Técnicas de Detección y Diagnosis de Errores en Consultas de Bases de Datos

Yolanda García Ruiz

Tesis Doctoral

Dirigida por el Doctor Rafael Caballero Roldán

Facultad de Informática
Universidad Complutense de Madrid

Febrero 2014

La presente tesis se enmarca dentro del trabajo desarrollado en el Grupo de Programación Declarativa de la Universidad Complutense de Madrid (grupo 910502 del catálogo de grupos reconocidos por la UCM) y ha contado con el apoyo de los siguientes proyectos de investigación:

- **FAST-STAMP.** Foundations and Applications of declarative Software Technologies (Project TIN2008-06622-C03).
- **PROMESAS-CAM.** Program in Methods for the Development of Dependable, High-Quality, and Secure Software (Project S-0505/TIC/0407)
- **PROMETIDOS-CM.** Programa de Métodos Rigurosos de Desarrollo de Software de la Comunidad de Madrid (Project S-2009/TIC-1465).

Además se ha contado con las ayudas al grupo de investigación mediante las convocatorias de referencias:

- UCM-BSCH-GR58/08-910502
- UCM-BSCH-GR35/10-A-910502

Agradecimientos

Quiero dar las gracias a Rafael Caballero, mi director, por sus bonitas ideas, energía e ilusión y por orientarme y ayudarme en todo lo que he necesitado.

A mis compañeros del Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid, por su amistad y apoyo tanto en el plano profesional como en el personal.

También quiero dar las gracias a mis padres, hermanos y especialmente a mis hijos, los seres más maravillosos del mundo, que han sufrido mis frustraciones y mis "deadlines".

Índice

Resumen

El objetivo de esta tesis es el diseño y desarrollo de técnicas para la detección y diagnosis de errores en el campo de las bases de datos y en particular, en consultas a bases de datos. Para ayudar a la detección de errores se desarrollan técnicas para la generación automática de casos de prueba. Estos casos de prueba no son más que instancias válidas de la base de datos que facilitan al usuario probar de forma sencilla la corrección de los resultados de las consultas. Para realizar el diagnóstico de errores se proponen técnicas relacionadas con la depuración declarativa o algorítmica. Estas técnicas se basan en la exploración de una estructura que representa el cómputo de la consulta a depurar, conteniendo, además de la información del resultado final, información de todos los resultados intermedios. Para localizar la causa del error, se realizan consultas a un oráculo al que se supone conocimiento de los resultados esperados.

Dentro del ámbito de las bases de datos, nos hemos centrado en las bases de datos deductivas, relacionales y semiestructuradas.

Las bases de datos deductivas se basan en la utilización de la Programación Lógica para mantener y consultar los datos. El lenguaje más conocido dentro de este campo es Datalog, cuya sintaxis puede verse como un subconjunto del lenguaje lógico Prolog. La mayor parte de las propuestas para depurar programas Datalog utilizan métodos usados tradicionalmente en depuración imperativa, tratando de explorar el cómputo para encontrar errores. Otros se basan en el análisis de los árboles de prueba asociados a un programa transformado, que resulta difícil de relacionar con el programa original. En esta tesis se propone una herramienta de depuración basada más en la semántica del programa que en el modelo de cómputo, extendiendo y adaptando las ideas genéricas de la depuración declarativa al caso de Datalog.

En el caso de las bases de datos relacionales, el tamaño de la instancia de la base de datos suele ser un obstáculo cuando se desea probar las consultas. En general, la fase de pruebas requiere el previo diseño de casos de prueba (instancias válidas y de tamaño reducido) para su posterior ejecución. Este diseño se realiza, en la mayoría de los casos, de forma manual y se vuelve especialmente difícil en el caso de consultas que involucran gran cantidad de relaciones. Los trabajos relacionados con la generación de casos de prueba para consultas SQL, se centran especialmente en el estudio del nivel de cobertura, más que en la propia generación. En esta tesis tratamos el problema de la generación automática de dichos casos de prueba.

Los casos de prueba permiten evaluar de forma sencilla si el resultado de una

consulta es el esperado. Sin embargo, en el caso de consultas SQL que se basan en vistas, el que una vista produzca un resultado incorrecto no implica necesariamente que sea incorrecta; una vista puede producir un resultado inesperado a causa de la errónea definición de otras vistas de las cuales depende. En estos casos, la falta de herramientas apropiadas hace difícil encontrar el fragmento de código al que achacar el error. Los complejos mecanismos de ejecución de estos lenguajes dificultan la ejecución paso a paso típica de otros paradigmas. Es por ello que en esta tesis aplicamos las técnicas de depuración declarativa como mecanismo para la detección y diagnosis de errores en consultas SQL que involucran varias vistas.

En los últimos tiempos se ha incrementado el interés por los lenguajes de acceso a bases de datos semiestructuradas como XML. En este ámbito se incluyen lenguajes de consulta como XQuery y XPath (subconjunto del anterior). Al igual que sucedía en los casos de las bases de datos relacionales, se trata generalmente de consultas sobre documentos de gran tamaño, lo que dificulta tanto la prueba como la depuración de las consultas. En esta tesis se ha realizado una inmersión del lenguaje de consulta XPath/XQuery en el lenguaje lógico-funcional $\mathcal{TOY}$ desarrollado por nuestro grupo. Esto nos ha permitido utilizar *patrones de orden superior* y las capacidades de *generación y prueba* propias de la programación lógico-funcional para localizar errores en las consultas y obtener casos de prueba en forma de documentos XML.

Palabras clave: depuración declarativa, depuración algorítmica, bases de datos relacionales, SQL, bases de datos deductivas, Datalog, XPath, XQuery, XML, test cases, casos de prueba, programación declarativa, programación lógico-funcional.

1 Introducción

1.1. Presentación y motivación

La fase de *testing* o pruebas es una parte fundamental dentro del ciclo de desarrollo del software [95]. El ejercicio de esta fase posibilita la localización de fallos en la implementación, y su correcto desarrollo es importante para generar software de calidad. El diseño y la generación de casos de prueba junto con la depuración son dos de las actividades más importantes dentro de esta fase.

En el campo de las bases de datos, la ejecución de la fase de pruebas puede convertirse en una tarea ardua para los desarrolladores. En las primeras fases del desarrollo de aplicaciones, la instancia de la base de datos suele estar vacía, y en fases posteriores, cuando se trabaja con instancias de base de datos reales, lo habitual es tener una gran cantidad de relaciones con centenares de filas. El problema se agrava cuando lo que se desea validar es la corrección de las consultas, ya que estas suelen depender de otras consultas previamente definidas, dando lugar a sistemas de vistas correlacionadas.

El diseño de casos de prueba que permitan detectar errores en la definición de las consultas es una actividad fundamental, sin embargo no es trivial debido a la cantidad de factores que han de tenerse en cuenta para que los casos de prueba generados sean eficientes, es decir, que tengan una alta probabilidad de encontrar errores. Una de las motivaciones de esta tesis se basa en que los trabajos relacionados con el diseño y la generación de casos de prueba para consultas de bases de datos, se centran más en el estudio de métodos para medir la calidad de los casos de prueba que en la propia generación. Y aquellos dedicados a la generación de casos de prueba, no consideran el caso de las consultas más habituales, que son las consultas correlacionadas.

En lo relativo a la depuración de las consultas de bases de datos, en general se han venido utilizando técnicas de depuración empleadas habitualmente para la depuración de lenguajes imperativos. Sin embargo, dado el carácter declarativo que tienen la mayor parte de los lenguajes de acceso a bases de datos, estas técnicas no resultan las más apropiadas. Mucho más conveniente para este tipo de lenguajes resulta la *depuración declarativa*, también conocida como *depuración algorítmica* [105]. La depuración declarativa supone una alternativa para el diseño de depuradores de este tipo de lenguajes, la cual fue inicialmente propuesta en el contexto de la programación lógica [111] y se ha extendido a otros paradigmas de programación [92, 39, 40, 41]. Sin embargo, no se ha dedicado mucho esfuerzo al diseño de los fundamentos teóricos

para la depuración declarativa de consultas de bases de datos, lo que ha motivado que esta tesis avance en este sentido.

1.1.1. Casos de prueba

En general, para realizar las tareas de prueba de un sistema, se construye previamente un conjunto de casos de prueba, cuyo propósito es la detección de errores. Dependiendo del tipo de aplicación software a probar, un caso de prueba [12] se compone de aquellos elementos necesarios (valores de entrada, pasos, resultados esperados, etc.) para poder ejecutar la aplicación. El diseño de dichos casos de prueba se puede estudiar desde dos puntos de vista: de caja blanca y de caja negra. En los procesos de prueba de caja negra (black-box testing), los casos de prueba se generan teniendo en cuenta el comportamiento del sistema sin considerar cómo está implementado. Por el contrario, en los procesos de prueba de caja blanca (white-box testing) se diseñan pruebas teniendo en cuenta los detalles procedurales. Este es el tipo de proceso de prueba que se persigue en esta tesis.

Para asegurar la calidad de las pruebas de un sistema es necesario que el diseño del conjunto de casos de prueba pase por la aplicación de ciertos *criterios de medida o de cobertura*. Un criterio de cobertura es una propiedad aplicable a un conjunto de casos de prueba, no a un caso de prueba individual y define las reglas que debe cumplir dicho conjunto con respecto al sistema que se desea probar. Solo si el conjunto de casos de prueba generado cumple con los criterios de cobertura establecidos, se asegura la calidad del mismo.

Entre los criterios de cobertura de caja blanca, podemos destacar algunos de ellos, los clasificados como *criterios estructurales* [12]:

- El *criterio de cobertura de sentencias* requiere que la ejecución del conjunto de casos de prueba garantice que se han recorrido todas las líneas o sentencias del código del programa al menos una vez.

- El *criterio de cobertura de decisiones o de predicado* (*predicate coverage*) requiere que cada posible valor (cierto o falso) que pueda tomar cada una de las decisiones del sistema bajo prueba se dé al menos una vez, donde se entiende que una decisión es un conjunto de condiciones relacionadas mediante operadores lógicos. Por ejemplo, si tenemos un sistema compuesto por una única instrucción como la siguiente:

  ```
  IF (Cond1 AND Cond2) THEN Sentencias;
  ```

  el conjunto de casos de prueba que satisface este criterio consta de dos casos de prueba, uno que evaluará la expresión `(Cond1 AND Cond2)` a cierto y otro que la evaluará a falso. Este criterio tiene el inconveniente de que los casos de prueba generados no servirán para detectar un posible error en la condición `Cond1` (resp. `(Cond2)`) cuando `Cond1` (resp. `(Cond2)`) se evalúa siempre a cierto. Al criterio que solo busca que el valor de cada una de las decisiones del sistema bajo prueba tome el valor cierto (resp. falso) se denomina *criterio de cobertura de decisión positivo* (resp. *criterio de cobertura de decisión negativo*).

- El *criterio cobertura de condiciones* requiere que cada condición atómica de cada decisión tome todos los posibles valores al menos una vez. En este caso, un ejemplo de conjunto (con dos casos de prueba) válido para el ejemplo anterior sería aquel que contiene un caso de prueba que evalúe $\texttt{Cond}_1$ a falso y $\texttt{Cond}_2$ a cierto y otro caso de prueba que evalúe $\texttt{Cond}_1$ a cierto y $\texttt{Cond}_2$ a falso. A pesar de que este criterio es más restrictivo que el anterior no alcanza la cobertura de decisiones, ya que en el ejemplo siempre se evaluaría a falso la decisión y nunca se ejecutaría el bloque de sentencias.

- El *criterio de cobertura de condición decisión* requiere que cada posible valor de cada condición de cada decisión del programa sea producida al menos una vez y que además se produzca cada posible valor de cada decisión. De esta forma, el conjunto de casos de prueba que cumpla este criterio, cumple también los dos anteriores. En este caso y con respecto al ejemplo anterior, un conjunto de casos de prueba tendría dos casos de prueba, uno que evaluase tanto $\texttt{Cond}_1$ como $\texttt{Cond}_2$ a falso y otro que evaluase tanto $\texttt{Cond}_1$ como $\texttt{Cond}_2$ a verdadero.

  Una variante de este criterio, es el llamado *criterio de cobertura modificada de condición decisión* (Modified Condition Decision Coverage, MCDC), el cual requiere además que cada condición afecte independientemente a cada decisión. Un conjunto de casos de prueba que verifica este criterio consta de tres casos de prueba, uno que evalúa las dos condiciones a verdadero, otro que evalúa $\texttt{Cond}_1$ a falso y $\texttt{Cond}_2$ a verdadero de manera que $\texttt{Cond}_1$ afecta independientemente a la decisión y otro que evalúe $\texttt{Cond}_1$ a verdadero y $\texttt{Cond}_2$ a falso.

