue reparado en 1995 [53]. Otro ejemplo es la demostración del teorema de clasificación de los grupos finitos. Dicha prueba es el resultado del trabajo colaborativo de un gran número de investigadores que ha dado lugar a más de 10 000 páginas publicadas en diferentes revistas. Los expertos creen que actualmente ambas pruebas están completadas. Sin embargo, ¿podemos confiar en estas pruebas?

Las matemáticas tradicionalmente se consideran como una ciencia exacta, libre de toda imperfección. La producción de un teorema requiere de cierta capacidad creativa que origina una demostración del mismo. Sin embargo, una vez probado, la actividad de verificar si dicha demostración es válida es una actividad objetiva. Por ello, generalmente se considera que los teoremas publicados en revistas, que han sido sometidos a un proceso de comprobación por parte de revisores, son correctos. Sin embargo, la historia de las matemáticas contiene demostraciones incorrectas de resultados que no se han detectado durante grandes periodos de tiempo (como por ejemplo la prueba errónea de A. Kempe del teorema de los cuatro colores [29]).

Los ordenadores son en la actualidad herramientas de apoyo en el trabajo de un matemático. Son de gran utilidad a la hora de redactar artículos (TEX), comunicar y obtener información (web y correo electrónico) o hacer determinados cálculos (por ejemplo con sistemas de cálculo simbólico como Mathematica o Maple). Otro aspecto en el que se están utilizando es en el proceso de demostración de teoremas.

En la demostración de teoremas el ordenador puede ser útil en diferentes circunstancias. Por un lado, puede utilizarse como parte de una demostración que necesite, por ejemplo, resolver un gran número de casos, debido a que la prueba equivalente con lápiz y papel se escape, de momento, al posible trabajo a realizar por el hombre en un tiempo razonable. Existen varios ejemplos de dicho uso. Quizás los más famosos son las demostraciones del teorema de los cuatro colores [2] y de la conjetura de Kepler [21]. Este uso viene a añadir todavía más controversia a la validez de la

*Parcialmente subvencionado por el Ministerio de Ciencia e Innovación, proyecto MTM2009-13842-C02-01, y por el programa FET dentro del 7.º programa marco de la Comisión Europea, proyecto FORMATH, n.º 243847.

demostración, pues la fiabilidad del ordenador se convierte ahora en un ingrediente de la prueba. ¿Podemos confiar en las demostraciones que utilizan esencialmente cálculos hechos por ordenadores como parte de la propia demostración? Un interesante artículo de opinión incluyendo reflexiones a este respecto es [31]. Y en [10] se muestra claramente, mediante ejemplos concretos, que no siempre podemos fiarnos.

Un segundo uso de los ordenadores intenta abordar los problemas generados por las anteriores preguntas a través de la formalización de las matemáticas. Una demostración formal asistida por ordenador es una prueba en la que cada paso de inferencia lógica que contiene ha sido verificado por un computador. La confianza ahora se traslada a creer en la especificación del teorema en un asistente a la demostración, en la lógica implementada en dicho asistente y en la ausencia de errores en el programa informático de la herramienta. En estos momentos, se están abordando pruebas formales impresionantes como la demostración del teorema de los cuatro colores (ya completada), la conjetura de Kepler o el teorema de clasificación de los grupos finitos comentados anteriormente. La relevancia de la formalización de matemáticas con ayuda de ordenador parece confirmada por el reciente monográfico dedicado a este tema en la revista *Notices of the AMS* [48].

El ordenador también se está utilizando para demostrar la corrección o, al menos, intentar aumentar la fiabilidad, de resultados en otras ramas científicas. Especialmente importante es la propia área de la computación, en la que se intentan formalizar algoritmos tratando de demostrar su corrección tanto relativa a su definición abstracta como a una implementación concreta del mismo, así como componentes *software* y *hardware*. Ejemplos de este uso son la verificación de los elementos computacionales que actúan en áreas críticas como la aeroespacial o la médica. También, de un modo en cierta medida recursivo, se puede intentar formalizar las componentes que intervienen en la demostración formal.

En este trabajo trataremos de explicar estos nuevos usos de los ordenadores. Distinguimos en las dos próximas secciones entre demostraciones *no formalizadas* y demostraciones *formalizadas* de teoremas por medio de programas informáticos. Entendemos que las primeras tienen una parte de la demostración obtenida gracias a una computadora que posteriormente no es reproducible en un tiempo razonable con lápiz y papel, mientras que las segundas o bien se han obtenido a través de un ordenador pero es posible trazarlas por un humano o bien ya disponían de una demostración tradicional que ahora se formaliza. En la sección 3 se explica cómo utilizar estas técnicas para aumentar la fiabilidad de algoritmos, *software* y *hardware*. En la sección 4 incluimos un ejemplo de demostración de un teorema con ayuda de un asistente a la demostración. El artículo termina con un apartado de conclusiones.

1. Demostraciones no formalizadas de conjeturas por medio de programas informáticos

Entendemos por demostraciones no formalizadas de conjeturas por medio de programas informáticos aquellas demostraciones de resultados matemáticos que utilizan como parte esencial de la prueba un algoritmo ejecutado por un ordenador. Las de-

mostraciones no se han realizado previamente con lápiz y papel, y sin la ayuda de dicho programa no se hubieran obtenido. Además, una vez obtenidas, no se pueden desprender de la parte computacional para ser reproducidas. En estos casos el ordenador se puede utilizar, por ejemplo, para descartar casos en demostraciones que tienen un número muy grande de alternativas. También puede ser útil como herramienta de cálculo simbólico de modo que se obtengan a través de su cómputo nuevos resultados no conocidos hasta el momento.

Describimos a continuación dos ejemplos paradigmáticos en el uso de ordenadores para analizar distintos casos en una demostración: el teorema de los cuatro colores y la conjetura de Kepler. Y como ejemplo de herramienta de cálculo simbólico incluimos Kenzo.

- El teorema de cuatro colores [26] establece que cualquier mapa geográfico puede colorearse con cuatro colores diferentes, de forma que no queden regiones adyacentes (que compartan un segmento como borde y no sólo un punto) con el mismo color.

  Este teorema fue planteado por Francis Guthrie en 1852. En 1879 Alfred Kempe publicó una demostración [29], pero Percy Heawood detectó un error en 1890 [25]. Tras muchos años intentando arreglar dicha prueba, Kenneth Appel y Wolfgang Haken desarrollaron una demostración con la ayuda de un ordenador en 1976 [2].

  La demostración la obtuvieron por reducción al absurdo del modo que esbozamos a continuación. Si el teorema fuera falso, entonces debería existir al menos un mapa con el menor número de regiones que necesita al menos cinco colores para colorearse. Pero este contraejemplo mínimo no existe. La prueba tiene dos partes. Una primera parte consiste en identificar una colección de mapas de modo que todo mapa debe contener una porción que se *parezca* a uno de ellos. Redujeron la lista de mapas a un total de 1936. Para demostrar esta propiedad necesitaron cientos de páginas de análisis manuscritos. En una segunda parte calcularon que cada uno de estos mapas no puede formar parte de dicho contraejemplo más pequeño. Appel y Haken utilizaron un programa de ordenador que necesitaba varios días de cómputo para comprobar, uno a uno, que estos mapas en realidad tenían esta propiedad.

  Esta prueba no fue aceptada por todos los matemáticos dado que una persona se vería imposibilitada de verificarla manualmente. Sólo queda aceptar la exactitud del programa, del compilador y del ordenador en el cual se ejecutó el código.

- La conjetura de Kepler [36] intenta dar respuesta a la pregunta de cuál es el empaquetamiento óptimo (más denso) para un conjunto de esferas. La conjetura afirma que la disposición óptima es la red cúbica centrada en las caras. Ésta consiste en colocar esferas inicialmente sobre un plano, tangentes entre sí y formando hileras intercaladas (de modo que una esfera en una hilera es tangente a dos de otra hilera), que crean una capa sobre la que podemos apilar las nuevas esferas situándolas tangentes entre cada tres de la formación inicial. Posteriormente se itera este procedimiento, arriba y abajo de la primera ca-

pa. Dicho procedimiento no es otro que el que aparece ilustrado en la manera habitual como disponen los fruteros, por ejemplo, las naranjas.

En 1998 Thomas Hales anunció que tenía una prueba de la conjetura de Kepler [21]. Dicha demostración contenía unas 300 páginas manuscritas junto con 40 000 líneas de código en lenguaje de programación Java. La parte manuscrita supone la reducción del problema inicial a 5094 casos individuales que fueron estudiados mediante complejos cálculos en el ordenador. La prueba original de este teorema es especialmente difícil de comprobar. El mismo autor relata en [22] que, en la carta de aceptación de la prueba para ser publicada en la revista *Annals of Mathematics*, el editor describe el proceso de revisión como algo agotador para un equipo de personas que avanzaban con lentitud tratando de examinar durante semanas cada uno de los pasos de la demostración de la parte manuscrita. Posteriormente el código no fue examinado línea a línea, confiando en los métodos que el autor había utilizado para minimizar las posibles fuentes de errores.

- Kenzo [15] es un sistema de cálculo simbólico en Topología Algebraica desarrollado por Francis Sergeraert en 1999. Kenzo es capaz de obtener grupos de homología de espacios topológicos complejos (como son los espacios de lazos iterados). Actualmente el sistema ha calculado nuevos resultados que no se han obtenido por ningún otro método, ni tradicional ni por medio de otros programas informáticos, por lo que no es posible confirmarlos o refutarlos. Tras años de cómputo, no se ha encontrado que alguno de los cálculos realizados por el sistema sea incorrecto. Además, algunos de los grupos de homología encontrados han servido como guía para la obtención de teoremas matemáticos demostrados de forma tradicional [11, 46].

El ordenador está asumiendo cada vez con más frecuencia tareas dentro de la demostración de teoremas matemáticos. No obstante, las dudas razonables que en la comunidad matemática plantea el uso de programas informáticos en estas pruebas han llevado a algunos autores, como por ejemplo al previamente citado T. Hales, a utilizar asistentes de demostración para aumentar la fiabilidad de las mismas. A este tipo de asistentes dedicaremos la siguiente sección.

2. Demostraciones formalizadas de teoremas por medio de programas informáticos

En esta sección se introducirá la demostración formalizada de teoremas apoyada en herramientas informáticas que implementan un sistema lógico sobre el cual se construyen las pruebas. Es posible distinguir dos tipos de procesos de demostración en este ámbito dependiendo del grado de automatismo de los mismos. En una primera sección trataremos procesos de demostración automática de teoremas, en los que las herramientas utilizadas generan de forma automatizada una demostración. En una segunda sección abordaremos los procesos de demostración asistida por ordenador en los que se guía a la herramienta en el desarrollo de una demostración. Aunque existen herramientas en ambos campos, la línea de división entre las mismas no está

bien definida y de hecho muchas herramientas permiten ambos tipos de desarrollos (automatizado y asistido).

2.1. Demostración automática de teoremas

La idea de reducir razonamiento a cálculo mecánico es un viejo sueño [23]. Y aunque hace cincuenta años se pronosticó que, tras una década, un ordenador descubriría y probaría un nuevo teorema matemático importante, este hecho no se produjo [22]. En lo que sí se han conseguido éxitos es en el desarrollo de algoritmos capaces de resolver problemas específicos en los que el ordenador, a través de búsquedas por casos, cadenas de razonamientos basadas en reglas lógicas (lógicas de primer orden, reescrituras, etc.) o técnicas parecidas, lleva a la demostración de los correspondientes teoremas. En estos sistemas se suministra como dato de entrada el problema a resolver y, sin más intervención del hombre, son capaces de encontrar automáticamente una demostración del mismo.

