Escuela Técnica Superior de Ingeniería Informática

Máster Universitario en Lógica,
Computación e Inteligencia Artificial

Trabajo de Fin de Máster

Formalización de conceptos básicos del Álgebra Lineal en ACL2

Autor:
José Luis Pro Martín

Tutores:
José Luis Ruiz Reina
Francisco Jesús Martín Mateos

Departamento de Ciencias de la Computación
e Inteligencia Artificial

Diciembre de 2014

Probamos por medio de la lógica,
pero descubrimos por medio de la intuición.

(Henri Poincaré)

La única forma lógica de vivir,
es descubrir lo que te gusta...
y tratar de hacerlo.

(Anónimo internauta)

Agradecimientos

Miguel Ángel Gutiérrez Naranjo, Agustín Riscos Núñez, Francisco José Romero Campero, Mario de Jesús Pérez Jiménez, José Francisco Quesada Moreno, Joaquín Borrego Díaz, Antonia Chávez González, Francisco Jesús Martín Mateos y José Luis Ruiz Reina.

Casi siempre las páginas de agradecimientos terminan con lo verdaderamente importante: aquellos a los que agradecemos por alguna razón el habernos encontrado en algún punto del camino que ha llegado hasta aquí. No quería que fuera así en esta ocasión. Habéis sido lo mejor del Máster que ahora acabo. Desde mi experiencia en el campo de la docencia tiendo a juzgar muy severamente a aquellos que me intentan enseñar algo, tanto en la forma como en el contenido. Podéis daros por satisfechos, ha sido un placer haber tenido la oportunidad de absorber de vuestros conocimientos todo lo que humanamente he podido. ¡Felicidades!

Pero eso no es todo. Desde el punto de vista humano me ha parecido estar siempre ante mis iguales y no ante mis evaluadores y jueces. Siempre con respeto y mostrando verdadero interés en que el alumnado aprenda. En definitiva, os habéis ganado mi respeto y admiración por *lo que hacéis y por cómo lo hacéis*. Hago extensivo mi agradecimiento al resto de componentes del departamento de Ciencias de la Computación e Inteligencia Artificial. No os conozco, es evidente, pero si vuestra calidad profesional y humana es parecida a lo que yo me he encontrado debo felicitaros a todos por el ambiente y las buenas sensaciones que habéis logrado transmitirme.

Debo destacar aquí el trabajo de los últimos meses realizado conmigo por parte de mis dos tutores en este Trabajo de Fin de Máster, José Luis Ruiz Reina y Francisco Jesús Martín Mateos. Vuestra experiencia en este campo ha sido crucial para que el enfoque de este trabajo haya sido el correcto. Pero además gracias por leer mis extensos correos, quedar para charlar del trabajo, y, sobre todo, en animarme a hacerlo (y a pararme los pies cuando correspondía).

Pero sí voy a dejar para el final el agradecimiento a ciertas personas por aguantar lo que no tendrían por qué: mis ojeras kilométricas, mis miradas perdidas en el infinito de un razonamiento que se escabuye, mis histéricas peticiones de un boli cuando quiero apuntar algo que se me acaba de ocurrir, y claro, no sólo aguantarme sino apoyarme en todo lo que se puede apoyar uno para seguir adelante. Nunca llegamos a saber con total seguridad qué cosas nos convienen en cada momento, pero hay una situación de la que sí me acuerdo:

Después de tres horas sin haber podido avanzar nada, se me acerca mi hija de seis años y se queda mirando la pantalla del portátil con cara de intensa y sincera preocupación hasta que, después de unos instantes, dice:

—Oye, Papi, ¿de verdad eso son matemáticas?

—Pues sí, hija, pero llevo haciendo el tonto un buen rato, ¿te apetece que juguemos a la pelota?

La forma en la que le cambió la cara no es expresable ni con palabras ni en ninguna lógica de cualquier orden. Ver esa nueva cara era, en verdad, lo que necesitaba en ese momento.

Simplemente gracias, Mari Carmen, y gracias, Pilariña.

Índice

Índice de figuras

1. Introducción y objetivos del trabajo

Para la obtención final del título oficial del Máster en Lógica, Computación e Inteligencia Artificial se necesita el desarrollo, redacción, presentación y posterior defensa de un Trabajo de Fin de Máster que refrende que el alumno ha asimilado los conceptos del plan de estudios de dicho máster.

Sirva, por tanto, el presente escrito, como la redacción de la memoria del Trabajo de Fin de Máster correspondiente al alumno José Luis Pro Martín, de título *"Formalización de conceptos básicos del Álgebra Lineal en ACL2"*.

Una de las asignaturas del máster, perteneciente al área de la *Lógica Matemática*, llamada "Razonamiento Asistido por Computador" trata sobre las características de un sistema (ACL2), que, además de ser un lenguaje de programación, es una lógica y un demostrador automático de teoremas y propiedades sobre esa lógica, en los algoritmos implementados en el lenguaje.

ACL2 cuenta con gran variedad de lemas y teoremas de muchas ramas de las matemáticas creados por la comunidad científica a lo largo sobre todo de las dos últimas décadas. Sin embargo, el Álgebra Lineal, y concretamente la aritmética de matrices como base del resto de este campo no ha sido formalizada, ni estudiada demasiado en este demostrador automático. El único antecedente válido en este sentido se da en el artículo escrito por Cowles, Gamboa y Van Baalen en 2003 (Gamboa, [6]). Desde entonces ACL2 ha evolucionado de forma que presenta algunas características nuevas y realmente muy útiles para tratar de atacar la formalización del álgebra matricial en ACL2.

Las mejoras se deben producir en al menos estos dos objetivos:

- La formalización debe ser más intuitiva y fácil de usar que la utilizada por Gamboa. La utilización de listas de asociación para representar matrices no es la más adecuada cuando normalmente se intenta acceder a la matriz directamente a través del índice de su fila y de su columna.
- Por otro lado, si se desea que los algoritmos implementados en ACL2 sirvan en la práctica, se pretende que la ejecución de dichos programas sean eficientes desde el punto de vista del tiempo de ejecución y de la memoria reservada de forma dinámica.

En la sección 2 veremos el sistema usado en una muy breve introducción. Se ha intentado una descripción escueta de las tres partes de las que consta ACL2, intentando sobre todo proporcionar ejemplos al lector sobre lo que es capaz de hacer este sistema. Una introducción más extensa y detallada se puede ver en [9].

En la sección 3 se explica con detalle las consideraciones de diseño necesarias para el desarrollo del trabajo haciendo especial hincapié en la formalización del propio concepto de matriz, pasando después a las funciones primitivas (más básicas posibles) para poder interactuar con objetos de este tipo. Posteriormente se definen funciones, cada vez más complejas, sobre matrices, como pueden ser la suma y la multiplicación. Por último, formalizaremos y definiremos el algoritmo de Gauss-Jordan para el cálculo de la matriz inversa. Se han seguido los apuntes de cátedra de la asignatura de "Álgebra Lineal" impartida en la Escuela de Ingeniería Informática de la Universidad de Sevilla ([5]).

En la sección 4 se detallan los teoremas y propiedades más importantes que se han podido demostrar en ACL2 sobre las definiciones y algoritmos definidos en la sección anterior. Se hace de forma incremental, desde el más sencillo hasta el último, bastante más complejo, que trata sobre la correción del algoritmo de Gauss-Jordan.

Por último en la sección 5 presentamos la forma de hacer tremendamente eficiente la ejecución de los algoritmos presentados mediante el uso de los stobjs abstractos.

El anexo A muestra, de forma esquemática, la mayor parte de los lemas intermedios que han hecho falta para realizar la demostración de los teoremas. En total se han conseguido más de 500 demostraciones con éxito en este Trabajo de Fin de Máster. El anexo B trata de la comparación de los algoritmos utilizados en este proyecto en relación al código de [6] y a la primera aproximación en cuanto al tiempo de ejecución y uso eficiente de la memoria.

Como la totalidad del trabajo se ha hecho programando en ACL2 se han incluido los *snippets* de código más importantes del proyecto. Para que la lectura de estos trozos de código sea más sencilla se ha usado el siguiente código de colores:

- `Color negro`: Para el uso de variables usadas en demostraciones y funciones.
- `Color rojo`: Para los delimitadores del código (sobre todo paréntesis).
- `Color azul`: Para las funciones *built-in* y predefinidas en ACL2, así como las palabras reservadas.
- `Color verde`: Para las funciones definidas y creadas en este trabajo.
- `Color marrón`: Para los nombres de los teoremas generados este proyecto.

Por ejemplo, para demostrar que $(A^T)^T = A$ se ha usado el siguiente código en ACL2:

```
(defthm pfm-idempotency-of-transpose
  (implies (matrixp A)
           (equal (transpose C (transpose B A)) A)))
```

2. Breve introducción a ACL2

ACL2 es el acrónimo de *A Computational Logic for Applicative Common Lisp*. Este sistema se puede usar para definir programas, por lo que, evidentemente, puede ser considerado como un lenguaje de programación, pero lo verdaderamente interesante de ACL2 es que está pensado para poder *razonar* sobre los programas que se han definido previamente. Por tanto, un uso especialmente útil de este sistema sera el de la verificación formal de algoritmos mediante el demostrador automático que incorpora. Dicho demostrador se basa en una lógica especial que permite cierto tipo de razonamientos, sobre todo basado en la recursión de funciones e inducción.

Los desarrolladores del sistema son Matt Kaufmann y J Strother Moore, de la Universidad de Texas en Austin (Estados Unidos). Y fue la evolución del sistema `Nqthm`, ya un demostrador automático creado por J Moore y Robert Boyer en los años 70 del siglo pasado.

ACL2 se ha usado con éxito en los campos de las matemáticas y la lógica pero tiene cierto valor añadido el haber sido utilizado para la verificación formal del algoritmos microprogramados en procesadores de Motorola® y AMD®, tales como la división en aritmética de coma flotante. Este tipo de verificación formal para algoritmos en hardware tiene una ventajas importantes respecto a otros sistemas de testeo más tradicionales basados en la ejecución del algoritmo a verificar sobre una batería de ejemplos más o menos extensa:

- Se establecen propiedades formales, esto es, se consiguen demostraciones para cualquier instancia de los datos de entrada. No hace falta, por tanto, una batería de ejemplos.
- La demostración la hace una máquina, lo que evita errores atribuibles a la condición humana.

Estas dos propiedades hacen que un procesador diseñado con algoritmos verificados formalmente aumente la confianza que ofrece a los usuarios finales y a los propios desarrolladores.

La sintaxis usada por el lenguaje de programación es muy parecida a la del lenguaje LISP, tradicionalmente usado desde su concepción para el desarrollo de algoritmos relacionados de alguna manera con campos de la Inteligencia Artificial. Asimismo, la sintaxis se mantiene cuando se trata de expresar propiedades en la lógica.

Por ejemplo, podemos definir la siguiente función, que devuelve el cuadrado del valor pasado como argumento:

```
1 (defun cuadrado (x)
2   (* x x))
```

La notación es tremendamente parecida a LISP (con sus ventajas e inconvenientes) así que puede ser necesario que el usuario de ACL2 haya tenido al menos alguna experiencia previa en el manejo de este lenguaje y que entienda los conceptos relacionados con las tres primitivas más importantes de LISP para el uso de *listas*:

- `CONS`: Añade un elemento en la cabeza (como primer elemento) de una lista.
- `CAR`: Devuelve la cabeza de la lista.
- `CDR`: Devuelve la cola de la lista (todos sus elementos exceptuando la cabeza).

¿Podríamos ahora establecer alguna propiedad sobre la función `cuadrado` definida anteriormente? Parece evidente que se podría decir que, independientemente del valor del parámetro `x`, mientras sea de

tipo *entero*, el resultado a devolver debe ser positivo o cero. Esta propiedad se podría expresar así en matemáticas convencionales:

$$\forall x \in \mathbb{Z} \longrightarrow x^2 \geq 0$$

En la lógica de ACL2 se podría expresar esta propiedad, lo que la hace susceptible de ser formalmente demostrada:

```
(implies (integerp x)
         (>= (cuadrado x) 0)))
```

Es más, gracias al demostrador automático incorporado en ACL2, podríamos hacer que se intentara demostrar el siguiente teorema:

```
(defthm teorema
  (implies (integerp x)
           (>= (cuadrado x) 0)))
```

Si evaluamos el evento anterior en ACL2 obtenemos la siguiente salida:

```
<< Starting proof tree logging >>
Goal'
Goal''

Q.E.D.
```

Con el famoso `Q.E.D.` (Quod Erat Demonstrandum, lo que se quería demostrar) escrito al final del intento de prueba. Evidentemente, dicha prueba ha tenido éxito, lo que, además de la propia demostración en sí, añade algo más que no se conocía al mundo lógico interno de ACL2. Es decir, a partir de este punto, ACL2 podría usar esta propiedad ya probada para otras demostraciones posteriores. De hecho gran parte del método por el cuál un usuario humano *asiste* a las demostraciones de ACL2 consiste en la formalización y demostración de los lemas previos necesarios para la demostración del teorema original.

Resumiendo, ACL2 es:

- Un lenguaje de programación.
- Una lógica que permite razonar sobre esos programas.
- Un demostrador automático basado en la lógica anterior.

2.1. ACL2 como lenguaje de programación

La primera cosa importante que hay que decir sobre ACL2 como lenguaje de programación es que se trata de un *subconjunto* de Common LISP. Es decir, todas las primitivas y funciones *built-in* también existen en Common LISP lo que posibilita compilar y ejecutar el programa en cualquier sistema que cumpla con el estándar Common LISP.

Además, dicho subconjunto es *aplicativo*, es decir, las funciones definidas con ACL2 devuelven un valor que no depende de otra cosa que de los argumentos pasados (no hay variables globales) a dicha función de forma que a valores iguales pasados como argumentos le corresponden valores devueltos iguales. Esto permite al demostrador simplificar mucho el razonamiento realizado sobre estas funciones. Asimismo se supone que, en la lógica de ACL2, todas las funciones tienen un dominio universal, es decir, puede recibir a la entrada cualquier objeto del mundo de ACL2.

Ésto último puede cambiar al ejecutar las funciones en ACL2 ya que se permiten *guardas* en la definición de las funciones, es decir, restringir el dominio de algunos o todos de los parámetros definidos en la función.

Cada tipo de datos en ACL2 se corresponde con su propia función reconocedora en ACL2:

- `acl2-numberp`: Números, que a su vez se dividen en:
  - `natp`: Números naturales como `0`, `1`, `2`,...
  - `integerp`: Números enteros como `0`, `1`, `-2`,...
  - `rationalp`: Números racionales como `2`, `1/3`, `-3/5`,...
  - `complex-rationalp`: Números complejos como `#c(1 2)`, equivalente a $1 + 2i$, siempre que la parte real y la imaginaria sean, a su vez racionales.
- `standar-char-p`: Caracteres como `\#\a`, `\#\Space`,...
- `stringp`: Cadenas de caracteres como `"Hola"` y `"Adios"`.
- `symbolp`: Símbolos como `t`, `nil`, `simbolo`,...
- `consp`: Pares punteados como `(a . b)` ó `(1 2 3)`.

Los requisitos para la correcta admisión en ACL2 de una función definida mediante el evento `(defun nombre...)` son los siguientes:

1. El nombre de la función (`nombre`) no debe estar en uso.
2. Los argumentos de la función deben ser todos diferentes.
3. No debe haber variables libres, es decir, todas las variables usadas en el cuerpo de la función deben ser, o argumentos de la función, o variables *locales* definidas con la función `let`.
4. Si se ha definido una guarda (comprobaciones a hacer antes de la *ejecución* de la función), se comprueba que, si la guarda se cumple en la ejecución actual, también se cumple en las llamadas a otras funciones del cuerpo de la función.
5. En las funciones recursivas se debe asegurar que la función *termina* para cualquier valor pasado como argumento. Para ello, ACL2 busca una *medida* (basada en los números naturales) de forma que se demuestre que dicha medida se decrementa en cada llamada recursiva a la función.

Algunas funciones y macros definidas sobre números serían:

`(zp x)`	Devuelve `t` si x = 0 o no natural.	$x = 0 \vee x \notin \mathbb{N}$
`(= x y)`	Igualdad.	$x = y$
`(< x y)`	Menor estricto.	$x < y$
`(<= x y)`	Menor o igual.	$x \leq y$
`(> x y)`	Mayor estricto.	$x > y$
`(>= x y)`	Mayor o igual.	$x \geq y$
`(+ x y)`	Suma.	$x + y$
`(- x y)`	Resta.	$x - y$
`(- x)`	Negación.	$-x$
`(* x y)`	Producto.	$x \cdot y$
`(/ x y)`	División.	x/y
`(/ x)`	Inversa.	$1/x$
`(1+ x)`	Incremento.	$x + 1$
`(1- x)`	Decremento.	$x - 1$

Figura 1: Funciones *built-in* en ACL2 para el tratamiento de números.

Las funciones más usuales para el tratamiento de listas son:

`(atom x)`	Reconocedor de objetos atómicos.
`(cons x y)`	Constructor de pares punteados.
`(car p)`	Primer componente de un par punteado.
`(cdr p)`	Segundo componente de un par punteado.
`(list x y ...)`	Lista con los elementos `(x y ...)`.
`(first l)`	Primer elemento de una lista.
`(second l)`	Segundo elemento de una lista.
`(rest l)`	Todos los elementos de una lista excepto el primero.
`(len l)`	Número de elementos de una lista.
`(nth i l)`	Elemento i-ésimo de una lista.
`(update-nth i x l)`	Actualiza el elemento i-esimo de la lista `l` al valor `x`.

Figura 2: Funciones *built-in* en ACL2 para el tratamiento de listas.

2.2. ACL2 como lógica de primer orden

En la lógica incorporada a ACL2 se pueden construir términos usando expresiones de tipo LISP del estilo `(+ 1 x)`, que en matemáticas sería $x + 1$ o `(factorial n)`, que corresponde a $n!$.

Las fórmulas en esta lógica se construyen gracias a los términos anteriores y los equivalentes en ACL2 del símbolo = y las conectivas $\wedge$, $\vee$, $\neg$, $\rightarrow$ y $\leftrightarrow$:

Por tanto, ya tenemos los términos y las conectivas (las de la lógica proposicional más la igualdad) para crear las fórmulas en esta lógica. Faltarían definir los axiomas de la lógica y las reglas de inferencia que permitan derivar nuevas fórmulas a partir de los axiomas. Las definiciones de las funciones estudiadas en la sección anterior se pueden ver, desde el punto de vista de la lógica, como axiomas en el mundo ACL2 ya que establecen que ciertas expresiones son iguales a otras.

`(equal x y)`	Igualdad.	$x = y$
`(and x y)`	Conjunción lógica.	$x \wedge y$
`(or x y)`	Disyunción lógica.	$x \vee y$
`(not x)`	Negación lógica.	$\neg x$
`(implies x y)`	Implicación lógica.	$x \rightarrow y$
`(iff x y)`	Equivalencia lógica.	$x \leftrightarrow y$

Figura 3: Conectivas lógicas y la igualdad en ACL2.

Por ejemplo, la definición de la función `(cuadrado)` se introduce en la lógica de ACL2, como un axioma que establece la siguiente igualdad:

```
1 (equal (cuadrado x) (* x x))
```

Que matemáticamente sería $cuadrado(x) = x * x$. Ni que decir tiene que la semántica es algo que el autor de la función puede o no considerar relevante, es decir, ACL2 manipula expresiones y símbolos y no le asigna al símbolo *cuadrado* ningún significado. Somos nosotros los que desde el momento de la definición consideramos que devuelve el cuadrado de un número y nos sorprendería si la función devolviera por ejemplo, el factorial de ese número. Entre otras cosas por eso se consideran axiomas en la lógica de ACL2, ya que este sistema no tiene forma de verificar que la función hace lo que su nombre supuestamente dice que hace.

Las reglas de inferencia que permiten derivar unas fórmulas de otras son las de la lógica proposional más la igualdad, junto con un principio de inducción. Por tanto, la definición de la función `(cuadrado)` tiene un doble papel:

- Actúa como función que computa y calcula el cuadrado de un número pasado como argumento.
- Es un axioma más que se introduce en la lógica de ACL2, a partir del cuál, se podrían demostrar teoremas y propiedades.

Por lo tanto, se está usando el mismo lenguaje para razonar y para programar lo que asegura que la implementación del algoritmo propuesto *cumple*, bajo cualquier circunstancia y valor de los argumentos, con las propiedades que se hayan podido demostrar gracias a la lógica de ACL2.

El axioma *de base* que trae ACL2 por defecto para la parte de lógica proposicional es el siguiente: $(\neg\phi \vee \phi)$. Y sus reglas de inferencia:

- **Expansión:** Si tenemos ϕ_2, se deriva $\phi_1 \vee \phi_2$.
- **Contracción:** Si tenemos $\phi \vee \phi$, se deriva ϕ.
- **Asociatividad:** Si tenemos $\phi_1 \vee (\phi_2 \vee \phi_3)$, se deriva $(\phi_1 \vee \phi_2) \vee \phi_3$.
- **Corte:** Si tenemos $\phi_1 \vee \phi_2$ y $\neg\phi_1 \vee \phi_3$, se deriva $\phi_2 \vee \phi_3$.

Los axiomas de la igualdad que usa ACL2 son los siguientes:

- **Reflexividad:** $x = x$.
- **Igualdad respecto a funciones:** $x_1 = y_1 \wedge x_2 = y_2 \rightarrow f(x_1, x_2) = f(y_1, y_2)$.

- **Igualdad:** $x_1 = y_1 \wedge x_2 = y_2 \rightarrow (x_1 = x_2 \rightarrow y_1 = y_2)$.

Y la regla de inferencia de instanciación, si tenemos ϕ, se deriva $\sigma(\phi)$, donde σ representa una sustitución y $\sigma(\phi)$ es el resultado de aplicar esa sustitución a la fórmula ϕ, lo cuál se traduce a que cada ocurrencia de una variable libre x en ϕ se sustituye por el término que σ le asigne a x. Esto permite considerar las variables de una fórmula lógica como *universalmente cuantificadas*.

Como ya se vió en un ejemplo anterior, la fórmula en lógica clásica:

$$\forall x \in \mathbb{Z} \longrightarrow x^2 \geq 0$$

Aparece con un cuantificador universal para la variable x. Sin embargo, este cuantificador no se expresa de forma explícita en la correspondiente fórmula en ACL2, ya que, como se ha visto, las variables están universalmente cuantificadas:

```
1 (implies (integerp x)
2          (>= (cuadrado x) 0)))
```

El resto de axiomas de ACL2 se corresponden con la definición de las funciones predefinidas o *built-in* de ACL2. No se van a listar todos estos axiomas ya que serían demasiados, pero damos algunos muy básicos y fáciles de entender. En primer lugar los que afectan a las funciones `CAR` y `CDR`:

- `(car (cons x y)) = x`.
- `(cdr (cons x y)) = y`.

Y en segundo lugar los que afectan al comportamiento de `if`:

- `x` $=$ `nil` $\rightarrow$ `(if x y z) = z`.
- `x` $\neq$ `nil` $\rightarrow$ `(if x y z) = y`.

La última regla de inferencia que veremos, aunque sin entrar en muchos detalles, es el *principio de inducción*. Como ya se sabe, la admisión de una función se lleva a cabo sólo si se consigue demostrar que la función termina para cualquier valor pasado como argumento. Para ello se busca una medida sobre dichos argumentos y se demuestra que, en cada llamada recursiva, dicha medida siempre se va *decrementando*.

La idea del principo de inducción es que dividimos la prueba de ϕ por inducción en una serie de casos. Estos casos pueden ser de dos tipos: Casos *base* o casos *inductivos* dependiendo de ciertas condiciones asociadas a cada uno de ellos. En los casos inductivos se admiten como ciertas una serie de *hipótesis de inducción* que asumen (y se debe demostrar así) que la fórmula ϕ es cierta para instancias cuya medida sea *menor* que la medida de la fórmula original.

Para ello, se deberán demostrar los casos base, y, a partir de ellos y de las hipotesis de inducción se obtendría la demostración de los casos inductivos.

2.3. ACL2 como demostrador automático

La tercera parte de ACL2 sería considerarlo como un demostrador automático. Es automático en el sentido de que, una vez que el demostrador hace un intento de prueba, no hay forma de interactuar con él hasta que no acabe dicha prueba, sea porque ha alcanzado una demostración o porque ha fallado al intentar encontrar dicha demostración.

Lo más usual cuando se hace un intento de prueba no trivial es que no se complete con éxito. Esto puede significar que la conjetura a demostrar realmente no era un teorema, pero la mayoría de los casos significa que el demostrador aún no posee la información necesaria para alcanzar la demostración deseada.

El papel del usuario de ACL2 será el de proporcionar esta información y *asistir* así al demostrador automático. Esta información se hace en base a definición de nuevas funciones y demostración de teoremas que se introducen (en forma de reglas de reescritura sobre todo) en el mundo lógico de ACL2 y que le ayudan en posteriores demostraciones. Otra forma de ayudar al demostrador es mediante *hints* o consejos a seguir en el desarrollo de la prueba.

El demostrador está organizado como sigue: existe una *bolsa* o *cesta* donde se almacenan las fórmulas pendientes de demostración. Si el número de fórmulas dentro de la bolsa es cero, el intento de demostración termina con éxito. Evidentemente, la fórmula a demostrar debe introducirse al principio. En cada turno de demostración se saca una fórmula de la cesta y se le aplican sucesivamente seis procesos diferentes. Si un proceso no puede tratar con la fórmula que hemos sacado se pasa dicha fórmula al siguiente proceso, y así sucesivamente. Si teminan los seis procesos y ninguno de ellos ha podido tratar la fórmula, el intento de demostración termina con fallo.

Si uno de los procesos puede tratar con la fórmula, genera una serie de nuevas fórmulas que, en caso demostrarse ciertas, se aseguraría la verdad de la fórmula original. En este caso, se introducen las nuevas fórmulas en la bolsa y se inicia un nuevo turno desde el principio.

Los seis procesos y una breve descripción de cada uno de ellos serían:

1. **Simplificación**. Se aplican las reglas de reescritura de forma que se obtengan expresiones teóricamente más simples a partir de lemas previamente demostrados.
2. **Eliminación de destructores**. Permite expresar ciertos términos de una conjetura de la forma en la que han sido construidos. Por ejemplo, si se cumple que `(consp l)` y en la conjetura aparecen `(car l)` y `(cdr l)` podemos hacer la siguiente sustitución que puede ayudar a ACL2 a conseguir el objetivo:
   a) `l` $\rightarrow$ `(cons l1 l2)`.
   b) `(car l)` $\rightarrow$ `l1`.
   c) `(cdr l)` $\rightarrow$ `l2`.
3. **Uso de equivalencias**. Si entre las hipótesis de una conjetura existe una igualdad del tipo `(equal x y)` se intentará sustituir las expresiones `x` por `y` en el resto de la conjetura, eliminando posteriormente esta hipótesis.
4. **Generalización**. Se intenta sustituir un término que aparece en las hipótesis y en las conclusiones por una variable de forma que generalice la fórmula a demostrar.
5. **Eliminación de irrelevancias**. Se eliminan las hipótesis que se consideren irrelevantes para el intento de demostración
6. **Inducción**. Se trata de elegir, mediante ciertas heurísticas, cuál es el esquema de inducción más adecuado para la aplicación del principio de inducción. Para ello se busca una medida y una división

por casos (base e inductivos) que parezca la adecuada para la demostración, aunque hay formas de sugerir al demostrador que use los esquemas de inducción escogidos por el usuario.

Un ejemplo práctico de cómo sería un intento de demostración completo podría ser el siguiente. Consideremos la definición de dos funciones que, aplicadas sobre una lista, eliminan una instancia o todas las instancias, respectivamente, del objeto pasado como argumento `x`:

```
(defun borra-uno (x l)
  (cond ((endp l) l)
        ((equal x (first l)) (rest l))
        (t (cons (first l) (borra-uno x (rest l))))))

(defun borra-todos (x l)
  (cond ((endp l) l)
        ((equal x (first l)) (borra-todos x (rest l)))
        (t (cons (first l) (borra-todos x (rest l))))))
```

Se puede comprobar que estas funciones, aplicadas sobre la lista de ejemplo `(x y x z x)` devuelven, respectivamente, la lista `(y x z x)` y la lista `(y z)`. Sabiendo que existe la función `len` en ACL2 que devuelve el número de elementos de una lista pasada como argumento podemos tratar de demostrar el siguiente teorema:

```
(defthm len-borra-todos-uno
  (<= (len (borra-todos x l)) (len (borra-uno x l))))
```

Dicho en otras palabras, la función `borra-todos` quita al menos, tantos elementos como los que quita `borra-uno`. Conjetura que puede parecer trivial pero que no se encuentra directamente en la definición de las funciones `borra-todos` y `borra-uno`, por lo que puede no resultar tan trivial para ACL2. En efecto, un intento de demostración de este teorema acaba en fracaso. Analizando la prueba fallida encontramos el siguiente *checkpoint*:

```
Subgoal *1/2'4'
(IMPLIES (<= (LEN (BORRA-TODOS L1 L2))
             (LEN (BORRA-UNO L1 L2)))
         (<= (LEN (BORRA-TODOS L1 L2))
             (LEN L2))).
^^^ Checkpoint Subgoal *1/2'4' ^^^
```

A partir del cuál se hace un intento de generalización que no llega a buen término. ¿Qué lema le puede hacer falta a ACL2 para que el anterior punto quede demostrado? Si pudiéramos demostrar que después de borrar todos los elementos de cierto tipo la longitud de la lista se decrementa o queda igual (si no había instancias de dicho elemento) habremos probado la conclusión de la conjetura `Subgoal *1/2'4'` y el demostrador podría seguir adelante. Para ello introducimos el siguiente evento cuya demostración sí se consigue de forma directa:

```
(defthm len-borra-todos
  (<= (len (borra-todos x l)) (len l)))
```

Si volvemos a intentar ahora demostrar la conjetura original se consigue hacer sin más problemas.

3. Formalización de conceptos del álgebra lineal en ACL2

3.1. Notación y definiciones básicas

Definición 1. ***Matriz.***

Una matriz *es una tabla de $m \times n$ elementos organizados en m filas y n columnas donde $m, n \in \mathbb{N}$, cumpliéndose que $m \geq 1, n \geq 1$.*

Las matrices se representarán mediante letras mayúsculas: A, B, C, ..., etc. y sus elementos de la forma a_{ij}, b_{ij}, c_{ij}, ..., etc. donde el primer subíndice (i) indica la fila donde está situado el elemento y el segundo subíndice (j) indica la columna donde está situado el elemento.

Supondremos que las filas se numeran partiendo desde el 0 de la primera fila y, por tanto, llegando hasta la fila $m-1$. Análogamente las columnas se numeran desde la 0 hasta la $n-1$. De esta forma, si a_{ij} es un elemento de la matriz A se deben cumplir la siguientes relaciones:

- $0 \leq i < m$
- $0 \leq j < n$

Por tanto una matriz A se puede representar así:

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

Por tanto, el primer esfuerzo que hay que hacer para poder demostrar propiedades y teoremas en ACL2 sobre matrices debe ser la elección de una estructura de datos adecuada para la representación de matrices en este lenguaje. Dicha representación debería cumplir las siguientes propiedades:

- *Simplicidad.* Una formalización simple y sencilla asegura *a posteriori* un desarrollo más eficaz y libre de errores de los algoritmos definidos sobre matrices y sus propiedades.
- *Consistencia.* Todas las matrices, independientemente de su número de elementos y propiedades de los mismos, deben estar representadas gracias a la misma estructura de datos. Esto permite acceder y actualizar los elementos de la estructura con el mismo conjunto de primitivas para todas las matrices.
- *Eficiencia temporal.* La forma de acceder y actualizar los elementos de la estructura debería estar dentro de orden constante, i.e. $\in \mathcal{O}(1)$, respecto al número de elementos de la misma. Esto permitirá que los algoritmos planteados sobre matrices sean eficientes respecto al tiempo de ejecución.
- *Eficiencia espacial.* Si la estructura de datos sólo guarda los elementos de la matriz distintos a un valor tomado por defecto se va a conseguir reservar espacio en memoria únicamente para esos elementos. Esto haría que, si la aplicación o algoritmos planteados usan las llamadas matrices *dispersas* (aquellas que tienen un número de elementos, distintos del valor por defecto, pequeño respecto al total de elementos de la matriz), la cantidad de memoria a reservar será mucho menor que si utilizamos matrices *densas* (aquellas que tienen pocos elementos iguales al valor por defecto en relación al total de los mismos).

Algunas de estas propiedades son, al menos, parcialmente excluyentes entre sí, por lo que siempre se intentará llegar a una solución de compromiso entre los objetivos propuestos. Por ejemplo, asegurar acceso constante necesita necesariamente de *arrays* o vectores en memoria, lo que contradice el hecho de reservar memoria sólo para los elementos distintos del valor por defecto.

Revisando los tipos en ACL2 se puede comprobar que sólo existe un tipo no atómico, es decir, con capacidad para almacenar un dato y, a la vez, referenciar a un posible dato siguiente en la estructura. Se trata del par punteado formado por `(cons x y)`. Si utilizamos secuencias de `cons` anidadas podremos simular una lista unidimensional de datos al más puro estilo LISP. Por ejemplo, la estructura formada por `(cons 'a (cons 'b (cons 'c nil)))` sería el equivalente a la lista formada por los elementos `a`, `b` y `c`: `(a b c)`.