- El *criterio de cobertura de condiciones múltiples*, utilizado tradicionalmente con lenguajes imperativos, cubre todos los anteriores y requiere que todas y cada una de las posibles combinaciones de valores de las diferentes condiciones dentro de una decisión se den, para cada una de las decisiones. En nuestro ejemplo necesitaríamos cuatro casos de prueba. No obstante, hay que tener en cuenta que este criterio es impracticable cuando se tiene un número elevado de condiciones en una decisión. Si tenemos k condiciones el número de casos de prueba se eleva a 2^k.

Dicho esto, en las tareas de diseño de casos de prueba resulta muy importante la elección del criterio de cobertura utilizado, de forma que el conjunto de casos de prueba sea significativo y del menor tamaño posible para que sea viable.

1.1.2. Depuración declarativa

En el campo de las bases de datos no resulta fácil diseñar depuradores eficientes y manejables para el usuario siguiendo las ideas de los depuradores tradicionales. Esto es debido al carácter declarativo que tienen los lenguajes de definición y consulta de bases de datos, lo que provoca un alejamiento entre su semántica y su mecanismo de cómputo. La depuración declarativa [105] supone una alternativa para el diseño de depuradores de este tipo de lenguajes. En su esquema general, esta técnica se basa en la formulación por parte del depurador declarativo de una serie de preguntas acerca del programa que un oráculo externo (normalmente el usuario) debe contestar. En

base a las respuestas, el depurador es capaz de encontrar la fuente del error. Las ideas básicas de esta técnica son las siguientes:

- El depurador parte de un comportamiento inesperado del programa, lo que se denomina *síntoma inicial*. Los síntomas iniciales de error pueden ser *respuestas incorrectas* o *respuestas perdidas*. Se consideran respuestas incorrectas aquellas que se obtienen de forma inesperada, mientras que se consideran respuestas perdidas aquellas que no se llegan a obtener.

- Se construye un árbol (*árbol de cómputo*) que representa el comportamiento del programa durante el cómputo asociado a dicho síntoma. El árbol de cómputo es una representación finita de un cómputo con resultado erróneo. Cada nodo del árbol se corresponde con el resultado de un cómputo intermedio. La raíz del árbol representa el resultado del cómputo principal, el que ha producido el síntoma inicial. El resultado de cada nodo del árbol depende del resultado de sus nodos hijos.

  El depurador declarativo tiene como objetivo encontrar fragmentos de código erróneos que se corresponden con los nodos del árbol del cómputo. A estos nodos se les conoce con el nombre de *nodos críticos*.

- El depurador recorre el árbol de cómputo haciendo preguntas a un oráculo externo acerca de la validez de los nodos. Si el usuario responde que el nodo es válido, indicando que el resultado asociado al nodo es correcto, el depurador marca dicho nodo como válido. En caso contrario lo marcará como no válido. Sin embargo, es posible que un nodo marcado como no válido se corresponda con un fragmento de código correcto. Esto es debido a que su resultado puede haberse obtenido a partir de otros resultados incorrectos. Por esta razón, el depurador solo marcará como críticos a aquellos nodos marcados como no válidos con hijos válidos. En otras palabras, aquellos nodos con resultado incorrecto a partir de resultados correctos.

Este esquema no garantiza que se vayan a encontrar todos los fragmentos de código erróneos que han intervenido en el cómputo, pero sí se puede garantizar que el depurador encuentra al menos uno de ellos [105], ya que partimos de un árbol finito con raíz errónea (el síntoma inicial) y es fácil probar que:

Teorema 1.1.1 (Completitud débil del esquema general de la depuración declarativa)**.** *Todo árbol de cómputo con raíz errónea tiene al menos un nodo crítico.*

Como indica Lee Naish en [91], este esquema de depuración puede ser aplicado a diferentes lenguajes y paradigmas, dando lugar a diferentes *instancias* del esquema general, todas ellas basadas en las mismas ideas (nodo erróneo, nodo crítico, representación del cómputo, completitud, etc).

En la práctica, este tipo de depuradores tiene el inconveniente de que en el proceso de depuración pueden realizarse un gran número de preguntas al usuario antes de localizar el error y en muchos casos, se plantea el problema adicional de la complejidad de dichas preguntas. En este sentido, en [106] se presentan distintas estrategias de búsqueda con el objetivo de reducir el número de preguntas realizadas al usuario.

Además se proponen técnicas que permiten simplificar las preguntas realizadas al usuario y deducir la validez/no validez de ciertos nodos a partir de la validez/no validez de otros nodos del árbol.

En los apartados siguientes se introducen los paradigmas de bases de datos que se estudian en esta tesis.

1.1.3. Bases de datos deductivas

La programación declarativa es un paradigma de programación basado en el desarrollo de programas que describen el problema que se desea resolver sin indicar cómo ha de resolverse. Esto garantiza la independencia entre la semántica del programa y el mecanismo de cómputo que permite resolverlo. Además, el mecanismo de cómputo es diferente y propio de cada lenguaje, lo que hace que actúe como una caja negra de cara al usuario. Esta independencia hace, en general, que el proceso de depuración de este tipo de lenguajes no sea nada fácil ni intuitivo para el usuario.

Las bases de datos deductivas, también llamadas bases de datos lógicas, se basan en la programación lógica, representándose mediante un programa lógico compuesto por un conjunto de reglas y hechos. Estas bases de datos permiten a los usuarios realizar consultas más expresivas que las habituales sobre bases de datos relacionales, como es el caso de las consultas recursivas. Además es posible deducir relaciones indirectas de los datos almacenados en la base de datos a través de reglas de inferencia.

Datalog [96] es un lenguaje declarativo basado en el modelo relacional que posibilita la definición y el acceso a bases de datos deductivas. El poder expresivo de los programas Datalog sin recursión es equivalente a las expresiones que utilizan las operaciones básicas del álgebra relacional. Las relaciones de una base de datos Datalog se expresan mediante un lenguaje lógico, al igual que las consultas a la base de datos. Puede considerarse como un subconjunto del lenguaje lógico Prolog pero con un mecanismo de cómputo diferente. Mientras que el mecanismo de cómputo de Prolog está basado en SLD-resolución [83], el modelo de cómputo de Datalog está basado en la implementación de técnicas más complicadas tales como *magic sets* [20, 17] o *tabling* [57]. En el caso de Datalog, la terminación de los cómputos está garantizada, ya que no permite el uso de símbolos de función y el uso de la negación está limitada a la negación conocida como negación estratificada [14]. Las relaciones y las consultas pueden ser consideradas desde el punto de vista de la teoría de modelos, donde las relaciones son consideradas conjuntos y las consultas operaciones sobre esos conjuntos.

En la figura 1.1 mostramos un ejemplo de programa Datalog que define una base de datos con información de relaciones familiares. Si deseamos conocer los antepasados de `fred`, basta con escribir la consulta Datalog:

```
antepasado(X,fred).
```

Cuyo resultado es el siguiente conjunto de tuplas:

```
padre(tom,amy).
padre(jack,fred).
padre(fred,carolIII).
madre(amy,fred).

progenitor(X,Y) :- padre(X,Y).
progenitor(X,Y) :- madre(X,Y).

antepasado(X,Y) :- antepasado(Z,Y), progenitor(X,Z).
antepasado(X,Y) :- progenitor(X,Y).
```

Figura 1.1: Ejemplo de base de datos deductiva [101].

```
{
  antepasado(amy,fred),
  antepasado(jack,fred),
  antepasado(tom,fred)
}
```

El programa lógico de la figura 1.1 es también correcto en Prolog, sin embargo el cómputo en Prolog del objetivo `antepasado(X,fred)` es no terminante.

En cuanto a la depuración de consultas Datalog, existen muy pocas herramientas y se basan en técnicas imperativas, analizando la traza seguida por el mecanismo de cómputo para encontrar los errores. En [99] se presenta un trabajo relacionado con depuración declarativa de programas Datalog. Aquí el resultado de una consulta no se trata como un conjunto, sino como un conjunto de resultados individuales. Usa una variante de SLD resolución para localizar errores en el programa analizando tantos árboles como resultados individuales obtenidos para una consulta, lo que puede llegar a ser impracticable. En el paradigma ASP (*answer set programming*) [18] existen varias propuestas para la detección de errores. En [110] se propone una técnica para detectar *conflict sets* y se explica cómo puede extenderse para localizar respuestas perdidas. En [28] se propone una técnica para transformar unos programas en otros equivalentes que incluyen información que será útil en el momento de la depuración. En ambos casos se trata de técnicas muy orientadas a las características particulares de programas ASP y que son de muy poca utilidad en el caso de programas Datalog.

En esta tesis presentamos un nuevo enfoque para la depuración de programas Datalog. Aunque se basa en las ideas de la depuración declarativa, se trata el conjunto de valores de la respuesta de una consulta como una entidad simple. Esto nos permite simplificar el proceso de depuración. Introducimos el concepto de *grafo de cómputo* como una estructura apropiada para representar el mecanismo de cómputo de Datalog. Esto supone una novedad con respecto a otros depuradores declarativos (Prolog [105],

Java [39] y $\mathcal{TOY}$ [30]) los cuales se basan en árboles de cómputo.

1.1.4. Bases de datos relacionales

SQL (acrónimo de Structured Query Language) [54] es un lenguaje de consulta estructurado de acceso a bases de datos relacionales. Es considerado como el lenguaje estándar para la realización de operaciones de recuperación y la manipulación de información en el modelo relacional. A pesar de ser un lenguaje esencialmente declarativo, SQL ofrece muchas posibilidades de tipo práctico (agregados, lógica trivaluada con valores NULL, orden) que dificultan la formulación de una semántica declarativa clara. Por ello, en la literatura sobre el tema se han propuesto diversas semánticas dependiendo de las características del lenguaje que se deseara tener en cuenta en cada caso. Una primera aproximación se basa en considerar las relaciones en sentido matemático, es decir como subconjuntos de los productos cartesianos de los valores base. Semánticas inspiradas en este concepto son el álgebra relacional [52] y el cálculo de tuplas [51, 85]. Aunque muy sencillas en su definición, estas semánticas no consideran la repetición de valores, habituales en las bases de datos relacionales prácticas y esenciales para la definición de operaciones de agregación en SQL. Por ello se han definido otras semánticas basadas en el concepto de multiconjunto, que proporcionan notaciones más precisas para representar consultas escritas en SQL. Entre estas semánticas se encuentra el Algebra Relacional Extendida (ERA) [71, 66]. ERA permite tratar la mayoría de las características de SQL, como son las vistas, funciones de agregación, filas duplicadas, etc.

La naturaleza declarativa y el alto nivel de abstracción de SQL permiten al usuario definir fácilmente complejas operaciones que podrían requerir cientos de líneas de código en cualquier lenguaje de propósito general. En particular, para consultar la información almacenada en una base de datos relacional, SQL pone a nuestra disposición la instrucción select. En muchos casos, cuando los esquemas de base de datos son complejos y las consultas son complicadas, resulta conveniente acceder a los datos mediante *vistas* en lugar de utilizar una única instrucción select.

En SQL las vistas se crean de forma dinámica proporcionando un nombre, una lista de atributos y una consulta expresada mediante una instrucción select. Mediante las vistas es posible acceder tanto a las tablas de la base de datos, como a otras vistas previamente definidas. Por esta razón, las vistas son consideradas el componente básico de las consultas SQL.

En el caso de software escrito en lenguaje SQL, resulta interesante disponer de técnicas que permitan detectar errores en la definición de las vistas con el mínimo coste en tiempo y esfuerzo. Con este objetivo, en esta tesis se estudian dos de las actividades más importantes dentro del proceso de prueba: la generación automática de casos de prueba y la depuración de vistas SQL.

Generación automática de casos de prueba para vistas SQL

Se trata del diseño y la generación de instancias válidas de la base de datos que permitan probar un conjunto de vistas facilitando la detección de errores. Es conveniente que dichas instancias sean efectivas, es decir que permitan detectar el mayor

número de fallos en su definición, y que sean de pequeño tamaño, de forma que el usuario pueda comprobar fácilmente la corrección de los resultados que generan las consultas que se están probando cuando se aplican a estas instancias. En el proceso de generación, se considera la cobertura de las condiciones que permiten seleccionar las filas devueltas por la consulta (ver sección 3.1.2).

