Verificación Estática de Propiedades en Bases de Datos

Tesina de grado presentada
por

Damián Soriano
S-2992/1

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

Octubre 2009

Supervisores

Ralph-Johan Back Mikolaj Olszewski

Åbo Akademi University

Turku, Finlandia

Resumen

Los motores de base de datos son actualmente empleados como componentes esenciales en el desarrollo de sistemas informáticos. Inicialmente utilizados para el almacenamiento físico de información, en la actualidad ofrecen funcionalidades avanzadas que distan mucho de sus versiones iniciales.

Entre estas funcionalidades se encuentran los *triggers* o disparadores, que ofrecen una alternativa eficiente para la incorporación de reglas de negocio en la base de datos. De esta manera es posible delegar responsabilidades que inicialmente son consideradas parte de la aplicación. Como cualquier componente en el desarrollo de software, es deseable garantizar la corrección de esta pieza del sistema.

El objetivo de este trabajo es proponer una técnica para la verificación formal de propiedades de las operaciones de base de datos con la incorporación de *triggers*. Para esto se crea un especificación de las operaciones, en el lenguaje PVS, a partir de las definiciones de los disparadores. Finalmente, este modelo es utilizado para la demostración de propiedades deseadas.

Agradecimientos

A mis viejos, Raúl y Adriana, dos personas extraordinarias. Por enseñarme desde chico el valor de la educación. Por escucharme cuando necesitaba un consejo y acompañarme cuando necesitaba apoyo. A ellos más que a nadie.

A mis hermanos, Iván, Diego, Romina y Agustin. Por las alegrías contidianas, por hacer que cada día sea mejor.

A Mercedes, por su inmenso afecto. Por su incondicional compañía, por estar conmigo cuando lo necesitaba, sin importar las distancias.

A mis amigos de siempre, a cada uno de ellos. Por su constante presencia y apoyo.

A mis amigos de la facultad, a todos. Porque fue con ellos con quienes transite este camino.

A mis profesores, por hacer de la educación pública una realidad. A Guido, Maximiliano y Gabriela, por ser mucho más que profesores.

A mis supervisores, Ralph y Miki. Por darme esta oportunidad.

Índice general

1. Introducción **1**
1.1. Sobre las Bases de Datos . . . 1
1.1.1. Sobre el uso de triggers . . . 2
1.2. PostgreSQL . . . 2
1.2.1. PL/pgSQL . . . 3
1.3. Sobre PVS . . . 3
1.3.1. El lenguaje de especificación de PVS . . . 4
1.3.2. El lenguaje de pruebas de PVS . . . 5

2. Manejador de reuniones: Un caso de estudio **7**
2.1. Presentación informal . . . 7
2.2. Diseño e implementación de *Deseo* . . . 8
2.3. Modelado . . . 9
2.3.1. Estructura de la Base de Datos . . . 10
2.3.2. Operaciones simples . . . 12
2.3.3. Operaciones complejas . . . 18
2.4. Algunos teoremas y demostraciones . . . 25

3. Generalización **29**
3.1. Definiciones . . . 29
3.2. Declaración de los tipos principales . . . 33
3.2.1. Tablas . . . 33
3.2.2. Base de Datos . . . 34
3.2.3. Selector . . . 35
3.2.4. Actualizador . . . 36
3.3. Operaciones básicas . . . 37
3.3.1. Select . . . 37
3.3.2. Delete . . . 38
3.3.3. Update . . . 41
3.3.4. Insert . . . 44
3.4. Incorporando triggers . . . 46
3.4.1. Definiciones . . . 46
3.4.2. Reglas modificadas . . . 47

3.4.3. Reglas de triggers 49

4. Conclusiones y trabajos futuros **52**

Capítulo 1

Introducción

Este trabajo esta basado mayoritariamente en tres componentes de software. Los Motores de Base de Datos son la pieza principal de estudio, en particular el motor PostgreSQL por las funcionalidades que ofrece. Sin embargo las conclusiones pueden ser traducidas a otros motores sin mayores modificaciones. El lenguaje de programación PL/pgSQL, utilizado para la definición de triggers sobre las operaciones de Base de Datos es el segundo componente importante de este trabajo. Finalmente, el lenguaje de especificación PVS es utilizado para crear las especificaciones de las operaciones y para probar propiedades de las mismas.

1.1. Sobre las Bases de Datos

Una Base de Datos es un repositorio común en el cual se guarda información organizada en registros o archivos. Estos repositorios ofrecen acceso a los datos a múltiples usuarios a través de operaciones bien definidas. Nos focalizaremos en este trabajo, en aquellas Bases de Datos que implementan el llamado modelo relacional. Este modelo esta basado en la lógica de predicado de primer orden y fue propuesto por E. F. Codd en 1969.

El modelo relacional asume que la información es almacenada en relaciones matemáticas, donde el orden de cada relación esta dado por la cantidad de datos que poseen los registros que van a ser guardados. Estas relaciones son vistas conceptualmente como tablas, cuyas columnas son los campos que tendrá cada registro que pertenezca a la misma.

El lenguaje SQL fue definido inicialmente como el lenguaje estándar de las bases de datos relacionales. Las operaciones que pueden realizarse sobre las tablas son las de lectura, escritura, borrado y actualización de registros. Estas operaciones son conocidas como *Select*, *Insert*, *Delete* y *Update* respectivamente, en el estándar SQL.

Para más información sobre las Bases de Datos y el modelo relacional se puede consultar [4, 12].

1.1.1. Sobre el uso de triggers

Un disparador o *trigger* de Base de Datos es un bloque de código que es ejecutado automáticamente, en respuesta a un determinado evento relacionado con una tabla, una vista o con la misma Base de Datos. Comúnmente son utilizados para inhibir cambios no deseados, para almacenar información de auditoría, para generar entradas en un historial de cambios o para forzar o ejecutar reglas de negocio de una aplicación en particular.

Si bien los *triggers* pueden ser definidos sobre el lenguaje de definición de datos (DDL del ingles *Data Definition Language*), en este trabajo nos centraremos en aquellos definidos sobre el lenguaje de manipulación de datos (DML del ingles *Data Manipulation Language*).

Los *DML triggers* son ejecutados en respuesta a alguna de las operaciones de manipulación de datos: *Insert*, *Delete* o *Update*. Pueden ser declarado a nivel de registros, en cuyo caso son lanzados por cada registro que fue afectado por la operación, o a nivel de procedimientos, en cuyo caso son lanzados una sola vez por cada operación a la que fue asociada, sin importar el número de registros involucrados.

Adicionalmente, cada *trigger* puede ser declarado como un *before trigger*, que es disparado antes de que la operación sea ejecutada, o un *after trigger*, que es lanzado luego de que la operación tuvo lugar. Existen también *instead of triggers* que remplazan el comportamiento normal de la operación sobre la cual se definen.

Es importante aclarar que los triggers son funciones que no aceptan parámetros, sino que reciben la información de variables globales implícitas. Para los triggers definidos a nivel de registro existen las variables *New* y *Old* que contienen el registro ya modificado y el registro antes de ser modificado respectivamente. Para los triggers definidos a nivel de procedimiento se reciben dos variables con la tabla antes y después de que la operación haya sido ejecutada.

Utilizaremos en este trabajo solo los triggers definidos a nivel de registros, para incorporar reglas de negocio de la aplicación a la Base de Datos.

1.2. PostgreSQL

PostgreSQL [5, 6] es un sistema de gestión de Bases de Datos relacional desarrollado en el Departamento de Ciencias de la Computación de Berkeley de la Universidad de California, distribuido bajo la licencia BSD. PostgreSQL implementa una gran parte del estándar SQL como consultas complejas, triggers, vistas, integridad transaccional, etc.

PostgreSQL comenzó con el nombre de POSTGRES [13] en el año 1986 impulsado por el profesor Michael Stonebraker y en 1988 se presento la primer versión utilizable en la Conferencia ACM-SIGMOD. Se liberaron distintas versiones hasta que en 1994, Andrew Yu y

Jolly Chen agregaron un interprete de SQL y el producto fue liberado con el nombre de Postgres95. En 1996 el sistema fue relanzado bajo el nombre de PostgreSQL para rescatar la relación entre el proyecto original POSTGRES y las nuevas versiones con soporte SQL. Desde entonces PostgreSQL ha continuado su crecimiento e incorporación de nuevas tecnologías.

PostgreSQL ofrece una gran cantidad de funcionalidades como tolerancia a una alta cantidad de operaciones concurrentes, una amplia variedad de tipos de datos nativos, triggers y funciones que pueden ser implementadas en distintos lenguajes soportados por el motor.

Entre los distintos lenguajes para definir procedimientos que se encuentran en PostgreSQL, nos focalizaremos en PL/pgSQL que es el que utilizaremos a lo largo de este trabajo.

1.2.1. PL/pgSQL

PL/pgSQL (*Procedural Language/PostgreSQL Structured Query Language*) es uno de los lenguajes de programación procedural incorporado en el Motor de Base de Datos PostgreSQL para la definición de funciones. Los objetivos de este lenguaje son:

- Permitir la implementación de funciones y triggers.
- Añadir control de flujo al lenguaje SQL.
- Permitir la definición de operaciones complejas.
- Acceso a las funciones, tipos y operadores definidos por el usuario.

Las funciones creadas en PL/pgSQL pueden ser utilizadas en cualquier lugar donde se puedan utilizar funciones nativas del Motor de Base de Datos.

SQL es el lenguaje soportado por PostgreSQL y la mayoría de los Motores de Base de Datos para la manipulación de los datos. PL/pgSQL ofrece una extensión al estándar SQL, permitiendo agrupar bloques de operaciones dentro del servidor de de Base de Datos. Las funciones implementadas con PL/pgSQL aceptan argumentos escalares o registros de la Base de Datos, pueden devolver valores de cualquiera de estos tipos también.

Utilizaremos este lenguaje para la definición de funciones que serán declaradas como triggers de distintas operaciones para la incorporación de reglas de negocio en el Motor de Base de Datos.

1.3. Sobre PVS

PVS [10] es un Sistema de Verificación de Prototipos (*Prototype Verification System*) utilizado para el desarrollo y análisis de especificaciones formales. Consta de un lenguaje de

especificación, un conjunto de teorías predefinidas, un chequeador de tipos y un demostrador de teoremas interactivo.

PVS fue desarrollado utilizando Common Lisp, en el Laboratorio de Ciencias de la Computación del Stanford Research Institute, California, Estados Unidos. La última versión al momento de la redacción es la 4.2, distribuido bajo la licencia GPL.

1.3.1. El lenguaje de especificación de PVS

El lenguaje de especificación de PVS esta basado en la lógica clásica de alto orden tipada. Cuenta con ciertos tipos básicos predefinidos como booleanos, enteros y reales, permitiendo la incorporación de nuevos tipos por parte del usuario.

El lenguaje contiene una serie de constructores de tipos incorporados como funciones, conjuntos, tuplas, registros, enumeraciones; tipos de datos definidos inductivamente, como listas o arboles; y tipos de datos definidos co-inductivamente como *streams*.

El sistema de tipos de PVS permite el tratamiento de subtipos y tipos dependientes para la introducción de restricciones. Estas restricciones pueden incurrir en obligaciones de pruebas durante el proceso de chequeo de tipos que deberán ser probadas por parte del usuario para garantizar la consistencia. En la practica la mayoría de estas obligaciones de pruebas son demostradas automáticamente.

Las especificaciones son organizadas en teorías parametrizadas que pueden imponer restricciones sobre los parámetros de las mismas. Las teorías pueden contener definiciones, axiomas, teoremas y pueden ser definidas sobre otras teorías.

Se debe garantizar la terminación de las definiciones que se realizan dentro de una teoría, es por esto que las definiciones recursivas de funciones generan obligaciones de pruebas de terminación que deberán ser demostradas por el usuario.

PVS cuenta con un *parser* encargado de asegurar la consistencia sintáctica de las teorías. El *parser* construye una representación interna de la teoría que es utilizada por los demás componentes del sistema.

El chequeador de tipos de PVS analiza las teorías para garantizar consistencia semántica. El sistema de tipos de PVS no es algorítmicamente decidible y por lo tanto, se generan obligaciones de pruebas, llamadas condiciones de corrección de tipo (o TCCs por *Type-Correctness Conditions*) que deberán ser demostradas por el usuario.

Puede consultarse [7, 8] para una definición formal de la sintaxis y semántica del lenguaje de especificación de PVS.

1.3.2. El lenguaje de pruebas de PVS

Las demostraciones de teoremas en PVS son realizadas de forma interactiva bajo la guía del usuario. Para esto ofrece una colección de procedimientos primitivos que incluyen un poderoso conjunto de comandos para el razonamiento proposicional, ecuacional y aritmético. Estos comandos pueden ser agrupados en estrategias de pruebas definidas por parte del usuario.

La combinación de una lógica expresiva y la potente capacidad de prueba del demostrador de PVS generan una fuerte integración entre el chequeador de tipos y el chequeador de pruebas. El chequeador de tipos explota la capacidad de deducción del chequeador de pruebas para eliminar las obligaciones de pruebas (o TCCs) que este genera, tratando de maximizar la automatización del chequeo de tipos.

El chequeador de pruebas de PVS es interactivo, aunque soporta un modo en el cual las pruebas almacenadas son corridas automáticamente. El demostrador mantiene un árbol de prueba y es responsabilidad del usuario construir uno que sea completo. Cada nodo en el árbol de pruebas es consecuencia de otro nodo a través de un paso de prueba. Cada nodo esta formado por una lista de antecedentes y una de consecuentes que son mostrados por PVS de la siguiente forma:

$$\begin{array}{cc}
\{-1\} & A_1 \\
\{-2\} & A_2 \\
\{-3\} & A_3 \\
 & \vdots \\
\hline
\{1\} & B_1 \\
\{2\} & B_2 \\
\{3\} & B_3 \\
 & \vdots
\end{array}$$

Donde el conjunto de A_i son antecedentes y B_j son consecuentes. Los números que acompañan a cada formula son utilizados para distinguir sobre cual se aplica una estrategia. La interpretación de la secuencia de antecedentes y consecuentes es la siguiente:

$$(A_1 \wedge A_2 \wedge A_3 \ldots) \Rightarrow (B_1 \vee B_2 \vee B_3 \ldots)$$

El árbol de prueba comienza con una raíz de la forma $\vdash B$, donde B es el teorema que se desea demostrar. Cada paso de prueba añade un subárbol de prueba al nodo al cual se aplica el paso. Cuando un nodo es reconocido como verdadero, a través de alguna estrategia utilizada por PVS, la rama a la que pertenece es considerada verdadera y se prosigue con la siguiente rama del árbol. Se trata de construir un árbol de pruebas en el que todas las ramas terminen de esta forma, en cuyo caso se reconoce al teorema como tal.