Una herramienta relevante actualmente de este tipo es Otter [54] (o su sucesor Prover9 [35]). Este sistema está diseñado para probar teoremas a través de una lógica de primer orden con igualdad. Por ejemplo, la herramienta es capaz de deducir automáticamente millones de conclusiones que se obtienen de unas hipótesis dadas por el usuario. Un ejemplo de aplicación en el que el sistema está proporcionando destacables conclusiones es en el problema veinticuatro de Hilbert [47]. Este problema está relacionado con la búsqueda de la demostración más simple para un determinado teorema. Dicha simpleza se puede medir en términos de longitud, tamaño, estructura de los términos, complejidad de las fórmulas, etc.

En la página web `http://www.mcs.anl.gov/research/projects/AR/new_results/` es posible encontrar un listado de los resultados originales encontrados por Otter (u otros sistemas de este tipo como EQP o Mace [35]). Quizás el resultado más importante dentro de esta área es la solución de la conjetura de Robbins en 1996 [34], un problema que había desafiado durante más de 60 años a la comunidad matemática. La prueba, que no se había obtenido de forma tradicional, se alcanzó con el demostrador de teoremas EQP. Tras introducir el enunciado en este sistema, esperar casi 8 días de cálculo y emplear 30 megabytes de memoria, se obtuvo la solución. La prueba así obtenida se pudo recuperar y comprobar a mano.

La conjetura de Robbins establece que las álgebras de Robbins son álgebras booleanas. En 1933, E. V. Huntington presentó un álgebra booleana como un conjunto con una operación binaria $+$, que es asociativa y conmutativa, y una operación unaria n, que verifican la ecuación de Huntington $n(nx+y)+n(nx+ny)=x$. Un poco más tarde, H. Robbins planteó la cuestión de si la ecuación de Huntington podía reemplazarse por la ecuación más simple $n(n(x+y)+n(x+ny))=x$, que define un álgebra de Robbins. Robbins y Huntington no encontraron una solución a esta conjetura. Años más tarde, S. Winker encontró condiciones suficientes más débiles para abordar el problema [34]. Entre ellas, que es suficiente que en un álgebra de Robbins existan elementos c y d tales que $c+d=c$. La figura 1 contiene la solución en EQP de la mencionada conjetura en la que, comenzando con la ecuación de Robbins, se encuentran los elementos $c=2x$ y $d=n(n(3x)+x)+2x$.

7	$n(n(n(x)+y)+n(x+y))=y$	[Robbins equation]
10	$n(n(n(x+y)+n(x)+y)+y)=n(x+y)$	[7 → 7]
11	$n(n(n(n(x)+y)+x+y)+y)=n(n(x)+y)$	[7 → 7]
29	$n(n(n(n(x)+y)+x+2y)+n(n(x)+y))=y$	[11 → 7]
54	$n(n(n(n(n(x)+y)+x+2y)+n(n(x)+y)+z)+n(y+z))=z$	[29 → 7]
217	$n(n(n(n(n(x)+y)+x+2y)+n(n(x)+y)+n(y+z)+z)+z)=n(y+z)$	[54 → 7]
674	$n(n(n(n(n(n(x)+y)+x+2y)+n(n(x)+y)+n(y+z)+z)+z+u)+n(n(y+z)+u))=u$	[217 → 7]
6736	$n(n(n(n(3x)+x)+n(3x))+n(n(n(3x)+x)+5x))=n(n(3x)+x)$	[10 → 674]
8855	$n(n(n(3x)+x)+5x)=n(3x)$	[6736 → 7,simp:54,flip]
8865	$n(n(n(n(3x)+x)+n(3x)+2x)+n(3x))=n(n(3x)+x)+2x$	[8855 → 7]
8866	$(n(n(3x)+x)+n(3x))=x$	[8855 → 7,simp:11]
8870	$n(n(n(n(3x)+x)+n(3x)+y)+n(x+y))=y$	[8866 → 7]
8871	$n(n(3x)+x)+2x=2x$	[8865,simp:8870,flip]

Figura 1: Prueba en EQP de la conjetura de Robbins como fue presentada en [34].

2.2. DEMOSTRACIÓN ASISTIDA POR ORDENADOR

Las demostraciones de teoremas matemáticos están habitualmente descritas de modo que un matemático (especializado en el campo en el que se establece dicho teorema) debería ser capaz de reproducirlas sin demasiada dificultad. Normalmente dichos teoremas se sitúan en un contexto que se asume, y los pasos rutinarios obvios se omiten. Llamaremos *demostraciones tradicionales* a este tipo de pruebas.

A medida que las matemáticas se hacen más especializadas, se reduce también el número de personas que son capaces de entender las demostraciones que se incluyen en las mismas. De hecho, existen teoremas cuyas demostraciones matemáticas son de una complejidad enorme, y que escapan a las posibilidades de comprensión en un tiempo razonable de muchos matemáticos especializados en áreas afines a la misma. En estos casos se delega en la reputación de la revista que ha debido elegir revisores adecuados que han confirmado la validez de dicho resultado.

Llamaremos *demostración formal* de un teorema matemático a una prueba en la que se hace explícita cada inferencia lógica que es necesaria en dicha demostración. La prueba comienza en los axiomas fundamentales de la lógica que se está utilizando, y ni un solo paso se deja a la intuición o se omite, por trivial que nos parezca.

La definición de demostración formal es independiente del uso de computadores y se remonta a los logicistas del siglo XIX. El propósito consistía en dotar a las matemáticas de unos fundamentos precisos a partir de los cuales deducir sus resultados. Quizás el primer sistema de deducción formal para las matemáticas fue introducido por Frege en 1879 [17]. Otros autores como Russell, Zermelo, Peano o Hilbert trabajaron posteriormente definiendo nuevos sistemas formales que intentaban evitar la existencia de paradojas en los mismos [23]. En los años 30 del siglo pasado el grupo de matemáticos francés llamado Bourbaki [6] se propuso revisar los fundamentos de las matemáticas con idea de dar demostraciones completas de las mismas. Su objetivo era suministrar un fundamento sólido de todas las matemáticas modernas.

No obstante, este grupo abandonó su propósito porque consideraron que el proyecto era inabordable. En sus propias palabras, la prueba formal de una demostración informal modesta requeriría de cientos de pasos lógicos. Dicha demostración, además, está llena de detalles tediosos y repetitivos, y por ello es incluso más propensa a error que la demostración tradicional [23].

El desarrollo de potentes herramientas de procesamiento ha favorecido en los últimos años un resurgimiento de la formalización de las matemáticas. Los pasos formales que para una persona pueden resultar aburridos son una de las especialidades de los ordenadores. Podemos implementar en un asistente una serie de axiomas y unas reglas de inferencia que fundamenten una determinada lógica. Posteriormente podemos ir utilizando dicho asistente para reproducir la demostración garantizando que los pasos que se van dando son acordes con la lógica.

Además, dichas máquinas pueden ayudar no sólo a validar la corrección de todos los pasos lógicos dados en la demostración, sino que pueden suministrar interfaces agradables en las que desarrollar la demostración, ayudando a construir los posibles pasos lógicos a dar e incluso encontrando de forma automática muchos de esos pasos. Por otro lado, una vez obtenida la demostración, se pueden ofrecer distintas vistas de la misma dependiendo del nivel de detalle exigido por el lector. Incluso pueden facilitar o llegar a sustituir parte de las tareas de los revisores de la misma. No nos debería sorprender que, si los ordenadores están revolucionando el trabajo en muchos ámbitos científicos, por dar un ejemplo podemos pensar en la meteorología, también puedan llegar a ser útiles en las demostraciones matemáticas.

El método de trabajo habitual es el siguiente. Una demostración formal de un teorema comienza por una demostración matemática tradicional que se escribe con detalle haciendo explícitas todas las premisas y todos los casos de que consta la demostración. Posteriormente, se elige un asistente a la demostración de los diversos que existen (de los cuales citaremos algunos más adelante) basándonos en sus lógicas implementadas o en las librerías desarrolladas en cada uno, se representa el teorema en la herramienta, y se reproduce dicha prueba con ayuda de la misma. El asistente garantizará que no se incluyan inferencias en la demostración que no sean producto de los axiomas y reglas lógicas que tenga implementados.

Esta traducción de la prueba tradicional a la prueba formal exige en general mucho esfuerzo y dedicación. Podríamos entonces preguntarnos, si no nos ayuda a realizar la demostración, que de hecho ya tenemos de partida, ¿qué beneficio obtenemos de dicha prueba? Podemos por un lado insistir en el hecho de que, como la máquina sólo admite inferencias lógicas permitidas, añade una mayor seguridad en la corrección de la prueba. Posteriormente, la demostración puede ser ejecutada (es decir, comprobada) en cualquier ordenador que tenga instalado el asistente que la generó. Además, y más importante, se alcanza una comprensión mucho mayor de la prueba, pues el propio proceso exige una reflexión profunda sobre los pasos que se van dando. Este procedimiento suministra un nuevo enfoque de la estructura de la demostración, lo que puede llevarnos a buscar simetrías, simplificaciones o generalizaciones de la misma.

Existen diferentes asistentes para la demostración, con sus propias características y lógicas subyacentes. Dichos sistemas compiten entre sí, en cierta medida del

mismo modo que lo hacen los lenguajes de programación o los sistemas operativos. En [52] puede encontrarse una lista de 100 teoremas de una diversidad de áreas matemáticas y las demostraciones dadas hasta el momento de los mismos en diferentes asistentes. Las herramientas que más se están utilizando, de acuerdo con el éxito en la demostración de esta lista de teoremas son: *HOL Light*, *Mizar*, *ProofPower*, *Isabelle* y *Coq*. Por ejemplo, el primer teorema de la lista se refiere a la irracionalidad de $\sqrt{2}$. Este teorema se ha utilizado en [50] para comparar 17 asistentes diferentes. Además en [49] se hace una comparación de los mismos atendiendo a criterios como el tamaño de la librería de teoremas ya probados que ofrecen, la lógica que implementan y el grado de automatización que consiguen. No se pretende decidir cuál es el mejor. En realidad son diferentes, cada uno implementa buenas ideas y se adapta mejor a determinados ámbitos. Lo que puede desprenderse de dicha comparación es la gran diversidad de herramientas que existen y lo difícil que resulta trasladar (y ya no decimos reutilizar) pruebas realizadas de una a otra.