Existen varios trabajos relacionados con la generación de casos de prueba en el contexto de las bases de datos. Entre los generadores comerciales de casos de prueba para bases de datos relacionales se encuentran Benchmark Factory [21] y EMS Data Generator [60]. Estas herramientas facilitan la generación de instancias de bases de datos previa parametrización del usuario. Las instancias son generadas con datos obtenidos de forma aleatoria, de forma incremental o incluso con datos obtenidos de una instancia real. En [49] se presenta una herramienta denominada AGENDA (*A (test) GENerator for Database Aplications*) que incluye un módulo para generar instancias de prueba de la base de datos. Los valores que toman los diferentes atributos de las tablas son propuestos por el usuario y se combinan de forma que la instancia generada cumpla las restricciones de integridad definidas en el esquema. En cualquiera de los tres casos, DataFactory, EMS Data Generator y AGENDA, los casos de prueba generados permiten básicamente evaluar la eficiencia del gestor de base de datos e identificar errores de diseño de la aplicación, pero no están orientados a la detección de errores en la definición de consultas o vistas SQL.

En [25] se propone una técnica llamada *Reverse Query Processing* (RQP) para la generación de instancias de bases de datos. El proceso de generación parte de un esquema de base de datos S_D, una consulta Q y una tabla R, produciendo (si es posible) una instancia D tal que $Q(D) = R$. La instancia así generada cumple las restricciones del esquema de base de datos S_D y las restricciones de integridad. La principal aplicación de esta técnica es la depuración y verificación de programas de bases de datos con código SQL embebido. Para realizar las pruebas de este tipo de aplicaciones es necesario computar todos los posibles estados del programa, a los que se llega, en muchos casos, a través de los resultados de una consulta. Esta técnica se aplica para obtener las instancias de la base de datos que permiten alcanzar los distintos estados de los programas a probar. Señalamos dos diferencias importantes con nuestro trabajo; por un lado, para aplicar la técnica y poder generar la instancia, es necesario conocer el resultado de la consulta. Por otro lado, las instancias generadas no son las más apropiadas cuando se trata de localizar errores en la definición de las consultas.

En lo relativo a la calidad de los casos de prueba generados, en [114] se propone un criterio apropiado para medir la calidad de los casos de prueba para consultas SQL denominado *Full Predicate Coverage* (SQLFpc). Este criterio se basa en el criterio de cobertura modificada de condición decisión y se expresa mediante un conjunto de reglas derivadas de la semántica de la consulta. Cada una de estas reglas se puede representar mediante una consulta escrita en lenguaje SQL. De esta forma, un conjunto de instancias C cumple el criterio SQLFpc si se verifica que el resultado de cada una de las consultas con respecto a alguna instancia del conjunto C contiene al menos una tupla.

En la figura 1.2 se muestra un sencillo ejemplo de consulta SQL y las reglas que definen el criterio de cobertura SQLFpc para dicha consulta. Se puede observar que

Q: **select** client, sales
from orders
where postalcode = '28' and client = 'Paul' ;

R1: **select** client, sales
from orders
where postalcode = '28' and client = 'Paul' ;

R2: **select** client, sales
from orders
where not(postalcode = '28') and client = 'Paul' ;

R3: **select** client, sales
from orders
where postalcode = '28' and not(client = 'Paul') ;

Figura 1.2: Reglas de cobertura que definen el criterio de cobertura SQLFpc para la consulta Q.

orders_tc$_1$

...	postalcode	...	client	...
...	28	...	Paul	...
...	32	...	Paul	...

orders_tc$_2$

...	postalcode	...	client	...
...	28	...	James	...

Figura 1.3: Conjunto de casos de prueba para la consulta del ejemplo 1.2.

para obtener un conjunto de casos de prueba C cumpliendo el criterio de cobertura modificada de condición decisión de la condición de la consulta inicial, basta con obtener conjuntos de casos de prueba C_1, C_2, C_3 cumpliendo el criterio de cobertura de decisión positivo para la condición de las consultas $R1$, $R2$ y $R3$. Entonces, el conjunto C se puede definir como la unión $C_1 \cup C_2 \cup C_3$.

En la figura 1.3 se muestra un conjunto C de casos de prueba que cumple el criterio de cobertura SQLFpc. Cada caso de prueba independiente es una instancia de la relación *orders*. La instancia *orders_tc*$_1$ satisface el criterio de cobertura de decisión (en particular satisface el criterio de cobertura de decisión positivo) para la condición de las consultas $R1$ y $R2$, mientras que el la instancia *orders_tc*$_2$ satisface el criterio de cobertura de decisión positivo para la condición de la consulta $R3$.

Este criterio permite por un lado, minimizar el número de instancias de base de datos que constituyen el conjunto final de casos de prueba, y por otro lado, cubrir

el mayor número de situaciones en las consultas SQL, es decir, alcanzar una mayor cobertura.

En un trabajo posterior [97], se presenta una herramienta para generar casos de prueba que satisfacen por un lado las restricciones del esquema de base de datos y por otro las reglas especificadas por el criterio de cobertura SQLFpc. Partiendo de estas reglas y del esquema de base de datos se modela un problema de satisfacción de restricciones. Por el momento tratan un subconjunto reducido de SQL que no incluyen las consultas agrupadas ni las subconsultas. Tampoco tratan las consultas que contienen expresiones aritméticas. Sin embargo, el denominador común de todas estas herramientas es que ninguna de ellas trata el caso de la generación efectiva de casos de prueba para sistemas de vistas correlacionadas. Para el caso de vistas correlacionadas la situación se complica ya que obtener el conjunto de casos de prueba para una vista no se reduce a obtener casos de prueba para las vistas de las cuales depende. Por ejemplo, para obtener un caso de prueba para la vista V definida como:

create view $V(a)$ **as**
select a
from *V1*
where a **not in** (**select** b **from** *V2*);

siendo *V1* y *V2* vistas, no basta con generar casos de prueba para las vistas *V1* y *V2*. Es necesario asegurar que la vista *V1* es capaz de generar al menos un valor de su atributo a tal que la vista *V2* no es capaz de producirlo en su atributo b.

Entre los trabajos mas cercanos al nuestro, podemos destacar [124], donde se aplica una técnica conocida como generación de casos de prueba basada en restricciones (constraint-based test data generation) [56]. En este trabajo se presenta un generador de casos de prueba para un subconjunto muy básico de SQL. Sin embargo, este trabajo no indica cómo generar de forma automática las restricciones asociadas a una consulta, lo que constituye una diferencia fundamental con nuestro trabajo.

En esta tesis se aborda este tema y se presenta una herramienta capaz de generar de forma automática casos de prueba para consultas escritas en SQL (y en particular vistas). En la figura 1.4 se describe el proceso de generación. Se parte de un esquema de base de datos, un conjunto de vistas correlacionadas y un conjunto de restricciones de integridad. El resultado es una instancia simbólica, con variables lógicas en lugar de valores concretos y un conjunto de restricciones sobre estas variables, lo que constituye un problema de satisfacción de restricciones (CSP [113]). La solución a dicho problema, en el caso de que sea satisfactible, representa un caso de prueba. Aunque la idea de usar restricciones en el proceso de generación de casos de prueba no es nueva [56], en esta tesis se presenta la primera formalización que permite la aplicación de la técnica general al caso de la generación de casos de prueba para conjuntos de vistas SQL.

Depuración de vistas SQL

La ejecución de los casos de prueba previamente generados permiten detectar vistas de la base de datos que producen resultados inesperados. Sin embargo no se debe suponer que la definición de la vista que ha devuelto el resultado incorrecto es

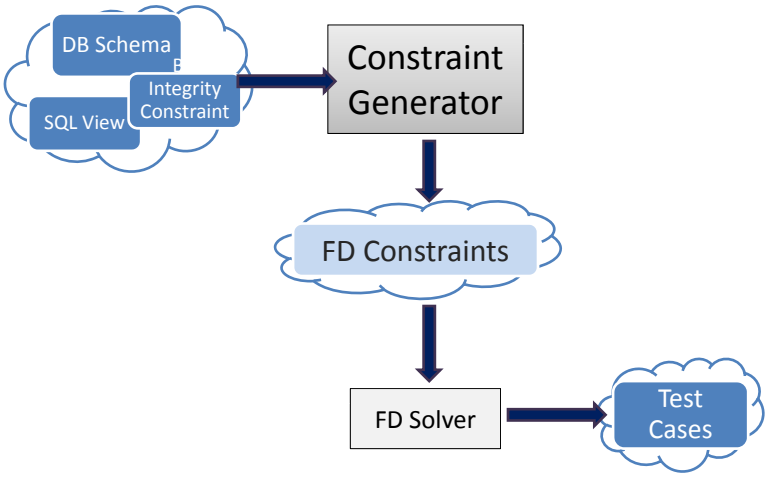


Figura 1.4: Proceso de generación de casos de prueba.

necesariamente errónea. Es posible que la vista esté perfectamente definida, y que la fuente del error se encuentre en la definición de alguna otra vista de la cual depende, o incluso en el contenido de las tablas a las cuales accede.

Existen muy pocas herramientas de depuración para lenguajes de acceso a bases de datos relacionales. La causa principal de esta escasez de herramientas, es que los depuradores utilizados en otros paradigmas de programación, como por ejemplo los depuradores de traza o paso a paso, no son muy útiles en el caso de los lenguajes de acceso a bases de datos, debido a su alto nivel de abstracción. En este tipo de depuradores, la instrucción select se traduce internamente en una secuencia de operaciones de bajo nivel, lo que constituye el *plan de ejecución* de una consulta. Analizar este conjunto de operaciones en busca de errores no es de mucha ayuda, ya que resulta bastante difícil relacionar posteriormente estas operaciones con la consulta original.

En el ámbito comercial encontramos varias herramientas relacionadas con la depuración de aplicaciones con código SQL embebido, [107, 58, 13]. En general estas herramientas permiten trazar y analizar las funciones definidas por el usuario, los procedimientos almacenados y disparadores, pero no resultan muy útiles cuando el objetivo es localizar errores en consultas a la base de datos definidas en función de muchas vistas o tablas intermedias.

En esta tesis se proponen dos técnicas de depuración de vistas SQL basadas en depuración declarativa.

- La primera es una aplicación directa de esta técnica. El punto de partida es un resultado inesperado para el usuario cuando se ejecuta una vista. En una primera fase, el depurador construye un árbol que expresa las dependencias jerárquicas

entre las vistas. La raíz del árbol está asociada a la vista a depurar, las hojas están asociadas a tablas del esquema de base de datos y los nodos intermedios a vistas del esquema de base de datos previamente definidas. Los hijos de un nodo N del árbol se corresponden con las relaciones (tablas o vistas) que aparecen en la consulta que define la vista asociada a dicho nodo. En una segunda fase, el árbol es navegado por el depurador, el cual realiza consultas a un oráculo (normalmente el usuario) acerca del resultado obtenido para la relación (vista o tabla) asociada al nodo. El depurador marca el nodo como válido si el resultado es el correcto y como no válido en otro caso. El proceso de depuración se da por finalizado cuando el depurador encuentra un nodo marcado como no válido con todos sus hijos marcados como válidos. Este nodo *crítico* se corresponde con una vista SQL cuya definición es errónea o con una tabla cuya instancia es incorrecta.

- La segunda técnica refina la anterior, permitiendo diferenciar el tipo de error contenido en el resultado de una vista. El depurador permite al oráculo indicar si se ha producido una *respuesta incorrecta* (el resultado de la vista contiene una fila inesperada) o una *respuesta perdida* (falta una fila en el resultado de la vista). Con esta información y aplicando una técnica similar a *slicing* dinámico [3], el depurador analiza la consulta que define la vista y se concentra solo en aquellas filas producidas por las relaciones intermedias que son relevantes para el error. Aunque esta técnica también está basada en depuración declarativa, no usa árboles de cómputo de forma explícita. En la fase inicial, se representa el árbol de cómputo mediante un conjunto de cláusulas lógicas, dando lugar a un programa lógico. Los átomos en el cuerpo de las cláusulas representan las preguntas que el depurador realiza al oráculo para localizar la fuente del error. El átomo en la cabeza de las cláusulas representa el error que se señalará si se pueden probar todos los átomos del cuerpo. El programa lógico se ejecuta y en cada iteración se selecciona un átomo que dará lugar a una pregunta al oráculo. La información que proporciona el oráculo permite añadir nuevas cláusulas al programa lógico. El proceso se repite hasta que se encuentra una relación errónea.

1.1.5. Bases de datos semiestructuradas

En los últimos años ha ido incrementándose la representación y el intercambio de datos semiestructurados en forma de documentos XML (*eXtensible Markup Language*) [117]. Generalmente se trata de documentos de gran tamaño que almacenan datos de forma jerárquica y en muchos casos con estructuras complejas. Los documentos XML pueden representarse mediante una estructura en forma de árbol cuyos nodos pueden ser de distintos tipos. En la figura 1.5 se muestra un fragmento de un documento XML que contiene información acerca de ciertos productos junto con su representación en forma de árbol.