Los pasos de pruebas de PVS pueden causar que un nodo genere dos subárboles de pruebas. Por ejemplo, aplicando el comando **split** al nodo

$$\Gamma \vdash A \wedge B$$

Se generan los siguientes nodos de prueba

$$\Gamma \vdash A \quad y \quad \Gamma \vdash B$$

La lista de comandos de pruebas y las funcionalidades del chequeador de pruebas son extensas y potentes, para más información sobre el lenguaje de pruebas de PVS se puede consultar [9].

Capítulo 2

Manejador de reuniones: Un caso de estudio

En este capítulo se describe en detalle el caso de estudio utilizado para realizar los primeros intentos en la formalización y traducción de las operaciones básicas y triggers correspondientes a las Bases de Datos.

Comenzaremos haciendo una breve introducción a la aplicación propuesta. Luego se presentará una traducción tentativa para finalmente poder atacar el problema de la verificación de diversas propiedades. Este enfoque servirá de punto de partida para las generalizaciones propuestas en el capítulo 3.

El contenido de este capítulo fue tomado de [1].

2.1. Presentación informal

El Manejador de Reuniones *Deseo* es una aplicación que facilita a sus usuarios la organización y planeamiento de reuniones. A través de esta, los usuarios pueden proponer nuevas reuniones que les interesen realizar y opinar sobre reuniones creadas por otros usuarios.

Un usuario cualquiera del sistema puede crear una nueva reunión, en cuyo caso se convierte en el anfitrión de la misma. Para su creación se debe especificar un titulo explicativo y una duración aproximada que tendrá la reunión. Una vez creada una reunión, el anfitrión puede realizar distintas propuestas para permitir a otros usuarios opinar sobre la fecha y hora en que tendrá lugar la misma.

De esta manera una reunión no contiene la fecha y hora prefijada sino que existen muchas propuestas para una sola reunión y son los usuarios los que deciden en que momento tendrá lugar.

El anfitrión de una reunión puede extender invitaciones a distintos usuarios, expresando el deseo de que estos asistan a la misma. Un usuario invitado a una reunión puede aprobar o aceptar una propuesta en caso de que desee que la reunión se realice según lo especifica la propuesta. En caso de que un usuario no este conforme con una propuesta realizada sobre una reunión a la cual fue invitado, este puede desaprobarla o rechazarla.

Si en algún momento, todos los invitados a una reunión hubiesen aceptado una propuesta en particular, el anfitrión puede confirmarla. Esto significa que la reunión se llevara a cabo en la fecha y hora especificada por la propuesta confirmada. Se dice que una reunión esta confirmada si existe alguna propuesta confirmada asociada a esta reunión. Nótese que una reunión puede tener a lo sumo una sola propuesta confirmada.

No es deseable que un usuario pueda aceptar una propuesta si esta entra en conflicto con una propuesta a una reunión ya confirmada. Si un usuario debe asistir a una reunión que ya fue confirmada se debe rechazar cualquier intento del mismo a aceptar una propuesta que entre en conflicto con la fecha y hora de la reunión confirmada a la que debe asistir.

El Manejador de Reuniones *Deseo* no es una aplicación de alta complejidad, pero como veremos más adelante, posee ciertas propiedades no triviales que nos serán de utilidad para estudiar la formalización de ciertas funcionalidades de las Bases de Datos en general.

2.2. Diseño e implementación de *Deseo*

Con una idea informal de lo que se desea del Manejador de Reuniones, podemos identificar distintos elementos para avanzar hacia una descripción más concreta del mismo. Se identifican entonces los siguientes elementos:

Users Usuarios del sistema.

Meetings Reuniones creadas por los usuarios. Una reunión contiene un usuario asociado (el anfitrión de la reunión), un titulo y una duración aproximada.

Invitations Invitaciones a los usuarios para asistir a una determinadas reuniones. Una invitación concreta asocia un usuario con la reunión a la cual ha sido invitado.

Proposals Distintas propuestas para las distintas reuniones. Una propuesta esta asociada a una reunión, tiene una fecha y hora tentativa y contiene información sobre si esta o no confirmada.

Answers Respuestas de los usuarios a distintas propuestas. Una respuesta esta asociada al usuario que la emitió y a la propuesta sobre la cual se respondió. Contiene también información sobre si la respuesta fue positiva o negativa.

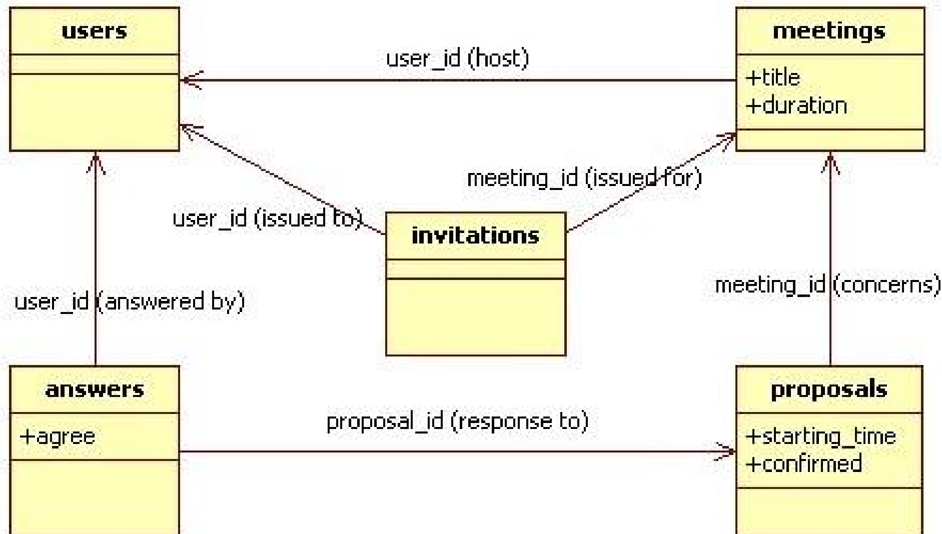

Figura 2.1: Modelo Relacional del Manejador de Reuniones *Deseo*

En la figura 2.1 se muestra un diagrama del modelo relacional con los elementos descriptos anteriormente.

Definida la estructura básica de los datos se deben definir las restricciones y comportamientos de los mismos. Estos son descriptos en la tabla 2.1.

Además de las restricciones descriptas anteriormente, existen restricciones asociadas a la integridad referencial de los datos.

La mayor parte de la estructura y restricciones aquí descriptas son implementadas en el Motor de Base de Datos, particularmente en PostgreSQL. Cada tipo de elemento es implementado como una tabla en la Base de Datos, junto a sus respectivas referencias y restricciones.

Las restricciones y comportamientos más complejos que no son contemplados en el modelo relacional puro, son implementados utilizando el mecanismo de triggers que permiten incorporar una mayor funcionalidad al Motor de Base de Datos. Ejemplos de este tipo de restricciones y comportamientos, son los descriptos en la tabla anterior.

2.3. Modelado

En esta sección introduciremos algunas definiciones de las tablas y triggers utilizadas para la implementación del Manejador de Reuniones que nos permitirán ilustrar las técnicas empleadas para la creación del modelo. Utilizaremos a lo largo de esta sección la tabla invitaciones como ejemplo ya que contiene ciertas características y propiedades no triviales que nos ayudaran a exponer las ideas que se desea mostrar.

Users	Ninguna.
Meetings	Existe solamente un anfitrión por reunión.
Proposals	No pueden existir dos propuestas para la misma reunión que contengan la misma fecha y hora de comienzo. Cada propuesta debe estar marcada como no confirmada cuando se crea. No pueden crearse propuestas con fechas pasadas. Se puede confirmar una propuesta solamente si todos los invitados aceptaron la misma y no entra en conflicto con otras reuniones confirmadas del anfitrión.
Invitations	El anfitrión no puede ser invitado a una reunión. Un usuario no puede tener dos invitaciones a la misma reunión. Borrada una invitación se deben borrar las respuestas del usuario a propuestas de la reunión.
Answers	Solamente un usuario invitado puede dar respuestas sobre alguna propuesta. Un usuario no puede generar dos respuestas para una propuesta. Un usuario no puede aceptar una propuesta si esta colisiona con una reunión confirmada a la cual fue invitado.

Cuadro 2.1: Restricciones y Comportamiento de los Datos

2.3.1. Estructura de la Base de Datos

Comencemos con la definición de los datos que se encuentran presente en *Deseo*. Cada dato del Manejador de Reuniones, como fue visto en las secciones anteriores, es implementado como una tabla del Motor de Base de Datos. Definamos entonces las tablas de usuarios y reuniones, que serán usadas por la tabla de invitaciones, en SQL estándar.

```
CREATE TABLE users(
  id integer primary key,
  name varchar(100) NOT NULL
  password varchar(100) NOT NULL
);

CREATE TABLE meetings(
  id integer primary key,
  user_id integer REFERENCES users(id) ON DELETE CASCADE NOT NULL,
  title varchar(100) NOT NULL
  duration int NOT NULL
);
```

Ahora podemos definir la tabla de las invitaciones extendidas a los usuarios para las distintas reuniones de la siguiente forma:

```
CREATE TABLE invitations(
  id integer primary key,
  user_id integer REFERENCES users(id) ON DELETE CASCADE NOT NULL,
  meeting_id integer REFERENCES meetings(id) ON DELETE CASCADE NOT NULL
);
```

Podemos observar que la clave primaria definida en la tabla `invitations` identifica unívocamente cada fila de la misma. Es decir, dada una clave primaria (de tipo entero) que se encuentre presente en la tabla, se puede obtener una única fila. Nótese que este comportamiento es similar a la definición matemática del concepto de función parcial. Usaremos esta intuición como fundamento de nuestra traducción.

Definimos primero un tipo de datos al cual pertenecen los valores de clave primaria, que en nuestro caso serán del tipo entero, junto a una función que nos indicara si una determinada clave primaria esta o no presente en la tabla. Para poder extraer los identificadores de usuario y reunión de una invitación, definimos un par de funciones que dado un identificador de invitación nos retornen los valores correspondientes. Los campos y columnas de la tabla `invitations`, quedan definidos en PVS de la siguiente forma:

```
invitationsId: TYPE = nat
invitations_id_type: set[invitationsId]
invitations_user_id_type: [invitationsId -> usersId]
invitations_meeting_id_type: [invitationsId -> meetingsId]
```

Finalmente, la tabla `invitations` es exactamente la agrupación de funciones cuyo tipos son los descriptos anteriormente. La tabla de las invitaciones se puede representar con una función que indique que valores de clave primaria están presentes en la tabla, junto a un par de funciones que retornen el usuario y la reunión asociada a cada invitación. En PVS esto se define como se muestra a continuación:

```
invitations_table_type: TYPE =
[# idnt: invitations_id_type,
   user_id: invitations_user_id_type,
   meeting_id: invitations_meeting_id_type
#]
```

Los tipos de datos `userId` y `meetingsId`, utilizados en la traducción de la tabla `invitations` son definidos cuando se realiza la traducción de las tablas `users` y `meetings` respectivamente.

De forma similar se definen las tablas `users_table_type`, `meetings_table_type`, `proposals_table_type` y `answers_table_type` que capturan la estructura de las demás tablas de *Deseo*, con sus correspondientes tipos y funciones. Por último, definimos la Base de Datos como el conjunto de todas las tablas.

```
database_type: TYPE =
[# users: users_table_type
   meetings: meetings_table_type
   invitations: invitations_table_type
   proposals: proposals_table_type
   answers: answers_table_type
#]
```

2.3.2. Operaciones simples

Las tablas describen la estructura de la Base de Datos, el modelo de datos a utilizar. Sin embargo la estructura no impone las restricciones deseadas ni agrega el funcionamiento requerido. Estos conceptos están relacionados a la manipulación de los datos y no a la forma en que se estructuran. Las operaciones que pueden ser realizadas sobre una tabla, aquellas que permiten la manipulación de sus datos, son cuatro: *Select*, *Delete*, *Update* e *Insert*.

Comenzaremos describiendo y analizando estas cuatro operaciones básicas para luego incorporar ciertos triggers a las mismas y así lograr la funcionalidad requerida.

Select

El operador *Select* es utilizado para extraer ciertos registros de una tabla en la base de datos, utilizando un predicado sobre los campos de la misma para decidir cuales son las filas a seleccionar y cuales no.

Para la definición de este operador, vamos a necesitar un tipo de datos que capture esta idea, la idea de un seleccionador. Definimos entonces un tipo de datos función, cuyo dominio sean las filas de la tabla `invitations` y codominio los booleanos, esto es, un predicado sobre las filas de la tabla `invitations`.

```
selector_invitations_type: TYPE =
        [[invitationsId, usersId, meetingsId] -> bool]
```

Como un seleccionador puede hacer referencias a información de otras tablas, este necesita una referencia a la base de datos sobre la cual se esta trabajando. Para esto, agregamos a la definición del tipo un parametro con la referencia correspondiente:

```
selector_invitations_type: TYPE =
        [[invitationsId, usersId, meetingsId, database_type] -> bool]
```

Por lo tanto, cualquier función cuyo tipo sea `selector_invitations_type`, será un predicado de selección en la tabla `invitations` que nos permitirá seleccionar ciertas filas pertenecientes a la tabla.

La operación de selección sobre la tabla `invitations` tomará una base de datos en un instante determinado, junto a un seleccionador. Su resultado será, una tabla con los mismos campos que la tabla `invitations` pero solo con aquellos registros que satisfacen el predicado del seleccionador. Definimos en PVS la siguiente función que captura la funcionalidad del operador *Select* sobre la tabla `invitations`:

```
select_invitations(db: database_type,
                   selector: selector_invitations_type):
                                         invitations_table_type =
  LET invitations' =
       (# idnt := {x: invitationsId | member(x, idnt(invitations(db)))
                       AND selector(x, user_id(invitations(db))(x),
                                  meeting_id(invitations(db))(x), db)},
          user_id := user_id(invitations(db)),
          meeting_id := meeting_id(invitations(db)) #)
  IN
  invitations'
```

Obsérvese que la tabla resultante solo se diferencia de la original en el campo que contiene las claves privadas de la tabla, la selección retorna un subconjunto de las filas de la tabla original. Esto es así ya que solo se puede acceder a los campos de usuario y reunión a través de una clave primaria si ésta se encuentra en la tabla, si se encuentra en el campo `idnt`.