Evidentemente, una lista como la mostrada en el ejemplo anterior podría modelar matrices unidimensionales, esto es, vectores, también llamados matrices fila o matrices columna. Necesitamos una estructura que aumente el número de dimensiones hasta dos. Lo más natural e intuitivo es tener una lista (llamada *primaria*) en la que cada elemento debe ser a su vez otra lista (llamadas en este caso *secundarias*), de elementos atómicos, de forma que el elemento 0 de la lista primaria es una lista secundaria que representará la primera fila de la matriz, el elemento 1 de la lista primaria será otra lista secundaria que representará la segunda fila de la matriz, y así sucesivamente.

Por ejemplo, si tenemos la matriz de 3 filas y 4 columnas:

$$A = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix}$$

debe ser representada en el mundo de ACL2 como la siguiente *"lista de listas"*:

```
((a b c d)
 (e f g h)
 (i j k l))
```

Por tanto, podemos concretar que un dato en ACL2 debe cumplir las siguientes propiedades para ser considerado una matriz:

1. Debe ser una *"lista verdadera"*, es decir, se debe construir, mediante la función `cons`, una serie de pares punteados anidados en la que se cumpla que el último par punteado debe ser del tipo `(cons x nil)`, esto es, su `cdr` debe ser `nil`.

2. El tamaño de esta lista *primaria* debe ser al menos 1, es decir, la matriz debe tener una fila al menos.

3. Cada elemento de esa lista debe ser también una lista verdadera en el mismo sentido que antes. Cada fila de la matriz vendrá representada, por tanto, por una lista.

4. El tamaño de cada lista *secundaria* debe ser al menos uno. La matriz debe tener al menos una columna.

5. El tamaño de todas las listas secundarias debe ser el mismo. Esto es, no hay filas más cortas o largas que las demás, todas las filas tienen el mismo número de elementos.

6. Aunque a priori no hay necesidad de esto, vamos a introducir una última propiedad: Cada elemento de las listas secundarias debe ser del tipo `rational` de ACL2. Esto se exige para después poder realizar operaciones aritméticas sobre los elementos de las matrices. Si quisieramos ser más genéricos aún con los tipos numéricos de ACL2 se puede escoger el tipo `acl2-number`, pero en principio esta decisión sólo afectaría a esta condición.

Por tanto, podemos definir ya nuestra primera función en ACL2. Se trata del *reconocedor de matrices* según las propiedades definidas más arriba:

```
(defun matrixp-aux (n x)
  (if (endp x)
    t
    (and (rational-listp (car x))
         (equal (len (car x)) n)
         (matrixp-aux n (cdr x)))))

(defun matrixp (x)
  (and (true-listp x)
       (<= 1 (len x))
       (<= 1 (len (car x)))
       (matrixp-aux (len (car x)) x)))
```

Vemos aquí una función (`matrixp`) que llama a otra función (`matrixp-aux`) en un esquema recursivo que se repetirá muchas más veces a lo largo de este proyecto. Expliquémoslo, por tanto, con más detalle.

En la llamada a `matrixp` se verifica que el objeto `x` pasado como argumento cumple con la propiedad de ser una lista verdadera. A su vez también comprueba que la longitud de la lista es mayor o igual que 1 y que el primer elemento de la lista primaria tiene al menos longitud 1. Con estas comprobaciones sabemos que el objeto `x` cumple con las propiedades 1 y 2 anteriormente descritas. La propiedad 4 también ha sido comprobada para, al menos, la primera lista secundaria. Su tamaño, además, se pasa como argumento a la función siguiente.

Con la llamada a `matrixp-aux` comprobamos si el resto de listas secundarias tiene el mismo tamaño `n` que la que se comprobó en la llamada a `matrixp`, además se comprueba que el resto de listas secundarias son listas verdaderas en las que cada elemento es del tipo `rational` de ACL2. Con esto tendremos verificadas las propiedades 3, 5 y 6.

Visto de otra forma, la llamada a `matrixp` hace ciertas comprobaciones un tanto generales sobre el objeto `x` y la llamada a la función recursiva `matrixp-aux` realiza otro tipo de comprobaciones para *todas las filas* de la matriz. Curiosamente, gran parte de los predicados que se implementarán sobre las matrices harán análogamente lo mismo, es decir, se llamará a una función auxiliar que procesará las filas de la matriz comprobando las propiedades que sean necesarias.

Dentro de esta definición de *Matriz* se pueden definir otras dos funciones básicas o primitivas sobre las matrices. Se trata de poder *acceder* a cada elemento de una matriz a través de su número de fila y columna y poder *modificar* cada uno de esos elementos dando además el nuevo valor que sustituirá al antiguo elemento. Así:

```
(defun lookup (matrix i j)
  (nth j (nth i matrix)))

(defun update (matrix i j v)
  (update-nth i (update-nth j v (nth i matrix)) matrix))
```

El uso de estas dos primitivas para acceder o modificar elementos individuales de cierta matriz es muy intuitivo, por ejemplo, sea:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Si quisiéramos acceder al elemento de la fila 2 y columna 1 usaríamos la siguiente llamada `(lookup A 2 1)` y obtendríamos el valor 8 (recuérdese que las filas y matrices se empiezan a numerar desde el 0). Si ejecutamos `(update A 2 1 10)` modificaríamos la matriz A de forma que pasaría a valer:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 10 & 9 \end{pmatrix}$$

Estas dos nuevas primitivas se basan en las operaciones `nth` y `update-nth` de ACL2 que, respectivamente, accede a un elemento concreto de una lista y actualiza un elemento de una lista.

Definición 2. ***Orden de una matriz.***

Una matriz se dice que tiene dimensión *o que es de* orden $m \times n$ *si tiene m filas y n columnas. Se denotará por $r(A)$ a la función que, dada un matriz, nos da su número de filas y por $c(A)$ a la función que, dada una matriz, nos da su número de columnas. Por tanto si la matriz A es de dimensión $m \times n$ se cumplirá que $r(A) = m$ y que $c(A) = n$.*

Se necesitarán, por tanto, dos primitivas adicionales en ACL2 que nos devuelvan el número de filas `(nrows A)` y columnas `(ncolumns A)` de una matriz. Según la formalización vista en la definición anterior, el número de filas será la longitud de la lista primaria y el número de columnas será la longitud de una cualquiera de las listas secundarias. Como se debe cumplir que debe haber al menos una fila podemos seleccionar la primera de ellas para calcular el número de columnas.

```
1  (defun nrows (matrix)
2    (len matrix))
3
4  (defun ncolumns (matrix)
5    (len (car matrix)))
```

También podemos aprovechar esta definición de dimensión de una matriz para presentar la quinta y última primitiva sobre las matrices. Debemos poder crear una matriz con cierto número de filas y cierto número de columnas. Por razones que se verán cuando introduzcamos los objetos de hebra simple en la formalización de las matrices en ACL2, está primitiva se llamará `(redim A m n)` y supondrá que la matriz `A` sea redimensionada de forma que su número de filas será `m` y su número de columnas será `n`.

```
1  (defun redim-rec (m n)
2    (if (zp m)
3      nil
4      (cons (make-list n :initial-element 0)
5            (redim-rec (1- m) n))))
6
7  (defun redim (matrix m n)
8    (declare (ignore matrix))
9    (redim-rec m n))
```

Puede sorprender el hecho de que la llamada a `redim` no use el parámetro `matrix` para nada. En realidad como `redim-rec` *construye* desde el principio la matriz (por la característica especialmente *aplicativa* de ACL2), no haría falta pasar este parámetro.

¿Por qué tener este argumento entonces? Cuando introduzcamos los *stobj's* más adelante veremos que existe la posibilidad de que la función modifique dicho argumento y pueda, ahora sí, redimensionar una matriz dada sin necesidad de cambiar la sintaxis de esta primitiva.

Recordemos que la función *built-in* `(make-list n)` en ACL2 crea una lista de `n` elementos a cierto valor inicial. Este valor inicial ha sido elegido como el valor 0. Por lo tanto, una llamada a `redim A 4 3` crea la siguiente matriz:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Definición 3. ***Relación de equidimensionalidad.***

Dos matrices A y B se dice que cumplen la relación de equidimensionalidad *o son* equidimensionales *si tienen el mismo número de filas y el mismo número de columnas, es decir, se debe cumplir simultáneamente:*

- $r(A) = r(B)$
- $c(A) = c(B)$

Para ello definimos la siguiente función no recursiva en ACL2:

```
(defun equidimensionalp (A B)
  (and (equal (nrows A) (nrows B))
       (equal (ncolumns A) (ncolumns B))))
```

Definición 4. ***Matriz cuadrada.***

Se dice que una matriz A es cuadrada *si tiene el mismo número de filas que de columnas, es decir, se debe cumplir que $r(A) = c(A)$.*

Para ello definimos la siguiente función no recursiva en ACL2:

```
(defun square-matrixp (A)
  (equal (nrows A) (ncolumns A)))
```

Definición 5. ***Igualdad entre matrices.***

Dos matrices A y B se dice que son iguales *si, siendo equidimensionales y de dimensión $m \times n$, se cumple que $a_{ij} = b_{ij}$ para valores de $i = 0, \ldots, m-1$ y de $j = 0, \ldots, n-1$.*

Para ello vamos a seguir una estrategia de abajo hacia arriba. Empezaremos definiendo la igualdad entre filas *concretas* de una matriz, es decir, dada una fila, hay que comprobar si dicha fila es igual en las matrices A y B `(equal-row A B m n)`. Después habrá que comprobar si dos matrices A y B tienen todas las filas iguales desde la fila 0 hasta cierta fila `(equal-rows A B m n)`. Por último, la igualdad entre matrices se definirá como la igualdad entre filas de las matrices desde la primera hasta la última `(equal-matrix A B)`. Esto se hace para seguir cierto esquema recursivo, que de forma resumida, significa que para que dos matrices sean iguales se necesita que sean iguales todas sus filas elemento a elemento.

Si empezamos por la primera función, `(equal-row A B m n)`, se comprueba que se cumple las siguientes igualdades:

$$a_{m0} = b_{m0} \ \wedge \ a_{m1} = b_{m1} \ \wedge \ \ldots \ \wedge \ a_{mn} = b_{mn}$$

Es decir, comprueba que la fila m de A y de B sean iguales elemento a elemento desde el elemento de la columna 0 hasta el elemento de la columna n.

```
(defun equal-row (A B m n)
  (if (zp n)
      (equal (lookup A m 0)
             (lookup B m 0))
      (and (equal (lookup A m n)
                  (lookup B m n))
           (equal-row A B m (1- n)))))
```

La segunda función `(equal-rows A B m n)` (adviértase el plural) comprueba que se cumple las siguientes igualdades:

$$\begin{array}{ccccccc}
a_{00} = b_{00} & \wedge & a_{01} = b_{01} & \wedge & \ldots & \wedge & a_{0n} = b_{0n} \\
a_{10} = b_{10} & \wedge & a_{11} = b_{11} & \wedge & \ldots & \wedge & a_{1n} = b_{1n} \\
\vdots & & \vdots & & \ddots & & \vdots \\
a_{m-1,0} = b_{m-1,0} & \wedge & a_{m-1,1} = b_{m-1,1} & \wedge & \ldots & \wedge & a_{m-1,n} = b_{m-1,n} \\
a_{m0} = b_{m0} & \wedge & a_{m1} = b_{m1} & \wedge & \ldots & \wedge & a_{mn} = b_{mn}
\end{array}$$

Con lo que se comprueba que las filas desde la 0 hasta la m son iguales elemento a elemento, desde la columna 0 hasta la columna n, en las dos matrices A y B.

```
(defun equal-rows (A B m n)
  (if (zp m)
    (equal-row A B 0 n)
    (and (equal-row A B m n)
         (equal-rows A B (1- m) n))))
```

Por último tendremos la función que lanza la primera llamada a `(equal-rows A B m n)` particularizando el valor de m a la última fila de las dos matrices A y B y el valor n a la última columna de dichas matrices. Respectivamente $r(A) - 1$ y $c(A) - 1$. Anteriormente hemos tenido que comprobar que A y B son matrices *verdaderas* y que, además, cumple con la relación de equidimensionalidad.

Es importante resaltar el hecho de que, si no se cumple que A sea un matriz verdadera, o que B sea una matriz verdadera o que no sean equidimensionales, se delega la ejecución del método a la función `(equal A B)` de ACL2. Con esto conseguiremos que la igualdad entre matrices *también* sirva para comprobar la igualdad entre el resto de tipos de datos de ACL2 (aquellos que no son matrices tal y como se definió en el predicado `matrixp`).

Tal y como está planteado podremos demostrar, por tanto, la siguiente propiedad:

$$\texttt{(equal-matrix A B)} \longrightarrow \texttt{(equal A B)}$$

Es decir, que si se cumple para dos objetos que son iguales en el sentido de matrices se cumple necesariamente que son iguales en el sentido de ACL2. Este hecho es importante ya que si no es así no se pueden crear reglas de reescritura en este sistema de demostración. Por ejemplo, se debe poder demostrar

más adelante que $A + B = B + A$, es decir la propiedad conmutativa de la suma de matrices. Claro que el símbolo = es la igualdad en sentido matricial, en ACL2:

```
(defthm commutativity-of-add-matrix-bad-rewrite-rule
  (equal-matrix (add-matrix A B) (add-matrix B A)))
```

Pero este teorema, por sí sólo, no puede usarse para poder *sustituir* la expresión `(add-matrix A B)` por la expresión `(add-matrix B A)` en posteriores demostraciones ya que la regla generada, dado el teorema anterior, es:

```
REEMPLAZA (equal-matrix (add-matrix A B) (add-matrix B A)) POR TRUE
```

Efectivamente, dicha regla *no* reescribe `(add-matrix A B)` por `(add-matrix B A)` pero gracias a la definición tan general de `(equal-matrix)` podremos demostrar que:

```
(defthm equal-matrix-implies-equal
  (implies (equal-matrix A B)
           (equal A B)))
```

Y, con este teorema y el anterior, podremos demostrar el siguiente resultado siguiendo razonamientos triviales:

```
(defthm commutativity-of-add-matrix-good-rewrite-rule
  (equal (add-matrix A B) (add-matrix B A)))
```

Que generará la siguiente regla de reescritura, más adecuada para usarla después en posteriores demostraciones:

```
REEMPLAZA (add-matrix A B) POR (add-matrix B A)
```

Aquí aparece ya esta importantísima función para el resto del proyecto:

```
(defun equal-matrix (A B)
  (if (and (matrixp A)
           (matrixp B)
           (equidimensionalp A B))
    (equal-rows A B (1- (nrows A)) (1- (ncolumns A)))
    (equal A B)))
```

Definición 6. ***Matriz nula.***

Se dice que una matriz A de dimensión $m \times n$ es nula *si se cumple que $a_{ij} = 0$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$. En otras palabras la matriz nula es la que tiene todos sus elementos a cero.*

Esta definición nos proporciona dos funciones en ACL2: la que genera una matriz nula de cierta dimensión y la reconocedora de una matriz nula, esto es, la función que nos dice si una matriz determinada es nula o no.

```
(defun get-zero-matrix (A m n)
  (redim A m n))
```

Definición trivial y no recursiva donde nos aprovechamos de que la función `(redim)` inicializa todos lo elementos de la matriz generada a cero. Por otro lado, para hacer la función reconocedora de las matrices nulas es necesario crear una serie de funciones auxiliares con el esquema de recursión análogo al que se utilizó al definir la función `(equal-matrix)`.

```
(defun zero-row (A m n)
  (if (zp n)
      (equal (lookup A m 0) 0)
      (and (equal (lookup A m n) 0)
           (zero-row A m (1- n)))))
```

Que comprueba si se cumple:

$$a_{m0} = 0 \ \wedge \ a_{m1} = 0 \ \wedge \ \ldots \ \wedge \ a_{mn} = 0$$

Es decir, comprueba que la fila m de A tiene todos sus elementos a cero desde el de la columna 0 hasta el de la columna n. Por otro lado, la función:

```
(defun zero-rows (A m n)
  (if (zp m)
    (zero-row A 0 n)
    (and (zero-row A m n)
         (zero-rows A (-1 m) n))))
```

Comprueba lo mismo que la anterior pero para las filas que van desde la 0 hasta la m:

$$\begin{array}{ccccccc}
a_{00} = 0 & \wedge & a_{01} = 0 & \wedge & \ldots & \wedge & a_{0n} = 0 \\
a_{10} = 0 & \wedge & a_{11} = 0 & \wedge & \ldots & \wedge & a_{1n} = 0 \\
\vdots & & \vdots & & \ddots & & \vdots \\
a_{m-1,0} = 0 & \wedge & a_{m-1,1} = 0 & \wedge & \ldots & \wedge & a_{m-1,n} = 0 \\
a_{m0} = 0 & \wedge & a_{m1} = 0 & \wedge & \ldots & \wedge & a_{mn} = 0
\end{array}$$

Por lo que faltaría la definición de la función principal:

```
(defun zero-matrixp (A)
  (and (matrixp A)
       (zero-rows A (1- (nrows A)) (1- (ncolumns A)))))
```

Definición 7. ***Diagonal principal.***

Dada una matriz cuadrada A se dice que su diagonal principal la forman los elementos de a_{ij} en los que se cumple que $i = j$.

Por ejemplo, dada la matriz de dimensión 4×4:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

Su diagonal principal la forman los elementos a_{00}, a_{11}, a_{22} y a_{33}, es decir, $1, 6, 11, 16$.

Definición 8. ***Matriz identidad.***

Se dice que una matriz A cuadrada de dimensión $n \times n$ es la matriz identidad *(y se denota por $A = I_n$) si se cumple que:*

$$a_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{1}$$

con $i = 0, \ldots, n-1$ y $j = 0, \ldots, n-1$. En otras palabras, la matriz identidad es la que tiene todos sus elementos a cero excepto los de la diagonal principal, que valen uno.

Igual que en el caso anterior tendremos que definir dos funciones en ACL2: la generadora de matrices identidad de cierta dimensión y la reconocedora de matrices identidad. Para ello definimos antes la siguiente función, que implementa la ecuación 1:

```
(defun get-element-of-identity-matrix (i j)
  (if (equal i j)
    1
    0))
```

Gracias a esta función podemos definir la generación de la matriz identidad de orden n en los tres niveles habituales, es decir, la generación de la fila m en la primera función, la generación de las filas desde la 0 hasta la m en la segunda función y la que lanza la primera llamada a esta última función:

```
(defun get-identity-matrix-row (A m n)
  (if (zp n)
    (update A m 0 (get-element-of-identity-matrix m 0))
    (seq A
         (update A m n (get-element-of-identity-matrix m n))
         (get-identity-matrix-row A m (1- n)))))

(defun get-identity-matrix-rows (A m n)
  (if (zp m)
    (get-identity-matrix-row A 0 n)
    (seq A
         (get-identity-matrix-row A m n)
         (get-identity-matrix-rows A (1- m) n))))

(defun get-identity-matrix (A n)
  (seq A
       (redim A n n)
       (get-identity-matrix-rows A (1- n) (1- n))))
```

En este caso ha aparecido una función nueva `(seq)`, que en realidad no es una *función* sino una *macro* en ACL2. Sirve para *secuenciar* "escrituras" sobre un objeto en ACL2 y está definido así:

```
(defmacro seq (obj &rest rst)
  (cond ((endp rst) obj)
        ((endp (cdr rst)) (car rst))
        (t `(let ((,obj ,(car rst)))
                 (seq ,obj ,@(cdr rst))))))
```

Por lo que, para cada línea dentro de `(seq A...` se van creado estructuras `let` anidadas que van actualizando paso a paso el valor del objeto `A`. De esta forma la función anterior `(get-identity-matrix-row)` se define internamente en ACL2 como:

```
(defun get-identity-matrix-row (A m n)
  (if (zp n)
    (update A m 0 (get-element-of-identity-matrix m 0))
    (let ((A (update A m n (get-element-of-identity-matrix m n))))
         (get-identity-matrix-row A m (1- n)))))
```

De nuevo este tipo de precauciones no es necesario en el ACL2 aplicativo y sin efectos colaterales que existe por defecto. La posterior introducción de los objetos de hebra simple hará necesario el uso de esta forma de ir actualizando el *stobj* en escrituras consecutivas al mismo.

Si queremos, no modificar un matriz para transformarla en la matriz identidad, sino comprobar si es la matriz identidad de cierto orden n, usaremos las siguientes definiciones, de nuevo en los tres niveles habituales:

```
(defun identity-matrixp-row (A m n)
  (if (zp n)
    (equal (lookup A m 0)
           (get-element-of-identity-matrix m 0))
    (and (equal (lookup A m n)
                (get-element-of-identity-matrix m n))
         (identity-matrixp-row A m (1- n)))))

(defun identity-matrixp-rows (A m n)
  (if (zp m)
    (identity-matrixp-row A 0 n)
    (and (identity-matrixp-row A m n)
         (identity-matrixp-rows A (1- m) n))))

(defun identity-matrixp (A)
  (and (square-matrixp A)
       (identity-matrixp-rows A
                              (1- (nrows A))
                              (1- (ncolumns A)))))
```

3.2. Operaciones unarias sobre matrices

En este apartado de la formalización de conceptos básicos del álgebra lineal en ACL2 vamos a proporcionar la definición de dos operaciones sobre matrices. Dichas operaciones serán unarias en el sentido

de que sólo precisarán, en sus argumentos de entrada, de un sólo objeto de tipo matriz aunque la salida sea otro objeto de tipo matriz.

Las operaciones escogidas han sido la trasposición de matrices y la negación de matrices (matriz opuesta de otra).

Definición 9. ***Traspuesta de una matriz.***

Se dice que una matriz B de orden $m \times n$ es la matriz traspuesta *de otra matriz A de orden $n \times m$ (y se denota por $B = A^T$) si es el resultado de cambiar, ordenadamente, las filas por las columnas de la matriz A de tal manera que se debe cumplir que $b_{ij} = a_{ji}$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$.*

La definición en los tres niveles habituales nos da las siguientes funciones:

```
(defun transpose-row (B A m n)
  (if (zp n)
    (update B m 0 (lookup A 0 m))
    (seq B
         (update B m n (lookup A n m))
         (transpose-row B A m (1- n)))))
```

Que modifica la fila m de la matriz B conforme a los valores de la columna m de A hasta el elemento n de la misma, en concreto:

$$b_{m0} = a_{0m}, b_{m1} = a_{1m}, \ldots, b_{mn} = a_{nm}$$

Gracias a esta definición podemos avanzar al siguiente nivel, es decir, que se completen las filas de la 0 hasta la m de la matriz B:

```
(defun transpose-rows (B A m n)
  (if (zp m)
    (transpose-row B A 0 n)
    (seq B
         (transpose-row B A m n)
         (transpose-rows B A (1- m) n))))
```

Con las siguientes operaciones:

$$\begin{array}{cccccc}
b_{00} = a_{00} & , & b_{01} = a_{10} & , & \ldots & , & b_{0n} = a_{n0} \\
b_{10} = a_{01} & , & b_{11} = a_{11} & , & \ldots & , & b_{1n} = a_{n1} \\
\vdots & & \vdots & & \ddots & & \vdots \\
b_{m-1,0} = a_{0,m-1} & , & b_{m-1,1} = a_{1,m-1} & , & \ldots & , & b_{m-1,n} = a_{n,m-1} \\
b_{m0} = a_{0m} & , & b_{m1} = a_{1m} & , & \ldots & , & b_{mn} = a_{nm}
\end{array}$$

Hay que hacer notar que estas dos funciones ya parten del hecho de que las dimensiones de ambas matrices son las correctas, esto es, que se cumple que $r(B) = c(A)$ y que $c(B) = r(A)$. De esta forma sólo nos queda definir la función `(transpose B A)`, que realizará la tarea de redimensionar la matriz B con las dimensiones apropiadas primero y después hacer la primera llamada a `transpose-rows` con los parámetros adecuados para cambiar todos los elementos de la matriz destino. Otra cuestión sintáctica que ha habido que asumir en esta función es que no hemos podido devolver la matriz A^T sobre la propia matriz

A ya que lo primero que se hace es cambiar su número de filas y columnas y, por lo tanto, se resetean todos sus elementos a cero, perdiendo la información necesaria para terminar la operación correctamente.

```
(defun transpose (B A)
  (seq B
       (redim B (ncolumns A) (nrows A))
       (transpose-rows B A (1- (ncolumns A)) (1- (nrows A)))))
```

Definición 10. ***Opuesta de una matriz.***

Se dice que una matriz B es la opuesta *de una matriz A de orden $m \times n$ (y se denota por $B = -A$) si se cumple que $b_{ij} = -a_{ij}$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$. Es decir, que tiene el mismo orden que la matriz A pero con los elementos cambiados de signo.*

Siguiendo la misma estrategia que en las definiciones anteriores podemos dar la siguiente secuencia de eventos en ACL2:

```
(defun negate-row (A m n)
  (if (zp n)
    (update A m 0 (- (lookup A m 0)))
    (seq A
         (update A m n (- (lookup A m n)))
         (negate-row A m (1- n)))))
```

Que realiza las operaciones siguientes sobre la propia matriz de entrada A:

$$a_{m0} = -a_{m0}, a_{m1} = -a_{m1}, \ldots, a_{mn} = -a_{mn}$$

Y avanzando al siguiente nivel:

```
(defun negate-rows (A m n)
  (if (zp m)
    (negate-row A 0 n)
    (seq A
         (negate-row A m n)
         (negate-rows A (1- m) n))))
```

Con las siguientes operaciones:

$$\begin{array}{cccccc} a_{00} = -a_{00} & , & a_{01} = -a_{01} & , & \ldots & , & a_{0n} = -a_{0n} \\ a_{10} = -a_{10} & , & a_{11} = -a_{11} & , & \ldots & , & a_{1n} = -a_{1n} \\ \vdots & & \vdots & & \ddots & & \vdots \\ a_{m0} = -a_{m0} & , & a_{m1} = -a_{m1} & , & \ldots & , & a_{mn} = -a_{mn} \end{array}$$

Y la operación que realiza la primera llamada a `(negate-rows)` será:

```
(defun negate (A)
  (negate-rows A (1- (nrows A)) (1- (ncolumns A))))
```

3.3. Aritmética de matrices

Las tres funciones que presentamos en este apartado se adentran en lo que es la aritmética de matrices, lo que supondrá que podamos usarlas en gran variedad de situaciones y problemas prácticos en las que participen matrices. Primero veremos una operación unaria sobre matrices que es la multiplicación de una matriz por un escalar (unaria en el sentido de que sólo necesita un argumento de tipo matriz), y posteriormente, veremos las operaciones binarias sobre matrices más importantes, la suma y la multiplicación.

Definición 11. ***Multiplicación de una matriz por un escalar.***

Sea A un matriz de orden $m \times n$ y α un escalar $\in \mathbb{Q}$, se define el producto o multiplicación por un escalar *de α por A a la matriz B del mismo orden tal que sus elementos son los de A multiplicados por α. Se denotará por $B = \alpha A$ y se deberá cumplir que $b_{ij} = \alpha a_{ij}$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$.*

Las definiciones que consiguen implementar esta operación en ACL2 seguirán el mismo patrón de recursión que las operaciones unarias del apartado anterior. En el caso concreto del producto de una matriz por un escalar tendremos que sustituir α por k (¡por ahora ACL2 no admite caracteres griegos!) pero el resto de la sintaxis es fácil de entender:

```
(defun scalar-multiply-row (k A m n)
  (if (zp n)
    (update A m 0 (* k (lookup A m 0)))
    (seq A
         (update A m n (* k (lookup A m n)))
         (scalar-multiply-row k A m (1- n)))))
```

Actualiza la fila m de A según la definición de producto de una matriz por un escalar de forma conveniente desde el elemento de la columna 0 hasta el elemento de la columna n:

$$a_{m0} = k \cdot a_{m0}, a_{m1} = k \cdot a_{m1}, \ldots, a_{mn} = k \cdot a_{mn}$$

La operación que hace la misma operación pero con las filas en el intervalo que va desde el 0 hasta el m es la función:

```
(defun scalar-multiply-rows (k A m n)
  (if (zp m)
    (scalar-multiply-row k A 0 n)
    (seq A
         (scalar-multiply-row k A m n)
         (scalar-multiply-rows k A (1- m) n))))
```

Que hace las operaciones:

$$\begin{array}{cccccc} a_{00} = k \cdot a_{00} & , & a_{01} = k \cdot a_{01} & , & \ldots & , & a_{0n} = k \cdot a_{0n} \\ a_{10} = k \cdot a_{10} & , & a_{11} = k \cdot a_{11} & , & \ldots & , & a_{1n} = k \cdot a_{1n} \\ \vdots & & \vdots & & \ddots & & \vdots \\ a_{m0} = k \cdot a_{m0} & , & a_{m1} = k \cdot a_{m1} & , & \ldots & , & a_{mn} = k \cdot a_{mn} \end{array}$$

Por último, la función `(scalar-multiply)` inicia la cadena de llamadas con los valores adecuados de m y n:

```
(defun scalar-multiply (k A)
  (scalar-multiply-rows k A (1- (nrows A)) (1- (ncolumns A))))
```

Definición 12. ***Suma de matrices.***

Sean A y B dos matrices equidimensionales de orden $m \times n$, se denomina matriz suma *de A y B, y se denota por $C = A + B$, a la matriz C, también de orden $m \times n$, tal que que se cumple $c_{ij} = a_{ij} + b_{ij}$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$.*

Usaremos la misma estrategia de implementación que en funciones anteriores definiendo:

```
(defun add-matrix-row (A B m n)
  (if (zp n)
    (update A m 0 (+ (lookup A m 0) (lookup B m 0)))
    (seq A
         (update A m n (+ (lookup A m n) (lookup B m n)))
         (add-matrix-row A B m (1- n)))))
```

Que implementa las siguientes operaciones, ya aplicada sobre la matriz A, en la fila m desde la columna 0 hasta la n:

$$a_{m0} = a_{m0} + b_{m0}, a_{m1} = a_{m1} + b_{m1}, \ldots, a_{mn} = a_{mn} + b_{mn}$$

Si queremos ejecutar la misma operación desde la fila 0 hasta la m:

```
(defun add-matrix-rows (A B m n)
  (if (zp m)
    (add-matrix-row A B 0 n)
    (seq A
         (add-matrix-row A B m n)
         (add-matrix-rows A B (1- m) n))))
```

Que implementa las siguientes operaciones, ya aplicada sobre la matriz A, en la filas desde la 0 hasta la m y, en cada una de ellas, desde la columna 0 hasta la n:

$$\begin{array}{cccccc} a_{00} = a_{00} + b_{00} & , & a_{01} = a_{01} + b_{01} & , & \ldots & , & a_{0n} = a_{0n} + b_{0n} \\ a_{10} = a_{10} + b_{10} & , & a_{11} = a_{11} + b_{11} & , & \ldots & , & a_{1n} = a_{1n} + b_{1n} \\ \vdots & & \vdots & & \ddots & & \vdots \\ a_{m0} = a_{m0} + b_{m0} & , & a_{m1} = a_{m1} + b_{m1} & , & \ldots & , & a_{mn} = a_{mn} + b_{mn} \end{array}$$

Por último, la función `(add-matrix)` inicia la cadena de llamadas con los valores adecuados de m y n:

```
(defun add-matrix (A B)
  (add-matrix-rows A B (1- (nrows A)) (1- (ncolumns A))))
```

Decir que esta función devuelve la matriz suma sobre la misma matriz A. Esto puede hacerse debido a que:

1. *La matriz no cambia de dimensiones.* Al no tener que redimensionar, no se pierde la información de la matriz original.

2. *La operación suma no es destructiva.* Es decir, mientras voy calculando la suma de un elemento cualquiera, no modifico otro elemento de la matriz que pueda servir para un cálculo posterior.

Existen otras operaciones como la siguiente (multiplicación de matrices), en que no se cumple alguna de las condiciones anteriores. Si esto es así hay que añadir un argumento adicional que permita ir rellenando la matriz de salida mientras aún uso las matrices originales. En efecto, cuando se multiplican dos matrices es posible que el orden de la matriz resultante sea distinto del de la matriz original A y, además, debo conservar hasta el final los elementos de la matriz original porque pueden servir para el cálculo de elementos de la matriz de salida. Por ejemplo, el elemento c_{00} necesita, para su cálculo, todos los elementos originales de la matriz A de la fila 0 y la columna 0 por lo que no pueden ser modificados antes. Esto lo veremos en detalle en la próxima definición.

Definición 13. ***Multiplicación de matrices.***

Sean la matriz A de orden $m \times n$ y la matriz B de orden $n \times p$ (es decir, el número de columnas de A es el mismo que el número de filas de B), se define la matriz producto o multiplicación *de A por B como la matriz C de orden $m \times p$ tal que cada elemento cumple:*

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \tag{2}$$

con $i = 0, \ldots, m-1$ y $j = 0, \ldots, p-1$.

Si expandimos el sumatorio 2 nos queda que:

$$c_{ij} = a_{i0} \cdot b_{0j} + a_{i1} \cdot b_{1j} + \cdots + a_{i,n-1} \cdot b_{n-1,j} \tag{3}$$

Que se puede definir como el *producto escalar* entre la fila i de la matriz A y la columna j de la matriz B. Como este producto escalar lo tenemos que ejecutar varias veces para el cálculo de todos los elementos de la matriz lo debemos definir aparte en una función de ACL2:

```
1 (defun dot-product (A B i k j)
2   (if (zp j)
3       (* (lookup A i 0)
4          (lookup B 0 j))
5       (+ (* (lookup A i k)
6             (lookup B k j))
7          (dot-product A B i (1- k) j))))
```

Que debe entenderse como una generalización de la expresión 3 en la que el sumatorio tiene, como límite superior, el valor k pasado como argumento. Es decir, la fila i de la matriz A sólo se tendrá en cuenta desde el elemento de la columna 0 hasta el elemento de la columna k y, análogamente, la columna j de la matriz B se tendrá en cuenta sólamente desde el elemento de la fila 0 hasta el elemento de la fila k.

$$\texttt{(dot-product A B i k j)} = a_{i0} \cdot b_{0j} + a_{i1} \cdot b_{1j} + \cdots + a_{ik} \cdot b_{kj}$$

Como se sabe de pasadas experiencias con el demostrador de ACL2, es preferible plantear las definiciones lo más generales posibles para que las demostraciones puedan aplicar el principio de inducción donde se requiera.