El lenguaje XPath [50], gracias a su simplicidad y su gran poder expresivo, se ha convertido en el lenguaje más popular para navegar, seleccionar y extraer datos de este tipo de documentos, siendo además un subconjunto de otros lenguajes con mayor poder expresivo, como puede ser XQuery [46, 121]. XQuery es un lenguaje

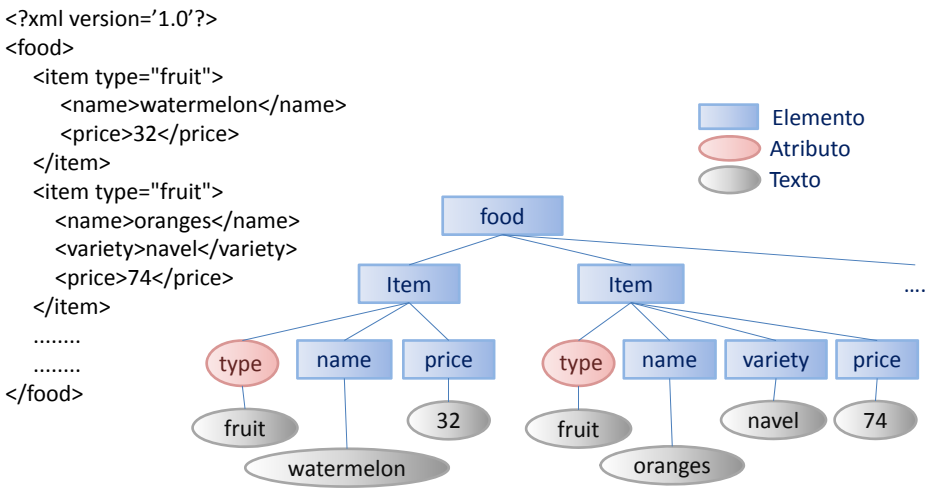


Figura 1.5: Representación parcial del documento "food.xml".

declarativo que permite definir de forma rápida y compacta, consultas o recorridos complejos sobre documentos XML los cuales devuelven todos los nodos que cumplan ciertas condiciones.

Por esta razón se ha dedicado mucho esfuerzo en incluir librerías de XPath/XQuery en muchos lenguajes, permitiendo así escribir consultas XPath/XQuery en sus programas. En particular, en el campo de los lenguajes funcionales, en [86] se implementa el lenguaje XPath en el lenguaje Objective Caml. Existen otras propuestas que definen nuevos lenguajes para procesar documentos XML en lugar de definir implementaciones de XPath/Xquery. Es el caso de los lenguajes XDuce [76] y CDuce [23, 24], que utilizan expresiones regulares y encaje de patrones para procesar documentos XML. En el caso del lenguaje funcional Haskell, HaXML y UUXML [112, 15, 120, 109] permiten el tratamiento de documentos XML. HaXML es una librería que permite representar y manipular los documentos XML como estructuras de datos recursivas en el lenguaje Haskell. Por otro lado, HXQ [61, 62] es un traductor del lenguaje XQuery a código Haskell mediante el uso de plantillas Haskell. HXQ permite almacenar los documentos XML en una base de datos relacional. Las consultas XQuery son traducidas a consultas SQL. Siguiendo esta misma idea, en [26] se presenta el compilador de XQuery llamado *Pathfinder* [27] en el sistema de gestión de bases de datos relacional *MonetDB*.

En el marco de la programación lógica, en la mayor parte de los casos, se proponen nuevos lenguajes de consulta de documentos XML. Algunas implementaciones conocidas del lenguaje Prolog, como por ejemplo SWI-Prolog [122] y Ciao [44], incluyen librerías que permiten tratar documentos XML. El proyecto *Xcerpt* [102, 29] propone un lenguaje de consulta de documentos XML basado en patrones y reglas. El lenguaje

XPathLog, integrado en el sistema *Lopix* [87], es una extensión para XPath al estilo Datalog que usa unificación de variables. $XPath^L$ [98] es un lenguaje lógico basado en reglas que permite procesar documentos XML. Incluye un predicado específico para manejar expresiones XPath en programas Datalog. *FNPath* [104] es un lenguaje de consulta equivalente a XPath que utiliza Prolog para su evaluación, incluyendo backtracking y unificación. Los documentos XML son representados mediante términos Prolog (*Prolog Document Object Model* - DOM). FNPath se basa en los DOM generados para evaluar las consultas XPath. En [5, 4] se presentan dos estrategias para implementar XQuery mediante programación lógica: una estrategia *top-down* y otra estrategia *bottom-up*, esta última en el línea de los programas Datalog.

En el campo de la programación lógico-funcional, en [75] se propone un lenguaje basado en reglas para procesar datos semiestructurados. Este lenguaje está implementado y embebido en el lenguaje lógico-funcional Curry [74].

En el campo de las bases de datos XML, se encuentran trabajos relacionados con el análisis y la optimización de consultas con el propósito de mejorar los tiempos de respuesta [81, 72, 10]. Sin embargo, no existen muchos trabajos relacionados con el desarrollo de técnicas que permitan realizar tareas de prueba y depuración de consultas escritas en XPath/XQuery. En [82] se estudia la generación de documentos XML como casos de prueba a partir del documento XML que se desea probar y su DTD asociada. El objetivo es encontrar errores de tipo semántico en los documentos XML. En [59] se estudia la generación de documentos XML mediante un conjunto de mutantes con respecto a un documento XML original. Sin embargo estos documentos no están orientados a la detección de errores en las consultas. En cuanto a la depuración de consultas XPath/XQuery, *XMLSpy* [11] incluye un depurador XQuery con el que puede probar y corregir las consultas. Este depurador sigue el estilo empleado en depuración de lenguajes imperativos, permitiendo el uso de puntos de parada y la posibilidad de visualizar el contenido de las variables.

En esta tesis estudiamos técnicas que permiten detectar errores en la definición de consultas XPath/XQuery. Con este objetivo, y al igual que hacíamos en el campo de las bases de datos relacionales, nos centramos en las tareas de generación de documentos XML que constituyan casos de prueba para consultas XPath/XQuery así como en la depuración de dichas consultas.

Para ello, en primer lugar se definen en el lenguaje lógico-funcional $\mathcal{TOY}$[1] [84] las estructuras necesarias para poder representar los documentos XML sobre los que se realizarán las consultas. Posteriormente, se presenta una implementación de un subconjunto de los lenguajes XPath/XQuery en dicho lenguaje. La propuesta es puramente declarativa lo que permite probar fácilmente la corrección y completitud con respecto a la semántica de XQuery [22].

La característica del lenguaje *generación y prueba* permite generar documentos XML como casos de prueba para consultas XPath/XQuery. Por otro lado, al ser $\mathcal{TOY}$ un lenguaje *no-determinista*, es posible definir de forma sencilla un marco para XPath que permita realizar la traza del cómputo de cada una de las respuestas individuales, lo que resulta muy útil en el caso de respuestas erróneas. Los *patrones*

[1] Las características específicas del lenguaje lógico-funcional $\mathcal{TOY}$ encajan con la naturaleza declarativa de los lenguajes XPath y XQuery

de orden superior de $\mathcal{TOY}$ permiten considerar las expresiones XPath como código ejecutable (cuando se aplican a un documento XML) o como estructuras de datos cuando se las considera como patrones de orden superior. El uso de *variables lógicas* permite obtener valores de cómputos intermedios. En nuestro caso permiten sugerir valores en forma de documentos XML para el depurador de respuestas perdidas.

1.2. Objetivos

El objetivo fundamental de esta tesis es el estudio, investigación y desarrollo de técnicas de detección de errores en consultas de bases de datos. Este objetivo se descompone en los siguientes puntos:

- Diseño de un marco teórico apropiado para la depuración declarativa de programas Datalog. Desarrollo de un prototipo de herramienta que permita detectar relaciones definidas de forma incorrecta en programas Datalog.
- Estudio e investigación del problema de la generación automática de casos de prueba en el contexto de las bases de datos relacionales. En concreto, de la generación de instancias válidas de la base de datos que permitan probar de manera efectiva consultas SQL y en particular vistas. Desarrollo de un prototipo eficiente capaz de generar dichos casos de prueba.

  Definición de un esquema de depuración para bases de datos relacionales que tenga en cuenta las características específicas de este paradigma, en particular la gran cantidad de tuplas que define un resultado y que puede dificultar mucho la labor a la hora de decidir a validez de un cómputo.
- Estudio e investigación del problema de la generación de casos de prueba y depuración en el contexto de las bases de datos semiestructuradas, y en particular para consultas XPath y XQuery sobre documentos XML.

1.3. Contribuciones principales

Las contribuciones principales de esta tesis se pueden dividirse en varias líneas:

Bases de datos deductivas.

En el capítulo 2 y en las publicaciones [31](A.1) y [32](A.2), presentamos una técnica para depurar programas Datalog basada en depuración declarativa, extendiendo y adaptando las ideas propuestas por L. Naish [91]. Introducimos el concepto de *grafo de cómputo* como una estructura apropiada para representar el mecanismo de cómputo de Datalog. Esto supone una novedad con respecto a otros depuradores declarativos (Prolog [105], Java [39] y $\mathcal{TOY}$ [30]) los cuales se basan en árboles de cómputo. Los resultados teóricos permiten el tratamiento tanto de respuestas perdidas (filas que se esperan en el resultado de una consulta pero que no aparecen) como de respuestas incorrectas (filas que aparecen en el resultado de una consulta pero que no se

esperaban). Los resultados teóricos desarrollados sirven como base para el desarrollo de un prototipo de herramienta que permite detectar relaciones definidas de forma incorrecta en programas Datalog. Las tareas desarrolladas se detallan como sigue:

- Proponemos el uso de grafos como una estructura finita adecuada para modelar los cómputos, ya que la representación en árbol utilizada habitualmente en Depuración Declarativa no sirve para el caso de Datalog. Este grafo es utilizado por el depurador declarativo para detectar relaciones definidas de forma incorrecta. Definimos formalmente el concepto de *relación incorrecta* y los tipos de errores que pueden encontrarse en un programa Datalog. Diseñamos un algoritmo que permite detectar relaciones incorrectas, probando la corrección y completitud de este método con respecto a la semántica operacional de Datalog.

- En base a los resultados anteriores, desarrollamos un prototipo de depurador declarativo de Datalog en el sistema de Bases de Datos deductivas DES (Datalog Educational System) [101]. El depurador parte de una respuesta inesperada, tanto errónea como incompleta, para después generar el grafo de cómputo y recorrer dicho grafo con ayuda del usuario, al que se van haciendo preguntas acerca de la validez de los nodos. Tras su finalización, el depurador puede señalar dos tipos de errores como causas del resultado inesperado: o bien encuentra una relación definida incorrectamente o encuentra un conjunto de relaciones mutuamente dependientes, donde al menos una de ellas es definida de forma incorrecta. Es este segundo tipo de error, que no se había estudiado hasta la fecha, el que exige el empleo de grafos en lugar de árboles.

Bases de datos relacionales.

En el caso de las bases de datos relacionales, el tamaño de las tablas suele ser un obstáculo para la fase de pruebas. Por ello hemos dividido esta línea en dos partes. La primera parte está dedicada a la generación de casos de prueba para consultas SQL, presentándose en la sección 3.2 y en la publicación [33](A.3). La segunda parte está dedicada a la depuración de consultas SQL que involucran varias vistas y ha sido estudiada en las publicaciones [36](A.4) y [37](A.7) y se presenta en la sección 3.3. En cuanto a la generación de casos de prueba, el proceso seguido ha sido el siguiente:

- Definimos los conceptos de *caso de prueba positivo* (PTC), *caso de prueba negativo* (NTC) y *caso de prueba positivo-negativo* (PNTC) para vistas SQL. Definimos formalmente las condiciones que se deben verificar para que una instancia de la base de datos sea considerada un PTC mediante un conjunto de fórmulas lógicas. Esto permite generar conjuntos de casos de prueba para consultas SQL verificando el criterio de cobertura de predicado.

- Definimos un algoritmo que permite reducir el problema de la generación automática de casos de prueba a un problema de satisfactibilidad de restricciones (CSP). Definimos el conjunto de variables que constituyen el problema de decisión y posteriormente traducimos el conjunto de fórmulas lógicas previamente

creado a un lenguaje de restricciones. Probamos la corrección y completitud (débil) con respecto a la semántica del Álgebra Relacional Extendida.