Para ilustrar mejor la definición del operador de selección, supongamos que se desean obtener todas las invitaciones extendidas al usuario "Ilka". Para esto, definimos el siguiente predicado:

```
selector_Ilka(inv: invitationId, usr: userId,
                 meet: meetingId, db: database_type): bool =
           (login(users(db))(usr) = Ilka)
```

Las invitaciones extendidas al usuario "Ilka" en la base de datos `db` se extraen aplicando el operador de selección sobre la base de datos y el predicado de selección antes definido.

```
tbl = select_inviations(db, selector_Ilka)
```

Delete

El operador *Delete* es utilizado para eliminar registros de una tabla, haciendo uso de un predicado sobre los campos de la misma para decidir que registros son eliminados, al igual que el operador *Select*. Nótese que este operador realiza exactamente lo contrario al operador *Select*, descarta las filas que satisfacen un predicado. Utilizado el tipo de datos `selector_invitations_type` declarado anteriormente, definimos el operador *Delete* de la siguiente forma:

```
delete_invitations(db: database_type,
                   selector: selector_invitations_type):
                                              database_type =
  LET invitations' =
       (# idnt := {x: invitationsId | member(x, idnt(invitations(db)))
                       AND NOT selector(x, user_id(invitations(db))(x),
                                  meeting_id(invitations(db))(x), db)},
          user_id := user_id(invitations(db)),
          meeting_id := meeting_id(invitations(db)) #),
      db' = db WITH [invitations := invitations']
  IN
  db'
```

La operación *Delete* retorna una nueva base de datos sin las filas seleccionadas por el predicado de selección, a diferencia de la función *Select* que retorna solamente la tabla involucrada.

A modo de ejemplo, supóngase que se desea eliminar la invitación al usuario "Olli" para asistir a la reunión titulada "Kalevala" de la Base de Datos `db`, se define de la siguiente forma:

```
delete_Olli2Kalevala(inv: invitationId, usr: userId,
                 meet: meetingId, db: database_type): bool =
           (login(users(db))(usr) = Olli
            AND title(meetings(db))(meet) = Kalevala)

db' = delete_invitations(db, delete_Olli2Kalevala)
```

Update

El operador *Update* es utilizada para modificar ciertos campos de información de una tabla, haciendo uso de un predicado sobre los campos de la misma para decidir que filas son modificadas. Sin embargo, este operador, además de una función de selección, debe utilizar una función que especifique la forma de alterar los campos de cada registro seleccionado. Una función que tome la información de una fila y devuelva los cambios que deben ser realizados en cada campo.

Para la definición de esta operación, vamos a necesitar un tipo de datos que capture esta idea, la idea de un actualizador de fila. Definimos entonces un tipo de datos función, cuyo dominio sean las filas de la tabla `invitations` junto a la base de datos y codominio aquellos campos que puedan ser alterados.

```
updater_invitations_type: TYPE =
                 [[invitationsId, usersId, meetingsId, database_type]
```

```
                                                    -> [usersId, meetingsId]]
```

La operación de actualización sobre la tabla `invitations` debe tomar una base de datos, un predicado de selección y una función de actualización para aquellos registros seleccionados. Su resultado será, una nueva Base de Datos con los valores actualizados. Mostramos a continuación nuestro primer intento en definir esta operación:

```
update_invitations(db: database_type,
                   selector: selector_invitations_type,
                   updater: updater_invitations_type):
                                            database_type =
  LET new_user_id(inv: invitationsId): usersId =
         LET user = user_id(invitations(db))(inv),
             meeting = meeting_id(invitations(db))(inv)
         IN
         (COND NOT selector(inv, user, meeting, db) -> user,
               ELSE -> proj_1(updater(inv, user, meeting, db))
         ENDCOND),
      new_meeting_id(inv:invitationsId): meetingsId
         LET user = user_id(invitations(db))(inv),
             meeting = meeting_id(invitations(db))(inv)
         IN
         (COND NOT selector(inv, user, meeting, db) -> meeting,
               ELSE -> proj_2(updater(inv, user, meeting, db))
         ENDCOND),
      new_invitations = (# idnt := idnt(invitations(db)),
                           user_id := new_user_id,
                           meeting_id := new_meeting_id #)
      db' = db WITH [invitations := new_invitations]
  IN
  db'
```

Se puede observar que se definen dos nuevas funciones correspondientes a aquellas que conforman la lista de invitados. Estas dos funciones serán las que definan la nueva tabla de invitaciones luego de su actualización, como resultado de la operación *Update*.

Por cada fila de la tabla `invitations`, la función `new_user_id` devolverá el usuario original en caso de que la fila no sea seleccionada para actualización, o el usuario actualizado en caso de que si haya sido seleccionada. La función `new_meeting_id` se comportara de forma similar pero sobre las reuniones de las invitaciones. La nueva Base de Datos cuenta con la tabla `invitations` actualizada de acuerdo a las funciones definidas anteriormente.

Sin embargo, esta función de actualización no se comporta correctamente. Esto se debe a la integridad referencial de los campos `user_id` y `meeting_id` de la tabla `invitations`.

No se realizan las verificaciones pertinentes sobre estos campos, es decir, no se verifica si la actualización de un usuario actualiza a un usuario existente en la tabla de usuarios.

Para esto, se debe pedir, antes de realizar la actualización, que el nuevo usuario y reunión al cual se desea actualizar, exista en la tabla pertinente.

Se define entonces una condición, `cond`, que deben cumplir las filas a actualizarse: el nuevo usuario y nueva reunión de una invitación, debe estar en su correspondiente tabla. Esta condicción es agregada a las funciones `new_user_id` y `new_meeting_id`.

```
update_invitations(db: database_type,
                   selector: selector_invitations_type,
                   updater: updater_invitations_type):
                                             database_type =l
  LET cond(inv: invitationsId): bool =
        LET old_user = user_id(invitations(db))(inv),
            old_meeting = meeting_id(invitations(db))(inv),
            new_user = proj_1(updater(inv,old_user, old_meeting)),
            new_meeting = proj_2(updater(inv,old_user, old_meeting))
        IN
        member(new_user, idnt(users(db)))
        AND member(new_meeting, idnt(meetings(db))),
      new_user_id(inv: invitationsId): usersId =
        LET user = user_id(invitations(db))(inv),
            meeting = meeting_id(invitations(db))(inv)
        IN
        (COND NOT selector(inv, user, meeting, db) -> user,
              NOT cond(inv) -> user,
              ELSE -> proj_1(updater(inv, user, meeting, db))
        ENDCOND),
      new_meeting_id(inv:invitationsId): meetingsId
        LET user = user_id(invitations(db))(inv),
            meeting = meeting_id(invitations(db))(inv)
        IN
        (COND NOT selector(inv, user, meeting, db) -> meeting,
              NOT cond(inv) -> meeting,
              ELSE -> proj_2(updater(inv, user, meeting, db))
        ENDCOND),
      new_invitations = (# idnt := idnt(invitations(db)),
                           user_id := new_user_id,
                           meeting_id := new_meeting_id #)
      db' = db WITH [invitations := new_invitations]
  IN
```

```
db’
```

A modo de ejemplo, si se desea cambiar una invitación del usuario “Matias” al usuario “Virpa” de la reunión “Kalevala”, se definen las siguientes funciones:

```
select_Matias&Kalevala(inv: invitationId, usr: userId,
                 meet: meetingId, db: database_type): bool =
           (login(users(db))(usr) = Matias
            AND title(meetings(db))(meet) = Kalevala)

update_Matias2Virpa(inv: invitationId, usr: userId,
                 meet: meetingId, db: database_type):
                                    [userId, meetingId] =
            (Virpa, meet)

db’ = update_invitations(db, select_Matias&Kalevala,
                             update_Matias2Virpa)
```

El código SQL asociado a esta operación se define de forma similar.

```
UPDATE invitations
SET user_id = Virpa
WHERE user_id = Matias AND meeting_id = Kalevala
```

Insert

Sólo queda definir el operador *Insert*, utilizado para la inserción de nuevos datos en una tabla. Este operador se distingue de los descriptos anteriormente en que no hace uso de funciones como las de selección o actualización. El ingreso de información se hace de forma explicita, ingresando una fila en particular y enumerando los valores de los campos.

La definición del operador *Insert* para la tabla `invitations`, toma distintos valores para cada uno de los campos definidos en la misma y los inserta como un nuevo registro. Este operador queda definido como se muestra a continuación:

```
insert_invitations(db: database_type, inv: invitationsId,
                   usr: usersId, meet: meetingsId): database_type =
   LET invitations’ =
                (# idnt := add(inv, idnt(invitations(db))),
                   user_id := user_id(invitations(db)) WITH [inv := usr]
                   meeting_id := meeting_id(invitations(db))
                                                    WITH [inv := meet] #),
       db’ = db WITH [invitations := invitations’]
   IN
   db’
```

Sin embargo, esta definición no es completamente correcta. Para que una fila pueda ser ingresada en la tabla, se debe garantizar que el identificador de invitación no exista en la misma, debido a las restricciones de clave primaria. También se debe garantizar, al igual que en el operador *Update*, que tanto el usuario como la reunión existan en la tabla correspondiente.

Cambiamos entonces la definición de *Insert* para que se realicen las corroboraciones pertinentes:

```
insert_invitations(db: database_type, inv: invitationsId,
                   usr: usersId, meet: meetingsId): database_type =
   LET invitations' =
                (# idnt := add(inv, idnt(invitations(db))),
                   user_id := user_id(invitations(db)) WITH [inv := usr]
                   meeting_id := meeting_id(invitations(db))
                                                    WITH [inv := meet] #),
       db' = (COND member(inv, idnt(invitations(db))) -> db,
                   NOT member(usr, idnt(users(db))) -> db,
                   NOT member(meet, idnt(meetings(db))) -> db,
                   ELSE -> db WITH [invitations := invitations']
              ENDCOND)
   IN
   db'
```

Supongamos desear extender una invitación al usuario “Johannes” para que asista a la reunión titulada “Joulu”. Definimos los siguientes predicados[1]:

```
db' = insert_invitations(db, 4532, Johannes, Joulu).
```

El código SQL asociado a esta operación se define de forma analoga:

```
INSERT INTO invitations
VALUES (4532, Johanes, Joulu)
```

2.3.3. Operaciones complejas

Los operadores de Bases de Datos describen la forma en que se manipula la información almacenada en las tablas. Sin embargo, estas operaciones básicas, no capturan las restricciones y funcionalidades complejas que se desea tener. Se requieren mecanismos más expresivos para poder especificar comportamientos adicionales. Estos mecanismos están presentes bajo el nombre de *triggers*.

[1]Consideramos, por simplicidad, que “Johannes” y “Joulu” son constantes definidas que representan los identificadores de usuario y reunión

Ya hemos visto en la sección 2.3.2 algunos triggers de forma implícita, relacionados a las clausulas `ON DELETE CASCADE`. Esta clausula crea un trigger asociado a la operación de eliminación de la tabla para mantener la integridad referencial de las tablas.

En esta sección veremos como podemos traducir aquellos triggers utilizados en la implementación de *Deseo* para añadir ciertas restricciones o comportamientos a la Base de Datos.

Delete

Como se describió en la sección 2.2, uno de los requerimientos funcionales del Manejador de Eventos exige que:

> Cuando una invitación para una reunión sea eliminada, todas las respuestas del invitado a las propuestas relacionadas con dicha reunión, deben ser eliminadas también.

Sin embargo, esta condicción no puede se expresada en el modelo relacional planteado sin el auxilio de los triggers. Para esto, definimos la siguiente función:

```
create function delete_answers_uninvited() return trigger as $dai$
begin
  DELETE FROM answers
  WHERE user_id = OLD.user_id
        AND proposal_id IN
            (SELECT id
             FROM proposals
             WHERE meeting_id = OLD.meeting_id);
  RETURN NULL;
end;
$dai$ language 'plpgsql'
```

Esta función implementa el comportamiento adicional deseado, solo resta asociarla como un *after* trigger sobre la tabla `invitations` para que sea ejecutada ante una operación de borrado sobre la misma.

Al asociar una función a una operación de una tabla como un *after* trigger, estamos obligando al Motor de Base de Datos a disparar el trigger una vez que se haya efectuado la operación pertinente. De esta manera, no tenemos forma de evitar que la operación se realice.

```
create trigger tr_delete_answers_uninvited after delete on invitations
for each row execute procedure delete_answers_uninvited();
```

El trigger enriquece la funcionalidad de la operación *Delete* sobre la tabla de invitaciones. Una vez creado el trigger, no existe forma de utilizar la operación *Delete* sin que el trigger sea disparado también. Con esta idea, veamos entonces como podemos incorporar la funcionalidad de `delete_answers_uninvited()` a la función `delete_invitations`, definida en la sección 2.3.2.

El operador *Delete* fue declarado de la siguiente forma:

```
delete_invitations(db: database_type,
                   selector: selector_invitations_type):
                                           database_type =
  LET invitations' =
       (# idnt := {x: invitationsId | member(x, idnt(invitations(db)))
                      AND NOT selector(x, user_id(invitations(db))(x),
                                 meeting_id(invitations(db))(x), db)},
          user_id := user_id(invitations(db)),
          meeting_id := meeting_id(invitations(db)) #),
      db' = db WITH [invitations := invitations']
  IN
  db'
```

Nótese que para incorporar la funcionalidad del trigger sobre esta operación, se debe definir un predicado auxiliar sobre las respuestas, que elimine aquellas que sean pertinentes.

Como la operación de eliminación es atómica en nuestro modelo, definimos una tabla temporal que contenga aquellas filas que fueron removidas. Esta tabla nos dará acceso a la información de las invitaciones eliminadas, que será usada en la definición del trigger.

Definimos entonces una primer aproximación a la operación *Delete*:

```
delete_invitations(db: database_type,
                   selector: selector_invitations_type):
                                           database_type =
  LET invitations' =
       (# idnt := {x: invitationsId | member(x, idnt(invitations(db)))
                      AND NOT selector(x, user_id(invitations(db))(x),
                                 meeting_id(invitations(db))(x), db)},
          user_id := user_id(invitations(db)),
          meeting_id := meeting_id(invitations(db)) #),
      db' = db WITH [invitations := invitations'],
      deleted_invitations = select_invitations(db, selector),
      answers_selector(ans: answerId, usr: userId, prop: proposalId,
                                     ok: bool): bool = ....,
```

```
    db'' = delete_answers(db', answers_selector)
  IN
  db''
```

Sólo resta definir el predicado answers_selector encargado de eliminar las respuestas pertinentes, es decir, la implementación del trigger propiamente dicha.