Ya podemos definir, por tanto, la siguiente función, que procesa la fila m de la matriz destino C desde el elemento de la columna 0 hasta el de la columna n teniendo como límite superior todos los productos escalares calculados el valor j pasado como argumento:

```
(defun multiply-matrix-row (C A B m j n)
  (if (zp n)
      (update C m 0 (dot-product A B m j 0))
      (seq C
           (update C m n (dot-product A B m j n))
           (multiply-matrix-row C A B m j (1- n)))))
```

En el siguiente nivel tenemos la función `(multiply-matrix-rows)` que procesa las filas desde la 0 hasta la m, cada una de ellas desde la columna 0 hasta la n con límite superior en los productos escalares calculados de j.

```
(defun multiply-matrix-rows (C A B m j n)
  (if (zp m)
      (multiply-matrix-row C A B 0 j n)
      (seq C
           (multiply-matrix-row C A B m j n)
           (multiply-matrix-rows C A B (1- m) j n))))
```

La función que inicia la secuencia de llamadas recursivas, redimensionando la matriz de salida C a las dimensiones adecuadas, se llamará `(multiply-matrix)` y debe inicializar los tres argumentos de tipo índice de `(multiply-matrix-rows)`:

- m: Representa la máxima fila a procesar de la matriz destino, por lo que $m = r(A) - 1$.
- j: Representa el límite superior de los productos escalares calculados, por lo que este valor debe situarse en el máximo número de columnas de A, o, alternativamente al máximo valor de las filas de B, por lo que $j = c(A) - 1 = r(B) - 1$.
- n: Representa la máxima columna a procesar de la matriz destino, por lo que $n = c(B) - 1$.

```
(defun multiply-matrix (C A B)
  (seq C
       (redim C (nrows A) (ncolumns B))
       (multiply-matrix-rows C A B (1- (nrows A))
                                   (1- (ncolumns A))
                                   (1- (ncolumns B)))))
```

3.4. Transformaciones elementales

Se denominan transformaciones elementales a ciertas transformaciones que se realizan en una matriz y que son de utilidad en la implementación de ciertos algoritmos sobre matrices tales como el cálculo de la matriz inversa de una matriz dada o la resolución de sistemas de ecuaciones lineales.

Estas transformaciones modifican, de determinadas formas, los elementos de una fila de la matriz o intercambian dos filas de ésta. Nos centraremos en este apartado del proyecto en el algoritmo de Gauss-Jordan para el cálculo de la matriz inversa y su determinante. Para la implementación de este algoritmo necesitaremos de una función auxiliar en ACL2 que asigne a una matriz todos lo elementos de otra, es decir, realizar una copia de una matriz origen a otra destino.

Definición 14. ***Copia de matrices.***

Sea la matriz A de orden $m \times n$, se denomina matriz copia *de A a otra matriz B, también de orden $m \times n$ en la que cada elemento cumple que $b_{ij} = a_{ij}$ con $i = 0, \ldots, m-1$ y $j = 0, \ldots, n-1$. Es decir, B tiene los mismos elementos que A y en la mismas posiciones.*

Para ello, y como es habitual, confiamos en el esquema recursivo en tres niveles:

```
(defun copy-row (B A m n)
  (if (zp n)
    (update B m 0 (lookup A m 0))
    (seq B
         (update B m n (lookup A m n))
         (copy-row B A m (1- n)))))
```

Que realiza la copia de la fila m desde la columna 0 hasta la n:

$$b_{m0} = a_{m0}, b_{m1} = a_{m1}, \ldots, b_{mn} = a_{mn}$$

Si queremos ejecutar la misma copia de filas desde la 0 hasta la m, y,en cada una de ellas, desde el elemento de la columna 0 hasta el de la columna n:

```
(defun copy-rows (B A m n)
  (if (zp m)
    (copy-row B A 0 n)
    (seq B
         (copy-row B A m n)
         (copy-rows B A (1- m) n))))
```

$$\begin{array}{cccccc} b_{00} = a_{00} & , & b_{01} = a_{01} & , & \ldots & , & b_{0n} = a_{0n} \\ b_{10} = a_{10} & , & b_{11} = a_{11} & , & \ldots & , & b_{1n} = a_{1n} \\ \vdots & & \vdots & & \ddots & & \vdots \\ b_{m0} = a_{m0} & , & b_{m1} = a_{m1} & , & \ldots & , & b_{mn} = a_{mn} \end{array}$$

Por último presentamos la función que realiza la primera llamada recursiva a `(copy-rows)`:

```
(defun copy-matrix (B A)
  (seq B
       (redim B (nrows A) (ncolumns A))
       (copy-rows B A (1- (nrows A)) (1- (ncolumns A)))))
```

Definición 15. ***Transformación*** F_{ij}**.**

Dada una matriz A, esta transformación intercambia los elementos de las filas i y j. Dicha transformación se denota por $F_{ij}A$.

Por ejemplo, si la matriz A es:

$$A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 5 \\ 1 & 0 & 2 & 3 \end{pmatrix}$$

Y efectúo la transformación F_{12}, nos queda:

$$F_{12}A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 1 & 0 & 2 & 3 \\ 4 & 2 & 1 & 5 \end{pmatrix}$$

En este caso las funciones definidas en ACL2 no necesitan de los tres niveles de llamadas a funciones, debido sobre todo a que estas transformaciones no modifican la *totalidad* de la matriz sino sólo, a lo sumo, dos filas de las misma. Nos vale un esquema de dos funciones como el que sigue para definir esta transformación:

```
(defun exchange-rows-aux (A i j n)
  (if (zp n)
    (exchange-elements A i 0 j 0)
    (seq A
         (exchange-elements A i n j n)
         (exchange-rows-aux A i j (1- n)))))

(defun exchange-rows (A i j)
  (exchange-rows-aux A i j (1- (ncolumns A))))
```

La primera de las funciones va intercambiando los elementos de las filas i y j desde el elemento de la columna n hasta el de la columna 0. Para ello se hace uso de una función que intercambia los elementos cuyos índice, por filas y por columnas, se pasan como argumento. Esta función, llamada `(exchange-elements A i j k l)` intercambia el valor de los elementos a_{ij} por el a_{kl}:

```
(defun exchange-elements (A i j k l)
  (let
    ((eij (lookup A i j))
     (ekl (lookup A k l)))
    (seq A
         (update A i j ekl)
         (update A k l eij))))
```

Definición 16. ***Transformación*** $F_i(\alpha)$**.**

Dada una matriz A, esta transformación multiplica todos los elementos de la fila i por α. Dicha transformación se denota por $F_i(\alpha)A$.

Por ejemplo, si la matriz A es:

$$A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 5 \\ 1 & 0 & 2 & 3 \end{pmatrix}$$

Y efectúo la transformación $F_1(3)$, nos queda:

$$F_1(3)A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 12 & 6 & 3 & 15 \\ 1 & 0 & 2 & 3 \end{pmatrix}$$

De forma análoga a la transformación anterior se tiene que:

```
(defun multiply-row-aux (A i alpha n)
  (if (zp n)
    (update A i 0 (* (lookup A i 0) alpha))
    (seq A
         (update A i n (* (lookup A i n) alpha))
         (multiply-row-aux A i alpha (1- n)))))

(defun multiply-row (A i alpha)
  (multiply-row-aux A i alpha (1- (ncolumns A))))
```

La primera de las funciones va multiplicando los elementos de la fila i por α desde el que está en la columna n hasta el de la columna 0.

Definición 17. ***Transformación*** $F_{ij}(\alpha)$**.**

Dada una matriz A, esta transformación le suma a todos los elementos de la fila i los elementos correspondientes de la fila j multiplicados por α. Dicha transformación se denota por $F_{ij}(\alpha)A$.

Por ejemplo, si la matriz A es:

$$A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 5 \\ 1 & 0 & 2 & 3 \end{pmatrix}$$

Y efectúo la transformación $F_{12}(3)$, nos queda:

$$F_{12}(3)A = \begin{pmatrix} 2 & 1 & 3 & 4 \\ 7 & 2 & 7 & 14 \\ 1 & 0 & 2 & 3 \end{pmatrix}$$

De forma análoga a la transformación anterior se tiene que:

```
(defun saxpy-row-aux (A i j alpha n)
  (if (zp n)
    (update A i 0 (+ (lookup A i 0)
                     (* (lookup A j 0) alpha)))
    (seq A
         (update A i n (+ (lookup A i n)
                          (* (lookup A j n) alpha)))
         (saxpy-row-aux A i j alpha (1- n)))))

(defun saxpy-row (A i j alpha)
  (saxpy-row-aux A i j alpha (1- (ncolumns A))))
```

Reseñar aquí que el nombre elegido es el de un algoritmo muy típico usado en *benchmarks* para la medida del rendimiento en procesadores comerciales. Hace referencia al cálculo de una combinación lineal entre dos vectores de datos, de ahí el acrónimo: `SAXPY = Simple precision Alpha X Plus Y`, es decir, $\vec{y} = \vec{y} + \alpha\vec{x}$.

3.5. Algoritmo de Gauss-Jordan

Aunque el algoritmo de Gauss-Jordan tiene aplicaciones en otros campos del álgebra (como por ejemplo la resolución de sistemas de ecuaciones lineales), lo vamos a usar para el cálculo de la matriz inversa de otra dada. Por tanto, empezaremos definiendo matriz inversa:

Definición 18. ***Inversa de una matriz.***

Dada una matriz A cuadrada de orden $n \times n$, se dice que tiene matriz inversa *si existe otra matriz cuadrada del mismo orden, denotada por A^{-1}, que cumple:*

$$A^{-1}A = AA^{-1} = I_n$$

Es decir, la inversa de una matriz es aquella matriz que, multiplicada por la matriz original nos da como resultada la matriz identidad.

3.5.1. El algoritmo y su descripción cualitativa

El algoritmo de Gauss-Jordan nos permite calcular la matriz inversa, de una matriz dada, usando sólo transformaciones elementales por filas, es decir, transformaciones del tipo F_{ij}, $F_i(\alpha)$ y $F_{ij}(\alpha)$. Para ello se parte de dos matrices, la original A, que suponemos cuadrada y de orden n y de otra matriz que llamaremos B, que inicialmente será igual a I_n.

El algoritmo aplica sucesivamente transformaciones por filas a la matriz A para tratar de convertirla en I_n. Las mismas tranformaciones que se apliquen a esta matriz A se deben aplicar también a la matriz B. Si esto se consigue tendremos en B, al finalizar el algoritmo, la matriz inversa de A, esto es, A^{-1}.

La forma de aplicar las transformaciones por filas debe intentar convertir la matriz original en la identidad. Para ello nos centramos en una fila y columna concreta de A, por ejemplo la m. Si nos fijamos en el elemento de la diagonal principal a_{mm}, al que llamaremos "pivote", hay dos posibilidades, que valga cero, o que valga distinto de cero.

Si el pivote es cero hay que buscar la primera fila que, siendo menor que m, tenga en su columna m un elemento distinto de cero. Si no se puede encontrar, el algoritmo acaba y no se ha podido encontrar la inversa de la matriz A. Si se ha podido encontrar esta fila, se intercambian la fila m por ésta y con ello se consigue que el pivote sea distinto de cero.

Una vez que se asegura que el pivote es distinto de cero podemos *anular* todos los elementos de la misma columna m gracias al valor del pivote si efectúamos operaciones de tipo $F_{im}(\alpha)$ con $0 \leq i < n,\ i \neq m$ y $\alpha = -a_{im}/a_{mm}$. Por esta razón, a esta parte del algoritmo de Gauss-Jordan se le denomina eliminación de columnas o reducción de columnas.

Finalizadas las operaciones anteriores debemos tener conseguido en la fila m todos los elementos iguales a cero excepto el correspondiente a la diagonal principal, el pivote. Para que se aproxime aún más a la matriz identidad habrá que ejecutar la transformación $F_m(\alpha)$ con $\alpha = 1/a_{mm}$ con lo que conseguiremos que la columna m haya quedado igual que la correspondiente de la matriz I_n.

Si se consigue realizar la reducción de columnas desde la primera a la última de la matriz A y se han efectuado las mismas transformaciones sobre la matriz B, la matriz original se habrá transformado en I_n, y la matriz B tendrá la martiz inversa de A.

Por cada columna concreta de la matriz, por tanto, existen tres *turnos* o partes para conseguir su reducción, una por cada tipo de transformación de esta manera:

- $turno = 2$: Transformación F_{ij}.
- $turno = 1$: Transformación $F_i(\alpha)$.
- $turno = 0$: Transformación $F_{ij}(\alpha)$.

Para el turno 2 va a hacer falta buscar la primera fila, *menor o igual* que la m, que tenga un elemento en la misma columna m distinto de cero. Llamemos a la función que realiza esta búsqueda $index(m)$. Dicha función devuelve -1 si no se encuentra el índice de la fila deseada. Esta será la primera de las condiciones de parada del algoritmo, si no se puede intercambiar las fila actual para que elemento pivote sea distinto de cero, el algoritmo termina podemos decir, *con fallo*, ya que no se ha podido encontrar la inversa de la matriz original.

Por otro lado, tengamos en cuenta que la variable i será la que almacene el índice de la fila cuyo elemento queremos poner a cero gracias a las transformaciones $F_{ij}(\alpha)$. Por tanto, habrá que implementar un *bucle* que procese todas las filas desde la primera hasta última de la matriz para ir eliminando y poniendo todos los elementos de la columna por la que discurra el algoritmo a 0. Esta eliminación deberá tener en cuenta que no debe hacerla cuando el numero de la fila a eliminar coincida con la columna actual del algoritmo.

El algoritmo empieza con el valor de m en el máximo posible, es decir $r(A)-1$ y va descendiendo hasta que se hace negativo. Como m representa la fila/columna que estamos reduciendo, se puede asegurar que hemos terminado cuando este valor se ha hecho negativo. Esta es la segunda circunstancia que hay que comprobar para ver que el algoritmo ha terminado, esta vez *con éxito*.

La descripción mediante diagrama de flujo de este algoritmo de reducción de columnas se encuentra en la página siguiente.

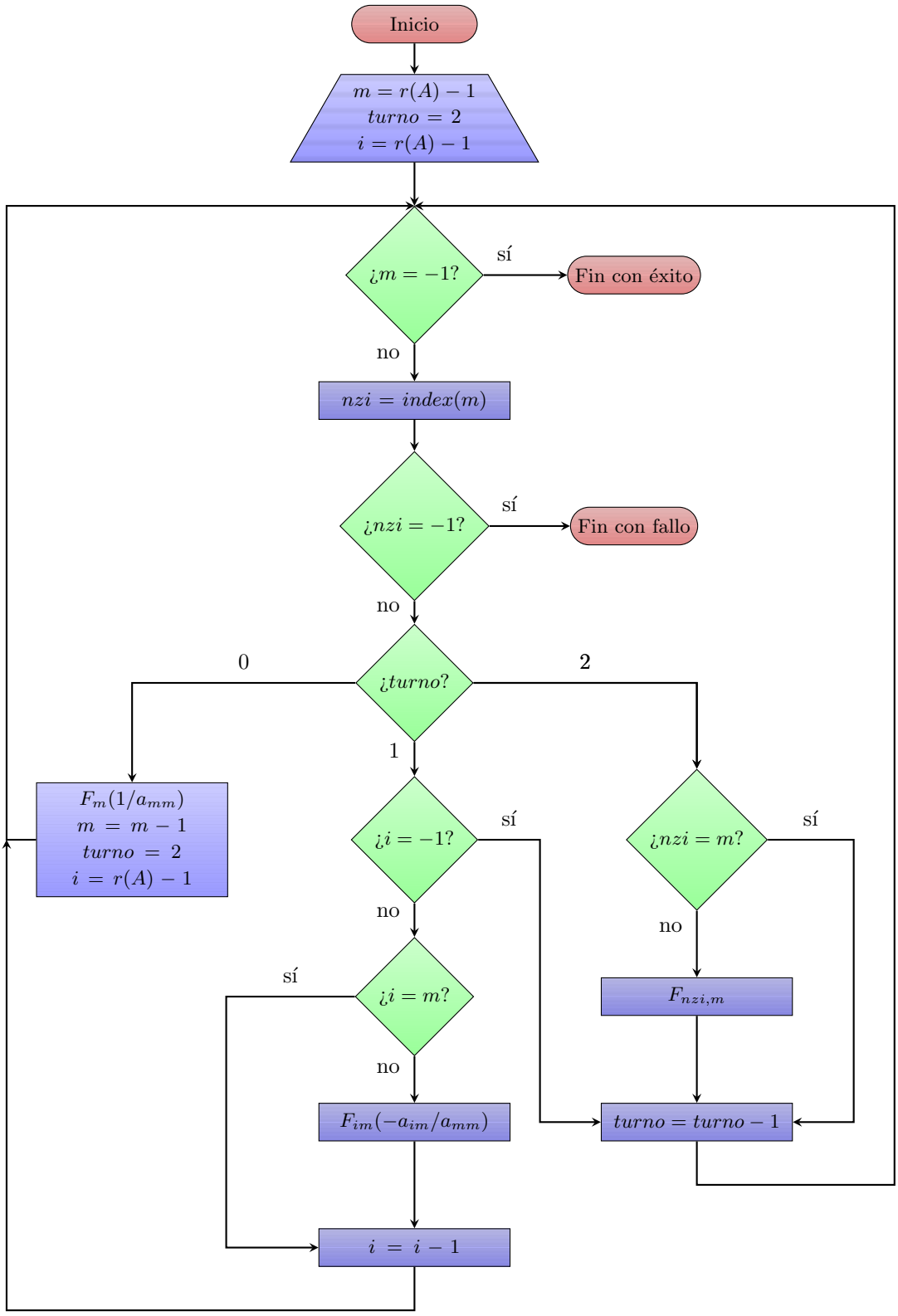


Figura 4: Diagrama de flujo del algoritmo de Gauss-Jordan

3.5.2. Detalles de la implementación: Introducción a los *stobj's* en ACL2

Empecemos con la definición en ACL2 de la función $index(m)$. Devuelve la posición de la fila del primer elemento distinto de cero de la columna c, y fila menor o igual que i. Si no encuentra dicho índice devuelve `nil`.

```
(defun get-index-of-non-zero (A i c)
  (if (zp i)
    (if (equal (lookup A 0 c) 0)
        nil
        0)
    (if (equal (lookup A i c) 0)
        (get-index-of-non-zero A (1- i) c)
        i)))
```

Una vez solventado lo anterior necesitamos alguna manera de conseguir modificar *dos matrices a la vez*. El algoritmo de Gauss-Jordan consiste en ir aplicando transformaciones elementales por fila a las dos matrices A y B, con A inicialmente la matriz a calcular su inversa (suponemos que es cuadrada y de orden n) y B inicialmente con la matriz identidad de orden n. Evidentemente, no existen funciones en ACL2 que puedan devolver dos matrices o modificar dos objetos a la vez (no existen efectos colaterales), por lo tanto, podríamos pensar en lo siguiente: concatenar las dos matrices I_n y A *adosando* ésta a la última columna de I_n. Por, ejemplo:

$$A = \begin{pmatrix} 2 & 1 & 3 \\ 7 & 2 & 7 \\ 1 & 0 & 2 \end{pmatrix} \longrightarrow I_n|A = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 2 & 1 & 3 \\ 0 & 1 & 0 & 7 & 2 & 7 \\ 0 & 0 & 1 & 1 & 0 & 2 \end{array}\right)$$

La matriz resultante $(I_n|A)$ será por tanto de orden $n \times 2n$. Para ello habría que definir la función `concatena-por-columnas` y demostrar los correspondientes lemas y teoremas sobre ella para poder usar esta definición en la lógica de ACL2.

Aún así, el uso de esta función se quedaría corto muy pronto si quisiéramos modificar un tercer objeto a la vez en el algoritmo. Efectivamente, podemos aprovechar las transformaciones por filas del algoritmo de Gauss-Jordan para el cálculo del determinante de la matriz A.

Cálculo del determinante en el algoritmo de Gauss-Jordan.

El determinante de una matriz nos da un criterio para saber si una matriz es o no es invertible (cuando dicho determinante es cero la matriz es no invertible). A cada matriz $A \in \mathbb{Q}^{n\times n}$ cuadrada se le asocia un número racional que llamaremos *determinante de* A y representaremos por $\det A$ o $|A|$.

Los determinantes poseen multitud de propiedades y existen en la literatura matemática muchas formas de calcularlo, con diferentes órdenes de complejidad, incluso en situaciones muy particulares de algunos tipos de matrices (recuérdese la *regla de Sarrus* para el cálculo del determinante en matrices cuadradas de orden 3). Nosotros nos centraremos aquí en el cálculo del determinante usando las mismas transformaciones elementales utilizadas en el algoritmo de Gauss-Jordan.

Recordemos que usamos en dicho algoritmo tres tipos de transformaciones por filas. Si tenemos una variable más en el algoritmo, llamada `determinante`, inicializada a uno, deberemos ir modificando esta variable de acuerdo a las siguientes reglas:

- Si se aplica una transformación del tipo F_{ij}: $\det \leftarrow -\det$.

- Si se aplica una transformación del tipo $F_i(\alpha)$: det $\leftarrow \alpha \cdot$ det.
- Si se aplica una transformación del tipo $F_{ij}(\alpha)$: det $\leftarrow$ det. Es decir, no cambia el determinante.

En caso de terminar el algoritmo de Gauss-Jordan con fallo (es decir, por no haber podido llegar a la matriz identidad partiendo de A) deberemos poner el determinante a cero (det $\leftarrow 0$) antes de terminar para comunicar al usuario del algoritmo que no se ha podido encontrar inversa para la matriz original.

Podemos, por tanto, modificar el algoritmo de Gauss-Jordan anteriormente propuesto para que la variable `determinante` sea modificada a la vez que las matrices A y B. Esta modificación aparece en la página siguiente.

Por tanto, a lo largo de la ejecución del algoritmo debemos ir modificando hasta *tres objetos* a la vez, la matriz original A, la matriz B y el valor del determinante. Debemos por tanto, descartar el uso y la definición de la función `concatena-por-columnas` ya que sólo nos soluciona en parte el hecho de querer modificar dos matrices (piénsese además que pasaría si queremos modificar tres o cuatro matrices a la vez). Se hace necesario entonces otra forma de plantear el algoritmo. Será a través del uso de los *objetos de hebra simple* en ACL2.

Introducción a los *stobj's* en ACL2.

En ACL2, un *stobj (Single Threaded OBJect)*, u objeto de hebra única en castellano, es una estructura de datos compuesta por otros tipos de datos cuyo uso está sintácticamente restringido para asegurar que sólo existe una instancia del mismo en la memoria. Esto permite actualizar sus campos de forma destructiva, es decir, si una función modifica el valor de alguno de los campos de la estructura, esta función debe devolver el mismo objeto que ha modificado y guardar lo que devuelve en el propio objeto modificado (estas son algunas de las restricciones sintácticas).

Estas restricciones sólo se dan cuando se intenta usar el *stobj* en el bucle de alto nivel de ACL2 o en la definición de funciones, es decir, en la parte *ejecutable* de ACL2. Desde el punto de vista *lógico* un *stobj* se trata como un objeto compuesto por `cons` y números. De forma que cuando se modifica algún campo de los *stobj's*, desde el mundo lógico, se crea un nuevo objeto.

Las consecuencias de que sólo pueda existir una referencia al objeto en memoria son las siguientes (supongamos que `OBJ` ha sido declarado como *stobj*):

- `OBJ` debe ser tratado como una variable global en ACL2 que contiene dicho objeto.
- Si una función usa como argumento `OBJ`, la única expresión que puede ser pasada en ese argumento es `OBJ`, no una función que meramente evalúa y devuelve un objeto del mismo tipo. La forma de poder conseguir esto sería a través del uso de `let's` anidados que vayan actualizando paso a paso el *stobj* que se quiere usar.
- Las funciones *built-in* de ACL2 tales como `CAR`, `CONS`, `EQUAL`, etc. no pueden ser aplicadas sobre *stobj's*.
- Cuando una función usa un *stobj*, sólo se puede pasar un objeto de este tipo en el argumento correspondiente. Esto ocurre en las funciones observadoras y modificadoras generadas automáticamente al declarar un *stobj*. La única función que puede ser aplicada a todos los tipos de objetos es la función *reconocedora*, también generada automáticamente en la declaración de un objeto de este tipo.
- Si una función llama a una función que actualiza un *stobj*, debe devolver un objeto del mismo tipo.
- Cuando una función devuelve un *stobj*, almacena el valor devuelto en la única referencia que existe a ese objeto.

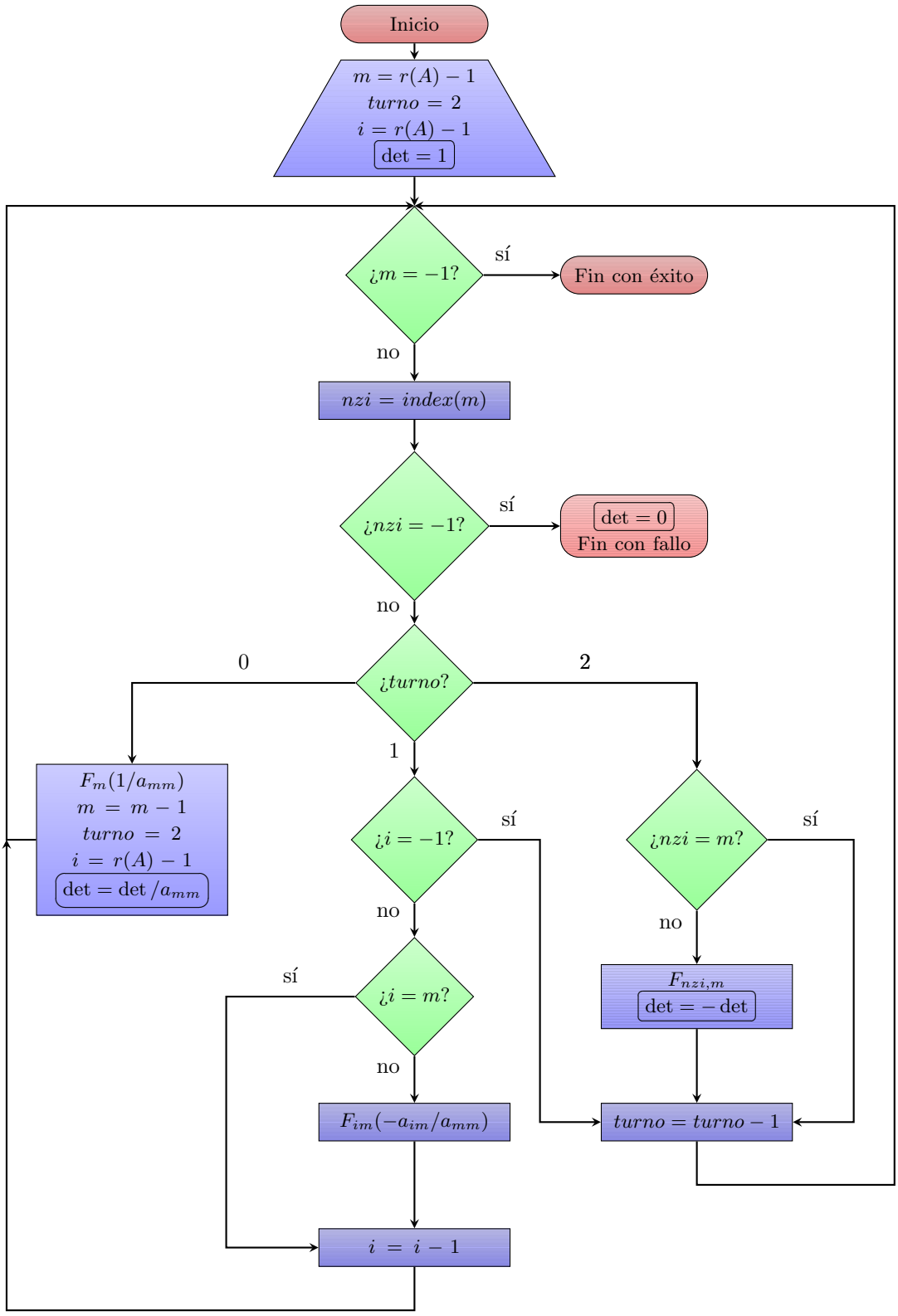


Figura 5: Diagrama de flujo del algoritmo de Gauss-Jordan para calcular el determinante.

Debido a estas restricciones sintácticas, una función *sí* que puede modificar el valor de varios campos de un stobj mientras esto se haga de forma *secuenciada* (con el uso de `let's`). Esto hace ideal el uso de los *stobj's* en el agoritmo de Gauss-Jordan propuesto ya que queremos modificar tres campos simultáneamente en cada paso del algoritmo.

Otra cuestión interesante de este tipo de estructuras de datos es la cantidad de tipos diferentes en los que se pueden declarar los campos de un *stobj*. A los habituales `integer` o `rational` se añaden los de tipo `array`. Es decir, se puede declarar un campo de un *stobj* de tipo vector de enteros, o vector de racionales, etcétera. Por ahora, y esto será importante más adelante, ACL2 sólo da soporte para vectores *unidimensionales* en los campos de un *stobj*.

Veamos un ejemplo de definición de un *stobj* muy básico y los métodos accesores y modificadores para cada campo introducido:

```
(defstobj ejemplo
  (x  :type rational              :initially 0)
  (y  :type (integer 5 10)        :initially 7)
  (a  :type (array (integer) (3)) :initially 1 :resizable t))
```

Con este evento se ha declarado un *stobj* llamado `ejemplo` compuesto por tres campos:

1. `x`: de tipo racional y valor inicial 0 (campos `rational` y `:initially`).
2. `y`: de tipo entero en un rango determinado desde el valor mínimo a 5 hasta el máximo valor a 10 `(integer 5 10)`. Inicialmente vale 7.
3. `a`: de tipo vector de enteros de tamaño tres `(array (integer) (3))`. Todos los elementos del vector se inicializan a 1. Este vector se podrá redimensionar gracias al campo `:resizable t`.

El efecto del anterior evento es introducir un nuevo objeto de hebra simple llamado `ejemplo` y las funciones observadoras, modificadoras y reconocedoras asociadas. Este nuevo objeto de hebra simple se trata como una variable global, que, *desde el punto de vista lógico*, es una lista de k elementos donde k es el número de campos proporcionados en la definición (tres en nuestro ejemplo). Los elementos están en dicha lista en el mismo orden en el que están declarados. Los campos de tipo vector serán tratados desde el punto de vista lógico también como listas de elementos de tipo atómico. Por lo tanto, la representación del anterior *stobj* será así justo después de la creación del objeto:

```
(0 7 (1 1 1))
```

Hay que hacer notar que la representación real del *stobj* en `LISP` que actúa por debajo puede diferir en gran medida del punto de vista lógico que hemos dado arriba. Por el momento sólo nos centramos en el mundo lógico aunque el mundo de la ejecución de las funciones asociadas a un *stobj* será de importancia capital en el desarrollo de este proyecto. Dejamos esta parte para la sección 5.

Por cada campo de tipo básico (por ejemplo `x` en nuestro ejemplo) se crean tres funciones con los siguientes nombres por defecto:

- `(xp objeto)`: **Reconocedor** del tipo `x`. Es decir, se comprueba que el objeto pasado como argumento es un racional.
- `(x ejemplo)`: **Observador** del campo `x` sobre el objeto `ejemplo`. Devuelve el valor de este campo.

- `(update-x valor ejemplo)`: **Modificador** del campo `x` sobre el objeto `ejemplo`. Modifica el campo `x` para que valga `valor`.

Estos nombres se pueden cambiar por otros pero realmente no hay necesidad de ello en el problema que nos ocupa, véase la documentación de los `stobj's` en ACL2 para más información de este punto.

Por cada campo de tipo vector (por ejemplo `a` en nuestro ejemplo) se crean las siguientes funciones:

- `(ap objeto)`: **Reconocedor** del tipo `a`. Es decir, se comprueba que el objeto pasado como argumento es un vector de enteros.
- `(ai i ejemplo)`: **Observador** del elemento i del campo `x` del objeto `ejemplo`. Devuelve el valor del elemento i-ésimo del vector `a`.
- `(update-ai i valor ejemplo)`: **Modificador** del elemento i del campo `x` del objeto `ejemplo`. Modifica el elemento i-ésimo del vector `a` para que valga `valor`.
- `(a-length ejemplo)`: Nos devuelve la longitud del vector `a` del objeto `ejemplo`.
- `(resize-a k ejemplo)`: Redimensiona el vector `a` del objeto `ejemplo` para que su longitud pase a valer k.

Finalmente también se crean automáticamente dos funciones que afectan al *stobj* al completo. Se tratan de:

- `(ejemplop objeto)`: **Reconocedor** del tipo `ejemplo`. Es decir, se comprueba que el objeto pasado como argumento es un *stobj* de tipo ejemplo, aunque claro, sólo puede haber una instancia de este tipo en todo el sistema. Veremos de todas formas más adelante la *congruencia entre stobj's* que permiten relajar esta restricción, hasta cierto punto al menos.
- `(create-ejemplo)`: Creador del objeto de tipo `ejemplo`.

