IMPLEMENTACIÓN DE UN INTÉRPRETE PARA UN LENGUAJE DE CONSULTAS PURAMENTE RELACIONAL Y COMPLETO

Barroso, Luis Jesús(*); Gagliardi, Edilma Olinda(*); Molina, Gladys Mabel(*)
Quiroga, José Armando(*); Turull Torres, José María(**)

Fac. de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
Avda. Ej. de los Andes 950 (5700) San Luis

Abstract

Se describe la caracterización de la clase de queries computables: **CQ**. Se define un lenguaje de programación relacionalmente completo, denominado Query Language (**QL**). Y se presenta la descripción de la implementación de un Intérprete para el lenguaje **QL**.

Palabras Claves

Queries computables: **CQ**. Lenguaje relacional completo. Intérprete para el lenguaje **QL**.

Introducción

El presente artículo describe la implementación de un lenguaje de consultas a bases de datos relacionales que es completo, llamado **QL**.

Este trabajo forma parte del Informe del Trabajo Final de la carrera Licenciatura en Ciencias de la Computación de la Facultad de Ciencias Físico, Matemáticas y Naturales, de la Universidad Nacional de San Luis, denominado "Implementación de un Intérprete para el Lenguaje QL". Este trabajo se realizó bajo la dirección del Dr. Turull Torres y la codirección de la Lic. Gagliardi.

La motivación de este trabajo se basó en considerar una visión teórica del tema que, con la implementación del lenguaje, permitió clarificar la demostración de completitud [CH,80]. Existe cierta circularidad en las definiciones de las macros de QL, que se utilizan para denotar la completitud del lenguaje. La implementación realizada respeta estrictamente la definición del mismo y admite definiciones de macros con expansiones en tiempo de precompilación. En consecuencia, esto aportó transparencia a expresiones de queries, como por ejemplo, los de cardinalidad. Se debería tomar en cuenta el concepto de queries no computables en las aplicaciones informáticas de bases de datos. Desde el punto de vista pedagógico, este intérprete contribuye a la construcción de una biblioteca de distintos tipos de formalismos, con diferentes grados de expresividad para formular consultas a base de datos relacionales.

SECCIÓN 1: Aspectos Teóricos

Considerando una base de datos como una representación simbólica de una realidad acotada, es deseable realizar consultas a la misma, con el fin de obtener información. Para ello, es necesario contar con un lenguaje que permita formular las consultas. Para el caso del *Modelo Relacional*, Codd [CO,70] desarrolló el *Álgebra Relacional (AR)* y el *Cálculo Relacional(CR)*, demostrando que ambos lenguajes son equivalentes en su poder expresivo, y más aún, equivalentes a la *Lógica de Primer Orden (FO)*. En función de ellos, Codd definió el concepto de *Completitud Relacional* como que dado un lenguaje cualquiera de consultas a bases de datos relacionales, el mismo se decía *relacionalmente completo*, si al menos podía expresar la misma clase de consultas, o *queries,* que el **AR** o **CR**.

A raíz de esto, surgieron inquietudes acerca de las características de tal clase, o más aún, saber cuál es la clase de todas las consultas posibles a una base de datos, y cómo se relacionan unas con otras.

Así Aho y Ullman [AU,79] demostraron que los queries de clausura no eran expresables ni en Álgebra ni en Cálculo; por ejemplo la clausura transitiva de un grafo, o preguntar si existe un paso de cualquier longitud entre un par de nodos dados, o pretender axiomatizar en **FO** la clase de grafos conectados. En consecuencia, el concepto de *completitud relacional* cambió radicalmente, demostrándose en este nuevo sentido que ni el **AR** ni el **CR** eran completos.

Para definir cuál es la clase de todos los queries computables, se consideró la clase de funciones efectivamente computables de Naturales en Naturales, es decir aquéllas para las cuales existe un algoritmo que las computa.

En virtud de la Tesis de Church, dichas funciones serían las recursivas parciales y también las computables por Máquinas de Turing. Pero Chandra y Harel [CH,80], observaron que los queries deben, además, preservar automorfismos. Ellos definieron el concepto de query computable, y notaron que la Máquina de Turing requiere una codificación de la base de datos en una estructura totalmente ordenada, para poder ser escrita en la cinta de input.

En conclusión, se determinó que ni la Máquina de Turing ni las funciones recursivas parciales, constituyen un formalismo correcto para definir la clase de queries computables.

