Trabajo Fin de Máster en Programación y Tecnología Software

Máster en Investigación en Informática

Facultad de Informática
Universidad Complutense de Madrid

MySQL4OCL: Un compilador de OCL a MySQL.

Carolina Inés Dania

Director: Manuel García Clavel
Colaboradora externa: Marina Soledad Egea González

Curso académico 2010/2011
Departamento de Sistemas Informáticos y Computación

Calificación: 8.5

Autorización de difusión

La abajo firmante, matriculada en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Máster: *MySQL4OCL: Un compilador de OCL a MySQL.*, realizado durante el curso académico 2010-2011 bajo la dirección de Manuel García Clavel y con la colaboración externa de dirección de Marina Soledad Egea González en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En Madrid, a los 8 días del mes de septiembre de 2011,

Carolina Inés Dania
30473217N

Abstract

In this work we present a compiler from OCL to MySQL that allows the automatic evaluation of OCL expressions on relational databases. This compiler, called MySQL4OCL, is defined as a recursive function on OCL expressions and covers a wide subset of the OCL language. The main ideas underlying the definition of MySQL4OCL were initially proposed in [16]: namely, the use of "stored-procedures" for the translation of iterator operations over collections.

As part of this work, we have also implemented MySQL4OCL as a Java component [17], which is designed to be integrated in modeling tools supporting the OCL language. In particular, MySQL4OCL is one of the key components of a model-based development framework called ActionGUI [8]. This framework allows the automatic generation of database management applications with access control policies. The first release of ActionGUI was presented in [13] and the methodology for software development that it implements is explained in [1].

Keywords: databases, models, compiler, tool, UML, OCL, MDA, SQL y MySQL.

Resumen

En este trabajo presentamos un compilador de OCL en MySQL que permite la evaluación automática de expresiones OCL sobre bases de datos relacionales. Este compilador, denominado MySQL4OCL, se define como una función recursiva sobre expresiones OCL y cubre un subconjunto muy significativo del lenguaje. Las ideas principales que subyacen a la definición de MySQL4OCL fueron inicialmente propuestas en [16]: a saber, la utilización de procedimiento almacenados ("stored-procedures") para la traducción de operaciones iteradoras sobre colecciones.

Además, como parte de este trabajo, hemos implementado MySQL4OCL como un componente Java [17], que está diseñado para su integración en herramientas de modelado que den soporte al lenguaje OCL. En particular, MySQL-4OCL es uno de los componentes principales del entorno de desarrollo basado en modelos ActionGUI [8]. Este entorno permite la generación automática de aplicaciones para la gestión de bases de datos con políticas de control de acceso. La primera versión de ActionGUI fue presentada en [13] y la metodología de desarrollo que implementa es objeto del tutorial [1].

Palabras claves: base de datos, modelos, compilador, herramienta, UML, OCL, MDA, SQL y MySQL.

Agradecimientos

En primer lugar, quiero agradecer a Manuel García Clavel por ser quien confió en mi, me dio la oportunidad de comenzar mi carrera de investigación en España y me adoptó bajo su supervisión para hacer mis primeros pasos en investigación.

En segundo lugar, quiero agradecer a Marina Egea González por ser también mi supervisora y por el apoyo cotidiano que me brindó cada día.

Este trabajo no hubiese sido posible sin la cooperación de ellos.

Además, quiero agradecer a Miguel Angel García de Dios por su ayuda y asesoramiento con el parser de OCL.

También quiero agradecer a David De Frutos, Mercedes Merayo y Rafael Del Vado por haber formado parte del jurado que evaluó este trabajo. En particular, a Rafael Del Vado por la gran cantidad de valiosos comentarios que me proporcionó.

Un agradecimiento especial a Narciso Martí Oliet por todo su apoyo y su buena disposición a lo largo del master.

A IMDEA Software por proporcionarme los medios para realizar mi investigación y a su gente por hacer tan agradable el entorno de trabajo.

A Alejandro, Javier y Julián por ser quienes me han acompañado en esta etapa del master.

Por último, quiero agradecer a aquellas personas que desde la otra parte del planeta me están apoyando. En primer lugar a Pedro, por ser la primer persona en confiar en mí y en apoyarme en la decisión de venirme aquí. En segundo lugar, a mi familia por ser quienes constantemente me apoyan en cada paso que decido emprender. Y por último, a todos mis amigos de allá que siempre están bancandome en todo.

Índice general

1 Introducción

1.1 Antecedentes

El trabajo presente se ha desarrollado dentro de la investigación en tecnologías de desarrollo de software basado en modelos que lleva a cabo el Laboratorio de Modelado de Software del Instituto de Investigación IMDEA Software en colaboración con el grupo FADoSS de la Universidad Complutense de Madrid.

Una de las primeras líneas de investigación emprendidas por el Laboratorio de Modelado fue la definición de una semántica formal para el lenguaje OCL [26]. Este lenguaje, que forma parte del estándar UML de la OMG [29], permite añadir restricciones a los modelos, así como realizar consultas sobre los mismos. Aunque OCL se puede utilizar en diferentes tipos de diagramas UML, su uso más extendido se da en la definición de invariantes para diagramas de clase y en la formalización de consultas sobre diagramas de objetos, es decir, sobre instancias o escenarios de diagramas de clase. Los resultados más importantes de esta primera línea de investigación aparecieron en la tesis doctoral de Marina Egea [15], en la que se propuso una semántica formal ejecutable de OCL utilizando la lógica de reescritura [6]. Básicamente, esta semántica permitía "compilar" las expresiones OCL sobre diagramas de clase y diagramas de objetos en teoría de reescritura, que, por construcción, eran Church-Rosser y terminantes, y que, por lo tanto, podían ejecutarse automáticamente en el sistema Maude [5]. Este compilador tenía un indudable valor teórico. Sin embargo, por razones de eficiencia, su uso en herramientas de modelado estaba muy limitado: el tiempo de carga en el sistema Maude de los escenarios sobre los que se realizarían las consultas OCL era manifiestamente excesivo, incluso para diagramas de objeto de tamaño mediano (esto es, con decenas de miles de objetos).

Aprovechando las ideas principales que subyacían a la semántica formal propuesta en [15], se inició entonces dentro del Laboratorio de Modelo el desarrollo de un interprete en Java para OCL que pudiera dar soporte eficaz a este lenguaje en herramientas de modelado. El resultado fue el componente EOS [7] que permitía efectivamente evaluar expresiones OCL sobre diagramas de objetos de tama
'uno mediano-grande (con cientos de miles de objetos) en un tiempo aceptable. Sin embargo, para diagramas de objetos realmente grandes (con millones de objetos), el tiempo de carga en memoria de los correspondientes escenarios seguía siendo excesivo. Por ejemplo, en las tablas comparativas que se incluian en [7] se advierte que todos los evaluadores de OCL implementados hasta ese

momento, incluyendo EOS, tardaban más de 45 segundos en cargar en memoria un escenario con 10^5 objetos y más de 20 minutos en hacerlo con un escenario con 10^6 objetos.[1]

A partir de las experiencias anteriores, se inició entonces una nueva línea de investigación dentro del Laboratorio de Modelado, orientada a resolver de forma definitiva el problema planteado por los escenarios realmente grandes. Para estos casos, la única solución que parecía viable era:

1. Almacenar los escenarios en bases de datos.

2. Compilar las consultas OCL como consultas sobre bases de datos;

3. Evaluar las consultas así compiladas sobre las bases de datos donde se hubieran almacenado los escenarios.

El objetivo de este trabajo de investigación era precisamente hacer posible esta solución, para lo cual debíamos, en primer lugar,

- Definir una esquema para el almacenamiento de escenarios en bases de datos.

Y, en segundo lugar,

- Definir una traducción de consultas OCL a consultas sobre bases de datos, teniendo en cuenta el esquema definido para el almacenamiento de los escenarios.

1.2 Contribuciones

Previa a nuestra investigación, ya existía en la literatura una primera propuesta para la compilación de OCL en SQL: a saber, los trabajos [34, 11, 12], que servían además como base para la herramienta OCL2SQL [28]. Sin embargo, la solución que ofrecia esta propuesta resultaba insatisfactoria, al menos por los siguientes motivos:

i. Unicamente cubría un conjunto muy limitado de *patrones* de expresiones OCL: en particular, no permitía expresiones con operadores iteradores anidados, ni expresiones con un tipo distinto de Boolean.

[1] En este punto, es importante señalar que tales escenarios no son meros supuestos teóricos. Consideremos, por ejemplo, el uso de OCL para evaluar métricas sobre programas Java. Desde el punto de vista del desarrollo de software basado en modelos, los programas Java son instancias del metamodelo de Java y sus métricas se pueden formalizar como consultas OCL, utilizando el lenguaje que proporciona este metamodelo. En este contexto, para ejecutar una métrica sobre un programa Java, no hay sino que evaluar la correspondiente consulta OCL sobre la instancia del metamodelo que corresponda a dicho programa: como es lógico, cuanto mayor sea el programa, mayor será la correspondiente instancia del metamodelo. Por ejemplo, el grupo Triskell de IRISA, en Francia, desarrolló una herramienta, llamada SpoonEMF, que genera para un programa Java de 10 líneas, un escenario con 113 objetos, y para un programa de apenas 500 líneas, un escenario con alrededor de 3,500 objetos. Es fácil imaginar que para programas Java de decenas de miles de líneas, el escenario resultante será de millones de objetos.

ii. Las consultas SQL compiladas eran extraordinariamente complejas, lo que se traducía en evaluaciones ineficientes sobre escenarios de tamaño mediano-grande.

Para superar las limitaciones de la propuesta anterior, en vez de utilizar patrones, decidimos basar nuestro compilador de OCL en una función de traducción definida recursivamente sobre las expresiones OCL. El resultado principal de nuestro trabajo es un compilador de OCL en MySQL que permite la evaluación automática de expresiones OCL sobre bases de datos relacionales. Este compilador, denominado MySQL4OCL, se define efectivamente como una *función recursiva* sobre expresiones OCL y cubre un subconjunto muy significativo del lenguaje. En concreto, MySQL4OCL es capaz de traducir expresiones que contienen iteradores en toda su generalidad, es decir, también cuando estas expresiones incluyen iteradores anidados. Las ideas iniciales que subyacen a la definición de MySQL4OCL, a saber, la utilización de procedimientos almacenados ("stored-procedures"), fueron presentados en el Workshop on OCL and Textual Modelling, asociado a la conferencia MODELS 2010 (Oslo, 2010), y posteriormente publicados en:

- M. Egea, C. Dania y M. Clavel. MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. Electronic Communications of the EASST, 36, 2010.

Además, como parte de nuestro trabajo de investigación, hemos implementado MySQL4OCL como un componente Java, diseñado especificamente para su integración en herramientas de modelado que den soporten el lenguaje OCL. Este componente, junto con su documentación básica y una colección de ejemplos, está disponible en:

- M. Egea, C. Dania y M. Clavel. The MySQL4OCL Code Generator. August 2010. `http://www.bm1software.com/mysql-ocl`.

Es interesante destacar que MySQL4OCL es de hecho uno de los componentes principalísimos del entorno de desarrollo basado en modelos ActionGUI. Este entorno permite la generación automática de aplicaciones para la gestión de bases de datos con políticas de control de acceso. Los modelos que maneja ActionGUI incluyen expresiones OCL, a distintos niveles y con distintos objetivos: en todos los casos, MySQL4OCL es el componente encargado de generar las consultas MySQL correspondientes a estas expresiones. La primera versión del entorno ActionGUI, denominada por aquel entonces SSG, fue presentada en ICSE 2010:

- M. A. García de Dios, C. Dania, M. Schläpfer, D. A. Basin, M. Clavel y M. Egea. SSG: A Model-Based Development Environment for Smart, Security-Aware GUIs. En J. Kramer, J. Bishop, P. T. Devanbu y S. Uchitel, editores, *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010* (Cape Town, South Africa, 1-8 Mayo 2010), páginas 311-312. ACM, 2010.

Más recientemente, la metodología de desarrollo que implementa ActionGUI —y en la que juega, como se ha mencionado anteriormente, un papel importantísimo MySQL4OCL— ha sido objeto de un curso específico dentro de FOSAD 2011:

- David A. Basin, Manuel Clavel, Marina Egea, Miguel Angel García de Dios, Carolina Dania, Gonzalo Ortiz y Javier Valdazo. Model-Driven Development of Security-Aware GUIs for Data-Centric Applications. En Alessandro Aldini y Roberto Gorrieri, editores, *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volumen 6858 de Lecture Notes in Computer Science, páginas 101-124. Springer, 2011.

Finalmente, este trabajo forma parte de las contribuciones del Laboratorio de Modelado de IMDEA Software en los siguientes proyectos:

- NESSoS. Red de Excelencia (NoE 256980) del 7 Programa Marco sobre "Engineering Secure Future Internet Software Services and Systems". Octubre 2010-Abril 2014.
- DESAFIOS-10. Proyecto del MICINN (TIN2009-14599-C03) sobre "Desarrollo de Software de Alta Calidad, Fiable, Distribuido y Seguro". Enero 2010-Diciembre 2012.
- PROMETIDOS-CM. Programa de la Comunidad de Madrid (S2009/TIC-1465) sobre "Métodos Rigurosos de Desarrollo de Software". Enero 2010-Diciembre 2012.

La autora agradece la ayuda financiera recibida de los proyectos anteriormente mencionados para la realización de su trabajo, y muy especialmente del proyecto DESAFIOS-10, por el que ha sido contratada desde octubre de 2010 a octubre de 2011.

1.3 Trabajo relacionado

En el ámbito general de las traducciones de OCL a lenguajes básados en SQL, los únicos trabajos directamente relacionados con el nuestro son los desarrollados por el grupo DresdenOCL [11, 34, 12, 20], de la Universidad de Dresden, Alemania, junto con sus herramientas asociadas. Como se ha mencionado anteriormente, la solución que ofrecen estos trabajos a la compilación de OCL a SQL es insatisfactoria, por dos razones principales: en primer lugar, sólo cubre un conjunto muy limitado de "patrones" de expresiones OCL y, en segundo lugar, las consultas SQL compiladas son innecesariamente ineficientes. Para superar las limitaciones de su propuesta, en vez de utilizar patrones, decidimos basar nuestro compilador de OCL en una función de traducción definida recursivamente sobre la expresiones OCL. Curiosamente, la idea clave de nuestra traducción, a saber, el uso de procedimientos ("stored-procedures") para la compilación de operaciones iteradoras sobre colecciones, ya había sido explorado —aunque sin éxito— en [34]. Por lo demás, durante toda nuestra investigación hemos mantenido una relación fluida con los investigadores del grupo DresdenOCL, comunicándoles los problemas semánticos y de eficiencia que encontramos en su herramienta (véase [16] para mayores detalles) y colaborando con ellos en la elaboración de un test suite que nos permitiría comparar las soluciones y herramientas de uno y otro grupo.

Aunque de forma menos directa, también guarda relación con nuestra investigado el trabajo desarrollado dentro SAP para la traducción de OCL a MQL

(Moin Query Language) [4], que es el lenguaje de consulta de SAP para sus modelos MOIN. En este trabajo, el subconjunto de OCL que cubre la traducción propuesta es muy limitado (aunque suficiente para sus objetivos): a modo de ejemplo, ni las operaciones iteradoras sobre colecciones, ni tan siquiera las operaciones booleanas sobre colecciones, son traducibles.

Finalmente, cabe reseñar los distintos "compiladores" de OCL en Java/AspectJ propuestos por el grupo DresdenOCL [14, 18, 37], que están basados en diferentes usos de la programación orientada a aspectos. Como en el caso del interprete EOS [7] para OCL, el problema de estas herramientas es la carga en memoria de escenarios de tamaño realmente grande (con millones de objetos), que consumiría un tiempo inaceptable en la práctica (véase [7] para más detalles). De hecho, como se ha explicado anteriormente, es precisamente la solución de este problema la motivación original del presente trabajo de investigación.

1.4 Organización

El resto de este trabajo está organizado en los siguientes capítulos:

- Capítulo 2: *Preliminares.* En este capítulo introducimos el ámbito general (metodologías, lenguajes) en el que se desarrolla nuestro trabajo.
- Capítulo 3: *Descripción general.* En este capítulo describimos los elementos principales (reglas, esquemas) que utilizamos para la compilación de expresiones OCL.
- Capítulo 4: *De diagramas UML a tablas MySQL.* En este capítulo introducimos las reglas que definen, a partir de los diagramas de clases y los diagramas de objetos que forman el contexto de una expresión OCL, las tablas MySQL sobre las que se ejecutarán estas expresiones una vez compiladas.
- Capítulo 5: *De expresiones OCL a procedimientos MySQL.* En este capítulo introducimos las reglas que definen las consultas o los procedimientos, según corresponda, que capturan en MySQL la semántica de una selección representativa de las operaciones OCL. Estas consultas o procedimientos son un componente clave para la compilación de expresiones OCL.
- Capítulo 6: *Implementación.* En este capítulo describimos brevemente la arquitectura de nuestro compilador y reportamos sobre la eficiencia de las expresiones OCL compiladas.
- Capítulo 7: *Conclusiones y trabajo futuro.* En este capítulo recogemos las conclusiones principales de nuestro trabajo y señalamos algunas líneas de desarrollo futuro.
- Apéndice: *Definiciones adicionales.* En este apéndice introducimos las reglas que definen las consultas o los procedimientos, según corresponda, que capturan en MySQL la semántica las operaciones OCL no tratadas en el capítulo 5.

2 Preliminares

En este capítulo introducimos el ámbito general en el que se desarrolla nuestro trabajo, a saber, el desarrollo de software basado en modelos (MDA), así como los lenguajes principales sobre los que versa nuestra investigación, a saber, el lenguaje unificado de modelo (UML), el lenguaje de restricciones sobre objetos (OCL), y los lenguajes de consultas sobre bases de datos relacionales.

2.1 MDA: La arquitectura de software dirigida por modelos

Model Driven Architecture (MDA) [25] es una metodología para el desarrollo de software, definida por el Object Management Group (OMG) [29].[1] La clave de MDA es la importancia que se le confiere a los *modelo*s en el proceso de desarrollo de software. En la especificación de MDA [25, cap. 2] se define un modelo de un sistema como:

- una descripción o especificación tanto del sistema como de su entorno
- en un lenguaje (gráfico y/o textual) bien definido
- para un propósito determinado.

La figura 2.1 describe la relación entre un modelo, el sistema que describe, y el lenguaje en el que está escrito este modelo.

Para MDA, el proceso de desarrollo de software consiste, en último término, en la *transformación* sucesiva de modelos hasta alcanzar el producto final. Tradicionalmente, en este proceso se distingue entre modelos PIM y modelos PSM:

- Un PIM es un modelo independiente de plataforma: es decir, es un modelo que describe un sistema sin hacer referencia a una plataforma concreta final para su despliegue o implantación.

[1] El Object Management Group es un consorcio sin ánimo de lucro de la industria de las tecnologías de la información (con participación académica en menor medida) cuyo objetivo es desarrollar, por acuerdo entre sus miembros, estándares de integración que tengan valor internacional.

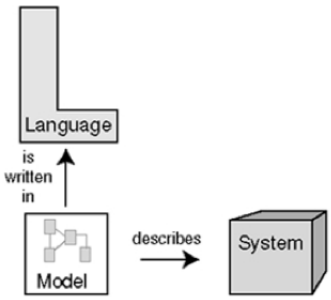

Figura 2.1: MDA: Modelos y lenguajes

- Un PSM es un modelo específico de plataforma: es decir, es un modelo que describe un sistema teniendo en cuenta su plataforma concreta final de despliegue o implantación.

Al igual que en el caso de los modelos, en MDA las transformaciones entre modelos también se escriben en un lenguaje bien definido, típicamente soportado por herramientas de transformación. La figura 2.2 describe el proceso general de transformación de modelos en esta metodología.