Con ayuda de los asistentes para la demostración se han conseguido recientemente éxitos notables de pruebas formales de teoremas matemáticos para nada triviales. Destacaremos a continuación algunos de los grandes hitos de la demostración mecanizada, ya conseguidos o actualmente en proyecto de desarrollo:

- Teorema de los cuatro colores. Quizás la demostración más espectacular realizada con un asistente para la demostración. Fue obtenida en 2005 por Georges Gonthier utilizando Coq [19]. Aunque la prueba tradicional de Appel y Haken utiliza una computadora, y Gonthier en su prueba formal también, el uso de esta herramienta en ambos casos es completamente distinto. En la primera, como se ha indicado previamente, se utiliza el ordenador como una calculadora para descartar casos, mientras que en la segunda se utiliza para reproducir de forma minuciosa una serie de pasos lógicos (dentro de la lógica implementada en Coq) que llevan a la consecución de la prueba.
- El proyecto Flyspeck [38], que Thomas Hales comenzó en 2003, pretende establecer una prueba formal de la conjetura de Kepler. Ésta garantizará un nivel de seguridad en la corrección de la demostración superior a la alcanzada hasta el momento. En el año 2008 casi la mitad del código empleado en la demostración estaba ya certificado [22] y la prueba continúa actualmente en proceso de desarrollo.
- Se están ofreciendo pruebas formales de teoremas en distintas áreas de las matemáticas. Ejemplos son el teorema del número primo, el teorema de la curva de Jordan, el teorema fundamental del Cálculo o el teorema fundamental del Álgebra [52]. En este momento se están intentando formalizar otros resultados como el teorema de Feit-Thompson (a cargo del equipo de Gonthier [20]), como paso previo a demostrar el teorema de clasificación de los grupos finitos. Se ha propuesto el reto a largo plazo de formalizar el último teorema de Fermat (incluido en la lista de teoremas anteriormente mencionada [52]).

Ahora bien, los asistentes para la demostración son programas informáticos, ¿podemos confiar en ellos? En [42] se indica que al hacer una demostración con una de estas herramientas debemos confiar en una sucesión de capas:

1. El código del asistente a la demostración, el compilador con el que fue procesado, el sistema operativo y el propio *hardware* que estamos utilizando.
2. La lógica utilizada y la representación de dicha lógica en el asistente.
3. La representación del propio teorema en el asistente, que se corresponda con lo que queremos probar.

No parece posible asegurar la corrección de todos estos aspectos, pero sí se pueden dar motivos para incrementar en el lector la confianza en estas demostraciones, de modo que deberían merecer finalmente mayor fiabilidad que una prueba tradicional.

El primer punto puede ser defendido atendiendo al hecho de que los asistentes se programan y se utilizan en distintos ordenadores y sistemas operativos. Una prueba se puede verificar en distintos computadores de modo que es raro que, si un error se debe a dichos aspectos, se repita exactamente en otro ordenador. Además, se intenta que el núcleo de los asistentes, es decir el conjunto de axiomas y reglas lógicas implementadas en el mismo, sea lo más pequeño posible para que pueda ser revisado con detalle. Dicho núcleo es la única herramienta que puede ser utilizada para elaborar el resto de resultados (éste es el llamado criterio de *de Bruijn*, cumplido por la mayoría de sistemas [49]). En el caso del sistema HOL Light este núcleo es especialmente pequeño: consiste en 285 líneas de código ML. Como el núcleo es tan pequeño, puede ser verificado a diferentes niveles tratando de garantizar su corrección, por ejemplo a través de la especificación formal del mismo.

En el segundo punto debemos tener en cuenta que las lógicas se han probado consistentes antes de su implementación, y que sólo se permite utilizar los mecanismos implementados en el sistema para desarrollar las pruebas. Dichos mecanismos tienen una representación sintáctica fijada con una especificación semántica asociada. Para probar la solidez del sistema axiomático se han utilizado pruebas tanto matemáticas como formales (habitualmente desarrolladas en otros asistentes para la demostración, como por ejemplo en [39] que traslada la lógica implementada en HOL a Isabelle). Además, como nivel adicional de protección, muchas de las pruebas realizadas con un asistente a la demostración han sido trasladadas a otro asistente. De este modo si la prueba tuviera un error debido a un fallo del sistema, entonces sería extraño que la prueba fuera también cierta en el otro sistema debido a otro fallo en el mismo [49].

Finalmente, si creemos ciertos los dos puntos anteriores, la confianza necesaria en el tercer punto es similar a la que necesitamos en las matemáticas tradicionales.

Uno de los objetivos de la comunidad de usuarios de un asistente a la demostración consiste en intentar mantener una librería de pruebas formalizadas de modo que cualquiera pueda utilizar un resultado de la misma. La suma del esfuerzo de distintos investigadores en esta librería común hace que la interacción con el sistema vaya poco a poco alejándose de los primeros axiomas que forman parte de la lógica hacia un uso que recuerde a la práctica a más alto nivel de las matemáticas. Éste es el objetivo del Manifiesto Q.E.D. [7]: un proyecto anónimo que persigue el sueño de codificar en una gran librería de pruebas formales la mayoría de resultados matemáticos significativos. La meta está puesta en que estos sistemas sean lo suficientemente cercanos a la forma de trabajar en matemáticas como para que se conviertan en he-

rramientas familiares en el trabajo en cada una de sus ramas. Podemos destacar líneas de trabajo en esta dirección, como por ejemplo las emprendidas por el proyecto europeo ForMath, *Formalisation of Mathematics* [16], en el que participa nuestro grupo de investigación.

Para terminar esta sección nos gustaría recomendar al lector interesado algunas referencias en las que profundizar en este tema. En primer lugar citamos el libro [33], donde se introduce el ámbito de la demostración por medio de programas desde el punto de vista de la repercusión que la misma puede tener en la credibilidad de las matemáticas, sobre la propia idea de demostración matemática convencional, o sobre algunos de los ejemplos más controvertidos de la misma. Para una introducción más práctica al área, con descripciones de métodos de demostración con ordenador, tipos de lógicas, múltiples ejemplos o ejercicios recomendamos [24]. Finalmente, queremos destacar también el primer intento de desarrollar una *máquina lógica*, ya que el mismo proviene del filósofo y místico mallorquín Ramón Llull (1232–1315). Durante un retiro místico en el monte Randa, Llull tuvo una revelación en la que se le explicó la forma en que debía confundir a los infieles y establecer los dogmas de su fe cristiana. A tal fin, Llull se encerró en un monasterio para realizar su *Ars Magna*. El objetivo de Llull era eminentemente teológico, ya que pretendía desarrollar una forma de razonamiento tan clara y detallada que pudiese convencer a cualquiera (y más en particular, a los musulmanes) de sus propias ideas religiosas (con el fin de que éstos no pudieran negarse a abrazar su misma fe). Llull diseñó y construyó una máquina lógica en la cual los sujetos y predicados teológicos estaban dibujados por figuras geométricas que interactuaban de forma adecuada. Para una explicación más detallada sobre su máquina, sobre la polémica suscitada por la misma en los siglos posteriores (especialmente entre Franciscanos y Dominicos, pero también por eminentes matemáticos como Leibniz), y sobre otros intentos posteriores de desarrollar máquinas lógicas, recomendamos el entretenido texto de Gardner [18].

3. Demostración asistida por ordenador aplicada a la ingeniería del software

Trabajamos a diario con programas informáticos que fallan debido, en muchas ocasiones, a errores en la programación de los mismos. Estos errores hacen que nuestro programa se bloquee, lo que en el mejor de los casos supone una pérdida de unos minutos de nuestro trabajo. Existen sin embargo otro tipo de errores que hacen perder bastante dinero a los usuarios de estos programas o, mucho peor, pueden suponer la pérdida de vidas humanas. Podemos pensar por ejemplo en programas informáticos en áreas críticas como la aeronáutica o la medicina. Ejemplos paradigmáticos son la explosión del cohete Ariane 5 o el error de uno de los últimos procesadores Intel debido a un fallo en el proceso de división en coma flotante [23].

Se sabe de la dificultad de conseguir *software* sin errores. Pero si pensamos en los citados sectores de la industria es fundamental tratar de garantizar que sus programas carezcan de fallos. Es en esta área donde los métodos formales tienen cada día más importancia a través de la *verificación formal* de programas. La idea

es no delegar simplemente en el uso de test, es decir, en comprobar que el programa funciona bien en una batería de posibles datos de entrada, sino en intentar aplicar a los programas informáticos mecanismos que aseguren la corrección de los mismos.

La forma habitual de trabajar en este campo consiste en tratar de verificar si un programa cumple unos requisitos impuestos al mismo. Para llevar a cabo esta tarea habitualmente se realiza previamente un proceso de abstracción del programa real que se traduce en un modelo matemático del mismo. Por otro lado, los requisitos también deben expresarse a través de una especificación matemática. Finalmente, la verificación formal es capaz de probar si el modelo matemático verifica la especificación matemática construida. En este procedimiento existen, no obstante, dos pasos en los que debemos confiar: que los modelos matemáticos elegidos para el programa y sus requisitos realmente cumplan su propósito. Una alternativa consiste en tratar de probar si es correcto directamente el código que se ha utilizado en la programación. Para ello existen demostradores de teoremas como ACL2 [28] que son a la vez un lenguaje de programación (en realidad, es una parte del lenguaje de programación funcional Common Lisp) y un asistente a la demostración. Otra opción es la extracción de código directamente del algoritmo que se ha implementado y probado correcto en el asistente a la demostración. Ejemplos de esta técnica pueden ser los ya mencionados sistemas Coq, que permite extraer directamente código a OCaml, Haskell o Scheme; o Isabelle, en el que se está trabajando para extraer código a distintos lenguajes de programación como ML.

Estas técnicas pueden ampliarse a la verificación de lenguajes de programación, compiladores, sistemas operativos e incluso diseños de *hardware*. Ejemplos son la verificación formal de un compilador para el lenguaje de programación C [32] o la del núcleo de un sistema operativo [30].

3.1. Verificación formal de algoritmos

En algún lugar entre el uso de la demostración asistida por ordenador aplicada a la ingeniería del software y aplicada a la formalización de matemáticas encontramos el uso de estas herramientas para verificar software de carácter matemático. En este contexto se desarrolla nuestra investigación. Más concretamente, en el área de los sistemas de cálculo simbólico en Álgebra Topológica a través del análisis del sistema Kenzo que hemos introducido en la sección 1.

Como hemos indicado, este programa es capaz de obtener grupos de homología desconocidos hasta el momento, y nuestro objetivo es tratar de aumentar la confianza en la corrección de dichos resultados. En este punto, los métodos formales pueden jugar un papel importante tratando de analizar y verificar partes de Kenzo. En particular nos estamos centrando en formalizar los algoritmos que se han implementado en Kenzo. Un resultado conseguido en esta línea ha sido la verificación del Lema Básico de Perturbación a través de Isabelle/HOL [3]. Este lema es esencial en la teoría de Homología Efectiva desarrollada por Francis Sergeraert [44]. Se está ahora trabajando en la extracción de código a partir de la representación realizada. Resultados obtenidos en esta línea se encuentran en [4]. También hemos formalizado el algoritmo que obtiene la homología efectiva de un bicomplejo, a través de Coq [14].

Otra línea de trabajo ha sido verificar en ACL2 directamente código Common Lisp usado en Kenzo [1]. Además, el uso de distintos asistentes para la demostración de teoremas nos permite hacer comparaciones entre ellos (ver por ejemplo [5]).