En el marco de la Teoría de Modelos Finitos, las bases de datos relacionales son consideradas como clases de estructuras relacionales finitas, y las instancias de bases de datos relacionales como estructuras relacionales finitas. Desde este punto de vista, un query puede verse como una función cuyo dominio es el conjunto de las estructuras relacionales finitas de un cierto vocabulario, y cuyo codominio es el conjunto de las relaciones definidas en el dominio de la estructura relacional finita correspondiente para alguna aridad.

Por ello propusieron como definición de la clase de queries computables a la clase de funciones definidas en las clases de estructuras relacionales finitas, para cada vocabulario relacional finito, que son funciones recursivas parciales, en alguna codificación lineal de las estructuras, y preserva automorfismos en ellas.

A dicha clase se la llamó **CQ**, y como modelo formal para la computación de queries definieron un lenguaje de programación denominado **Query Language (QL)**.

Las aplicaciones clásicas de queries a bases de datos relacionales en Informática no toman en cuenta este concepto de query; considérense los siguientes ejemplos:

- Sea Q una consulta a base de datos que requiera gran cantidad de tiempo de procesamiento para su resolución, y que eventuales interrupciones pudieran ocurrir, en consecuencia sucesivos reinicios, entonces se define una computación "por etapas". Qué ocurre si entre etapa y etapa se realiza una reorganización de la base de datos a nivel de soporte magnético ? Se obtendría posiblemente un resultado incorrecto, por resultados parciales inconsistentes entre si.
- Sea Q una consulta a una base de datos distribuida en red, cuyo procesamiento implica que cada computadora de la red resolverá una parte, no necesariamente disjunta, de la misma. Qué ocurre si un mismo conjunto está representado en dos computadores de la red con distintos ordenamientos? También en este caso se obtendrían resultados parciales inconsistentes entre si.

Los lenguajes de consulta a base de datos actuales que se utilizan en el mercado, como por ejemplo **SQL**, computan queries no computables. SQL al definir cursores, frente a una consulta del tipo "déme el primero", responde entregando el elemento que físicamente está almacenado en las direcciones de memoria primero utilizadas. Es decir, utiliza el orden impuesto por las direcciones físicas de almacenamiento, para decidir cuál es el primer elemento de un conjunto, cuando nunca se puede considerar tal orden implícito entre los elementos del mismo, porque dicho orden no forma parte del esquema de la base de datos. En **QL,** realizar una consulta de este tenor, implica recibir como resultado una variable atómica conteniendo una relación, en consecuencia no es posible tomar una nupla bajo algún criterio de orden que no figure explicitamente en el esquema.

SECCIÓN 2: El lenguaje QL

2.1 Introducción

QL es un lenguaje de programación que computa relaciones finitas sobre un dominio finito dado. Los queries se aplican a una Base de Datos **B=(D,R_1,R_2,...,R_k)** de tipo **a=(a_1,a_2,...,a_k)**, donde **D** es el dominio y **R_1, R_2, ..., R_k** son relaciones, de aridad **a_1, a_2, ..., a_k** respectivamente.

Una característica muy importante del lenguaje **QL** es que usa variables de tipo "*relación*" y de aridad no fija, es decir, su contenido es siempre una relación de cualquier aridad (rango) y de cualquier cantidad de tuplas (cardinalidad), incluyendo aquellas de aridad y/o extensión vacía.

Las operaciones propias del lenguaje son: asignación, complemento, intersección, testeo de vacío, "project in" y "project out".

2.2 Sintaxis

Formalmente, el lenguaje **QL** está definido de la siguiente manera:

2.2.1 Términos

Sean **y_1 , y_2 , ... , y_n** variables, entonces:
y_i es un término, i >=1
E es un término
rel_i es un término, i >=1
si **e** y **e'** son términos
(e ∩ e') (¬ e) (e↓) (e↑) (e~) también lo son. (ver operadores)

2.2.2 Programas

El conjunto de programas se define inductivamente :

1. **y_i = e** , i >=1, es un programa.
2. si **P** y **P'** son Programas,
   (P ; P') y **While y_i do P**, i >=1, son Programas.