Visto esto, ya podemos definir nuestra primera función sobre este *stobj* de muestra. Además una que sea capaz de modificar varios campos a la vez:

```
1 (defun modify-x-y-a (ejemplo)
2   (declare (xargs :stobjs (ejemplo)))
3   (let ((ejemplo (update-x 0 ejemplo)))
4     (let ((ejemplo (update-y 5 ejemplo)))
5       (resize-a 10 ejemplo))))
```

Nótese el uso del comando `let` para conseguir ir secuenciando las actualizaciones que se producen sobre el *stobj* `ejemplo`. Para mayor claridad en el uso de estas actualizaciones secuencializadas, se usará la macro `seq` explicada en la defición de la función que genera la matriz identidad de orden $m \times n$ `get-identity-matrix`:

```
1 (defun modify-x-y-a (ejemplo)
2   (declare (xargs :stobjs (ejemplo)))
3   (seq ejemplo
4        (update-x 0 ejemplo)
5        (update-y 5 ejemplo)
6        (resize-a 10 ejemplo)))
```

La palabra reservada `:stobjs` en la parte declarativa de la función anterior es obligatoria en ACL2 para todas aquellas funciones que tengan algún parámetro formal de tipo *stobj*.

Definición del *stobj* `concat` para el algoritmo de Gauss-Jordan.

Para conseguir modificar tres objetos a la vez se propone la declaración y uso de un *stobj* que tenga los siguientes campos:

- `left`: De tipo matriz, representará la matriz B que inicialmente es I_n y al final deberá contener, si el algoritmo ha tenido éxito, la inversa de la matriz A.
- `right`: De tipo matriz, representará la matriz A que inicialmente será la matriz original, y al final, si el algoritmo ha tenido éxito, deberá ser la matriz identidad I_n al final del mismo.
- `determinant`: De tipo racional, representa el determinante de la matriz A. Inicialmente valdrá 1 y, si el algoritmo no ha tenido éxito, deberá valer cero al final del mismo.

El ejemplo de declaración de *stobj* visto anteriormente usa los tipos por defecto que vienen incluidos en ACL2 por lo que no tendríamos problemas al representar como tipo `rational` el campo `determinant`. Sin embargo, el tipo "matriz" no existe por defecto en ACL2, de hecho, es un tipo definido por nosotros en base a si el objeto en cuestión satisface el predicado dado por la función `matrixp`. Pues bien, existe la posibilidad de declarar un campo en los *stobj's* cuyo tipo sea `(satisfies pred)`, es decir, que satisfga cierto predicado. Por lo tanto, podemo introducir el *stobj* `concat` (de concatenación de matrices) como:

```
(defstobj concat
  (left :type (satisfies matrixp) :initially ((0)))
  (right :type (satisfies matrixp) :initially ((0)))
  (determinant :type rational :initially 1))
```

La inicialización de los dos campos de tipo *matriz* se hace a la matriz más simple posible, la que tiene una fila y una columna, y el único elemento que tiene es cero: `((0))`. Por tanto, después de la definición anterior, hay que suponer declarados a partir de este punto los métodos accesores, modificadores, y reconocedores listados a continuación:

```
   (concatp object)
   (create-concat)
   (leftp object)
   (rightp object)
   (determinantp object)
   (left concat)
   (right concat)
   (determinant concat)
   (update-left A concat)
   (update-right B concat)
   (update-determinant d concat)
```

Definición de funciones que usan el *stobj* `concat`.

Con la creación inicial de `concat` no tenemos más que dos matrices de orden 1 concatenadas. Debe haber, por tanto, alguna forma de inicializar el *stobj* de forma que contenga las dos matrices izquierda y derecha al valor inicial que queramos. Para ello disponemos de la siguiente función, que asigna a la matriz izquierda de `concat` la matriz C y la matriz derecha de `concat` la matrix D.

```
(defun initialize-concat (C D concat)
  (declare (xargs :stobjs concat))
  (seq concat
       (update-left C concat)
       (update-right D concat)
       (update-determinant 1 concat)))
```

Después tenemos dos funciones que acceden a cada parte (izquierda o derecha) de `concat` y devuelve *copias* de las dos matrices:

```
(defun get-left (C concat)
  (declare (xargs :stobjs concat))
  (copy-matrix C (left concat)))

(defun get-right (C concat)
  (declare (xargs :stobjs (concat)))
  (copy-matrix C (right concat)))
```

Después tenemos una serie de funciones que acceden al número de filas y columnas, respectivamente, de la parte izquierda y derecha de `concat`:

```
(defun nrows-get-left (concat)
  (declare (xargs :stobjs (concat)))
  (nrows (left concat)))

(defun ncolumns-get-left (concat)
  (declare (xargs :stobjs (concat)))
  (ncolumns (left concat)))

(defun nrows-get-right (concat)
  (declare (xargs :stobjs (concat)))
  (nrows (right concat)))

(defun ncolumns-get-right (concat)
  (declare (xargs :stobjs (concat)))
  (ncolumns (right concat)))
```

Y, por último, las funciones que implementan las transformaciones por filas vistas anteriormente pero que se aplican a objetos de tipo `concat`. Estas funciones deben realizar las transformaciones sobre las dos matrices (izquierda y derecha) y, además, actualizar el determinante de forma adecuada. Primero veamos la transformación F_{ij}, después la transformación $F_i(\alpha)$ y, por último, la $F_{ij}(\alpha)$, donde se ve el uso intensivo de la macro `seq` para actualizar varios campos a la vez del objeto `concat`:

```
(defun exchange-rows-concat (concat i j)
  (declare (xargs :stobjs (concat)))
  (seq concat
       (update-left
         (exchange-rows (left concat) i j) concat)
       (update-right
         (exchange-rows (right concat) i j) concat)
       (update-determinant
         (- (determinant concat)) concat)))
```

```
10
11 (defun multiply-row-concat (concat i alpha)
12   (declare (xargs :stobjs (concat)))
13   (seq concat
14        (update-left
15          (multiply-row (left concat) i alpha) concat)
16        (update-right
17          (multiply-row (right concat) i alpha) concat)
18        (update-determinant
19          (* alpha (determinant concat)) concat)))
20
21 (defun saxpy-row-concat (concat i j alpha)
22   (declare (xargs :stobjs (concat)))
23   (seq concat
24        (update-left
25          (saxpy-row (left concat) i j alpha) concat)
26        (update-right
27          (saxpy-row (right concat) i j alpha) concat)))
```

3.5.3. El algoritmo en ACL2

A partir de aquí ya tenemos todo preparado para la definición del algoritmo de Gauss-Jordan. Lo que hacemos en primer lugar es inicializar el *stobj* `concat` para que, en su matriz izquierda, tenga la matriz identidad del mismo orden que la matriz original A pasada como argumento a la función. Para ello realizamos la llamada a `(get-identity-matrix)` con el número de filas de la matriz A: `(nrows A)`. Como matriz derecha dejamos la propia matriz A, de forma que la llamada a `(initialize-concat)` queda definida.

```
1 (defun gauss-jordan (concat A)
2   (declare (xargs :stobjs (concat)))
3   (seq concat
4        (initialize-concat
5            (get-identity-matrix (create-matrix) (nrows A))
6            A
7            concat)
8        (reduce-columns concat (1- (nrows A)) 2 (1- (nrows A)))))
```

A continuación se realiza la llamada a la función `(reduce-columns)` que implementa las acciones, bucles y condiciones descritas en el diagrama de flujo anterior.

Aparte de lo que es la propia complejidad del algoritmo, con tres rutas diferentes dependiendo del valor de la variable turno, hay que fijarse en que, en cada llamada, se asegura que sólo se hace una llamada a funciones que modifican el objeto `concat`. En principio sería posible hacer más de una llamada a modificadores de concat en cada llamada (de hecho el algoritmo resultante es bastante más corto y por tanto, más legible). El problema aparece más tarde cuando intentamos demostrar propiedades con el motor de ACL2. Hay muchas más dificultades. Planteando la función como se ha hecho en el proyecto se consigue que el demostrador la "entienda" mejor, lo que viene a demostrar que una función cuya semántica puede parecer clara para un humano no tiene por qué ser tan evidente para el demostrador y viceversa.

El trabajo del programador, en este caso, es definir las funciones no sólo de forma correcta, que se sobreentiende, sino hacerlo de forma que se minimicen posteriormente los problemas a la hora de

demostrar teoremas y propiedades con ACL2.

Otro detalle a tener en cuenta en la definición de `(reduce-columns)` es la *medida* usada para asegurar que la función termina. Recuérdese que para que ACL2 admita la definición de una función es necesario que se asegure que no se van a introducir inconsistencias en la lógica debido a esa definición. En particular, sólo se admitirá una función si previamente se demuestra que termina para cualquier dato de entrada.

Para ello se busca una medida en los naturales y, en función de los argumentos de entrada, se comprueba que decrezca siempre en cada llamada recursiva del algoritmo. Normalmente, en la gran mayoría de las funciones definidas, el demostrador es capaz de encontrar una medida basada en alguno de los argumentos de entrada que verifica lo anterior. De hecho, en todas las funciones definidas hasta ahora ha sido así. Pero hay ejemplos en los que hay que saber conjugar de forma especial los argumentos de la función. Por ejemplo, si la función de Ackermann la definimos así:

```
1 (defun ack (x y)
2   (if (zp x)
3     (1+ y)
4     (if (zp y)
5       (ack (1- x) 1)
6       (ack (1- x) (ack x (1- y))))))
```

Se puede comprobar que no existe una medida en $\mathbb{N}$ que sirva para demostrar que la función acaba (aunque realmente acaba). De forma que hay generalizar el orden entre los naturales y pasar a ordinales de órdenes superiores. Así, se define la medida en la función de Ackermann como el *orden lexicográfico* entre los dos argumentos de entrada:

$$(x_1, y_1) <_{lex} (x_2, y_2) \Longleftrightarrow (x_1 < x_2) \vee (x_1 = x_2 \wedge y_1 < y_2)$$

Este ordinal sería $\omega \cdot (x + 1) + y$ que en ACL2 se representa por los pares punteados `((1 . x + 1) . y)`, es decir, `(cons (cons 1 (1+ x)) y)`. En el caso de nuestra función `(reduce-columns)` tenemos *tres* parámetros cuyo orden lexicográfico habrá que considerar como medida de terminación de la función (llamemos t al segundo argumento para simplificar):

$$\begin{aligned}(m_1, t_1, i_1) <_{lex} (m_2, t_2, i_2) \Longleftrightarrow \quad & (m_1 < m_2) \vee \\ & (m_1 = m_2 \ \wedge \ t_1 < t_2) \vee \\ & (m_1 = m_2 \ \wedge \ t_1 = t_2 \ \wedge \ i_1 < i_2)\end{aligned}$$

Este ordinal sería $\omega^2 \cdot (m + 2) + \omega \cdot (t + 1) + i$ que en ACL2 se representa por los pares punteados `((2 . m + 2) (1 . t + 1) . i)`, es decir, `(cons (cons 2 (+ 2 m)) (cons (cons 1 (+ 1 t)) i))`. Esta función se da en la página siguiente.

```
(defun reduce-columns (concat m turn i)
  (declare (xargs :measure
                  (cons (cons 2 (+ 2 (acl2-count m)))
                        (cons (cons 1 (+ 1 (acl2-count turn)))
                              (acl2-count i)))
                  :stobjs concat))
  (let ((pivot (lookup (right concat) m m))
        (nzi (get-index-of-non-zero (right concat) m m))
        (norm (lookup (right concat) i m)))
    (if nzi
      (if (zp m)
        (cond ((zp turn)
               (multiply-row-concat concat 0 (/ pivot)))
              ((= (mod turn 3) 1)
               (if (zp i)
                 (reduce-columns concat 0 (1- turn) 0)
                 (seq concat
                      (saxpy-row-concat concat i 0
                                        (- (/ norm pivot)))
                      (reduce-columns concat 0 turn (1- i)))))
              (t (reduce-columns concat 0 (1- turn) i)))
        (cond ((zp turn)
               (seq concat
                    (if (equal nzi m)
                      (multiply-row-concat concat m (/ pivot))
                      concat)
                    (reduce-columns concat
                                    (1- m)
                                    2
                                    (1- (nrows (right concat))))))
              ((= (mod turn 3) 1)
               (if (zp i)
                 (seq concat
                      (if (equal nzi m)
                        (saxpy-row-concat concat 0 m
                                          (- (/ norm pivot)))
                        concat)
                      (reduce-columns concat m (1- turn) i))
                 (if (equal i m)
                   (reduce-columns concat m turn (1- i))
                   (seq concat
                        (if (equal nzi m)
                          (saxpy-row-concat concat i m
                                            (- (/ norm pivot)))
                          concat)
                        (reduce-columns concat m turn
                                        (1- i))))))
              (t (seq concat
                      (if (not (equal nzi m))
                        (exchange-rows-concat concat nzi m)
                        concat)
                      (reduce-columns concat m (1- turn) i)))))
      (update-determinant 0 concat))))
```

4. Principales propiedades y teoremas demostrados

En esta sección citaremos las principales propiedades que se han podido demostrar sobre el álgebra lineal dadas las definiciones de funciones de la sección anterior.

Hagamos hincapié ahora en la forma general en la que se ha llevado a cabo la definición de las funciones. Hemos definido e implementado una serie de primitivas muy básicas sobre las matrices de forma que toda función posterior *sólo* debe hacer uso de estas primitivas. Esto simplificará en gran medida las demostraciones posteriores. Recuérdese que estas primitivas, junto con su descripción simplificada, son:

- `(matrixp A)`: Reconocedor de objetos de tipo matriz.
- `(nrows A)`: Devuelve el número de filas de la matriz `A`.
- `(ncolumns A)`: Devuelve el número de columnas de la matriz `A`.
- `(lookup A i j)`: Devuelve el elemento a_{ij} de la matriz `A`.
- `(update A i j v)`: Actualiza el elemento a_{ij} de la matriz `A` y lo cambia por `v`.
- `(redim A m n)`: Redimensiona la matriz `A` para que tenga orden $m \times n$.

Podemos ahora dividir las funciones definidas sobre matrices en dos tipos:

1. **Funciones observadoras:** Son aquellas que sólo *observan* a la matriz pasada como argumento. Es decir, no modifica ninguna de sus propiedades, ni sus dimensiones (número de filas o de columnas), ni el valor de ninguno de sus elementos. Ejemplos de estas funciones entre las primitivas básicas podrían ser: `(lookup A i j)`, `(nrows A)` y `(ncolumns A)`.
2. **Funciones modificadoras:** Son aquellas que, aplicadas a una matriz, *sí modifican* alguna de sus propiedades, es decir, cambian el orden de la matriz, o modifican el valor de alguno de sus elementos. Los ejemplos de funciones entre las primitivas básicas son dos: `(update i j v)` y `(redim A m n)`.

Como se ha podido comprobar, la única primitiva que no está clasificada es `(matrixp A)`. Esto es así ya que se va a considerar, precisamente, que el objeto `A` de entrada a las funciones observadoras o modificadoras anteriores ya ha sido validado en el sentido de que se ha comprobado que, efectivamente, cumple con las propiedades e invariantes de una matriz definida con `(matrixp A)`. Por eso considera esta función un caso especial y no debe caer en ninguna de las dos clasificaciones.

Ejemplos de funciones basadas en las primitivas que son del tipo observadoras podrían ser las siguientes: `(identity-matrixp A)`, `(zero-matrixp A)`, `(square-matrixp)` y `(equidimensionalp A B)`, esta última con dos parámetros de tipo matriz.

Los ejemplos de funciones modificadoras cuya definición se basa en las primitivas básicas habría que buscarlas en aquellas que devuelven el tipo matriz, como por ejemplo, `(negate A)`, `(transpose B A)`, `(add-matrix A B)` y `(mutiply-matrix C A B)`. En este caso no hay funciones modificadoras que *devuelvan dos matrices*, ya que en ACL2 (como en la gran mayoría de lenguajes de alto nivel funcionales), sólo se puede devolver un objeto. Además, como no se admiten efectos colaterales en ACL2, podemos tener la seguridad que no se modificarán ninguno de los argumentos pasados a las funciones, o lo que es lo mismo, no se admiten parámetros de entrada y salida en ACL2.

A lo largo de los muchos intentos de demostración que se han llevado a cabo, se ha descubierto un esquema común de lemas básicos que es conveniente demostrar de las funciones *modificadoras* antes de

pasar a demostrar propiedades más complicadas. Estos lemas, si no se demuestran desde el principio terminan precisándose más tarde o más temprano a lo largo de otras demostraciones.

Estos lemas fundamentales son:

- **Clausura**: Recordemos que la clausura en matemáticas es la propiedad que tienen algunas operaciones sobre ciertos conjuntos que consiste en que si se efectúa la operación entre elementos de ese conjunto, lo que se obtiene no sale de éste. Aplicado al conjunto de las matrices deberemos demostrar que toda modificadora aplicada a una matriz devuelve otra matriz.
- **Número de filas**: Como las modificadoras pueden modificar las filas de una matriz, hay que saber cómo queda ese número de filas después de efectuada la operación.
- **Número de columnas**: Ídem con el número de columnas de una matriz.
- **Elementos individuales**: Algo especialmente útil en posteriores demostraciones es saber qué valor tienen todos los elementos individuales de la matriz modificada, es decir, lo que devolvería la aplicación de la operación `(lookup A i j)` después de aplicada la operación modificadora. Por ejemplo, al aplicar la función `(get-zero-matrix A m n)` se devolvería siempre el valor cero para cada uno de los elementos de la matriz resultado.

Un ejemplo más completo lo veremos en la sección siguiente donde demostraremos estos lemas básicos para las primitivas modificadoras, esto es, `(update A i j v)` y `(redim A m n)`.

4.1. Lemas básicos sobre las primitivas[1]

Empezaremos con algo de notación. Como este proyecto se ha realizado con matrices cuyos elementos son de tipo racional se denota por $A \in \mathbb{Q}$ a una matriz A cuyos elementos son de tipo racional sin especificar sus dimensiones y a $A \in \mathbb{Q}^{m \times n}$ a una matriz A cuyos elementos son de tipo racional y tiene orden $m \times n$, es decir tiene m filas, $r(A) = m$ y n columnas, $c(A) = n$.

Lema 1. ***Clausura sobre `(update A i j v)`.***

Sea $A \in \mathbb{Q}$ y B el resultado de aplicar a A la actualización $a_{ij} \leftarrow v$ con $v \in \mathbb{Q}$, $0 \leq i < m$ y $0 \leq j < n$, entonces $B \in \mathbb{Q}$.

```
(defthm pfm-matrixp-update
  (implies (and (matrixp A)
                (rationalp v)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (matrixp (update A i j v))))
```

Lema 2. ***Número de filas de `(update A i j v)`.***

Sea $A \in \mathbb{Q}$ y B el resultado de aplicar a A la actualización $a_{ij} \leftarrow v$ con $v \in \mathbb{Q}$, $0 \leq i < m$ y $0 \leq j < n$, entonces $r(B) = r(A)$. En otras palabras, la operación ***`(update A i j v)`*** conserva *el número de filas.*

[1] Demostrados en el fichero del proyecto `abstract-stobj.lisp`

```
(defthm pfm-nrows-update
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (equal (nrows (update A i j v)) (nrows A))))
```

Lema 3. ***Número de columnas de*** `(update A i j v)`.

Sea $A \in \mathbb{Q}$ *y* B *el resultado de aplicar a* A *la actualización* $a_{ij} \leftarrow v$ *con* $v \in \mathbb{Q}$, $0 \leq i < m$ *y* $0 \leq j < n$, *entonces* $c(B) = c(A)$. *En otras palabras, la operación* **`(update A i j v)`** conserva *el número de columnas.*

```
(defthm pfm-ncolumns-update
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (equal (ncolumns (update A i j v)) (ncolumns A))))
```

Lema 4. ***Elementos de*** `(update A i j v)` ***en la misma posición.***

Sea $A \in \mathbb{Q}$ *y* B *el resultado de aplicar a* A *la actualización* $a_{ij} \leftarrow v$, *entonces* $b_{ij} = v$.

```
(defthm pfm-lookup-update-same
  (equal (lookup (update A i j v) i j) v))
```

Lema 5. ***Elementos de*** `(update A i j v)` ***en posición diferente.***

Sea $A \in \mathbb{Q}$ *y* B *el resultado de aplicar a* A *la actualización* $a_{ij} \leftarrow v$, *con* $k \neq i$ *o* $l \neq j$ *entonces* $b_{kl} = a_{kl}$.

```
(defthm pfm-lookup-update-diff
  (implies (or (not (equal i k))
               (not (equal j l))))
  (equal (lookup (update A i j v) k l)
         (lookup A k l))))
```

Seguimos este apartado con la demostración de los mismos lemas para la modificadora `(redim A m n)`.

Lema 6. ***Clausura de*** `(redim A m n)`.

Sea A *el resultado de aplicar la operación* **`(redim A m n)`**, *con* $m \geq 1$ *y* $n \geq 1$, *entonces* $A \in \mathbb{Q}$.

```
(defthm pfm-matrixp-redim
  (implies (and (natp m)
                (<= 1 m)
                (natp n)
                (<= 1 n))
           (matrixp (redim A m n))))
```

Lema 7. ***Número de filas de*** *(redim A m n).*

Sea A el resultado de aplicar la operación **(redim A m n)**, *con $m \geq 1$ y $n \geq 1$, entonces $r(A) = m$.*

```
(defthm pfm-nrows-redim
  (implies (and (natp m)
                (<= 1 m)
                (natp n)
                (<= 1 n))
           (equal (nrows (redim A m n)) m)))
```

Lema 8. ***Número de columnas de*** *(redim A m n).*

Sea A el resultado de aplicar la operación **(redim A m n)**, *con $m \geq 1$ y $n \geq 1$, entonces $c(A) = n$.*

```
(defthm pfm-ncolumns-redim
  (implies (and (natp m)
                (<= 1 m)
                (natp n)
                (<= 1 n))
           (equal (ncolumns (redim A m n)) n)))
```

Lema 9. ***Elementos de*** *(redim A m n).*

Sea A el resultado de aplicar la operación **(redim A m n)**, *con $m \geq 1$ y $n \geq 1$, entonces $a_{ij} = 0$ si $0 \leq i < m$ y $0 \leq j < n$.*

```
(defthm pfm-lookup-redim
  (implies (and (natp m)
                (<= 1 m)
                (natp n)
                (<= 1 n)
                (natp i)
                (natp j)
                (< i m)
                (< j n))
           (equal (lookup (redim A m n) i j) 0)))
```

Las siguientes propiedades se usan como apoyo en la demostración de teoremas posteriores más importantes:

Lema 10. ***Filas y columnas de una matriz.***

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $m \leq 1$ *y* $n \leq 1$*.*

```
(defthm pfm-matrixp-nrows
  (implies (matrixp A)
           (<= 1 (nrows A)))
  :rule-classes :linear)

(defthm pfm-matrixp-ncolumns
  (implies (matrixp A)
           (<= 1 (ncolumns A)))
  :rule-classes :linear)
```

Hacer notar que se han generado dos reglas de tipo `:linear` que, en ACL2, se utiliza para operar con aritmética lineal y operadores de comparación del tipo $<$, $\leq$, $>$ y $\geq$.

Lema 11. ***Elementos de una matriz.***

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $a_{ij} \in \mathbb{Q}$ *si* $0 \leq i < m$ *y* $0 \leq j < n$*.*

```
(defthm pfm-rationalp-lookup
  (implies (and (natp i)
                (natp j)
                (matrixp A)
                (< i (nrows A))
                (< j (ncolumns A)))
           (rationalp (lookup A i j))))
```

4.2. Teoremas sobre las definiciones básicas[2]

A partir de este punto se van a demostrar una serie de propiedades y teoremas sobre las funciones definidas en la sección anterior. Si los lemas anteriores están bien planteados podemos prescindir de las definiciones originales de las primitivas y razonar directamente con ellas. Por ejemplo, hay que hacer que el demostrador no sustituya `lookup` y `update` por su definición en base a `nth` y `update-nth` (ya que razonar con ellas es bastante tedioso) y hacer las demostraciones con las propias primitivas.

Esto se consigue con el siguiente evento en ACL2:

```
(in-theory (disable ncolumns
                    nrows
                    update
                    lookup
                    redim
                    matrixp))
```

[2]Demostrados en el fichero del proyecto `defun-thms.lisp`

Teorema 1. ***La relación de equidimensionalidad es una relación de equivalencia.***

Sea $equidim(A, B)$ *la relación de equidimensionalidad entre dos matrices* A *y* B*, entonces se cumple que esta relación:*

- ***Es reflexiva:*** $equidim(A, A)$
- ***Es simétrica:*** $equidim(A, B) \rightarrow equidim(B, A)$
- ***Es transitiva:*** $equidim(A, B) \wedge equidim(B, C) \rightarrow equidim(A, C)$

Es decir, la relación de equidimensionalidad es una relación de equivalencia.

Gracias a la propia definición de `(equidimensionalp A B)`, basta con escribir el siguiente evento `defequiv` en ACL2 para conseguir la demostración. Recuérdese que este evento crea una regla de la clase `:equivalence` que se utiliza para demostrar congruencias en ACL2 (eventos `defcong`).

```
(defequiv equidimensionalp)
```

Teorema 2. ***La igualdad entre matrices es una relación de equivalencia.***

Sea $A = B$ *la igualdad entre dos matrices* A *y* B*, entonces se cumple que esta relación:*

- ***Es reflexiva:*** $A = A$
- ***Es simétrica:*** $A = B \rightarrow B = A$
- ***Es transitiva:*** $A = B \wedge B = C \rightarrow A = C$

Es decir, la relación de igualdad entre matrices es una relación de equivalencia.

```
(defthm pfm-reflexivity-of-equal-row
  (equal-row X X m n))

(defthm pfm-symmetry-of-equal-row
  (implies (equal-row M1 M2 m n)
           (equal-row M2 M1 m n)))

(defthm pfm-transitivity-of-equal-row
  (implies (and (equal-row M1 M2 m n)
                (equal-row M2 M3 m n))
           (equal-row M1 M3 m n)))

(defthm pfm-reflexivity-of-equal-rows
  (equal-rows X X m n))

(defthm pfm-symmetry-of-equal-rows
  (implies (equal-rows M1 M2 m n)
           (equal-rows M2 M1 m n)))

(defthm pfm-transitivity-of-equal-rows
  (implies (and (equal-rows M1 M2 m n)
                (equal-rows M2 M3 m n))
           (equal-rows M1 M3 m n)))

(defequiv equal-matrix)
```

En este caso ha habido que demostrar los tres lemas intermedios anteriores para las funciones auxiliares `(equal-matrix-row)` y `(equal-matrix-rows)` antes de llegar al evento `defequiv` final. Recuérdese que estas dos funciones eran las auxiliares que le hacían falta a `(equal-matrix)` para estar definida correctamente.

Esto ha sido una tónica habitual en el desarrollo de todo el proyecto ya que la gran mayoría de funciones se han definido usando esta técnica. Si se logra demostrar la propiedad deseada para las dos funciones auxiliares se debe lograr demostrar la propiedad sobre la función principal.

Este tipo de equivalencias sirven para demostrar congruencias en ACL2. Las congruencias se usan para probar que una relación de equivalencia *mantiene* otra en un determinado argumento de una función. Por ejemplo:

```
(defcong equal-matrix equal (matrixp M) 1)
```

Sirve para demostrar directamente este teorema:

Teorema 3. ***Congruencia de `equal-matrix` sobre `matrixp`.***

Si $A = B$, entonces si A es una matriz, B también lo es y si A no es una matriz, B tampoco lo es.

```
(defthm equal-matrix-implies-equal-matrixp-1
  (implies (equal-matrix M M-equiv)
           (equal (matrixp M) (matrixp M-equiv)))
  :rule-classes (:congruence))
```

Teorema 4. ***Clausura de `(get-zero-matrix A m n)`.***

Si A es el resultado de la operación `(get-zero-matrix A m n)`, entonces $A \in \mathbb{Q}$ con $m \geq 1$ y $n \geq 1$.

```
(defthm pfm-matrixp-zero-matrix
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n))
           (matrixp (get-zero-matrix A m n))))
```

Teorema 5. ***Número de filas de `(get-zero-matrix A m n)`.***

Si A es el resultado de la operación `(get-zero-matrix A m n)`, entonces $r(A) = m$ con $m \geq 1$ y $n \geq 1$.

```
(defthm pfm-nrows-zero-matrix
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n))
           (equal (nrows (get-zero-matrix A m n)) m)))
```

Teorema 6. ***Número de columnas de*** *(get-zero-matrix A m n).*

Si A es el resultado de la operación **(get-zero-matrix A m n)**, *entonces $c(A) = n$ con $m \geq 1$ y $n \geq 1$.*

```
(defthm pfm-ncolumns-zero-matrix
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n))
           (equal (ncolumns (get-zero-matrix A m n)) n)))
```

Teorema 7. ***Elementos de*** *(get-zero-matrix A m n).*

Si A es el resultado de la operación **(get-zero-matrix A m n)**, *entonces $a_{ij} = 0$ con $m \geq 1$, $n \geq 1$, $0 \leq i < m$ y $0 \leq j < n$.*

```
(defthm pfm-lookup-zero-matrix
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n)
                (natp i)
                (natp j)
                (< i m)
                (< j n))
           (equal (lookup (get-zero-matrix A m n) i j) 0)))
```

Teorema 8. ***Clausura del predicado*** *(zero-matrixp A).*

Si A cumple con el predicado **(zero-matrixp A)**, *entonces $A \in \mathbb{Q}$.*

```
(defthm pfm-zero-matrixp-matrixp
  (implies (zero-matrixp A)
           (matrixp A)))
```

Teorema 9. ***Elementos del predicado*** *(zero-matrixp A).*

Si A cumple con el predicado **(zero-matrixp A)**, *entonces $a_{ij} = 0$ con $0 \leq i < m$ y $0 \leq j < n$.*

```
(defthm pfm-zero-matrixp-lookup
  (implies (and (zero-matrixp A)
                (natp i)
                (< i (nrows A))
                (natp j)
                (< j (ncolumns A)))
           (equal (lookup A i j) 0)))
```

Teorema 10. ***Clausura de la función*** *`(get-identity-matrix A n)`.*

Si A es el resultado de la función `(get-identity-matrix A n)` *con $n \geq 1$, entonces $A \in \mathbb{Q}$ y $A = I_n$.*

```
(defthm pfm-matrixp-identity-matrix
  (implies (and (natp n)
                (<= 1 n))
           (matrixp (get-identity-matrix A n))))
```

Teorema 11. ***Número de filas de*** *`(get-identity-matrix A n)`.*

Si $A = I_n$ con $n \geq 1$, entonces $r(A) = n$.

```
(defthm pfm-nrows-identity-matrix
  (implies (and (natp n)
                (<= 1 n))
           (equal (nrows (get-identity-matrix A n))
                  n)))
```

Teorema 12. ***Número de columnas de*** *`(get-identity-matrix A n)`.*

Si $A = I_n$ con $n \geq 1$, entonces $c(A) = n$.

```
(defthm pfm-ncolumns-identity-matrix
  (implies (and (natp n)
                (<= 1 n))
           (equal (ncolumns (get-identity-matrix A n))
                  n)))
```

Teorema 13. ***La matriz identidad es cuadrada.***

Si $A = I_n$, entonces $r(A) = c(A)$. Es decir, la matriz identidad de cualquier orden es cuadrada.

```
(defthm pfm-identity-matrixp-square-matrixp
  (implies (identity-matrixp A)
           (square-matrixp A)))
```

Teorema 14. ***Elementos de la matriz identidad.***

Si $A = I_n$ con $n \geq 1$, $0 \leq i < n$ y $0 \leq j < n$ entonces:

$$a_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases}$$

```
(defthm pfm-lookup-identity-matrix
  (implies (and (natp i)
                (natp j)
                (natp n)
                (<= 1 n)
```

```
                (< i n)
                (< j n))
           (equal (lookup (get-identity-matrix A n) i j)
                  (if (equal i j) 1 0))))

(defthm pfm-lookup-identity-matrixp
  (implies (and (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A))
                (identity-matrixp A))
           (equal (lookup A i j)
                  (if (equal i j) 1 0))))
```

4.3. Teoremas sobre las operaciones unarias[3]

Empezamos demostrando propiedades sobre la operación de trasposición de matrices.

Teorema 15. ***Clausura de la trasposición de matrices.***

Si $A \in \mathbb{Q}$ *, entonces* $A^T \in \mathbb{Q}$.

```
(defthm pfm-matrixp-transpose
  (implies (matrixp A)
           (matrixp (transpose B A))))
```

Teorema 16. ***Número de filas de la trasposición de matrices.***

Si $A \in \mathbb{Q}^{m \times n}$ *, entonces* $r(A^T) = n$.

```
(defthm pfm-nrows-transpose
  (implies (matrixp A)
           (equal (nrows (transpose B A))
                  (ncolumns A))))
```

Teorema 17. ***Número de columnas de la trasposición de matrices.***

Si $A \in \mathbb{Q}^{m \times n}$ *, entonces* $c(A^T) = m$.

```
(defthm pfm-ncolumns-transpose
  (implies (matrixp A)
           (equal (ncolumns (transpose B A))
                  (nrows A))))
```

Teorema 18. ***La traspuesta de una matriz cuadrada es también cuadrada.***

Si $r(A) = c(A)$ *, entonces* $r(A^T) = c(A^T)$.