- Presentamos un prototipo inicial de generador de casos de prueba para consultas SQL en el sistema de Bases de Datos deductivas DES (Datalog Educational System) [101]. Este prototipo utiliza el resolutor de restricciones de dominios finitos de SICSTus Prolog para resolver el problema CSP resultante.

En relación con la depuración, las contribuciones principales pueden resumirse en los siguientes puntos:

- En una primera aproximación aplicamos las técnicas de depuración declarativa para la detección de errores en consultas SQL que involucran varias vistas. Definimos una estructura de datos en forma de árbol para representar las relaciones entre las distintas tablas y vistas que intervienen en el cómputo de una consulta. Este árbol será utilizado por el depurador declarativo para detectar tanto vistas SQL definidas de forma errónea, como instancias erróneas de tablas de la base de datos. Definimos formalmente los conceptos de respuesta inesperada, relación errónea y los tipos de errores que pueden darse en una consulta SQL.

- Definimos un algoritmo que permite detectar vistas definidas de forma incorrecta, probando la corrección de la técnica utilizada con respecto al Álgebra Relacional Extendida.

- Posteriormente se refina esta técnica incorporando información acerca de respuestas perdidas e incorrectas, lo que permite reducir el conjunto de filas que debe considerarse para localizar el error, facilitando por tanto la labor al usuario.

Bases de datos semiestructuradas

Hemos estudiado la aplicación de técnicas declarativas al caso de consultas sobre bases de datos semiestructuradas. En particular para consultas XPath y XQuery sobre documentos XML.

- En la sección 4.1 y en la publicación [35](A.5) se definen las estructuras necesarias para representar los documentos XML en el lenguaje lógico-funcional $\mathcal{TOY}$ desarrollado por nuestro grupo. Los documentos XML se representan en $\mathcal{TOY}$ mediante términos. Los constructores básicos de XPath se definen mediante funciones $\mathcal{TOY}$ aplicadas a términos XML.

- En [35](A.5) se utilizan las capacidades de *generación y prueba* propias de la programación lógico-funcional para obtener casos de prueba en forma de documentos XML.

- En [6](A.6), se presenta una implementación de un subconjunto del lenguaje de consulta XQuery usando las características puramente declarativas del lenguaje $\mathcal{TOY}$, lo que permite probar que ambos lenguajes (XPath y XQuery), se han incluido de forma correcta con respecto a la semántica de XQuery presentada en [22].

- En las publicaciones [9](A.8) y [8](A.9) se aplican *patrones de orden superior* [89] para analizar consultas XPath y localizar errores. La utilización de este recurso lógico-funcional ha servido además para aplicar distintas técnicas de optimización a las consultas.

1.4. Estructura de capítulos

Esta tesis sigue el formato de *tesis por publicaciones* según la normativa vigente en la Universidad Complutense de Madrid. El presente capítulo es una introducción e incluye los objetivos de la tesis y un resumen de las contribuciones. Del capítulo 2 al 5 se presenta un resumen de los resultados que aparecen en las publicaciones asociadas a la tesis. En el capítulo 2 se presenta un marco teórico para depurar programas Datalog basado en depuración declarativa. En el capítulo 3 se trata la generación de casos de prueba y la depuración de errores en sistemas de consultas sobre bases de datos relacionales basados en la utilización de vistas SQL. En el capítulo 4 se presenta una extensión del lenguaje lógico-funcional $\mathcal{TOY}$ que permite realizar consultas XPath y XQuery sobre documentos XML. Posteriormente se trata la generación de casos de prueba y depuración para consultas XPath y XQuery sobre documentos XML. En el capítulo 5 presentamos nuestras conclusiones y trabajo futuro.

Las publicaciones asociadas a la tesis se recogen en los apéndices A y B en su formato y longitud original. En el apéndice A se encuentran las publicaciones principales que avalan los resultados de la tesis. En el apéndice B se encuentran versiones extendidas de algunos artículos del apéndice A. Por último, en el apéndice C se encuentra un resumen en inglés.

2 | Bases de datos deductivas: Datalog

En este capítulo se presenta un marco teórico para depurar programas Datalog basado en depuración declarativa. Definimos formalmente el concepto de *relación incorrecta* y los tipos de errores que pueden encontrarse en un programa Datalog. Presentamos el concepto de *grafo de cómputo* como una estructura que permite representar el mecanismo de cómputo de Datalog. Se describe un depurador declarativo basado en el recorrido de dichos grafos, el cual es capaz de encontrar dos tipos de errores: o bien encuentra una relación definida incorrectamente o encuentra un conjunto de relaciones mutuamente dependientes, donde al menos una de ellas es definida de forma incorrecta.

En cuanto a la estructura de este capítulo, en la sección 2.1 se introduce el lenguaje Datalog, su sintaxis, su semántica y los tipos de errores que pueden encontrarse en un programa Datalog. En la sección 2.2 se introducen los conceptos de grafo de cómputo, vértice crítico y circuito crítico. En la sección 2.4 se presenta un prototipo de depurador declarativo para el sistema de bases de datos deductivas DES (Datalog Educational System) que permite detectar relaciones definidas de forma incorrecta en programas Datalog. Finalmente, en la sección 2.5 presentamos nuestras conclusiones.

2.1. Programas Datalog

Datalog, desarrollado originalmente para dar soporte al desarrollo de bases de datos deductivas, apareció por primera vez a finales de los años 70 [65], y fue a finales de los años 80 cuando se hizo más popular [116, 2]. Puede considerarse como el resultado de la combinación de las bases de datos y la programación lógica. Datalog se define en sus comienzos como un lenguaje lógico muy simple pero con un poder expresivo mayor que otros lenguajes relacionales. Con el paso del tiempo ha ido evolucionando y se han definido tanto nuevas extensiones al lenguaje que permiten uso de la negación y restricciones, como nuevos lenguajes basados en Datalog (Axlog [1], Elog [19]).

En esta sección pretendemos aportar la información básica acerca de la sintaxis y de la semántica de los programas Datalog y definimos los diferentes tipos de error

que manejamos en nuestros resultados.

2.1.1. Sintaxis de un programa Datalog

Un *término* es una variable o una constante. Un *átomo* es de la forma $p(t_1, \ldots, t_n)$ donde p es un predicado de aridad n y t_i son términos, para $1 \leq i \leq n$. Un átomo también se puede representar de manera abreviada como $p(\bar{t}_n)$.

Un *literal* es un átomo o un átomo negado. Un literal *positivo* es un átomo, y un literal *negativo* es un átomo negado. Si A es un átomo, $not(A)$ representa el átomo negado. El átomo asociado a un literal L se denota por *atom(L)*.

Una *regla* R es una expresión de la forma $A :- L_1, \ldots, L_n$, donde A es un átomo y L_i son literales. La Regla R representa la fórmula de primer orden $p(\bar{t}_n) \leftarrow L_1 \wedge \ \ldots \ \wedge \ L_m$. Se asume que las variables en el cuerpo de una regla están cuantificadas existencialmente y las variables en la cabeza de una regla están cuantificadas universalmente y que se cumple $vars(A) \subseteq (vars(L_1) \cup \cdots \cup vars(L_n))$. Un *hecho* es una regla con cuerpo vacío.

Las relaciones de una base de datos Datalog se expresan mediante un conjunto finito de hechos y reglas, constituyendo así un programa Datalog. En un programa Datalog P, una *relación* de la base de datos se define como el conjunto de reglas en el programa P con el mismo símbolo de predicado y con la misma aridad. Una *consulta* es un literal (un átomo o un átomo negado). El significado, también llamado *respuesta*, de una consulta es el conjunto de instancias básicas del literal que pueden ser probadas con respecto al programa (ver sección 2.1.2 para una definición formal).

Consideramos en esta tesis programas Datalog como programas lógicos con recursión [14, 115]. Datalog es sintácticamente un subconjunto de Prolog puro (sin características no declarativas), con algunas diferencias. Por un lado, Datalog no permite el uso de símbolos de función como argumentos de un átomo, lo que garantiza la terminación de los cómputos. Por otro lado, y para garantizar una semántica clara, consideramos programas Datalog con negación estratificada [47], una forma de negación introducida en el contexto de las bases de datos.

La figura 2.1 muestra un ejemplo de programa Datalog. Este programa define dos relaciones. La relación *star* está definida por un único hecho que indica que el cuerpo *sun* es una estrella. La relación *orbits* está definida mediante dos hechos y una regla que establece el cierre transitivo de la relación. La relación *orbits(X,Y)* establece la relación entre los cuerpos X e Y, indicando que X gira alrededor de Y. Para conocer los cuerpos que giran alrededor de *sun*, escribimos la consulta *orbits(X,sun)*, cuya respuesta es el conjunto de instancias {*orbits(earth,sun), orbits(moon,sun)*}.

2.1.2. Semántica de un programa Datalog

La semántica de un programa Datalog sigue la teoría de modelos [48], considerando *interpretaciones de Herbrand* y *modelos de Herbrand*, es decir, interpretaciones de Herbrand que hacen que toda instancia de Herbrand de las reglas del programa sean ciertas.

Una *instancia* de una fórmula F es el resultado de aplicar una sustitución θ a F. Usamos la notación $F\theta$ para representar instancias y *Subst* para denotar el conjunto

```
star(sun).
orbits(earth, sun).
orbits(moon, earth).
orbits(X,Y):- orbits(X,Z), orbits(Z,Y).
```

Figura 2.1: Programa Datalog.

de todas las posibles sustituciones.

Dada una interpretación de Herbrand I del programa Datalog P, la notación $I \models F$ [14] indica que la fórmula F es cierta en I. El *significado de una consulta* Q con respecto a la interpretación I, denotado como Q_I, es el conjunto de instancias básicas $Q\theta$ tal que $I \models Q\theta$, es decir:

$$Q_I = \{Q\theta \mid Q\theta \in I \text{ con } \theta \in Subst\}$$

En programación lógica sin negación es posible calcular el *menor modelo de Herbrand* para todo programa P. En el caso de programas lógicos con negación se habla de *modelo estándar* [14] y *modelo estable* [67]. El concepto de *modelo estándar* se aplica a programas lógicos con negación estratificada, como es el caso de Datalog, mientras que el concepto de *modelo estable* se aplica tanto a programas estratificados como no estratificados. En nuestro contexto, y dado que la mayoría de los sistemas que implementan Datalog se limitan a programas estratificados, es posible asegurar la existencia del modelo estándar, el cual denotamos como $\mathcal{M}$. $\mathcal{M}$ es el menor modelo de Herbrand de un programa Datalog P. Dado que en los programas Datalog no se permite el uso de símbolos de función, se puede asegurar que $\mathcal{M}$ es finito y puede ser calculado por los sistemas Datalog.

Los sistemas de bases de datos deductivas, y en particular el sistema DES [101], son capaces de calcular el conjunto de valores $Q_{\mathcal{M}}$ para toda consulta Q. Este conjunto de valores constituye la *respuesta* de la consulta Q. Esto no se puede garantizar en el contexto general de los lenguajes lógicos.

```
p(X) :- q(X).
q(X) :- p(X).
```

Figura 2.2: Programa válido en Prolog y en Datalog.

El programa de la figura 2.2 es válido tanto en Prolog como en Datalog. Sin embargo, el resultado de la consulta (el objetivo) `p(X)` es diferente en Prolog y Datalog. En Prolog, el cómputo del objetivo `p(X)` es no terminante. Sin embargo, en Datalog tiene éxito con respuesta {}, lo que significa que no se puede deducir del programa ninguna instancia de `p(X)`.

2.1.3. Síntomas y errores

La *interpretación pretendida* $\mathcal{I}$ de un programa P es el modelo de Herbrand que el usuario tiene en mente para el programa. Consecuentemente, la *respuesta pretendida* de una consulta Q se define como:

$$Q_{\mathcal{I}} = \{Q\theta \mid Q\theta \in \mathcal{I} \text{ con } \theta \in Subst\}$$

Decimos que un programa está *bien definido* si $\mathcal{M} = \mathcal{I}$. Decimos que el programa es *erróneo* si $\mathcal{M} \neq \mathcal{I}$.

La depuración declarativa de programas Datalog se basa en la comparación entre la interpretación pretendida del programa y el modelo estándar calculado por el sistema. El síntoma de error es una respuesta inesperada para una consulta. Decimos que $Q_{\mathcal{M}}$ es una *respuesta inesperada* para la consulta Q si $Q_{\mathcal{M}} \neq Q_{\mathcal{I}}$. Una respuesta inesperada puede ser de dos tipos:

- $Q_{\mathcal{M}}$ es una *respuesta errónea* si existe $Q\theta \in Q_{\mathcal{M}}$ tal que $Q\theta \notin Q_{\mathcal{I}}$. En este caso, $Q\theta$ es una *instancia errónea* de la consulta Q.