2.3 Operadores

A continuación se describen los operadores del lenguaje, con una breve descripción de su funcionamiento y ejemplos que muestran el resultado de aplicar los mismos a relaciones vacías (casos extremos) y a una base de datos BD definida así:

BD = { D, R_1, R_2 }; de tipo **a = (2, 2)**;

las relaciones que la componen, definidas por extensión, son las siguientes:

D = D^1 = { a, b, c }
R_1 = { (a, b), (a, c) }
R_2 = { (b, c), (c, b), (c, c) }

por propósitos de claridad, se incluye la extensión de la relación $\mathbf{D^2}$ resultante de aplicar la operación **project in** a **D.** Ver 2.3.4.

$$\mathbf{D^2 = \{ (a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c) \}}$$

. [CH,80], [BQM,96].

2.3.1 Intersección ($\cap$) :

Es el único operador binario. El valor computado es la intersección usual de conjuntos cuando el rango de ambos operandos es igual. Si el resultado de la intersección es vacía, devuelve ϕ^{i}, si los operandos son de distinto rango devuelve ϕ^{0}.

Ejemplos:

$\mathbf{D^0 \cap D^0 = D^0}$ $\qquad$ $\mathbf{D^0 \cap \phi^0 = \phi^0}$
$\mathbf{\phi^1 \cap D^1 = \phi^1}$ $\qquad$ $\mathbf{\phi^0 \cap D^1 = \phi^0}$
$\mathbf{R_1 \cap R_2 = \phi^2}$ $\qquad$ $\mathbf{R_1 \cap D^2 = R_1}$

2.3.2 Complementación ($\neg$) :

Operador unario. El resultado es equivalente a la diferencia entre la relación $\mathbf{D^i}$ y la relación a la cual se le aplica, o sea que la relación resultante tendrá las tuplas de $\mathbf{D^i}$ que no formaban parte de la relación original. Formalmente: $\mathbf{\neg e = D^i - e}$

Ejemplos:

$\mathbf{\neg \phi^2 = D^2}$
$\mathbf{\neg D^2 = \phi^2}$
$\mathbf{\neg \phi^0 = D^0 = \{( )\}}$
$\mathbf{\neg R_1 = \{(a,a), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c) \}}$

2.3.3 Project Out ($\downarrow$) :

Operador unario. La relación resultante de aplicar el operador está formada por las tuplas de la relación original **e**, a las cuales se les ha suprimido su primer componente. Se eliminan las tuplas repetidas.

Formalmente: $\mathbf{e\downarrow = \{ (d_2,..., d_i) \mid (d_1, d_2,..., d_i) \in e \}}$

Ejemplos:

$\mathbf{D^2\downarrow = D^1}$ $\qquad$ $\mathbf{D^1\downarrow = D^0}$
$\mathbf{D^0\downarrow = \phi^0}$ $\qquad$ $\mathbf{\phi^1\downarrow = \phi^0}$
$\mathbf{\phi^0\downarrow = \phi^0}$ $\qquad$ $\mathbf{R_1\downarrow = \{ (b), (c) \}}$
$\mathbf{R_2\downarrow = \{ (c), (b) \};}$

2.3.4 Project In (a derecha) ($\uparrow$) :

Operador unario. El resultado de su aplicación a una relación **e** es una nueva relación formada por el producto cartesiano de **e** con **D**, es decir, a cada tupla de la relación original se le agrega, en su extremo derecho, un elemento perteneciente a **D**, por lo tanto su cardinalidad es igual a la cardinalidad de **e** incrementada **| D |** veces.

Formalmente: $\mathbf{e\uparrow = \{ (d_1 ,..., d_i , d) \mid d \in D, (d_1,..., d_i) \in e \}}$

Ejemplos:

$\mathbf{\phi^0\uparrow = \phi^1}$ $\qquad$ $\mathbf{D^0\uparrow = D^1}$
$\mathbf{R_1\uparrow = \{(a,b,a), (a,c,a), (a,b,b), (a,c,b), (a,b,c), (a,c,c)\}}$

2.3.5 Exchange (~) :

Operador unario. Su aplicación a una relación **e** hace que se intercambien,en la relación resultado, los dos últimos elementos del extremo derecho de cada tupla perteneciente a la relación original, siempre que el rango de **e** sea mayor que 1, caso contrario, su aplicación no tiene efecto. Formalmente:

$$e\sim = \{ (d_1,..., d_{i-2} , d_i , d_{i-1}) / (d_1,...,d_{i-2}, d_{i-1}, d_i) \text{ e } e \} \quad i>1$$

ó $e\sim = e \quad i<=1$

Ejemplos:

$$\phi^{i}\sim = \phi^{y} \qquad D^{n}\sim = D^{n}$$

$$R_1\sim = \{ (b, a), (c, a) \}$$

SECCIÓN 3: Implementación de QL

La implementación consta de un ambiente de ejecución, donde el usuario a través de un conjunto de comandos puede instanciar una base de datos, ver el catálogo de sus relaciones, y el contenido de las mismas, también puede cargar desde un archivo o crear consultas escritas en **QL**, editarlas, ejecutarlas y ver sus resultados.