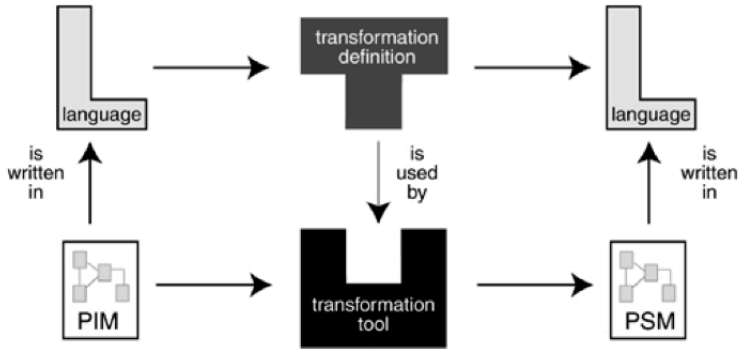

Figura 2.2: Descripción general de la estructura básica de MDA

2.2 UML: El lenguaje unificado de modelado

El lenguaje unificado de modelo (Unified Modeling Language, UML) [27, 19, 33, 3] es un lenguaje visual, de propósito general, para especificar, construir y documentar los modelos de un sistema. UML ofrece distintos tipos de diagramas para modelar los diferentes aspectos o vistas de un sistema. En este trabajo únicamente nos referiremos a dos de estos diagramas, a saber, los diagramas de clase y los diagramas de objetos, que introducimos a continuación.

Diagramas de clases

Los diagramas de clases se utilizan para modelar la vista *estructural* de un sistema. Esta vista es estática, esto es, no describe el comportamiento del sistema. Cuando se modela el *diseño* de un sistema, los diagramas de clase son los primeros que se desarrollan y sobre ellos se completa el resto del diseño.

Un diagrama de clases está compuesto por:

- *Clases.* Se utilizan para modelar los objetos del sistema que tienen las mismas propiedades, relaciones y métodos. Los objetos pertenecientes a una clase son sus *instancias.*
- *Atributos.* Se utilizan para modelar las propiedades estructurales de los objetos de una clase. Cada atributo tiene un nombre y un tipo, que especifica el dominio de los posibles valores del atributo.
- *Asociaciones.* Se utilizan para modelar las relaciones estructurales entre clases. Cada conexión de una asociación se llama extremo de asociación.
- *Multiplicidades.* Se utilizan para indicar cuántas instancias de la clase que está conectada a un extremo de asociación pueden estar relacionados con una instancia de la clase que está conectada al otro extremo de la asociación. En concreto, la multiplicidad $*$ significa 0 o más instancias y es la multiplicidad que, por defecto, se asocia a un extremo de asociación. Las multiplicidades pueden definirse también mediante intervalos.
- *Métodos.* Se utilizan para modelar las operaciones que todas las instancias de una clase implementan.
- *Generalizaciones.* Se utilizan para modelar una relación taxonómica entre dos clases. Una generalización especializa una clase general en otra más específica. Cada instancia de la clase específica es también instancia de la clase general: tiene las caracteríticas (propiedades, relaciones, métodos) de la clase general, además de las de su propia clase.

Ejemplo 1. *La figura 2.3 muestra el diagrama de clases que modela la "estructura" básica de una empresa que ofrece vehículos tanto para servicios de viajes regulares como para viajes privados. En concreto,*

- *La clase* Trip *modela los viajes. Contiene dos atributos* origin *y* destination *que modelan el origen y el destino de los viajes.*
- *La clase* Coach *modela los vehículos. Contiene dos atributos* model *y* numberOfSeats *que especifican el modelo del vehículo y los asientos que tiene.*
- *Las clases* PrivateTrip *y* RegularTrip *son subclases de* Trip *y modelan los dos tipos posibles de viajes: en concreto,*
  - *La clase* PrivateTrip *modela los viajes privados.*
  - *La clase* RegularTrip *modela los viajes regulares. Contiene un atributo* availableSeats *que modela el número de asientos todavía disponibles en el vehículo que va a realizar la ruta.*
- *La clase* Person *modela los pasajeros de un viaje regular. Contiene un atributo* name *que modela el nombre del pasajero.*

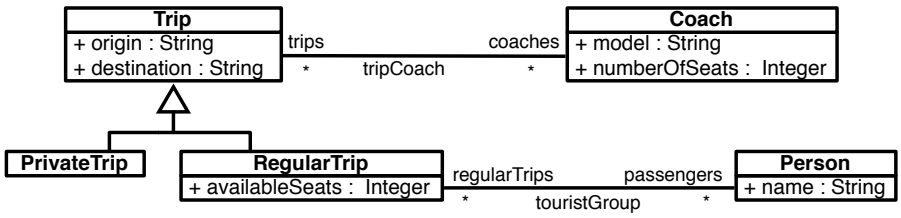

Figura 2.3: Diagrama de clases Coach_Company

Diagramas de objetos

Un diagrama de objetos modela el estado de un sistema en un momento particular. Su notación es similar a la de los diagramas de clases. Los diagramas de objetos se utilizan, principalmente, para el análisis y validación de los modelos que describen los diagramas de clase.

Un diagrama de objetos está compuesto por:

- *Objetos.* Son instancias de las clases. Pueden tener valores asignados sus atributos (tanto propios como "heredados").
- *Enlaces* ("links"). Son instancias de las asociaciones entre clases.

Ejemplo 2. *La figura 2.4 muestra el diagrama de objetos que modela un estado concreto de la empresa de vehículos descrita en la figura 2.3, en el que*

- *la empresa dispone actualmente de una flota de cinco vehículos, aunque cuatro de ellos ya están comprometidos.*
- *la ruta Madrid-Barcelona tiene actualmente un pasajero, y la ruta Sevilla-Madrid tiene dos pasajeros.*

2.3 OCL: Un lenguaje de restricciones y consultas

El modelado, especialmente el modelado de software, ha sido tradicionalmente sinónimo de producir diagramas. La mayoría de los modelos consisten en dibujos de "flechas y burbujas" con algún texto explicativo. De este modo la información contenida en los modelos suele ser incompleta, informal, imprecisa y en ocasiones, incluso inconsistente. Muchos de los defectos en un modelo se deben a las limitaciones de los diagramas que se están usando. Un diagrama simplemente no puede expresar las declaraciones que deberían ser parte de una especificación minuciosa.

La notación UML está fuertemente basada en diagramas. Para dotarlo del nivel de concisión y expresividad que son necesarios en ciertos aspectos de un diseño se extendió el estándar con la especificación del lenguaje de restricciones de objetos (Object Contraint Language, OCL) [26, 36]. OCL es un lenguaje textual con un estilo notacional similar al de los lenguajes orientados a objetos. En UML 1.1, OCL aparece como el estándar para especificar invariantes, precondiciones y postcondiciones. Sin embargo, a partir de UML 2.0 el uso asignado a

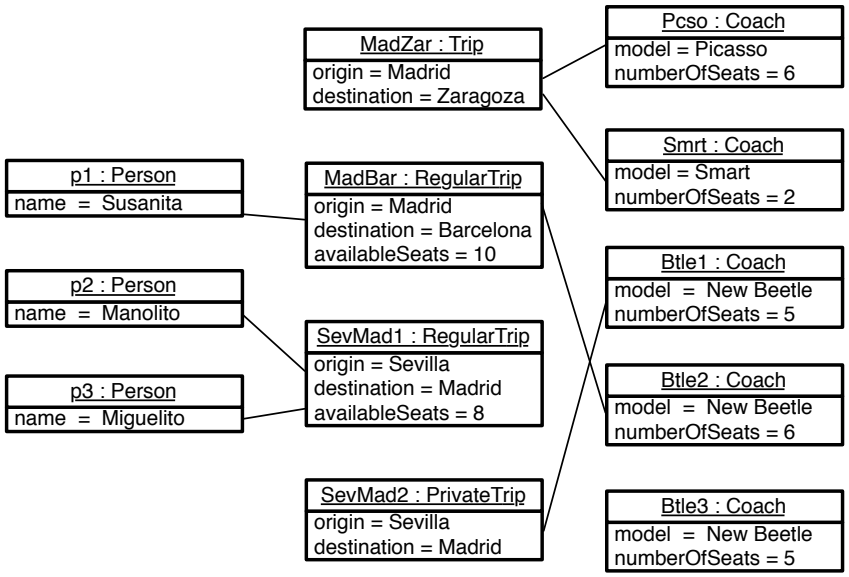

Figura 2.4: Diagrama de objetos Coach_Company_Sample

OCL es mucho más amplio: actualmente, OCL es utilizado, por ejemplo, en la definición de metamodelos de dominio específico, la transformación de modelos, y el testing y la validación de modelos.

OCL es un lenguaje de especificación puro: la evaluación de una expresión OCL es instantánea de modo que cuando una expresión se evalúa, simplemente devuelve un valor sin cambiar nada en el modelo. OCL es un lenguaje fuertemente tipado, con una notación simple, sin símbolos lógicos, y que se basa en un conjunto pequeño de conceptos esenciales. Esto es así porque fue diseñado con el objetivo fundamental de hacerlo ampliamente utilizable: "deberá ser fácilmente escrito y leído por todos los profesionales de la tecnología de objetos y por sus clientes, es decir, gente que no son matemáticos o ingenieros en informática" [22].

OCL es un lenguaje contextualizado: sus expresiones se escriben en el contexto que proporciona un *modelo contextual*. OCL es un lenguaje tipado. Toda expresión OCL tiene un tipo asociado que describe el dominio del resultado de dicha expresión. OCL es también un lenguaje tipado: las expresiones bien formadas tienen un tipo. Los tipos se organizan en una jerarquía de tipos, que determina cuándo dos tipos distintos son *conformes*. Se dice que un tipo *tipo1* es conforme con un tipo *tipo2* cuando una instancia del *tipo1* puede utilizarse en cada lugar donde se espera una instancia del *tipo2*. La relación de *ser conforme* es transitiva. Los **tipos** de OCL se pueden organizar en las siguientes categorías:

- *Tipos primitivos*. Son los tipos básicos Boolean, Integer, Real y String.
- *Tipos clase*. Son las clases del modelo contextual. Por ejemplo, Trip es un tipo OCL cuando el modelo contextual es el diagrama de clases Coach_Company en la figura 2.3.

- *Tipos colección.* Son los tipos parameterizados Set, Bag, OrderedSet y Sequence. Sus parametros pueden ser cualquier otro tipo, incluidos los tipos colección. Por ejemplo, el tipo Set(Integer) de los conjuntos de enteros, o el tipo Sequence(Trip) de las secuencias de viajes, en el contexto del diagrama de clases Coach_Company en la figura 2.3.
- *Tipos tupla.* Son los tipos parameterizados Tuple de pares de elementos. Para cada tipo Tuple los tipos de sus elementos pueden ser cualesquiera. Por ejemplo, Tuple{name:String, age:Integer} para pares de cadenas y enteros, o Tuple{name:String, parents:Set(Person)} para pares de cadenas y conjuntos de pasajeros, en el contexto del diagrama de clases Coach_Company en la figura 2.3.
- *Tipos especiales.* Son los tipos Invalid, Void y Any. Invalid conforma a todos los tipos excepto a Void: la única instancia del tipo Invalid es el valor oclInvalid. Void representa un tipo que conforma a todos los tipos: la única instancia de Void es undefined (o null). Any es el tipo al que todos los demás tipos conforman.

OCL proporciona **operaciones** predefinidas sobre sus distintos tipos. En particular, OCL incluye numerosas operaciones para manipular colecciones, para comprobar propiedades y para generar nuevas colecciones a partir de colecciones existentes. Además, es útil distinguir entre operaciones iteradoras y operaciones no iteradoras: las primeras consisten en coleccionar primero el valor de una expresión dada sobre cada uno de los elementos de la colección original (*fuente*) y entonces comprobar una cierta propiedad (*cuerpo*) sobre las colección resultante.

En las tablas 2.1, 2.2, 2.3 y 2.4 describimos de forma breve e informal un subconjunto significativo de las operaciones predefinidas en OCL sobre los tipos primitivos, sobre los tipos clase proporcionados por el modelo contextual, y sobre los tipos colección. La descripción completa del lenguaje OCL se encuentra en [26].

Ejemplo 3. *Para ilustrar el uso de OCL, formalizamos a continuación algunas expresiones sobre el diagrama de objetos* Coach_Company_Sample *en la figura 2.4.*

- *"El viaje MadBar de Madrid a Barcelona " se formaliza en OCL como:*

  MadBar

- *"Todos los viajes" se formaliza en OCL como:*

  Trip.allInstances()

- *"El número total de viajes" se formaliza en OCL como:*

  Trip.allInstances()−>size()

- *"El número total de asientos disponibles en los viajes SevMad1 y SevMad2 de Sevilla a Madrid" se formaliza en OCL como:*

  SevMad1.availableSeats + SevMad2.availableSeats

Integer	
-(exp_1:Integer)	Devuelve el opuesto de exp_1
(exp_1:Integer).+(exp_2:Integer)	Devuelve la suma de exp_1 y exp_2
(exp_1:Integer).toString()	Devuelve el entero exp_1 como una cadena de caracteres
Real	
(exp_1:Real).-(exp_2:Real)	Devuelve la resta de exp_1 y exp_2
(exp_1:Real).*(exp_2:Real)	Devuelve el producto de exp_1 y exp_2
(exp_1:Real)./(exp_2:Real)	Devuelve la división de exp_1 y exp_2
String	
(exp_1:String).concat(exp_2:String)	Devuelve la concatenación de exp_1 y exp_2
(exp_1:String).at(exp_2:Integer)	Devuelve el carácter en la posición exp_2 en la cadena exp_2.
(exp_1:String).characters()	Devuelve los caracteres de la cadena exp_1 como una secuencia de caracteres.
Boolean	
(exp_1:Boolean).and(exp_2:Boolean)	Devuelve la conjunción de exp_1 y exp_2
(exp_1:Boolean).or(exp_2:Boolean)	Devuelve la disyunción exp_1 y exp_2
(exp_1:Boolean).not()	Devuelve la negación de exp_1

Cuadro 2.1: Operaciones sobre tipos primitivos.

Sobre el modelo	
(exp:$Col(Basic)$).allInstances()	Devuelve todas las instancias de exp.
(exp:[$Class$ | $Col(Class)$]).atr	Devuelve el valor del atributo atr para cada uno de los objetos incluidos en exp.
(exp:[$Class_A$ | $Col(Class_A)$]).rl_A	Devuelve todos los objetos enlazados por el extremo de asociación rl_A para cada uno de los objetos incluidos en exp.
(exp_1:$Class$)->oclIsKindOf(exp_2:Classifier)	Devuelve verdadero si exp_1 es una instancia (directa o indirecta) de la clase exp_2.

Cuadro 2.2: Operaciones sobre el modelo.

Collection(*Basic*)

$(exp_1$:$Col(Basic))$−>size()	Devuelve el número de elementos de la colección *exp*.
$(exp_1$:$Col(Basic))$−>count(exp_2:$Basic$)	Devuelve la cantidad de elementos exp_2 que ocurren en la colección exp_1.
$(exp_1$:$Col(Basic))$−>isEmpty()	Devuelve verdadero si la colección *exp* es vacía.
$(exp_1$:Col($Basic$))−>notEmpty()	Devuelve verdadero si la colección *exp* no es vacía.

Set(*Basic*)

$(exp_1$:Set($Basic$)).=(exp_2:Set($Basic$))	Devuelve verdadero si los conjuntos exp_1 y exp_2 son iguales.
$(exp_1$:Set($Basic$))−>union(exp_2:Set($Basic$))	Devuelve la union de los conjuntos exp_1 y exp_2.

OrderedSet(*Basic*)

$(exp_1$:OrderedSet($Basic$))−>first()	Devuelve el primer elemento del conjunto ordenado *exp*.
$(exp$:OrderedSet($Basic$))−>last()	Devuelve el último elemento del conjunto ordenado *exp*.

Bag(*Basic*)

$(exp_1$:Bag($Basic$))−>intersection(exp_2:Bag($Basic$))	Devuelve la intersección entre los multiconjuntos exp_1 y exp_2.
$(exp_1$:Bag($Basic$))−>including(exp_2:$Basic$)	Devuelve el resultado de añadir exp_2 al multiconjunto exp_1.

Sequence(*Basic*)

$(exp_1$:Sequence($Basic$))−>append(exp_2:$Basic$)	Devuelve el resultado de añadir exp_2 al final de la secuencia exp_1.
$(exp_1$:Sequence($Basic$))−>reverse().	Devuelve la secuencia *exp* en orden inverso.

Cuadro 2.3: Operaciones no iteradoras sobre colecciones

Collection(*Basic*)	
(*fuente*:Collection(*Basic*))−>collect(*var*|*cuerpo*)	Devuelve el conjunto aplanado de elementos que resulta de evaluar *cuerpo* sobre cada elemento de la colección *fuente*.
(*fuente*:Collection(*Basic*))−>forAll(*var*|*cuerpo*)	Devuelve verdadero si la evaluación de *cuerpo* devuelve verdadero para cada elemento de la colección *fuente*.
(*fuente*:Collection(*Basic*))−>select(*var*|*cuerpo*)	Devuelve la colección formada por todos los elementos de *fuente* para los que *cuerpo* evalúa a verdadero.

Cuadro 2.4: Operaciones iteradoras sobre colecciones.

- *"Los modelos de todos los vehículos" se formaliza en OCL como:*

  Coach.allInstances().model

- *"Todos los vehículos que están comprometidos para realizar viajes" se formaliza en OCL como:*

  Trip.allInstances().coaches

- *"Todos los viajes que tienen como origen la ciudad de Madrid" se formaliza en OCL como:*

  Trip.allInstances()−>select(t\|t.origin = 'Madrid')

- *"Si todos los viajes tienen como origen la ciudad de Madrid" se formaliza en OCL como:*

  Trip.allInstances()−>forAll(t\|t.origin='Madrid')

- *"Los destinos de todos los viajes" se formaliza en OCL como:*

  Trip.allInstances()−>collect(t\|t.destination)

- *"Los nombres de los pasajeros de todos los viajes que tienen como destino la ciudad de Madrid" se formaliza en OCL como:*

  Trip.allInstances()−>select(t\|t.destination = 'Madrid').passengers.name

2.4 RDB: Las bases de datos relacionales

De manera informal, las base de datos [32] son almacenes de información. Una *base de datos relacional* [9, 2] es una base de datos en la cual la información se almacena en tablas, relacionadas entre sí de acuerdo con una cierta estructura. Una *tabla* es un multiconjunto de filas. Las filas son la menor unidad de información que puede ser insertada o borrada de una tabla. Cada *fila* tiene al menos una *columna* y el orden entre las columnas es el mismo para todas las filas de una tabla. Cada fila contiene un *valor* para cada una de sus columnas.

SQL y MySQL

El lenguaje de consulta estructurado (SQL) [10, 23] es un lenguaje declarativo para el manejo de bases de datos relacionales que permite especificar diversos tipos de operaciones sobre éstas. MySQL [24] es un gestor de bases de datos relacionales que implementa el lenguaje SQL y lo extiende con un lenguaje *procedural* para la realización de consultas o cambios sobre bases de datos utilizando estructuras como condiciones o bucles. Otros gestores de bases de datos relacionales, como PostgreSQL [31], ORACLE [30], Informix [21], SQLite [35] también soportan lenguajes procedurales similares al de MySQL, con diferencias sintácticas menores. En este trabajo utilizaremos el lenguaje MySQL como lenguaje de consultas sobre bases de datos relacionales.

Tablas. MySQL soporta, fundamentalmente, dos tipos de tablas: a saber, tablas *base* y tablas *virtuales* o *derivadas*. Una tabla base es una tabla cuyos datos están almacenados, ya sea en disco o en memoria. En contraste, una tabla virtual contiene datos derivados, que tienen su origen último en una o más tablas base sobre las que se ha realizado una consulta y, por ello, únicamente existen mientras se resuelve esta consulta.

Sentencias y consultas. En MySQL, para manejar las tablas se ejecutan sentencias. Una *sentencia* es una secuencia de expresiones que especifican una o varias operaciones sobre tablas. Cada sentencia termina con un delimitador de sentencias (típicamente un punto y coma ';').