[3] Demostrados en el fichero del proyecto `unary-ops-thms.lisp`

```
(defthm pfm-square-matrixp-transpose
  (implies (square-matrixp A)
           (square-matrixp (transpose B A))))
```

Teorema 19. ***Elementos de la matriz traspuesta.***

Sea $A \in \mathbb{Q}^{m \times n}$ *y* $B = A^T$*, entonces* $b_{ij} = a_{ji}$ *con* $0 \leq i < n$ *y* $0 \leq j < m$*.*

```
(defthm pfm-lookup-transpose
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (ncolumns A))
                (< j (nrows A)))
           (equal (lookup (transpose B A) i j)
                  (lookup A j i))))
```

Teorema 20. ***Idempotencia de la operación de trasposición.***

Sea $A \in \mathbb{Q}$*, entonces* $A = (A^T)^T$*.*

```
(defthm pfm-idempotency-of-transpose
  (implies (matrixp A)
           (equal (transpose C (transpose B A)) A)))
```

Teorema 21. ***Trasposición de la matriz nula.***

Si A *es el resultado de la operación* `(get-zero-matrix A m n)`*, entonces* A^T *es el resultado de la operación* `(get-zero-matrix A n m)` *con* $m \geq 1$*,* $n \geq 1$*.*

```
(defthm pfm-equal-matrix-transpose-zero-matrix
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n))
           (equal (transpose B (get-zero-matrix A m n))
                  (get-zero-matrix A n m))))
```

Teorema 22. ***Congruencia de*** `equal-matrix` ***sobre*** `transpose`***.***

Si $A = B$*, entonces* $A^T = B^T$*.*

```
(defcong equal-matrix equal-matrix (transpose B A) 2)
```

Teorema 23. ***Trasposición de la matriz identidad.***

La operación de trasposición sobre una matriz identidad de cualquier orden deja la matriz inalterada, es decir, $(I_n)^T = I_n$*.*

```
(defthm pfm-transpose-identity-matrix
  (implies (and (natp n)
                (<= 1 n))
           (equal (transpose B (get-identity-matrix A n))
                  (get-identity-matrix A n))))
```

Pasamos ahora a demostrar teoremas sobre la operación de negación de una matriz (opuesta de una matriz).

Teorema 24. ***Clausura de la matriz opuesta.***

Si $A \in \mathbb{Q}$ *y* $B = -A$*, entonces* $B \in \mathbb{Q}$*.*

```
(defthm pfm-matrixp-negate
  (implies (matrixp A)
           (matrixp (negate A))))
```

Teorema 25. ***Número de filas de la matriz opuesta.***

Si $A \in \mathbb{Q}^{m \times n}$ *y* $B = -A$*, entonces* $r(B) = m$*.*

```
(defthm pfm-nrows-negate
  (implies (matrixp A)
           (equal (nrows (negate A)) (nrows A))))
```

Teorema 26. ***Número de columnas de la matriz opuesta.***

Si $A \in \mathbb{Q}^{m \times n}$ *y* $B = -A$*, entonces* $c(B) = n$*.*

```
(defthm pfm-ncolumns-negate
  (implies (matrixp A)
           (equal (ncolumns (negate A)) (ncolumns A))))
```

Teorema 27. ***Elementos de la matriz opuesta.***

Si $A \in \mathbb{Q}^{m \times n}$ *y* $B = -A$*, entonces* $b_{ij} = -a_{ij}$ *con* $0 \leq i < m$ *y* $0 \leq j < n$*.*

```
(defthm pfm-lookup-negate
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (equal (lookup (negate A) i j)
                  (- (lookup A i j)))))
```

Teorema 28. ***Idempotencia de la operación de negación sobre matrices.***

Si $A \in \mathbb{Q}$, *entonces* $A = -(-A)$.

```
(defthm pfm-idempotency-of-negate
  (implies (matrixp A)
           (equal (negate (negate A)) A)))
```

Teorema 29. ***La operación de trasposición y la de negación son intercambiables.***

Si $A \in \mathbb{Q}$, *entonces* $(-A)^T = -(A^T)$.

```
(defthm pfm-transpose-negate
  (implies (matrixp A)
           (equal (transpose B (negate A))
                  (negate (transpose B A)))))
```

Teorema 30. ***Congruencia de*** `equal-matrix` ***sobre*** `negate`***.***

Si $A = B$, *entonces* $-A = -B$.

```
(defcong equal-matrix equal-matrix (negate A) 1)
```

4.4. Teoremas sobre la multiplicación de matrices por un escalar[4]

Los siguientes teoremas tratan sobre propiedades de la multiplicación de una matriz por un escalar.

Teorema 31. ***Clausura de la multiplicación de una matriz por un escalar.***

Si $A \in \mathbb{Q}$ *y* $B = \alpha A$ *con* $\alpha \in \mathbb{Q}$, *entonces* $B \in \mathbb{Q}$.

```
(defthm pfm-matrixp-scalar-multiply
  (implies (and (matrixp A)
                (rationalp k))
           (matrixp (scalar-multiply k A))))
```

Teorema 32. ***Número de filas de la multiplicación de una matriz por un escalar.***

Si $A \in \mathbb{Q}^{m \times n}$ *y* $B = \alpha A$ *con* $\alpha \in \mathbb{Q}$, *entonces* $r(B) = m$.

```
(defthm pfm-nrows-scalar-multiply
  (implies (and (matrixp A)
                (rationalp k))
           (equal (nrows (scalar-multiply k A))
                  (nrows A))))
```

[4] Demostrados en el fichero del proyecto `unary-ops-thms.lisp`

Teorema 33. ***Número de columnas de la multiplicación de una matriz por un escalar.***

Si $A \in \mathbb{Q}^{m\times n}$ y $B = \alpha A$ con $\alpha \in \mathbb{Q}$, entonces $c(B) = n$.

```
(defthm pfm-ncolumns-scalar-multiply
  (implies (and (matrixp A)
                (rationalp k))
           (equal (ncolumns (scalar-multiply k A))
                  (ncolumns A))))
```

Teorema 34. ***Elementos de la multiplicación de una matriz por un escalar.***

Si $A \in \mathbb{Q}^{m\times n}$ y $B = \alpha A$ con $\alpha \in \mathbb{Q}$, entonces $b_{ij} = \alpha \cdot a_{ij}$ con $0 \leq i < m$ y $0 \leq j < n$.

```
(defthm pfm-lookup-scalar-multiply
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A))
                (rationalp k))
           (equal (lookup (scalar-multiply k A) i j)
                  (* k (lookup A i j)))))
```

Teorema 35. ***Propiedad asociativa de la multiplicación de una matriz por un escalar.***

Sean $A \in \mathbb{Q}$ y $\alpha, \beta \in \mathbb{Q}$, entonces $\alpha(\beta A) = (\alpha \cdot \beta)A$.

```
(defthm pfm-associativity-of-scalar-multiply
  (implies (and (matrixp A)
                (rationalp k1)
                (rationalp k2))
           (equal (scalar-multiply k1 (scalar-multiply k2 A))
                  (scalar-multiply (* k1 k2) A))))
```

Teorema 36. ***Elemento neutro de la multiplicación de una matriz por un escalar (1^a parte).***

Sea $A \in \mathbb{Q}^{m\times n}$, entonces $0 \cdot A$ es la matriz nula de orden $m \times n$.

```
(defthm pfm-neutral-element-of-scalar-multiply
  (implies (matrixp A)
           (equal (scalar-multiply 0 A)
                  (get-zero-matrix B (nrows A) (ncolumns A)))))

(defthm pfm-neutral-element-of-scalar-multiply-2
  (implies (matrixp A)
           (zero-matrixp (scalar-multiply 0 A))))
```

Teorema 37. ***Elemento neutro de la multiplicación de una matriz por un escalar (2^a parte).***

Sea $A \in \mathbb{Q}^{m\times n}$ la matriz nula de orden $m \times n$ y $\alpha \in \mathbb{Q}$, entonces $\alpha \cdot A$ es la matriz nula de orden $m \times n$.

```
(defthm pfm-neutral-element-of-scalar-multiply-3
  (implies (and (natp m)
                (natp n)
                (<= 1 m)
                (<= 1 n)
                (rationalp k))
           (equal (scalar-multiply k (get-zero-matrix A m n))
                  (get-zero-matrix A m n))))

(defthm pfm-neutral-element-of-scalar-multiply-4
  (implies (and (zero-matrixp A)
                (rationalp k))
           (zero-matrixp (scalar-multiply k A))))
```

Teorema 38. ***Elemento neutro de la multiplicación de una matriz por un escalar (3^a parte).***

Sea $A \in \mathbb{Q}$, entonces $1 \cdot A = A$.

```
(defthm pfm-neutral-element-of-scalar-multiply-5
  (implies (matrixp A)
           (equal (scalar-multiply 1 A) A)))
```

Teorema 39. ***Elemento opuesto de la multiplicación de una matriz por un escalar.***

Sea $A \in \mathbb{Q}$, entonces $-1 \cdot A = -A$.

```
(defthm pfm-opposite-element-of-scalar-multiply
  (implies (matrixp A)
           (equal (scalar-multiply -1 A) (negate A))))
```

Teorema 40. ***Las operaciones de trasposición y multiplicación de una matriz por un escalar son intercambiables.***

Sea $A \in \mathbb{Q}$ y $\alpha \in \mathbb{Q}$, entonces $(\alpha \cdot A)^T = \alpha \cdot A^T$.

```
(defthm pfm-transpose-scalar-multiply
  (implies (and (matrixp A)
                (rationalp k))
           (equal (transpose B (scalar-multiply k A))
                  (scalar-multiply k (transpose B A)))))
```

Teorema 41. ***Las operaciones de negación y multiplicación de una matriz por un escalar son intercambiables.***

Sea $A \in \mathbb{Q}$ y $\alpha \in \mathbb{Q}$, entonces $\alpha \cdot (-A) = -(\alpha \cdot A)$.

```
(defthm pfm-scalar-multiply-negate
  (implies (and (matrixp A)
                (rationalp k))
           (equal (scalar-multiply k (negate A))
                  (negate (scalar-multiply k A)))))
```

4.5. Teoremas sobre la suma de matrices[5]

Teorema 42. ***Clausura sobre la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$ *y* $C = A + B$*, entonces* $C \in \mathbb{Q}$*.*

```
(defthm pfm-matrixp-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (matrixp (add-matrix A B))))
```

Teorema 43. ***Número de filas de la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$ *y* $C = A + B$*, entonces* $r(C) = m$*.*

```
(defthm pfm-nrows-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (nrows (add-matrix A B)) (nrows A))))
```

Teorema 44. ***Número de columnas de la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$ *y* $C = A + B$*, entonces* $c(C) = n$*.*

```
(defthm pfm-ncolumns-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (ncolumns (add-matrix A B)) (ncolumns A))))
```

Teorema 45. ***Elementos de la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$ *y* $C = A + B$*, entonces* $c_{ij} = a_{ij} + b_{ij}$ *con* $0 \leq i < m$ *y* $0 \leq j < n$*.*

```
(defthm pfm-lookup-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B)
                (natp i)
```

[5] Demostrados en el fichero del proyecto `add-matrix-thms.lisp`

```
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (equal (lookup (add-matrix A B) i j)
                  (+ (lookup A i j) (lookup B i j))))))
```

Teorema 46. ***Propiedad conmutativa de la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m\times n}$, entonces $A + B = B + A$.

```
(defthm pfm-commutativity-of-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (add-matrix A B)
                  (add-matrix B A))))
```

Teorema 47. ***Propiedad asociativa de la suma de matrices.***

Sean $A, B, C \in \mathbb{Q}^{m\times n}$, entonces $(A + B) + C = A + (B + C)$.

```
(defthm pfm-associativity-of-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (matrixp C)
                (equidimensionalp A B)
                (equidimensionalp A C))
           (equal (add-matrix (add-matrix A B) C)
                  (add-matrix A (add-matrix B C)))))
```

Teorema 48. ***Elemento neutro por la derecha de la suma de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\emptyset$ la matriz nula de orden $m \times n$, entonces $A + \emptyset = A$.

```
(defthm pfm-neutral-element-of-add-matrix-rigth
  (implies
    (matrixp A)
    (equal
      (add-matrix A (get-zero-matrix B (nrows A) (ncolumns A)))
      A)))
```

Teorema 49. ***Elemento neutro por la izquierda de la suma de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\emptyset$ la matriz nula de orden $m \times n$, entonces $\emptyset + A = A$.

```
(defthm pfm-neutral-element-of-add-matrix-left
  (implies
    (matrixp A)
    (equal
      (add-matrix (get-zero-matrix B (nrows A) (ncolumns A)) A)
      A)))
```

Teorema 50. ***Elemento opuesto por la derecha de la suma de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\emptyset$ la matriz nula de orden $m \times n$, entonces $A + (-A) = \emptyset$.

```
(defthm pfm-opposite-element-of-add-matrix-right
  (implies (matrixp A)
           (equal (add-matrix A (negate A))
                  (get-zero-matrix B (nrows A) (ncolumns A)))))
```

Teorema 51. ***Elemento opuesto por la izquierda de la suma de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\emptyset$ la matriz nula de orden $m \times n$, entonces $(-A) + A = \emptyset$.

```
(defthm pfm-opposite-element-of-add-matrix-left
  (implies (matrixp A)
           (equal (add-matrix (negate A) A)
                  (get-zero-matrix B (nrows A) (ncolumns A)))))
```

Teorema 52. ***Congruencias de `equal-matrix` sobre la suma de matrices.***

Si $A = B$ entonces $A + C = B + C$ y $C + A = C + B$.

```
(defcong equal-matrix equal (add-matrix A B) 1)
(defcong equal-matrix equal (add-matrix A B) 2)
```

Teorema 53. ***Propiedad distributiva de la multiplicación de un escalar sobre la suma de racionales.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\alpha, \beta \in \mathbb{Q}$, entonces $(\alpha + \beta) \cdot A = \alpha A + \beta A$.

```
(defthm pfm-distributivity-of-scalar-multiply-over-+
  (implies (and (matrixp A)
                (rationalp k1)
                (rationalp k2))
           (equal (scalar-multiply (+ k1 k2) A)
                  (add-matrix (scalar-multiply k1 A)
                              (scalar-multiply k2 A)))))
```

Teorema 54. ***Propiedad distributiva de la multiplicación de un escalar sobre la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m\times n}$ y $\alpha \in \mathbb{Q}$, entonces $\alpha \cdot (A + B) = \alpha A + \alpha B$.

```
(defthm pfm-distributivity-of-scalar-multiply-over-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B)
                (rationalp k))
           (equal (scalar-multiply k (add-matrix A B))
                  (add-matrix (scalar-multiply k A)
                              (scalar-multiply k B)))))
```

Teorema 55. ***La operación de trasposición y la de suma de matrices son intercambiables.***

Sean $A, B \in \mathbb{Q}^{m \times n}$*, entonces* $(A + B)^T = A^T + B^T$*.*

```
(defthm pfm-transpose-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (transpose C (add-matrix A B))
                  (add-matrix (transpose C A)
                              (transpose D B)))))
```

Teorema 56. ***Unicidad del elemento opuesto en la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$ *y* $\emptyset$ *la matriz nula de orden* $m \times n$*, entonces si* $A + B = \emptyset$ *se cumple que* $A = -B$*.*

```
(defthm pfm-uniqueness-of-opposite-element-add-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B)
                (equal-matrix
                  (add-matrix A B)
                  (get-zero-matrix C (nrows A) (ncolumns A))))
           (equal A (negate B))))
```

Teorema 57. ***Propiedad distributiva de la operación de negación sobre la suma de matrices.***

Sean $A, B \in \mathbb{Q}^{m \times n}$*, entonces* $-(A + B) = (-A) + (-B)$*.*

```
(defthm pfm-distributivity-of-negate-over-+
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (negate (add-matrix A B))
                  (add-matrix (negate A) (negate B)))))
```

4.6. Corolarios sobre la suma de matrices[6]

Con los teoremas demostrados anteriormente tenemos una verdadera aritmética de matrices (al menos la parte que involucra a la suma) que nos permite demostrar resultados como los siguientes.

Corolario 1. ***Multiplicar una matriz por el escalar "2" es sumarla sobre sí misma.***

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $A + A = 2 \cdot A$*.*

```
(defthm pfm-double-add-matrix-scalar-multiply
  (implies (matrixp A)
           (equal (add-matrix A A)
                  (scalar-multiply 2 A))))
```

[6] Demostrados en el fichero del proyecto `add-matrix-thms.lisp`

Corolario 2. ***Elemento neutro de la suma de matrices (2ª parte).***

Sean $A, B \in \mathbb{Q}^{m\times n}$*, entonces* $A + (-A) + B = B$*.*

```
(defthm pfm-neutral-element-add-matrix-2
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (add-matrix (add-matrix A (negate A)) B)
                  B)))
```

Corolario 3. ***Elemento neutro de la suma de matrices (3ª parte).***

Sean $A, B \in \mathbb{Q}^{m\times n}$*, entonces* $(-A) + A + B = B$*.*

```
(defthm pfm-neutral-element-add-matrix-3
  (implies (and (matrixp A)
                (matrixp B)
                (equidimensionalp A B))
           (equal (add-matrix (add-matrix (negate A) A) B)
                  B)))
```

4.7. Teoremas sobre la multiplicación de matrices[7]

La verdadera riqueza en la aritmética de matrices se conseguirá gracias a la multiplicación entre matrices y sus propiedades y teoremas demostrados a continuación:

Teorema 58. ***Clausura de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$*,* $B \in \mathbb{Q}^{n\times p}$ *y* $C = AB$ *entonces* $C \in \mathbb{Q}$*.*

```
(defthm pfm-matrixp-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B)))
           (matrixp (multiply-matrix C A B))))
```

Teorema 59. ***Número de filas de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$*,* $B \in \mathbb{Q}^{n\times p}$ *y* $C = AB$ *entonces* $r(C) = m$*.*

```
(defthm pfm-nrows-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B)))
           (equal (nrows (multiply-matrix C A B))
                  (nrows A)))
```

[7] Demostrados en el fichero del proyecto `multiply-matrix-thms.lisp`

Teorema 60. ***Número de columnas de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{n\times p}$ y $C = AB$ entonces $c(C) = p$.

```
(defthm pfm-ncolumns-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B)))
           (equal (ncolumns (multiply-matrix C A B))
                  (ncolumns B))))
```

Teorema 61. ***Elementos de la multiplicación de matrices.***

Si $M \in \mathbb{Q}^{m\times n}$ denotemos por $\overrightarrow{f_i}(M)$ a la fila i-ésima (con $0 \leq i < m$) de la matriz M y por $\overrightarrow{c_j}(M)$ a la fila j-ésima (con $0 \leq j < n$) de la matriz M. Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{n\times p}$ y $C = AB$ entonces $c_{ij} = \overrightarrow{f_i}(A) \cdot \overrightarrow{c_j}(B)$ siendo en esta caso la operacion $\cdot$ el producto escalar entre vectores.

```
(defthm pfm-lookup-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B))
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns B)))
           (equal (lookup (multiply-matrix C A B) i j)
                  (dot-product A B i (1- (ncolumns A)) j))))
```

Teorema 62. ***Congruencias de `equal-matrix` en los dos operandos de la multiplicación de matrices.***

Si $A = B$, entonces $AC = BC$ y $CA = CB$.

```
(defcong equal-matrix equal (multiply-matrix C A B) 2)
(defcong equal-matrix equal (multiply-matrix C A B) 3)
```

Teorema 63. ***Elemento neutro por la izquierda de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ y $\emptyset_{mn}$ la matriz nula de orden $m \times n$, entonces $\emptyset_{km} \cdot A = \emptyset_{kn}$.

```
(defthm pfm-neutral-element-multiply-matrix-left
  (implies (and (matrixp A)
                (natp k)
                (<= 1 k))
           (equal (multiply-matrix C
                   (get-zero-matrix B k (nrows A))
                   A)
                  (get-zero-matrix B k (ncolumns A)))))
```

Teorema 64. ***Elemento neutro por la derecha de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ *y* $\emptyset_{mn}$ *la matriz nula de orden* $m \times n$*, entonces* $A \cdot \emptyset_{nk} = \emptyset_{mk}$*.*

```
(defthm pfm-neutral-element-multiply-matrix-right
  (implies (and (matrixp A)
                (natp k)
                (<= 1 k))
           (equal (multiply-matrix C
                   A
                   (get-zero-matrix B (ncolumns A) k))
                  (get-zero-matrix B (nrows A) k))))
```

Teorema 65. ***Elemento unidad por la izquierda de la multiplicación de matrices.***

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $I_m \cdot A = A$*.*

```
(defthm pfm-unity-element-multiply-matrix-left
  (implies (matrixp A)
           (equal (multiply-matrix C
                   (get-identity-matrix B (nrows A))
                   A)
                  A)))
```

Teorema 66. ***Elemento unidad por la derecha de la multiplicación de matrices.***

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $A \cdot I_n = A$*.*

```
(defthm pfm-unity-element-multiply-matrix-right
  (implies (matrixp A)
           (equal (multiply-matrix C
                   A
                   (get-identity-matrix B (ncolumns A)))
                  A)))
```

Teorema 67. ***Propiedad asociativa de la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$*,* $B \in \mathbb{Q}^{n\times p}$ *y* $C \in \mathbb{Q}^{p\times q}$*, entonces* $(AB)C = A(BC)$*.*

```
(defthm pfm-associativity-of-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (matrixp C)
                (equal (ncolumns A) (nrows B))
                (equal (ncolumns B) (nrows C)))
           (equal (multiply-matrix E
                   (multiply-matrix D A B) C)
                  (multiply-matrix E
                   A (multiply-matrix D B C)))))
```

Teorema 68. ***Propiedad distributiva de la multiplicación de matrices sobre la suma de matrices por la izquierda.***

Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{n\times p}$ *y* $C \in \mathbb{Q}^{n\times p}$, *entonces* $A(B+C) = AB + AC$.

```
(defthm pfm-distributivity-of-multiply-matrix-add-matrix-left
  (implies (and (matrixp A)
                (matrixp B)
                (matrixp C)
                (equidimensionalp B C)
                (equal (ncolumns A) (nrows B)))
           (equal (multiply-matrix D A (add-matrix B C))
                  (add-matrix (multiply-matrix D A B)
                              (multiply-matrix E A C)))))
```

Teorema 69. ***Propiedad distributiva de la multiplicación de matrices sobre la suma de matrices por la derecha.***

Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{m\times n}$ *y* $C \in \mathbb{Q}^{n\times p}$, *entonces* $(A+B)C = AC + BC$.

```
(defthm pfm-distributivity-of-multiply-matrix-add-matrix-right
  (implies (and (matrixp A)
                (matrixp B)
                (matrixp C)
                (equidimensionalp A B)
                (equal (ncolumns A) (nrows C)))
           (equal (multiply-matrix D (add-matrix A B) C)
                  (add-matrix (multiply-matrix D A C)
                              (multiply-matrix E B C)))))
```

Teorema 70. ***Las operaciones de multiplicación de matrices y la negación de matrices son intercambiables por la izquierda.***

Sean $A \in \mathbb{Q}^{m\times n}$ *y* $B \in \mathbb{Q}^{n\times p}$, *entonces* $(-A)\cdot B = -(AB)$.

```
(defthm pfm-multiply-matrix-negate-left
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B)))
           (equal (multiply-matrix C (negate A) B)
                  (negate (multiply-matrix C A B)))))
```

Teorema 71. ***Las operaciones de multiplicación de matrices y la negación de matrices son intercambiables por la derecha.***

Sean $A \in \mathbb{Q}^{m\times n}$ *y* $B \in \mathbb{Q}^{n\times p}$, *entonces* $A\cdot(-B) = -(AB)$.

```
(defthm pfm-multiply-matrix-negate-right
  (implies (and (matrixp A)
                (matrixp B)
```

```
                (equal (ncolumns A) (nrows B)))
           (equal (multiply-matrix C A (negate B))
                  (negate (multiply-matrix C A B)))))
```

Teorema 72. ***Las operaciones de multiplicación de matrices y la multiplicación de matrices por un escalar son intercambiables por la izquierda.***

Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{n\times p}$ *y* $\alpha \in \mathbb{Q}$, *entonces* $(\alpha A)\cdot B = \alpha\cdot(AB)$.

```
(defthm pfm-multiply-matrix-scalar-multiply-left
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A)(nrows B))
                (rationalp k))
           (equal (multiply-matrix C (scalar-multiply k A) B)
                  (scalar-multiply k (multiply-matrix C A B)))))
```

Teorema 73. ***Las operaciones de multiplicación de matrices y la multiplicación de matrices por un escalar son intercambiables por la derecha.***

Sean $A \in \mathbb{Q}^{m\times n}$, $B \in \mathbb{Q}^{n\times p}$ *y* $\alpha \in \mathbb{Q}$, *entonces* $A\cdot(\alpha B) = \alpha\cdot(AB)$.

```
(defthm pfm-multiply-matrix-scalar-multiply-right
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A)(nrows B))
                (rationalp k))
           (equal (multiply-matrix C A (scalar-multiply k B))
                  (scalar-multiply k (multiply-matrix C A B)))))
```

Teorema 74. ***La operación de trasposición y la multiplicación de matrices.***

Sean $A \in \mathbb{Q}^{m\times n}$ *y* $B \in \mathbb{Q}^{n\times p}$, *entonces* $(AB)^T = B^T\cdot A^T$.

```
(defthm pfm-transpose-multiply-matrix
  (implies (and (matrixp A)
                (matrixp B)
                (equal (ncolumns A) (nrows B)))
           (equal (transpose D (multiply-matrix C A B))
                  (multiply-matrix E (transpose D B)
                                     (transpose C A)))))
```

4.8. Teoremas sobre las transformaciones por filas[8]

Teorema 75. ***Clausura de la transformación*** F_{ij}.

Sea $A \in \mathbb{Q}^{m\times n}$, *entonces* $F_{ij}A \in \mathbb{Q}$ *con* $0 \le i < m$ *y* $0 \le j < m$.

[8] Demostrados en el fichero del proyecto `row-operations.lisp`

```
(defthm pfm-matrixp-exchange-rows
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (nrows A)))
           (matrixp (exchange-rows A i j))))
```

Teorema 76. ***Número de filas de la transformación*** F_{ij}**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $r(F_{ij}A) = m$ *con* $0 \leq i < m$ *y* $0 \leq j < m$*.*

```
(defthm pfm-nrows-exchange-rows
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (nrows (exchange-rows A i j))
                  (nrows A))))
```

Teorema 77. ***Número de columnas de la transformación*** F_{ij}**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $c(F_{ij}A) = n$ *con* $0 \leq i < m$ *y* $0 \leq j < m$*.*

```
(defthm pfm-ncolumns-exchange-rows
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (ncolumns (exchange-rows A i j))
                  (ncolumns A))))
```

Teorema 78. ***Elementos de la transformación*** F_{ij}**.**

Sea $A \in \mathbb{Q}^{m\times n}$ *y* $B = F_{ij}A$ *con* $0 \leq i < m$ *y* $0 \leq j < m$*, entonces:*

$$b_{mn} = \begin{cases} a_{jn} & i = m \\ a_{in} & j = m \\ a_{mn} & e.o.c \end{cases}$$

```
(defthm pfm-lookup-exchange-rows
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (natp m)
                (natp n)
                (< i (nrows A))
                (< j (nrows A))
```

```
                (< m (nrows A))
                (< n (ncolumns A)))
           (equal (lookup (exchange-rows A i j) m n)
                  (cond ((= i m) (lookup A j n))
                        ((= j m) (lookup A i n))
                        (t (lookup A m n))))))
```

Teorema 79. ***Intercambiar una fila con sí misma deja igual a la matriz.***

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $F_{ii}A = A$ *con* $0 \leq i < m$*.*

```
(defthm pfm-exchange-rows-i-i
  (implies (and (matrixp A)
                (natp i)
                (< i (nrows A)))
           (equal (exchange-rows A i i) A)))
```

Teorema 80. ***Equivalencia entre la transformación*** F_{ij} ***y la multiplicación de*** $F_{ij}I_m$**.**

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $F_{ij}I_m \cdot A = F_{ij}A$ *con* $0 \leq i < m$ *y* $0 \leq j < m$*.*

```
(defthm pfm-exchange-rows-identity-matrix
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (multiply-matrix C
                   (exchange-rows
                    (get-identity-matrix B (nrows A))
                     i
                     j)
                   A)
                  (exchange-rows A i j))))
```

Teorema 81. ***Clausura de la transformación*** $F_i(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $F_i(\alpha)A \in \mathbb{Q}$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-matrixp-multiply-row
  (implies (and (matrixp A)
                (natp i)
                (rationalp alpha)
                (< i (nrows A)))
           (matrixp (multiply-row A i alpha))))
```

Teorema 82. ***Número de filas de la transformación*** $F_i(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m \times n}$*, entonces* $r(F_i(\alpha)A) = m$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-nrows-multiply-row
  (implies (and (matrixp A)
                (natp i)
                (rationalp alpha)
                (< i (nrows A)))
           (equal (nrows (multiply-row A i alpha))
                  (nrows A))))
```

Teorema 83. ***Número de columnas de la transformación*** $F_i(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $c(F_i(\alpha)A) = n$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-ncolumns-multiply-row
  (implies (and (matrixp A)
                (natp i)
                (rationalp alpha)
                (< i (nrows A)))
           (equal (ncolumns (multiply-row A i alpha))
                  (ncolumns A))))
```

Teorema 84. ***Elementos de la transformación*** $F_i(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$ *y* $B = F_i(\alpha)A$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*, entonces:*

$$b_{mn} = \begin{cases} \alpha \cdot a_{in} & i = m \\ a_{mn} & e.o.c \end{cases}$$

```
(defthm pfm-lookup-multiply-row
  (implies (and (matrixp A)
                (natp i)
                (natp m)
                (natp n)
                (rationalp alpha)
                (< i (nrows A))
                (< m (nrows A))
                (< n (ncolumns A)))
           (equal (lookup (multiply-row A i alpha) m n)
                  (cond ((= i m) (* (lookup A m n) alpha))
                        (t (lookup A m n))))))
```

Teorema 85. ***Equivalencia entre la transformación*** $F_i(\alpha)$ ***y la multiplicación de*** $F_i(\alpha)I_m$**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $F_i(\alpha)I_m \cdot A = F_i(\alpha)A$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-multiply-row-identity-matrix
  (implies (and (matrixp A)
                (natp i)
                (rationalp alpha)
                (< i (nrows A)))
           (equal (multiply-matrix C
```

```
                    (multiply-row
                     (get-identity-matrix B (nrows A))
                      i
                      alpha)
                    A)
                   (multiply-row A i alpha))))
```

Teorema 86. ***Clausura de la transformación*** $F_{ij}(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $F_{ij}(\alpha)A \in \mathbb{Q}$ *con* $0 \leq i < m$*,* $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-matrixp-saxpy-row
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (rationalp alpha)
                (< i (nrows A))
                (< j (nrows A)))
           (matrixp (saxpy-row A j i alpha))))
```

Teorema 87. ***Número de filas de la transformación*** $F_{ij}(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $r(F_{ij}(\alpha)A) = m$ *con* $0 \leq i < m$*,* $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-nrows-saxpy-row
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (rationalp alpha)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (nrows (saxpy-row A j i alpha))
                  (nrows A))))
```

Teorema 88. ***Número de columnas de la transformación*** $F_{ij}(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$*, entonces* $c(F_{ij}(\alpha)A) = n$ *con* $0 \leq i < m$*,* $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-ncolumns-saxpy-row
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (rationalp alpha)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (ncolumns (saxpy-row A j i alpha))
                  (ncolumns A))))
```

Teorema 89. ***Elementos de la transformación*** $F_{ij}(\alpha)$**.**

Sea $A \in \mathbb{Q}^{m\times n}$ *y* $B = F_{ij}(\alpha)A$ *con* $0 \leq i < m$, $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$, *entonces:*

$$b_{mn} = \begin{cases} \alpha \cdot a_{jn} + a_{in} & i = m \\ a_{mn} & e.o.c \end{cases}$$

```
(defthm pfm-lookup-saxpy-row
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (natp m)
                (natp n)
                (rationalp alpha)
                (< i (nrows A))
                (< j (nrows A))
                (< m (nrows A))
                (< n (ncolumns A)))
           (equal (lookup (saxpy-row A i j alpha) m n)
                  (cond ((= i m) (+ (lookup A i n)
                                    (* (lookup A j n) alpha)))
                        (t (lookup A m n))))))
```

Teorema 90. ***Equivalencia entre la transformación*** $F_{ij}(\alpha)$ ***y la multiplicación de*** $F_{ij}(\alpha)I_m$**.**

Sea $A \in \mathbb{Q}^{m\times n}$, *entonces* $F_{ij}(\alpha)I_m \cdot A = F_{ij}(\alpha)A$ *con* $0 \leq i < m$, $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$.

```
(defthm pfm-saxpy-row-identity-matrix
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (rationalp alpha)
                (< i (nrows A))
                (< j (nrows A)))
           (equal (multiply-matrix C
                   (saxpy-row
                    (get-identity-matrix B (nrows A))
                     j
                     i
                     alpha)
                   A)
                  (saxpy-row A j i alpha))))
```

Teorema 91. ***Clausura de la copia de matrices.***[9]

Sea $A \in \mathbb{Q}^{m\times n}$ *y* B *el resultado de* `(copy-matrix B A)`, *entonces* $B \in \mathbb{Q}$.