4. UN EJEMPLO DESARROLLADO

En esta sección incluiremos, por medio de un ejemplo, una herramienta de demostración asistida por ordenador, como las presentadas en la sección 2.2. Nuestro propósito no es mostrar la capacidad de dichas herramientas para tratar con resultados matemáticos de gran complejidad (lo cual ya ha sido introducido por medio de los ejemplos más conocidos de uso de éstas en las secciones anteriores), sino ilustrar algunas de sus características y aspectos de la forma de trabajar con ellas por medio de un ejemplo que debería resultar pedagógico. En este caso, el sistema de demostración utilizado será *Isabelle* [37], que es una de las herramientas que utilizamos en nuestro grupo de trabajo, y nuestro ejemplo será un resultado bien conocido en Álgebra Abstracta, como es el teorema de la división para polinomios con coeficientes en un anillo. La razón para elegir este resultado es que el mismo ha sido formalizado en Isabelle por uno de los autores del artículo, pasando a formar parte de la librería estándar del sistema sobre polinomios. El resultado nos permitirá mostrar las características más importantes del sistema relativas a representación de estructuras algebraicas, polinomios, y demostración sobre dichos ingredientes.

Empezaremos por enunciar el resultado que queremos demostrar (extraído de [27, Teorema 2.14], traducido por los autores). Dado $f(x)$ un polinomio, denotaremos por $\deg f(x)$ el *grado* del mismo. Por *coeficiente principal* de un polinomio $f(x)$ entenderemos el coeficiente del mismo en grado $\deg f(x)$.

TEOREMA 1. *Sean $f(x)$ y $g(x)$ polinomios en $R[x]$, con $g(x) \neq 0$, R un anillo conmutativo, y sea m el grado y b_m el coeficiente principal de $g(x)$. Entonces, existen $k \in \mathbb{N}$ y polinomios $r(x), q(x) \in R[x]$ con $\deg r(x) < \deg g(x)$ tales que*

$$b_m^k f(x) = q(x)g(x) + r(x). \tag{1}$$

DEMOSTRACIÓN. Realizaremos la demostración por inducción en el grado de $f(x)$. En particular, suponiendo que el resultado es cierto para cualquier polinomio de grado menor que el grado de $f(x)$, debemos ser capaces de demostrarlo para $f(x)$.

Distinguimos dos casos:

- Si $\deg f(x) < \deg g(x)$, entonces $f(x) = 0 \cdot g(x) + f(x)$, con $\deg f(x) < \deg g(x)$.
- En otro caso, $(\deg f(x) = n) \geq (\deg g(x) = m)$. De nuevo separamos la demostración en dos casos:
  - Consideramos primero el caso en que $\deg f(x) = 0$. Sea a_0 el coeficiente de $f(x)$ en grado 0. Entonces tenemos que $f(x) = a_0$. Como $\deg g(x) \leq \deg f(x)$ tenemos que $\deg g(x) = 0$; si denotamos por b_0 el coeficente de $g(x)$ en grado 0, tenemos que $g(x) = b_0$. De las anteriores igualdades, podemos concluir ahora que $b_0 \cdot f(x) = a_0 \cdot g(x) + 0$, por ser R conmutativo.

- En el caso de que $\deg f(x) \neq 0$, sea a_n el coeficiente principal de $f(x)$. Definimos
$$b_m f(x) - a_n x^{n-m} g(x) = f_1(x). \tag{2}$$
Como los coeficientes de x^n en $b_m f(x)$ y en $a_n x^{n-m} g(x)$ son ambos $a_n b_m$, entonces $\deg f_1(x) < \deg f(x)$ (ya que el grado de $f(x)$ no puede ser igual a 0). Por lo tanto, podemos introducir ahora la hipótesis de inducción para obtener $k_1 \in \mathbb{N}$, $q_1(x)$ y $r_1(x) \in R[x]$ con $\deg r_1(x) < \deg g(x)$ y tales que
$$b_m^{k_1} f_1(x) = g(x)q_1(x) + r_1(x). \tag{3}$$
Entonces, por (2) y (3), tenemos
$$b_m^{k_1+1} f(x) = b_m^{k_1} a_n x^{n-m} g(x) + g(x)q_1(x) + r_1(x) = g(x)q(x) + r_1(x) \tag{4}$$
donde $q(x) = b_m^{k_1} a_n x^{n-m} + q_1(x)$ (y $\deg r_1(x) < \deg g(x)$). □

Cabe destacar que en el texto de donde hemos sacado la demostración original (véase [27, pág. 129]), el caso en el que $\deg f(x) = 0$, que en la demostración debe ser tratado de manera independiente, no llega a ser considerado. Sirva este ejemplo para dar una idea del nivel de detalle que se suele alcanzar en las demostraciones escritas a mano en textos de matemáticas (cabe reseñar que el texto citado es un libro de iniciación al Álgebra Abstracta, y que podría ser considerado como prolijo en detalles). En nuestro caso, notamos la ausencia de dicho caso al tratar de realizar la demostración formal del teorema en Isabelle, que nos obligó a considerar el mismo explícitamente.

En la breve demostración anterior se pueden observar dos técnicas diferentes de razonamiento habituales en el razonamiento matemático:

- Razonamiento por casos (con respecto a $\deg f(x) < \deg g(x)$ y $\deg f(x) = 0$).
- Razonamiento por inducción (en el grado de $f(x)$).

Ambas son ejemplos de razonamientos clásicos en matemáticas que también pueden ser implementados en un asistente de demostración (en nuestro caso, en Isabelle).

Dividiremos esta sección en dos subsecciones; comenzaremos por introducir el asistente Isabelle/HOL de un modo genérico (sección 4.1), tanto la lógica que implementa como la forma de demostrar teoremas con el mismo, y en la segunda parte detallaremos cómo hemos aplicado el mismo al ejemplo sobre divisibilidad de polinomios presentado (sección 4.2).

4.1. Introducción a Isabelle/HOL

Isabelle [37] es un demostrador de teoremas genérico (en el sentido de que sobre el mismo se pueden implementar diversas lógicas). Está programado en *ML*, un lenguaje de programación funcional bastante extendido. En el núcleo de Isabelle (conocido como *Isabelle/Pure*, o como metalógica) podemos encontrar una serie de reglas de inferencia básicas que se corresponden con un fragmento de lógica de orden

superior, tal como la desarrolló Alonzo Church [9], también llamada teoría de tipos simple.

Se puede encontrar una descripción detallada de dicha metalógica en [40]. Sin entrar en detalle, simplemente mencionaremos que la misma está formada por dos componentes principales:

- Un *sistema de tipos*, similar al que podemos encontrar en otros lenguajes de programación convencionales (Java, C++...), pero en este caso mucho más sencillo, ya que sólo admite tipos no vacíos (σ, τ) y tipos funcionales ($\sigma \to \tau$) formados a partir de tipos ya existentes. Dentro de estos tipos hay un tipo especial, denotado *prop*, que contiene las proposiciones del sistema. En particular, contiene las constantes verdadero, $\top$ y falso, $\bot$.
- Una serie de *reglas de inferencia* que actúan sobre elementos del tipo *prop*, y que expresan las propiedades de los conectores de la metalógica. Éstos son $\phi \Longrightarrow \psi$, que significa «$\phi$ implica ψ», el cuantificador universal $\bigwedge$, de tal modo que $\bigwedge x.\phi$ equivale a «para todo x, ϕ es verdad», y la igualdad $a \equiv b$.[1]

La forma de definir funciones en el sistema es la siguiente. Si a cada constante x de un tipo σ le asignamos un valor $b(x)$ de un tipo τ, la λ-abstracción $\lambda x\colon \sigma.b(x)$ denota la función de tipo $\sigma \to \tau$ que a cada x le asigna $b(x)$.

Sobre esta metalógica se pueden implementar diversas lógicas. Por ejemplo, la distribución estándar de Isabelle incluye implementaciones de lógica de primer orden, de la teoría de conjuntos de Zermelo-Fraenkel, de lógica de funciones computables, o de lógica clásica computacional. También contiene una implementación de lógica de orden superior (HOL), en la cual nos centraremos, ya que nuestros desarrollos posteriores se realizarán sobre dicha lógica. Si bien parecería más natural implementar resultados matemáticos en teoría de conjuntos de Zermelo-Fraenkel, HOL es la teoría más utilizada en Isabelle, por lo cual cuenta con herramientas y facilidades adicionales (relacionadas, por ejemplo, con generación de código o definición de funciones recursivas, las cuales no describiremos aquí), y su capacidad expresiva ha servido para formalizar numerosos resultados en matemáticas.

Para poder implementar HOL sobre la metalógica expuesta debemos definir un *sistema de tipos* que recoja las propiedades de HOL (conocida también como teoría de tipos simple), así como un conjunto de *reglas* (o *axiomas*) que definan dicho sistema lógico.[2]

Con respecto a los tipos, y siguiendo la noción de tipo que había en la metalógica, se pueden definir nuevos tipos siempre que éstos no sean vacíos. También existe un mecanismo que nos permite definir nuevos tipos como *subconjuntos* de tipos ya existentes. De este modo se puede definir el tipo *producto* de dos tipos (y así poder trabajar con tuplas), el tipo *suma* de dos tipos ya definidos (produciendo el conjunto de las sumas directas de los elementos de ambos tipos), o también tipos definidos

[1] Los símbolos $\Longrightarrow$, $\bigwedge$ y $\equiv$ son elegidos para la metalógica, dejando así disponibles para las lógicas que implementemos sobre ella los más habituales $\longrightarrow$, $\forall$ e $=$.

[2] En general, cuando trabajamos con demostradores de teoremas, hay que ser especialmente cuidadoso al incluir *axiomas*, ya que un axioma inadecuado o erróneo puede hacer que nuestro sistema se vuelva inconsistente, permitiendo probar cualquier resultado.

por inducción. El propio sistema ofrece facilidades que convierten la definición de estos tipos en una tarea simple. Con dichas facilidades, podemos definir (de hecho, pertenecen a la librería estándar de Isabelle/HOL) tipos para los números naturales, enteros, reales, complejos, tipos representando registros, polinomios, matrices, grupos, anillos, espacios vectoriales, y prácticamente cualquier tipo de estructura que pueda aparecer en un texto de matemáticas (en general, en un demostrador de teoremas, es más fácil modelar o representar estructuras que demostrar propiedades sobre las mismas).

Con respecto a las nuevas reglas (o axiomas) añadidas, la lista, como se puede observar, es bastante reducida:

```
refl: t = t
subst: [ s = t ; P s ] ⟹ P t
ext: (⋀x. f x = g x) ⟹ (λx. f x) = (λx. g x)
impI:(P ⟹ Q) ⟹ P⟶ Q
mp: [ P ⟶ Q ; P ] ⟹ Q
iff: ( P ⟶ Q ) ⟶ ( Q ⟶ P ) ⟶ ( Q = P )
someI: P x ⟹ P (εx. P x)
True_or_False: (P = True) ∨ (P = False)
```

De los axiomas anteriores, **ext** expresa la extensionalidad de funciones (con respecto al cuantificador universal $\bigwedge$ de la metalógica). La regla **iff** impone que las fórmulas lógicamente equivalentes son iguales. La regla **True_or_False** (conocida también como *tercio excluso*) hace que la lógica implementada sea clásica. Eliminando este axioma podríamos implementar también lógicas constructivas sobre la metalógica propia de Isabelle. La regla **impI** relaciona el elemento de la lógica $\longrightarrow$ con el elemento de la metalógica $\Longrightarrow$. El resto de elementos utilizados en la lógica (las constantes *True* y *False*, los conectores $\neg$, $\forall$, $\wedge$, $\vee$, el existencial único $\exists_1 \ldots$) se pueden definir (sin necesidad de introducirlos axiomáticamente) a partir de las conectivas anteriores. Por ejemplo, *True* se define igual a $(\lambda x.x) = (\lambda x.x)$ y $\neg$ se define como $\neg P = (P \longrightarrow \mathit{False})$.