Ejemplo 4. *La siguiente sentencia crea en la base de datos una tabla* `Trip` *con una columna* `pk` *de tipo* `Int`*.*

```
create table Trip(pk Int);
```

Las *consultas* son expresiones que únicamente contienen operaciones de consulta sobre tablas (base o virtuales). Los resultados de las consultas ("result-sets") son tablas virtuales, a las que se les puede asociar un *alias*. Una subconsulta es simplemente una consulta que ocurre dentro de otra consulta.

Procedimientos. En MySQL es posible definir *procedimientos almacenados* ("stored-procedures") para manejar las tablas. Para ejecutar un procedimiento almacenado es preciso *invocarlo* y puede aceptar múltiples entradas y/o parámetros de salida. Una característica clave de los procedimientos almacenados

es que permiten realizar consultas utilizando estructuras condicionales o bucles. Además, dentro de un procemiento almacenado se pueden utilizar *cursores* para acceder a cada uno de los elementos que resulten de la ejecución de una operación de consulta `select`.

Ejemplo 5. *A modo de ejemplo, explicamos a continuación la definición de un procedimiento almacenado* `example()`. *Este procedimiento simplemente inserta cada uno de los elementos de una tabla* `Trip` *en otra tabla* `ExampleTbl`.

```
1  create procedure example()
2  begin
3  declare var Int;
4  declare done int default 0;
5  declare crs cursor for select pk as val from Trip;
6  declare continue handler for sqlstate '02000' set done = 1;
7  create table exampleTbl(val Int);
8  open crs;
9  repeat
10  fetch crs into var;
11  if not done then
12  insert into exampleTbl(val)
13  (select var as val);
14  end if;
15 until done end repeat;
16 close crs;
17 end;
```

- **Línea 1:** *Define el nombre (*`example`*) del procedimiento.*
- **Línea 2:** *Señala el comienzo del "cuerpo" del procedimiento.*
- **Líneas 3-4:** *Declaran las variables que el procedimiento va a utilizar.*
- **Línea 5:** *Declara un cursor y la colección a cuyos elementos apuntará este cursor: en este caso, los valores almacenados en la columna* `pk` *de la tabla* `Trip`.
- **Línea 6:** *Define que la variable* `done` *recibirá el valor* `1` *cuando el cursor ya no apunte a ningún dato.*
- **Línea 7:** *Crea la tabla* `ExampleTbl`.
- **Línea 8:** *Activa el cursor: le asocia un espacio de memoria.*
- **Líneas 9-15** *Define un bucle que se ejecuta hasta que una condición se cumple: en este caso, que la variable* `done` *tenga asignado el valor* `1` *(o verdadero).*
- **Línea 10:** *Asigna a la variable* `var` *la fila siguiente a la fila actualmente apuntada por el cursor en su tabla asociada (o la primera, cuando en este punto el cursor ha sido únicamente activado).*
- **Líneas 11-14** *Define una estructura condicional: la parte* `then` *sólo se ejecuta si la condición se cumple; en este caso, si la variable* `done` *no tiene asignado el valor* `0`.

- **Líneas 12-13** *Inserta en la tabla* `exampleTbl` *el resultado de una consulta: en este caso, la fila de la tabla* `Trip` *que corresponde a la fila asignada a la variable* `var`*.*
- **Línea 16:** *Desactiva el cursor: libera el espacio de memoria asociado.*

3 Descripción general

En este capítulo describimos los elementos (reglas, esquemas) principales que utilizamos en la definición del compilador MySQL4OCL.

Básicamente, MySQL4OCL consta de dos componentes principales:

- Un conjunto de reglas, que denominamos **toRDB**, que definen el modo de almacenar en tablas los diagramas de objetos sobre los cuales se evaluarán las expresiones OCL compiladas. En el capítulo 4 se introducirán en detalle las reglas **toRDB**.
- Un conjunto de reglas, que denominamos **toProc**, que definen el modo de generar los procedimientos ("stored-procedures") que se utilizan para compilar las expresiones OCL. Una característica común a todos los procedimientos generados por las reglas **toProc** es que sus resultados quedan almacenados en tablas. En el capítulo 5 se introducirán en detalle las reglas **toProc**.

De manera informal, podemos decir que nuestro compilador MySQL4OCL es correcto en el sentido que el resultado de evaluar una expresión OCL *exp* sobre un diagrama de objetos *dob* es equivalente al resultado de ejecutar el procedimiento generado por las reglas **toProc** para la expresión *exp* sobre la base de datos generadas por las reglas **toRDB** para el diagrama de objetos *dob*. Más específicamente, sean en adelante:

- rdb(*dob*) la base de datos generada por las reglas **toRDB** para un diagrama de objetos *dob*.
- proc(*exp*) el procedimiento generado por las reglas **toProc** para una expresión *exp*.
- table(proc(*exp*)) la tabla en la que el procedimiento proc(*exp*) almacena el resultado de su ejecución.

Utilizando esta notación, podemos reformular la propiedad de corrección como sigue: nuestro compilador MySLQ4OCL es correcto en el sentido que el resultado de evaluar una expresión OCL *exp* sobre un diagrama de objetos *dob* es equivalente al resultado de ejecutar, en la base de datos rdb(*dob*), la consulta

`select * from` table(proc(*exp*))`;`

después de invocar el procedimiento proc(*exp*), es decir, después de ejecutar la sentencia:

`call` proc(*exp*)`;`

Las tablas table(proc(*exp*))**.** Como veremos en detalle en el capítulo 5, la estructura concreta de las tablas table(proc(*exp*)) —-donde los procedimientos proc(*exp*)) generados por las reglas **toProc** almacenan sus resultados— depende, lógicamente, del tipo de la expresión *exp*. En este punto es importante recordar que MySQL4OCL no cubre, en su presentación actual, las expresiones de tipo colección de colecciones o de tipo tupla.

En el **caso general**, table(proc(*exp*)) contiene una sólo columna val cuyo tipo es el tipo MySQL que corresponde al tipo OCL de la expresión *exp*, de acuerdo con las siguientes reglas:

- Si el tipo de *exp* es Boolean, Set(Boolean) o Bag(Boolean), entonces el tipo de la columna `val` es `Tinyint(1)`. La idea es que 1 representa true y 0 representa false.
- Si el tipo de *exp* es Integer, Set(Integer) o Bag(Integer), entonces el tipo de la columna `val` es `Int`.
- Si el tipo de *exp* es Real, Set(Real) o Bag(Real), entonces el tipo de la columna `val` es `Real`.
- Si el tipo de *exp* es String, Set(String) o Bag(String), entonces el tipo de la columna `val` es `Varchar(250)`.
- Si el tipo de *exp* es Classifier, Set(Classifier) o Bag(Classifier), entonces el tipo de la columna `val` es `Varchar(250)`. La idea es que las clases "per se" se representan utilizando su nombre.
- Si el tipo de *exp* es un tipo clase *Class*, Set(*Class*) o Bag(*Class*), entonces el tipo de la columna `val` es `Int`. La idea es que los objetos se representan, de manera unívoca, utilizando enteros.

El **caso especial** lo constituyen las expresiones *exp* cuyos tipos son `Sequence` u `OrderedSet` sobre tipos predefinidos (`Boolean`, `Integer`, `Real`), sobre el tipo `Classifier`, o `String`) o sobre tipos clase. En estos casos, la table table(proc(*exp*)) contiene, además de la columna `val` (con el tipo que le corresponda según las reglas anteriores), una columna `pos` de tipo `Int`, en la que se guarda la posición del elemento en la sequencia o en el conjunto ordenado que resulte de la ejecución del procedimiento proc(*exp*).

En adelante, sea:

- colums(table(proc(*exp*))) la estructura de la tabla table(proc(*exp*)) en la que el procedimiento proc(*exp*) almacena el resultado de su ejecución.

Los procedimientos proc(*exp*)**.** Como veremos en detalle en el capítulo 5, la definición de los procedimientos que generan las reglas **toProc** es *recursiva*. Las reglas **toProc** distinguen dos casos principales, el caso general y el caso especial, según sea o no posible capturar la semántica de la operación-raiz de la expresión que se quiere mediante simples *consultas* MySQL (**caso general**) o si, por el contrario, es necesario recurrir a estructuras condionales o bucles que, como es

sabido, sólo son admisibles en MySQL dentro de *procedimientos* (**caso especial**). Además, las operaciones iteradores requieren de un tratamiento específico, dentro del caso especial, como se explicará al final de este capítulo.

En el **caso general**, dada una expresión $op(exp_1, \ldots, exp_n)$, el "cuerpo" del procedimiento proc($op(exp_1, \ldots, exp_n)$) consistirá en:

- La ejecución de los procedimientos proc(exp_1), ..., proc(exp_n) .
- La ejecución de la *consulta* que captura la semántica específica de la operación-raiz *op*, utilizando como "argumentos" los resultados almacenados en las tablas table(proc(exp_1)), ..., table(proc(exp_n)).
- El almacenamiento del resultado de la consulta anterior en la tabla table(proc($op(exp_1, \ldots, exp_n)$)).

En adelante, sea:

- query($op(exp_1, \ldots, exp_n)$) la consulta que captura, en el caso general de las reglas **toProc**, la semántica de la operación-raiz *op* dentro del "cuerpo" del procedimiento proc($op(exp_1, \ldots, exp_n)$).

Utilizando la notación que hemos introducido hasta ahora, en la figura 3.1 se muestra el "esqueleto" de los procedimientos generados, para el caso general, por las reglas **toProc**.

```
create procedure proc(op(exp_1, ..., exp_n))
begin
  call proc(exp_1);
  ⋮
  call proc(exp_n);
  create table table(proc(op(exp_1, ..., exp_n)))
    (colums(table(proc(op(exp_1, ..., exp_n)))));
  insert into table(proc(op(exp_1, ..., exp_n)))
                (colums(table(proc(op(exp_1, ..., exp_n)))))
          (query(op(exp_1, ..., exp_n)));
end;
```

Figura 3.1: Esqueleto del procedimiento generado por las reglas **toProc**: caso general.

El **caso especial** viene dado por las operaciones OCL que aparecen listadas en la tabla 3.1. Para capturar la semántica de estas operaciones es preciso recurrir a estructuras condicionales o bucles, que sólo son admisibles dentro de procedimientos. En estos casos, dada una expresión $op(exp_1, \ldots, exp_n)$, el "cuerpo" del procedimiento proc($op(exp_1, \ldots, exp_n)$) consistirá en:

- La ejecución de los los procedimientos proc(exp_1), ..., proc(exp_n).
- La ejecución del *procedimento* que capture la semántica específica de la operación-raiz *op*, utilizando como "argumentos" los resultados almacenados en las tablas table(proc(exp_1)), ..., table(proc(exp_n)).

Operaciones	Comentarios
allInstances	
oclIsTypeOf	
oclIsKindOf	
oclType	
oclAsType	
=	entre conjuntos y multiconjuntos
characters	
<>	entre conjuntos y multiconjuntos
asOrderedSet	desde conjunto, multiconjunto o secuencia
asSequence	desde conjunto o multiconjunto
intersection	entre multiconjuntos
excluding	desde una secuencia
iterate	
collect	
forAll	
exists	
select	
reject	
any	
one	
sortedBy	

Cuadro 3.1: Operaciones cuya semántica se captura mediante procedimientos.

- El almacenamiento del resultado de la ejecución del procedimiento anterior en la tabla table(proc($op(exp_1, \ldots, exp_n)$)).

En adelante, sea:

- $\text{proc}^\sharp(op(exp_1, \ldots, exp_n))$ el procedimiento que captura, en el caso especial de las reglas **toProc**, la semántica de la operación-raiz *op* dentro del "cuerpo" del procedimiento proc($op(exp_1, \ldots, exp_n)$).

Utilizando la notación introducida hasta ahora, en la figura 3.2 se muestra el "esqueleto" de los procedimientos generados, para el caso especial, por las reglas **toProc**.

Finalmente, las operaciones **iteradoras** requieren un tratamento específico, dentro del esquema que acabamos de presentar para el caso especial. Por una parte, para capturar la semántica de una operación iteradora *iter* es preciso utilizar estructuras de bucles, que sólo son admisibles dentro de procedimientos: en este sentido, las operaciones iteradoras forman parte de las operaciones que caen dentro del caso especial. Por otra parte, el "cuerpo" en una expresión iteradora, aún siendo un argumento de la operación iteradora *iter*, es precisamente lo que se ejecuta en cada iteración: por tanto, dada una expresión $iter(fuente, cuerpo)$, la invocación del procedimiento $proc(cuerpo)$ no puede ejecutarse *antes* de la llamada al procedimiento $\text{proc}^\sharp(iter(fuente, cuerpo))$ sino,

```
create procedure proc(op(exp_1, ..., exp_n))
begin
  call proc(exp_1);
  ⋮
  call proc(exp_n);
  call proc♯(op(exp_1, ..., exp_n));
  create table table(proc(op(exp_1, ..., exp_n)))
    (colums(table(proc(op(exp_1, ..., exp_n)))));
  insert into table(proc(op(exp_1, ..., exp_n)))
               (colums(table(proc(op(exp_1, ..., exp_n)))))
         (select * from table(proc♯(op(exp_1, ..., exp_n))));
end;
```

Figura 3.2: Esqueleto del procedimiento generado por las reglas **toProc**: caso especial.

precisamente, *dentro* de este procedimiento. En concreto, el "cuerpo" del procedimiento proc(*iter*(*fuente*, *cuerpo*)) consistirá en:

- La ejecución del procedimiento proc(*fuente*).
- La ejecución del *procedimento* proc♯(*iter*(*fuente*, *cuerpo*)) que captura la semántica específica de la operación-raiz *iter*, utilizando como "argumento" el resultado almacenado en la tabla table(proc(*fuente*)). En el "cuerpo" de este procedimiento se ejecutará, dentro de un bucle, el procedimiento proc(*cuerpo*), de acuerdo con la semántica específica de la operación-raiz *iter*.
- El almacenamiento del resultado de la ejecución del procedimiento anterior en la tabla table(proc(*iter*(*fuente*, *cuerpo*))).

En la figura 3.3 se muestra el "esqueleto" de los procedimientos generados, para el caso especial de las operaciones iteradoras, por las reglas **toProc**.

```
create procedure proc(iter(fuente, cuerpo))
begin
  call proc(fuente);
  call proc♯(iter(fuente, cuerpo));
  create table table(proc(iter(fuente, cuerpo)))
    (colums(table(proc(iter(fuente, cuerpo)))));
  insert into table(proc(iter(fuente, cuerpo)))
               (colums(table(proc(iter(fuente, cuerpo)))))
         (select * from table(proc♯(iter(fuente, cuerpo))));
end
```

Figura 3.3: Esqueleto del procedimiento generado por la traducción: caso especial para los iteradores.

4 De diagramas UML a tablas MySQL

En este capítulo introducimos el conjunto de reglas **toRDB** que definen, a partir del diagrama de clases *dcl* y el diagrama de objetos *dob* que forman el contexto de una expresión OCL *exp*, las tablas MySQL sobre las que se ejecutarán los procedimientos generados por las reglas **toProc** al compilar esta expresión utilizando MySQL4OCL. En adelante, asumimos que:

- Existe una función nm() que genera un *nombre* unívoco (más en concreto, una cadena de caracteres, sin espacios) para cada una de las clases y de las asociaciones, y para cada uno de los atributos de cada clase, del diagrama de clases *dcl*.
- Existe una function pk() que genera un *identificador* (más en concreto, un número entero) para cada uno de los objetos de una misma clase que aparecen *dob*.

Además, sea:

- type(*tp*) el tipo MySQL asociado al tipo UML *tp*, de acuerdo con las reglas introducidas en el capítulo anterior. En concreto:

$$\begin{aligned} \text{type}(\mathsf{Boolean}) &= \texttt{Tinyint(1)} \\ \text{type}(\mathsf{Integer}) &= \texttt{Int} \\ \text{type}(\mathsf{Real}) &= \texttt{Real} \\ \text{type}(\mathsf{String}) &= \texttt{Varchar(250)} \\ \text{type}(\mathit{Class}) &= \texttt{Int} \end{aligned}$$

- value(*vl*) el valor MySQL asociado al valor UML *vl*, de acuerdo con las siguientes reglas:

$$\begin{aligned} \text{value}(\text{true}) &= \texttt{1} \\ \text{value}(\text{false}) &= \texttt{0} \\ \text{value}(i) &= i, \text{ para cualquier entero } i \\ \text{value}(r) &= r, \text{ para cualquier real } r \\ \text{value}(s) &= s, \text{ para cualquier cadena } s \\ \text{value}(ob) &= \text{pk}(ob), \text{ para cualquier objeto } ob. \end{aligned}$$

Las reglas toRDB. Sea *dcl* un diagrama de clase y *dob* un diagrama de objetos, que sea una instancia de *dcl*. Entonces, la base de datos toRDB(*dcl*, *dob*) se construye como sigue:

- Para cada clase *cl* en el diagrama *dcl* se crea una tabla nm(*cl*), que tiene una columna `pk` de tipo `Int` como su clave primaria. En notación de MySQL:

  `create table` nm(*cl*)`(pk int primary key);`

- Para cada clase cl_i que sea *subclase directa*[1] de una clase *cl* en el diagrama *dcl* se agrega una restricción a la tabla nm(cl_i): a saber, que la clave primaria `pk` de esta tabla tiene que ser, además, clave foránea de la clave primaria de la tabla nm(*cl*). En notación de MySQL:

  `alter table` nm(cl_i) `add constraint fk-`nm(cl_i)`-`nm(*cl*)
  `foreign key (pk) references` nm(*cl*)`(pk);`

- Para cada atributo *atr* de tipo *tp* de una clase *cl* en el diagrama *dcl* se a 'unade una columna nm(*atr*) en la correspondiente tabla nm(*cl*). En notación de MySQL:

  `alter table` nm(*cl*) `add` nm(*atr*) type(*tp*)`;`

  Además si el atributo *atr* es de tipo clase, supongamos cl_i, entonces se agrega una restricción a la columna nm(*atr*): a saber, que esta columna será también clave foránea de la columna `pk` de la tabla nm(cl_i). En notación de MySQL:

  `alter table` nm(*cl*) `add constraint fk-`nm(*atr*)`-`nm(cl_i)
  `foreign key (`nm(*atr*)`) references` nm(cl_i)`(pk);`

- Para cada asociación *Asoc* en el diagrama *dcl* entre dos clases cl_A y cl_B, con extremos de asociación ("association-ends") rl_A (en la clase cl_A) y rl_B (en la clase cl_B), hay que distinguir casos, según sea la multiplicidad de la asociación *Asoc*:

  - Multiplicidad *..*. Se crea una tabla nm(*Asoc*) con dos columnas nm(rl_A) y nm(rl_B), ambas de tipo `Int`. En notación de MySQL:

    `create table` nm(*Asoc*)
    `(`nm(rl_A) `int not null,` nm(rl_B) `int not null,`
    `foreign key (`nm(rl_A)`) references` nm(cl_A)`(pk),`
    `foreign key (`nm(rl_B)`) references` nm(cl_B)`(pk));`

  - Multiplicidad 0..1, 1..1 o 1..*. Se agrega una columna en la tabla de destino de la asociación. En notación MySQL:

    `alter table` nm(cl_B) `add` nm(rl_A) `int;`

[1] Decimos que la clase *B* es una *subclase directa* de la clase *A*, si *B* es una subclase de *A* y además no existe niguna subclase *C*, tal que *C* sea subclase de *A* y superclase de *B*.