Observemos que en la definición del trigger se realiza una subconsulta en donde se seleccionan las propuestas de la reunión que referencia la invitación eliminada. Esta subconsulta es definida en las lineas:

```
SELECT id
FROM proposals
WHERE meeting_id = OLD.meeting_id
```

Dentro del seleccionador interno answers_selector, debe existir una función de selección auxiliar que realice esta subconsulta. Suponiendo que la variable old esta cuantificada sobre las invitaciones eliminadas, como veremos más adelante, podemos definir la función de selección auxiliar de la siguiente forma:

```
proposal_selector(prop': proposalId, meet': meetingId,
                             str': string, conf': bool): bool =
                meet' = meeting_id(deleted_invitations(old)),
proposals' = select_proposals(db', proposal_selector)
```

La variable proposals' contiene todas las propuestas sobre la reunión a la que hace referencia la invitación old.

Estamos ya en condicciones de poder definir la función de selección de respuestas a borrar answers_selector. Supongamos borrar una invitación I que vincula el usuario U con la reunión R. Una respuesta va a ser borrada si fue dada por el usuario U y hace referencia a una propuesta de la reunión R.

El predicado de selección sobre la tabla de respuestas se define de la siguiente forma:

```
answers_selector(ans: answerId, usr: userId, prop: proposalId,
                                        ok: bool): bool =
   EXISTS (old: invitationId):
      LET proposal_selector(prop': proposalId, meet': meetingId,
                             str': string, conf': bool): bool =
          meet' = meeting_id(deleted_invitations(old)),
          proposals' = select_proposals(db', proposal_selector)
      IN
      member(old, idnt(deleted_invitations))
      AND usr = user_id(delete_invitations)(old)
      AND member(prop, idnt(proposals'))
```

Finalmente, podemos definir con total precisión la operación *Delete* sobre la tabla invitaciones y su correspondiente trigger asociado de la siguiente manera:

```
delete_invitations(db: database_type,
                   selector: selector_invitations_type):
                                            database_type =
  LET invitations' =
       (# idnt := {x: invitationsId | member(x, idnt(invitations(db)))
                       AND NOT selector(x, user_id(invitations(db))(x),
                                  meeting_id(invitations(db))(x), db)},
          user_id := user_id(invitations(db)),
          meeting_id := meeting_id(invitations(db)) #),
      db' = db WITH [invitations := invitations'],
      deleted_invitations = select_invitations(db, selector),
      answers_selector(ans: answerId, usr: userId, prop: proposalId,
                                                  ok: bool): bool =
          EXISTS (old: invitationId):
                 LET proposal_selector(prop': proposalId, meet': meetingId,
                                       str': string, conf': bool): bool =
                           meet' = meeting_id(deleted_invitations(old)),
                     proposals' = select_proposals(db', proposal_selector)
                 IN
                 member(old, idnt(deleted_invitations))
                 AND usr = user_id(delete_invitations)(old)
                 AND member(prop, idnt(proposals')),
      db'' = delete_answers(db', answers_selector)
  IN
  db''
```

El trigger, al ser declarado como un *after trigger*, será ejecutado luego de que la operación de borrado haya tenido lugar. Esto se ve reflejado, en nuestra traducción, en el hecho de que se define una nueva base de datos `db'` y la función que captura el comportamiento del trigger trabaja sobre esta nueva base de datos y no sobre la original.

Update

Otro requerimiento del Manejador de Eventos *Deseo*, descripto en la sección 2.2 dice que:

> Una vez que se ofreció una invitación, esta no puede ser modificada.

Esta restricción puede implementada a través de un *before* trigger sobre la operación *Update* de la tabla `invitations` de la siguiente forma:

```
create function disable_invitation_change() return trigger as $dic$
begin
  IF NEW.user_id <> OLD.user_id OR NEW.meeting_id <> OLD.meeting_id
     THEN RETURN NULL;
     ELSE RETURN NEW;
  END IF;
end;
$dic$ language 'plpgsql'

create trigger tr_disable_invitation_change before update on invitations
for each row execute procedure disable_invitation_change();
```

Los *before* triggers son lanzados antes de que la operación se lleve a cabo y por lo tanto son generalmente utilizados para establecer condicciones adicionales a las operaciones. De esta forma pueden impedir que las operaciones se lleven a cabo si no se satisfacen condicciones extras. Así, el trigger anterior establece una condicción sobre la actualización de los registros de una tabla.

Debe observarse que este tipo de comportamiento ya fue analizado en la sección 2.3.2 cuando se debía chequear la transparencia referencial en la operación *Update*. Podemos valernos de esta experiencia para definir este trigger de forma similar, agregando una condicción adicional a satisfacer para que los registros sean actualizados.

```
update_invitations(db: database_type,
                   selector: selector_invitations_type,
                   updater: updater_invitations_type):
                                                     database_type =l
  LET cond(inv: invitationsId): bool =
         LET old_user = user_id(invitations(db))(inv),
             old_meeting = meeting_id(invitations(db))(inv),
             new_user = proj_1(updater(inv,old_user, old_meeting)),
             new_meeting = proj_2(updater(inv,old_user, old_meeting))
         IN
         member(new_user, idnt(users(db)))
         AND member(new_meeting, idnt(meetings(db))),
      cond'(inv: invitationsId): bool =
         LET old_user = user_id(invitations(db))(inv),
             old_meeting = meeting_id(invitations(db))(inv),
             new_user = proj_1(updater(inv,old_user, old_meeting)),
             new_meeting = proj_2(updater(inv,old_user, old_meeting))
         IN
         new_user = old_user
         AND new_meeting = old_meetin,
```

```
    new_user_id(inv: invitationsId): usersId =
       LET user = user_id(invitations(db))(inv),
           meeting = meeting_id(invitations(db))(inv)
       IN
       (COND NOT selector(inv, user, meeting, db) -> user,
             NOT cond(inv) -> user,
             NOT cond'(inv) -> user,
             ELSE -> proj_1(updater(inv, user, meeting, db))
       ENDCOND),
    new_meeting_id(inv:invitationsId): meetingsId
       LET user = user_id(invitations(db))(inv),
           meeting = meeting_id(invitations(db))(inv)
       IN
       (COND NOT selector(inv, user, meeting, db) -> meeting,
             NOT cond(inv) -> meeting,
             NOT cond'(inv) -> meeting,
             ELSE -> proj_2(updater(inv, user, meeting, db))
       ENDCOND),
    new_invitations = (# idnt := idnt(invitations(db)),
                         user_id := new_user_id,
                         meeting_id := new_meeting_id #)
    db' = db WITH [invitations := new_invitations]
  IN
  db'
```

Vemos entonces como `cond'` implementa la restricción que impone el trigger sobre la operación de actualización antes de realizar la operación de modificación de datos.

Insert

Un último requerimiento a analizar de *Deseo* es el siguiente:

> El anfitrión de una reunión no puede ser invitado a la misma.

Una vez más, esta restricción se debe llevar a cabo a través del uso de triggers. Al igual que con la operación *Update*, se necesita un *before* trigger, ya que se agregará una condición adicional sobre la operación. En SQL, definimos este trigger, sobre la operación *Insert* de la tabla `invitations`, de la siguiente forma:

```
create function disable_inviting_host() return trigger as $dih$
declare
  host integer;
begin
  SELECT INTO host FROM meetings WHERE id = NEW.meeting_id;
```

```
  IF host = NEW.user_id
     THEN RETURN NULL;
     ELSE RETURN NEW;
  END IF;
end;
$dih$ language 'plpgsql'

create trigger tr_disable_inviting_host before insert on invitations
for each row execute procedure disable_inviting_host();
```

De forma similar a como se definió el *before* trigger para la operación de actualización sobre la tabla de invitaciones, podemos definir el trigger para la operación de inserción de datos. Tomando como base la operación *Insert* descripta en la sección , incorporamos el comportamiento del trigger de la siguiente forma:

```
insert_invitations(db: database_type, inv: invitationsId,
                   usr: usersId, meet: meetingsId): database_type =
   LET host = user_id(meetings(db))(meet),
       cond = host = usr,
       invitations' =
                (# idnt := add(inv, idnt(invitations(db))),
                   user_id := user_id(invitations(db)) WITH [inv := usr]
                   meeting_id := meeting_id(invitations(db))
                                                   WITH [inv := meet] #),
       db' = (COND member(inv, idnt(invitations(db))) -> db,
                   NOT member(usr, idnt(users(db))) -> db,
                   NOT member(meet, idnt(meetings(db))) -> db,
                   cond -> db
                   ELSE -> db WITH [invitations := invitations']
              ENDCOND)
   IN
   db'
```

De esta forma, solo los usuarios que no sean el anfitrión de la reunión podrán ser invitados a la misma.

2.4. Algunos teoremas y demostraciones

La definición de una especificación formal como se vio en las secciones anteriores, permite razonar matemáticamente sobre las propiedades deseables del sistema subyacente. Mostraremos en esta sección como podemos probar, de manera formal, que el sistema posee ciertas propiedades y restricciones haciendo uso del modelo creado.

Comencemos nuestro análisis por la operación *Delete* de la tabla `invitations`. En la sección 2.3.2 se definió la operación básica de borrado de invitaciones, para luego incorporarle una funcionalidad auxiliar propia de nuestro aplicación, como se vio en la sección 2.3.3. De esta manera, el borrado de una invitación dispara la eliminación de cientas respuestas que tienen una relación especifica con la invitación eliminada.

Si bien no es posible realizar este tipo de análisis directamente en el lenguaje SQL, si lo es en el modelo matemático creado a partir de estas sentencias. Definimos entonces un teorema que enuncie la propiedad que deseamos en *Deseo* para posteriormente tratar de demostrarlo. De ser posible, obtenemos una prueba (en el sentido matemático) de la validez de dicha propiedad en el sistema, siempre y cuando nuestra traducción haya sido correcta.

Para definir la propiedad deseada sobre la operación *Delete*, supongamos tener una invitación cualquiera `inv` y una respuesta cualquiera `ans`. Sea `inv_user` el usuario invitado a la reunión `inv_meeting` a través de la invitación `inv`. Sea `ans_user` el usuario que dio una respuesta sobre la propuesta `ans_proposal` de la reunión `ans_meeting` a través de la respuesta `ans`.

Lo que deseamos demostrar es que si `inv_user` coincide con `ans_user` e `inv_meeting` coincide con `ans_meeting` entonces, si eliminamos la invitación `inv` de la Base de Datos, la respuesta `ans` será eliminada también.

Esto mismo puede ser definido como un teorema en PVS de la siguiente forma:

```
delete_invitations_theorem: THEOREM
   FORALL (db: database_type, selector: selector_invitations_type,
                             ans: answersId, inv: invitationsId):
   LET db' = delete_invitations(db, selector),
       deleted_invitations = select_invitations(db, selector),
       inv_user = user_id(invitations(db))(inv),
       inv_meeting = meeting_id(invitations(db))(inv),
       ans_user = user_id(answers(db))(ans),
       ans_proposal = proposal_id(answers(db))(ans),
       ans_meeting = meeting_id(proposals(db))(ans_proposal)
   IN
   member(inv, idnt(inv_del))
   AND member(ans, idnt(answers(db)))
   AND member(ans_proposal, idnt(proposals(db)))
   AND inv_user = ans_user
   AND inv_meeting = ans_meeting
   =>
   NOT member(ans, idnt(answers(db')))
```

Una invitación, una vez creada, no puede ser modificada, esta es otra propiedad que consideramos deseable en *Deseo*. Para esto se definió el trigger `tr_disable_invitation_change`. Sin embargo no estamos seguros de esto con certeza ya que no contamos con ninguna demostración de su veracidad.

Deseamos probar que para cualquier invitación `inv`, sin importar el comportamiento del predicado de selección ni el de la función de actualización, el usuario y la reunión que vincula la invitación no cambie. Esta idea es capturada en la definición del siguiente teorema:

```
update_invitations_theorem: THEOREM
   FORALL (db: database_type, selector: selector_invitations_type,
           updater: updater_invitations_type, inv: invitationsId):
   LET db’ = update_invitations(db, selector, updater)
   IN
   user_id(invitations(db))(inv) = user_id(invitations(db’))(inv)
   AND meeting_id(invitations(db))(inv) = meeting_id(invitations(db’))(inv)
```

Una última propiedad deseada en *Deseo* es que el anfitrión de una reunión no puede ser invitado a la misma. Para lograr esta restricción se definió, como vimos, el trigger `tr_disable_inviting_host`, pero una vez más, no tenemos seguridad absoluta de que esta función implemente correctamente nuestra idea.

Supongamos querer insertar un registro cuya identificador de invitación esta dado por la variable `invitation`, el identificador del usuario a invitar por la variable `user` y la reunión por `meeting`. Deseamos que si el anfitrión de la reunión `meeting` coincide con `user`, el registro no sea insertado.

Para esto, definimos el teorema:

```
insert_invitations_theorem: THEOREM FORALL (db: database_type,
                    invitation: invitationsId, user: usersId,
                    meeting: meetingsId):
   LET db’ = insert_invitations(db, invitation, user, meeting),
       meeting_host = user_id(meetings(db))(meeting)
   IN
   NOT member(invitation, idnt(invitations(db)))
   AND user = meeting_host
   =>
   NOT member(invitation, idnt(invitations(db’)))
```

Estos tres teoremas, enuncian las propiedades y restricciones deseables, correspondientes a la tabla de invitaciones y pueden ser probadas solamente con el uso del comando `(grind)` de PVS. De esta manera, se concentra el esfuerzo en la especificación de los teoremas y no

en la prueba de los mismos.

De forma análoga, se enunciaron los teoremas correspondientes a las propiedades y restricciones de las demás tablas, que fueron demostrados con el uso del comando `(grind)` y si intervención del usuario.

El hecho de que las demostraciones no requieran intervención del usuario y se realicen de forma totalmente automática, es en parte gracias al poder de las técnicas de demostración de PVS y en parte a la definición del modelo. Se realizaron originalmente otras propuestas para el modelado del sistema en las que la demostración de las propiedades no se realizaba de forma automática debido a, según la opinión del autor, su incorrecto nivel de abstracción.

Capítulo 3

Generalización

En este capítulo se generaliza el enfoque utilizado para la especificación del modelo correspondiente al Manejador de Reuniones *Deseo* visto en el capítulo 2.