Veamos ahora un sencillo ejemplo de cómo llevar a cabo una demostración dentro de la lógica de orden superior implementada, con un resultado bien conocido en lógica clásica:

```
lemma "¬ (A ∧ B) ⟶ ¬ A ∨ ¬ B"
proof
  assume n: "¬ (A ∧ B)"
  show "¬ A ∨ ¬ B"
  proof (rule ccontr)
    assume nn: "¬ (¬ A ∨ ¬ B)"
    have "¬ A"
    proof (rule notI)
      assume a: "A"
      have "¬ B"
      proof (rule notI)
        assume b: "B"
        from a and b have "A ∧ B" by (rule conjI)
```

```
        with n show False by rule
      qed
      hence "¬ A ∨ ¬ B"
        using disjI2 [of "¬ B" "¬ A"] by fast
      with nn show False by fast
    qed
    hence "¬ A ∨ ¬ B"
      using disjI1 [of "¬ A" "¬ B"] by fast
    with nn show False by fast
  qed
qed
```

El enunciado expresa que la negación de la conjunción de dos proposiciones cualesquiera *A* y *B* es igual a la negación disjunta de las mismas. Como se puede observar, la notación dentro del sistema de demostración es bastante similar a una demostración en lenguaje natural. Por medio del comando **show** fijamos una proposición que pretendemos demostrar. El comando **have** nos permite fijar resultados parciales necesarios en la demostración, que pueden ser utilizados más adelante. Cada vez que utilizamos el comando **proof** abrimos un nuevo contexto en el que intentaremos demostrar la proposición fijada (por medio de **show** o **have**). Cada una de esas demostraciones se realiza por medio de lo que se conocen como *procedimientos de demostración*. Estos procedimientos actúan sobre la prueba que queremos realizar (del mismo modo que un algoritmo lo hace sobre sus datos de entrada). Por ejemplo, en la demostración presentada podemos observar algunos de ellos como *rule* o *fast*. También pueden recibir ayudas adicionales en la forma de axiomas o lemas previos ya demostrados. Por ejemplo, de los dos mencionados anteriormente, *rule* trata de aplicar el teorema que tiene como parámetro al resultado que queremos probar. Si ambos resultados coinciden (el sistema tratará de instanciar las variables correspondientes), el estado de nuestra demostración cambiará, para convertirse en las premisas de la regla que hemos aportado como parámetro de *rule*. Por medio de cambios sucesivos en el resultado que queremos demostrar lo vamos convirtiendo en resultados más sencillos; si estos resultados coinciden con alguna de nuestras premisas (en el resultado mostrado, $\neg(A \wedge B)$) o con algún resultado previo ya probado en Isabelle/HOL, la demostración (del resultado total o de los resultados intermedios) estará completada (lo que marcamos por medio del comando **qed**).

En cualquier caso, al igual que pasa con las demostraciones matemáticas, podemos proporcionar distintas demostraciones para un mismo resultado. En particular, para el resultado anterior, y haciendo uso de procedimientos de demostración más avanzados dentro de los que ofrece el sistema, el resultado anterior se puede demostrar en una simple línea:

```
lemma "¬ (A ∧ B) ⟶ ¬ A ∨ ¬ B"
  by simp
```

En la demostración anterior, *simp* es una combinación de reglas de demostración que nos ofrece el sistema. El usuario interesado en conocer de forma detallada qué reglas de demostración han sido usadas por el método *simp*, y de ese modo poder

reconstruir la demostración de forma exhaustiva, puede hacerlo dentro del propio sistema con total transparencia.

4.2. Formalización en Isabelle/HOL de un resultado de Álgebra Abstracta

Pasamos ahora a aplicar la herramienta mostrada en la sección anterior al Teorema 1. Esta sección la dividiremos en tres partes. En la sección 4.2.1 presentamos una forma de representar las estructuras algebraicas que aparecen en el teorema. En la sección 4.2.2 implementaremos polinomios sobre los anillos ya introducidos. Finalmente, en la sección 4.2.3 presentamos la demostración del teorema de estudio.

4.2.1. Representación de estructuras algebraicas

Para poder implementar la estructura del anillo R que aparece en el Teorema 1 empezamos por la estructura más simple de monoide. La definición de monoide constará, por una parte, del tipo de los objetos que poseen la misma estructura que un monoide (con respecto al sistema de tipos introducido), y por otra parte de las propiedades que definen qué objetos de los de tipo monoide son realmente monoides. Las definiciones matemáticas, salvo que digamos lo contrario, seguirán las dadas en [27].

Veamos en primer lugar la definición del tipo de los *monoides*:

```
record 'a monoid =
  carrier :: "'a set"
  mult    :: "['a, 'a] ⇒ 'a" (infixl "⊗₁" 70)
  one     :: 'a ("1₁")
```

Con respecto al tipo definido, debemos aclarar que la definición ha sido hecha por medio de un *registro*. Los registros forman parte del sistema de tipos de Isabelle/HOL, ya que internamente son codificados como un *producto de tipos* en el que cada uno de los tipos está etiquetado por el nombre del campo que se le asigna en dicho registro. En el caso anterior, el registro está formado por un campo de nombre `carrier`, que representa el conjunto soporte de la estructura algebraica, y por dos operaciones, una binaria (`mult`) y una operación unaria o constante (`one`). El tipo subyacente de la estructura (denotado por `'a`) es un tipo variable, lo cual nos permitirá representar monoides de enteros, naturales, listas, o cualquier otro tipo de dato de que disponga el sistema.

El sistema también ofrece ciertas capacidades para introducir sintaxis más adecuada definida por el propio usuario. Por ejemplo, por medio de la anotación «**infixl** $\otimes_1$ `70`» hemos creado un alias de la operación `mult G`, que podrá ser denotada por medio del operador infijo $\otimes_G$ (el cual se ha declarado como asociativo a izquierda, con un orden de prioridad con respecto a otras operaciones de 70). Si la estructura implícita (G) es clara del contexto, también podemos abreviar la anterior notación a $\otimes$. De igual modo $\mathbf{1}_G$ (o $\mathbf{1}$, si existe un único monoide en nuestro contexto de trabajo) será lo mismo que `one G`.

No toda estructura con un soporte, una operación binaria y una constante es un monoide; ésta debe cumplir una serie de propiedades con respecto a dichas operaciones. Por la implementación propia de Isabelle/HOL, dichas propiedades deben ser especificadas de forma separada. Veamos cómo podemos hacerlo:

```
locale monoid =
  fixes G (structure)
  assumes m_closed [intro, simp]:
        "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊗ y ∈ carrier G"
      and m_assoc:
        "⟦x ∈ carrier G; y ∈ carrier G; z ∈ carrier G⟧
         ⟹ (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
      and one_closed [intro, simp]: "𝟭 ∈ carrier G"
      and l_one [simp]: "x ∈ carrier G ⟹ 𝟭 ⊗ x = x"
      and r_one [simp]: "x ∈ carrier G ⟹ x ⊗ 𝟭 = x"
```

En el fragmento de código anterior, `G` hace referencia al *monoide* que estamos definiendo, y cada una de las líneas posteriores representa una de las propiedades características de un monoide. Entre ellas podemos observar que la operación binaria ($\otimes$) es cerrada sobre el soporte de `G` (`m_closed`), que es asociativa (`m_assoc`), que la unidad (**1**) también está en el soporte (`one_closed`), y que la unidad se comporta como tal tanto a derecha como a izquierda para todos los elementos del soporte (`l_one` y `r_one`). Para definir las condiciones anteriores hemos hecho uso de un **locale**, una especie de módulo en el que podemos, por ejemplo, fijar ciertas variables (en el ejemplo anterior `G`) y declarar ciertas premisas sobre las mismas (`m_closed`, `m_assoc...`).

De la definición anterior podemos obtener la de *monoide abeliano* imponiendo que dicha estructura debe satisfacer las propiedades características de un monoide, pero para su operación propia `add` (también denotada como $\oplus$) y la constante `zero` (o **0**) (previamente debemos haber definido un tipo de dato registro que contenga dichos campos), y además que la adición sea conmutativa:

```
locale abelian_monoid =
  fixes G (structure)
  assumes a_monoid:
     "monoid (| carrier = carrier G, mult = add G, one = zero G |)"
  and a_comm: "⟦x ∈ carrier G; y ∈ carrier G⟧ ⟹ x ⊕ y = y ⊕ x"
```

Añadiendo una premisa adicional a la definición de monoide, que impone que todos los elementos del dominio son unidades (es decir, poseen un elemento que actúa como inverso a derecha e izquierda) obtenemos la definición de *grupo*[3]:

[3]Otra posibilidad para definir la estructura de grupo es añadir una nueva operación *inv* que a cada elemento del grupo le asigne su inverso. De nuevo, la implementación propuesta trata de seguir las definiciones presentadas en [27]. En general, las estructuras algebraicas admiten caracterizaciones muy variadas, dependiendo, por ejemplo, del texto que sigamos o del propósito final de nuestra representación. Si queremos trabajar con distintas representaciones de una misma estructura algebraica el sistema nos provee con mecanismos que permiten identificarlas o probarlas equivalentes.

```
definition  Units :: "'a monoid => 'a set"
   where "Units G == {y. y ∈ carrier G ∧
    (∃x ∈ carrier G. x ⊗ y = 𝟭 ∧ y ⊗ x = 𝟭)}"

locale group = monoid +
  assumes Units: "carrier G <= Units G"
```

Si empleamos la definición de grupo para definir un grupo sobre la operación `add` y `zero`, y le añadimos la propiedad conmutativa, obtenemos la definición de *grupo abeliano*.

Para conseguir el tipo de dato que representará la estructura de *anillo* debemos añadir al registro inicial (`monoid`) dos campos adicionales, representando la parte aditiva del mismo. De nuevo introducimos notaciones abreviadas para las nuevas operaciones ($\oplus$, $\mathbf{0}$):

```
record 'a ring = "'a monoid" +
  zero :: 'a ("𝟬ı")
  add :: "['a, 'a] ⇒ 'a" (infixl "⊕ı" 65)
```

La estructura anterior, como en el caso de los monoides, define el *tipo de dato* que debe tener un anillo en nuestro sistema. Como se puede ver, la hemos definido haciendo uso de una especie de *herencia* (del registro `monoid`) disponible para registros en el sistema. Esto es una gran ventaja desde el punto de vista de la *reutilización de código*, ya que el tipo de dato anillo ha sido definido como un monoide con ciertas operaciones adicionales (como un subtipo, en jerga propia de la programación orientada a objetos). De este modo, y gracias al polimorfismo de parámetros de las funciones en HOL, cualquier función o predicado que admita una variable del tipo de dato *monoide*, admitirá también variables del tipo de dato *anillo*.