```
(defthm pfm-matrixp-copy-matrix
  (implies (matrixp A)
           (matrixp (copy-matrix B A))))
```

[9]Demostrado en el fichero del proyecto `gauss-jordan.lisp`

Teorema 92. ***Número de filas de la copia de matrices.***[10]

Sea $A \in \mathbb{Q}^{m \times n}$ y B el resultado de `(copy-matrix B A)`*, entonces $r(B) = m$.*

```
(defthm pfm-nrows-copy-matrix
  (implies (matrixp A)
           (equal (nrows (copy-matrix B A))
                  (nrows A))))
```

Teorema 93. ***Número de columnas de la copia de matrices.***[11]

Sea $A \in \mathbb{Q}^{m \times n}$ y B el resultado de `(copy-matrix B A)`*, entonces $c(B) = n$.*

```
(defthm pfm-ncolumns-copy-matrix
  (implies (matrixp A)
           (equal (ncolumns (copy-matrix B A))
                  (ncolumns A))))
```

Teorema 94. ***Elementos de la copia de matrices.***[12]

Sea $A \in \mathbb{Q}^{m \times n}$ y B el resultado de `(copy-matrix B A)`*, entonces $b_{ij} = a_{ij}$ con $0 \leq i < m$ y $0 \leq j < n$.*

```
(defthm pfm-lookup-copy-matrix
  (implies (and (matrixp A)
                (natp i)
                (natp j)
                (< i (nrows A))
                (< j (ncolumns A)))
           (equal (lookup (copy-matrix B A) i j)
                  (lookup A i j))))
```

Teorema 95. ***Equivalencia de la copia de matrices.***[13]

Sea $A \in \mathbb{Q}^{m \times n}$ y B el resultado de `(copy-matrix B A)`*, entonces $B = A$.*

```
(defthm pfm-equal-copy-matrix
  (implies (matrixp A)
           (equal (copy-matrix B A) A)))
```

4.9. Corolarios sobre las transformaciones por filas[14]

Corolario 4. ***La transformación F_{ij} y la operación de multiplicación entre matrices son intercambiables por la izquierda.***

Sea $A \in \mathbb{Q}^{m \times n}$ y $B \in \mathbb{Q}^{n \times p}$, entonces $F_{ij}B \cdot A = F_{ij}(BA)$ con $0 \leq i < m$ y $0 \leq j < m$.

[10] Demostrado en el fichero del proyecto `gauss-jordan.lisp`
[11] Demostrado en el fichero del proyecto `gauss-jordan.lisp`
[12] Demostrado en el fichero del proyecto `gauss-jordan.lisp`
[13] Demostrado en el fichero del proyecto `gauss-jordan.lisp`
[14] Demostrados en el fichero del proyecto `row-operations.lisp`

```
(defthm pfm-exchange-rows-multiply
  (implies (and (matrixp A)
                (matrixp B)
                (natp i)
                (natp j)
                (< i (nrows B))
                (< j (nrows B))
                (equal (ncolumns B) (nrows A)))
           (equal (multiply-matrix D (exchange-rows B i j) A)
                  (exchange-rows (multiply-matrix D B A) i j))))
```

Corolario 5. ***La transformación*** $F_i(\alpha)$ ***y la operación de multiplicación entre matrices son intercambiables por la izquierda.***

Sea $A \in \mathbb{Q}^{m\times n}$ *y* $B \in \mathbb{Q}^{n\times p}$*, entonces* $F_i(\alpha)B \cdot A = F_i(\alpha)(BA)$ *con* $0 \leq i < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-multiply-row-multiply
  (implies (and (matrixp A)
                (matrixp B)
                (rationalp alpha)
                (natp i)
                (< i (nrows B))
                (equal (ncolumns B) (nrows A)))
           (equal (multiply-matrix D (multiply-row B i alpha) A)
                  (multiply-row (multiply-matrix D B A)
                   i alpha))))
```

Corolario 6. ***La transformación*** $F_{ij}(\alpha)$ ***y la operación de multiplicación entre matrices son intercambiables por la izquierda.***

Sea $A \in \mathbb{Q}^{m\times n}$ *y* $B \in \mathbb{Q}^{n\times p}$*, entonces* $F_{ij}(\alpha)B \cdot A = F_{ij}(\alpha)(BA)$ *con* $0 \leq i < m$*,* $0 \leq j < m$ *y* $\alpha \in \mathbb{Q}$*.*

```
(defthm pfm-saxpy-row-multiply
  (implies (and (matrixp A)
                (matrixp B)
                (rationalp alpha)
                (natp i)
                (natp j)
                (< i (nrows B))
                (< j (nrows B))
                (equal (ncolumns B) (nrows A)))
           (equal (multiply-matrix D (saxpy-row B j i alpha) A)
                  (saxpy-row (multiply-matrix D B A)
                   j i alpha))))
```

4.10. Teoremas sobre el algoritmo de Gauss-Jordan[15]

Empecemos con la última tanda de teoremas demostrados en este Trabajo de Fin de Máster. El objetivo final es el de tratar de demostrar la corrección del algoritmo de Gauss-Jordan. Se supone que este algoritmo parte de la matriz concatenada $(I_n|A)$ y, aplicando sucesivas transformaciones por filas, se debe conseguir terminar con la matriz $(A^{-1}|I_n)$.

Hay que tratar de demostrar que si la parte *derecha* de la matriz concatenada que devuelve Gauss-Jordan es la matriz identidad (I_n), la parte *izquierda* de esta misma matriz es la inversa de la matriz original, esto es, $A^{-1}A = I_n$.

Para hacer esto hay que demostrar una serie de lemas y teoremas que desembocarán en este resultado final. Curiosamente empezamos con un par de lemas necesarios referentes a la aritmética de números racionales que ACL2 no trae por defecto:

Lema 12. ***La negación de una fracción de racionales es un racional.***

Sean $x, y \in \mathbb{Q}$, entonces $-x/y \in \mathbb{Q}$.

```
(defthm pfm-rationalp-negate-fraction
  (implies (and (rationalp x)
                (rationalp y))
           (rationalp (- (/ x y)))))
```

Lema 13. ***La negación de una fracción de racionales (expresada como producto de un racional por la inversa de otro) es un racional.***

Sean $x, y \in \mathbb{Q}$, entonces $-[(1/x) \cdot y] \in \mathbb{Q}$.

```
(defthm pfm-rationalp-inverse-fraction
  (implies (and (rationalp x)
                (rationalp y))
           (rationalp (- (* (/ x) y)))))
```

Recordemos que el algoritmo de Gauss-Jordan empieza inicializando una estructura de datos de dos matrices concatenadas y, posteriormente, aplica el algoritmo de reducción de columnas a dicha estructura. Para empezar hay que demostrar que, en el proceso de inicialización, se crean en el *stobj* `concat`, dos matrices cuya parte izquierda es el primer argumento de la función `initialize-concat` y que la parte derecha es el segundo argumento de esta misma función.

```
(defthm pfm-lemma-left-initialize-concat
  (implies (matrixp A)
           (equal (left (initialize-concat A B concat)) A)))

(defthm pfm-lemma-right-initialize-concat
  (implies (matrixp B)
           (equal (right (initialize-concat A B concat)) B)))
```

[15] Demostrados en el fichero del proyecto `gauss-jordan.lisp`

A continuación demostramos el teorema más importante (y, por cierto, más difícil de probar, por la gran cantidad de lemas necesarios) de este proyecto. Se trata de establecer que la aplicación del algoritmo de *reducción de columnas* visto en el capítulo anterior no altera las propiedades de las dos matrices que forman la concatenación en cuanto a la operación de la multiplicación entre matrices.

Teorema 96. ***El algoritmo de reducción de columnas mantiene las propiedades multiplicativas.***

Sean $A \in \mathbb{Q}^{n\times n}$ *y* $(I_0|D_0)$ *dos matrices adosadas por columnas (concatenación de dos matrices) y supongamos que se cumple* $I_0A = D_0$*. Sea* $(I_1|D_1)$ *el resultado de aplicar el algoritmo de reducción de columnas a* $(I_0|D_0)$*, es decir:*

$$(I_1|D_1) = reduce(I_0|D_0)$$

Entonces se mantiene que $I_1A = D_1$*.*

```
(defthm pfm-reduce-columns-preserves-multiply-matrix
  (implies (and (matrixp A)
                (square-matrixp A)
                (concatp concat)
                (natp m)
                (natp i)
                (natp turn)
                (equal (ncolumns-get-left concat) (nrows A))
                (< i (nrows-get-left concat))
                (< i (nrows-get-right concat))
                (equal (nrows-get-left concat)
                       (nrows-get-right concat))
                (equal (nrows-get-left concat)
                       (ncolumns-get-left concat))
                (< m (nrows A))
                (< m (nrows-get-left concat))
                (< m (nrows-get-right concat))
                (< m (ncolumns-get-left concat))
                (< m (ncolumns-get-right concat))
                (equal (multiply-matrix B (get-left B concat) A)
                       (get-right B concat)))
           (equal (multiply-matrix B (get-left B
                    (reduce-columns concat m turn i)) A)
                  (get-right B
                   (reduce-columns concat m turn i)))))
```

Con el siguiente teorema unimos las dos partes del algoritmo de Gauss-Jordan (la inicialización y la reducción de columnas). Hay que darse cuenta que la inicialización impone que la matriz *izquierda* sea la matriz identidad (I_n). Para ello tenemos este teorema:

Teorema 97. ***El algoritmo de reducción de columnas sobre la inicialización de Gauss-Jordan mantiene las propiedades multiplicativas.***

Sean $A \in \mathbb{Q}^{n\times n}$ *y* $(I_n|A)$ *dos matrices adosadas por columnas (concatenación de dos matrices). Sea* $(I'_n|A')$ *el resultado de aplicar el algoritmo de reducción de columnas a* $(I_n|A)$*, es decir:* $(I'_n|A') = reduce(I_n|A)$*. Entonces se tiene que* $I'_n \cdot A = A'$*.*

```
(defthm pfm-initialize-reduce-columns
  (implies (and (matrixp A)
                (square-matrixp A)
                (concatp concat)
                (equal Id (get-identity-matrix
                           (create-matrix) (nrows A))))
           (equal (multiply-matrix B
                   (get-left B
                    (reduce-columns
                     (initialize-concat Id A concat)
                     (1- (nrows A)) 2 (1- (nrows A))))
                   A)
                  (get-right B
                   (reduce-columns
                    (initialize-concat Id A concat)
                    (1- (nrows A)) 2 (1- (nrows A)))))))
```

Y, por último, fijémonos en la conclusión del anterior teorema:

$$I'_n \cdot A = A'$$

Si se logra que $A' = I_n$, entonces se tiene que $I'_n \cdot A = I_n$ y, por tanto $I'_n = A^{-1}$ para conseguir:

$$A^{-1} \cdot A = I_n$$

Teorema 98. ***El algoritmo de Gauss-Jordan es correcto.***

Sean $A \in \mathbb{Q}^{n \times n}$, la matriz a calcular su inversa y $(B|C)$ la matriz resultante de aplicar el algoritmo de Gauss-Jordan a la matriz A, entonces si $C = I_n$, $B \cdot A = I_n$ por lo que $B = A^{-1}$.

```
(defthm pfm-gauss-jordan-is-correct
  (implies (and (matrixp A)
                (square-matrixp A)
                (concatp concat)
                (equal (get-right B (gauss-jordan concat A))
                       (get-identity-matrix B (nrows A))))
           (equal (multiply-matrix B
                   (get-left B (gauss-jordan concat A)) A)
                  (get-identity-matrix B (nrows A)))))
```

5. Mejora del tiempo de ejecución: uso de *stobj's abstractos* en ACL2

Resumiendo lo que hemos visto hasta ahora podemos decir que hemos conseguido, por un lado, formalizar los conceptos básicos del álgebra lineal en ACL2. Por otro lado hemos usado dicha formalización para la demostración y verificación formal de propiedades y teoremas que se derivan de las definiciones de funciones y operaciones asociadas a las matrices. Proceso que ha terminado con la demostración de que el algoritmo de Gauss-Jordan es correcto en el sentido de que devuelve, bajo ciertas condiciones, la matriz inversa de una dada. Todas las declaraciones, definiciones y demostraciones correspondientes a esta sección se encuentran en el fichero del proyecto `abstract-stobj.lisp`.

La formalización ha consistido en idear una estructura de datos que dé soporte a las matrices bidimensionales en ACL2. Se ha utilizado quizás la estructura más intuitiva (una lista de listas) y una serie de condiciones e invariantes para verificar que, en efecto, estamos ante una matriz.

A partir de esta formalización definimos una serie de primitivas con las que poder operar con matrices. Las dos primitivas más importantes son `lookup` y `update`, ya que, a partir de ellas, se definen el resto de operaciones sobre matrices que se han estudiado a lo largo de este trabajo.

Pensemos ahora que ahora queremos centrarnos en la parte de ejecución de estas funciones, es decir, queremos una ejecución *eficiente* de los algoritmos involucrados para poder tratar con matrices arbitrariamente grandes (del orden de centenares de filas y columnas) sin penalizar en exceso el tiempo de ejecución.

Por tanto, una óptima implementación de las primitivas básicas (`lookup` y `update`) será necesaria para conseguir este objetivo. No es tan importante optimizar el redimensionamiento de matrices (con la primitiva `redim`) ya que no va a ser una primitiva a usar repetidamente en llamadas recursivas de una función (léase, en el cuerpo de bucles iterativos). Esta primitiva se usa sobre todo en la inicialización de ciertos algoritmos.

Si nos fijamos en la definición de las primitivas `lookup` y `update` se puede comprobar que están basadas en unas funciones más simples de ACL2, llamadas `nth` y `update-nth`. ¿Cómo están implementadas estas funciones? Por ejemplo, `nth` está declarada de forma recursiva como:

```
1 (defun nth (n l)
2   (and (consp l)
3        (if (zp n)
4            (car l)
5            (nth (1- n) (cdr l)))))
```

Que podría expresarse en palabras como sigue:

1. Si $n = 0$ devolvemos el primer elemento de la lista.
2. Si $n > 0$ devolvemos el elemento $(n-1)$-ésimo del *resto* de la lista.

Se puede comprobar fácilmente que el orden de complejidad de este algoritmo es $\in \mathcal{O}(n)$. Es decir, el tiempo de ejecución varía de forma `lineal` respecto al tamaño de la lista en el caso peor. Esto significa que si una lista tiene diez veces más elementos que otra, se va a tardar diez veces más, en media, en ejecutar la función `nth`. La consecuencia de esto es que cuanto más grande es nuestra matriz (que es cuando más útil resulta, es decir, con mucha información) más van a tardar los algoritmos definidos en ejecutarse, no sólo como consecuencia de que la matriz sea grande sino de que sus primitivas son poco eficientes.

La definición de la función `nth` y su implementación coinciden en este caso ya que así es como está definida en Common Lisp. La función `update-nth` se define e implementa de forma similar y también presenta un orden de complejidad temporal $\in \mathcal{O}(n)$.

Esto entra en contraposición con uno de los requisitos para decidir sobre la estructura de datos a utilizar. Si queremos mantener las primitivas `lookup` y `update` en orden constante ($\in \mathcal{O}(1)$) se hace necesario el uso de *arrays* o *vectores*. Esto asegura que, independientemente del tamaño de las listas o matrices, el acceso a un elemento cualquiera tarda siempre lo mismo.

Recuérdese que un vector o array es una estructura de datos, consecutivos en memoria y del mismo tamaño, de forma que se puede acceder a cualquiera de ellos a través de un `índice` de tipo entero, a través del cuál, mediante operaciones aritméticas de suma y desplazamiento a nivel de código máquina, podemos calcular muy rápidamente la dirección efectiva del dato en cuestión.

5.1. Uso de los *stobj's* para mejorar el tiempo de ejecución

Curiosamente, el uso de `stobj's` está especialmente indicado para mejorar el tiempo de ejecución en ACL2 ya que asegura una implementación eficiente si el tipo de datos escogido en los campos de la estructura es de tipo `array`. Ya se comentó en la sección dedicada a la definición del algoritmo de Gauss-Jordan las funciones de acceso y modificación a los elementos individuales del tipo vector.

Por tanto, la pregunta ahora es: ¿Cómo diseñar un stobj para dar soporte a una matriz? Probemos con la siguiente definición:

```
1 (defstobj matrix$c
2   (m$c          :type (array rational (1))
3                 :initially 0
4                 :resizable t)
5   (nrows$c      :type (integer 1 *)
6                 :initially 1)
7   (ncolumns$c   :type (integer 1 *)
8                 :initially 1))
```

El sufijo `...$c` en los nombres de los campos y en el propio nombre del objeto viene de "concreto", la explicación a esto se dará más adelante. Comentemos ahora cada uno de los campos de esta estructura:

- El campo `nrows$c` nos almacena el número de filas de la matriz. Debe ser un entero cuyo menor valor sea 1 y no tenga por qué tener límite superior (representado por `1 *` en la definición anterior).
- El campo `ncolumns$c` nos almacena el número de columnas de la matriz. Debe ser un entero cuyo menor valor sea 1 y no tenga por qué tener límite superior.
- El campo `m$c` es de tipo vector de racionales de tamaño 1, de forma que inicialmente ese único elemento vale 0 y con posibilidad de redimensionar el vector.

Lo primero que llama la atención es el uso de un vector *unidimensional* para almacenar los elementos de la matriz (una estructura evidentemente bidimensional). La razón es que, en la versión actual de ACL2, no se permiten vectores bidimensionales como campos de un stobj. Esto en principio no es tan difícil de conseguir en otros lenguajes de programación de alto nivel. En C, por ejemplo, podemos declarar un vector bidimensional de M filas y N columnas de la siguiente forma:

```
int m[M][N];
```

Se asegura, en estos lenguajes que el tiempo de acceso al elemento `a[i][j]` es prácticamente el mismo que al acceder a un elemento de un vector unidimensional, y, por contra, se gana mucho en legibilidad y facilidad de diseño de algoritmos al acceder directamente al elemento de cierta fila i y cierta columna j.

Sin embargo, en ACL2 tenemos que adaptarnos al uso, mucho menos intuitivo, de un vector unidimensional. Para ello, vamos a hacer la correspondencia mediante un ejemplo. Dada la matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

Se representaría de la siguiente forma usando un vector unidimensional, nótese que ahora, la notación `(...)` representa un vector y no una lista.

```
B = (1 2 3 4 5 6 7 8 9 10 11 12)
```

Por tanto cabe preguntarse ahora por la relación existente entre el elemento a_{ij} de la matriz A y el elemento correspondiente del vector B. Se puede ver fácilmente que el elemento a_{ij} se mapea así en el vector B:

$$a_{ij} \mapsto b_{i \cdot c(A)+j}$$

El mapeo inverso es menos intuitivo, pero igualmente se comprueba:

$$b_k \mapsto a_{\lfloor k/r(A) \rfloor, k\,\%r(A)}$$

Donde el operador $\%$ es el operador módulo o resto de la división entera.

Ahora hay que implementar las mismas primitivas que definimos con la lista de listas, pero ahora con este nuevo *stobj*. Empezamos con el reconocedor de las matrices, donde introducimos un invariante que no se ha podido imponer en la declaración de la estructura: La longitud del vector debe ser el resultado de multiplicar los campos que representan filas y columnas:

```
(defun matrix$cp+ (matrix$c)
  (declare (xargs :stobjs matrix$c))
  (and (matrix$cp matrix$c)
       (equal (m$c-length matrix$c)
              (* (nrows$c matrix$c) (ncolumns$c matrix$c)))))
```

Nótese el uso de la función `(matrix$cp)` generada automáticamente al declarar el *stobj* y de la declaración del argumento como de tipo *stobj*. También generadas automáticamente tenemos las primitivas `nrows$c` y `ncolumns$c` para el acceso al número de filas y columnas de la matriz. Las observadoras `lookup$c` y `update$c` deben hacer uso del mapeo anteriormente definido:

```
(defun lookup$c (matrix$c i j)
  (declare (xargs :stobjs matrix$c))
  (m$ci (+ (* i (ncolumns$c matrix$c)) j) matrix$c))

(defun update$c (matrix$c i j v)
  (declare (xargs :stobjs matrix$c))
  (update-m$ci (+ (* i (ncolumns$c matrix$c)) j) v matrix$c))
```

La última primitiva, la de redimensionamiento, requiere de una función auxiliar que obligue a todos los elementos de la matriz a resetearse al valor 0. Esto es debido a que la función `resize-m$c`, generada automáticamente, mantiene el valor de los elementos que quepan en el vector `m$c`.

```
(defun reset-matrix$c (matrix$c n)
  (declare (xargs :stobjs matrix$c))
  (if (zp n)
    matrix$c
    (seq matrix$c
         (update-m$ci (1- n) 0 matrix$c)
         (reset-matrix$c matrix$c (1- n)))))

(defun redim$c (matrix$c nrows ncolumns)
  (declare (xargs :stobjs matrix$c))
  (seq matrix$c
       (update-nrows$c nrows matrix$c)
       (update-ncolumns$c ncolumns matrix$c)
       (resize-m$c (* nrows ncolumns) matrix$c)
       (reset-matrix$c matrix$c (* nrows ncolumns))))
```

5.2. Los objetos de hebra simple *abstractos* en ACL2

Con el *stobj* de la sección anterior conseguimos crear una posible formalización de matrices que asegura la eficiencia en la ejecución de las primitivas definidas. El problema ahora es doble:

- No cumplimos con el requisito de estructura de datos simple e intuitiva.
- Todo el trabajo realizado al definir todas las funciones sobre matrices y demostrar todos los lemas, teoremas y corolarios hay que revisarlo desde el principio al tener implementadas de forma radicalmente diferente todas las primitivas básicas de acceso y modificación de matrices.

Pues bien, existe un mecanismo en ACL2 que permite aprovechar todo el trabajo desarrollado hasta ahora con sólo un pequeño esfuerzo adicional de demostraciones adicionales. Se trata de los *stob'j abstractos* u objetos abstractos de hebra simple.

Estos objetos se crean mediante el evento `defabsstobj` y proporcionan una forma de dar definiciones alternativas para las primitivas asociadas a un *stobj*. Estas primitivas pueden consistir en un reconocedor, un constructor y cualquier otro método que queramos *exportar* y que esté relacionado con la estructura de datos declarada en el *stobj*. Por lo tanto, se establece una interfaz de acceso al *stobj* aunque, y aquí viene lo importante, la implementación concreta de cada primitiva sea radicalmente diferente a como se definen en un principio.

El evento `defabsstobj` introduce un nuevo *stobj* en el sistema, que llamaremos *stobj abstracto*, el cual está asociado con un *stobj concreto* definido previamente de la forma habitual con `defstobj`. Con la declaración del *stobj abstracto* se especifican dos definiciones separadas para cada primitiva que queramos exportar. Una de las definiciones (`:LOGIC`) servirá para la parte de la lógica y demostraciones en ACL2 y la otra (`:EXEC`) cuando estemos evaluando y ejecutando dichas primitivas.

Hay varias razones para querer hacer esto pero el uso que le estamos dando dentro del contexto de este trabajo es el de poder tener una definición lógica *muy adaptada* a conseguir demostraciones de forma simple y rápida y, por otro lado, tener una definición en ejecución muy eficiente desde el punto de vista del acceso a memoria.

En vez de fijarnos en un ejemplo cualquiera para explicar el evento `defabsstobj` vamos a hacer el trabajo directamente sobre el *stobj* usado en este trabajo:

```
1  (defabsstobj matrix
2    :concrete matrix$c
3    :recognizer (matrixp :logic matrix$ap :exec matrix$cp)
4    :creator (create-matrix
5              :logic create-matrix$a
6              :exec create-matrix$c
7              :correspondence create-matrix{correspondence}
8              :preserved create-matrix{preserved})
9    :corr-fn matrix$corr
10   :exports ((nrows              :logic nrows$a
11                                 :exec  nrows$c)
12             (ncolumns           :logic ncolumns$a
13                                 :exec  ncolumns$c)
14             (lookup             :logic lookup$a
15                                 :exec  lookup$c)
16             (update             :logic update$a
17                                 :exec  update$c)
18             (redim              :logic redim$a
19                                 :exec  redim$c
20                                 :protect t)))
```

Pasemos a explicar cada campo más detalladamente:

- `:concrete` nos dice cuál es el *stobj* concreto correspondiente al *stobj* abstracto declarado.
- `:recognizer` nos dice el nombre que tendra el reconocedor del *stobj* abstracto definido. Para ello hay que dar su definición en la lógica (`:LOGIC`) y en ejecución (`:EXEC`).
- `:creator` nos da el constructor del *stobj* abstracto. Los campos `:correspondence` y `:preserved` sirven para establecer *las pruebas obligatorias* y se verán más adelante.
- `:corr-fn` nos da el nombre de la función que sirve para establecer las correspondencias entre el stobj concreto y abstracto.
- `:exports` nos da la lista de primitivas exportadas por el stobj abstracto, dando como siempre, su definición en la lógica y en ejecución.

Volvemos ahora a las pruebas obligatorias necesarias para la admisión del evento anterior. Las hay de tres tipos: `:correspondence`, `:preserved` y `:guard-thm`. Las primeras de ellas, pruebas de correspondencia, sirven para garantizar que las evaluaciones en la lógica están de acuerdo con la del stobj concreto. Para ello hay dos tipos de correspondencias.

Primero tenemos las correspondencias de aquellas funciones observadoras, es decir, las que no devuelven el mismo stobj. En este caso la correspondencia trata de demostrar que lo que se devuelve en la lógica es igual (en el sentido de la función `equal` de ACL2) a lo que devuelve la función definida en ejecución.

Las correspondencias de las funciones modificadoras, aquellas que modifican (y por lo tanto, devuelven) el stobj en cuestión, hacen esencialmente lo mismo pero gracias a una función `matrix$corr` que verifica si el stobj concreto y el stobj abstracto pasados como argumento se corresponden, es decir, representan o modelan la misma estructura de datos. Recuérdese que `equal` no se puede aplicar a stobj's.

Los lemas de tipo `:preserved` tratan de garantizar que las funciones modificadoras preservan su reconocedor, es decir, siguen siendo del mismo tipo que el original.

Por último las pruebas de tipo `:guard-thm` sirven para garantizar que la guarda se cumple para cada llamada a las funciones definidas en ejecución.

Pero en nuestro ejemplo, ¿quién debe actuar como stobj abstracto? Debe ser nuestra estructura *lista de listas*, con lo que sólo quedaría renombrar los nombres de las funciones primitivas para que tengan como sufijo `...$a` (que viene de stobj *asbtracto*):

```
1  (defun matrix$ap-aux (n x)
2    (if (endp x)
3      t
4      (and (rational-listp (car x))
5           (equal (len (car x)) n)
6           (matrix$ap-aux n (cdr x)))))
7
8  (defun matrix$ap (x)
9    (and (true-listp x)
10        (<= 1 (len x))
11        (<= 1 (len (car x)))
12        (matrix$ap-aux (len (car x)) x)))
13
14 (defun create-matrix$a ()
15   '((0)))
16
17 (defun nrows$a (matrix$a)
18   (len matrix$a))
19
20
21 (defun ncolumns$a (matrix$a)
22   (len (car matrix$a)))
23
24 (defun lookup$a (matrix$a i j)
25   (nth j (nth i matrix$a)))
26
27 (defun update$a (matrix$a i j v)
28   (update-nth i (update-nth j v (nth i matrix$a)) matrix$a))
29
30 (defun redim$a-rec (nrows ncolumns)
31   (if (zp nrows)
32     nil
33     (cons (make-list ncolumns :initial-element 0)
34           (redim$a-rec (1- nrows) ncolumns))))
35
```

```
(defun redim$a (matrix$a nrows ncolumns)
  (declare (ignore matrix$a))
  (redim$a-rec nrows ncolumns))
```

Por otro lado la función `matrix$corr` se ha definido buscando la igualdad entre todos los elementos que forman la matriz hecha mediante lista de listas y mediante el stobj concreto basado en vectores:

```
(defun-nx equal-row (Ma Mc m n)
  (if (zp n)
    (equal (lookup$a Ma m 0)
           (lookup$c Mc m 0))
    (and (equal (lookup$a Ma m n)
                (lookup$c Mc m n))
         (equal-row Ma Mc m (- n 1)))))

(defun-nx equal-rows (Ma Mc m n)
  (if (zp m)
    (equal-row Ma Mc 0 n)
    (and (equal-row Ma Mc m n)
         (equal-rows Ma Mc (- m 1) n))))

(defun-nx matrix$corr (matrix$c matrix$a)
  (and (matrix$cp+ matrix$c)
       (matrix$ap matrix$a)
       (equal (nrows$c matrix$c)
              (nrows$a matrix$a))
       (equal (ncolumns$c matrix$c)
              (ncolumns$a matrix$a))
       (equal-rows matrix$a
                   matrix$c
                   (1- (nrows$c matrix$c))
                   (1- (ncolumns$c matrix$c)))))
```

Como esta función sólo se necesita para establecer correspondencias lógicas y demostrar propiedades no hace falta que sea una función *ejecutable*, de ahí la definición mediante `defun-nx`.

Una vez definidos los dos stobjs podemos intentar el evento `defabsstobj`. En principio nos faltarían demostrar las pruebas obligatorias que automáticamente genera el script de salida de este evento:

```
(DEFTHM CREATE-MATRIX{CORRESPONDENCE}
        (MATRIX$CORR (CREATE-MATRIX$C)
                     (CREATE-MATRIX$A))
        :RULE-CLASSES NIL)

(DEFTHM CREATE-MATRIX{PRESERVED}
        (MATRIX$AP (CREATE-MATRIX$A))
        :RULE-CLASSES NIL)

(DEFTHM NROWS{CORRESPONDENCE}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX))
                 (EQUAL (NROWS$C MATRIX$C)
                        (NROWS$A MATRIX)))
```

```
        :RULE-CLASSES NIL)

(DEFTHM NCOLUMNS{CORRESPONDENCE}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX))
                 (EQUAL (NCOLUMNS$C MATRIX$C)
                        (NCOLUMNS$A MATRIX)))
        :RULE-CLASSES NIL)

(DEFTHM LOOKUP{CORRESPONDENCE}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX)
                      (NATP I)
                      (NATP J)
                      (< I (NROWS$A MATRIX))
                      (< J (NCOLUMNS$A MATRIX)))
                 (EQUAL (LOOKUP$C MATRIX$C I J)
                        (LOOKUP$A MATRIX I J)))
        :RULE-CLASSES NIL)

(DEFTHM LOOKUP{GUARD-THM}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX)
                      (NATP I)
                      (NATP J)
                      (< I (NROWS$A MATRIX))
                      (< J (NCOLUMNS$A MATRIX)))
                 (AND (NATP I)
                      (NATP J)
                      (< (+ (* I (NCOLUMNS$C MATRIX$C)) J)
                         (M$C-LENGTH MATRIX$C))))
        :RULE-CLASSES NIL)

(DEFTHM UPDATE{CORRESPONDENCE}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX)
                      (NATP I)
                      (NATP J)
                      (RATIONALP V)
                      (< I (NROWS$A MATRIX))
                      (< J (NCOLUMNS$A MATRIX)))
                 (MATRIX$CORR (UPDATE$C MATRIX$C I J V)
                              (UPDATE$A MATRIX I J V)))
        :RULE-CLASSES NIL)

(DEFTHM UPDATE{GUARD-THM}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (MATRIX$AP MATRIX)
                      (NATP I)
                      (NATP J)
                      (RATIONALP V)
                      (< I (NROWS$A MATRIX))
                      (< J (NCOLUMNS$A MATRIX)))
```

```
                  (AND (NATP I)
                       (NATP J)
                       (RATIONALP V)
                       (< (+ (* I (NCOLUMNS$C MATRIX$C)) J)
                          (M$C-LENGTH MATRIX$C))))
        :RULE-CLASSES NIL)

(DEFTHM UPDATE{PRESERVED}
        (IMPLIES (AND (MATRIX$AP MATRIX)
                      (NATP I)
                      (NATP J)
                      (RATIONALP V)
                      (< I (NROWS$A MATRIX))
                      (< J (NCOLUMNS$A MATRIX)))
                 (MATRIX$AP (UPDATE$A MATRIX I J V)))
        :RULE-CLASSES NIL)

(DEFTHM REDIM{CORRESPONDENCE}
        (IMPLIES (AND (MATRIX$CORR MATRIX$C MATRIX)
                      (NATP NROWS)
                      (NATP NCOLUMNS)
                      (<= 1 NROWS)
                      (<= 1 NCOLUMNS))
                 (MATRIX$CORR (REDIM$C MATRIX$C NROWS NCOLUMNS)
                              (REDIM$A MATRIX NROWS NCOLUMNS)))
        :RULE-CLASSES NIL)

(DEFTHM REDIM{PRESERVED}
        (IMPLIES (AND (MATRIX$AP MATRIX)
                      (NATP NROWS)
                      (NATP NCOLUMNS)
                      (<= 1 NROWS)
                      (<= 1 NCOLUMNS))
                 (MATRIX$AP (REDIM$A MATRIX NROWS NCOLUMNS)))
        :RULE-CLASSES NIL)
```

Dos cosas a comentar de la salida de este script. En primer lugar se ven algunas correspondencias de tipo `:guard-thm`. En principio las guardas no se han tenido en cuenta en este documento ya que no aportan gran cosa al desarrollo de los objetivos ni a su explicación en esta memoria. En la práctica, si queremos asegurar que las funciones definidas en ACL2 son compatibles con el estándar Common LISP, es necesario verificar todas las guardas de las funciones definidas.

En segundo lugar, todas las pruebas obligatorias no generan ninguna regla para su posterior uso en el demostrador (`:RULE-CLASSES NIL`) ya que en realidad establecen unas correspondencias que en teoría sólo tienen utilidad para la admisión del stobj abstracto. Por cierto que todos esos teoremas se han tenido que demostrar para la realización de este trabajo (en el fichero `abstract-stobj.lisp`).

Debido a que podemos tener varias matrices diferentes a la vez en memoria habría que definir todas y cada una de ellas de forma separada ya que no se admite más de una instancia de cada stobj. Por ejemplo, podemos crear así el stobj abstracto `A`.

```
(defabsstobj A
  :concrete matrix$c
  :recognizer (matrixp&a :logic matrix$ap :exec matrix$cp)
  :creator (create-matrix&a
            :logic create-matrix$a :exec create-matrix$c
            :correspondence create-matrix{correspondence}
            :preserved create-matrix{preserved})
  :corr-fn matrix$corr

  :exports ((nrows&a              :logic nrows$a
                                  :exec  nrows$c)
            (ncolumns&a           :logic ncolumns$a
                                  :exec  ncolumns$c)
            (lookup&a             :logic lookup$a
                                  :exec  lookup$c)
            (update&a             :logic update$a
                                  :exec  update$c)
            (redim&a              :logic redim$a
                                  :exec  redim$c
                                  :protect t))
  :congruent-to matrix)
```

Sin embargo, dichos stobjs, que en realidad constan de los mismos campos, funciones y correspondencias con el stobj concreto `matrix$c` deberían poder intercambiarse entre sí en los parámetros de aquellas funciones que requieran un stobj de tipo `matrix`. Recuérdese las fuertes restricciones sintácticas que se requieren para el uso en general de los stobjs.