Además se agregan restricciones sobre estas columnas: a saber, que nm(rl_A) es también clave foránea de la columna `pk` de la tabla nm(cl_A)
En notación de MySQL:

```
alter table nm(cl_B)
  add constraint fk-nm(Asoc)-nm(rl_A)
  foreign key nm(rl_A) references nm(cl_A)(pk);
```

- Para cada objecto ob de cada clase cl en el diagrama dob se a
'unade una nueva fila en la tabla nm(cl) y se inserta en su columna `pk` el identificador nm(ob) correspondiente al objeto ob. En notación de MySQL:

  ```
  insert into nm(cl)(pk) values(pk(ob));
  ```

  Además, para cada superclase cl_i de la clase cl, se a
  'unade una nueva fila en la tabla nm(cl_i) y se inserta en su columna `pk` el identificador nm(ob) correspondiente al objeto ob. En notación de MySQL:

  ```
  insert into nm(cl_i)(pk) values(pk(ob));
  ```

- Para cada valor vl de cada atributo atr de cada objeto ob de cada clase cl en el diagrama dob, se inserta el correspondiente valor en la columna asociada al atributo atr de la fila asociada al objeto ob en la tabla asociada a la clase cl. En notacion de MySQL:

  ```
  insert into nm(cl)(nm(atr)) values (value(vl));
  ```

- Para cada enlace ("link") entre dos objetos ob_A y ob_B, que sea instancia de una asociación $Asoc$ entre dos clases cl_A y cl_B, con extremos de asociación ("association-ends") rl_A (en la clase cl_A) y rl_B (en la clase cl_B), en el diagrama dob, hay que distinguir casos, según sea la multiplicidad de la asociación $Asoc$:

  - Multiplicidad `*..*`. Se a
  'unade una nueva fila en la tabla nm($Asoc$), y se inserta en la columna nm(rl_A) asociada al extremo de asociación rl_A el identificador pk(ob_A) correspondiente al objeto ob_A, y en la columna nm(rl_B) asociada al extremo de asociación rl_B se inserta el identificador pk(ob_B) correspondiente al objeto ob_B. En notación de MySQL:

    ```
    insert into nm(Asoc)(nm(rl_A),nm(rl_B))
           values (pk(ob_A),pk(ob_B));
    ```

  - Multiplicidad `0..1`, `1..1` o `1..*`: Se añade una nueva fila en la tabla nm(cl_A) y se inserta en la columna nm(rl_B) asociada al extremo de asociación rl_B el identificador pk(ob_B) correspondiente al objeto ob_B. En notación de MySQL:

    ```
    insert into nm(cl_A)(nm(rl_B)) values (pk(ob_B));
    ```

Ejemplo 6. *A continuación mostramos las sentencias MySQL generadas por las reglas* ***toRDB*** *correspondientes al diagrama de clases* Coach_ Company *de la figura 2.3 en el capítulo 2.*

```
create table Coach(pk int primary key);
create table Person(pk int primary key);
create table PrivateTrip(pk int primary key);
create table RegularTrip(pk int primary key);
create table Trip(pk int primary key);

alter table PrivateTrip add constraint fk-PrivateTrip-Trip
  foreign key (pk) references Trip(pk);
alter table RegularTrip add constraint fk-RegularTrip-Trip
  foreign key (pk) references Trip(pk);

alter table Coach add model Varchar(250);
alter table Coach add numberOfSeats Int;
alter table Person add name Varchar(250);
alter table RegularTrip add availableSeats Int;
alter table Trip add origin Varchar(250);
alter table Trip add destination Varchar(250);

create table TripCoach(trips int not null, coaches int not null,
  foreign key (trips) references Trip(pk),
  foreign key (coaches) references Coach(pk));
create table TouristGroup(regularTrips int not null,
  passengers int not null,
  foreign key (regularTrips) references RegularTrip(pk),
  foreign key (passengers) references Person(pk));
```

Ejemplo 7. *En la figura 4.1 representamos gráficamente la base de datos MySQL que se corresponde, de acuerdo con las reglas* ***toRDB****, con el diagrama de objetos* Coach_Company_Sample *de la figura 2.4 en el capítulo 2.*

Trip

pk	origin	destination
1	Madrid	Zaragoza
2	Madrid	Barcelona
3	Sevilla	Madrid
4	Sevilla	Madrid

Coach

pk	model	numberOfSeats
1	Picasso	6
2	Smart	5
3	New Beetle	5
4	New Beetle	6
5	New Beetle	5

PrivateTrip

pk
4

RegularTrip

pk	availableSeats
2	10
3	8

Person

pk	name
1	Susanita
2	Manolito
3	Miguelito

TripCoach

trips	coaches
1	1
1	2
2	4
4	3

TouristGroup

regularTrips	passengers
2	1
3	2
3	3

Figura 4.1: Estructura de RDB correspondiente al diagrama de clases la figura 2.3.

5 De expresiones OCL a procedimientos MySQL

Como se explicó en el capítulo 3, la reglas **toProc** definen el modo de generar los procedimientos proc(*exp*) que se utilizan en MySQL4OCL para compilar expresiones OCL *exp*. La definición de estos procedimientos es *recursiva* y, según sea la operación-raiz de las expresiones *exp*, sigue uno de los tres patrones introducidos en el capítulo 3: concretamente, en las figuras 3.1, 3.2 y 3.3. Dejando al margen el patrón que se utilice, como se explicó en el capítulo 3, las únicas diferencias en los "cuerpos" de los procedimientos proc(*exp*) vienen dadas por las *consultas* query(*exp*) o los *procedimientos* proc$^\sharp$(*exp*), según sea el caso, que se definen en dichos "cuerpos" para capturar la semántica específica de la operación-raiz de las expresiones *exp*. En este capítulo introducimos la definición de las consultas query($op(exp_1, \ldots, exp_n)$) o de los procedimientos proc$^\sharp$($op(exp_1, \ldots, exp_n)$), según corresponda, para una selección representativa de operaciones *op*; las definiciones correspondientes al resto de las operaciones de OCL, se incluyen en el apéndice A. Para todos los ejemplos que incluimos en este capítulo, el diagrama de clases y el diagrama de objetos que sirven de contexto a las expresiones OCL son, respectivamente, el diagrama Coach_Company y el diagrama Coach_Company_Sample de las figuras 2.3 y 2.4 en el capítulo 2.

En adelante,

- *Class* denota cualquier tipo clase.
- *Primitive* denota cualquier tipo primitivo, a saber: Boolean, Integer, Real, y String.
- *Basic* denota cualquier tipo clase o cualquier tipo primitivo.
- *Col*(*Basic*) denota cualquier tipo colección (Set, Bag, OrderedSet, Sequence) sobre un tipo básico *Basic*.

5.1 Operadores específicos del modelo

▷ *Class*:**Classifier.** Devuelve en OCL el nombre de la clase asociada al tipo Classifier. Su semántica en MySQL se captura utilizando una consulta. En concreto, query(*Class*:Classifier) es igual a:

```
select 'nm(Class)' as val
```

▷ *object*:*Class*. Devuelve en OCL el identificador asociado al objeto de tipo Class. Su semántica en MySQL se captura utilizando una consulta. En concreto, query(*object*:*Class*) es igual a:

```
select pk(object) as val
```

▷ (exp_1:**Classifier**)**.allInstances().** Su semántica se captura en MySQL utilizando un procedimiento. Supongamos que $c_1, \ldots, c_n$ son las clases del modelo que sirve de contexto. En concreto, el procedimiento proc$^\sharp$(*exp*), donde $exp = (exp_1$:Classifier).allInstances() se define como:

```
create procedure proc♯(exp)()
begin
declare v varchar(250);
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val int);
select val into v
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
case v
  when 'nm(c1)' then
    insert into table(proc♯(exp))(val)
           (select pk as val from nm(c1));
   ⋮
  when 'nm(cn)' then
    insert into table(proc♯(exp))(val)
           (select pk as val from nm(cn));
  else
    insert into table(proc♯(exp))(val)
           (select val
           from (select null as val) as temp0
           where false);
end case;
end;
```

Esquema 5.1: Procedimiento que describe la semántica de la operación allInstances.

▷ (exp_1:[*Class* | *Col*(*Class*)])**.***atr*. Su semántica se captura en MySQL utilizando una consulta. Supongamos que *atr* es un atributo de la clase *Class* en el modelo que sirve como contexto a la expresión. En concreto, query((exp_1:[*Class* | *Col*(*Class*)]).*atr*) es igual a:

```
select nm(Class).nm(atr) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
left join nm(Class) on alias(table(proc(exp1))).val = nm(Class).pk;
```

▷ (exp_1:[$Class_A$ | *Col*(*Class*$_A$)])$.rl_A$. Su semántica se captura en MySQL utilizando una consulta. Supongamos que rl_A y rl_B son los dos extremos de una asociación *Asoc* entre una clase $Class_A$ y una clase $Class_B$ en el modelo que

sirve como contexto a la expresión. Supongamos también que rl_A pertenece a la clase $Class_A$ y que rl_B pertenece a la clase $Class_B$. Por último, supongamos que esta asociación tiene multiplicidad *..*. En concreto, query((*exp*$_1$:[$Class_A$ | $Col(Class_A)$]).rl_A) es igual a:

```
select nm(Asoc).nm(rl_A) as val
from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
left join nm(Asoc) on alias(table(proc(exp_1))).val = nm(Asoc).nm(rl_B)
where nm(Asoc).nm(rl_A) is not null
  or alias(table(proc(exp_1))).val is null
```

▷ (*exp*$_1$:*Class*)**.oclIsKindOf(***exp*$_2$**:Classifier)** Su semántica se captura utilizando un procedimiento. Supongamos que $c_1, \ldots, c_n$ son las clases del modelo que sirven de contexto. En concreto, el procedimiento proc$^\sharp$(*exp*) donde *exp* = (*exp*$_1$:*Class*).oclIsKindOf(*exp*$_2$:Classifier) se define como:

```
create procedure proc#(exp)()
begin
declare v varchar(250);
drop table if exists table(proc#(exp));
create table table(proc#(exp)) (val int);
select val into v
  from (select val from table(proc(exp_1))) as alias(table(proc(exp_1)));
case v
  when 'nm(c_1)' then
    insert into table(proc#(exp))(val)
      select alias(table(proc(exp_1))).val in
       (select pk as val from nm(c_1))
      from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
   ⋮
  when 'nm(c_n)' then
     insert into table(proc#(exp))(val)
      select alias(table(proc(exp_1))).val in
       (select pk as val from nm(c_n))
      from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
  else
    insert into table(proc#(exp))(val) (select false as val);
end case;
end;
```

Esquema 5.2: Procedimiento que describe la semántica de la operación oclIsKindOf.

5.2 Tipos primitivos

Integer

▷ *i*:**Integer.** Devuelve en OCL el mismo entero *i*. Su semántica se captura en MySQL utilizando una consulta. En concreto, query(*i*:Integer) es igual a:

```
select toValue(i) as val
```

▷ $-(exp_1$**:Integer**$)$**.** Su semántica se captura en MySQL utilizando una consulta. En concreto, query$(-(exp_1$:Integer$))$ es igual a:

```
select -alias(table(proc(exp1))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ $(exp_1$**:Integer**$)$**.+(**exp_2**:Integer)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query$((exp_1$:Integer$).+($$exp_2$:Integer$))$ es igual a:

```
select alias(table(proc(exp1))).val + alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ $(exp_1$**:Integer**$)$**.toString()** Su semántica se captura en MySQL utilizando una consulta. En concreto, query$((exp_1$:Integer$)$.toString()$)$ es igual a:

```
select cast(alias(table(proc(exp1))).val as char) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Real

▷ r**:Real.** Devuelve en OCL el mismo real r. Su semántica se captura en MySQL utilizando una consulta. En concreto, query$(r$:Real$)$ es igual a:

```
select toValue(r) as val
```

▷ $(exp_1$**:Real**$)$**.−(**exp_2**:Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query$((exp_1$:Real$).-($$exp_2$:Real$).)$ es igual a:

```
select alias(table(proc(exp1))).val - alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ $(exp_1$**:Real**$)$**.∗(**exp_2**:Real)**. Esta operación devuelve el producto entre los argumentos exp_1 y exp_2. Su semántica se captura utilizando una consulta. En concreto, query$((exp_1$:Real$).*($$exp_2$:Real$))$ es igual a:

```
select alias(table(proc(exp1))).val * alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:Real**)**./(**exp_2**:Real).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real)./(exp_2:Real)) es igual a:

```
select alias(table(proc(exp1))).val / alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

String

▷ s**:String.** Devuelve en OCL la misma cadena de caracteres s. Su semántica se captura en MySQL utilizando una consulta. En concreto, query(s:String) es igual a:

```
select toValue(s) as val
```

▷ (exp_1**:String**)**.concat(**exp_2**:String).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:String).concat(exp_2:String)) es igual a:

```
select
  concat(alias(table(proc(exp1))).val, alias(table(proc(exp2))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:String**)**.at(**exp_2**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:String).at(exp_2:Integer)) es igual a:

```
select case when alias(table(proc(exp2))).val <= 0
    or query(exp2).val > length(alias(table(proc(exp1))).val)
  then 0 as val
  else substr(alias(table(proc(exp1))).val,
                  alias(table(proc(exp2))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:String**)**.characters().** Su semántica se captura en MySQL utilizando un procedimiento. En concreto, el procedimiento $\text{proc}^\sharp(exp)$, donde $exp = (exp_1\text{:String}).\text{characters}()$ se define como:

```
create procedure proc#(exp)()
begin
  declare v1 Int default 1;
  declare v2 Int default 0;
  drop table if exists table(proc(exp));
  create table table(proc(exp))(pos INT not null auto_increment,
                                val char(65), primary key(pos));
```

```
  select length(alias(table(proc(exp1))).val) into v2
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
  while v1 <= v2 do
    insert into table(proc(exp))(val)
      select substring(alias(table(proc(exp1))).val,v1,1) as val
      from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
    set v1 = v1 + 1;
  end while;
end;
```

Esquema 5.3: Procedimiento que describe la semántica de la operación characters.

Boolean

▷ *b*:**Boolean.** Devuelve en OCL el mismo valor booleano *b*. Su semántica se captura en MySQL utilizando una consulta. En concreto, query(*b*:Boolean) es igual a:

```
select toValue(b) as val
```

▷ (*exp*$_1$:**Boolean**).**not().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Boolean).not()) es igual a:

```
select not(alias(table(proc(exp1))).val) as val
from (table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (*exp*$_1$:**Boolean**).**and(***exp*$_2$:**Boolean).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Boolean).and(*exp*$_2$:Boolean)) es igual a:

```
select alias(table(proc(exp1))).val and alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$:**Boolean**).**or(***exp*$_2$:**Boolean).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Boolean).or(*exp*$_2$:Boolean)) es igual a:

```
select alias(table(proc(exp1))).val or alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

5.3 Tipos colección

Collection

▷ (exp_1:$Col(Basic)$)−>**size().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:$Col(Basic)$)−>size()) es igual a:

```
select count(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1:$Col(Basic)$)−>**count(**exp_2:$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:$Col(Basic)$)−>count(exp_2:$Basic$)) es igual a:

```
select count(alias(table(proc(exp1))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
where alias(table(proc(exp1))).val = alias(table(proc(exp2))).val
```

▷ (exp_1:$Col(Basic)$)−>**isEmpty().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:$Col(Basic)$)−>isEmpty()) es igual a:

```
select count(alias(table(proc(exp1))).val) = 0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1:**Col(**$Basic$**)**)−>**notEmpty().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Col($Basic$))−>notEmpty()) es igual a:

```
select count(alias(table(proc(exp1))).val) <> 0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Set

▷ (exp_1:**Set(**$Basic$**)**).**=(**exp_2:**Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando un procedimiento. Este procedimiento, básicamente,

- Calcula, en primer lugar, el cardinal de cada conjunto.
- Si los cardinales son diferentes, entonces los conjuntos no pueden ser iguales, por lo que termina y devuelve '`false`'.
- Si los cardinales coinciden, declara dos cursores que utiliza para iterar sobre cada uno de los conjuntos; además, ordena ambos conjuntos utilizando un mismo criterio.
- A continuación, itera sobre los dos conjuntos, comparando en cada paso de la iteración, los valores apuntados por ambos cursores.

- Si en algún paso de la iteración encuentra una desigualdad entre los valores apuntados por ambos cursores, termina y devuelve '`false`'; en caso contrario devuelve '`true`' al final de la iteración.

Entonces, el procedimiento proc$^\sharp(exp)$,
donde $exp = (exp_1{:}\mathsf{Set}(Basic)).{=}(exp_2{:}\mathsf{Set}(Basic))$ se define como:

```
create procedure proc♯(exp)()
begin
declare result boolean;
declare size1 int;
declare size2 int;
declare size3 int;
declare done int default 0;
declare var1 type(Basic);
declare var2 type(Basic);
declare crs1 cursor for
  select alias(table(proc(exp1))).val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  order by alias(table(proc(exp1))).val;
declare crs2 cursor for
  select alias(table(proc(exp2))).val
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
  order by alias(table(proc(exp2))).val;
declare continue handler for sqlstate '02000' set done = 1;
select count(val) into size1
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
select count(val) into size2
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)));
if (size1 = size2) then
  open crs1;
  open crs2;
  set result = true;
  repeat
    fetch crs1 into var1;
    fetch crs2 into var2;
    if (var1 <> var2) then
      set result = false;
      set done = 1;
    end if;
  until done=1 end repeat;
  close crs1;
  close crs2;
else
  set result = false;
end if;
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val boolean);
insert into table(proc♯(exp))(val) values (result);
```

```
end;
```

Esquema 5.4: Procedimiento que describe la semántica de la operación de igualdad entre conjuntos.

▷ (exp_1**:Set(**$Basic$**)−>union(**exp_2**:Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Set(*Basic*))−>union(*exp*$_2$:Set(*Basic*))) es igual a:

```
select * from
  (select * from table(proc(exp1))
  union
  select * from table(proc(exp2))) as alias(table(proc(exp1)))
```

OrderedSet

▷ (exp_1**:OrderedSet(**$Basic$**)−>first().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:first(*Basic*))−>first()) es igual a:

```
select alias(table(proc(exp1))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
where alias(table(proc(exp1))).pos = 1
```

▷ (*exp***:OrderedSet(**$Basic$**)−>last().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>last()) es igual a:

```
select alias(table(proc(exp1))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select max(alias(table(proc(exp1))).pos) as val
  from (select * from table(proc(exp1)))
    as alias(table(proc(exp1)))) as alias(table(proc(exp'1)))
where alias(table(proc(exp'1))).val = alias(table(proc(exp1))).pos
```

Bag

▷ (exp_1**:Bag(**$Basic$**)−>intersection(**exp_2**:Bag(**$Basic$**)).** Su semántica se captura en MySQL utilizando un procedimiento. En este procedimiento, básicamente,

1. Declara un cursor sobre el conjunto que resulta de la intersección de ambos multiconjuntos, eliminando las repeticiones.

2. Crea una tabla resultado para almacenar el multiconjunto resultante.

3. Para cada elemento del conjunto origen extraído por el cursor, cuenta el número de repeticiones del elemento que posee cada uno de los multiconjuntos de origen.

4. Inserta en la tabla resultado el elemento guardado por la variable iteradora tantas veces como el número mínimo de repeticiones con que aparezca en los multiconjuntos origen.

5. Repite los dos últimos pasos hasta que el cursor termina de avanzar por la colección fuente.

Entonces, el procedimiento proc$^\sharp(exp)$,
donde $exp = (exp_1{:}\mathsf{Bag}(Basic)).\mathsf{intersection}(exp_2{:}\mathsf{Bag}(Basic))$ se define como:

```
create procedure proc♯(exp)
begin
declare result type(Basic);
declare size1 int;
declare size2 int;
declare done int default 0;
declare var type(Basic);
declare crs cursor for
  select distinct alias(table(proc(exp'1))).val
  from
    (select distinct *
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as alias(table(proc(exp'1)))
  left join
    (select distinct *
    from (select * from table(proc(exp2)))
      as alias(table(proc(exp2)))) as alias(table(proc(exp'2)))
  on alias(table(proc(exp'1))).val = alias(table(proc(exp'2))).val
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val Basic);
open crs;
repeat
fetch crs into var;
if not done then
  select count(val) into size1
    from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
    where alias(table(proc(exp1))).val = var;
  select count(val) into size2
    from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
    where alias(table(proc(exp2))).val = var;
  if (size1 <= size2) then
  insert into table(proc♯(exp))(val)
    select alias(table(proc(exp1))).val as val
    from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
    where alias(table(proc(exp1))).val = var;
  else
  insert into table(proc♯(exp))(val)
    select alias(table(proc(exp2))).val as val
    from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
    where alias(table(proc(exp2))).val = var;
```

```
  end if;
end if;
until done=1 end repeat;
close crs;
end;
```

Esquema 5.5: Procedimiento que describe la semántica de la operación de intersection.