Aparte del tipo de dato, debemos dar una definición que caracterice las estructuras que realmente satisfacen las condiciones de un anillo:

```
locale ring = abelian_group R + monoid R for R (structure) +
  assumes l_distr: "[| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |]
      ⟹ (x ⊕ y) ⊗ z = x ⊗ z ⊕ y ⊗ z"
    and r_distr: "[| x ∈ carrier R; y ∈ carrier R; z ∈ carrier R |]
      ⟹ z ⊗ (x ⊕ y) = z ⊗ x ⊕ z ⊗ y"
```

De nuevo para la definición de las propiedades de anillo *reutilizamos* definiciones anteriores, minimizando así el esfuerzo que conlleva realizar las mismas. La forma de definir las propiedades de un anillo es asumir que el mismo es un grupo abeliano con respecto a sus operaciones aditivas ($\mathbf{0}$ y $\oplus$), que es un monoide con respecto a las operaciones multiplicativas ($\mathbf{1}$ y $\otimes$), y finalmente que satisface la distributividad del producto con respecto a la suma, tanto a izquierda (`l_distr`) como a derecha (`r_distr`). Añadiendo una nueva premisa sobre la conmutatividad de $\otimes$, obtenemos la definición de anillo conmutativo.

4.2.2. REPRESENTACIÓN DE POLINOMIOS

Nuestro próximo objeto de interés serán los polinomios univariados con coeficientes en un anillo R. La representación de polinomios univariados posiblemente se ofrezca a más diversas interpretaciones que la propia de las estructuras algebraicas. Representaciones naturales podrían resultar, entre otras, listas de pares coeficiente-potencia, vectores de pares, o funciones de los naturales en el dominio de los coeficientes. De entre estas distintas opciones nos quedaremos con la última, ya que la implementación de las operaciones (como la suma de polinomios, multiplicación por un escalar o el producto de polinomios), como veremos más adelante, se puede hacer de forma más natural. Además, esta forma sigue fielmente la definición de polinomios tal y como son introducidos en [27].

En la representación escogida, por ejemplo, el polinomio $\mathbf{0}_R \cdot x^0$ admitiría la representación $(\lambda i.\, \mathbf{0}_R)$ y el polinomio $\mathbf{1}_R \cdot x^1$ admitiría la representación ($\lambda i.$ `if` $i =$ 1 `then` $\mathbf{1}_R$ `else` $\mathbf{0}_R$). Se puede notar que hay otras funciones que podrían representar a los anteriores polinomios (dicho de otra forma, no tenemos unicidad de representación), pero este problema lo evitamos gracias a que en Isabelle/HOL la igualdad definida para funciones es la *igualdad extensional*, por la cual si dos funciones coinciden en todos los elementos de su dominio, en este caso los naturales, serán iguales (y por tanto los polinomios que representan dichas funciones serán iguales).

Para poder definir los polinomios como funciones de los naturales en el dominio de los coeficientes primero debemos añadir algunas condiciones adicionales. En primer lugar, las funciones que utilicemos deberán tener un dominio finito (si no, estaríamos trabajando con series indexadas por los naturales, además de con polinomios). Para conseguir esta propiedad imponemos la siguiente condición, que expresa que las funciones `f` que usaremos, por encima de un cierto número natural `n`, serán siempre iguales a $\mathbf{0}_R$:

```
locale bound =
  fixes n :: nat
   and f :: "nat ⇒ 'a"
  assumes bound: "∀m. n < m ⟹ f m = 0_R"
```

Estas variables (`n`, `f`) podrán ser posteriormente reemplazadas (o instanciadas) por valores *ad-hoc* en el contexto que nos interese. Por ejemplo, por medio de una premisa como `bound 12 p` estamos expresando la condición de que una función `p`, que a cada natural le asigna un coeficiente de tipo `'a`, verifica que `p m` $= \mathbf{0}_R$ para cualquier `m` > 12 (o, pensando en `p` como en un polinomio, que `p` es un polinomio de grado menor o igual que 12).

Con la condición expresada por la anterior estructura podemos dar una definición del conjunto de los polinomios (o `up`, *univariate polynomials*, para abreviar). Los polinomios definidos tienen sus coeficientes sobre un anillo (conmutativo) dado, de ahí que en la definición del conjunto `up` el primer elemento sea de tipo `('a ring)`, y en la definición de las propiedades utilicemos `R`. Con respecto a su tipo, los polinomios univariados serán un (sub)conjunto de las funciones que a cada natural le asignan un elemento del tipo del anillo `(nat ⇒ 'a) set`. Las propiedades que debe verificar

una de dichas funciones para considerarse en el conjunto de los polinomios son que a cada elemento del conjunto de los números naturales (*UNIV*) le debe asignar un elemento del soporte del anillo (*f ∈ UNIV → carrier R*) y que debe existir una cota (*n*) a partir de la cual dichas funciones sean siempre iguales a $\mathbf{0}_R$ (es decir, *∃n. bound n f*).

Los elementos de dicho conjunto serán los que consideremos en adelante polinomios univariados (o *up R*, cuando el anillo *R* sea explícito):

```
definition up :: "'a ring ⇒ (nat => 'a) set"
  where up_def: "up R == {f. f ∈ UNIV → carrier R ∧ (∃n. bound n f)}"
```

Veamos ahora algunas de las operaciones sobre los mismos. Por ejemplo, la suma de dos polinomios *p* y *q* será la función que a cada número natural *i* le asigne la suma (en el anillo subyacente *R*) de los coeficientes de *p* y *q* para dicho *i*. La misma admite también la notación $\oplus_P$, donde *P* representa el anillo de los polinomios:

```
add = (λp∈up R. λq∈up R. λi. p i ⊕_R q i)
```

Del mismo modo, los polinomios constantes 0 y 1 son dos funciones que asignan a cada número natural i el $\mathbf{0}_R$ y el $\mathbf{1}_R$ en grado 0 respectivamente (con $\mathbf{0}_R$ y $\mathbf{1}_R$ la unidad aditiva y multiplicativa de *R*). Los mismos admiten la notación $\mathbf{0}_P$ y $\mathbf{1}_P$ respectivamente:

```
one = (λi. if i = 0 then 1_R else 0_R)
zero = (λi. 0_R)
```

La multiplicación de dos polinomios *p* y *q* ($\otimes_P$) es un polinomio que en cada grado *n* tiene como coeficiente el sumatorio desde *0* hasta *n* del producto de los coeficientes *p i* y *q (n-i)* (para lo que hacemos uso del operador sumatorio sobre un conjunto *{0...n}*):

```
mult = (λp∈up R. λq∈up R. λn. ⨁_R i ∈ {0..n}. p i ⊗_R q (n-i))
```

La multiplicación de un escalar *a* por un polinomio *p* (es decir, *a* $\odot_P$ *p*) se define como la función que a cada número natural *i* le asigna el producto de *a* por el coeficiente *p i*:

```
smult = (λa∈carrier R. λp∈up R. λi. a ⊗_R p i)
```

La forma general de trabajar en un asistente de demostración nos lleva primero a demostrar que las anteriores operaciones, con coeficientes sobre un anillo cualquiera *R*, son *cerradas* con respecto al conjunto de los polinomios; por ejemplo, que el polinomio que hemos definido como $\mathbf{1}_P$ realmente es miembro del conjunto *up R*. Para demostrar este hecho nos basta con probar que está acotado, cuya demostración es directa en el sistema (el sistema automáticamente ha provisto una cota para la función dada y comprobado que verifica la propiedad que hemos impuesto sobre los polinomios, por medio del uso del comando *force*). Aunque no aparezca en la colección de premisas del lema, la premisa de que *R* sea un anillo la hemos hecho explícita en nuestro contexto de trabajo:

```
lemma up_one_closed: "1_P ∈ up R"
  using up_def by force
```

Finalmente, consideraremos también como una operación adicional sobre polinomios el *coeficiente* de un polinomio `p` (sobre un anillo subyacente `R`) en un número natural `n`:

```
coeff = (λp∈up R. λn. p n)
```

La operación `coeff`, junto con las anteriores, son definidas como campos de un registro en el que hemos introducido el conjunto de los polinomios válidos y sus operaciones. De este modo, cada vez que queramos conocer el coeficiente de un determinado polinomio `p` en un cierto grado `n`, deberemos hacer explícitos:

- El registro que representa el anillo de polinomios univariados con el que trabajamos (por ejemplo `P`, donde el anillo subyacente será `R`).
- El polinomio `p` cuyo coeficiente queremos averiguar.
- El grado `n` que requerimos.

A pesar de que la notación pueda parecer un poco oscura a priori, resulta bastante natural y a la larga es bastante fácil y conveniente de utilizar. Además, coincide con lo que hacemos en demostraciones *sobre papel*, aunque en las mismas a menudo obviemos algunos de dichos detalles (por ejemplo, los anillos de polinomios o de coeficientes sobre los que trabajamos aparecerían muchas veces implícitos, aunque estuviésemos utilizando sus propiedades de forma explícita). El registro `P` así definido se demuestra que es un anillo (siempre y cuando `R` lo sea), lo cual nos permite usar todos los resultados demostrados para un anillo genérico sobre los elementos de `P`.

A partir de aquí supondremos que trabajamos en un contexto en el que `R` es un anillo conmutativo cualquiera. Esta premisa se puede definir por medio de un `locale`, que nos permite fijar variables (en este caso `R`) y asumir ciertas premisas sobre la misma. Definimos `P` como el anillo de polinomios univariados sobre `R`. Ya hemos visto que se puede demostrar que `P` es un anillo (bajo la premisa de que `R` lo sea). En adelante deberemos prestar atención en nuestra notación a distinguir entre las operaciones y constantes propias de `R` (como $\mathbf{0}_R$, $\otimes_R$) y aquéllas de `P` ($\mathbf{0}_P$, $\otimes_P$). Por medio del contexto generado por el **locale**, conseguimos un comportamiento similar al que en un texto de matemáticas hacen sentencias como «de aquí en adelante, y salvo que se indique lo contrario, suponemos que `R` es un anillo, y que `P` es el anillo de polinomios univariados sobre `R`».

A partir de la definición dada de coeficiente, el *grado* (o `deg`) sobre un anillo `R` de un polinomio `p` se puede definir como el menor natural a partir del cual todos los coeficientes de `p` son iguales a $\mathbf{0}_R$, es decir, recuperando la definición de `bound` que introdujimos previamente:

```
definition deg :: "['a ring, nat => 'a] ⇒ nat"
  where "deg R p == LEAST n. bound n p"
```

En la definición anterior es necesaria la presencia del anillo subyacente `R` ya que la definición de `bound` exige que todos los coeficientes a partir de un determinado `n` sean iguales a $\mathbf{0}_R$.

Finalmente, para simplificar nuestra notación, haremos uso también de la noción de *coeficiente principal* (o `lcoeff`) de un polinomio `p`, definido como el coeficiente de `p` en el grado del mismo. Dicha noción la podemos introducir en el sistema como una simple abreviatura a partir de las nociones ya definidas:

```
abbreviation lcoeff :: "(nat ⇒ 'a) ⇒ 'a"
  where "lcoeff p == coeff P p (deg R p)"
```

4.2.3. Representación y demostración del resultado

Pasamos ahora a mostrar el enunciado del Teorema 1 tal y como lo hemos formalizado en Isabelle/HOL. De la notación que podemos encontrar en el enunciado la única operación que no hemos introducido previamente es $(\hat{})_R$, que representa la potencia en el anillo `R`:

```
lemma long_div_theorem:
  assumes "g ∈ carrier P"
  and  "f ∈ carrier P"
  and "g ≠ 0_P"
  shows "∃ q r (k::nat). (q ∈ carrier P) ∧ (r ∈ carrier P)
∧ (lcoeff g)(^)_R k ⊙_P f = g ⊗_P q ⊕_P r ∧ (r = 0_P | deg R r < deg R g)"
```

El enunciado del mismo se puede entender como «si asumimos que `g` y `f` son elementos del soporte del anillo de los polinomios en una variable sobre un anillo `R`, y que `g` es distinto de $\mathbf{0}$, se puede demostrar que existen `q`, `r` y `k`, tales que `q` y `r` están en el soporte del conjunto de polinomios y `k` es un natural, de modo que el coeficiente principal de `g` elevado a `k` multiplicado por `f` es igual a `g` multiplicado por `q` más `r`, donde `r` es el polinomio nulo o un polinomio de menor grado que `g`». Se puede comprobar la adecuación del enunciado dado con la del Teorema 1.