Para relajar en cierta medida esta restricción se dispone, en ACL2, de los stobjs congruentes entre sí (`:congruent-to matrix`). Dos stobjs pueden ser congruentes entre sí en caso de tener la misma estructura en los campos que lo componen. Cuando dos stobjs están declarados como congruentes entre sí, ACL2 permite sustituir uno por el otro y viceveresa en una llamada a una función.

5.3. Cambios en la definición de funciones debido al uso del *stobj* abstracto

Después de este último esfuerzo de demostraciones para las correspondencias de los stobjs abstractos hemos conseguido una herramienta, potente computacionalmente hablando y que a la vez nos permite demostrar con cierta soltura propiedades y teoremas. Las demostraciones no deben cambiar en prácticamente nada pero las definiciones de las funciones que antes usaban listas de listas deben cambiar algo su definición para, al menos, declarar uno de sus argumentos de esta forma: `(declare (xargs :stobjs matrix))`.

Una función que sí hay que replantearse es la igualdad entre matrices. Como no podemos, en ejecución, llamar a `equal` pasándole como argumento un stobj, se hace necesario hacer una doble definición de esta función: la que se ejecuta, sujeta a una guarda y la lógica, exenta de ella, y con la posibilidad de esa llamada a `equal` que nos va a permitir usar `equal-matrix` como una igualdad general entre cualquier tipo de objetos en el mundo de ACL2. Para conseguir esta doble definición se hace uso de `defexec` y `mbe`:

```
(defun-nx equal-matrix-nx (A B)
  (if (and (matrixp A)
           (matrixp B)
           (equidimensionalp A B))
    (equal-rows A B (1- (nrows A)) (1- (ncolumns B)))
```

```
    (equal A B)))

(defexec equal-matrix (A B)
  (declare (xargs :stobjs (A B)
                  :guard (and (matrixp A)
                              (matrixp B))))
  (mbe :logic
       (equal-matrix-nx A B)
       :exec
       (if (equidimensionalp A B)
         (equal-rows A B (1- (nrows A)) (1- (ncolumns B)))
         nil)))
```

Por otro lado, cuando llegamos a la definición del algoritmo de Gauss-Jordan, se utiliza un stobj concreto, concat, para representar las dos matrices concatenadas de las que hace uso el algoritmo. La propia definición de este stobj debe cambiar ya que ahora debe almacenar dos estructuras que, a su vez, son stobj's:

```
(defstobj concat
  (left :type matrix)
  (right :type matrix)
  (determinant :type rational :initially 1))
```

El problema es que hacer uso de un stobj que contiene otros stobjs debe hacerse con `stobj-let`, una función en ACL2 que permite el acceso y modificación de los campos de tipo stobj. Por ello, habría que cambiar todas las funciones que usan el stobj `concat`, entre otras, el algoritmo de reducción de columnas e `initialize-concat` (que incluimos como ejemplo de uso de `stobj-let`). De todas formas estos cambios son mínimos y fáciles de hacer una vez que se entiende el funcionamiento de lo stobjs en ACL2.

```
(defun initialize-concat (C D concat)
  (declare (xargs :stobjs (concat C D)
                  :guard (and (matrixp C)
                              (matrixp D))))
  (seq concat
       (stobj-let
        ((A (left concat))
         (B (right concat)))
        (A B)
        (let*
          ((A (copy-matrix A C))
           (B (copy-matrix B D)))
          (mv A B))
        concat)
       (update-determinant 1 concat)))
```

Y, ya por último, un aspecto también útil y a tener en cuenta son los stobj's locales, es decir, usar de forma temporal y en la llamada actual de una función, un stobj concreto creando una copia local de dicho stobj y, por tanto, sin necesidad de actuar sobre el objeto que está declarado como global en ACL2. Para ello se usa la función `with-local-stobj`. Para ilustrar este hecho introucimos un par de funciones más que llamen al algoritmo de Gauss-Jordan usando para ello una copia local del stobj `concat`. Con ello podemos centrarnos en la parte que nos interesa del algoritmo, que puede ser o bien, la matriz inversa de la matriz A (parte izquierda de `concat`) o el determinante de `concat`.

```
(defun inverse (A)
  (declare (xargs :stobjs A
                  :guard (and (matrixp A)
                              (square-matrixp A))))
  (with-local-stobj
   concat
   (mv-let (A concat)
           (let ((concat (gauss-jordan concat A)))
             (seq A
                  (get-left A concat)
                  (mv A concat)))
           A)))

(defun get-determinant (A)
  (declare (xargs :stobjs A
                  :guard (and (matrixp A)
                              (square-matrixp A))))
  (with-local-stobj
   concat
   (mv-let (val concat)
           (let ((concat (gauss-jordan concat A)))
             (mv (determinant concat) concat))
           val)))
```

Esto hace innecesario tener que pasar la referencia al objeto `concat` global cuando realmente con la copia local nos basta. Por último habría que justificar el uso de este stobj abstracto siempre que se consiga reducir el tiempo de ejecución respecto de la implementación de lista de listas. Para un estudio de los tiempos de ejecución más detallado véase el anexo B.

6. Conclusiones y posibles expansiones del trabajo

En este Trabajo de Fin del Máster de Lógica, Computación e Inteligencia Artificial se ha conseguido formalizar los conceptos básicos del álgebra lineal en el sistema ACL2 para lograr los siguientes objetivos:

- Tener un marco para poder hacer verificación formal de algoritmos sobre matrices.
- Tener una implementación en Common Lisp de dichos algoritmos con tiempos de ejecución aceptables.

Las conclusiones finales después del desarrollo de este trabajo han sido las siguientes:

- ACL2 es un excelente sistema de demostración automática *sobre algoritmos*, es decir, funciona de forma excelente cuando se quiere razonar sobre recursiones gracias a su potente principio de inducción.
- El usuario del sistema es capaz de controlar hasta el último detalle de estas demostraciones tanto por el lado de poder dar las correspondientes pistas y consejos al sistema como por proporcionar los lemas necesarios para que el demostrador complete el trabajo.
- Por contra, la curva de aprendizaje de ACL2 puede ser frustrante al principio, por lo que se recomienda por experiencia previa, hacer cierta demostraciones más difíciles a mano para saber de antemano los posibles problemas que pueda tener el sistema al intentar la demostración.
- ACL2 no olvida la eficiencia en la ejecución de los algoritmos escritos en su lenguaje. Como es totalmente compatible con el estándar Common Lisp, permite el uso de vectores directamente en memoria. Además, se permite el proceso de compilación, lo que proporciona otra ventaja sobre los lenguajes interpretados.
- Se ha conseguido una librería sobre matrices muy intuitiva (gracias al uso de un conjunto muy reducido de primitivas) que no existía con anterioridad. El único trabajo serio que estaba escrito con anterioridad a este es el de Gamboa, Cowles y Van Baalen (2003) pero adolece de usar primitivas poco consistentes, eficiencia temporal limitada y excesivas llamadas a sistema operativo para la reserva dinámica de memoria (ver anexo B sobre este trabajo anterior).
- Además se ha conseguido, gracias a los stobj's de ACL2, usar pocas llamadas a sistema para reserva dinámica de memoria y un excelente comportamiento en tiempo de ejecución frente a una implementación más clásica basada en listas de listas.

Un proyecto de estas características siempre admite alguna continuación o expansión. Por ejemplo, una vez que hemos llegado a formalizar y demostrar la corrección del algoritmo de Gauss-Jordan se nos antoja asequible seguir con las siguientes líneas (también relacionadas con el álgebra lineal):

- **Propiedades de los determinantes.** Muy utilizados en multitud de aplicaciones científicas y de ingeniería.
- **Resolución de sistemas de ecuaciones lineales.** Es otro de las aplicaciones del algoritmo de reducción de columnas visto en este trabajo. Debería ser razonablemente rápido modificarlo para poder resolver sistemas de ecuaciones lineales de tamaño arbitrario.
- **Espacios vectoriales.** También fácilmente ampliable por esta ruta ya que los vectores no son sino casos particulares de matrices.
- **Variedades lineales y ortogonalidad.**

- **Autovalores y autovectores.**
- **Matrices de adyacencia en grafos.** En realidad una manera de sobras conocida de manipular grafos de forma computacional es gracias a una matriz bidimensional de *adyacencia*. Los índices de filas representan *vértices origen* y los índices por columnas representan *vértices destino*. Por tanto, el elemento (i, j) de la matriz representará el *peso* de la arista dirigida con origen en el vértice i y destino en el j. Si cambiamos las primitivas por otras cuya sintaxis y semántica se refieran a *grafos* en vez de a matrices tendremos una posible formalización de los grafos en ACL2, lo que abriría un abanico inmensamente grande de posibles verificaciones formales en algoritmos sobre grafos.

Por último, comentar que este Trabajo de Fin de Máster se encuadra perefectamente en el temario y plan de estudios del Máster en Lógica, Computación e Inteligencia Artificial, impartido por el departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla.

En concreto, en la parte de lógica del Máster, la asignatura que da soporte y base a todo este trabajo es la de Razonamiento Asistido por Computador, impartida por D. José Luis Ruiz Reina, miembro del departamento anterior. Este proyecto ha sido propuesto y tutorizado en su totalidad por D. José Luis Ruiz Reina y D. Francisco Jesús Martín Mateos, también miembro del citado departamento.

A nivel personal este Trabajo Fin de Máster supone la culminación de los estudios de este Máster. Mi titulación es la de Ingeniero en Telecomunicación, y no he cursado ningún curso superior en matemáticas y mucho menos en la parte de la Lógica Matemática. Se puede decir sin dudar que hace escasamente seis meses no tenía claro que significaba eso de la regla del *modus ponens*. Todo lo que aprendido durante este tiempo ha sido a la vez nuevo y excitante hasta extremos difíciles de prever (las noches dedicadas a la demostración de algunos teoremas dan fe de ello).

Q.E.D.

7. Anexo A: Lemas usados en las demostraciones

El código fuente del proyecto se ha estructurado en una serie de ficheros con extensión `.lisp`. Con el nombre del mismo se pretende resumir el contenido. Por ejemplo, el fichero `abstract-stobj.lisp` tiene las declaraciones del stobj concreto y el abstracto usados para la modelización de las matrices en ACL2, pero también contiene una serie de lemas básicos sobre las primitivas definidas.

Cada fichero representa, además, un libro que se puede certificar automáticamente de principio a fin con `certify-book`. Las dependencias entre ficheros son fáciles de entender ya que se ha hecho de forma progresiva. Así, en la siguiente lista, el fichero i depende de los $i-1$ anteriores.

1. `abstract-stobj.lisp`
2. `matrix-defuns.lisp`
3. `equal-matrix-equal.lisp`
4. `defun-thms.lisp`
5. `unary-ops-thms.lisp`
6. `add-matrix-thms.lisp`
7. `multiply-matrix-thms.lisp`
8. `row-operations.lisp`
9. `gauss-jordan.lisp`
10. `time-metrics.lisp`

La única dependencia externa de todo el proyecto es al último libro de lemas y propiedades de la aritmética instalado por defecto en la distribución de ACL2: `arithmetic-5/top`.

La versión de ACL2 usada para la realización de este trabajo es la 6.3 con fecha de compilación de 1 de Octubre de 2013. El entorno de desarrollo ha sido ACL2s (ACL2 Sedan) versión 1.1.3 bajo sistema operativo Windows.

Para cada fichero se dará información sobre el número de líneas, eventos que lo componen, número de definiciones de funciones, de lemas locales y de teoremas del mismo así como la lista de lemas y teoremas demostrados en cada uno de ellos.

7.1. Análisis de cada fichero del proyecto

`abstract-stobj.lisp`

- **Número de líneas:** 1171
- **Número de eventos:** 119
- **Número de funciones:** 18
- **Número de lemas locales:** 78

- **Número de teoremas globales:** 14

Lista de lemas locales:

```
pfm-elements-true-listp-01
create-matrix{correspondence}
create-matrix{preserved}
nrows{correspondence}
ncolumns{correspondence}
pfm-elements-equal-row-02
pfm-equal-rows-equal-row-03
pfm-elements-equal-rows-04
lookup{correspondence}
pfm-linear-ij-property-05
lookup{guard-thm}
pfm-mcp-update-06
pfm-matrixcp-update-07
pfm-true-listp-updatea-08
pfm-len-update-nth-09
pfm-len-car-updatea-10
pfm-len-updatea-11
pfm-rational-listp-update-nth-12
pfm-matrixap-aux-updatea-13
pfm-matrixtap-updatea-14
pfm-nrows-updatec-15
pfm-ncolumns-updatec-16
pfm-elements-updatea-17
pfm-inequality-property-18
pfm-inequality-product-property-19
pfm-inequality-product-property-20
pfm-elements-updatec-21
pfm-elements-updatea-updatec-22
pfm-equal-row-updatea-updatec-23
pfm-equal-rows-updatea-updatec-24
update{correspondence}
update{guard-thm}
update{preserved}
pfm-len-redima-rec-25
pfm-len-make-list-ac-26
pfm-car-redima-rec-27
pfm-rational-listp-make-list-ac-28
pfm-redima-rec-29
pfm-matrixap-aux-redima-rec-30
redim{preserved}
pfm-true-listp-reset-matrixc-31
pfm-len-reset-matrixc-32
pfm-ncolumns-reset-matrixc-33
pfm-nrows-reset-matrixc-34
pfm-integerp-ncolumns-reset-matrixc-35
pfm-integerp-nrows-reset-matrixc-36
pfm-mcp-update-nth-37
pfm-mcp-car-reset-matrixc-38
pfm-len-resize-list-39
```

```
pfm-mcp-car-reset-matrixc-40
pfm-mcp-resize-list-41
pfm-linear-nrows-reset-matrixc-42
pfm-linear-ncolumns-reset-matrix-43
pfm-len-car-reset-matrixc-44
pfm-cadr-nth-1-45
pfm-nth-make-list-46
pfm-equal-redima-rec-make-list-47
pfm-equal-nth-car-48
pfm-equal-car-redima-rec-make-list-49
pfm-all-equal-elements-redima-rec-50
pfm-len-redima-rec-51
pfm-nth-redima-rec-make-list-52
pfm-elements-redima-rec-53
pfm-consp-car-update-i-54
pfm-caar-update-i-55
pfm-matrixcp+-update-i-56
pfm-len-update-i-57
pfm-nth-reset-matrix-58
pfm-nth-update-i-59
pfm-nth-reset-matrixc-60
pfm-elements-reset-matrixc-61
pfm-elements-redima-rec-reset-matrixc-62
pfm-equal-row-redima-rec-reset-matrixc-63
pfm-equal-rows-redima-rec-reset-matrixc-64
redim{correspondence}
pfm-rational-listp-matrixpap-aux-65
pfm-rational-rational-listp-66
pfm-len-nth-i-car-67
```

Lista de teoremas:

```
pfm-lookup-update-same
pfm-lookup-update-diff
pfm-lookup-update-update-same
pfm-lookup-update-update-exchange
pfm-nrows-update
pfm-ncolumns-update
pfm-matrixp-update
pfm-nrows-redim
pfm-ncolumns-redim
pfm-matrixp-redim
pfm-lookup-redim
pfm-matrixp-nrows
pfm-matrixp-ncolumns
pfm-rationalp-lookup
```

`matrix-defuns.lisp`

- **Número de líneas:** 879
- **Número de eventos:** 67

- **Número de funciones:** 39
- **Número de lemas locales:** 24
- **Número de teoremas globales:** 0

Lista de lemas locales:

```
pfm-matrixp-identity-row-01
pfm-nrows-identity-row-02
pfm-ncolumns-identity-row-03
pfm-matrixp-transpose-row-04
pfm-nrows-transpose-row-05
pfm-ncolumns-transpose-row-06
pfm-matrixp-negate-row-07
pfm-nrows-negate-row-08
pfm-ncolumns-negate-row-09
pfm-nrows-scalar-row-10
pfm-ncolumns-scalar-row-11
pfm-matrixp-update-scalar-multiply-12
pfm-matrixp-scalar-row-13
pfm-nrows-add-row-14
pfm-ncolumns-add-row-15
pfm-matrixp-add-row-16
pfm-rationalp-product-17
pfm-rationalp-dot-product-18
pfm-matrixp-multiply-row-19
pfm-nrows-multiply-row-20
pfm-ncolumns-multiply-row-21
pfm-nrows-copy-row-22
pfm-ncolumns-copy-row-23
pfm-matrixp-copy-row-24
```

`equal-matrix-equal.lisp`

- **Número de líneas:** 272
- **Número de eventos:** 32
- **Número de funciones:** 2
- **Número de lemas locales:** 23
- **Número de teoremas globales:** 1

Lista de lemas locales:

```
pfm-reflexivity-of-equal-row
pfm-symmetry-of-equal-row
pfm-transitivity-of-equal-row
pfm-reflexivity-of-equal-rows
pfm-symmetry-of-equal-rows
```

```
pfm-transitivity-of-equal-rows
pfm-nth-contrajemplo-acotado
pfm-nth-contraejemplo-incorrecto
pfm-nth-contraejemplo-correcto
pfm-equal-matrix-contrajemplo-acotado
pfm-equal-matrix-contraejemplo-incorrecto
pfm-equal-matrix-contraejemplo-correcto
pfm-equal-rows-implies-equal-row-01
pfm-rationalp-nth-02
pfm-rational-listp-03
pfm-rationalp-nth-nth-04
pfm-rationalp-car-nth-05
pfm-consp-nth-06
pfm-equal-nth-nth-nth-nth-07
pfm-equal-len-nth-ncolumns-08
pfm-equal-row-equal-09
pfm-equal-rows-equal-10
pfm-equal-equal-11
```

Lista de teoremas:

```
equal-matrix-implies-equal
```

`defun-thms.lisp`

- **Número de líneas:** 338
- **Número de eventos:** 41
- **Número de funciones:** 0
- **Número de lemas locales:** 15
- **Número de teoremas globales:** 18

Lista de lemas locales:

```
pfm-zero-row-element-0
pfm-zero-rows-element-0
pfm-matrixp-identity-row
pfm-nrows-identity-row-02
pfm-ncolumns-identity-row-03
pfm-matrixp-identity-rows-04
pfm-elements-identity-row1-05
pfm-elements-identity-row2-06
pfm-elements-identity-row3-07
pfm-elements-identity-rows1-08
pfm-elements-identity-rows2-09
pfm-nrows-identity-rows-10
pfm-ncolumns-identity-rows-11
pfm-elements-identity-row4-12
pfm-elements-identity-rows2-13
```

Lista de teoremas:

```
pfm-reflexivity-of-equal-row
pfm-symmetry-of-equal-row
pfm-transitivity-of-equal-row
pfm-reflexivity-of-equal-rows
pfm-symmetry-of-equal-rows
pfm-transitivity-of-equal-rows
pfm-matrixp-zero-matrix
pfm-lookup-zero-matrix
pfm-nrows-zero-matrix
pfm-ncolumns-zero-matrix
pfm-zero-matrixp-matrixp
pfm-zero-matrixp-lookup
pfm-lookup-identity-matrix
pfm-matrixp-identity-matrix
pfm-nrows-identity-matrix
pfm-ncolumns-identity-matrix
pfm-identity-matrixp-square-matrixp
pfm-lookup-identity-matrixp
```

`unary-ops-thms.lisp`

- **Número de líneas:** 1267
- **Número de eventos:** 114
- **Número de funciones:** 2
- **Número de lemas locales:** 70
- **Número de teoremas globales:** 27

Lista de lemas locales:

```
pfm-matrixp-transpose-row-01
pfm-nrows-transpose-row-02
pfm-ncolumns-transpose-row-03
pfm-matrixp-transpose-rows-04
pfm-nrows-transpose-rows-05
pfm-ncolumns-transpose-rows-06
pfm-elements-transpose-row-07
pfm-elements-transpose-row-08
pfm-elements-transpose-rows1-09
pfm-elements-transpose-rows2-10
pfm-elements-transpose-rows3-11
pfm-equal-rows-transpose-transpose-12
pfm-equal-rows-transpose-redim-13
pfm-reflexivity-of-equal-row
pfm-symmetry-of-equal-row
pfm-transitivity-of-equal-row
pfm-reflexivity-of-equal-rows
```

```
pfm-symmetry-of-equal-rows
pfm-transitivity-of-equal-rows
pfm-equal-row-equal-column-14
pfm-equal-rows-equal-columns-15
pfm-elements-equal-column-16
pfm-elements-equal-columns-17
pfm-equal-columns-equal-row-0-18
pfm-not-equal-element-not-equal-column-19
pfm-not-equal-row-not-equal-columns-20
pfm-equal-columns-equal-rows1-21
pfm-equal-columns-equal-rows2-22
pfm-equal-columns-equal-rows3-23
pfm-equal-row-transpose-identity-24
pfm-equal-rows-transpose-identity-25
pfm-matrixp-negate-row-26
pfm-nrows-negate-row-27
pfm-ncolumns-negate-row-28
pfm-mtrixp-negate-rows-29
pfm-nrows-negate-rows-30
pfm-ncolumns-negate-rows-31
pfm-elements-negate-row-32
pfm-elements-negate-rows-33
pfm-equal-rows-negate-negate-34
pfm-equal-rows-transpose-negate-35
pfm-equal-row-negate-negate-36
pfm-equal-rows-negate-negate-37
pfm-nrows-scalar-multiply-38
pfm-ncolumns-scalar-multiply-row-39
pfm-matrixp-update-scalar-product-40
pfm-matrixp-scalar-multiply-row-41
pfm-matrixp-scalar-multiply-rows-42
pfm-nrows-scalar-multiply-rows-43
pfm-ncolumns-scalar-multiply-rows-44
pfm-elements-scalar-multiply-row-45
pfm-elements-scalar-multiply-rows-46
pfm-equal-row-scalar-multiply-scalar-multiply-47
pfm-equal-rows-scalar-multiply-scalar-multiply-48
pfm-elements-null-matrix-49
pfm-equal-row-scalar-multiply-0-50
pfm-equal-rows-scalar-multiply-0-51
pfm-matrixp-get-zero-matrix-52
pfm-nrows-get-zero-matrix-53
pfm-ncolumns-get-zero-matrix-54
pfm--zero-rows-scalar-multiply-0-55
pfm-matrixp-get-zero-matrix-56
pfm-nrows-get-zero-matrix-57
pfm-ncolumns-get-zero-matrix-58
pfm-equal-row-scalar-multiply-get-zero-matrix-59
pfm-equal-rows-scalar-multiply-get-zero-matrix-60
pfm-zero-row-scalar-multiply-61
pfm-zero-rows-scalar-multiply-62
pfm-equal-row-scalar-multiply-1-63
pfm-equal-rows-scalar-multiply-2-64
```

```
pfm-equal-row-scalar-multiply--1-65
pfm-equal-rows-scalar-multiply--1-66
pfm-equal-row-transpose-scalar-multiply-67
pfm-equal-rows-transpose-scalar-multiply-68
pfm-equal-rows-scalar-multiply-negate-69
```

Lista de teoremas:

```
pfm-matrixp-transpose
pfm-square-matrixp-transpose
pfm-nrows-transpose
pfm-ncolumns-transpose
pfm-lookup-transpose
pfm-idempotency-of-transpose
pfm-equal-matrix-transpose-zero-matrix
pfm-transpose-identity-matrix
pfm-matrixp-negate
pfm-nrows-negate
pfm-ncolumns-negate
pfm-lookup-negate
pfm-idempotency-of-negate
pfm-transpose-negate
pfm-matrixp-scalar-multiply
pfm-nrows-scalar-multiply
pfm-ncolumns-scalar-multiply
pfm-lookup-scalar-multiply
pfm-associativity-of-scalar-multiply
pfm-neutral-element-of-scalar-multiply
pfm-neutral-element-of-scalar-multiply-2
pfm-neutral-element-of-scalar-multiply-3
pfm-neutral-element-of-scalar-multiply-4
pfm-neutral-element-of-scalar-multiply-5
pfm-opposite-element-of-scalar-multiply
pfm-transpose-scalar-multiply
pfm-scalar-multiply-negate
```

`add-matrix-thms.lisp`

- **Número de líneas:** 750
- **Número de eventos:** 63
- **Número de funciones:** 0
- **Número de lemas locales:** 36
- **Número de teoremas globales:** 18

Lista de lemas locales:

```
pfm-nrows-add-matrix-rows-01
pfm-ncolumns-add-matrix-row-02
pfm-matrixp-add-matrix-row-03
pfm-matrixp-add-matrix-rows-04
pfm-nrows-add-matrix-rows-05
pfm-ncolumns-add-matrix-rows-06
pfm-elements-add-matrix-row-07
pfm-elements-add-matrix-rows-08
pfm-equal-row-add-add-09
pfm-equal-rows-add-add-10
pfm-equidimensionalp-add-matrix-11
pfm-matrixp-add-add-ab-c-12
pfm-matrixp-add-add-a-bc-13
pfm-nrows-add-matrix-ab-c-14
pfm-nrows-add-matrix-a-bc-15
pfm-ncolumns-add-matrix-ab-c-16
pfm-ncolumns-add-matrix-a-bc-17
pfm-elements-add-matrix-ab-c-18
pfm-elements-add-matrix-a-bc-19
pfm-equal-row-add-add-20
pfm-equal-rows-add-add-21
pfm-equal-row-add-redim-22
pfm-equal-row-add-redim-23
pfm-equal-row-redim-add-24
pfm-equal-rows-redim-add-25
pfm-equal-row-scalar-add-26
pfm-equal-rows-scalar-add-27
pfm-equal-row-add-scalar-28
pfm-equal-rows-add-scalar-29
pfm-equal-row-transpose-add-30
pfm-equal-rows-transpose-add-31
pfm-not-equal-add-redim-32
pfm-equal-row-add-negate-33
pfm-equal-rows-add-negate-34
pfm-equal-row-negate-add-35
pfm-equal-rows-negate-add-36
```

Lista de teoremas:

```
pfm-matrixp-add-matrix
pfm-nrows-add-matrix
pfm-ncolumns-add-matrix
pfm-lookup-add-matrix
pfm-commutativity-of-add-matrix
pfm-associativity-of-add-matrix
pfm-neutral-element-of-add-matrix-rigth
pfm-neutral-element-of-add-matrix-left
pfm-distributivity-of-scalar-multiply-over-+
pfm-opposite-element-of-add-matrix-right
pfm-opposite-element-of-add-matrix-left
pfm-distributivity-of-scalar-multiply-over-add-matrix
pfm-double-add-matrix-scalar-multiply
pfm-transpose-add-matrix
```

```
pfm-neutral-element-add-matrix-2
pfm-neutral-element-add-matrix-3
pfm-uniqueness-of-opposite-element-add-matrix
pfm-distributivity-of-negate-over-+
```

multiply-matrix-thms.lisp

- **Número de líneas:** 1115
- **Número de eventos:** 70
- **Número de funciones:** 2
- **Número de lemas locales:** 52
- **Número de teoremas globales:** 16

Lista de lemas locales:

```
pfm-rationalp-product-01
pfm-rationalp-dot-product-02
pfm-matrixp-multiply-matrix-row-03
pfm-nrows-multiply-matrix-row-04
pfm-ncolumns-multiply-matrix-row-05
pfm-matrixp-multiply-matrix-rows-06
pfm-nrows-multiply-matrix-rows-07
pfm-ncolumns-multiply-matrix-rows-08
pfm-elements-multiply-matrix-row-09
pfm-elements-multiply-matrix-rows-10
pfm-dot-redim-11
pfm-equal-row-multiply-matrix-redim-12
pfm-equal-rows-multiply-matrix-redim-13
pfm-redim-dot-14
pfm-equal-row-redim-multiply-matrix-15
pfm-equal-rows-redim-multiply-matrix-16
pfm-equal-dot-identity-17
pfm-equal-row-multiply-matrix-identity-18
pfm-equal-rows-multiply-matrix-identity-19
pfm-equal-identity-dot-20
pfm-equal-row-identity-multiply-matrix-21
pfm-equal-rows-identity-multiply-matrix-22
pfm-equal-pa-bc-dot-23
pfm-equal-pab-c-pa-bc-24
pfm-equal-dot-multiply-matrix-pab-c-25
pfm-equal-elements-multiply-multiply-26
pfm-equal-pa-bc-dot-27
pfm-dot-multiply-matrix-pa-bc-28
pfm-elements-multiply-matrix-pa-bc-29
pfm-equal-row-multiply-multiply-30
pfm-equal-rows-multiply-multiply-31
pfm-dot-add-32
pfm-equal-row-multipy-add-matrix-33
```

```
pfm-equal-rows-multiply-add-matrix-34
pfm-equal-add-dot-35
pfm-equal-row-add-multiply-matrix-36
pfm-equal-rows-add-multiply-matrix-37
pfm-equal-dot-negate-38
pfm-equal-row-multiply-negate-39
pfm-equal-rows-multiply-negate-40
pfm-negate-dot-41
pfm-equal-row-negate-multiply-42
pfm-equal-rows-negate-multiply-43
pfm-dot-scalar-44
pfm-equal-row-scalar-multiply-45
pfm-equal-rows-scalar-multiply-46
pfm-scalar-dot-47
pfm-equal-row-multiply-scalar-48
pfm-equal-rows-multiply-scalar-49
pfm-dot-transpose-transpose-50
pfm-equal-row-transpose-transpose-51
pfm-equal-rows-transpose-transpose-52
```

Lista de teoremas:

```
pfm-matrixp-multiply-matrix
pfm-nrows-multiply-matrix
pfm-ncolumns-multiply-matrix
pfm-lookup-multiply-matrix
pfm-neutral-element-multiply-matrix-left
pfm-neutral-element-multiply-matrix-right
pfm-unity-element-multiply-matrix-left
pfm-unity-element-multiply-matrix-right
pfm-associativity-of-multiply-matrix
pfm-distributivity-of-multiply-matrix-add-matrix-left
pfm-distributivity-of-multiply-matrix-add-matrix-right
pfm-multiply-matrix-negate-left
pfm-multiply-matrix-negate-right
pfm-multiply-matrix-scalar-multiply-left
pfm-multiply-matrix-scalar-multiply-right
pfm-transpose-multiply-matrix
```

`row-operations.lisp`

- **Número de líneas:** 1081
- **Número de eventos:** 74
- **Número de funciones:** 7
- **Número de lemas locales:** 38
- **Número de teoremas globales:** 19

Lista de lemas locales:

```
pfm-lookup-exchange-elements
pfm-matrixp-exchange-elements
pfm-nrows-exchange-elements
pfm-ncolumns-exchange-elements
pfm-elements-exchange-rows-aux-01
pfm-matrixp-exchane-rows-aux-02
pfm-nrows-exchange-rows-aux-03
pfm-ncolumns-exchange-rows-aux-04
pfm-equal-row-echange-rows-05
pfm-equal-rows-echange-rows-06
pfm-dot-identity-07
pfm-dot-exchange-rows1-08
pfm-dot-exchange-rows2-09
pfm-dot-exchange-rows3-10
pfm-dot-exchange-rows4-11
pfm-dot-exchange-rows5-12
pfm-equal-row-multiply-exchange-rows-13
pfm-equal-rows-multiply-exchange-rows-14
pfm-elements-multiply-row-aux-15
pfm-rationalp-product-lookup-16
pfm-matrixp-multiply-row-aux-17
pfm-nrows-multiply-row-aux-18
pfm-ncolumns-multiply-row-aux-19
pfm-dot-multiply-row1-20
pfm-dot-multiply-row2-21
pfm-dot-multiply-row3-22
pfm-equal-row-multiply-multiply-row-23
pfm-equal-rows-multiply-multiply-row-24
pfm-elements-saxpy-row-aux-25
pfm-rationalp-add-product-26
pfm-matrixp-saxpy-row-aux-27
pfm-nrows-saxpy-row-aux-28
pfm-ncolumns-saxpy-row-aux-29
pfm-dot-saxpy-row1-30
pfm-dot-saxpy-row2-31
pfm-dot-saxpy-row3-32
pfm-equal-row-multiply-saxpy-row-33
pfm-equal-rows-multiply-saxpy-row-34
```

Lista de teoremas:

```
pfm-lookup-exchange-rows
pfm-matrixp-exchange-rows
pfm-nrows-exchange-rows
pfm-ncolumns-exchange-rows
pfm-exchange-rows-i-i
pfm-exchange-rows-identity-matrix
pfm-lookup-multiply-row
pfm-matrixp-multiply-row
pfm-nrows-multiply-row
pfm-ncolumns-multiply-row
pfm-multiply-row-identity-matrix
pfm-lookup-saxpy-row
```

```
pfm-matrixp-saxpy-row
pfm-nrows-saxpy-row
pfm-ncolumns-saxpy-row
pfm-saxpy-row-identity-matrix
pfm-exchange-rows-multiply
pfm-multiply-row-multiply
pfm-saxpy-row-multiply
```

gauss-jordan.lisp

- **Número de líneas:** 853
- **Número de eventos:** 78
- **Número de funciones:** 15
- **Número de lemas locales:** 16
- **Número de teoremas globales:** 36

Lista de lemas locales:

```
pfm-<=-index-03
pfm-not-equal-0-index1-04
pfm-not-equal-0-index2-05
pfm-nrows-copy-row-06
pfm-ncolumns-copy-row-07
pfm-matrixp-copy-row-08
pfm-elements-copy-row-lemma09
pfm-nrows-copy-rows-10
pfm-ncolumns-copy-rows-11
pfm-matrixp-copy-rows-12
pfm-elements-copy-rows-13
pfm-equal-row-copy-matrix-14
pfm-equal-rows-copy-matrix-15
pfm-multiply-left-reduce-columns-right-reduce-columns-41
pfm-rows-columns-properties-initialize-concat-44
pfm-multiply-left-reduce-columns-initialize-concat-46
```

Lista de teoremas:

```
pfm-rationalp-negate-fraction
pfm-rationalp-inverse-fraction
pfm-nrows-copy-matrix
pfm-ncolumns-copy-matrix
pfm-matrixp-copy-matrix
pfm-lookup-copy-matrix
pfm-equal-copy-matrix
pfm-get-left-initializa-concat-16
pfm-get-right-initialize-concat-17
pfm-matrixp-get-left-18
pfm-matrixtp-get-right-19
```

```
pfm-rationalp-right-20
pfm-concatp-exchange-rows-21
pfm-nrows-left-exchange-rows-22
pfm-ncolumns-left-exchange-rows-23
pfm-nrows-right-exchange-rows-24
pfm-ncolumns-right-exchange-rows-25
pfm-concatp-multiply-row-26
pfm-nrows-left-multiply-row-27
pfm-ncolumns-left-multiply-row-28
pfm-nrows-right-multiply-row-29
pfm-ncolumns-right-multiply-row-30
pfm-concatp-saxpy-row-31
pfm-nrows-left-saxpy-row-32
pfm-ncolumns-left-saxpy-row-33
pfm-nrows-right-saxpy-row-34
pfm-ncolumns-right-saxpy-row-35
pfm-multiply-left-exchange-rows-right-36
pfm-multiply-left-multiply-row-right-37
pfm-multiply-left-saxpy-row-right-38
pfm-left-update-determinant-39
pfm-right-update-determinant-40
pfm-left-initialize-concat-42
pfm-right-initialize-concat-43
pfm-concatp-initialize-concat-45
pfm-gauss-jordan-is-correct
```

`time-metrics.lisp`

- **Número de líneas:** 167
- **Número de eventos:** 95
- **Número de funciones:** 4
- **Número de lemas locales:** 0
- **Número de teoremas globales:** 0

7.2. Resumen en cifras del proyecto

Aunque se han contabilizado casi todos los eventos que forman parte del código fuente, hay algunas demostraciones que no aparecen en las tablas anteriores por su dificultad para que el "parser" las encuentre, como las *congruencias*, las *equivalencias* y *refinamientos*.