Al ejecutar una consulta, el código fuente es traducido en un código intermedio, el cual es ejecutado por un administrador de archivos propio que realiza las operaciones sobre la base de datos. Las relaciones, así como las variables de una consulta, constan de un descriptor en memoria y las tuplas que contienen se almacenan en archivos en el disco.

El principal problema de la implementación consiste en la elección de una gramática que represente fielmente la sintaxis y la semántica del **QL** sin agregado de tipos nuevos, estructuras de control, etc., que desvirtúen su carácter de lenguaje formal.

Por ejemplo, la proyección que se define como [CH,80]:

$$e_{[i1\ ...ip]} = (e \times D^{p}) \cap \bigcap_{1<=j<=p} (D^{i_j - 1} \times E(\uparrow\sim)^{rank(e)-i_j+j-1} \times D^{pj})) (\downarrow^{rank(e)})$$

se usa una lista de *relaciones* $_{[i1\ ...ip]}$ donde la cardinalidad de éstas indica qué columnas de **e** se quieren proyectar, pero en **QL** no existen tipos de listas ordenadas. Aquí la solución natural hubiera sido implementar un nuevo tipo estructurado de arreglo o lista de relaciones cuyos componentes pudieran ser referenciados por su posición, pero esto se descartó por el motivo antes mencionado y se resolvió el problema convirtiendo la lista de índices en un número, usando la codificación de Gödel [DW,83], para luego ser descompuesto mediante factorización, obteniendo nuevamente los índices a proyectar. Recordemos que **QL** no tiene tipos ni operaciones numéricas (el único tipo es la relación) entonces los números se representan por la aridad de una relación, y en base a los operadores **project in** y **project out** se construyen los operadores aritméticos sobre los números naturales. [CH,80].

En cuanto a la estructuración del lenguaje se adoptó la definición de funciones con pasaje de parámetros por valor, ampliando de hecho el **QL** para una precompilación, no existen variables globales, sólo las relaciones de la base de datos son de alcance global. Esto se definió así por las siguientes razones :

1. Si bien la definición de **QL** no permite definición de funciones, son usadas de hecho en el artículo de Chandra y Harel a modo de pseudo operaciones. Nótese en el siguiente ejemplo el uso del término $\mathbf{e\ (\uparrow\sim)^n}$ que se ha definido previamente como " *el valor de* y_2 *en el programa :*

$y_2 \leftarrow e$;
while $n \neq 0$ *do*
$(y_2 \leftarrow ((y_2 \uparrow) \sim) ; n \leftarrow n + 1)$"

luego es invocado como una función instanciando **e** con **E** y **n** con **rank(e)** :

.... $\mathbf{y_3} \leftarrow \mathbf{E(\uparrow \sim)}^{\mathbf{rank(e)}}$; .... [CH,80]

2. Para facilitar la escritura de consultas mediante el uso de bibliotecas de funciones de uso habitual.

3. No altera la especificación del **QL** dado que el resultado de la etapa de precompilación (por la expansión de las funciones), es siempre un programa **QL**.

3.1 Ejemplos

A continuación se desarrollan dos ejemplos con el fin de ilustrar el modo de programar en esta implementación de **QL**. El primero de ellos muestra como se evita la utilización de listas en el pasaje de parámetros, mediante el uso de la codificación de Gödel, el segundo es un query no es expresable en **FO** pero necesario para muchos queries como cálculo intermedio. Se pone en evidencia el hecho de que los operadores primitivos de **QL** son muy básicos, dado que su elección obedece a criterios teóricos y apunta a simplificar la demostración de completitud. De hecho, en [CH,80], se demuestra que ningún operador es redundante y, sin embargo, su poder computacional es superior al **AR** y al **CR**.