- $Q_{\mathcal{M}}$ es una *respuesta incompleta* si existe $Q\theta \in Q_{\mathcal{I}}$ tal que $Q\theta \notin Q_{\mathcal{M}}$. En este caso, $Q\theta$ es una *instancia perdida* de la consulta Q.

El programa de la figura 2.3 amplía el programa de la figura 2.1 con dos nuevas relaciones: *intermediate* y *planet*.

```
star(sun).
orbits(earth, sun).
orbits(moon, earth).
orbits(X,Y):- orbits(X,Z), orbits(Z,Y).
intermediate(X,Y) :- orbits(X,Y), orbits(Z,Y).
planet(X)         :- orbits(X,Y), star(Y),
                     not(intermediate(X,Y)).
```

Figura 2.3: Programa Datalog erróneo.

La relación *intermediate* está definida en función de la relación *orbits* y pretende asociar dos cuerpos `X` e `Y` si existe un cuerpo intermedio entre ellos. Esta relación está mal definida ya que la variable `Y` del primer átomo del lado derecho de su regla debería ser `Z`. La relación *planet* permite definir un planeta como un cuerpo `X` que orbita sobre una estrella y no hay ningún cuerpo intermedio entre la estrella y él mismo.

Si suponemos que la instancia `planet(earth)` está en la interpretación pretendida $\mathcal{I}$, el resultado de la consulta $Q =$ `planet(X)` es una respuesta incompleta ya que $Q_{\mathcal{M}} = \{\ \}$.

En la siguiente sección veremos cómo estos errores pueden ser detectados usando depuración declarativa.

Una respuesta inesperada puede ser errónea e incompleta al mismo tiempo. La siguiente proposición enuncia que si una consulta Q produce una respuesta inesperada, su negación ($\neg Q$) también produce una respuesta inesperada (y viceversa).

Proposición 2.1.1. *Sea P un programa con al menos una constante, $\mathcal{I}$ su modelo pretendido y Q una consulta asociada a un átomo positivo. Entonces, $Q_{\mathcal{M}}$ es una respuesta incompleta de Q si y solo si $(\neg Q)_{\mathcal{M}}$ es una respuesta errónea de $\neg Q$, y $Q_{\mathcal{M}}$ es una respuesta errónea de Q si y solo si $(\neg Q)_{\mathcal{M}}$ es una respuesta incompleta de $\neg Q$.*

A continuación introducimos los conceptos de *relación errónea* y *relación incompleta*. Estos dos tipos de errores ya se consideraban en depuración declarativa de programas lógicos (*predicado erróneo* y *predicado incompleto*).

Definición 2.1.2. *Sea P un programa Datalog e $\mathcal{I}$ la interpretación pretendida del programa P. Decimos que $p \in P$ es una **relación errónea** con respecto a $\mathcal{I}$ si existe una variante de regla $p(\bar{t}_n) :- l_1, \ldots, l_m$ en el programa P y una sustitución θ tal que $\mathcal{I} \models l_i\theta$, $i = 1 \ldots m$ e $\mathcal{I} \not\models p(\bar{t}_n)\theta$.*

Definición 2.1.3. *Sea P un programa Datalog e $\mathcal{I}$ la interpretación pretendida del programa P. Decimos que $p \in P$ es una **relación incompleta** con respecto a $\mathcal{I}$ si existe un átomo $p(\overline{s}_n)$ tal que $\mathcal{I} \models p(\overline{s}_n)$ y para toda variante de una regla de p de la forma $p(\overline{s}_n) :- l_1, \ldots, l_m$ se cumple $\mathcal{I} \not\models l_i$ para algún $1 \leq i \leq m$.*

Para ilustrar la definición de relación incompleta, consideremos de nuevo el programa Datalog de la figura 2.2. Supongamos que la interpretación pretendida es $\mathcal{I} = \{p(a)\}$. La respuesta calculada por el sistema para la consulta `p(X)` es $\{\}$, la cual es una respuesta incompleta, siendo `p(a)` una instancia perdida. La única regla que define la relación p no puede producir el valor `p(a)` ya que $\mathcal{I} \not\models q(a)$. Por tanto, podemos decir que la relación p es incompleta con respecto a $\mathcal{I}$.

En el contexto de los programas Datalog con recursión es necesario considerar otra posible causa de error denominada *conjunto incompleto de relaciones*. Este concepto depende de otra definición, la de *conjunto no cubierto de átomos*.

Definición 2.1.4. *Sea P un programa Datalog e $\mathcal{I}$ la interpretación pretendida del programa P. Sea U un conjunto de átomos tal que $U \subseteq \mathcal{I}$. Decimos que U es un **conjunto no cubierto de átomos** si para cada regla*

$$p(\bar{t}_n) :- l_1, \ldots, l_m$$

del programa P y cada sustitución θ tal que :

- $p(\bar{t}_n)\theta \in U$,
- $\mathcal{I} \models l_i\theta$ *for* $i = 1 \ldots m$

existe algún literal $l_j\theta \in U$, $1 \leq j \leq m$, con l_j un literal positivo.

Los átomos en el conjunto U están en $\mathcal{I}$, pero no pueden ser producidos porque dependen circularmente unos de otros. La definición de *conjunto incompleto de relaciones* generaliza la idea de relación incompleta:

Definición 2.1.5. *Sea P un programa Datalog y S un conjunto de relaciones definidas en el programa P. Decimos que S es un* ***conjunto incompleto de relaciones*** *en P si y solo si existe un conjunto U de átomos no cubiertos tal que para cada relación $p \in S$, $p(\bar{t}_n) \in U$ para ciertos $t_1, \ldots, t_n$.*

Para ilustrar la definición anterior, supongamos que la interpretación pretendida del programa Datalog de la figura 2.2 es $\mathcal{I} = \{p(a), q(a)\}$. La respuesta calculada por el sistema para la consulta `p(X)` es $\{\}$, la cual es una respuesta incompleta, siendo `p(a)` una instancia perdida. Sin embargo, ninguna de las relaciones son incompletas ya que sus reglas pueden producir los valores `p(a)`,`q(a)` por medio de la instancia dada por la sustitución $\theta = \{X \mapsto a\}$. En cambio, el conjunto $U = \{p(a), q(a)\}$ es un conjunto no cubierto de átomos y $S = \{p, q\}$ es un conjunto incompleto de relaciones.

Este tipo de error no se ha considerado antes en el contexto de la depuración de programas Datalog, sin embargo es necesario considerarlo para realizar un diagnóstico correcto de los errores que se pueden producir en este tipo de programas.

Definición 2.1.6. *Decimos que una relación es* ***crítica*** *si es errónea, incompleta o es miembro de un conjunto incompleto de relaciones. Decimos que una relación está* ***bien definida*** *en otro caso.*

Dado el uso de la negación y como consecuencia de la proposición 2.1.1, una respuesta errónea no se corresponde siempre con una relación errónea. Por ejemplo, sea el siguiente programa Datalog:

```
p(X) :-  r(X), not(q(X)).
r(a).
```

Supongamos que la interpretación pretendida del programa es $\mathcal{I} = \{q(a), r(a)\}$. La relación q es incompleta ya que falta el hecho `q(a)`. Como consecuencia, la consulta `p(X)` produce la respuesta errónea `{p(a)}`. Sin embargo, la causa del error no es una relación errónea en el programa, sino una relación incompleta.

2.2. Grafos de cómputo

En esta sección definimos una estructura que permite representar de forma adecuada el mecanismo de cómputo en Datalog. Normalmente, en lenguajes de programación lógica, los cómputos se representan mediante alguna estructura en forma de árbol, como por ejemplo los árboles SLD [83] en el caso de Prolog. En el caso de Datalog, no es posible representar los cómputos con una estructura de árbol ya que el cómputo de los programas recursivos en Datalog tienen un tratamiento diferente.

En la figura 2.4 se presenta un programa lógico correcto tanto en Prolog como en Datalog. En Prolog, el árbol SLD que representa el cómputo del objetivo `p(X)` contiene una rama infinita, representando un cómputo no terminante. Sin embargo,

```
p(a).
r(b).
p(X) :- q(X),r(X).
q(X) :- p(X).
```

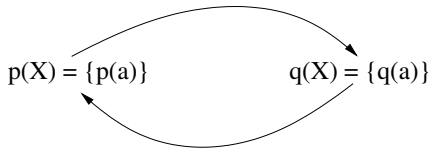


Figura 2.4: Representación del cómputo en Datalog del objetivo `p(X)` con respecto al programa de la izquierda.

en Datalog el mismo objetivo es terminante devolviendo como resultado la respuesta `{p(a)}`. El mecanismo de cómputo de Datalog detecta la repetición del subobjetivo `p(X)`, evitando un bucle infinito. En Datalog este tipo de situaciones pueden ser representadas de forma finita mediante una estructura de grafo, como se muestra en el lado derecho de la figura. Este grafo contiene dos vértices que representan los subobjetivos que han aparecido en el cómputo del objetivo inicial `p(X)`. Cada uno de los vértices aparece en el grafo con su correspondiente resultado. Esta estructura nos permite indicar que los objetivos `p(X)` y `q(X)` son mutuamente dependientes.

El grafo que representa el cómputo de una consulta Datalog es diferente del *árbol de dependencias sintácticas* [123] de un programa Datalog. El árbol de dependencias sintácticas permite representar las conexiones entre las relaciones definidas en el programa desde un punto de vista sintáctico. El grafo de dependencias del programa de la figura 2.4 incluiría un vértice asociado a la relación `r` conectado con el vértice asociado a la relación `p`. Sin embargo, en el grafo de cómputo no aparece ningún vértice asociado a la relación `r` ya que `r` no interviene en el cómputo del objetivo `p(X)`.

El *grafo de cómputo (CG)* de una consulta Q con respecto a un programa P es un grafo dirigido $G = (V, E)$ tal que cada vértice en V es de la forma $[Q' = Q'_{\mathcal{M}}]$, donde Q' es una de las subconsultas que aparecen durante el cómputo de la consulta principal Q y $Q'_{\mathcal{M}}$ es la respuesta computada por el sistema de la subconsulta Q'. El proceso de construcción del grafo de cómputo de una consulta con respecto a un programa P se describe a continuación.

Definición 2.2.1. *Sea P un programa Datalog y Q una consulta de la forma $p(\bar{a}_n)$ o $not(p(\bar{a}_n))$. El* ***grafo de cómputo*** *de Q con respecto al programa P se representa mediante el par (V, E), donde V es un conjunto de vértices y E un conjunto de aristas definidos de la siguiente manera:*

En el proceso de construcción del grafo se usa un conjunto auxiliar A de vértices que deben ser expandidos para completar el grafo.

1. *Inicialización:*
   $V = A = \{p(\bar{a}_n)\}$
   $E = \emptyset$

2. *Mientras $A \neq \emptyset$ hacer:*

a) *Seleccionar un vértice u del conjunto A con consulta $q(\bar{b}_n)$. $A = A \setminus \{u\}$.*

b) *Para cada regla del programa R que define q, $R = (q(\bar{t}_n) :- l_1, \ldots, l_m)$ con $m > 0$, tal que existe $\theta = mgu(\bar{t}_n, \bar{b}_n)$, el depurador crea un conjunto S de nuevos vértices. Inicialmente, $S = \emptyset$. Posteriormente se incluyen nuevos vértices asociados a cada literal l_i, $i = 1 \ldots m$ de la siguiente forma:*

1) Si $i = 1$, $S = S \cup \{atom(l_1)\theta\}$.

2) Si $i > 1$, para cada conjunto de sustituciones $\{\sigma_1, \ldots, \sigma_i\}$ con $dom(\sigma_1 \cdot \ldots \cdot \sigma_{i-1}) \subseteq var(l_1) \cup \cdots \cup var(l_i)$ tal que para cada $1 < j \leq i$:

- *$atom(l_{j-1})(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}) \in S$ y*
- *$l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_j) \in (l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}))_{\mathcal{M}}$*

se incluye un nuevo vértice a S:

$$S = S \cup \{atom(l_i)(\sigma_1 \cdot \ldots \cdot \sigma_i)\}$$

c) *Para cada vértice $v \in S$, comprobar si existe ya un vértice $v' \in V$ tal que v y v' son variantes (iguales salvo renombramiento de variables). Existen dos posibilidades:*

- *Si existe v', entonces, $E = E \cup \{(u, v')\}$. (Si existe el vértice v', simplemente se añade una arista desde el vértice u al vértice v'.)*
- *En otro caso, $V = V \cup \{v\}$, $A = A \cup \{v\}$, y $E = E \cup \{(u, v)\}$.*

3. Completar los vértices incluyendo la respuesta calculada por el sistema $Q_{\mathcal{M}}$ para cada subconsulta Q.

Los valores $Q_{\mathcal{M}}$ incluidos en el paso 3[1] se corresponden con la respuesta computada por un sistema Datalog para la consulta Q. De esta forma representamos que el resultado del cómputo de la consulta asociada a un nodo del grafo depende del resultado del cómputo de las subconsultas asociadas a sus nodos hijos.