Comenzaremos definiendo la sintaxis del lenguaje de Base de Datos y del lenguaje que utilizaremos para el modelado. Se definirán los tipos de datos utilizados para luego pasar a la declaración de las reglas de traducción que generan las operaciones básicas de manipulación de datos. Finalmente se propondrá una extensión de la sintaxis para contemplar el tratamiento de los *triggers*.

3.1. Definiciones

Comenzaremos definiendo la gramática del lenguaje que se utilizará como dominio de las reglas de traducción. Esta gramática está inspirada en el Lenguaje de Definición de Datos (DDL, del inglés *Data Definition Language*) del estándar SQL.

$$
\begin{array}{rcl}
BaseDeDatos & ::= & \textbf{BaseDeDatos}\ Tablas \\
\\
Tablas & ::= & \textbf{Tablas}\ Tabla\ Tablas \\
 & & \mid \epsilon_{tablas} \\
\\
Tabla & ::= & \textbf{Tabla}\ nombre\ clave\ tipo\ Campos \\
\\
Campos & ::= & \textbf{Campos}\ nombre\ tipo\ Campos \\
 & & \mid \epsilon_{campos}
\end{array}
$$

Una Base de Datos puede ser tratada, conceptualmente, como una lista de una o más tabla. Cada tabla es identificada con un nombre, posee una clave primaria, el tipo de datos correspondiente a la clave primaria y un conjunto de Campos. Por último, Cada campo contiene un nombre y un tipo de datos que almacena.

A modo de convención, los elementos *nombre* y *clave* son cualquier cadena de caracteres legal en el Motor de Base de Datos utilizado. De la misma forma, *tipo*, es cualquier tipo de datos que el Motor acepte como tipo de dato legal. Supondremos que estos elementos tiene una traducción directa al lenguaje de especificación que definiremos a continuación.

Puede observarse que se obviaron las referencias entre las tablas, es decir, las clausulas "REFERENCES" no son contempladas en la gramática. Para esto, definimos el tipo de datos *Referencias* que captura las relaciones entre las tablas de una Base de Datos.

$$\begin{array}{lll} Referencias & ::= & \textbf{Referencias}\ tabla_1\ campo_1\ tabla_2\ campo_2\ Referencias \\ & & \mid \epsilon_{referencias} \end{array}$$

Los elementos $tabla_1$, $campo_1$, $tabla_2$ y $campo_2$ son cadenas de caracteres cuyo significado es que el campo $campo_1$ de la tabla $tabla_1$ hace referencia a la clave primaria (cuyo nombre es $campo_2$) de la tabla $tabla_2$. Supondremos contar con una variable global **Referencias** de tipo *Referencias*, para la definición de las reglas, que contenga todas las referencias que existen entre las tablas de una Base de Datos.

De la misma forma, se debe definir el lenguaje al que se realizará la traducción. Basandonos en [8], se define una sublenguaje de PVS que será el necesario para la definición de las operaciones de Base de Datos.

Se comienza definiendo el lenguaje de tipos que será utilizado [1]:

$$\begin{array}{lll} PVSTipo & ::= & \textbf{Nat} \\ & & \mid \textbf{Bool} \\ & & \mid [PVSTipo \rightarrow PVSTipo] \\ & & \mid \{PVSTipo\} \\ & & \mid [PVSTipo^{+}] \\ & & \mid [\#\ TypedIds\ \#] \\ & & \mid nombre : PVSTipo \\ & & \mid \textbf{Tipo}\ nombre \end{array}$$

Los distintos elementos de la gramática se corresponden con los números naturales, los booleanos, las funciones, los conjuntos, las tuplas y los registros respectivamente. A un tipo de datos se le puede asignar un identificador que podrá ser utilizado para identificar un tipo ya definido, como muestran los últimos dos elementos de la gramática.

Se define a continuación la gramática que será utilizada como sublenguaje del lenguaje de especificación PVS:

[1] T^{+} es una abreviación para identificar la lista de una o más instancia del elemento T, ya sea un tipo a una expresión

$$
\begin{aligned}
Exp \; ::= \; & true \\
& \mid false \\
& \mid Numero \\
& \mid String \\
& \mid Exp\ BinOp\ Exp \\
& \mid UnOp\ Exp \\
& \mid \exists\ TypedIds \,\textbf{.}\, Exp \\
& \mid Exp\ Argumentos \\
& \mid \{TypedIds \mid Exp\} \\
& \mid String \in Exp \\
& \mid \textbf{Let}\ LetBinding^{+}\ \textbf{In}\ Exp \\
& \mid Exp\ \textbf{With}\ Asignaciones \\
& \mid (\#\ Asignaciones\ \#) \\
& \mid \textbf{Cond}\ (Exp \rightarrow Exp)^{+},\ \textbf{Else} \rightarrow Exp\ \textbf{EndCond}
\end{aligned}
$$

$$
\begin{aligned}
TypedIds \; & ::= \; TypedId,\ TypedIds \\
& \quad\;\; \mid \epsilon_{typedids} \\[1em]
TypedId \; & ::= \; nombre\ :\ PVSTipo \\[1em]
Argumentos \; & ::= \; \textbf{Arg}\ Exp\ Argumentos \\
& \quad\;\; \mid \epsilon_{argumentos} \\[1em]
Asignaciones \; & ::= \; nombre\ :=\ Exp,\ Asignaciones \\
& \quad\;\; \mid \epsilon_{asignaciones} \\[1em]
LetBinding \; & ::= \; LetBind\ =\ Exp \\[1em]
LetBind \; & ::= \; nombre \\
& \quad\;\; \mid nombre\ TypedIds\ :\ PVSTipo \\[1em]
BinOp \; & ::= \; + \mid - \mid * \mid / \mid \wedge \mid \vee \mid ... \\[1em]
UnOp \; & ::= \; \sim \mid - \mid ...
\end{aligned}
$$

Los elementos de la gramática son, por lo general, bastante intuitivos. Puede consultarse [8] para mayor información sobre la gramática completa del lenguaje de especificación PVS y su correspondiente semántica.

Se hará un pequeño abuso de notación en cuanto a *Argumentos*, utilizando la sintaxis tradicional (x_1, x_2, x_3, ...) en lugar del constructor **Arg** cuando sea claro que es lo que se esta realizando.

Definimos a continuación el lenguaje que utilizaremos para la especificación de las tablas y funciones de una Base de Datos particular.

$$
\begin{array}{lll}
Decl & ::= & \textbf{Tipo}\ nombre : PVSTipo \\
 & & \mid \textbf{Func}\ nombre\ Argumentos\ PVSTipo\ Exp \\
 & & \mid Decl,\ Decl \\
 & & \mid \epsilon_{decl}
\end{array}
$$

Definimos a continuación la familia de funciones $EstructuraTabla$ que serán utilizadas más adelante.

$EstructuraTabla :: BaseDeDatos \rightarrow String \rightarrow TypedIds$
$EstructuraTabla(\textbf{BaseDeDatos}\ tablas) = EstructuraTabla(tablas)$

$EstructuraTabla :: Tabla \rightarrow String \rightarrow TypedIds$
$EstructuraTabla(\textbf{Tablas}\ tabla\ tablas) =$
$EstructuraTabla(tablas) \circ EstructuraTabla(tabla)$

$EstructuraTabla(\epsilon_{tablas}) = \oslash_{Est}$

$EstructuraTabla :: Tabla \rightarrow String \rightarrow TypedIds$
$EstructuraTabla(\textbf{Tabla}\ nombre\ clave\ tipo\ campos) =$
$[nombre \rightarrow EstructuraTabla(campos)]$

$EstructuraTabla :: Campos \rightarrow TypedIds$
$EstructuraTabla(\textbf{Campos}\ nombre\ tipo\ campos) =$
$nombre : tipo, EstructuraTabla(campos)$
$EstructuraTabla(\epsilon_{campos}) = \epsilon_{typedids}$

Donde $\oslash_{Est}$ es la función definida de la siguiente forma:

$$\forall x : String \cdot \oslash_{Est}(x) = \epsilon_{typedids}$$

El operador $\circ$ es la composición funcional y $[x \rightarrow y]$ es la función que asigna y al valor x. Para la definición de las reglas supondremos contar con una variable global **Estructura** tal que

$$
\begin{array}{lll}
\textbf{Estructura} & :: & String \rightarrow TypedIds \\
\textbf{Estructura} & = & EstructuraTabla(baseDeDatos)
\end{array}
$$

Donde *baseDeDatos* es la Base de Datos sobre la cual se trabaja.

Por último supondremos contar con la función **Clave** que toma el nombre de una tabla y retorna el nombre de la clave primaria de dicha tabla.

$$\textbf{Clave} :: String \rightarrow String$$

3.2. Declaración de los tipos principales

Para la definición de las operaciones de Base de Datos, deben ser declarados previamente, los tipos de datos que estas operaciones utilizan. Estos tipos son los correspondientes a las tablas, a la Base de Datos como una entidad, a las funciones de selección y de actualización de registros.

3.2.1. Tablas

Las funciones de traducción deberán generar objetos del tipo *Decl* a partir de la representación de la Base de Datos. La definición de los tipos que capturan la estructura de las tablas de una Base de Datos esta dada por la familia de funciones $\ll\gg_{Tablas}$ como se muestra con las siguientes reglas:

$$\ll\gg_{Tablas} :: \ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Tablas} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Tablas} \rightarrow decl}$$

$$\ll\gg_{Tablas} :: \ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Tablas} \rightarrow decl_1 \qquad \ll tablas \gg_{Tablas} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Tablas} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Tablas} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Tabla} :: \ Tabla \rightarrow Decl$$

$$\frac{\ll campos,\ tipo \gg_{Tablas} \rightarrow typedIds}{\begin{array}{c}\ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Tablas} \rightarrow \\ \textbf{Tipo}\ nombre_type : [\#\ clave : \{\textbf{Tipo}\ tipo\},\ typedIds\ \#]\end{array}}$$

$$\ll\gg_{Tabla} :: \ Campos \times String \rightarrow TypedIds$$

$$\frac{\ll campos,\ tipoClave \gg_{Tablas} \rightarrow typedIds}{\begin{array}{c}\ll \textbf{Campos}\ nombre\ tipo\ campos,\ tipoClave \gg_{Tablas} \rightarrow \\ nombre : [tipoClave \rightarrow tipo],\ typedIds\end{array}}$$

$$\frac{}{\ll \epsilon_{campos},\ tipoClave \gg_{Tablas} \rightarrow \epsilon_{typedids}}$$

Obsérvese que las definiciones de las reglas se asemejan notablemente a las utilizadas en el caso de estudio descrito en el capítulo anterior.

Para la definición de un tipo que capture la estructura de una tabla en particular, utilizaremos el nombre de la tabla, seguido por un guión bajo y la cadena de caracteres *type* como identificador del tipo. De esta manera, podremos hacerle referencia posteriormente. La función **TipoTabla** que nos permitirá obtener el tipo especifico de una tabla en particular.

$$\textbf{TipoTabla} :: String \rightarrow PVSTipo$$

$$\textbf{TipoTabla}\ str = \textbf{Tipo}\ (str\ +\ \text{“_}type\text{”})$$

3.2.2. Base de Datos

Una vez definidos los tipos de datos que capturan la estructura de las tablas, se puede definir el tipo que captura la estructura de la Base de Datos a través de una familia de funciones $\ll\gg_{BaseDeDatos}$.

$$\ll\gg_{BaseDeDatos} ::\ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{BaseDeDatos} \rightarrow typedIds}{\ll \textbf{BaseDeDatos}\ tablas \gg_{BaseDeDatos} \rightarrow \textbf{Tipo}\ database_type : [\#\ typedIds\ \#]}$$

$$\ll\gg_{BaseDeDatos} ::\ Tablas \rightarrow TypedIds$$

$$\frac{\ll tabla \gg_{BaseDeDatos} \rightarrow typedId \qquad \ll tablas \gg_{BaseDeDatos} \rightarrow typedIds}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{BaseDeDatos} \rightarrow typedId,\ typedIds}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{BaseDeDatos} \rightarrow \epsilon_{typedids}}$$

$$\ll\gg_{Tabla} ::\ Tabla \rightarrow TipedId$$

$$\frac{}{\ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Tablas} \rightarrow nombre : (\textbf{TipoTabla}\ nombre)}$$

Como muestran las reglas, el tipo de datos que captura la noción de Base de Datos es un registro cuyos elementos son las tablas que la componen. Estas tablas puede ser accedidas a través de sus nombres.

Tal como se utilizó una convención para los nombres que identifican los tipos de datos de las tablas, utilizaremos el nombre *database_type* para hacer referencia al tipo de datos correspondiente a la Base de Datos. Utilizaremos este nombre como una constante en las futuras definiciones.