Ejemplo 1 :

Obtener la proyección de la columnas 1, 2 y 4 de la relación:

$$\mathbf{R_3} = \{(a, b, c, d), (c, b, a, b), (c, c, c, a) \}$$

$$\pi_{1, 2, 4} (\mathbf{R_3})$$

cuyo resultado es: y1= { (a, b, d), (c, b, b), (c, c, a) }.

La secuencia de instrucciones de QL ampliado que computa esta proyección es la siguiente:

```
#include biblio.qll
/*  Esta es una directiva al intérprete que incluye el código de las funciones usadas que se asumen
en el archivo "biblio.qll"  */
main( )          /* Funcion principal (debe estar presente en todo programa QL ) */
{
        i1 ← uno( );
        i2 ← inc( i1 );
        i3 ← inc(inc( i2 ));
/* i1, i2, i3, son relaciones que indican, con su aridad, las columnas a proyectar */

        p1 ← primo(D);        /* primer primo */
        p2 ← primo(E);        /* segundo primo */
        p3 ← primo( E↑ );     /* tercer primo */
/* Hasta aqui se obtienen tantas relaciones como columnas se quieran proyectar cuyos rangos
indican números primos mayores que uno */

        a1 ← exp(p1,i1);
        a2 ← exp(p2,i2);
        a3 ← exp(p3,i3);
        r1 ← multi(multi(a1,a2),a3);
```

/* el resultado de la codificación de Gödel para las relaciones usadas para indicar las columnas a proyectar de la relación R2, queda en **r1** */

```
        y1 ← proy(e, r1, 3);   /* la función proy realiza la proyección */
}.
```

Ejemplo 2 :

El siguiente programa **QL** aumentado, cuenta la cantidad de tuplas de una relación, en este caso del dominio (**D**).

```
#include biblio.qll              /* contiene las funciones a utilizar */
main( )
/* Cuenta en n la cardinalidad de D */
{
        y1 ← dec(E);  n ← E↓↓            /* a y1 le asigna D [BQM,96] */
        while not(y1) do
        {
                y2 ¬ inc(y1);
                y1 ¬distrel(y2)
                n ← inc(n);
        };
        y1 ← dec(y2)            /* y1 = perm(D) */
}.
```

3.3 Funciones usadas en los ejemplos

Las funciones presentadas aquí se encuentran, según se asume en los ejemplos, en un archivo llamado **biblio.qll**, usado como biblioteca general de funciones, el cual se anexa al programa que se desea ejecutar. La implementación del intérprete permite al usuario definir sus propias funciones e incluirlas tanto en esta biblioteca general como en alguna propia, creada por el mismo.