La demostración del teorema está basada en la que hemos presentado antes. En primer lugar debemos aplicar inducción en el grado de `f`. La regla de inducción requiere que el elemento `f` que estamos considerando sea un polinomio. En el comando **proof** `(...)` especificamos la regla de demostración que utilizamos. En este caso es una regla de inducción aplicada sobre el grado de `f`, donde `f` debe ser una variable libre (`arbitrary`), ya que luego querremos aplicar la hipótesis de inducción no sobre `f`, sino sobre un cierto `f1` de grado inferior a `f`. De las diversas reglas de inducción que contiene el sistema, especificamos que queremos usar la regla de inducción `nat_less_induct`. Dicha regla de inducción especifica que dado un predicado sobre los naturales, del cual sabemos que cuando es cierto para cualquier número `m` menor que un cierto `n` entonces es cierto para `n`, entonces dicho predicado es cierto para cualquier número natural `k`. Aplicando esta regla de inducción, el primer paso de la demostración será el siguiente:

```
 using "f ∈ carrier P"
proof (induct "deg R f" arbitrary: "f" rule: nat_less_induct)
```

El resultado que debemos probar ahora es que, sabiendo que el teorema es cierto para cualquier polinomio `f1` de grado menor que el grado de `f`, el resultado es cierto

para `f`. Siguiendo de nuevo la demostración en [27], debemos distinguir en casos por la siguiente condición booleana:

```
proof (cases "deg R f < deg R g")
```

El caso en el que el grado de `f` es menor que el de `g`, que se resuelve considerando `q` igual a $\mathbf{0}_P$, `r` igual a `f` y `k` igual a 0, lo podemos demostrar en el sistema de forma similar a como lo hicimos en la demostración matemática, proveyendo al sistema con dicha instancia de `q`, `r` y `k` (`?pred` es una abreviatura del predicado asociado al existencial en el enunciado del teorema, del mismo modo que `?thesis` actúa de abreviatura del resultado que debemos demostrar):

```
case True
have "?pred 0_P f 0" using True by force
then show ?thesis by fast
```

El caso restante es cuando `deg R g` $\leq$ `deg R f`. Como hicimos en la demostración matemática, distinguimos de nuevo dos casos.

En el caso en que `deg R f` $= 0$, entonces `deg R g` $= 0$, y por tanto podemos encontrar la siguiente solución al predicado existencial:

```
have "?pred f 0_P 1"
 using deg_zero_impl_monom [OF g_in_P deg_g]
 using sym [OF monom_mult_is_smult [OF coeff_closed [OF g_in_P, of 0] f_in_P]]
 using deg_g by simp
then show ?thesis by blast
```

En este caso la demostración ha sido un poco más compleja. En primer lugar hemos debido indicarle al sistema que, siendo `g` un polinomio de grado 0, entonces puede ser visto como un monomio de grado 0 (`deg_zero_impl_monom`). Posteriormente, hemos debido indicarle al sistema que multiplicar por un monomio de grado 0 es lo mismo que multiplicar por un escalar (`monom_mult_is_smult`). Con eso, y el hecho de que el grado de `g` sea igual a 0 (`deg_g`), el sistema es capaz de demostrar que `?pred` es cierto, y por tanto dispone de una instancia explícita para probar que el teorema en este caso también lo es.

Finalmente, nos queda el caso en el que `deg R g` $\leq$ `deg R f` y `deg R f` $\neq 0$. Definimos $f_1(x) = b_m f(x) - a_n x^{n-m} g(x)$, con b_m el coeficiente principal de $g(x)$ y a_n el coeficiente principal de $f(x)$. Podemos notar que el polinomio $f_1(x)$ está bien definido, ya que $\deg g(x) \leq \deg f(x)$. También debemos demostrar que el grado de $f_1(x)$ es menor (estricto) que el grado de $f(x)$, lo cual es cierto ya que los coeficientes principales de $b_m f(x)$ y $a_n x^{n-m} g(x)$ son iguales ($b_m a_n$) y por tanto se anulan. Esto es cierto siempre y cuando el grado de $f(x)$ no sea 0, caso que se obvia en la demostración en [27], y que nosotros hemos considerado antes.

Como $\deg f_1(x) < \deg f(x)$, podemos aplicar la hipótesis de inducción provista por la regla `nat_less_induct`, la cual nos permitirá conseguir candidatos $q_1(x)$, $r_1(x)$ y k_1 tales que $b_m^{k_1} f_1(x) = g(x)q_1(x) + r_1(x)$. De ahí podemos conseguir, por simple reescritura, polinomios $q(x)$ y $r(x)$ y un natural k tales que

$$b_m^{k_1+1} f(x) = b_m^{k_1} a_n x^{n-m} g(x) + g(x)q_1(x) + r_1(x) = g(x)q(x) + r_1(x).$$

Presentamos a continuación sólo algunos detalles de la demostración de esta última parte en Isabelle. Por medio de los mandatos **let** creamos abreviaturas *ad hoc* para nuestra demostración. Por medio de ... hemos dejado señaladas ciertas partes de la demostración que se pueden consultar en la versión completa de la misma (cuya referencia se puede encontrar más adelante) pero que consideramos menos relevantes:

```
let ?lg = "lcoeff g" and ?lf = "lcoeff f"
let ?k = "1::nat"
let ?q = "monom P (?lf) (deg R f - deg R g)"
let ?f1 = "(g ⊗P ?q ) ⊕P ⊖P (?lg ⊙P f)"
have exist: " ?lg (^) ?k ⊙P f = g ⊗P ?q ⊕P ⊖P ?f1"
 by ...
have deg_remainder_l_f: "deg R (⊖P ?f1) < deg R f"
 by ...
then obtain q' r' k'
 where rem_desc: "?lg (^) (k'::nat) ⊙P (⊖P ?f1) = g ⊗P q' ⊕P r'"
 and rem_deg: "(r' = 0P ∨ deg R r' < deg R g)"
 and q'_in_carrier: "q' ∈ carrier P" and r'_in_carrier: "r' ∈ carrier P"
 using "1.hyps" using f1_in_carrier by blast
show ?thesis
 proof (rule exI3 [of _ "((?lg (^) k') ⊙P ?q ⊕P q')" r' "Suc k'"])
 ...
 qed
```

La demostración completa es más extensa que la equivalente en matemáticas. En particular, esta última parte requiere manipulaciones simbólicas que deben ser aplicadas de forma organizada y detallada (por ejemplo, las reglas de asociatividad o introducir o extraer factor común en las expresiones). Así y todo, la demostración total ocupa unos dos folios, menos de 100 líneas de código (a comparar con la media página que ocupaba la demostración original en [27]).

La demostración en Isabelle se podría hacer incluso en menos espacio, aunque disminuiría la legibilidad de la misma. Lo que hemos primado, en este caso, es que el código obtenido, con un mínimo conocimiento de la sintaxis del sistema, no sea especialmente complicado y puede ser revisado con facilidad por cualquier lector con un cierto conocimiento de matemáticas (o por nosotros en el futuro). La demostración completa puede consultarse en `http://isabelle.in.tum.de/library/HOL/HOL-Algebra/UnivPoly.html`.

Además, la demostración formal nos ha permitido completar ciertos detalles que en la demostración matemática se habían obviado (como el caso en el que $\deg f(x) = 0$), y también nos ha permitido obtener un mejor conocimiento de en qué orden se aplicaban los distintos pasos en la demostración matemática (por ejemplo, cuándo se debía aplicar inducción y separación en casos, hecho que no estaba completamente claro en el texto original).

Una última ventaja de la demostración formal es que puede ser comprobada en cualquier ordenador que tenga instalado Isabelle, lo cual hace que no tengamos que

confiar en la veracidad del teorema, sino que nosotros mismos podamos *ejecutar* o llevar a cabo la prueba en nuestra máquina paso a paso y ver cómo ha sido realizada.

5. CONCLUSIONES

En el presente artículo hemos intentado presentar de forma genérica el ámbito que podríamos denominar como *demostración asistida por ordenador*, y hemos mostrado diversas posibilidades en esta área. En particular hemos distinguido tres tipos de uso. Por un lado, hemos mostrado cómo programas de cálculo simbólico pueden ser utilizados como *oráculos* para conjeturar resultados en matemáticas (que luego pueden ser demostrados). Aún más, hoy en día podemos encontrar programas que son capaces de observar, por ejemplo, secuencias de números, y plantear conjeturas sobre los mismos [12], lo cual va un paso más allá en el uso de los ordenadores en las matemáticas, dejando a las propias máquinas la capacidad de *inventar* resultados, en un área conocida como *creatividad computacional*; otro campo de aplicación bien conocido en esta línea es el descubrimiento de teoremas en geometría (véase [8, 43, 13]). En segundo lugar, hemos visto cómo la capacidad de cálculo de los ordenadores los hace también adecuados (si no necesarios) para realizar ciertas comprobaciones exhaustivas en ámbitos muy extensos (como el teorema de los cuatro colores o la solución de la conjetura de Kepler). Parece difícil que dichos resultados se hubieran alcanzado sin la intervención de aplicaciones informáticas.

Finalmente, hemos presentado un tercer tipo de aplicación de programas informáticos a la demostración de resultados en matemáticas. Son los conocidos como *asistentes de demostración*. Sus aplicaciones van más allá de las matemáticas, y se utilizan con éxito también en la verificación de algoritmos, programas o protocolos de seguridad o de comunicación. Dentro de las matemáticas, su *capacidad* está empezando a quedar patente por la envergadura de los resultados que se han abordado con ellos. También hay diversos argumentos que se pueden esgrimir en favor de la *fiabilidad* de los mismos. Por ejemplo, la formalización de demostraciones de teoremas a veces ayuda a detectar ciertas imprecisiones u omisiones en las demostraciones originales. Además, el núcleo o base lógica de estos asistentes suele ser lo suficientemente pequeño y simple como para que se pueda comprobar de diversas formas la ausencia de errores en el mismo. Si bien los asistentes no están libres de *errores* informáticos, éstos generalmente tienen que ver con algunas de las facilidades de uso que contienen los mismos (no con el propio núcleo lógico), y las amplias comunidades de usuarios son capaces de detectarlos y resolverlos con celeridad.