▷ (exp_1**:Bag(***Basic***)**)−>**including(***exp*$_2$**:***Basic***).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Bag(*Basic*))−>including(*exp*$_2$:*Basic*)) es igual a:

```
select *
from (
  (select * from table(proc(exp1)))
  union all
  (select * from table(proc(exp2)))) as alias(table(proc(exp1)))
```

Sequence

▷ (exp_1**:Sequence(***Basic***)**)−>**append(***exp*$_2$**:***Basic***).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Sequence(*Basic*))−>append(*exp*$_2$:*Basic*)) es igual a:

```
select * from
  (select alias(table(proc(exp1))).pos as pos,
    alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  union all
  select alias(table(proc(exp'1))).val + alias(table(proc(exp2))).pos as pos,
    alias(table(proc(exp2))).val as val
  from
    (select max(alias(table(proc(exp1))).pos) as val
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as alias(table(proc(exp'1))),
    (select * from table(proc(exp2)))
      as alias(table(proc(exp2)))) as alias(table(proc(exp'2)))
```

▷ (exp_1**:Sequence(***Basic***)**)−>**reverse().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*:Sequence(*Basic*))−>reverse()) es igual a:

```
select max(alias(table(proc(exp1))).pos)
  - alias(table(proc(exp1))).pos + 1 as pos,
  alias(table(proc(exp1))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

5.4 Expresiones iteradoras predefinidas

▷ *var*:*Basic*. Devuelve en OCL el mismo valor *var*. Su semántica se captura en MySQL utilizando una consulta. En concreto, query(*var*:*Basic*) es igual a:

```
select var as val
```

Patrón general de procedimiento almacenado que traduce operadores iteradores.

La semántica de las expresiones iteradoras se capturan en MySQL utilizando procedimientos. Más aún, existe un patrón general que captura todos los iteradores, aunque debemos instanciarlo para cada iterador. El patrón genereal se muestra a continuación, en el esquema 5.6.

```
create procedure proc♯(exp)
begin
declare done int default 0;
declare var [ tipo específico del cursor ];
declare crs cursor for [ traducción de la fuente ];
declare continue handler for sqlstate '02000 ' set done = 1;
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(colums(table(proc♯(exp))));
[ código específico de inicialización (sólo forAll, exists y one) ]
open crs;
  repeat
  fetch crs into var;
  if not done then
  call proc(cuerpo);
  [ código específico de procesamiento ]
  end if;
  until done end repeat;
close crs;
[ código específico de finalización (sólo sortedBy) ]
end;
```

Esquema 5.6: Patrón general para el mapeo de expesiones iteradoras como procedimientos almacenados.

Set

A continuación detallamos cada uno de los iteradores, para los cuales el 'hueco' *traducción de la fuente* en el patrón general mostrado en el esquema 5.6 se completa de la siguiente manera:

```
select val from table(proc(fuente))
```

▷ (*fuente*:**Set(***Basic***)**)−>**collect(***var*|*cuerpo***)** En concreto, el procedimiento proc♯((*fuente*:Set(*Basic*))−>collect(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del valor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos del cuerpo.
- *código de procesamiento del iterador:*

  ```
  insert into proc♯(expr)(val) (select * from proc(cuerpo));
  ```

▷ (*fuente*:**Set(***Basic***)**)−>**forAll(***var*|*cuerpo***)** En concreto, el procedimiento proc♯((*fuente*:Set(*Basic*))−>forAll(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* `Int`.
- *código específico de inicialización:*

  ```
  insert into table(proc♯(exp))(val) values (1);
  ```

- *código específico de procesamiento:*

  ```
  update table(proc♯(exp)) set val = 0
  where (select * from table(proc(cuerpo))) = 0;
  if exists (select 1 from table(proc♯(exp)) where val = 0)
  then set done = 1;
  end if;
  ```

▷ (*fuente*:**Set(***Basic***)**)−>**select(***var*|*cuerpo***)** En concreto, el procedimiento **proc**((*fuente*:Set(*Bas- ic*))−>select(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el esquema 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código específico de procesamiento:*

  ```
  if exists
    (select 1
    from
      (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
    where alias(table(proc(cuerpo))).val = 1)
  then insert into table(proc♯(exp))(val) values (var);
  end if;
  ```

5.5 Ejemplos

El contexto de las expresiones es un escenario concreto del diagrama de clases Coach_Company en la figura 2.3.

Ejemplo 8. *"Todos los viajes" se formaliza en OCL como:* Trip.allInstances(), *y se obtienen en MySQL los siguientes procedimentos (para cada una de las subexpresiones de la expresión de entrada, incluida ella misma).*

- *Trip.allInstances()*

```
create procedure allInstances0()
begin
call trip0;
call allInstances0Shp;
create table allInstances0Tbl(val int);
insert into allInstances0Tbl(val)
  (select val from allInstances0ShpTbl);
end;

create procedure allInstances0Shp()
begin
declare v varchar(250);
drop table if exists allInstances0ShpTbl;
create table allInstances0ShpTbl(val int);
select val into v from (select * from Trip0Tbl) as temp0;
case v
  when 'Trip' then
    insert into allInstancesShpTbl(val)
           (select pk as val from Trip);
  when 'RegularTrip' then
    insert into allInstancesShpTbl(val)
           (select pk as val from RegularTrip);
  when 'PrivateTrip' then
    insert into allInstancesShpTbl(val)
           (select pk as val from PrivateTrip);
  when 'Coach' then
    insert into allInstancesShpTbl(val)
           (select pk as val from Coach);
  when 'Person' then
    insert into allInstancesShpTbl(val)
           (select pk as val from Person);
  else
    insert into allInstancesShpTbl(val)
           (select val
           from (select null as val) as temp0
           where false);
end case;
end;
```

- *Trip*

```
create procedure trip0()
begin
create table trip0Tbl(val int);
insert into table trip0Tbl(val)
  (select 'Trip' as val);
end;
```

Ejemplo 9. *"Origenes de viajes" se formaliza en OCL como:* Trip.allInstances()-.origin *y se obtienen en MySQL los siguientes procedimentos (para cada una de las subexpresiones de la expresión de entrada, incluida ella misma).*

- *Trip.allInstances().origin*

```
create procedure origin0()
begin
create table origin0Tbl(val int);
insert into table origin0Tbl(val)
  (select Trip.origin as val
  from (select * from allInstances0Tbl) as temp0
  left join Trip on temp0.val = Trip.pk);
end;
```

Ejemplo 10. *"Viajes que tienen su origen en Madrid" se formaliza en OCL como:* Trip.allInstances()−>select(t|t.origin='Madrid') *y se obtienen en MySQL los siguientes procedimentos (para cada una de las subexpresiones de la expresión de entrada, incluida ella misma).*

- *Trip.allInstances()−>select(t|t.origin='Madrid')*

```
create procedure select0()
begin
call allInstances0;
call select0Shp;
create table select0Tbl(val int);
insert into select0Tbl(val)
  (select val from select0ShpTbl);
end;

create procedure select0Shp()
begin
declare done int default 0;
declare var int;
declare crs cursor for select val from allInstances0Tbl;
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists select0ShpTbl;
create table select0ShpTbl(val int);
open crs;
  repeat
  fetch crs into var;
  if not done then
    call equal0(var);
```

```
    if exists
      (select 1
      from
        (select * from equal0Tbl) as temp0
      where temp0.val = 1)
    then insert into select0ShpTbl(val) values (var);
    end if;
  end if;
  until done end repeat;
close crs;
end;
```

- *t.origin='Madrid'*

```
create procedure equal0()
begin
create table equal0Tbl(val boolean);
insert into table equal0Tbl(val)
  (select temp0.val = temp1.val
  from
    (select * from origin1Tbl) as temp0
    (select * from constM0Tbl) as temp1;
end;
```

- *'Madrid'*

```
create procedure constM0()
begin
create table constM0Tbl(val varchar(250));
insert into table contM0Tbl(val)
  (select 'Madrid' as val);
end;
```

- *t.origin*

```
create procedure origin1()
begin
create table origin1Tbl(val varchar(250));
insert into table origin1Tbl(val)
  (select Trip.origin as val
  from (select * from t0Tbl) as temp0
  left join Trip on temp0.val = Trip.pk);
end;
```

- *t*

```
create procedure vart0()
begin
create table vart0Tbl(val int);
insert into table vart0Tbl(val)
  (select t as val);
end;
```

6 Implementación

En este capítulo describimos brevemente la arquitectura de nuestra implementación de MySQL4OCL y reportamos sobre la "eficiencia" de las expresiones OCL compiladas, es decir, sobre el tiempo de ejecución de los procedimientos generados sobre escenarios medianos-grandes.

6.1 Arquitectura

La arquitectura de nuestra implementación MySQL4OCL, que denominaremos igualmente MySQL4OCL, se muestra en la figura 6.1. Los datos de entrada para MySQL4OCL son:

- un diagrama de clases UML *dcl* y
- una expresión OCL *exp*.

Los datos de salida de MySQL4OCL son:

- Las sentencias MySQL que generan las tablas correspondientes al diagrama de clases de entrada *dcl*, de acuerdo con las reglas **toRDB** definidas en el capítulo 4.
- Los procedimientos MySQL correspondientes a cada una de las subexpresiones OCL que aparecen en la expresión de entrada *exp* (incluida ella misma), de acuerdo con las reglas **toProc** definidas en el capítulo 5.

Los componentes de MySQL4OCL son:

- *EOS parser*. Es el analizador sintáctico de EOS [7], un interprete Java para la evaluación de expresiones OCL. Se encarga de generar el árbol léxico correspondiente a la expresión de entrada *exp*.
- *MySQL4OCL tabgen*. Se encarga de aplicar las reglas **toRDB** al diagrama de clases de entrada *dcl*.
- *MySQL4OCL codgen*. Se encarga de aplicar las reglas **toProc** a cada uno de las subexpresiones que aparecen en la expresión de entrada *exp* (incluida ella misma).

Los componentes *MySQL4OCL tabgen* y *MySQL4OCL codgen* son resultados originales de este trabajo. Ambos están escritos en Java y ocupan unas 5,000 líneas de código.

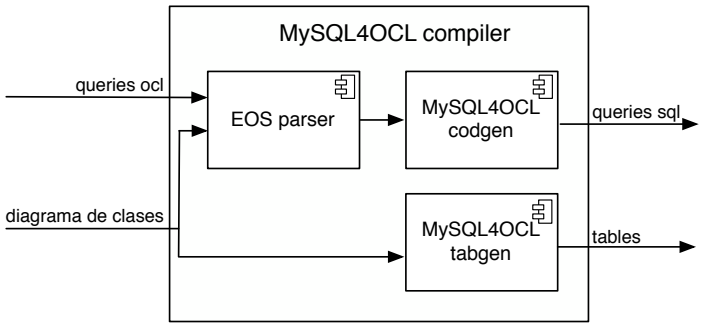

Figura 6.1: Arquitectura de la implementación de MySQL4OCL.

Optimización. De acuerdo con las reglas **toProc**, para cada subexpresión exp_i que aparece en la expresión de entrada *exp* (incluida ella misma), el componente *MySQL4OCL codgen* genera un procedimento proc(exp_i) que, al invocarse, creará una tabla permanente table(proc(exp_i)) donde se almacenarán los resultados de ejecución. Como es sabido, la creación de tablas tiene un coste en tiempo y espacio. Para evitar estos costes, hemos implementado un algoritmo "bottom-up" que optimiza el resultado de *MySQL4OCL codgen*, reemplazando, siempre que sea posible, la invocación del procedimiento $proc(exp_i)$ asociado a una subexpresiones exp_i por la ejecución de una consulta equivalente. Obviamente, esto sólo es posible (i) cuando el "cuerpo" del procedimiento no incluye ninguna llamada a otro procedimiento sino que él mismo es una consulta, o (ii) cuando incluye llamadas a procedimientos, pero estas pueden realizarse "fuera" del procedimiento, es decir, en el procedimiento asociado a la expresión de la que exp_i es una subexpresión inmediata.

Ejemplo 11. *La formalización en OCL de la pregunta acerca de los vehículos que, en un escenario concreto del diagrama de clases Coach_Company en la figura 2.3, realizan viajes que tienen un origen diferente a su destino es la siguiente expresión:*

Coach.allInstances()−>select(c|c.trips.origin<>c.trips.destination)

Si aplicamos el componente MySQL4OCL codgen *a esta expresión, obtenemos los siguientes procedimentos (para cada una de las subexpresiones de la expresión de entrada, incluida ella misma).*

- *Coach.allInstances()−>select(c|c.trips.origin<>c.trips.destination)*

```
create procedure select0()
begin
call allInstances00;
call select0Shp;
create table select0Tbl(val int);
insert into select0Tbl(val)
  (select val from select0ShpTbl);
end;
```

```
create procedure select0Shp()
begin
declare done int default 0;
declare var int;
declare crs cursor for select val from allInstances00Tbl;
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists select0ShpTbl;
create table select0ShpTbl(val int);
open crs;
  repeat
  fetch crs into var;
  if not done then
    call notEqual010(var);
    if exists
      (select 1
      from
        (select * from notEqual010Tbl) as temp0
      where temp0.val = 1)
    then insert into select0ShpTbl(val) values (var);
    end if;
  end if;
  until done end repeat;
close crs;
end;
```

- *Coach.allInstances()*

```
create procedure allInstances00()
begin
call coach000;
call allInstances00Shp;
create table allInstances00Tbl(val int);
insert into allInstances00Tbl(val)
  (select val from allInstances00ShpTbl);
end;

create procedure allInstances00Shp()
begin
declare v varchar(250);
drop table if exists allInstances00ShpTbl;
create table allInstances00ShpTbl(val int);
select val into v from (select val from coach000) as temp0;
case v
  when 'Trip' then
    insert into allInstances00ShpTbl(val)
          (select pk as val from Trip);
  when 'RegularTrip' then
    insert into allInstances000ShpTbl(val)
          (select pk as val from RegularTrip);
  when 'PrivateTrip' then
```

```
    insert into allInstances000ShpTbl(val)
           (select pk as val from PrivateTrip);
  when 'Coach' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from Coach);
  when 'Person' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from Person);
  else
    insert into allInstances00ShpTbl(val)
           (select val
           from (select null as val) as temp0
           where false);
end case;
end;
```

- *Coach*

```
create procedure coach000()
begin
create table coach000Tbl(val int);
insert into table coach000Tbl(val)
  (select 'Coach' as val);
end;
```

- *c.trips.origin<>c.trips.destination*

```
create procedure notEqual010(var int)
begin
create table notEqual010Tbl(val Boolean);
call origin0100(var int);
call destination0101(var int);
call notEqual010Shp(val int);
insert into notEqual010Tbl(val Boolean)
  (select val from notEqual010ShpTbl)
end;

create procedure notEqual010Shp(var int)
begin
declare result boolean;
declare size1 int;
declare size2 int;
declare size3 int;
declare done int default 0;
declare var1 int;
declare var2 int;
declare crs1 cursor for
  select val from origin0100Tbl order by val;
declare crs2 cursor for
  select val from destination0101Tbl order by val;
declare continue handler for sqlstate '02000' set done = 1;
```

```
select count(val) into size1
  from (select * from origin0100Tbl) as temp0;
select count(val) into size2
  from (select * from destination0101) as temp0;
if (size1 = size2) then
  open crs1;
  open crs2;
  set result = false;
  repeat
    fetch crs1 into var1;
    fetch crs2 into var2;
    if (var1 <> var2) then
      set result = true;
      set done = 1;
    end if;
  until done=1 end repeat;
  close crs1;
  close crs2;
else
  set result = true;
end if;
drop table if exists notEqual010ShpTbl;
create table notEqual010ShpTbl(val boolean);
insert into notEqual010ShpTbl(val) values (result);
end;
```

- *c.trips.origin*

```
create procedure origin0100(var int)
begin
call trips01000(val int);
select Trip.origin as val
from (select * from trips01000Tbl) as temp0
left join Trip on temp0.val = Trip.pk;
end;
```

- *c.trips*

```
create procedure trips01000(var int)
begin
create table trips01000Tbl(val int);
call varC010000(var int);
insert into trips01000Tbl(val)
  select tripCoach.trips as val
  from (select * from varC01000Tbl) as temp0
  left join tripCoach on temp0.val = tripCoach.coaches
  where tripCoach.trips is not null
    or temp0.val is null
end;
```

- *c*

```
create procedure varC010000(var int)
begin
create table varC010000Tbl(val int);
insert into trips01000Tbl(val)
  (select var as val);
end;
```

- *c.trips.destination*

```
create procedure destination0110(var int)
begin
call trips01100(val int);
select Trip.destination as val
from (select * from trips01100Tbl) as temp0
left join Trip on temp0.val = Trip.pk;
end;
```

- *c.trips*

```
create procedure trips01100(var int)
begin
create table trips01100Tbl(val int);
call varC011000(var int);
insert into trips01100Tbl(val)
  select tripCoach.trips as val
  from (select * from varC01100Tbl) as temp0
  left join tripCoach on temp0.val = tripCoach.coaches
  where tripCoach.trips is not null
    or temp0.val is null
end;
```

- *c*

```
create procedure varC011000(var int)
begin
create table varC011000Tbl(val int);
insert into trips01100Tbl(val)
  (select var as val);
end;
```

Ahora bien, si aplicamos nuestro algoritmo de optimización a este resultado de MySQL4OCL codgen *todos los procedimientos anteriores se "simplifican" en los siguientes:*

```
create procedure select0()
begin
call allInstances00Shp;
call select0Shp;
create table select0Tbl(val int);
insert into select0Tbl(val)
  (select val from select0ShpTbl);
end;
```

```
create procedure allInstances000Shp()
begin
declare v varchar(250);
drop table if exists allInstances000ShpTbl;
create table allInstances000ShpTbl(val int);
select val into v from (select 'Coach' as val) as temp0;
case v
  when 'Trip' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from Trip);
  when 'RegularTrip' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from RegularTrip);
  when 'PrivateTrip' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from PrivateTrip);
  when 'Coach' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from Coach);
  when 'Person' then
    insert into allInstances000ShpTbl(val)
           (select pk as val from Person);
  else
    insert into allInstances000ShpTbl(val)
           (select val
           from (select null as val) as temp0
           where false);
end case;
end;

create procedure select0Shp
begin
declare done int default 0;
declare var int;
declare crs cursor for select val from allInstances00ShpTbl;
declare continue handler for sqlstate '02000' set done = 1;
drop table if exists select0ShpTbl;
create table select0ShpTbl(val int);
open crs;
  repeat
  fetch crs into var;
  if not done then
    call notEqual010Shp(var);
    if exists
      (select 1
      from
        (select * from notEqual010ShpTbl) as temp0
      where temp0.val = 1)
    then insert into select0ShpTbl(val) values (var);
    end if;
```

```
  end if;
  until done end repeat;
close crs;
end;

create procedure notEqual010Shp(var int)
begin
declare result boolean;
declare size1 int;
declare size2 int;
declare size3 int;
declare done int default 0;
declare var1 int;
declare var2 int;
declare crs1 cursor for
  select temp2.val
  from (
    select Trip.origin as val
    from (select tripCoach.trips as val
      from (select var as val) as temp0
      left join tripCoach on temp0.val = tripCoach.coaches
      where tripCoach.trips is not null
      or temp0.val is null) as temp1
    left join Trip on temp1.val = Trip.pk) as temp2 order by val;
declare crs2 cursor for
  select temp2.val
  from (
    select Trip.destination as val
    from (
      select tripCoach.trips as val
      from (select var as val) as temp0
      left join tripCoach on temp0.val = tripCoach.coaches
      where tripCoach.trips is not null
      or temp0.val is null) as temp1
  left join Trip on temp1.val = Trip.pk) as temp2 order by val;
declare continue handler for sqlstate '02000' set done = 1;
select count(temp2.val) into size1
  from (
    select Trip.origin as val
    from (select tripCoach.trips as val
      from (select var as val) as temp0
      left join tripCoach on temp0.val = tripCoach.coaches
      where tripCoach.trips is not null
      or temp0.val is null) as temp1
    left join Trip on temp1.val = Trip.pk) as temp2;
select count(temp2.val) into size2
  from (
    select Trip.destination as val
    from (
      select tripCoach.trips as val
```