3.2.3. Selector

De la misma forma, se debe definir el tipo de datos que encapsula la idea de selector. Como se vio en la sección 2.3.2, el selector es una función que selecciona los registros de una tabla en particular. Para esto, se define una familia de funciones $\ll\gg_{Selector}$ de la siguiente manera:

$$\ll\gg_{Selector} ::\ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Selector} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Selector} \rightarrow decl}$$

$$\ll\gg_{Selector} ::\ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Selector} \rightarrow decl_1 \qquad \ll tablas \gg_{Selector} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Selector} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Selector} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Selector} ::\ Tabla \rightarrow Decl$$

$$\frac{\ll campos \gg_{Selector} \rightarrow pvsTipo^{+}}{\begin{array}{c}\ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Selector} \rightarrow \\ \textbf{Tipo}\ selector_nombre_type : [[tipo,\ pvsTipo^{+},\ database_type] \rightarrow \textbf{Bool}]\end{array}}$$

$$\ll\gg_{Selector} ::\ Campos \rightarrow PVSTipo^{+}$$

$$\frac{\ll campos \gg_{Selector} \rightarrow pvsTipo^{+}}{\ll \textbf{Campos}\ nombre\ tipo\ campos \gg_{Selector} \rightarrow tipo,\ pvsTipo^{+}}$$

$$\frac{}{\ll \epsilon_{campos} \gg_{Selector} \rightarrow \epsilon_{PVSTipo^{+}}}$$

La función **TipoSelector** permite obtener el tipo de datos que encapsula una función de selección para una tabla en particular.

$$\textbf{TipoSelector} :: String \rightarrow PVSTipo$$

$$\textbf{TipoSelector}\ str = \textbf{Tipo}\ (\text{“}selector_\text{”} + str\ +\ \text{“}_type\text{”})$$

3.2.4. Actualizador

Sólo queda definir el tipo de datos que encapsula la noción de actualizador de registros, como se vio en la sección 2.3.2. Esta función toma los mismos argumentos que la función de selección pero retorna el registro modificado en vez de un valor booleano. Para su definición se utiliza una familia de funciones $\ll\gg_{Actualizador}$.

$$\ll\gg_{Actualizador} :: \ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Actualizador} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Actualizador} \rightarrow decl}$$

$$\ll\gg_{Actualizador} :: \ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Actualizador} \rightarrow decl_1 \qquad \ll tablas \gg_{Actualizador} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Actualizador} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Actualizador} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Actualizador} :: \ Tabla \rightarrow Decl$$

$$\frac{\ll campos \gg_{Actualizador} \rightarrow pvsTipo^{+}}{\begin{array}{c}\ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Actualizador} \rightarrow \\ \textbf{Tipo}\ actualizador_nombre_type : [[tipo,\ pvsTipo^{+},\ database_type] \rightarrow [pvsTipo^{+}]]\end{array}}$$

$$\ll\gg_{Actualizador} :: \ Campos \rightarrow PVSTipo^{+}$$

$$\frac{\ll campos \gg_{Actualizador} \rightarrow pvsTipo^{+}}{\ll \textbf{Campos}\ nombre\ tipo\ campos \gg_{Actualizador} \rightarrow tipo,\ pvsTipo^{+}}$$

$$\frac{}{\ll \epsilon_{campos} \gg_{Actualizador} \rightarrow \epsilon_{pvsTipo^{+}}}$$

Para poder referenciar los distintos tipos de actualizadores se define la función **TipoActualizador** de la siguiente forma:

$$\textbf{TipoActualizador} :: String \rightarrow PVSTipo$$

$$\textbf{TipoActualizador}\ str = \textbf{Tipo}\ (\text{``}actualizador_\text{''} + str\ +\ \text{``}_type\text{''})$$

3.3. Operaciones básicas

Una vez declarados los principales tipos de datos, puede continuarse con la especificación de las reglas que definen las operaciones básicas de Base de Datos. Comenzaremos con las reglas correspondientes a la operación *Select*, siguiendo con las correspondientes a *Delete*, *Update* y finalmente *Insert*. La especificación de las reglas seguirán los lineamientos utilizados en la sección 2.3.2.

3.3.1. Select

Para la especificación del operador *Select*, se define una familia de funciones $\ll\gg_{Select}$. El resultado de aplicar esta familia de funciones a la definición de una Base de Datos, será un conjunto de declaraciones que definen las funciones de selección para las distintas tablas.

Cada una de estas funciones, toma como parámetros la función de selección correspondiente y la base de datos sobre la que se realizará la selección. Cada función una de estas funciones retornará una tabla que contenga solo aquellas columnas que satisfagan el predicado de selección.

$$\ll\gg_{Select} :: \ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Select} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Select} \rightarrow decl}$$

$$\ll\gg_{Select} :: \ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Select} \rightarrow decl_1 \qquad \ll tablas \gg_{Select} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Select} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Select} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Select} :: \ Tabla \rightarrow Decl$$

$$\frac{\ll campos,\ bd,\ nombre \gg_{Select} \rightarrow asign \qquad \ll \textbf{Estructura}(nombre) \gg_{Ids} \rightarrow argsIds}{\begin{array}{c} \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Select} \rightarrow \\ \textbf{Fun}\ select_nombre\ (bd:\ database_type,\ selector:\ \textbf{TipoSelector}(nombre)) \\ \textbf{TipoTabla}(nombre) \end{array}}$$

$$\begin{array}{lll} \textbf{Let} & clave' = \{x:\ \textbf{Tipo}\ tipo \mid x \in clave(nombre(bd)) \wedge \sim selector(x,\ argIds)\}, \\ & tbl = (\#\ clave',\ asign\ \#) \\ \textbf{In} & \\ tbl & \end{array}$$

$$\ll\gg_{Select} ::\ Campos \rightarrow Asignaciones$$

$$\frac{\ll campos,\ bd,\ tabla \gg_{Select} \rightarrow asign}{\ll \textbf{Campos}\ nombre\ tipo\ campos,\ bd,\ tabla \gg_{Select} \rightarrow \\ nombre\ :=\ nombre(tabla(bd)),\ asign}$$

$$\frac{}{\ll \epsilon_{campos} \gg_{Select} \rightarrow \epsilon_{asignaciones}}$$

La función de selección de una tabla en particular define un conjunto con los elementos pertenecientes a la tabla original que satisfacen el criterio de selección. Luego retorna una tabla con los elementos de este nuevo conjunto y los demás campos de las filas sin ser alterados.

Sólo queda definir la función $\ll\gg_{Ids}$, utilizada anteriormente, de la siguiente forma:

$$\ll\gg_{Ids} :: TypedIds \rightarrow Argumentos$$

$$\frac{\ll typedIds \gg_{Ids} \rightarrow args}{\ll nombre : tipo,\ typedIds \gg_{Ids} \rightarrow \textbf{Arg}\ nombre\ args}$$

$$\frac{}{\ll \epsilon_{typedids} \gg_{Ids} \rightarrow \epsilon_{argumentos}}$$

Cada función que capture el comportamiento de la operación *Select*, para una tabla en particular, tendrá por nombre, la cadena "*select_*" seguida por el nombre de la tabla correspondiente. Esta convención se ve reflejado en la regla de traducción $\ll\gg_{Select} ::\ Tabla \rightarrow Decl$. Definimos entonces la función **Select** que permitirá hacer referencia a la función de selección en futuras reglas.

$$\textbf{Select} :: String \rightarrow Exp$$

$$\textbf{Select}\ str = \text{"}select_\text{"} + str$$

3.3.2. Delete

La familia de funciones $\ll\gg_{Delete}$ es utilizada para la definición de la función correspondiente al operador *Delete*. Estas toman como parámetros un predicado que selecciona las filas que deberán ser borradas y una basa de datos sobre la cual se deberá realizar la operación. El resultado será una nueva base de datos con las filas de la tabla correspondientes eliminadas.

Para el tratamiento de las referencias entre las tablas se define una tabla auxiliar utilizada para acceder a los registros que han sido eliminadas. Se utiliza la variable global **Referencias** para realizar las eliminaciones necesarias en las demás tablas de la base de datos, implementado el comportamiento de eliminación en cascada.

$$\ll\gg_{Delete} :: \ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Delete} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Delete} \rightarrow decl}$$

$$\ll\gg_{Delete} :: \ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Delete} \rightarrow decl_1 \qquad \ll tablas \gg_{Delete} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Delete} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Delete} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Delete} :: \ Tabla \rightarrow Decl$$

$$\frac{\begin{array}{c}\ll \textbf{Referencias},\ bd',\ nombre,\ registrosBorrados \gg_{OnCascadeDelete} \rightarrow exp \\ \ll campos,\ bd,\ nombre \gg_{Delete} \rightarrow asign \qquad \ll \textbf{Estructura}(nombre) \gg_{Ids} \rightarrow argsIds\end{array}}{\begin{array}{l}\qquad\qquad \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Delete} \rightarrow \\ \textbf{Fun}\ delete_nombre\ (bd :\ database_type,\ selector :\ \textbf{TipoSelector}(nombre))\ database_type \\ \quad \textbf{Let}\ \ clave' \ = \ \{x :\ \textbf{Tipo}\ tipo \mid x \in clave(nombre(bd)) \wedge \sim selector(x,\ argIds)\}, \\ \qquad\quad nombre' \ = \ (\#\ clave',\ asign\ \#), \\ \qquad\quad bd' \ = \ bd\ \textbf{With}\ (nombre \ := \ nombre'), \\ \qquad\quad registrosBorrados \ = \ \textbf{Select}(nombre)(bd, selector), \\ \qquad\quad bd'' = exp \\ \quad \textbf{In} \\ \quad bd''\end{array}}$$

$$\ll\gg_{Delete} :: \ Campos \rightarrow Asignaciones$$

$$\frac{\ll campos,\ bd,\ tabla \gg_{Delete} \rightarrow asign}{\begin{array}{c}\ll \textbf{Campos}\ nombre\ tipo\ campos,\ bd,\ tabla \gg_{Delete} \rightarrow \\ nombre \ := \ nombre(tabla(bd)),\ asign\end{array}}$$

$$\frac{}{\ll \epsilon_{campos} \gg_{Delete} \rightarrow \epsilon_{asignaciones}}$$

Las reglas para el operador *Delete* son similares a las definidas para el operador *Select*. Una de las diferencias principales es que la función de eliminación debe retornar la nueva base de datos sin los registros que fueron borrados, mientras que la función de selección retorna la tabla sobre la cual se realizo la operación.

La función de eliminación requiere el tratamiento de las referencias entre las tablas, para esto se define la función $\ll\gg_{OnCascadeDelete}$, encargada de realizar las eliminaciones pertinentes en las demás tablas de la Base de Dato. Esta función debe recibir las referencias que existen entre las tablas, la Base de Datos sobre la cual se realizaran las operaciones (es decir, la Base de Datos con los registros eliminados), el nombre de la tabla sobre la cual se esta trabajando y el nombre de la tabla que contiene los registros que fueron eliminados.

$$\ll\gg_{OnCascadeDelete} :: \ Referencias \times String \times String \times String \rightarrow Exp$$

$$\frac{\begin{array}{l}\ll refs,\ bd',\ tabla,\ registros \gg_{OnCascadeDelete} \rightarrow exp \\ VariableFresca = \mathbf{GeneradorVariablesFrescas}()\end{array}}{\begin{array}{l}\ll \mathbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ tabla,\ registros \gg_{OnCascadeDelete} \rightarrow \\ \mathbf{Let}\quad VariableFresca\ (\mathbf{Clave}(tbl_1),\ \mathbf{Estructura}\ (tbl_1),\ database_type) : \mathbf{Bool} = \\ \qquad\qquad\qquad cmp_1 \in cmp_2(registros), \\ \qquad bd' = \mathbf{Delete}(tbl_1)(bd,\ VariableFresca), \\ \mathbf{In} \\ exp\end{array}}\ Cond_1$$

$$Cond_1 \equiv tabla = tbl_2$$

$$\frac{\ll refs,\ bd,\ tabla,\ registros \gg_{OnCascadeDelete} \rightarrow exp}{\ll \mathbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ tabla,\ registros \gg_{OnCascadeDelete} \rightarrow exp}\ Cond_2$$

$$Cond_2 \equiv tabla \neq tbl_2$$

$$\frac{}{\ll \epsilon_{referencias},\ bd,\ tabla,\ registros \gg_{OnCascadeDelete} \rightarrow bd}$$

La función $\ll\gg_{OnCascadeDelete}$ examina las referencias a la tabla sobre la cual se realizo la eliminación de registros para hacer una nueva eliminación sobre estas tablas de los registros que apunten a registros que han sido eliminados.

Asumiremos contar con una función auxiliar **GeneradorVariablesFrescas()** utilizada para obtener nombres de variables frescos.

Al igual que con la operación *Select*, utilizaremos una convención similar para nombrar las funciones de eliminación, dada por la función **Delete**.

$$\mathbf{Delete} :: String \rightarrow Exp$$

$$\mathbf{Delete}\ str = \text{“}delete_\text{”} + str$$

3.3.3. Update

Para la especificación del operador de Bases de Datos *Update* se define la familia de funciones $\ll\gg_{Update}$. Al igual que para las demás operaciones, las reglas generan un conjunto de declaraciones de funciones que implementan el operador *Update* para las distintas tablas del sistema.

Cada una de estas funciones, además del predicado de selección y la Base de Datos sobre la cual se realizará la operación, debe tomar una función de actualización que especifica como se deben realizar los cambios en los campos de cada registro.

$$\ll\gg_{Update} :: \ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Update} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Update} \rightarrow decl}$$

$$\ll\gg_{Update} :: \ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Update} \rightarrow decl_1 \qquad \ll tablas \gg_{Update} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Update} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Update} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Update} :: \ Tabla \rightarrow Decl$$

$$\frac{\begin{array}{c}\ll \textbf{Referencias},\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow exp \\ \ll campos \gg_{ActualizaCampos} \rightarrow asignaciones \\ \ll campos,\ bd,\ nombre,\ tipo,\ condiciones \gg_{Update} \rightarrow letBindings\end{array}}{\begin{array}{l}\qquad\qquad \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Update} \rightarrow \\ \textbf{Fun}\ update_nombre\ (bd : database_type,\ selector : \textbf{TipoSelector}(nombre), \\ \qquad\qquad updater : \textbf{TipoUpdater}(nombre))\ database_type \\ \quad \textbf{Let} \quad condiciones(x : tipo) = exp, \\ \quad\qquad letBindings, \\ \quad\qquad nombre' = (\#\ clave := clave(nombre(bd)),\ asignaciones\ \#), \\ \quad\qquad bd' = bd\ \textbf{With}\ (nombre := nombre') \\ \quad \textbf{In} \\ \quad bd'\end{array}}$$

$$\ll\gg_{Update} :: \ Campos \times String \times String \times String \times String \rightarrow LetBinding^{+}$$

$$\frac{\begin{array}{c}\ll \textbf{Estructura}(tabla),\ bd,\ tabla,\ nombre,\ x \gg_{ObtieneCampos} \rightarrow CamposObtenidos \\ \ll bd,\ tabla,\ nombre,\ x \gg_{ObtieneCampo} \rightarrow CampoObtenido \\ \ll campos,\ bd,\ tabla,\ clave,\ tipoClave,\ condiciones \gg_{Update} \rightarrow letBindings\end{array}}{\begin{array}{l}\ll \textbf{Campos}\ nombre\ tipo\ campos,\ bd,\ tabla,\ tipoClave,\ condiciones \gg_{Update} \rightarrow \\ nombre'\ (x : tipoClave) : tipo = \\ \ \textbf{Cond}\ \ (\sim selector(x,\ camposObtenidos) \rightarrow campoObtenido, \\ \qquad\quad \sim condiciones(x) \rightarrow campoObtenido, \\ \qquad\quad \textbf{Else} \rightarrow nombre(updater(x,\ camposObtenidos))) \\ \ \textbf{EndCond}, \\ letBindings\end{array}}$$

$$\frac{}{\ll \epsilon_{campos},\ bd,\ tabla,\ tipoClave,\ condiciones \gg_{Update} \rightarrow \epsilon_{letbinding}}$$

Por cada tabla se generan las condiciones que deben satisfacer las actualizaciones de los registros para poder ser llevadas a cabo. Se definen las actualizaciones para cada campo de la tabla siempre que el registro correspondiente sea seleccionado por el predicado de selección y cumpla las condiciones antes generadas. Finalmente se actualizan la Base de Datos con los registros modificados.