La terminación del proceso de construcción del grafo está garantizada ya que en nuestro contexto la signatura es finita y el grafo no puede contener vértices repetidos (según indica el paso $2c$).

El grafo de cómputo de una consulta Datalog depende únicamente de la consulta inicial y de las subconsultas que han aparecido en dicho cómputo, por tanto, dicho grafo es finito y no depende del tamaño del programa.

En la figura 2.5 puede verse el grafo de cómputo de la consulta `planet(X)` con respecto al programa de la figura 2.3. En el paso 1 del proceso de construcción se incluye en el grafo el vértice correspondiente al átomo `planet(X)`. A partir de este vértice y su única regla (`planet(X) :- orbits(X,Y), star(Y), not(intermediate(X,Y)).`), se añaden cinco vértices más. El primer vértice se corresponde con el primer literal `orbits(X,Y)`. Los valores de `Y` que hacen cierta la subconsulta `orbits(X,Y)` son `Y=sun`, `Y=earth`. Por tanto se introducen dos nuevos vértices `star(sun)` y `star(earth)` asociados al segundo literal `star`. La subconsulta `star(earth)` produce

[1] La respuesta computada de las subconsultas no influyen en la estructura final del grafo, por lo que se incluyen en el grafo en el último paso.

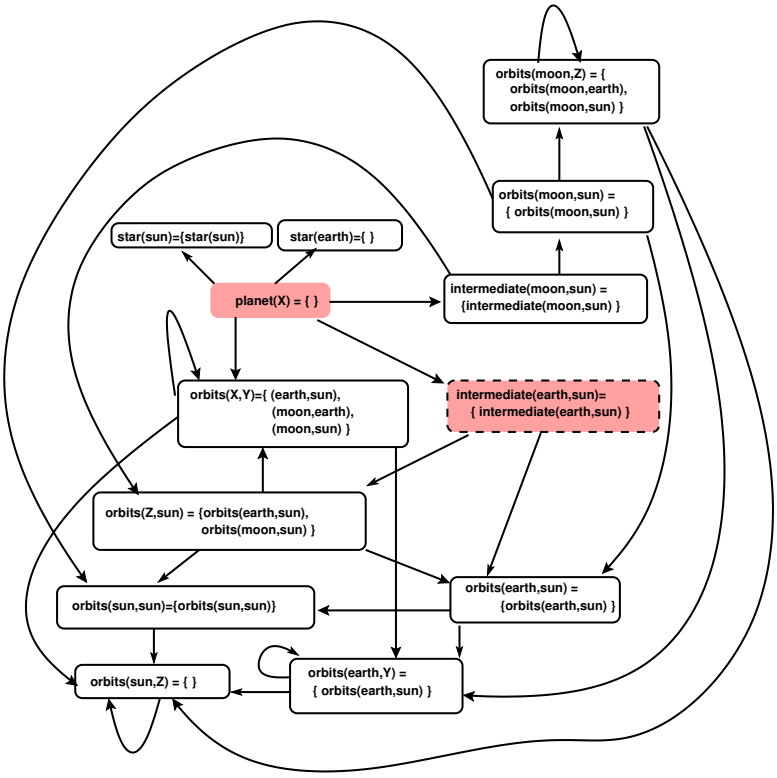


Figura 2.5: Grafo de cómputo de la consulta `planet(X)` con respecto al programa de la figura 2.3.

una respuesta vacía, mientras que `star(sun)` tiene éxito. Por tanto, se añaden dos vértices más correspondientes al último literal de la regla `not(intermediate(X,Y))`, con los valores de `X`, e `Y` que satisfacen los anteriores literales: `intermediate(earth,sun)` y `intermediate(moon,sun)`. Puede observarse en el grafo de la figura 2.5 que las consultas que aparecen en cada uno de los vértices se corresponden con átomos sin negación. Esto hace que las preguntas que realiza el depurador acerca de la validez de los nodos en el proceso de depuración sean más sencillas sin que afecte a los resultados teóricos ya que la validez de un literal positivo implica la validez de su negación y viceversa. Según la proposición 2.1.1, lo único que cambia es el tipo de error. El resto de vértices del grafo se añaden expandiendo cada uno de los sucesores de `planet(X)` y repitiendo el proceso hasta que no se puedan añadir más vértices.

Una vez construido el grafo de cómputo de una consulta, es posible calificar los nodos como válidos o no válidos. Decimos que un vértice es *válido* si la respuesta computada $Q_{\mathcal{M}}$ y la respuesta pretendida $Q_{\mathcal{I}}$ de la consulta Q asociada al nodo

coinciden, y *no válido* en otro caso.

2.3. Vértices críticos y circuitos críticos

En el esquema de depuración declarativa tradicional basado en árboles de cómputo [91], los errores de programa se corresponden con *nodos críticos*. En nuestro contexto, el esquema de depuración está basado en el recorrido de grafos, donde un error en el programa se corresponde con un *vértice crítico* o con un *circuito crítico*.

En teoría de grafos, un *circuito* de un grafo dirigido G es una secuencia de vértices $u_1, u_2, \ldots, u_n$ tal que existe una arista de u_i a u_{i+1} en G para todo $i = 1 \ldots n-1$ y $u_1 = u_n$.

Definición 2.3.1. *Sea $CG = (V, A)$ un grafo de cómputo.*

*a) Definimos **circuito crítico** como un circuito $W = v_1 \ldots v_n$ en CG tal que para todo $1 \leq i \leq n$:*

   1. v_i es un vértice no válido.

   2. Si $(v_i, u) \in A$ y u es no válido, entonces $u \in W$.

*b) Decimos que $v \in V$ es un **vértice crítico** si es no válido y todos sus vértices sucesores son válidos.*

Intuitivamente, un vértice crítico es un vértice no válido con todos sus sucesores válidos, mientras que un circuito crítico es un circuito en el grafo formado por vértices no válidos.

En la figura 2.6(a) se muestra un ejemplo de grafo de cómputo con un vértice crítico (marcado con un aspa), mientras que en la figura 2.6(b) se muestra un ejemplo de grafo de cómputo con un circuito crítico. Los nodos no sombreados representan nodos que no han sido visitados por el depurador. Los nodos etiquetados con `V` representan nodos válidos, mientras que los nodos etiquetados con `NV` representan nodos no válidos.

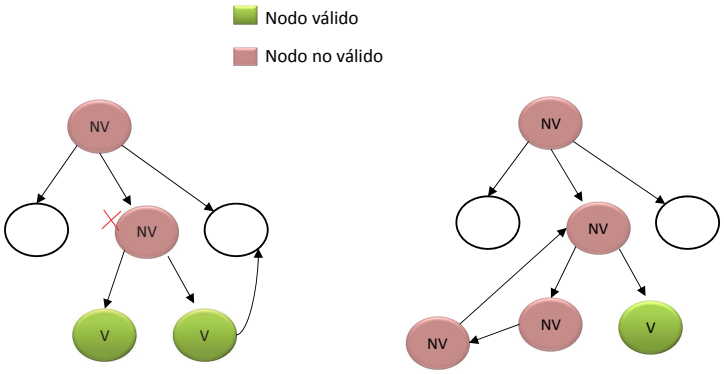
La siguiente proposición enuncia que todo grafo de cómputo con un vértice no válido contiene un vértice crítico o un circuito crítico.

Proposición 2.3.2. *Sea $G = (V, A)$ un grafo de cómputo y $v \in V$ un vértice no válido. Entonces, G contiene al menos un vértice crítico o un circuito crítico.*

Con todo lo anterior, es posible afirmar que si el sistema Datalog produce una respuesta para una consulta Q con respecto a un programa P, de forma que dicha respuesta no es la que el usuario espera, el depurador encuentra un error en el programa P. Las fases del proceso de depuración son las siguientes:

1. El depurador construye el grafo de cómputo CG de la consulta Q con respecto al programa P.

2. El depurador realiza un recorrido del grafo realizando preguntas al usuario acerca de la validez de los vértices.



(a) Vértice crítico

(b) Circuito crítico

Figura 2.6: (a) Representación de un grafo de cómputo con vértice crítico. (b) Representación de un grafo de cómputo con un circuito crítico.

3. El proceso de depuración finaliza cuando el depurador encuentra un vértice crítico o un circuito crítico. Si el depurador encuentra un vértice crítico, es posible afirmar que la relación asociada al vértice es una relación crítica. En el caso de que el depurador encuentre un circuito crítico, es posible afirmar que o bien al menos una de las relaciones del conjunto de relaciones asociadas a los vértices que intervienen en el circuito es crítica, o bien se trata de un conjunto de relaciones incompleto.

En la publicación [32](A.2) se prueba la consistencia de la técnica utilizada y se demuestran los siguientes resultados de corrección y completitud.

Teorema 2.3.3 (Corrección)**.** *Sea P un programa Datalog, Q una consulta y CG el grafo de cómputo de la consulta Q con respecto al programa P. Entonces:*

1. *La relación asociada a un vértice crítico del grafo G es una relación crítica.*

2. *Todo circuito crítico del grafo G contiene o bien un vértice crítico o un conjunto incompleto de relaciones.*

Teorema 2.3.4 (Completitud)**.** *Sea P un programa Datalog, Q una consulta y $Q_{\mathcal{M}}$ la respuesta producida por el sistema tal que $Q_{\mathcal{M}}$ es una respuesta inesperada. Entonces, el grafo de cómputo CG de la consulta Q con respecto al programa P contiene un vértice crítico o un circuito crítico.*

2.4. Implementación y prototipo

Como parte de esta tesis, hemos implementado un prototipo de depurador declarativo siguiendo las ideas teóricas aquí presentadas, el cual se ha incluido en el sistema Datalog DES, disponible en la dirección:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/DD`

En esta página se da la información necesaria para la instalación, ejecución y utilización del depurador.

Nuestro depurador construye el grafo de cómputo de la consulta inicial, la consulta a depurar, cuando el usuario detecta que el sistema ha producido una respuesta inesperada para dicha consulta. Los valores intermedios $[Q = Q_{\mathcal{M}}]$ se almacenan mientras se calcula el resultado de la consulta evitando la repetición de cálculos e incrementando la eficiencia en la construcción del grafo.

El depurador ofrece la posibilidad de depurar a dos niveles dependiendo de las necesidades del usuario y de su conocimiento acerca de la respuesta pretendida: a nivel de relación o a nivel de cláusula. Si el proceso de depuración actúa a nivel de relación, el depurador es capaz de encontrar o bien una relación crítica o bien un conjunto incompleto de relaciones. En otro caso, si actúa a nivel de cláusula, el depurador podría proporcionar información adicional, indicando la regla concreta que origina el error.

```
DES> /debug planet(X) p

Info: Starting debugger...

Is orbits(sun,sun) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(earth,Y) = {orbits(earth,sun)}
                       valid(v)/invalid(n)/abort(a) [v]? v
Is intermediate(earth,sun) = {intermediate(earth,sun)}
                        valid(v)/invalid(n)/abort(a) [v]? n
Is orbits(sun,Y) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(X,sun) = {orbits(earth,sun),orbits(moon,sun)}
                          valid(v)/invalid(n)/abort(a) [v]? v

Error in relation: intermediate/2
Witness query:
      intermediate(earth,sun) = {intermediate(earth,sun)}
```

Figura 2.7: Sesión de depuración de la consulta `planet(X)` con respecto al programa de la figura 2.3.

En la figura 2.7 se muestra un ejemplo de sesión depuración de la consulta

`planet(X)` con respecto al programa de la figura 2.3. El nivel de depuración elegido es nivel de relación. Para ello basta con ejecutar el comando `DES>/debug planet(X) p`. Una vez construido del grafo de cómputo para la consulta `planet(X)`, el depurador realiza un recorrido de dicho grafo realizando preguntas al usuario acerca de la validez de los nodos intermedios. El usuario tiene dos posibles respuestas: `valid` o `invalid`.