```
      from (select var as val) as temp0
      left join tripCoach on temp0.val = tripCoach.coaches
      where tripCoach.trips is not null
      or temp0.val is null) as temp1
    left join Trip
    on temp1.val = Trip.pk) as temp2;
if (size1 = size2) then
  open crs1;
  open crs2;
  set result = false;
  repeat
    fetch crs1 into var1;
    fetch crs2 into var2;
    if (var1 <> var2) then
      set result = true;
      set done = 1;
    end if;
  until done=1 end repeat;
  close crs1;
  close crs2;
else
  set result = true;
end if;
drop table if exists notEqual010ShpTbl;
create table notEqual010ShpTbl(val boolean);
insert into notEqual010ShpTbl(val) values (result);
end;
```

6.2 Eficiencia

Aunque el objetivo principal de nuestro trabajo no es la evaluación eficiente de expresiones OCL, sino más bien superar los límites de los evaluadores actuales para tratar con escenarios realmente grandes, hemos incluido en esta sección una comparativa sobre la eficiencia de MySQL4OCL y EOS [7] en la evaluación de expresiones OCL sobre escenarios pequeño-medianos. La eficiencia de MySQL4OCL se entiende en términos del tiempo de ejecución de los procedimientos generados por *MySQL4OCL codegen* sobre las bases de datos que almacenen los escenarios que sirvan de contexto para la evaluación de las expresiones.

Para nuestra comparativa utilizaremos, básicamente, el mismo banco de pruebas ("benchmark") que se utilizó [7]. En este punto es interesante señalar que, con respecto a este banco de pruebas, EOS demostró ser el evaluador más eficiente de OCL. En concreto, en nuestra comparativa, todas las expresiones se evaluan sobre una instancia del diagrama de clases Coach_Company en la figura 2.3, que contiene 10^3 viajes, servidos cada uno por 10 vehículos distintos, ninguno de ellos del tipo "Picasso". En el caso de MySQL4OCL, este escenario se ha almacenado previamente en una base de datos de acuerdo con las reglas **toRDB**. En el caso de EOS, este escenario ha sido previamente cargado en memoria. Los tiempos que consumen EOS y MySQL4OCL para evaluar las

distintas expresiones de nuestro "benchmark" se muestran en la tabla 6.1[1]. Es importante advertir que, dado que no es posible cargar en EOS, dentro de un tiempo razonable, escenarios realmente grandes (con millones de objetos), en las expresiones de nuestro "benchmark" se ha aumentado artificialmente el tamaño de las colecciones "fuente" sobre las que se evaluan las operaciones size, collect y forall: en concreto, p denota la expresión Coach.allInstances().trips.coaches, que, en nuestro escenario, evalúa a una colección de 10^5 vehículos.

Scenario I: 10^3 viajes $\times$ 10 vehículos no "Picasso", p = Coach.allInstances().trip.coaches	EOS	MySQL *4*OCL
`p->size()`	30ms	130ms
`p->collect(x|x.modelo)->size()`	80ms	7.38s
`p->collect(x|x.modelo <> 'Picasso')->size()`	90ms	7.25s
`p->collect(x|x.trips.coaches)->size()`	240ms	12.23s
`p->collect(x|x.trips.coaches->includes(x))->size()`	221ms	12.98s
`p->forAll(x|x.trips.coaches->includes(x))`	251ms	12.36s
`p->select(x|x.trips.coaches->includes(x))->size()`	260ms	14.87s
`p->collect(x|x.trips.coaches.modelo)->size()`	290ms	29.08s
`p->collect(x|x.trips.coaches.modelo->size())->sum()`	270ms	22.82s
`p->forAll(x|x.trips.coaches.modelo->excludes('Picasso'))`	280ms	22.36s

Cuadro 6.1: Comparación de eficiencia entre EOS y MySQL4OCL.

Como es lógico, sobre escenarios de tamaño pequeño-mediano, es más rápido evaluar expresiones OCL con EOS que ejecutar el código generado por MySQL4OCL. Es interesante, sin embargo, observar que el coste de ejecutar el código producido por MySQL4OCL parece depender, como en el caso de la EOS, de dos medidas: en primer lugar, del número máximo de veces que se accederá a las propiedades de los objetos y, en segundo lugar, del tamaño máximo de las colecciones que se han de construir.

Una vez más, recordamos que la ventaja de MySQL4OCL (y su razón principal de ser) se obtiene al evaluar expresiones OCL sobre escenarios realmente grandes. Como se discutió en [7], ninguno de los evaluadores OCL disponibles, incluyendo el propio EOS, pudo terminar de cargar un escenario con 10^6 coches en menos de 20 minutos (momento en el que se interrumpió la carga). Por contra, cargar este escenario en un servidor MySQL puede llevar menos de un minuto.

[1]En el caso de MySQL4OCL, el benchmark fue ejecutado en una computadora con dos procesadores de 2.40GHz, 2GB de RAM y los seteos por defecto para mySQL 5.1 Community Server. En el caso de EOS, el benchmark fue ejecutado, con un sólo procesador de 2GHz, 1GB de RAM, y el seteo de los parámetros JVM son -Xms y -Xmx to 1024m.

7 Conclusiones y trabajo futuro

En este trabajo hemos presentado un compilador de OCL en MySQL que permite la evaluación automática de expresiones OCL sobre bases de datos relacionales. De esta forma, salvamos —en el estilo propugnado por la metodología de desarrollo de software basado en modelos (MDA)— la distancia que separa el universo de los modelos (en el que diseñamos nuestras aplicaciones) del universo de las bases de datos (en el que ejecutamos estas aplicaciones). Este compilador, denominado MySQL4OCL, se define como una función recursiva sobre expresiones OCL y cubre un subconjunto muy significativo del lenguaje. En concreto, y a diferencia de propuestas anteriores [11, 12, 20, 34, 37, 28] que estaban basadas en "patrones", MySQL4OCL es capaz de traducir expresiones que contienen iteradores en toda su generalidad, es decir, también cuando estas expresiones incluyen iteradores anidados.

Las ideas principales que subyacen a la definición de MySQL4OCL fueron inicialmente propuestas en [16]: a saber, la utilización de procedimiento almacenados ("stored-procedures") para la traducción de operaciones iteradoras sobre colecciones. Sin embargo, en este trabajo, con el fin de presentar con mayor detalle la definición recursiva de MySQL4OCL, hemos utilizado procedimientos almacenados para traducir todas las operaciones OCL, y no sólo las operaciones iteradoras. Por lo demás, es importante señalar que las construcciones de MySQL que utilizamos en MySQL4OCL están también soportadas por otras bases de datos relacionales como Oracle y PostgresSQL, sólo lo diferencian pequeñas diferencias sintácticas.

Por último, hemos presentado en este trabajo nuestra implementación de MySQL4OCL como un componente Java [17], y hemos explicado su uso principalísimo dentro del entorno de desarrollo basado en modelos ActionGUI [8]. Como se ha explicado, este entorno permite la generación automática de aplicaciones para la gestión de bases de datos con políticas de control de acceso. La primera versión de ActionGUI fue presentada en [13] y la metodología de desarrollo que implementa es objeto del tutorial [1].

Como parte de nuestro trabajo futuro, pretendemos, en primer lugar, extender MySQL4OCL con el fin de cubrir también aquellas partes del lenguaje OCL que quedan actualmente fuera de nuestra traducción: en concreto, las operaciones sobre colecciones de colecciones y las operaciones sobre tuplas. Para ello, debemos generalizar primero el esquema que utilizamos para almacenar los resultados de las ejecuciones de los procedimientos que traducen las operaciones

OCL, de forma que estos resultados (cuando su tipo sea colección de colecciones o tuplas) puedan almacenarse en varias tablas, en vez de en una única tabla como hasta ahora.

En segundo lugar, nos planteamos abordar la traducción, dentro de MySQL-4OCL, de las operaciones OCL que el propio modelador pueda definir a partir de las operaciones del estándar del lenguaje. En este caso, la dificultad principal estriba en el manejo de la recursión que el modelador pueda utilizar para la definición de estas operaciones.

En tercer lugar, dentro del trabajo en MySQL4OCL, estudiaremos posibles optimizaciones del código MySQL que genera automáticamente nuestra traducción. En concreto, sería lógico *aplanar*, en una fase de post-compilación, las consultas que incluyan left-joins, con el fin de aprovecharse de las optimizaciones que proporciona MySQL para este operador. Además, sería interesante *reescribir*, en una fase de pre-compilación, las expresiones OCL que haya que traducir en otras que fueran lógicamente equivalentes pero para cuya traducción supiéramos que MySQL4OCL genera código más eficiente.

Por último, como parte de un proyecto de investigación más amplio sobre el modelado y análisis de políticas de privacidad y su implantación y monitorización automáticas sobre bases de datos, pretendemos utilizar MySQL4OCL como componente esencial de un sistema de control de acceso a bases de datos que garantice que las respuestas que éstas den a las consultas que reciban respetan las políticas de privacidad. Más en concreto, para aplicaciones para las que se ha modelado su política de privacidad utilizando OCL, aspiramos a construir, sirviéndonos de MySQL4OCL, un sistema de acceso a las bases de datos que traduzca automáticamente cada consulta en una consulta *segura*, es decir, en una consulta que contenga en sí misma las comprobaciones pertinentes para que su ejecución sea consistente con la política de privacidad especificada en el modelo.

A Definiciones adicionales

A.1 Operadores específicos del modelo

▷ (exp_1:[*Class* | *Col*(*Class*)]).*atr*. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:[*Class* | *Col*(*Class*)]).*atr*) es igual a:

- Caso: `atr` pertenece directamente a la clase $Class$.

  ```
  select nm(Class).nm(atr) as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  left join nm(Class)
  on alias(table(proc(exp1))).val = nm(Class).pk;
  ```

- Caso: `atr` está definido en una superclase de $Class$, a la que llamaremos $Class_s$.

  ```
  select nm(Class_s).nm(atr) as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  left join nm(Class_s)
  on alias(table(proc(exp1))).val = nm(Class_s).pk;
  ```

▷ (exp_1:[$Class_A$ | *Col*(*Class*$_A$)]).rl_A. Su semántica se captura en MySQL utilizando una consulta. Supongamos que rl_A y rl_B son los dos extremos de una asociación $Asoc$ entre una clase $Class_A$ y una clase $Class_B$ en el modelo que sirve como contexto a la expresión. Supongamos también que rl_A pertenece a la clase $Class_A$ y que rl_B pertenece a la clase $Class_B$. En concreto, query((exp_1:[$Class_A$ | *Col*(*Class*$_A$)]).rl_A) es igual a:

- asociación con multiplicidad *..*

  ```
  select nm(Asoc).nm(rl_A) as val
  from (select * from table(proc(exp1)))) as alias(table(proc(exp1)))
  left join nm(Asoc)
  on alias(table(proc(exp1))).val = nm(Asoc).nm(rl_B)
  where nm(Asoc).nm(rl_A) is not null
    or alias(table(proc(exp1))).val is null
  ```

- asociación con multiplicidad 0..1, 1..1 y 1..*

```
select nm(Class_A).nm(rl_A) as val
from (select * from table(proc(exp_1)))) as alias(table(proc(exp_1)))
left join nm(Class_A)
on alias(table(proc(exp_1))).val = nm(Class_A).pk
where nm(Class_A).nm(rl_A) is not null;
```

▷ $(exp_1$:*Class*$)$**.oclIsTypeOf(**exp_2**:Classifier)** Su semántica se captura en MySQL utilizando un procedimiento. Supongamos que $c_1, \ldots, c_n$ son las clases del modelo que sirven de contexto. Y llamemosle $c_i^1, \cdots, c_i^s$ al subconjunto de clases de $c_1, \ldots, c_n$ que son *subclases directas* de c_i. Entonces, el procedimiento $\text{proc}^\sharp(exp)$ donde $exp = (exp_1$:*Class*$)$.oclIsType- Of(exp_2:Classifier) se define como:

```
create procedure proc♯(exp)()
begin
declare v varchar(250);
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val Int);
select val into v
  from (select val from table(proc(exp_2))) as alias(table(proc(exp_2)));
case v
  when 'nm(c_1)' then
    insert into table(proc♯(exp))(val)
      select alias(table(proc(exp_1))).val in
        (select pk as val from nm(c_1))
       and alias(table(proc(exp_1))).val not in
        (select pk as val from nm(c_1^1))
          ⋮
       and alias(table(proc(exp_1))).val not in
        (select pk as val from nm(c_1^s))
      from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)));
   ⋮
  when 'nm(c_n)' then
    insert into table(proc♯(exp))(val)
      select alias(table(proc(exp_1))).val in
        (select pk as val from nm(c_n))
       and alias(table(proc(exp_1))).val not in
        (select pk as val from nm(c_n^1))
          ⋮
       and alias(table(proc(exp_1))).val not in
        (select pk as val from nm(c_n^s))
    from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)));
  else
    insert into table(proc♯(exp))(val) (select false as val);
end case;
end;
```

Esquema A.1: Procedimiento que describe la semántica de la operación oclIsTypeOf.

▷ (exp_1:*Class*)**.oclAsType(**exp_2**:Classifier)** Su semántica se captura en MySQL utilizando una consulta. En concreto, **query**((exp_1:*Class*).oclAsType(exp_2:Classifier)) es igual a:

```
select
  if(alias(table(proc(exp1))).val in (select val from table(proc(exp2))),
    alias(table(proc(exp1))).val, null) as val
from (table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1:*Class*)**.oclType()** Devuelve el tipo que evalua exp_1 como una instancia de éste. Su semántica se captura utilizando un procedimiento. Supongamos que $c_1, \ldots, c_n$ son las clases del modelo que sirven de contexto. Y llamemosle $c_i^1, \cdots, c_i^s$ al subconjunto de clases de $c_1, \ldots, c_n$ que son *subclases directas* de c_i. Entonces, el procedimiento proc$^\sharp(exp)$ donde $exp = (exp_1$:*Class*).oclType() se define como:

```
create procedure proc♯(exp)()
begin
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val Int);
if select alias(table(proc(exp1))).val in
       (select pk as val from nm(c1))
      and alias(table(proc(exp1))).val not in
       (select pk as val from nm(c1^1))
         ⋮
      and alias(table(proc(exp1))).val not in
       (select pk as val from nm(c1^s))
     from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
then
  insert into table(proc♯(exp))(val) (select nm(c1) as val);
elseif select alias(table(proc(exp1))).val in
       (select pk as val from nm(c2))
      and alias(table(proc(exp1))).val not in
       (select pk as val from nm(c2^1))
         ⋮
      and alias(table(proc(exp1))).val not in
       (select pk as val from nm(c2^s))
   from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
then
  insert into table(proc♯(exp))(val) (select nm(c2) as val);
         ⋮
elseif select alias(table(proc(exp1))).val in
       (select pk as val from nm(cn))
      and alias(table(proc(exp1))).val not in
       (select pk as val from nm(cn^1))
         ⋮
      and alias(table(proc(exp1))).val not in
```

```
      (select pk as val from nm(c_n^s))
  from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
then
  insert into table(proc♯(exp))(val) (select nm(c_n) as val);
end if;
end;
```

Esquema A.2: Procedimiento que describe la semántica de la operación oclType.

▷ (exp_1:*Basic*)**.oclIsUndefined()** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*:*Basic*).oclIsUndefined()) es igual a:

```
select isNull(alias(table(proc(exp_1))).val) as val
from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
```

A.2 Tipos primitivos

Real

Nota. Tengamos en cuenta que Integer es una subclase de tipo Real, luego se puede utilizar un número entero como parámetro de tipo Real.

▷ (exp_1**:Real**)**.+(** exp_2**:Real).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).+(exp_2:Real).) es igual a:

```
select alias(table(proc(exp_1))).val + alias(table(proc(exp_2))).val as val
from
  (select * from table(proc(exp_1))) as alias(table(proc(exp_1))),
  (select * from table(proc(exp_2))) as alias(table(proc(exp_2)))
```

▷ $-$(exp_1**:Real**)**.** Su semántica se captura en MySQL utilizando una consulta. En concreto, query($-$(exp_1:Real)) es igual a:

```
select -alias(table(proc(exp_1))).val as val
from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
```

▷ (exp_1**:Real**)**.abs().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).abs()) es igual a:

```
select abs(alias(table(proc(exp_1))).val) as val
from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
```

▷ (exp_1**:Real**)**.round().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).round()) es igual a:

```
select round(alias(table(proc(exp_1))).val) as val
from (select * from table(proc(exp_1))) as alias(table(proc(exp_1)))
```

▷ (*exp*:**Real**).**floor()**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).floor()) es igual a:

```
select floor(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (*exp*$_1$:**Real**).**max(***exp*$_2$:**Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).max(*exp*$_2$:Real)) es igual a:

```
select case
when alias(table(proc(exp1))).val > alias(table(proc(exp2))).val
  then alias(table(proc(exp1))).val as val
  else alias(table(proc(exp2))).val as val end
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$:**Real**).**min(***exp*$_2$:**Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).min(*exp*$_2$:Real)) es igual a:

```
select case
when alias(table(proc(exp1))).val < alias(table(proc(exp2))).val
  then alias(table(proc(exp1))).val as val
  else alias(table(proc(exp2))).val as val end
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$:**Real**).**=(***exp*$_2$:**Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).=(*exp*$_2$:Real)) es igual a:

```
select alias(table(proc(exp1))).val = alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$:**Real**).**<>(***exp*$_2$:**Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).<>(*exp*$_2$:Real)) es igual a:

```
select alias(table(proc(exp1))).val <> alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$:**Real**).**<(***exp*$_2$:**Real)**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Real).<(*exp*$_2$:Real)) es igual a:

```
select alias(table(proc(exp1))).val < alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1:**Real**).**>(**exp_2:**Real).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).>(exp_2:Real)) es igual a:

```
select alias(table(proc(exp1))).val > alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1:**Real**).≤**(**exp_2:**Real).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).≤(exp_2:Real)) es igual a:

```
select alias(table(proc(exp1))).val <= alias(table(proc(exp2))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1:**Real**).≥**(**exp_2:**Real).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Real).≥(exp_2:Real)) es igual a:

```
select alias(table(proc(exp1))).val <= alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1:**Real**).**toString().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp:Real).toString()) es igual a:

```
select concat(alias(table(proc(exp1))).val, '') as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Integer

▷ (exp_1:**Integer**).−**(**exp_2:**Integer).** Se traduce de igual manera que la operación − definida para el tipo Real.

▷ (exp_1:**Integer**).∗**(**exp_2:**Integer).** Se traduce de igual manera que la operación ∗ definida para el tipo Real.

▷ (exp_1:**Integer**)./**(**exp_2:**Integer).** Se traduce de igual manera que la operación / definida para el tipo Real.

▷ (exp_1:**Integer**).**abs()** Se traduce de igual manera que la operación abs definida para el tipo Real.

▷ (exp_1:**Integer**).**div(**exp_2:**Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Integer).div(exp_2:Integer)) es igual a:

```
select case
when alias(table(proc(exp2))).val <> 0 and
    alias(table(proc(exp1))).val/alias(table(proc(exp2))).val > 0
  then
    floor(alias(table(proc(exp1))).val/alias(table(proc(exp2))).val) as val
when
  alias(table(proc(exp2))).val <> 0 and alias(table(proc(exp1))).val < 0
  then
    -floor(
      -alias(table(proc(exp1))).val/alias(table(proc(exp2))).val) as val
else null as val end
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$**:Integer**)**.mod(***exp*$_2$**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Integer).mod(*exp*$_2$:Integer)) es igual a:

```
select
  mod(alias(table(proc(exp1))).val, alias(table(proc(exp2))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$**:Integer**)**.max(***exp*$_2$**:Integer).** Se traduce de igual manera que la operación max definida para el tipo Real.