A pesar de todo ello, la comunidad matemática todavía mira a los asistentes con cierto escepticismo. Podemos apuntar varios motivos para ello, con los cuales los autores estamos plenamente de acuerdo. En primer lugar, la *diversidad* de herramientas. Sólo en este artículo ya hemos citado varios asistentes. Con un mínimo esfuerzo podríamos encontrar varias decenas disponibles, con su comunidad de usuarios correspondiente. Esto es debido, por ejemplo, a que muchos se desarrollan de forma experimental en centros de investigación, como proyectos de desarrollo de software, consiguiendo posteriormente una comunidad de usuarios estable o creciente que exige su mantenimiento. Otros se desarrollan para verificar software pero por

sus características son también usados en la formalización de matemáticas. De forma más sencilla todavía, algunos implementan lógicas distintas, y por tanto ofrecen distintas capacidades. En cualquier caso, una de las consecuencias de esta diversidad es la dispersión de esfuerzos dentro de la comunidad, y la sensación de falta de cohesión que se transmite hacia fuera de la misma. En segundo lugar, su *dificultad de uso*. Si bien en los ejemplos que hemos introducido en este artículo hemos tratado de presentar código de la forma más «comprensible» que hemos podido, tanto la sintaxis como la semántica y el propio funcionamiento de los asistentes es a menudo poco intuitivo, y su proceso de aprendizaje se puede considerar como bastante largo y complejo. Cada vez hay más documentación sobre los mismos (libros y tutoriales publicados por editoriales de prestigio), pero sigue siendo complicado comprender los entresijos de los sistemas si uno no cuenta con la asistencia de un experto. A pesar de ello, en algunos grados universitarios se usan asistentes de demostración en prácticas de lógica o de métodos formales [41], lo cual nos hace ser optimistas con respecto a este punto a corto o medio plazo. En tercer lugar, los sistemas de tipos y las lógicas subyacentes que implementan los asistentes de demostración tienen como consecuencia una serie de *limitaciones intrínsecas* difíciles de evitar. Por ejemplo, en los asistentes que implementan lógica de primer orden puede resultar difícil (al menos de una forma natural) representar y demostrar ciertos resultados de Álgebra Abstracta o de Teoría de Categorías. Para el lector interesado en algunas de estas limitaciones recomendamos [51].

El *objetivo* que se fija la comunidad alrededor de los asistentes de demostración es que algún día los mismos se puedan utilizar como herramienta en la actividad diaria del matemático, de un modo similar a como los programas de cálculo simbólico son utilizados de forma habitual para realizar cálculos, explorar nuevas hipótesis o asegurarnos de la certeza o falsedad de ciertas conjeturas (como el contraejemplo recientemente encontrado a la conjetura de Hirsch [45]). Así, los asistentes deberían ser útiles para descartar casos triviales o casi directos en demostraciones, presentarnos el estado de una demostración después de algún paso complejo, asegurarnos que el desarrollo de una demostración ha sido el correcto, sugerirnos el camino a seguir en una demostración, ser capaces de refutar tesis erróneas por medio de contraejemplos construidos por el propio asistente, o capacitar a los revisores de una revista para reproducir una demostración enviada a la misma. Es difícil aventurar la cercanía de esta meta, aunque la evolución de este campo de investigación entre las Matemáticas y la Informática por el momento no se detiene.

Agradecimientos

Expresamos nuestro agradecimiento a Julio Rubio por sus comentarios y sugerencias a lo largo de la redacción de este trabajo. También agradecemos al *referee* anónimo sus aportaciones, que han mejorado la versión definitiva de este artículo.

REFERENCIAS

[1] M. ANDRÉS, L. LAMBÁN Y J. RUBIO, Executing in Common Lisp, proving in ACL2, *Calculemus 2007*, Vol. 4573, 1–12, *Lecture Notes in Computer Science*, Springer, 2007.

[2] K. APPEL Y W. HAKEN, Every map is four colourable, *Bulletin of the American Mathematical Society* **82** (1976), 711–712.

[3] J. ARANSAY, C. BALLARIN Y J. RUBIO, A mechanized proof of the Basic Perturbation Lemma, *Journal of Automated Reasoning* **40** (4) (2008), 271–292.

[4] J. ARANSAY, C. BALLARIN Y J. RUBIO, Generating certified code from formal proofs: a case study in homological algebra, *Formal Aspects of Computing* **22** (2) (2010), 193–213.

[5] J. ARANSAY Y C. DOMÍNGUEZ, Formalizing simplicial topology in Isabelle/HOL and Coq. En *Contribuciones científicas en honor de Mirian Andrés Gómez*, 21–42, L. Lambán, A. Romero y J. Rubio (eds.), Universidad de La Rioja, 2010.

[6] N. BOURBAKI, *Eléments de mathématique, Théorie des ensembles*, Hermann, 1970.

[7] R. S. BOYER, The QED Manifesto, *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, Vol. 814, 238–251, *Lecture Notes in Computer Science*, Springer, 1994.

[8] S. C. CHOU, *Mechanical geometry theorem proving*, Vol. 41, *Mathematics and its Applications*, D. Reidel Publishing Company, 1988.

[9] A. CHURCH, A formulation of the simple theory of types, *The Journal of Symbolic Logic* **5** (2) (1940), 56–68.

[10] Ó. CIAURRI Y J. L. VARONA, ¿Podemos fiarnos de los cálculos efectuados con ordenador?, *La Gaceta de la RSME* **9** (2) (2006), 483–514.

[11] F. R. COHEN Y R. LEVI, On the homotopy type of infinite stunded projective spaces, *Progress in Mathematics* **196** (2001), 79–90.

[12] S. COLTON, *Automated Theory Formation in Pure Mathematics*, Springer, 2002.

[13] G. DALZOTTO Y T. RECIO, On protocols for the automated discovery of theorems in elementary geometry, *Journal of Automated Reasoning* **43** (2) (2009), 203–236.

[14] C. DOMÍNGUEZ Y J. RUBIO, Effective homology of bicomplexes, formalized in Coq, *Theoretical Computer Science* **414** (11) (2011), 962–970.

[15] X. DOUSSON, F. SERGERAERT Y Y. SIRET, The Kenzo Program, `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`, Institut Fourier, Grenoble, 1999.

[16] ForMath project, `http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/`, 2010.

[17] G. FREGE, *Begriffsschrift: eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*, Halle, 1879.

[18] M. GARDNER, *Logic Machines and Diagrams*, McGraw-Hill, New York, 1958.

[19] G. Gonthier, Formal proof — The Four-Color Theorem, en [48], 1382–1393.

[20] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi y L. Théry, A modular formalisation of finite group theory, *Theorem Proving in Higher Order Logics 2007*, Vol. 4732, 86–101, *Lecture Notes in Computer Science*, Springer, 2007.

[21] T. C. Hales, A proof of the Kepler conjecture, *Annals of Mathematics, Second Series* **163** (3) (2005), 1065–1185.

[22] T. C. Hales, Formal Proof, en [48], 1370–1380.

[23] J. Harrison, Formal Proof — Theory and Practice, en [48], 1395–1406.

[24] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, 2009.

[25] P. J. Heawood, Map-colour theorems, *Quarterly Journal of Mathematics* **24** (1890), 332–338.

[26] H. Hudson, No basta con cuatro colores, *La Gaceta de la RSME* **8** (2) (2005), 361–368.

[27] N. Jacobson, *Basic Algebra I*, Second Edition, Freeman and Company, 1985.

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

[29] A. B. Kempe, On the geographical problem of four-colors, *American Journal of Mathematics* **2** (1879), 193–200.

[30] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch y S. Winwood, seL4: Formal verification of an operating-system kernel, *Communications of the ACM* **53** (6) (2010), 107–115.

[31] C. W. H. Lam, How reliable is a computer-based proof?, *The Mathematical Intelligencer* **12** (1) (1990), 8–12.

[32] X. Leroy, Formal verification of a realistic compiler, *Communications of the ACM* **52** (7) (2009), 107–115.

[33] D. Mackenzie, *Mechanizing Proof: Computing, Risk and Trust*, The Mit Press, 2001.

[34] W. McCune, Solution of the Robbins Problem, *Journal of Automated Reasoning* **19** (1997), 263–276.

[35] W. McCune, Prover9 and Mace4, `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[36] P. J. Miana y N. Romero, La historia de la conjetura de Kepler. En *Contribuciones científicas en honor de Mirian Andrés Gómez*, 367–374, L. Lambán, A. Romero y J. Rubio (eds.), Universidad de La Rioja, 2010.

[37] T. Nipkow, L. C. Paulson y M. Wenzel, *Isabelle/HOL: A proof assistant for higher order logic*, Vol. 2283, *Lecture Notes in Computer Science*, Springer, 2002.

[38] S. Obua y T. Nipkow, Flyspeck II: the basic linear programs, *Annals of Mathematics and Artificial Intelligence* **56** (2009), 245–272.

[39] S. OBUA Y S. SKALBERG, Importing HOL into Isabelle/HOL, *Automated Reasoning, Third International Joint Conference, IJCAR 2006*, Vol. 4130, 298–302, *Lecture Notes in Computer Science*, Springer, 2006.

[40] L. PAULSON, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (3) (1989), 363–397.

[41] B. PIERCE, Proof assistants as teaching assistants: a view from the trenches, *Interactive Theorem Proving 2010*, Vol. 6172, 8, *Lecture Notes in Computer Science*, Springer, 2010.

[42] R. POLLACK, How to believe a machine-checked proof, En *Twenty Five Years of Constructive Type Theory*, Oxford University Press, 1998.

[43] T. RECIO Y M. P. VÉLEZ, Automatic discovery of theorems in elementary geometry, *Journal of Automated Reasoning* **23** (1) (1999), 63–82.

[44] J. RUBIO Y F. SERGERAERT, Constructive Algebraic Topology, *Bulletin des Sciences Mathématiques* **126** (2002), 389–412.

[45] F. SANTOS, Sobre un contraejemplo a la conjetura de Hirsch, *La Gaceta de la RSME* **13** (3) (2010), 525–538.

[46] V. A. SMIRNOV Y F. SERGERAERT, The homology of iterated loop spaces, *Forum Mathematicum* **14** (2002), 345–381.

[47] R. THIELE Y L. WOS, Hilbert's Twenty-Fourth Problem, *Journal of Automated Reasoning* **29** (2002), 67–89.

[48] VARIOS AUTORES, A special issue on Formal Proof, *Notices of the American Mathematical Society* **55** (11) (2008).

[49] F. WIEDIJK, Comparing mathematical provers, *Mathematical Knowledge Management, 2nd International Conference, Proceedings*, 188–202, Springer, 2003.

[50] F. WIEDIJK (ED.), *The Seventeen Provers of the World*, with a foreword by D. S. Scott, Lecture Notes in Artificial Intelligence, 3600, Springer, 2006.

[51] F. WIEDIJK, The QED Manifesto revisited, *Studies in Logic, Grammar and Rhetoric* **10** (23) (2007), 121–133.

[52] F. WIEDIJK, Página web personal, `http://www.cs.ru.nl/~freek/100/`

[53] A. WILES, Modular elliptic curves and Fermat's Last Theorem, *Annals of Mathematics, Second Series* **141** (3) (1995), 443–551.

[54] L. WOS Y G. W. PIEPER, *Automated Reasoning and the Discovery of Missing and Elegant Proofs*, Rinton Press, 2003.

JESÚS MARÍA ARANSAY AZOFRA, DPTO. DE MATEMÁTICAS Y COMPUTACIÓN, UNIVERSIDAD DE LA RIOJA
Correo electrónico: `jesus-maria.aransay@unirioja.es`
Página web: `http://www.unirioja.es/cu/jearansa`

CÉSAR DOMÍNGUEZ PÉREZ, DPTO. DE MATEMÁTICAS Y COMPUTACIÓN, UNIVERSIDAD DE LA RIOJA
Correo electrónico: `cesar.dominguez@unirioja.es`
Página web: `http://www.unirioja.es/cu/cedomin`