Podemos resumir en grandes cifras las secciones anteriores en la siguiente tabla:

Nombre	Líneas	Eventos	Funciones	Lemas	Teoremas
abstract-stobj.lisp	1171	119	18	78	14
matrix-defuns.lisp	879	67	39	24	0
equal-matrix-equal.lisp	272	32	2	23	1
defun-thms.lisp	338	41	0	15	18
unary-ops-thms.lisp	1267	114	2	70	27
add-matrix-thms.lisp	750	63	0	36	18
multiply-matrix-thms.lisp	1115	70	2	52	16
row-operations.lisp	1081	74	7	38	19
gauss-jordan.lisp	853	78	15	16	36
time-metrics.lisp	167	95	4	0	0
TOTAL	7893	753	89	352	149

Figura 6: Contabilidad en el código fuente del proyecto.

Total de teoremas (lemas locales más teoremas globales) demostrados: 501

8. Anexo B: Medida de la eficiencia temporal y espacial

En este anexo vamos a medir la eficiencia de los algoritmos planteados en base a dos criterios: Tiempo de ejecución y cantidad de bytes reservados de forma dinámica por el sistema.

Medir el tiempo de ejecución es importante en este trabajo ya que trataremos de comprobar que las mejoras en este aspecto introducidas por el uso de stobj's existen y se puede cuantificar.

Por otro lado es importante también medir la cantidad de memoria reservada de forma dinámica. La reserva de memoria en algoritmos de computación puede hacerse siempre de dos formas. En forma estática la reserva de memoria se resuelve en tiempo de compilación y el sistema operativo, al ejecutar el proceso, ya tiene reservada toda la memoria que pueda necesitar el algoritmo. Por contra, la reserva de memoria dinámica es más flexible, ya que permite la reserva de memoria en cantidades que no se conocen hasta estar en tiempo de ejecución.

La reserva dinámica de memoria tiene, sin embargo, ciertos inconvenientes. Dicha reserva se hace en base a una llamada a sistema de forma que se puedan gestionar y modificar las tablas de páginas del proceso en ejecución. Esta llamada, cuya implementación depende en gran medida del sistema operativo utilizado, puede ser más o menos compleja dependiendo de los algoritmos de gestión de memoria usados, pero tienen una característica en común: El tiempo que tarda en resolverse la llamada no está acotado y puede variar bastante dependiendo del momento en el que se haga la reserva dinámica.

Además, esta reserva dinámica siempre desperdicia algo de la memoria que el sistema asigna al proceso, así que el aprovechamiento de la memoria física es peor. Consideramos, por tanto, que un algoritmo que use y reserve menos memoria dinámica será mejor.

Vamos a medir el tiempo de ejecución y la memoria reservada de forma dinámica en tres implementaciones diferentes de los algoritmos descritos y definidos en este trabajo:

- **TFM**: Con estas letras designaremos el enfoque seguido en este Trabajo Fin de Máster, es decir, con stobj's concretos.
- **Gamboa**: Mediremos la eficiencia en los algoritmos implementados por Gamboa, Cowles y Van Baalen en 2003 en su artículo "Using ACL2 to Formalize Matrix Algebra", único antecedente válido del presente trabajo.
- **LoL**: Y por último haremos uso de los algoritmos siguiendo la implementación más intuitiva, pero a la vez menos teóricamente eficiente, con la que se plantearon los stobj's abstractos en este proyecto. Llamaremos a esta implementación LoL de "List of Lists".

Todas las pruebas se han realizado sobre el mismo hardware, consistente en un procesador $i7^{\text{TM}}$ *Lynnfield* de Intel® con frecuencia de reloj de 2,93 GHz, 4 núcleos y 8 hilos con 4 GBytes de memoria principal. Dicho chip fue punta de lanza en los procesadores de propósito general cuando fue adquirido hace tres años y sigue siendo un excelente representante de los mejores procesadores actuales.

Los algoritmos usados para la medición de estas cantidades han sido: la suma de matrices, la multiplicación de matrices y el cálculo de la matriz inversa por Gauss-Jordan. Siempre se han realizado sobre matrices cuadradas de cierto orden n, por lo que las matrices a considerar siempre tendrán n filas y n columnas. En concreto se han realizado mediciones para los siguientes valores de n:

- *Suma de matrices:* Diez pruebas con $n = \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$.
- *Multiplicación de matrices*: Diez pruebas con $n = \{30, 60, 90, 120, 150, 180, 210, 240, 270, 300\}$.
- *Inversa de una matriz*: Diez pruebas con $n = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

Para cada una de las pruebas anteriores se ha medido el tiempo de ejecución y la memoria reservada de forma dinámica. Dicho tiempo de ejecución siempre se dará en *segundos*, mientras que la cantidad de memoria se medirá en *Kilobytes*. Por lo tanto, se respetan estás unidades en todos los gráficos y tablas que vienen a continuación.

La forma de generar todas las matrices ha sido la siguiente: Cada elemento se genera de forma aleatoria de manera que tenga valores enteros comprendidos entre -10 y 10, ambos inclusive. Las funciones que generan una matriz de esta forma se dan a continuación:

```
(defttag t)
(remove-untouchable create-state t)
(set-state-ok t)

(defun random-element ()
  (with-local-state (mv-let (result state)
                            (random$ 21 state)
                            result)))

(defun random-row (A m n)
  (if (zp n)
    (update A m 0 (- (random-element) 10))
    (seq A
         (update A m n (- (random-element) 10))
         (random-row A m (1- n)))))

(defun random-rows (A m n)
  (if (zp m)
    (random-row A 0 n)
    (seq A
         (random-row A m n)
         (random-rows A (1- m) n))))

(defun random-matrix (A m n)
  (seq A
       (redim A m n)
       (random-rows A (1- m) (1- n))))
```

Los eventos `defttag`, `remove-untouchable` y `set-state-ok` sirven aquí para poder usar de forma local el objeto `<state>` de ACL2 para la generación de números aleatorios.

Se dan, en cada caso, los datos en bruto para su posible cotejo y una representación de las magnitudes medidas por ejes, donde el de abcisas representa el orden de las matrices y en el eje de ordenadas tendremos la cantidad a medir. Para lograr medir estas magnitudes se puede usar un comando en ACL2, llamado `time$` cuyo argumento debe ser la llamada a la función a medir, por ejemplo:

```
(random-matrix A 1000 1000)
(random-matrix B 1000 1000)
(time$ (add-matrix C A B))

(random-matrix A 300 300)
(random-matrix B 300 300)
(time$ (multiply-matrix C A B))

```

```
9  (random-matrix A 100 100)
10 (time$ (inverse A))
```

Estas definiciones y llamadas están en el fichero del proyecto `time-metrics.lisp`.

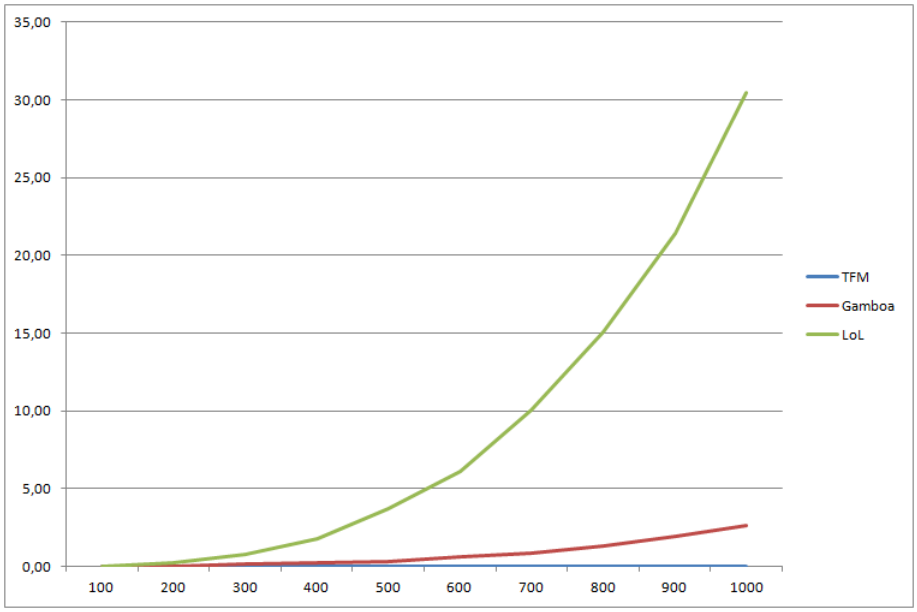
8.1. Medida del tiempo de ejecución

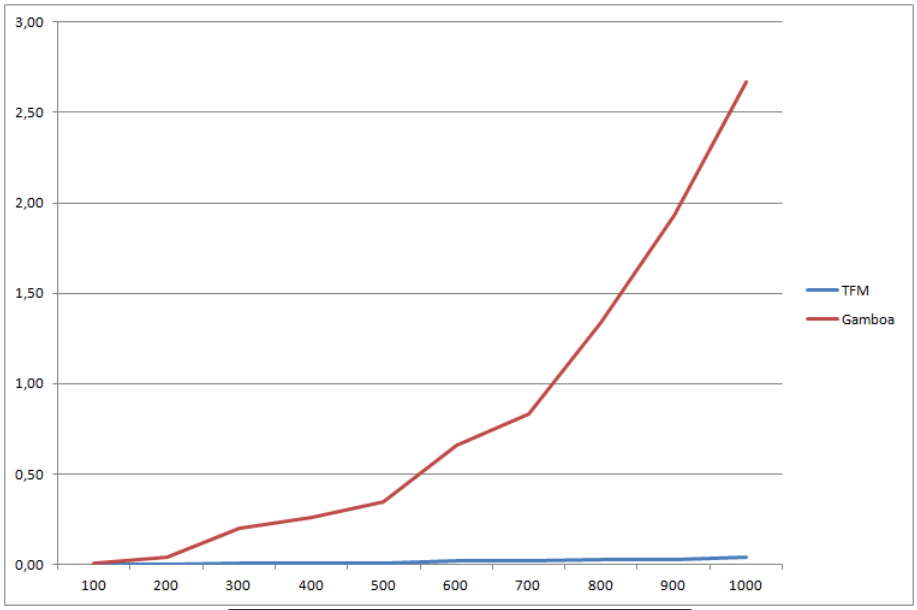
Las tres primeras figuras siguientes nos dan las medidas y comparaciones de los tiempos de ejecución para los tres algoritmos elegidos.

En la suma de matrices no hay paliativos. El algoritmo usando stobj's concretos es mucho mejor que el resto. Y el de Gamboa es mejor que LoL. Como dato curioso se quiso forzar la máquina con una matriz de 10000×10000, es decir, con 100 millones de elementos, dando como resultado en nuestro proyecto (TFM) 3,67 segundos de tiempo de ejecución. Prueba, por cierto, imposible de realizar en el resto de algoritmos.

Por otro lado, en la multiplicación de matrices se mantiene la misma tendencia, es decir, el mejor algoritmo, con diferencia, es TFM, en segundo lugar está Gamboa y, por último, el peor algoritmo en cuanto a eficiencia es LoL.

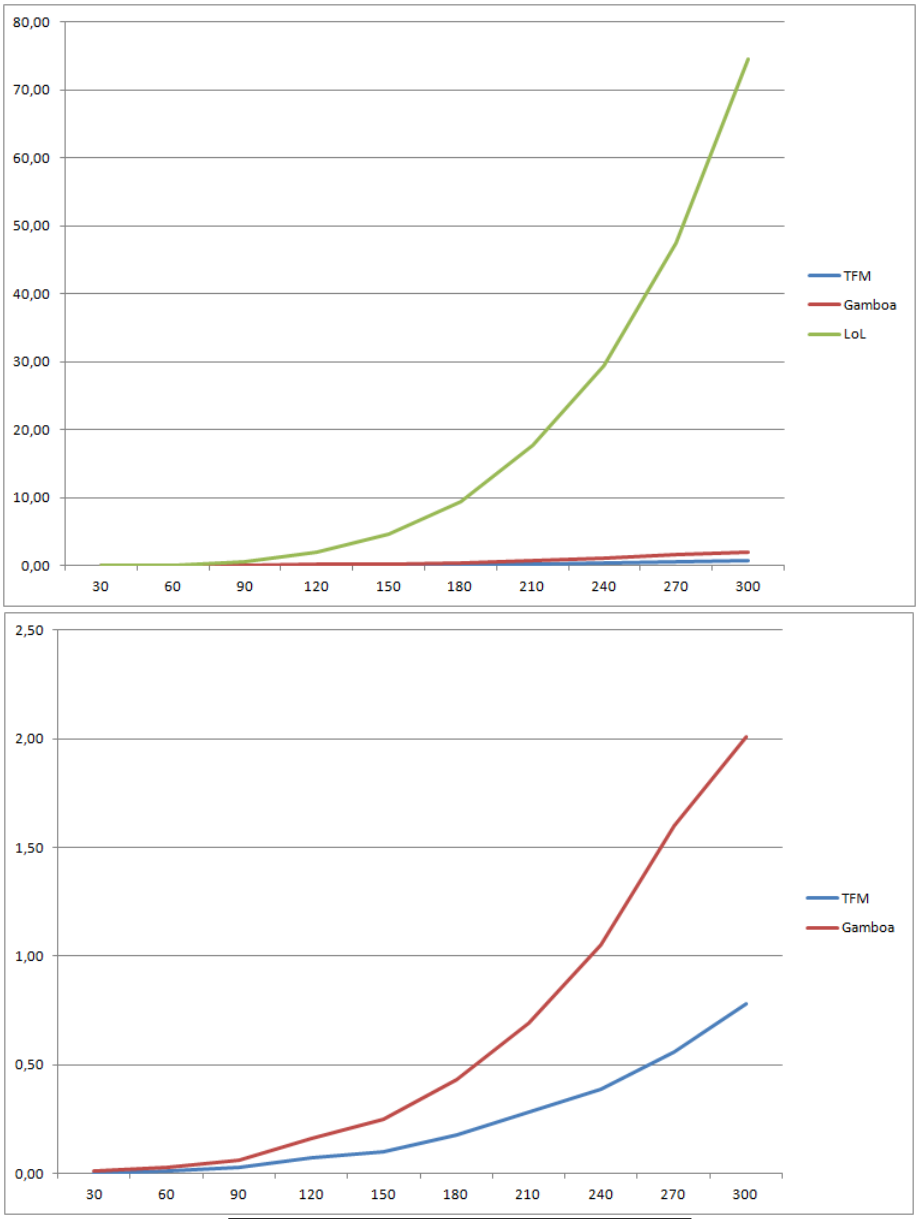
En cambio, al observar las gráficas del algoritmo del cálculo de la matriz inversa, se ve que se equiparan en cierta medida los tiempos de ejecución. De todas formas TFM sigue siendo un 20 % mejor que Gamboa en el tiempo de ejecución. Como nota curiosa hay que hacer destacar que el algoritmo LoL se ha situado en segundo lugar en este caso.





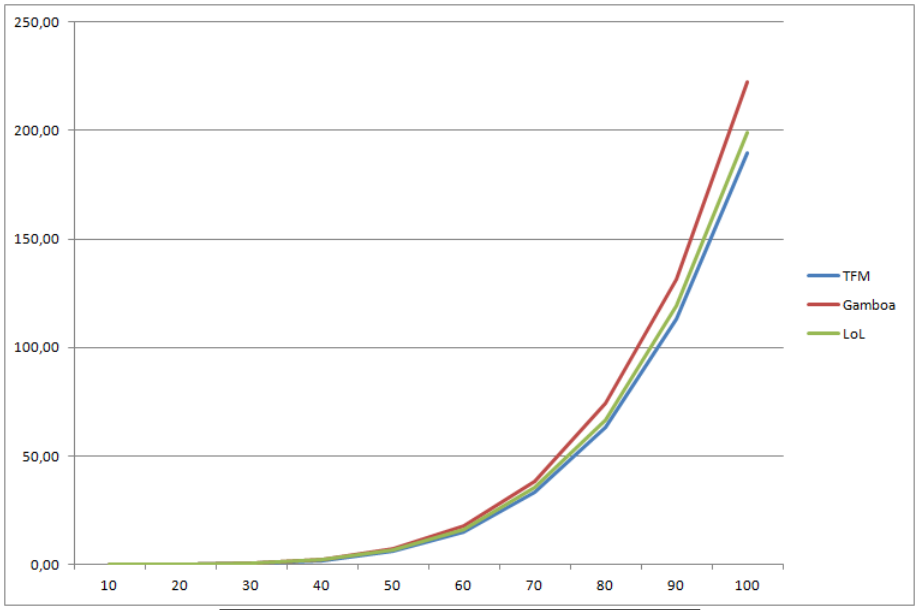
Orden	TFM	Gamboa	LoL
100	0,00	0,01	0,03
200	0,00	0,04	0,24
300	0,01	0,20	0,77
400	0,01	0,26	1,79
500	0,01	0,35	3,71
600	0,02	0,66	6,10
700	0,02	0,83	10,06
800	0,03	1,34	15,05
900	0,03	1,93	21,43
1000	0,04	2,67	30,50

Figura 7: Tiempo de ejecución en el algoritmo de suma de matrices.



Orden	TFM	Gamboa	LoL
30	0,00	0,01	0,00
60	0,01	0,03	0,02
90	0,03	0,06	0,62
120	0,07	0,16	1,88
150	0,10	0,25	4,57
180	0,18	0,43	9,44
210	0,28	0,69	17,74
240	0,39	1,05	29,41
270	0,56	1,60	47,56
300	0,78	2,01	74,65

Figura 8: Tiempo de ejecución en el algoritmo de multiplicación de matrices.



Orden	TFM	Gamboa	LoL
10	0,00	0,00	0,00
20	0,06	0,09	0,05
30	0,48	0,58	0,52
40	1,98	2,43	2,28
50	6,15	7,32	6,80
60	15,16	17,80	16,18
70	33,06	38,44	35,31
80	63,36	74,19	66,37
90	113,03	131,34	119,28
100	189,71	222,63	198,88

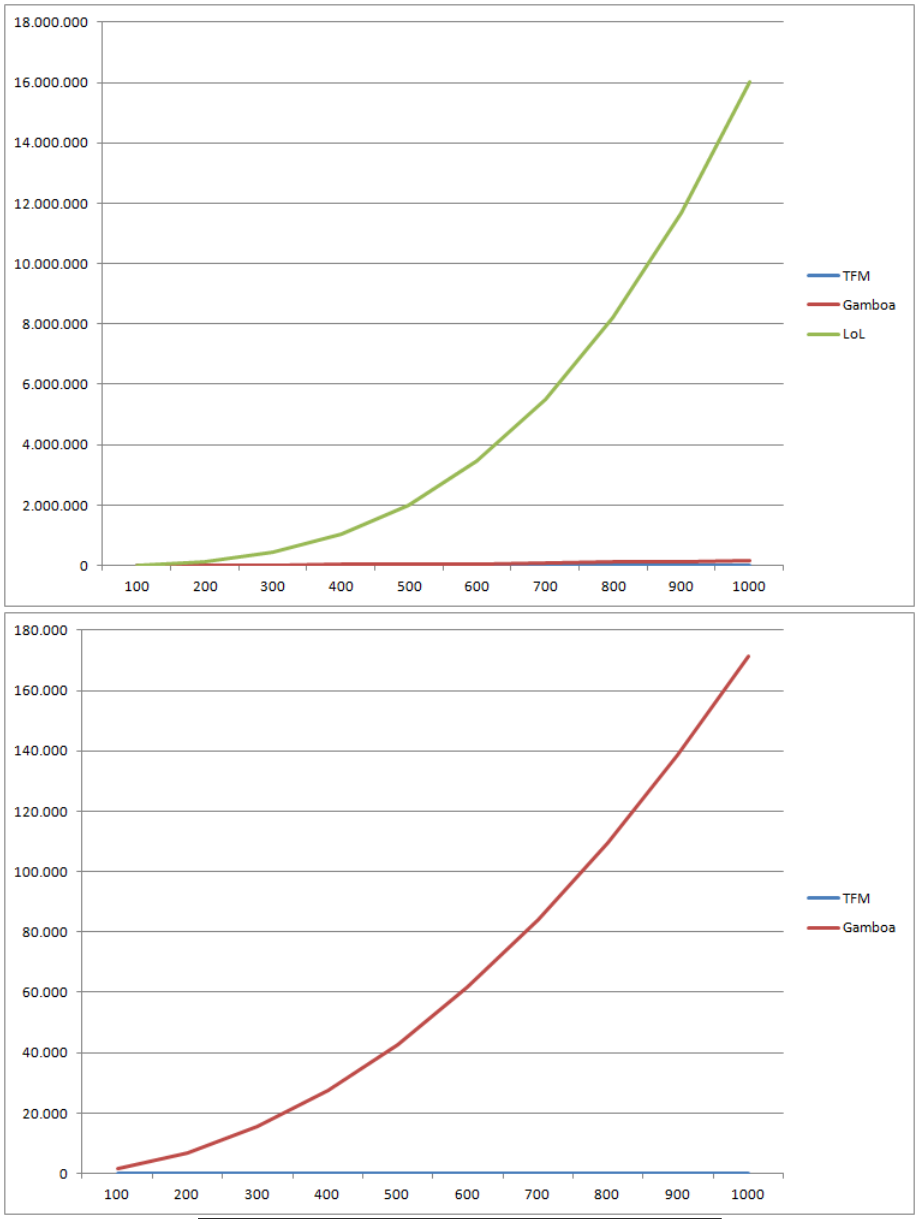
Figura 9: Tiempo de ejecución en el algoritmo de Gauss-Jordan.

8.2. Medida de la cantidad de memoria reservada de forma dinámica

En cuanto a la cantidad de memoria reservada en tiempo de ejecución, las tres gráficas siguientes muestran los resultados obtenidos.

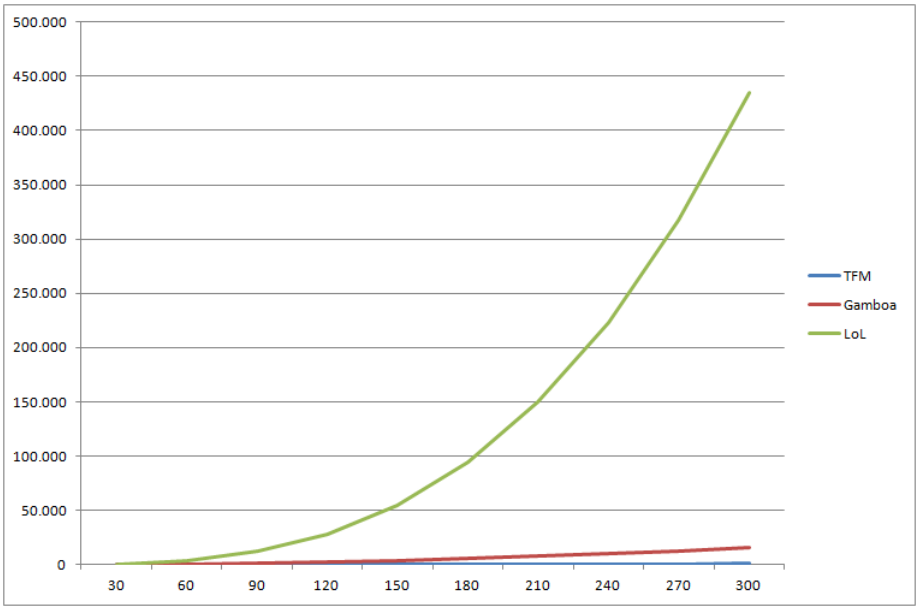
Para empezar cabe destacar que, en la suma de matrices, no existe apenas reserva de memoria dinámica en TFM. Esto es debido a que la sintaxis de la llamada a `(add-matrix A B)` devuelve el resultado sobre la *propia* matriz A y, además, no necesita ser redimendionada. En efecto, redimensionar una matriz en tiempo de ejecución sí que hace uso de memoria reservada de forma dinámica.

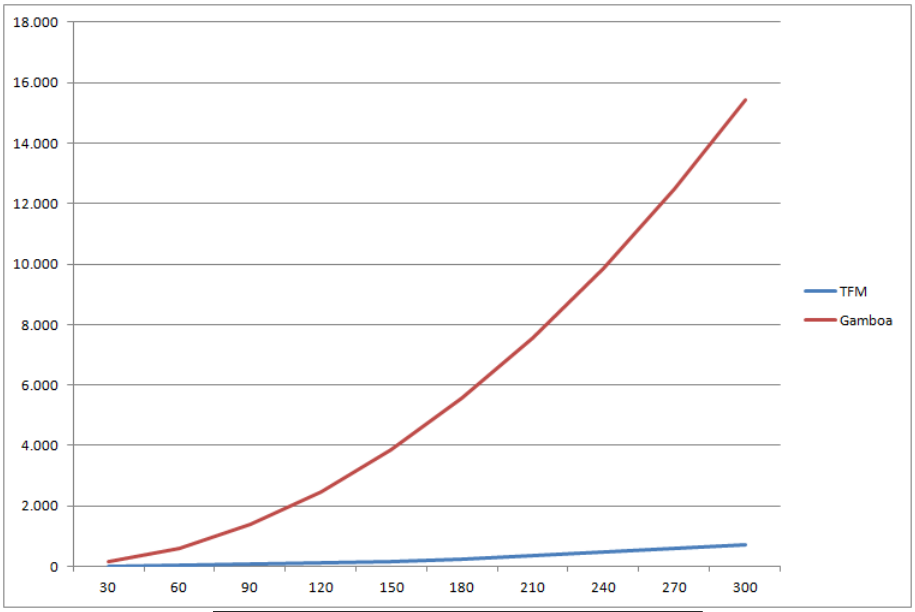
Por eso, en el algoritmo de multipliación de matrices sí se precisa de memoria dinámica para el almacenamiento de la matriz destino C, recuérdese esta sintaxis especial de las llamadas a `(multiply-matrix C A B)`. Aún así, la gestión de la memoria es mucho mejor en nuestro algoritmo que en el resto. La razón principal de esto es que los algoritmos de Gamboa se basan en el uso de listas de asociación y su implementación mediante arrays redimensionables de forma que casi cada inserción de un nuevo elemento en la matriz destino redimensiona el vector destino. Esto repercute negativamente en el tiempo de ejecución, como se vio en las gráficas anteriores, y en el uso y gestión de la memoria. La tendencia se mantiene uniforme en las gráficas correspondientes a la matriz inversa.



Orden	TFM	Gamboa	LoL
100	1	1.716	16.161
200	1	6.855	128.641
300	1	15.421	433.441
400	1	27.425	1.026.561
500	1	42.857	2.004.001
600	1	61.714	3.461.761
700	1	83.984	5.495.841
800	1	109.710	8.202.241
900	1	138.836	11.676.961
1000	2	171.402	16.016.001

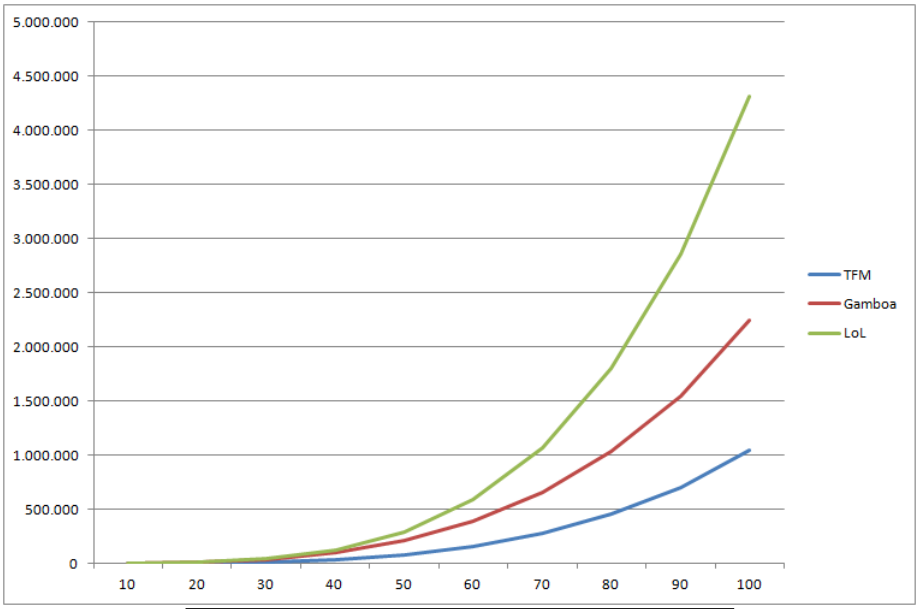
Figura 10: Memoria reservada de forma dinámica en el algoritmo de suma de matrices.





Orden	TFM	Gamboa	LoL
30	8	154	462
60	30	618	3.573
90	66	1.388	11.925
120	116	2.468	28.111
150	181	3.854	54.723
180	260	5.554	94.352
210	354	7.556	149.591
240	462	9.869	223.032
270	584	12.491	317.266
300	721	15.428	434.886

Figura 11: Memoria reservada de forma dinámica en el algoritmo de multiplicación de matrices.



Orden	TFM	Gamboa	LoL
10	106	710	472
20	2.471	8.560	8.048
30	12.395	36.003	40.280
40	36.225	97.240	122.359
50	82.825	207.253	291.376
60	160.408	385.651	588.802
70	282.560	651.577	1.069.614
80	461.032	1.034.553	1.800.019
90	705.055	1.546.362	2.859.853
100	1.051.436	2.246.356	4.313.391

Figura 12: Memoria reservada de forma dinámica en el algoritmo de Gauss-Jordan.

9. Bibliografía

[1] M. Kaufmann, P. Manolios, J Strother Moore. *"Computer-aided reasoning: An Approach"*. Kluwer Academic Publishers, 2000.

[2] S. Goel, W. Hunt, M. Kaufmann. *"Abstract stobjs and their application to ISA modeling"*. ACL2 Workshop 2013.

[3] J.L. Ruiz Reina. *"Una teoría computacional acerca de la lógica ecuacional"*. Tesis doctoral, Universidad de Sevilla, 2001.

[4] J.L. Ruiz Reina, F.J. Martín Mateos, J.A. Alonso y M. J. Hidalgo. *"Verificación formal y eficiencia: Un caso de estudio aplicado a la unificación de términos"*. I Taller Iberoamericano de Deducción e Inteligencia Artificial, 2002.

[5] F. J. Cobos, A. Osuna, R. Robles, B. Silva. *"Apuntes de Álgebra Lineal para la titulación de Ingeniería Técnica en informática de Gestión"*. Universidad de Sevilla.

[6] J. Cowles, R. Gamboa, J.V. Baalen. *"Using ACL2 arrays to formalize matrix algebra"*. ACL2 Workshop 2003.

[7] G. Infante, A. Ezequiel, J Strother Moore. *"A suit of tools for analyzing ACL2 books"*. 2010.

[8] M. Kaufmann, P. Manolios, J Strother Moore. *"Computer-Aided Reasoning: ACL2 Case Studies"*. Kluwer Academic Publishers, 2000.

[9] J.L. Ruiz Reina. *"Una introducción al sistema ACL2"*. Universidad de Sevilla, 2005.

[10] M. Kaufmann, J Strother Moore. `http://www.cs.utexas.edu/users/moore/acl2/`. Documentación online de ACL2.