▷ (*exp*$_1$**:Integer**)**.min(***exp*$_2$**:Integer).** Se traduce de igual manera que la operaciń min definida para el tipo Real.

▷ (*exp***:Integer**)**. toString()** Se traduce de igual manera que la operación toString definida para el tipo Real.

String

▷ (*exp*$_1$**:String**)**.=(***exp*$_2$**:String).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:String).=(*exp*$_2$:String)) es igual a:

```
select alias(table(proc(exp1))).val = alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*$_1$**:String**)**.+(***exp*$_2$**:String).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:String).+(*exp*$_2$:String)) es igual a:

```
select
  concat(alias(table(proc(exp1))).val, alias(table(proc(exp2))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:Integer**)**.size().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(((exp_1:Integer).size()) es igual a:

```
select length(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1**:String**)**.toInteger().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(((exp_1:String).toInteger()) es igual a:

```
select alias(table(proc(exp1))).val + 0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1**:String**)**.toReal().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(((exp_1:String).toReal()) es igual a:

```
select alias(table(proc(exp1))).val + 0.0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp**:String**)**.toUpperCase().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(((exp_1:String).toUpperCase()) es igual a:

```
select upper(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1**:String**)**.toLowerCase().** Su semántica se captura utilizando una consulta. En concreto, query(((exp_1:String).toLowerCase()) es igual a:

```
select lower(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1**:String**)**.equalsIgnoreCase(**exp_2**:String).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(((exp_1:String).equalsIgnoreCase(exp_2:String)) es igual a:

```
select alias(table(proc(exp1))).val = alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:String**).**substring(**exp_2**:Integer,** exp_3**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:String).substring(*exp*$_2$:Integer, *exp*$_3$:Integer)) es igual a:

```
select case
when alias(table(proc(exp2))).val > 0 and
  alias(table(proc(exp2))).val < alias(table(proc(exp3))).val and
  alias(table(proc(exp3))).val < length(alias(table(proc(exp1))).val)
then
  substring(alias(table(proc(expr1))).val,alias(table(proc(exp2))).val,
    alias(table(proc(exp3))).val) as val
else '' as val end
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2))),
  (select * from table(proc(exp3))) as alias(table(proc(exp3)))
```

▷ (exp_1**:String**).**indexOf(**exp_2**:String).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:String).indexOf(*exp*$_2$:String)) es igual a:

```
select case when length(alias(table(proc(exp2))).val) = 0
then 0 as val
else
  instr(alias(table(proc(exp1))).val, alias(table(proc(exp2))).val) as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:String**).**toBoolean().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:String).toBoolean()) es igual a:

```
select alias(table(proc(exp1))).val = 'true' as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Boolean

▷ (exp_1**:Boolean**).**xor(**exp_2**:Boolean).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Boolean).xor(*exp*$_2$:Boolean)) es igual a:

```
select alias(table(proc(exp1))).val xor alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:Boolean**).**implies(**exp_2**:Boolean).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Boolean).implies(*exp*$_2$:Boolean)) es igual a:

```
select not(alias(table(proc(exp1))).val)
  or alias(table(proc(exp2))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (*exp*:**Boolean**).**toString().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*:Boolean).toString()) es igual a:

```
select cast(alias(table(proc(exp1))).value as char) as value
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

A.3 Tipos colección

Collection

▷ (exp_1:**Collection(***Basic***)**)−>**sum().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Collection(*Basic*))−>sum()) es igual a:

```
select sum(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1:**Collection(***Basic***)**)−>**max().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Collection(*Basic*))−>max()) es igual a:

```
select max(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (exp_1:**Collection(***Basic***)**)−>**min().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Collection(*Basic*))−>min()) es igual a:

```
select min(alias(table(proc(exp1))).val) as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Set

▷ (exp_1:**Set(***Basic***)**).<>(*exp*$_2$:**Set(***Basic***)).** Su semántica se captura en MySQL utilizando un procedimiento. En este procedimiento, básicamente,

- Calcula, en primer lugar, el cardinal de cada conjunto.
- Si los cardinales son diferentes, entonces los conjuntos no pueden ser iguales, por lo que terminamos el procedimiento y devolvemos '`true`'.
- Si los cardinales coinciden, declara dos cursores que utilíza para iterar sobre cada uno de los conjuntos; además, ordena ambos conjuntos utilizando un mismo criterio.

- A continuación, itera sobre los dos conjuntos, comparando en cada paso de la iteración, los valores apuntados por ambos cursores.
- Si en algún paso de la iteración encuentra una desigualdad entre los valores apuntados por ambos cursores, termina el procedimiento y devuelve '`true`'; en caso contrario devuelve '`false`' al final de la iteración.

Entonces, el procedimiento proc$^\sharp(exp)$,
donde $exp = (exp_1$:Set($Basic$)).<>(exp_2:Set($Basic$)) se define como:

```
create procedure proc♯(exp)
begin
declare result boolean;
declare size1 int;
declare size2 int;
declare size3 int;
declare done int default 0;
declare var1 type(Basic);
declare var2 type(Basic);
declare crs1 cursor for
  select alias(table(proc(exp1))).val
  from (table(proc(exp1))) as alias(table(proc(exp1)))
  order by alias(table(proc(exp1))).val;
declare crs2 cursor for
  select alias(table(proc(exp2))).val
  from (table(proc(exp2))) as alias(table(proc(exp2)))
  order by alias(table(proc(exp2))).val;
declare continue handler for sqlstate '02000' set done = 1;
select count(val) into size1
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
select count(val) into size2
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)));
if (size1 = size2) then
  open crs1;
  open crs2;
  set result = false;
  repeat
    fetch crs1 into var1;
    fetch crs2 into var2;
    if (var1 <> var2) then
      set result = true;
      set done = 1;
    end if;
  until done=1 end repeat;
  close crs1;
  close crs2;
else
  set result = true;
end if;
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(val boolean);
```

```
insert into table(proc♯(exp))(val) values (result);
end;
```

Esquema A.3: Procedimiento que describe la semántica de la operación de desigualdad entre conjuntos.

▷ (exp_1**:Set(**$Basic$**)−>includes(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>includes(exp_2:$Basic$)) es igual a:

```
select alias(table(proc(exp2))).val
     in (select * from table(proc(exp1))) as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:Set(**$Basic$**)−>excludes(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>excludes(exp_2:$Basic$)) es igual a:

```
select alias(table(proc(exp2))).val
    not in (select val from table(proc(exp1))) as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:Set(**$Basic$**)−>includesAll(**exp_2**:Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>includesAll(exp_2:Set($Basic$))) es igual a:

```
select count(alias(table(proc(exp2))).val) = 0 as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2))),
where
  alias(table(proc(exp2))).val not in (select val from table(proc(exp1)))
```

▷ (exp_1**:Set(**$Basic$**)−>excludesAll(**exp_2**:Collection(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, **query**(exp_1:Set($Basic$))−>excludesAll(exp_2:Collection($Basic$))) es igual a:

```
select count(alias(table(proc(exp2))).val) = 0 as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
where alias(table(proc(exp2))).val
    in (select val from table(proc(exp1)))
```

▷ (exp_1**:Set(**$Basic$**)−>union(**exp_2**:Bag(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>union(exp_2:Bag($Basic$))) es igual a:

```
select *
from
  (select * from table(proc(exp1))
  union all
  select * from table(proc(exp2))) as alias(table(proc(exp1)))
```

▷ (exp_1**:Set(**$Basic$**)−>intersection(**exp_2**:Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>intersection(exp_2:Set($Basic$))) es igual a:

```
select distinct alias(table(proc(exp1))).val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
left join (select * from table(proc(exp2))) as alias(table(proc(exp2)))
on alias(table(proc(exp1))).val = alias(table(proc(exp2))).val
```

▷ (exp_1**:Set(**$basic$**)−>intersection(**exp_2**:Bag(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($basic$))−>intersection(exp_2:Bag($Basic$))) es igual a:

```
select alias(table(proc(exp1))).val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
left join
  (select distinct alias(table(proc(exp2))).val
  from
    (select *
    from table(proc(exp2))) as table(proc(exp2)))
      as alias(table(proc(exp2')))
on alias(table(proc(exp1))).val = alias(table(proc(exp2'))).val
```

▷ (exp_1**:Set(**$Basic$**)−>−(**exp_2**:Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>−(exp_2:Set($Basic$))) es igual a:

```
select alias(table(proc(exp1))).val
from (alias(table(proc(exp1)))) as alias(table(proc(exp1)))
where alias(table(proc(exp1))).val
    not in (select val from table(proc(exp2)))
```

▷ (exp_1**:Set(**$Basic$**)−>including(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, **query**((exp_1:Set($Basic$))−>including(exp_2:$Basic$)) es igual a:

```
select * from
  (select * from table(proc(exp1))
  union
  select * from table(proc(exp2))) as alias(table(proc(exp1)))
```

▷ (exp_1**:Set(**$Basic$**)−>excluding(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>excluding(exp_2:$Basic$)) es igual a:

```
select alias(table(proc(exp1))).val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
where alias(table(proc(exp1))).val != alias(table(proc(exp2))).val
```

▷ (exp_1**:Set(**$Basic$**)**)−>**symmetricDifference(**exp_2**:Set(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1 :Set($Basic$))−>symmetricDifference(exp_2:Set($Basic$))) es igual a:

```
select * from
  (select alias(table(proc(exp1))).val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  where alias(table(proc(exp1))).val not in
    (select val from table(proc(exp2)))
union all
  select alias(table(proc(exp2))).val
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
  where alias(table(proc(exp2))).val not in
    (select val from table(proc(exp1)))) as alias(table(proc(exp'1)))
```

▷ (exp_1**:Set(**$Basic$**)**)−>**asSet().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>asSet()) es igual a:

```
select * from table(proc(exp1))
```

▷ (exp_1**:Set(**$Basic$**)**)−>**asOrderedSet().** Su semántica se captura en MySQL utilizando un procedimiento. Entonces, el procedimiento proc$^\sharp$(exp), donde exp = (exp_1:Set($Basic$))−>asOrderedSet() se define como:

```
create procedure proc♯(exp)()
begin
drop table if exists proc♯(exp);
create table table(proc♯(exp))(pos INT not null auto_increment,
     val type(Basic), primary key(pos));
insert into table(proc♯(exp))(val)
  select alias(table(proc♯(exp1))).val as val
  from (select * from table(proc♯(exp1))) as alias(table(proc♯(exp1)));
end;
```

Esquema A.4: Procedimiento que describe la semántica de la operación asOrderedSet.

▷ exp_1**:Set(**$Basic$**)**)−>**asBag().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:Set($Basic$))−>asBag()) es igual a:

```
select * from table(proc(exp1))
```

▷ (exp_1**:Set(**$Basic$**)**)−>**asSequence().** Su semántica se captura en MySQL utilizando un procedimiento. Entonces, el procedimiento proc$^\sharp$(exp) donde exp = (exp_1:Set($Basic$))−>asSequence() se define como:

```
create procedure proc♯(exp)()
begin
drop table if exists proc♯(exp);
create table table(proc♯(exp))(pos INT not null auto_increment,
```

```
     val Basic, primary key(pos));
insert into table(proc♯(exp))(val)
  select alias(table(proc(exp1))).val as val as
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
end;
```

Esquema A.5: Procedimiento que describe la semántica de la operación asSequence.

OrderedSet

▷ (exp_1**:OrderedSet(**$Basic$**)−>includes(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:OrderedSet($Basic$))−>includes(exp_2:$Basic$)) es igual a:

```
select
  alias(table(proc(exp2))).val in
    (select alias(table(proc(exp1))).val
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:OrderedSet(**$Basic$**)−>excludes(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((exp_1:OrderedSet($Basic$))−>excludes(exp_2:$Basic$)) es igual a:

```
select
  alias(table(proc(exp2))).val not in
    (select alias(table(proc(exp1))).val
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as val
from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
```

▷ (exp_1**:orderedSet(**$Basic$**)−>includesAll(**exp_2**:Collection(**$Basic$**)).** Su semántica se captura en MySQL utilizando una consulta. En concreto, **query**((exp_1:orderedSet($Basic$))−>includesAll(exp_2:Collection($Basic$))) es igual a:

```
select count(alias(table(proc(exp1))).val) = 0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
where alias(table(proc(exp2))).val in
  (select alias(table(proc(exp1))).val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1))))
```

▷ (exp_1**:orderedSet(**$Basic$**)−>excludesAll(**exp_2**:Collection(**$Basic$**)).** Su semántica se captura utilizando una consulta. En concreto, query((exp_1:OrderedSet($Basic$))−>excludesAll(exp_2:Collection(Ba- sic))) es igual a:

```
select count(alias(table(proc(exp1))).val) = 0 as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
where alias(table(proc(exp2))).val not in
```

```
(select alias(table(proc(exp1))).val
from (select * from table(proc(exp1))) as alias(table(proc(exp1))))
```

▷ (*exp*$_1$**:OrderedSet(***Basic***)).subOrderedSet(***exp*$_2$**:Integer,***exp*$_3$**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, **query**((*exp*$_1$:OrderedSet(*Basic*)).subOrderedSet(*exp*$_2$:Integer,*exp*$_3$:Integer)) es igual a:

```
select
  alias(table(proc(exp1))).pos - alias(table(proc(exp2))).val + 1 as pos,
  alias(table(proc(exp1))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2))),
  (select * from table(proc(exp3))) as alias(table(proc(exp3)))
where alias(table(proc(exp2))).val < alias(table(proc(exp1))).pos and
  alias(table(proc(exp3))).val + alias(table(proc(exp2))).val
    >= alias(table(proc(exp1))).pos
```

▷ (*exp*$_1$**:OrderedSet(***Basic***))−>at(***exp*$_2$**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>at(*exp*$_2$:Integer)) es igual a:

```
select alias(table(proc(exp1))).val as val
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
where alias(table(proc(exp1))).pos = alias(table(proc(exp2))).val
```

▷ (*exp*$_1$**:OrderedSet(***Basic***))−>indexOf(***exp*$_2$**:Integer).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>indexOf(*exp*$_2$:Integer)) es igual a:

```
select alias(table(proc(exp1))).pos as pos
from
  (select * from table(proc(exp1))) as alias(table(proc(exp1))),
  (select * from table(proc(exp2))) as alias(table(proc(exp2)))
where alias(table(proc(exp1))).val = alias(table(proc(exp2))).val limit 1
```

▷ (*exp*$_1$**:OrderedSet(***Basic***))−>reverse().** Se traduce de igual manera que la operación reverse definida entre secuencias.

▷ (*exp*$_1$**:OrderedSet(***Basic***))−>asSet().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>asSet()) es igual a:

```
select alias(table(proc(exp1))).val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

▷ (*exp*:**OrderedSet(***Basic***)**)−>**asOrderedSet().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query(*exp*$_1$:OrderedSet(*Basic*))−>asOrderedSet() es igual a:

```
select * from table(proc(exp1))
```

▷ (*exp*$_1$:**OrderedSet(***Basic***)**)−>**asSequence().** . Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>asSequence()) es igual a:

```
select * from table(proc(exp1))
```

▷ (*exp*$_1$:**OrderedSet(***Basic***)**)−>**asBag().** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:OrderedSet(*Basic*))−>asBag()) es igual a:

```
select alias(table(proc(exp1))).val as val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
```

Bag

▷ (*exp*$_1$:**Bag(***Basic***)**).**=(***exp*$_2$:**Bag(***Basic***))**. Se traduce de igual manera que la operación = definida entre conjuntos.

▷ (*exp*$_1$:**Bag(***Basic***)**).**union(***exp*$_2$:**Bag(***Basic***))**. Se traduce de igual manera que la operación union definida entre un conjunto y un multiconjunto.

▷ (*exp*$_1$:**Bag(***Basic***)**).**union(***exp*$_2$:**Set(***Basic***))**. Se traduce de igual manera que la operación union definida entre un conjunto y un multiconjunto.

▷ (*exp*$_1$:**Bag(***Basic***)**).**intersection(***exp*$_2$:**Set(***Basic***))**. Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Bag(*Basic*)).intersection(*exp*$_2$:Set(*Basic*))) es igual a:

```
select distinct alias(table(proc(exp1))).val
from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
left join
  (select alias(table(proc(exp2))).val
  from (select * from table(proc(exp2)))
    as alias(table(proc(exp2)))) as alias(table(proc(exp'2)))
on alias(table(proc(exp1))).val = alias(table(proc(exp'2))).val
```

▷ (*exp*$_1$:**Bag(***Basic***)**)−>**excluding(***exp*$_2$:*Basic***).** Se traduce de igual manera que la operación excluding definida para conjuntos.

▷ (*exp*$_1$:**Bag(***Basic***)**)−>**asBag().** Se traduce de igual manera que la operación asBag definida para conjuntos.

▷ (exp_1**:Bag(**$Basic$**)**)−>**asSequence().** Se traduce de igual manera que la operación asSequence definida para conjuntos.

▷ (exp_1**:Bag(**$Basic$**)**)−>**asSet().** Esta operación se traduce de igual manera que la operación asSet definida para conjuntos ordenados.

▷ (exp_1**:Bag(**$Basic$**)**)−>**asOrderedSet().** Su semántica se captura en MySQL utilizando un procedimiento. Entonces, el procedimiento proc(exp), donde exp = (exp_1:Bag($Basic$))−>asOrderedSet() se define como:

```
create procedure proc♯(exp)()
begin
drop table if exists table(proc♯(exp));
create table table(proc♯(exp))(pos INT not null auto_increment,
                    val type(Basic), primary key(pos));
insert into table(proc♯(exp))(val)
  select distinct alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)));
end;
```

Esquema A.6: Procedimiento que describe la semántica de la operación asOrderedSet.