El listado de las funciones existentes en la biblioteca del **QL** ampliado y la gramática del lenguaje, estan disponibles en [BQM,96].

```
function distrel(e)
/* devuelve todas las tuplas de la relación e, cuyos componentes son distintos, en otro caso ϕ^K
donde k es igual a la cardinalidad de e */
{
        y1 ← e;  i ← uno( ); j ← uno( );
        while menor(i, rank(e)) do
        {
                while menor(j, rank(e)) do
                {
                        y1 ← if (menor(i,j), disttupla(y1, i, j), y1);
                        j ← inc( j );
                };
                i ← inc( i );
                j ← i;
        };
```

```
};

function disttupla(e, i, j)
/* devuelve las tuplas de e en las cuales los elementos indicados por las posiciones i y j son
distintos, en otro caso ϕ^k donde K es igual a la cardinalidad de e */
{
        y2 ← exp(D, resta(i, j));
        y3 ← pi_excha(E, resta(resta(i, j), uno( ));
        y4 ← exp(D, resta(e, j));
        x ← pcar(pcar(y2, y3), y4);
        y1 ← e ∩ ¬x;
};

function if(e, e1, e2)
/* Si e es ϕ^i devuelve e1 si no devuelve e2 */
{
        y2 ← E↓↓↓;
        while e do
        {
                y1 ← e1;
                e ←E;
                y2 ←E;
        };
        while y2 do
        {
                y1 ← e2;
                y2 ←E;
        };
};

function not(e)
/* si e es ϕ^i devuelve E si no E↓↓↓ lo cual equivale a ϕ^0 */
{
         y1 ← if(e, E, E↓↓↓);
};

function inc(n);
/* Retorna en y1 n+1 */
{
        y1 ← n↑;
};

function dec(n);
/* Retorna en y1 n-1 */
{
        y1 ← n↓;
};
```

```
function cero( );
/* Devuelve D^0 , una relacion de rango cero*/
{
        y1 ← E↓↓;
};

function uno( )
/* devuelve una relación de rango uno */
{
         y1 ← D;
};

function rank(e);
/* Devuelve una relación cuyo rango es igual al rango de e, aún si es ϕ^i. También se podría usar e
en lugar de rank(e), pero por razones de claridad se usa la función */
{
        if(e, y2 ← ¬e, y2 ← e);
        y1 ← cero( );
        while not(y2) do
        {
                y2 ← y2↓;
                y1 ← inc(y1)
        };
        y1 ← dec(y1)
};

function not_cero(n);
/* Si n= ϕ^i o n=D^i devuelve ϕ^0 (equivale a Verdadero); en otro caso devuelve E (equivale a Falso)
*/
{
        if(n, y1 ← n, if (n↓,  y1← E, y1← E↓↓↓));
};

function pi_excha(e, n);
/* e ( ↑~ )^n    calcula e con n columnas proyectadas hacia adentro a la izquierda de la última
columna */
{
        y1 ← e;
        while not_cero(n) do            /* while n ≠ 0 */
        {
                y1 ← ((y1↑)~);
                n  ← dec(n)
        };
};

function pil(e)
/* (↑e)  project in a la izquierda de una relación */
{
```

```
        y1 ← e↑;
        n  ← rank(e);
        while not_cero(n) do                    /* while n ≠ 0 */
        {
                y2 ← pi_excha(E, rank(e));
                y1 ← (y1↑ ∩ y2)↓;
                n  ← dec(n);
        };
};

function pirn(e,n)
/* ( e ↑ )^n  n project in a derecha de e */
{
        y1 ← e;
        while not_cero(n) do            /* while n ≠ 0 */
        {
                y1 ← y1↑;
                n  ← dec(n);
        };
};

function piln(e,n)
/* ( ↑e )^n  n project in a izquierda de e */
{
        y1 ← e;
        while not_cero(n) do            /* while n ≠ 0 */
        {
                y1 ← pil(y1);
                n  ← dec(n);
        };
};

function porn(e,n)
/* e(↓)^n  project out a la derecha de e n veces*/
{
        y1 ← e;
        while not_cero(n) do            /* while n ≠ 0 */
        {
                y1 ← y1↓;
                n  ← dec(n);
        };
};

function pcar(e1, e2);
/* (e1(↑^rank(e2)) ∩ (↑^rank(e1) )e2) devuelve el producto cartesiano de las relaciones e1 y e2 */
{
        y1 ← pirn(e1, rank(e2)) ∩ piln(e2, rank(e1));
};
```

```
function suma(e1,e2)
/* devuelve una relación cuyo rango es la suma de los rangos de e1 y e2 */
{
        y1 ← e1;
        while not_cero(e2) do                   /* while n ≠ 0 */
        {
                y1 ← y1↑;
                e2  ← e2↓;
        };
};

function resta(e1,e2)
/* devuelve una relación cuyo rango es la resta de los rangos de e1 y e2 */
{
        y1 ← e1;
        while not_cero(e2) do                           /* while n ≠ 0 */
        {
                y1 ← y1↓;
                e2  ← e2↓;
        };
};

function multipli(e1,e2)
/* devuelve una relación cuyo rango es la multiplicación de los rangos de e1 y e2 */
{
        y1 ← cero();
        while not_cero(e2) do                   /* while n ≠ 0 */