La función auxiliar $\ll\gg_{CondUpdate}$ es utilizada para generar todas las condiciones que debe satisfacer cada registro para poder ser actualizado. Estas condiciones son las relacionadas con la integridad referencial que debe existir entre las tablas de la Base de Datos. Para esto se utilizan las referencias que existen entre las tablas.

$$\ll\gg_{CondUpdate} ::\ Referencias \times String \times String \times String \times String \rightarrow Exp$$

$$\frac{\begin{array}{c}\ll refs,\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow exp \\ \ll \textbf{Estructura}(nombre),\ bd,\ tabla,\ clave \gg_{ObtieneCampos} \rightarrow campos\end{array}}{\begin{array}{l}\ll \textbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow \\ \ \textbf{Let}\ \ fila = update(x, campos), \\ \qquad\ valorCampo = cmp_1(fila) \\ \ \textbf{In} \\ valorCampo \in cmp_2(tbl_2(bd))\ \wedge\ exp\end{array}}\ Cond_1$$

$$Cond_1 \equiv tbl_1 = nombre$$

$$\frac{\ll refs,\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow exp}{\ll \textbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow exp}\ Cond_2$$

$$Cond_2 \equiv tbl_1 \neq nombre$$

$$\frac{}{\ll \epsilon_{referencias},\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow true}$$

La función $\ll\gg_{ActualizaCampos}$ es utilizada para generar las asignaciones correspondientes para actualizar la tabla en cuestión. Se utiliza la siguiente convención:

> Las funciones que encapsulan las columnas de la tabla ya actualizadas son nombradas con el nombre original de la columna primado.

Estos nombres son generados por la llamada recursiva de $\ll\gg_{Update}$ (en la regla correspondiente a $Tabla$) y utilizados en $\ll\gg_{ActualizaCampos}$ de la siguiente manera:

$$\ll\gg_{ActualizaCampos} ::\ Campos \rightarrow Asignaciones$$

$$\frac{\ll campos \gg_{ActualizaCampos} \rightarrow asignaciones}{\ll \mathbf{Campos}\ nombre\ tipo\ campos \gg_{ActualizaCampos} \rightarrow nombre := nombre',\ asignaciones}$$

$$\frac{}{\ll \epsilon_{campos} \gg_{ActualizaCampos} \rightarrow \epsilon_{asignaciones}}$$

Las funciones $\ll\gg_{ObtieneCampos}$ y $\ll\gg_{ObtieneCampo}$ son funciones que ayudan a la construcción de tipos de datos necesarios para la simplificación de las reglas de actualización.

$$\ll\gg_{ObtieneCampos} ::\ TypedIds \times String \times String \times String \rightarrow Argumentos$$

$$\frac{\ll typedIds,\ bd,\ tabla,\ clave \gg_{ObtieneCampos} \rightarrow arg}{\begin{array}{c}\ll nombre : tipo\ typedIds,\ bd,\ tabla,\ clave \gg_{ObtieneCampos} \rightarrow \\ \mathbf{Arg}\ (nombre(tabla(bd))(clave))\ arg\end{array}}$$

$$\frac{}{\ll \epsilon_{typedids},\ bd,\ tabla,\ clave \gg_{ObtieneCampos} \rightarrow \epsilon_{argumentos}}$$

$$\ll\gg_{ObtieneCampo} ::\ String \times String \times String \times String \rightarrow Argumentos$$

$$\frac{}{\ll bd,\ tabla,\ campo,\ clave \gg_{ObtieneCampo} \rightarrow \mathbf{Arg}\ (nombre(tabla(bd))(clave))\ \epsilon_{argumentos}}$$

Para la utilización de las operaciones de actualización en las distintas tablas, seguiremos una convención similar al resto de las operaciones:

$$\mathbf{Update} :: String \rightarrow Exp$$

$$\mathbf{Update}\ str = \text{``}update_\text{''} + str$$

3.3.4. Insert

Solo queda definir la familia de funciones $\ll\gg_{Insert}$ para la especificación del operador *Insert*. Este operador es particularmente distinto a los anteriores ya que para la inserción de un registro se deben dar explícitamente los valores de los campos a insertar, como se vio en la sección 2.3.2.

$$\ll\gg_{Insert} ::\ BaseDeDatos \rightarrow Decl$$

$$\frac{\ll tablas \gg_{Insert} \rightarrow decl}{\ll \textbf{BaseDeDatos}\ tablas \gg_{Insert} \rightarrow decl}$$

$$\ll\gg_{Insert} ::\ Tablas \rightarrow Decl$$

$$\frac{\ll tabla \gg_{Insert} \rightarrow decl_1 \qquad \ll tablas \gg_{Insert} \rightarrow decl_2}{\ll \textbf{Tablas}\ tabla\ tablas \gg_{Insert} \rightarrow decl_1,\ decl_2}$$

$$\frac{}{\ll \epsilon_{tablas} \gg_{Insert} \rightarrow \epsilon_{decl}}$$

$$\ll\gg_{Insert} ::\ Tabla \rightarrow Decl$$

$$\frac{\begin{array}{c} claveFresca = \textbf{GeneradorVariablesFrescas}() \\ \ll \textbf{Estructura}(nombre) \gg_{ParametrosFrescos} \rightarrow parFrescos \\ \ll \textbf{Referencias},\ bd,\ nombre,\ parFrescos \gg_{CondInsert} \rightarrow exp \\ \ll campos,\ bd,\ nombre,\ claveFresca,\ parFrescos \gg_{Insert} \rightarrow asignaciones \end{array}}{\begin{array}{l} \qquad\qquad \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos \gg_{Insert} \rightarrow \\ \textbf{Fun}\ insert_nombre\ (bd : database_type,\ claveFresca : tipo,\ parFrescos)\ database_type \\ \qquad \textbf{Let} \quad condiciones = exp, \\ \qquad\qquad clave' = add(claveFresca,\ clave(nombre(bd))), \\ \qquad\qquad nombre' = (\#\ clave',\ asignaciones\ \#), \\ \qquad\qquad bd' = \textbf{Cond}\ (\ claveFresca \in clave(nombre(bd)) \rightarrow bd, \\ \qquad\qquad\qquad\qquad \sim condiciones \rightarrow bd, \\ \qquad\qquad\qquad\qquad \textbf{Else} \rightarrow bd\ \textbf{With}\ (nombre := nombre') \\ \qquad\qquad\qquad \textbf{EndCond}) \\ \qquad \textbf{In} \\ \qquad bd' \end{array}}$$

$$\ll\gg_{Insert} ::\ Campos \times String \times String \times String \times Argumentos \rightarrow Asignaciones$$

$$\frac{\ll campos,\ bd,\ tabla,\ claveFresca,\ parFrescos \gg_{Insert}\rightarrow asignaciones}{\begin{array}{c}\ll \textbf{Campos}\ nombre\ tipo\ campos,\ bd,\ tabla,\ claveFresca,\ \textbf{Arg}\ arg\ parFrescos \gg_{Insert}\rightarrow \\ nombre := nombre(tabla(bd))\ \textbf{With}\ [claveFresca := arg],\ asignaciones\end{array}}$$

$$\frac{}{\ll \epsilon_{campos},\ bd,\ tabla,\ claveFresca,\ \epsilon_{argumentos} \gg_{Insert}\rightarrow \epsilon_{asignaciones}}$$

Para cada tabla se generan las condiciones que debe satisfacer el registro que se desea insertar. Si la clave no se encuentra en la tabla y el registro satisface las condiciones antes definidas, se inserta en la tabla y se actualiza la Base de Datos.

La función auxiliar $\ll\gg_{CondInsert}$ es utilizada para generar las condiciones que debe satisfacer el registro que será insertado. Estas condiciones son las relacionadas con la integridad referencial que debe existir entre las tablas de la Base de Datos.

$$\ll\gg_{CondInsert} ::\ Referencias \times String \times String \times TypedIds \rightarrow Exp$$

$$\frac{\ll refs,\ bd,\ tabla,\ parFrescos \gg_{CondInsert}\rightarrow exp}{\begin{array}{c}\ll \textbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ tabla,\ parFrescos \gg_{CondInsert}\rightarrow \\ cmp_1(parFrescos) \in cmp_2(tbl_2(bd)) \wedge exp\end{array}}\ Cond_1$$

$$Cond_1 \equiv tbl_1 = tabla$$

$$\frac{\ll refs,\ bd,\ tabla,\ parFrescos \gg_{CondInsert}\rightarrow exp}{\ll \textbf{Referencias}\ tbl_1\ cmp_1\ tbl_2\ cmp_2\ refs,\ bd,\ tabla,\ parFrescos \gg_{CondInsert}\rightarrow exp}\ Cond_2$$

$$Cond_2 \equiv tbl_1 \neq tabla$$

$$\frac{}{\ll \epsilon_{referencias},\ bd,\ tabla,\ parFrescos \gg_{CondInsert}\rightarrow true}$$

La función $\ll\gg_{ParametrosFrescos}$ genera parámetros frescos para los campos del registro a insertar.

$$\ll\gg_{ParametrosFrescos} ::\ TypedIds \rightarrow TypedIds$$

$$\frac{\begin{array}{c}\ll typedIds \gg_{ParametrosFrescos}\rightarrow parFrescos \\ varFresca = \textbf{GeneradorVariablesFrescas()}\end{array}}{\ll nombre : tipo,\ typedIds \gg_{ParametrosFrescos}\rightarrow varFresca : tipo,\ parFrescos}$$

$$\overline{\ll \epsilon_{typedids} \gg_{ParametrosFrescos} \rightarrow \epsilon_{typedids}}$$

Para las operaciones de inserción seguiremos una convención similar al resto de las operaciones dada por la siguiente función:

$$\textbf{Insert} :: String \rightarrow Exp$$

$$\textbf{Insert}\ str = \text{“}insert_\text{”} + str$$

3.4. Incorporando triggers

Una vez especificadas las reglas que definen las operaciones básicas de una Base de Datos, se extiende el enfoque para la incorporación de los triggers. Para esto se modifica la sintaxis utilizada para la declaración de las tablas y se definen nuevas categorías sintácticas para los triggers.

3.4.1. Definiciones

Extendemos la sintaxis vista en la sección 3.1, utilizada para la declaración de las tablas, a fin de contemplar la incorporación de los triggers en las operaciones de Base de Datos. Tendremos en cuenta solamente los *after* triggers sobre las operaciones de eliminación y actualización de registros.

$$
\begin{array}{rcl}
BaseDeDatos & ::= & \textbf{BaseDeDatos}\ Tablas \\
\\
Tablas & ::= & \textbf{Tablas}\ Tabla\ Tablas \\
 & & \mid \epsilon_{tablas} \\
\\
Tabla & ::= & \textbf{Tabla}\ nombre\ clave\ tipo\ Campos\ Decl_{Tr}\ Decl_{Tr} \\
\\
Campos & ::= & \textbf{Campos}\ nombre\ tipo\ Campos \\
 & & \mid \epsilon_{campos}
\end{array}
$$

Se incorporan dos nuevos parámetros en la categoría *Tabla*. El primero es el *after* trigger correspondiente a la operación *Delete* de la tabla, el segundo es el correspondiente al operador *Update*.

Por cuestiones de simplicidad y de clarificación, para la declaración de los triggers consideraremos un lenguaje reducido dado por la siguiente gramática:

$$
\begin{array}{rcl}
Decl_{Tr} & ::= & Decl_{Tr},\ Decl_{Tr} \\
 & & \mid \textbf{Delete From } String \textbf{ Where } Cond_{Tr} \\
 & & \mid \textbf{Update } String \textbf{ Set } Udt_{Tr} \textbf{ Where } Cond_{Tr} \\
 & & \mid \epsilon_{declTr} \\
 & & \\
Cond_{Tr} & ::= & \textbf{Exp } Exp_{Tr} \\
 & & \mid Exp_{Tr}\ [\textbf{Not}]\ \textbf{In } SubQ_{Tr} \\
 & & \mid \textbf{Exists } SubQ_{Tr} \\
 & & \mid Cond_{Tr}\ [\wedge \mid \vee]\ Cond_{Tr} \\
 & & \\
SubQ_{Tr} & ::= & \textbf{Select From } String \textbf{ Where } Cond_{Tr} \\
 & & \\
Udt_{Tr} & ::= & String = Exp_{Tr},\ Udt_{Tr} \\
 & & \mid \epsilon_{udtTr} \\
 & & \\
Exp_{Tr} & ::= & true \\
 & & \mid false \\
 & & \mid Numero \\
 & & \mid String \\
 & & \mid \textbf{Old}.String \\
 & & \mid Exp_{Tr}\ BinOp_{Tr}\ Exp_{Tr} \\
 & & \mid UnOp_{Tr}\ Exp_{Tr} \\
 & & \\
BinOp_{Tr} & ::= & + \mid - \mid * \mid / \mid \wedge \mid \vee \mid ... \\
 & & \\
UnOp_{Tr} & ::= & \sim\mid - \mid ...
\end{array}
$$

Un trigger puede agregar operaciones de eliminación y actualización sobre distintas tablas de la Base de Datos. Estas operaciones pueden contener condiciones de selección que contemplen subconsultas, como muesta la estructura sintáctica $Cond_{Tr}$. Para la operación de actualización se debe definir como serán los cambios en los campos de cada registro. Se puede acceder a los registros eliminados o modificados a través de la variable **Old**, presente en el entorno de ejecución del trigger.

3.4.2. Reglas modificadas

La modificación de la sintaxis utilizada para la declaración de las tablas lleva a una modificación en las reglas para el tratamiento de las mismas.