Sequence

▷ (exp_1**:Sequence(**$Basic$**)**)−>**prepend(**exp_2**:**$Basic$**).** Su semántica se captura utilizando una consulta. En concreto, query((exp_1:Sequence($Basic$))−>prepend(exp_2:$Basic$)) es igual a:

```
select * from
  (select 1 as pos, alias(table(proc(exp2))).val as val
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
  union all
  select alias(table(proc(exp1))).pos + 1 as pos,
    alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1)))
    as alias(table(proc(exp1)))) as alias(table(proc(exp'1)))
```

▷ (exp_1**:Sequence(**$Basic$**)).=(**exp_2**:Sequence(**$Basic$**)).** Esta operación se traduce de igual manera que la operación = definida para conjuntos ordenados.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**includes(**exp_2**:**$Basic$**).** Esta operación se traduce de igual manera que la operación includes definida para conjuntos ordenados.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**excludes(**exp_2**:**$Basic$**).** Esta operación se traduce de igual manera que la operación excludes definida para conjuntos ordenados.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**includesAll(**exp_2**:Collection(**$Basic$**)).** Esta operación se traduce de igual manera que la operación includesAll definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)**)**−>excludesAll(***exp*$_2$**:Collection(***Basic***)).** Se traduce de igual manera que la operación excludesAll definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)**)**−>union(***exp*$_2$**:Sequence(***Basic***)).** Su semántica se captura utilizando una consulta. En concreto, query((*exp*$_1$:Sequence(*Basic*))−>union(*exp*$_2$:*Basic*)) es igual a:

```
select * from
  (select alias(table(proc(exp1))).pos as pos,
    alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  union all
  select alias(table(proc(exp'1))).val + alias(table(proc(exp2))).pos as pos,
    alias(table(proc(exp2))).val as val
  from
    (select max(alias(table(proc(exp1))).pos) as val
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as alias(table(proc(exp'1))),
    (select * from table(proc(exp2)))
      as alias(table(proc(exp2))))
```

▷ (*exp*$_1$**:Sequence(***Basic***)**)**−>prepend(***exp*$_2$**:***Basic***).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Sequence(*Basic*))−>prepend(*exp*$_2$:*Basic*)) es igual a:

```
select * from
  (select 1 as pos, alias(table(proc(exp2))).val as val
  from (select * from table(proc(exp2))) as alias(table(proc(exp2)))
  union all
  select alias(table(proc(exp1))).pos + 1 as pos,
    alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1)))
    as alias(table(proc(exp1)))) as alias(table(proc(exp'1)))
```

▷ (*exp*$_1$**:Sequence(***Basic***)**)**−>insertAt(***exp*$_2$**:Integer,***exp*$_3$**:***Basic***).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Sequence(*Basic*))−>insertAt(*exp*$_2$:Integer,*exp*$_3$:*Basic*)) es igual a:

```
select * from
  (select * from
   ((select alias(table(proc(exp1))).pos as pos,
      alias(table(proc(exp1))).val as val
    from (select * from table(proc(exp1))) as alias(table(proc(exp1))),
     (select * from table(proc(exp2))) as alias(table(proc(exp2)))
    where alias(table(proc(exp2))).val > alias(table(proc(exp1))).pos)
    union all
    (select alias(table(proc(exp2))).val as pos,
      alias(table(proc(exp3))).val as val
    from
```

```
    (select * from table(proc(exp2))) as alias(table(proc(exp2))),
    (select * from table(proc(exp3))) as alias(table(proc(exp3)))))
      as alias(table(proc(exp'1)))
  union
   (select alias(table(proc(exp1))).pos + 1 as pos,
     alias(table(proc(exp1))).val as val
   from (select * from table(proc(exp1))) as alias(table(proc(exp1))),
    (select * from table(proc(exp2))) as alias(table(proc(exp2)))
   where alias(table(proc(exp2))).val <= alias(table(proc(exp1))).pos))
    as alias(table(proc(exp'2)))
```

▷ (*exp*$_1$**:Sequence(***Basic***)).subSequence(***exp*$_2$**:Integer,***exp*$_3$**:Integer).** Se traduce de igual manera que la operación subOrderedSet definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)->at(***exp*$_2$**:Integer,***exp*$_3$**:***Basic***).** Se traduce de igual manera que la operación at definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)->indexOf(***exp*$_2$**:***Basic***,***exp*$_3$**:***Basic***).** Se traduce de igual manera que la operación indexOf definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)->first().** Se traduce de igual manera que la operación first definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)->last().** Se traduce de igual manera que la operación last definida para conjuntos ordenados.

▷ (*exp*$_1$**:Sequence(***Basic***)->including(***exp*$_2$**:***Basic***).** Su semántica se captura en MySQL utilizando una consulta. En concreto, query((*exp*$_1$:Sequence(*Basic*))->including(*exp*$_2$:*Basic*)) es igual a:

```
select * from
  (select alias(table(proc(exp1))).pos as pos,
    alias(table(proc(exp1))).val as val
  from (select * from table(proc(exp1))) as alias(table(proc(exp1)))
  union all
  select alias(table(proc(exp'1))).val + 1 as pos,
    alias(table(proc(exp2))).val as val
  from
    (select max(alias(table(proc(exp1))).pos)
    from (select * from table(proc(exp1)))
      as alias(table(proc(exp1)))) as alias(table(proc(exp'1))),
    (select * from table(proc(exp2)))
      as alias(table(proc(exp2)))) as alias(table(proc(exp'2)))
```

▷ (exp_1**:Sequence(**$Basic$**)**)−>**excluding(**exp_2**:**$Basic$**).** Su semántica se captura en MySQL utilizando un procedimiento. Entonces, el procedimiento proc(exp), donde exp = (exp_1:Sequence($Basic$))−>excluding(exp_2:$Basic$) se define como:

```
create procedure proc(exp)()
begin
  drop table if exists proc(exp);
  create table proc(exp)(pos INT not null auto_increment,
                val type(Basic), primary key(pos));
    insert into proc(exp)(val)
      (select alias(table(proc(exp1))).val as val
      from
        (select * from table(proc(exp1))) as alias(table(proc(exp1))),
        (select * from table(proc(exp2))) as alias(table(proc(exp2)))
      where alias(table(proc(exp1))).val != alias(table(proc(exp2))).val
end;
```

Esquema A.7: Procedimiento que describe la semántica de la operación excluding.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**asBag().** Se traduce de igual manera que la operación asBag definida para conjuntos ordenados.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**asSequence().** Se traduce de igual manera que la operación asSequence definida para conjuntos ordenados.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**asSet().** Se traduce de igual manera que la operación asSet definida para multiconjuntos.

▷ (exp_1**:Sequence(**$Basic$**)**)−>**asOrderedSet().** Se traduce de igual manera que la operación asOrderedSet definida para multiconjuntos.

A.4 Expresiones iteradoras predefinidas

Set

A continuación detallamos cada uno de los iteradores, para los cuales el 'hueco' *traducción de la fuente* en el patrón general mostrado en el esquema 5.6 se completa de la siguiente manera: 0

```
select val from table(proc(fuente))
```

▷ ($fuente$**:Set(**$Basic$**)**)−>**exists(**$var|cuerpo$**)** En concreto, el procedimiento proc$^\sharp$(($fuente$:Set($Basic$))−>exists($var|cuerpo$)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* `Int`.
- *código específico de inicialización:*

```
insert into table(proc♯(exp))(val) values (0);
```

- *código específico de procesamiento:*

```
update table(proc♯(exp)) set val = 1
where (select * from table(proc(cuerpo))) = 1;
if exists (select 1 from table(proc♯(exp)) where val = 1)
then set done = 1;
end if;
```

▷ (*fuente*:**Set(***Basic***)**)−>**one(***var*|*cuerpo***)** En concreto, el procedimiento proc♯((*fuente*:Set(*Basic*))−>one(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* `Int`.
- *código de inicialización:*

```
declare counter int default 0;[1]
insert into table(proc♯(exp)) (val) values (0);
```

- *código de procesamiento del iterador:*

```
if exists
  (select 1
  from
    (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
  where val = 1)
then
  set counter = counter +1;
  update table(proc♯(exp)) set val = 1;
end if;
if counter = 2 then
  update table(proc♯(exp)) set val = 0;
  set done = 1;
end if;
```

▷ (*fuente*:**Set(***Basic***)**)−>**any(***var*|*cuerpo***)** En concreto, el procedimiento proc♯((*fuente*:Set(*Basic*))−>any(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código de procesamiento del iterador:*

[1] Esta declaración es colocada antes de crear la tabla debido a una restricción de los procedimientos almacenados de MySQL que solo permiten que las declaraciones de variables ocurran al inicio del procedimiento.

```
if exists
  (select 1
  from
    (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
  where val = 1)
then
  insert into table(proc♯(exp))(val) values (var);
  set done = 1;
end if;
```

▷ (*fuente*:**Set(***Basic***)**)−>**reject(***var*|*cuerpo***)** En concreto, el procedimiento proc((*fuente*:Set(*Basic*))−>reject(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el esquema 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código específico de procesamiento:*

```
if exists
  (select 1
  from
    (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
  where alias(table(proc(cuerpo))).val = 0)
then insert into table(proc♯(exp))(val) values (var);
end if;
```

▷ (*fuente*:**Set(***Basic***)**)−>**sortedBy(***var*|*cuerpo***)** En concreto, el procedimiento **proc**((*fuente*:Set(*Basic*)) −>sortedBy(*var*|*cuerpo*))[2] se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el esquema 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código de inicialización:*

```
create table table(proc(exp) − disord)
  (val tipo específico del valor-fuente,
  body tipo específico del valor-cuerpo);

create table table(proc♯(exp))(pos Int not null auto_increment,
  val tipo específico del valor-fuente, primary key(pos));
```

[2]Nótese que para poder ordenar los valores del cuerpo es necesario que haya una función de orden total definida sobre el tipo del mismo. No consideramos los tipos colecciones sobre el cuerpo.

- *tipo específico del valor-fuente:* `val type`. Donde `type` es el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos del fuente.
- *tipo específico del valor-cuerpo:* `body type`. Donde `type` es el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos del cuerpo.
- *código específico de procesamiento:*

  `insert into` table(proc$^\sharp$(*exp*))`(val, body)`
  `select (select var as val)`
  `(select val from` table(proc(*cuerpo*)))`;`

- *código de finalización de procedimiento:*

  `insert into` table(proc$^\sharp$(*exp*))`(val)`
  `(select val from` table(proc(*exp*) − *disord*) `order by body desc);`

Bag

▷ (*fuente*:**Bag(***Basic***)**−>**collect(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador collect definida entre conjuntos.

▷ (*fuente*:**Bag(***Basic***)**−>**forAll(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos, salvo por lo detallado a continuación.

- *traducción de la fuente:*

  `select distinct` alias(table(proc(*fuente*)))`.val`
  `from (select * from` table(proc(*fuente*)))` as` alias(table(proc(*fuente*)))

▷ (*fuente*:**Bag(***Basic***)**−>**exists(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador exists definida entre conjuntos, salvo por lo detallado a continuación.

- *traducción de la fuente:*

  `select distinct` alias(table(proc(*fuente*)))`.val`
  `from (select * from` table(proc(*fuente*)))` as` alias(table(proc(*fuente*)))

▷ (*fuente*:**Bag(***Basic***)**−>**one(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador one definida entre conjuntos.

▷ (*fuente*:**Bag(***Basic***)**−>**any(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador any definida entre conjuntos.

▷ (*fuente*:**Bag(***Basic***)**−>**select(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador select definida entre conjuntos.

▷ (*fuente*:**Bag(***Basic***)**)−>**reject(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador reject definida entre conjuntos.

▷ (*fuente*:**Bag(***Basic***)**)−>**sortedBy(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador sortedBy definida entre conjuntos.

OrderedSet

A continuación detallamos cada uno de los iteradores, para los cuales el 'hueco' *traducción de la fuente* en el patrón general mostrado en la figura 5.6 se completa de la siguiente manera:

```
select alias(table(proc(fuente))).val
from (select * from table(proc(fuente))) as alias(table(proc(fuente)))
order by alias(table(proc(fuente))).pos
```

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**collect(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos.

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**forAll(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos, salvo por lo detallado anteriormente sobre la *traducción de la fuente*.

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**exists(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador exists definida entre conjuntos, salvo por lo detallado anteriormente sobre la *traducción de la fuente*.

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**one(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador one definida entre conjuntos, salvo por lo detallado anteriormente sobre la *traducción de la fuente*.

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**any(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador any definida entre conjuntos, salvo por lo detallado anteriormente sobre la *traducción de la fuente*.

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**select(***var*|*cuerpo***)** En concreto, el procedimiento proc((*fuente*:OrderedSet(*Basic*))−>select(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el esquema 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código de procesamiento del iterador:*

```
if exists
  (select 1
  from
    (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
  where val = 1)
```

```
then
  insert into table(proc♯(exp))(val) values (var);
end if;
```

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**reject(***var*|*cuerpo***)** En concreto, el procedimiento proc((*fuente*:OrderedSet(*Basic*))−>select(*var*|*cuerpo*)) se define como se muestra en el esquema 5.6. Donde los 'huecos' en el patrón general mostrado en el esquema 5.6 introducido anteriormente se completan de la siguiente manera:

- *tipo específico del cursor:* el tipo de MySQL, que representa, de acuerdo a nuestra transformación, el tipo de los elementos de la fuente.
- *código de procesamiento del iterador:*

```
if exists
  (select 1
  from
    (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
  where val = 0)
then insert into table(proc♯(exp))(val) values (var);
end if;
```

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**sortedBy(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador sortedBy definida entre conjuntos, salvo por lo el detalle sobre la *traducción de la fuente*.

- *traducción de la fuente:*

```
select alias(table(proc(fuente))).val
from (select * from table(proc(fuente))) as alias(table(proc(fuente)))
order by alias(table(proc(fuente))).pos
```

Sequence

▷ (*fuente*:**OrderedSet(***Basic***)**)−>**collect(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador collect definida entre conjuntos, salvo por lo detallado a continuación:

```
select alias(table(proc(fuente))).val
from (select * from table(proc(fuente))) as alias(table(proc(fuente)))
order by alias(table(proc(fuente))).pos
```

▷ (*fuente*:**Sequence(***Basic***)**)−>**forAll(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos ordenados, salvo por lo detallado a continuación:

- *traducción de la fuente:*

```
select distinct alias(table(proc(cuerpo))).val
from (select * from table(proc(cuerpo))) as alias(table(proc(cuerpo)))
```

▷ (*fuente*:**Sequence(***Basic***)**)−>**exists(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador exists definida entre secuencias, salvo por lo detallado a continuación:

- *traducción de la fuente:*

  `select distinct` alias(table(proc(*cuerpo*)))`.val`
  `from (select * from` table(proc(*cuerpo*)))` as`
  alias(table(proc(*cuerpo*)))

▷ (*fuente*:**Sequence(***Basic***)**)−>**one(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos ordenados.

▷ (*fuente*:**Sequence(***Basic***)**)−>**any(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador forAll definida entre conjuntos ordenados.

▷ (*fuente*:**Sequence(***Basic***)**)−>**select(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador select definida entre conjuntos ordenados.

▷ (*fuente*:**Sequence(***Basic***)**)−>**reject(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador reject definida entre conjuntos ordenados.

▷ (*fuente*:**Sequence(***Basic***)**)−>**sortedBy(***var*|*cuerpo***)** Esta operación se traduce de igual manera que el iterador sortedBy definida entre conjuntos ordenados.

Bibliografía

[1] D. Basin, M. Clavel, M. Egea, M. A. García de Dios, C. Dania, G. Ortiz y J. Valdazo: *Model-Driven Development of Security-Aware GUIs for Data-Centric Applications.* En Alessandro Aldini y Roberto Gorrieri (editores): *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volumen 6858 de *LNCS*, páginas 101–124. Springer, 2011, ISBN 978-3-642-23081-3.

[2] M. Blaha y W. Premerlani: *Object-Oriented Modeling and Design for Database Applications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997, ISBN 0-13-123829-9.

[3] G. Booch, J. Rumbaugh y I. Jacobson: *The Unified Modeling Language - User Guide.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999, ISBN 0-201-57168-4.

[4] E. Burger: *Query Infrastructure and OCL within the SAP Project "Modeling Infrastructure"- Studienarbeit.* Informe técnico, Institut für Theoretische Informatik - Technische Universität Karlsruhe, Germany, 2006.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer y C. Talcott: *The Maude System*, 2008. `http://maude.cs.uiuc.edu`.

[6] M. Clavel y M. Egea: *ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams.* En M. Johnson y V. Vene (editores): *AMAST*, volumen 4019 de *LNCS*, páginas 368–373. Springer, 2006, ISBN 3-540-35633-9.

[7] M. Clavel, M. Egea y M. A. García de Dios: *Building and Efficient Component for OCL Evaluation.* En *Proc. of 8th OCL Workshop at the UML/-MoDELS Conference: OCL Concepts and Tools: From Implementation to Evaluation and Comparison*, volumen 15 de *ECEASST*, Tolouse, France, September 2008.

[8] M. Clavel, M. Egea, M. A. García de Dios, C. Dania, G. Ortiz y J. Valdazo: *ActionGUI*, May 2011. `http://www.bm1software.com/actiongui.html`.

[9] E. F. Codd: *A Relational Model of Data for Large Shared Data Banks.* Commun. ACM, 13(6):377–387, 1970.

[10] Digital Equipment Corporation: *Information Technology - Database Language SQL*. Informe técnico, Digital Equipment Corporation, 1992.

[11] B. Demuth y H. Hußmann: *Using UML/OCL Constraints for Relational Database Design*. En R. B. France y B. Rumpe (editores): *Proc. of UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volumen 1723 de *LNCS*, páginas 598–613. Springer, 1999.

[12] B. Demuth, H. Hußmann y S. Loecher: *OCL as a Specification Language for Business Rules in Database Applications*. En *Proc. of UML 2001: The Unified Modeling Language. Modeling languages, Concepts and Tools.*, volumen 2185 de *LNCS*, páginas 104–117, Toronto, Canada, 2001. Springer, ISBN 3-540-42667-1.

[13] M. A. García de Dios, C. Dania, M. Schläpfer, D. A. Basin, M. Clavel y M. Egea: *SSG: A Model-Based Development Environment for Smart, Security-Aware GUIs*. En J. Kramer, J. Bishop, P. T. Devanbu y S. Uchitel (editores): *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, páginas 311–312. ACM, 2010. `http://www.bm1software.com`.

[14] W.J. Dzidek, L.C. Briand y Y. Labiche: *Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java*. En *Proc. of the 4th OCL workshop at MoDELS'05 Conference: Tool Support for OCL and Related Formalisms - Needs and Trends*, volumen 3844 de *LNCS*, páginas 10–19, Montego Bay, Jamaica, October 2005. Springer-Verlag Berlin Heidelberg.

[15] M. Egea: *An executable formal semantics for OCL with Applications to Formal Analysis and Validation*. Tesis de Doctorado, Universidad Complutense de Madrid, Spain, 2008.

[16] M. Egea, C. Dania y M. Clavel: *MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL*. Electronic Communications of the EASST, 36, 2010.

[17] M. Egea, C. Dania y M. Clavel: *The MySQL-OCL Code Generator*, August 2010. `http://www.bm1software.com/mysql-ocl`.

[18] K. Eisenreich: *Varianzanalyse zur Generierung imperativen Codes aus OCL-Ausdrcken-Grosser Beleg*. Informe técnico, Fakultät Informatik - Institut für Software un Multimediatechnik - Technische Universität Dresden - Lehrstuhl Softwaretechnolgie, Germany, 2006.

[19] M. Fowler: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, spanish3 edición, 2003, ISBN 0321193687.

[20] F. Heidenreich, C. Wende y B. Demuth: *A Framework for Generating Query Language Code from OCL Invariants*. En *Proc. of 7th OCL Workshop at the UML/MoDELS Conference: Ocl4All: Modelling Systems with OCL*, volumen 9 de *ECEASST*, Nashville, Tennessee, October 2008.

[21] *Informix*. `http://www-01.ibm.com/software/data/informix`.

[22] A. Kleppe, J. Warmer y W. Bast: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003, ISBN 032119442X.

[23] J. Melton y A. Simon: *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, ISBN 1-55860-245-3.

[24] *MySQL*. `http://www.mysql.org`.

[25] Object Management Group: *Model Driven Architecture Guide v. 1.0.1*. Informe técnico, OMG, 2003. OMG documento disponible en `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[26] Object Management Group: *Object Constraint Language*, Feb 2010. OMG documento disponible en `http://www.omg.org/spec/OCL/2.2`.

[27] Object Management Group: *Unified Modeling Language*, Mar 2011. OMG documento disponible en `http://www.omg.org/spec/UML/2.4`.

[28] *OCL2SQL*, 2001. `http://sourceforge.net/projects/dresden-ocl/files/dresden-ocl/1.1`.

[29] *Object Management Group*. `http://www.omg.org`.

[30] *Oracle*. `http://www.oracle.com`.

[31] *PostgreSQL*. `http://www.postgresql.org`.

[32] R. Ramakrishnan y J. Gehrke: *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, spanish3 edición, 2003, ISBN 0072465638, 9780072465631.

[33] J. Rumbaugh, I. Jacobson y G. Booch (editores): *The Unified Modeling Language - Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999, ISBN 0-201-30998-X.

[34] A. Schmidt: *Untersuchungen zur Abbildung von OCL-Ausdrücken auf SQL*. Tesis de Licenciatura, Institut für Softwaretechnik II - Technische Universität Dresden, Germany, 1998.

[35] *SQLite*. `http://www.sqlite.org`.

[36] J. Warmer y A. Kleppe: *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, spanish2 edición, 2003, ISBN 0321179366.

[37] C. Wilke: *Java Code Generation for Dresden OCL2 for Eclipse- Grosser Beleg*. Informe técnico, Fakultät Informatik - Institut für Software un Multimediatechnik - Technische Universität Dresden - Lehrstuhl Softwaretechnolgie, Germany, 2009.