        {
                y1 ← suma(y1,e2);
                e2  ← e2↓;
        };
};

function divide(e1,e2)
/* devuelve una relación cuyo rango es la división entera de los rangos de e1 y e2 */
{
        y2 ← e1; y1 ← cero();
        while not_cero(inc(resta(y2,e2))) do    /* y2 >= e2 */
        {
                y2 ← resta(y2,e2);
                inc(y1);
        };
};

function mod(e1,e2)
/* devuelve una relación cuyo rango es el resto de dividir los rangos de e1 y e2 */
{
```

```
        y2 ← cero( ); y1 ← e1;
        while not_cero(inc(resta(y1,e2))) do
        {
                y1 ← resta(y1,e2);
                inc(y2);
        };
};

function exp(e1,e2)
/* devuelve una relación cuyo rango es igual al rango de e1 elevado al rango de e2 */
{
        y1 ← e1;
        while not_cero(e2) do
        {
                y1 ← multi(y1,y1);
                dec(e2);
        };
};

function union(e1,e2)
/* devuelve una relación es la unión de las relaciones e1 y e2 */
{
        y1 ← ¬(¬e1 ∩ ¬e2);
};

function es_primo(e)
/* Si e es primo devuelve D^0 (equivale a cero )*/
{
        y2 ← E;  y1 ← dividir(e, E);
        while not_cero(y2) do
        {
                y2 ← mod(e, y1);
                y1 ←dec(y1);
        };
};

function primo(n)
/* Devuelve el n_ésimo primo*/
{
        y1 ← E;  y3 ← n;
        while not_cero(y3) do
        {
                y2 ← E↓↓↓;
                while not_cero(y2) do
                {
                        y2 ← es_primo(y1);
                        y1 ←inc(y1);
                };
```

```
                y3 ←dec(y3);
        };
        y1 ←dec(y1);
};

function dif_cart(e1, e2)
/* devuelve una relación que es la diferencia cartesiana de las relaciones e1 y e2 */
{
        y1 ← e1 ∩ ¬e2;
};

function menor(x, y)
/* devuelve ϕ i si x es menor que y  o E si x es mayor o igual a y */
{
        x ← if(x, ¬x, x);
        y ← if(y, ¬y, y);
        y1 ←resta(x,y);
}

function inv(r,x)
/* Descompone la relación r y devuelve una relación cuyo rango es la x-ésima columna a proyectar
*/
{
        y1 ← cero();
        j ← uno();
        a1 ← r;
        while menor(j,x) do
        {
                r1 ← cero( );
                p_j  ← primo(j);
                while igual(r1, cero( )) do
                {
                        a1 ← div(a1, p_j)
                        r1 ← div(a1, p_j)
                        y1 ← inc(y1)
                };
                inc(j);
        };
};

function aux(e, i_j, j, p)
/* (D^(i_j -1) x E(↑~)^(rank(e)-i_j +j-1) x D^(p-j))*/
{
        x1 ← exp(D, resta(i_j, uno()),
        x2 ← pi_excha(E, suma(resta(rank(e),i_j), resta(j, uno())))
        x3 ← exp(D, resta(p,j))
        y1 ← procar(x1, procar(x2, x3))
}
```

```
function inter_gen(e, ri, p)
/* ∩1<=j<=p (D^(i_j -1) x E(↑~)^(rank(e)-i_j +j-1) x D^(p-j))*/
{
        j ← uno();
        i_j ← inv(ri, j);
        y1 ← aux(e, i_j, j, p);
        inc(j);
        while i<=j do
        {
                i_j ← inv(ri, j);
                y1 ← y1 ∩ aux(e, i_j, j, p);
                inc(j)
        };
}
```

```
function proy(e, ri, p)
/*e[i1 ..ip] =(e x D^p)∩ ∩1<=j<=p (D^(i_j -1) x E(↑~)^(rank(e)-i_j +j-1) x D^(pj))(↓^rank(e))
devuelve una relación de aridad p formada por las componentes i1, ... , ip de la relación e */
{
        y2 ← procar(e, exp(D,p)) ∩ inter_gen(e, ri, p);
        y1 ← porn(y2, rank(e));
}
```

Bibliografía

* [AHV,95] Abiteboul,S; Hull and Vianu, S.. "Foundations of Databases". Addison-Wesley Publishing Company, 1995.
* [BQM,96] Barroso, L.; Quiroga, J.; Molina, G. "Implementación de un Intérprete para el Lenguaje QL". Informe del Trabajo Final para la Licenciatura en Ciencias de la Computación. Universidad Nacional de San Luis. 1996.
* [CH,80] Chandra, A.K.; Harel, D. "Computable Queries for Relational Data Bases". Journal of Computer and System Sciences 21, 156-178. 1980.
* [DW,83] Davis, Weyuker, "Computability, Complexity and Languages", edición corregida, Academic Press, 1983.
* [Mai,83] Maier, D.. "The Theory of Relational Databases". Computers Science Press. 1983.
* [Tur,95] Turull Torres, José M. "De la computabilidad sobre Números Naturales a la computabilidad sobre Estructuras o Bases de Datos". Anales Infocom '95. 1995.
* [Ull,88] Ullman, Jeffrey D. "Principles of Database and Knowledge Base Systems". Computers Science Press, 1988.
* [Ull,79] Ullman, Jeffrey D. "Principles of Database Systems". Second Edition - Computers Science Press.

––*––