Utilizaremos la función $\ll\gg_{AfterTrigger}$ que toma la declaración de los triggers, junto a otros datos necesarios, como veremos más adelante. Como resultado retorna una secuencia de declaraciones auxiliares junto a una nueva Base de Datos modificado por el comportamiento del trigger. Esta función tiene la siguiente signatura.

$$\ll\gg_{AfterTrigger} :: \; Decl_{Tr} \times String \times String \times String \rightarrow LetBinding^{+} \times String$$

Modificamos a continuación las reglas relacionadas con la definición de las operaciones de eliminación y actualización de las tablas.

$$\ll\gg_{Delete} :: \; Tabla \rightarrow Decl$$

$$\frac{\begin{array}{c}\ll \textbf{Referencias},\ bd',\ nombre,\ registrosBorrados \gg_{OnCascadeDelete} \rightarrow exp \\ \ll campos,\ bd,\ nombre \gg_{Delete} \rightarrow asign \qquad \ll \textbf{Estructura}(nombre) \gg_{Ids} \rightarrow argsIds \\ \ll triggers_1,\ bd'',\ nombre,\ clave,\ tipo, registrosBorrados \gg_{AfterTriggers} \\ \rightarrow < letBindings_{AfterTriggers}, bd_{AfterTriggers} > \end{array}}{\begin{array}{l} \qquad\qquad \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos\ triggers_1\ triggers_2 \gg_{Delete} \rightarrow \\ \textbf{Fun}\ delete_nombre\ (bd:\ database_type,\ selector:\ \textbf{TipoSelector}(nombre))\ database_type \\ \textbf{Let} \qquad clave' = \{x:\ \textbf{Tipo}\ tipo \mid x \in clave(nombre(bd)) \wedge \sim selector(x,\ argIds)\}, \\ \qquad\qquad nombre' = (\#\ clave',\ asign\ \#), \\ \qquad\qquad bd' = bd\ \textbf{With}\ (nombre := nombre'), \\ \qquad\qquad registrosBorrados = \textbf{Select}(nombre)(bd, selector), \\ \qquad\qquad bd'' = exp \\ \qquad\qquad letBindings_{AfterTriggers} \\ \textbf{In} \\ bd_{AfterTriggers} \end{array}}$$

$$\ll\gg_{Update} :: \; Tabla \rightarrow Decl$$

$$\frac{\begin{array}{c}\ll \textbf{Referencias},\ bd,\ nombre,\ x,\ updater \gg_{CondUpdate} \rightarrow exp \\ \ll campos \gg_{ActualizaCampos} \rightarrow asignaciones \\ \ll campos,\ bd,\ nombre,\ tipo,\ condiciones \gg_{Update} \rightarrow letBindings \\ \ll triggers_2,\ bd',\ nombre,\ clave,\ tipo, registrosModificados \gg_{AfterTriggers} \\ \rightarrow < letBindings_{AfterTriggers}, bd_{AfterTriggers} > \end{array}}{\begin{array}{l} \qquad\qquad \ll \textbf{Tabla}\ nombre\ clave\ tipo\ campos\ triggers_1\ triggers_2 \gg_{Update} \rightarrow \\ \textbf{Fun}\ update_nombre\ (bd: database_type,\ selector: \textbf{TipoSelector}(nombre), \\ \qquad\qquad updater: \textbf{TipoUpdater}(nombre))\ database_type \\ \textbf{Let} \qquad condiciones(x: tipo) = exp, \\ \qquad\qquad letBindings, \\ \qquad\qquad nombre' = (\#\ clave := clave(nombre(bd)),\ asignaciones\ \#), \\ \qquad\qquad bd' = bd\ \textbf{With}\ (nombre := nombre') \\ \qquad\qquad registrosModificados = \textbf{Select}(nombre)(bd, selector), \\ \qquad\qquad letBindings_{AfterTriggers} \\ \textbf{In} \\ bd_{AfterTriggers} \end{array}}$$

3.4.3. Reglas de triggers

Finalmente se deben declarar las reglas para el tratamiento de los triggers, para esto se define una familia de funciones $\ll\gg_{AfterTrigger}$.

La función $\ll\gg_{AfterTrigger}$ utilizada anteriormente en la definición de los operadores *Delete* y *Update* retorna una secuencia de acciones a realizar luego de que se lleve a cabo la operación, junto a la Base de Datos modificada por las acciones definidas.

$$\ll\gg_{AfterTrigger} :: \; Decl_{Tr} \times String \times String \times String \rightarrow LetBinding^{+} \times String$$

$$\frac{\begin{array}{c}\ll D_1,\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow < letBindings_1,\ bd_1 > \\ \ll D_2,\ bd_1,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow < letBindings_2,\ bd_2 >\end{array}}{\ll (D_1,\ D_2),\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow < (letBindings_1,\ letBindings_2),\ bd_2 >}$$

$$\frac{\ll cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow exp}{\begin{array}{c}\ll \textbf{Delete From}\ tabla_1\ \textbf{Where}\ cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow \\ < selector : \textbf{TipoSelector}(tabla_1) = \exists\ old : \textbf{TipoClave}(tabla_2)\ \centerdot\ old \in filas\ \wedge\ exp, \\ bd' = \textbf{Delete}(tabla_1)(bd,\ selector),\ bd' >\end{array}}$$

$$\frac{\ll cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow exp_1 \qquad \ll upd \gg_{AfterTrigger} \rightarrow exp_2}{\begin{array}{c}\ll \textbf{Update}\ tabla_1\ \textbf{Set}\ udt\ \textbf{Where}\ cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow \\ < selector : \textbf{TipoSelector}(tabla_1) = \exists\ old : \textbf{TipoClave}(tabla_2)\ \centerdot\ old \in filas\ \wedge\ exp_1, \\ actualizador : \textbf{TipoActualizador}(tabla_1) = exp_2, \\ bd' = \textbf{Update}(tabla_1)(bd,\ selector,\ actualizador),\ bd' >\end{array}}$$

$$\frac{}{\ll \epsilon_{declTr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow < \epsilon_{letBinding^{+}},\ bd >}$$

Obsérvese que en la regla relacionada con la operación *Update*, dentro de la definición de la función de actualización no puede haber referencias a la variable **Old**, ya que no esta cuantificada como en la función de selección. El tratamiento de la variable **Old** en la función de actualización es un punto de partida para futuras investigaciones.

También se deben definir reglas para el tratamiento de las condiciones de selección, dadas por la gramática $Cond_{Tr}$.

$$\ll\gg_{AfterTrigger} :: \; Cond_{Tr} \times String \times String \times String \rightarrow Exp$$

$$\frac{\ll exp_{Tr} \gg_{AfterTrigger} \rightarrow exp}{\ll \textbf{Exp}\ exp_{Tr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow exp}$$

$$\frac{\ll subQ_{Tr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow subQ \qquad \ll exp_{Tr} \gg_{AfterTrigger} \rightarrow exp}{\ll exp_{Tr}\ \textbf{In}\ subQ_{Tr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow exp \in subQ}$$

$$\frac{\ll subQ_{Tr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow subQ \qquad \ll exp_{Tr} \gg_{AfterTrigger} \rightarrow exp}{\ll exp_{Tr}\ \textbf{Not In}\ subQ_{Tr},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow \sim exp \in subQ}$$

$$\frac{\ll \textbf{Select From}\ tabla_1\ \textbf{Where}\ cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow subQ \qquad x = \textbf{GeneradorVariablesFrescas()}}{\ll \textbf{Exists}\ (\textbf{Select From}\ tabla_1\ \textbf{Where}\ cond),\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow \exists\ x : \textbf{TipoClave}(tabla_1) \cdot x \in subQ}$$

$$\frac{\ll cond_{Tr1}\ [\wedge, \vee]\ cond_{Tr2},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow cond_1 \qquad \ll cond_{Tr2}\ [\wedge, \vee]\ cond_{Tr2},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow cond_2}{\ll cond_{Tr1}\ [\wedge \mid \vee]\ cond_{Tr2},\ bd,\ tabla,\ filas \gg_{AfterTrigger} \rightarrow cond_1\ [\wedge \mid \vee]\ cond_2}$$

De la misma forma, para la consulta de registros se define la siguiente función:

$$\ll\gg_{AfterTrigger} ::\ SubQ_{Tr} \times String \times String \times String \rightarrow Exp$$

$$\frac{\ll cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow exp}{\begin{array}{l}\ll \textbf{Select From}\ tabla_1\ \textbf{Where}\ cond,\ bd,\ tabla_2,\ filas \gg_{AfterTrigger} \rightarrow \\ \textbf{Let}\quad selector : \textbf{TipoSelector}(tabla_1) = \exists\ old : \textbf{TipoClave}(tabla_2)\ \textbf{.} \\ \qquad\qquad\qquad\qquad old \in filas\ \wedge\ exp, \\ \textbf{In} \\ \textbf{Select}(tabla_1)(bd, selector)\end{array}}$$

Por último, la función utilizada para el tratamiento de las expresiones en el lenguaje de triggers, sólo lleva cada elemento sintáctico a su equivalente en las expresiones definidas en la sección 3.1.

$$\ll\gg_{AfterTrigger} ::\ Exp_{Tr} \rightarrow Exp$$

$$\frac{}{\ll true \gg_{AfterTrigger} \rightarrow true}$$

$$\frac{}{\ll false \gg_{AfterTrigger} \rightarrow false}$$

$$\frac{}{\ll numero \gg_{AfterTrigger} \rightarrow numero}$$

$$\frac{}{\ll string \gg_{AfterTrigger} \rightarrow string}$$

$$\frac{}{\ll \textbf{Old}.campo \gg_{AfterTrigger} \rightarrow campo(old)}$$

$$\frac{\ll exp_{Tr1} \gg_{AfterTrigger} \rightarrow exp_1 \qquad \ll exp_{Tr2} \gg_{AfterTrigger} \rightarrow exp_2}{\ll exp_{Tr1} \otimes exp_{Tr2} \gg_{AfterTrigger} \rightarrow exp_1 \otimes exp_2}$$

$$\frac{\ll exp_{Tr1} \gg_{AfterTrigger} \rightarrow exp_1}{\ll \odot\ exp_{Tr1} \gg_{AfterTrigger} \rightarrow \odot\ exp_1}$$

Donde $\otimes$ y $\odot$ están cuantificados sobre las operaciones binarias y unarias respectivamente, definidas en Exp_{Tr} y Exp.

Capítulo 4

Conclusiones y trabajos futuros

Un metodología ampliamente utilizado para la verificación de propiedades de software consiste en la creación de una especificación formal de la aplicación, sobre la cual se realizaran las verificaciones pertinentes. Luego este modelo es utilizado como base para el desarrollo del programa propiamente dicho. Si bien este proceso puede ser realizado formalmente utilizando herramientas como Coq, en la mayoría de los casos un programador construye la aplicación a partir de la especificación sin ninguna relación formal entre ambas.

Otro enfoque es la construcción de modelos a partir de una aplicación ya desarrollada para la verificación de propiedades sobre este modelo. Una vez más, la relación entre estas entidades no puede ser establecida formalmente.

La metodología propuesta en este trabajo ofrece otra perspectiva a este problema. A partir de una aplicación ya desarrollada a través de la utilización de *triggers* de Base de Datos, se extrae una especificación formal sobre la cual se realizarán las verificaciones formales de las propiedades deseadas. Obsérvese que en este enfoque, la relación formal entre el software y la especificación esta garantizada por las reglas de traducción definidas a lo largo de este trabajo.

Algunos trabajos relacionados como [2, 3, 11] definen nuevos lenguajes, sobre los cuales se pueden demostrar propiedades, que luego son traducidos a operaciones de Base de Datos, asegurando de esta forma la relación entre la especificación y la aplicación. Nuestro enfoque propone la utilización del estándar SQL para la generación del modelo evitando de esta forma la utilización de un nuevo lenguaje.

Trabajos futuros en esta linea pueden brindar un carácter más maduro a este enfoque, entre estos encontramos los siguientes:

- En la sección 3.4, se utilizó un lenguaje para la definición de triggers que puede ser ampliado. La incorporación de sentencias que faciliten el control de flujo y la inserción masiva de registros en un trigger son características deseadas para la definición de los

mismos.

- Un estudio sobre la forma general de modelar *after triggers*. Este tipo de triggers tienen una complejidad adicional que es la capacidad de inhibir la operación sobre la cual se esta definiendo. En el capitulo 2 se dan algunos lineamientos que pueden ser utilizados como ayuda para el tratamiento general de este tipo de triggers.

- En este trabajo se presento la definición de triggers a nivel registro. Sin embargo, muchos Motores de Base de Datos, entre ellos PostgreSQL, soportan triggers a nivel de procedimiento. Un estudio del tratamiento de este tipo de triggers ayudaría sin duda a expandir los alcances de este trabajo.

- El uso de *Joins* entre tablas de una Base de Datos no fue contemplado en este trabajo. La definición de estos es un área pendiente a investigar.

- Otros Motores de Base de Datos ofrecen lenguajes similares a PL/pgSQL, como Transact-SQL. Estudiar la posible aplicabilidad de las reglas definidas a estos lenguajes ayudaría a lograr una verdadera interoperabilidad de este enfoque.

- Es imprescindible demostrar que las reglas de traducción fueron definidas correctamente. Para esto es necesario probar que la traducción preserva la semántica del lenguaje utilizado para la definición de los triggers.

- El tratamiento de *Nulls* de forma explicita en los campos de la Base de Datos fue dejamos para futuros trabajos.

Finalmente, una herramienta que realice las traducciones necesarias y posiblemente asista en la demostración de las propiedades es un trabajo que impulsaría fuertemente el uso de estas técnicas en aplicaciones reales.

Bibliografía

[1] Ralph-Johan Back, Mikołaj Olszewski, and Damián Soriano. Deseo meeting scheduler: Towards automated verification of database integrity constraints. *Symposium on Automatic Program Verification, Universidad Nacional de Río Cuarto*, Febrero 2009.

[2] Véronique Benzaken, Serenella Cerrito, and Sébastien Praud. Static verification of dynamical integrity constraints a semantics based approach. *Networking and Information Systems Journal*, 10, 2000.

[3] Véronique Benzaken and Xavier Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *In Proceedings of ECOOP, volume 1241 of LNCS*, pages 60–85. Springer-Verlag, 1997.

[4] C. J. Date. *An Introduction to Database Systems*. Addison Wesley, 2003.

[5] Joshua D. Drake and John C. Worsley. *Practical PostgreSQL*. O'Reilly, 2002.

[6] Peter Eisentraut and Bernd Helmle. *PostgreSQL Administration*. O'Reilly, 2008.

[7] S. Owre and N. Shankar. *The Formal Semantic of PVS*. SRI International, Computer Science Laboratory, March 1999.

[8] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, Computer Science Laboratory, Noviembre 2001.

[9] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, Computer Science Laboratory, Noviembre 2001.

[10] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, Computer Science Laboratory, Noviembre 2001.

[11] Tim Sheard and David Stemp. *Automatic Verification of Database Transaction Safety*. University of Massachusetts, 1989.

[12] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fifth edition, 2005.

[13] M. Stonebraker and L. A. Rowe. *The Design of POSTGRES*. Department of Electrical Engineering and Computer Sciences, University of California, 1986.