La primera pregunta que realiza el depurador es acerca de la relación `orbits`. El depurador pregunta si es correcto que el resultado de la consulta `orbits(sun,sun)` sea el conjunto vacío, es decir si se espera que el objetivo falle. En este caso, el resultado es correcto ya que no se espera que el cuerpo `sun` orbite sobre sí mismo y el usuario responde `valid`. La segunda pregunta que realiza el depurador es si el cuerpo `earth` orbita únicamente alrededor del cuerpo `sun`. La respuesta es correcta según nuestro modelo pretendido y el usuario responde `valid` de nuevo. La respuesta a la tercera pregunta no es correcta ya que el objetivo `intermediate(earth,sun)` debería fallar. Esto es porque según el modelo pretendido, el cuerpo `earth` orbita directamente alrededor del cuerpo `sun` sin cuerpos intermedios. En este caso el usuario responde `invalid`. Las respuestas a las últimas dos preguntas formuladas por el depurador son correctas, dándose por finalizada la sesión de depuración. El depurador concluye que existe un nodo crítico en el grafo de cómputo asociado a la relación `intermediate/2`, la cual es una relación crítica.

Con el objetivo de minimizar el número de preguntas al usuario antes de localizar el error, en cada paso el depurador selecciona un vértice siguiendo una estrategia de recorrido del grafo similar a *divide & query* presentada en [105]. En otros paradigmas, se ha comprobado que esta estrategia requiere una media de $\log_2 n$ preguntas al usuario para localizar el error [30], con n el número de nodos del grafo de cómputo. Nuestros experimentos confirman que esto también se cumple en el caso de grafos sin ciclos, lo que ocurre la mayoría de las veces. En el caso de grafos con ciclos, los resultados siguen esta tendencia aunque es necesario realizar más experimentos.

2.5. Conclusiones

En este capítulo se ha presentado una técnica para depurar programas Datalog basada en programación declarativa. El estudio de las características de Datalog, su semántica y su mecanismo de cómputo nos ha llevado a pensar que las instancias del esquema de depuración declarativa usadas tradicionalmente en programas lógicos, no sirven para este tipo de programas. Por ello, se ha definido una nueva instancia basada en una estructura capaz de representar el mecanismo de cómputo de Datalog: *grafo de cómputo*. En Datalog el cómputo de una consulta es siempre terminante y no es posible representarlo fácilmente mediante una estructura de árbol como los usados tradicionalmente en depuración declarativa. El mecanismo de cómputo de Datalog es capaz de detectar la repetición de subcómputos en el cómputo de una consulta, evitando así las ramas infinitas que pueden darse, por ejemplo, en SLD resolución. Mediante un grafo es posible representar de forma finita (mediante ciclos en el grafo) ciertos cómputos que, en otros sistemas como por ejemplo Prolog, son infinitos. Esto supone una novedad con respecto a otros trabajos relacionados con la depuración declarativa de programas lógicos.

Por otro lado, los errores considerados tradicionalmente en programación lógica y que se corresponden con nodos críticos en el árbol de cómputo, no cubren los diferentes errores que pueden darse en programas Datalog. En Datalog aparece un nuevo tipo de error que se corresponde con lo que denominamos un *circuito crítico* del grafo de cómputo. Por ello resulta necesario definir formalmente los conceptos de *vértice crítico*, *circuito crítico* y los diferentes tipos de errores, como son *relación definida incorrectamente*(relación errónea o incompleta) y *conjunto de relaciones mutuamente dependientes, donde al menos una de ellas es definida de forma incorrecta.* Con respecto al proceso de depuración, se describe un depurador declarativo basado en el recorrido de dichos grafos.

En base al esquema de depuración presentado y a los resultado teóricos obtenidos, se ha desarrollado un prototipo de depurador declarativo de programas Datalog en el sistema de bases de datos deductivas DES. En esta línea, podemos decir que los objetivos de la tesis han quedado cubiertos, tanto por el esquema de depuración presentado como por el prototipo de depurador declarativo desarrollado.

Como trabajo futuro, consideramos la posibilidad de permitir al usuario proporcionar más información acerca del resultado de una consulta. Por ejemplo permitiendo al usuario indicar el motivo por el cual un vértice es no válido (respuesta errónea o incompleta). Por otro lado, consideramos la tarea de desarrollar y comparar diferentes estrategias de recorrido del grafo con el objetivo de minimizar el número de vértices visitados antes de localizar el error.

Publicaciones asociadas

(A.1) **A New Proposal for Debugging Datalog Programs**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2007), volume 216 of *Electronic Notes in Theoretical Computer Science*, pages 79–92, Paris, France, June 25 2007.

→ **Página** 125

(A.2) **A Theoretical Framework for the Declarative Debugging of Datalog Programs**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In 3rd International Workshop on Semantics in Data and Knowledge Bases (SDKB 2008), volume 4925 of *Lecture Notes in Computer Science*, pages 143–159, Nantes, France, March 29, 2008. Springer-Verlag.

→ **Página** 139

3 | Bases de datos relacionales: SQL

Este capítulo discute la detección y corrección de errores en sistemas de consultas sobre bases de datos relacionales basados en la utilización de vistas SQL. En primer lugar tratamos la generación automática de casos de prueba. Para ello presentamos una técnica que reduce el problema a una instancia del problema CSP [113], que es resuelto dentro del marco de la programación lógica con restricciones. En la segunda parte del capítulo, se introduce una técnica de detección de errores basada en depuración declarativa [105] aplicada a vistas SQL. El punto de partida del proceso de depuración es un resultado inesperado producido en la ejecución de una vista. El depurador permite al usuario especificar el tipo de error, facilitando así la localización de la relación (vista o tabla) que lo ha originado.

En cuanto a la estructura de este capítulo, en la sección 3.1 se introducen conceptos básicos relacionados con las bases de datos relacionales, la sintaxis y la semántica del lenguaje de consulta SQL. En la sección 3.2 se define el concepto de *caso de prueba* para una relación de la base de datos y en particular para el caso de vistas. El proceso de generación de los casos de prueba se describe en la sección 3.2.2 y el prototipo inicial y los detalles de implementación se presentan en la sección 3.2.3.

En la sección 3.3 se definen los conceptos básicos de depuración declarativa aplicados al caso de consultas SQL, siendo en las secciones 3.3.2 y 3.3.3 donde se presentan dos técnicas de depuración, la primera como una instancia del esquema general basada en árboles de cómputo y la segunda como un intento de refinar la anterior, considerando información adicional relacionada con el tipo de error. La sección 3.4 recoge las conclusiones.

3.1. Bases de datos relacionales

El modelo relacional propuesto por Codd [52] en los años 70 está considerado como uno de los pilares más importantes dentro del ámbito de las bases de datos. El elemento fundamental de este modelo es el concepto de *relación*, así como su representación en forma de tabla. Cada relación se identifica de forma única mediante un nombre

y en ella podemos distinguir las columnas, denominadas atributos, que representan las propiedades de la relación y que se caracterizan por un nombre. Las filas de las relación se llaman tuplas. Las columnas de una relación representan su esquema y normalmente se mantienen inalterables. En cambio las tuplas representan la información almacenada en un momento dado y constituyen la llamada *instancia* de la tabla. Pese a que la estructura lógica del modelo relacional es muy simple, su estructura teórica es muy sólida. Sobre el modelo relacional se ha definido el lenguaje de definición y manipulación de bases de datos relacionales SQL [54]. La gran mayoría de las bases de datos existentes se ajustan en mayor o menor medida al modelo relacional, siendo éste uno de los más extendidos.

En la siguiente sección definimos los conceptos básicos relativos a bases de datos relaciones, así como la sintaxis y semántica del lenguaje SQL.

3.1.1. Esquema de base de datos relacional e instancias

Definimos *esquema de base de datos relacional* D como una tupla $(\mathcal{T}, \mathcal{C})$, donde $\mathcal{T}$ es un conjunto finito de esquemas de tabla y $\mathcal{C}$ es un conjunto de restricciones que especifican el conjunto de valores admisibles en la tablas.

Un *esquema de tabla* $\mathcal{R}$ se representa de la forma $R(A_1, \ldots, A_n)$ donde R es un nombre de tabla y $(A_1, \ldots, A_n)$ es una lista de atributos. Cada atributo A_i tiene un dominio asociado (*integer*, *string*, ...) denotado como $dom(A_i)$. El dominio del esquema de tabla $\mathcal{R}$ se denota como $dom(\mathcal{R}) = dom(A_1) \times \cdots \times dom(A_n)$.

Dado un esquema de tabla $R(A_1, \ldots, A_n)$, se denomina *instancia de tabla* R (o simplemente tabla) a un multiconjunto de elementos en $dom(A_1) \times \cdots \times dom(A_n)$ y lo denotamos como $d(R)$. Una *fila* t de esquema de tabla $\mathcal{R}$ es un elemento del dominio $dom(\mathcal{R})$. Usamos la notación $|R|_t$ para representar la multiplicidad de una fila t en la instancia de tabla $d(R)$. De esta forma decimos que la fila $t \in R$ si $|R|_t > 0$ y $t \notin R$ en otro caso.

Cada fila t de la tabla R puede ser considerada como una función tal que $dom(t) = \{A_1, \ldots, A_n\}$, siendo $t(A_i)$ el valor del atributo A_i en t. Usamos la notación $t(R.A_i)$ para hacer referencia al valor del atributo A_i de la fila t en la tabla R. Desde el punto de vista de la programación lógica, cada atributo $R.A_i$ en el dominio $dom(t)$ puede ser considerado como una variable lógica y t como una sustitución.

La concatenación de dos filas t, s con dominio disjunto se define como la unión de ambas funciones y lo denotamos mediante la expresión $t \odot s$. Dada una fila t y una expresión aritmética e definida sobre atributos en $dom(t)$, usamos la notación $e(t)$ para representar el valor obtenido al aplicar la función t considerada como sustitución a la expresión e. Dada una secuencia de expresiones aritméticas $l = (e_1, \ldots, e_m)$ con variables en $dom(t)$, $m > 1$, la proyección $\pi_l(t)$ se define como una nueva fila de la forma $(e_1(t), \ldots, e_m(t))$.

En el esquema relacional se asume la existencia de *funciones de agregación* SUM, MAX, MIN, que realizan un cálculo sobre un multiconjunto de valores, devolviendo como resultado un solo valor. Llamamos *expresiones de agregación* a aquellas que incluyen alguna función de agregación. Dado un multiconjunto de filas $S = \{|t_1, \ldots, t_n|\}$ y una expresión de agregación e, $e(S)$ representa el valor obtenido después de aplicar

las funciones de agregación que aparezcan en e. Si el atributo $T.A$ aparece en una función de agregación de e, dicha función se aplica sobre el multiconjunto de valores $\{|t_1(T.A), \ldots, t_n(T.A)|\}$. Las expresiones de agregación pueden incluir atributos que no se ven afectados por funciones de agregación, pero solo si toman los mismos valores para todas las tuplas del conjunto afectado, como sucede con $T.B$ en el ejemplo siguiente:

Ejemplo 3.1.1. *Sea la expresión* $e = SUM(T.A) + T.B$ *y* $S = \{|t_1, t_2, t_3|\}$ *un multiconjunto de 3 filas con:*

$$\begin{array}{ll} t_1 = & \{T.A \mapsto 2, T.B \mapsto 5\} \\ t_2 = & \{T.A \mapsto 3, T.B \mapsto 5\} \\ t_3 = & \{T.A \mapsto 4, T.B \mapsto 5\} \end{array}$$

Entonces, $e(S) = sum(\{|2, 3, 4|\}) + 5 = 14$.

Definimos *fila parcial* a aquella fila que contiene el símbolo especial $\perp$ en alguno de sus atributos y *fila total* en otro caso. La notación $t(i)$ representa la i-ésima posición de la fila t. Dada la fila parcial t y S un multiconjunto de filas con el mismo número de posiciones que t, decimos que $t \in_\perp S$ si existe una fila $t' \in S$ tal que $t(i) = t'(i)$ para todo $i = 1 \ldots n$ que verifique $t(i) \neq \perp$, $t'(i) \neq \perp$.

En el conjunto $\mathcal{C}$ se especifican las condiciones de integridad del esquema. Consideramos tres tipos de condiciones: las condiciones de dominio, las condiciones de clave y las condiciones de integridad referencial.

Las *condiciones de dominio* definen los valores que pueden tomar los atributos de las tablas y vienen impuestas por el dominio del esquema, por lo que para toda fila t de la instancia de tabla R, se cumple $t(R.A_i) \in dom(A_i)$ para todo atributo A_i en $dom(t)$.

La *condición de integridad de clave primaria* de una tabla de esquema $R(A_1, \ldots, A_n)$ es un conjunto de atributos $\{A_{k_1}, \ldots, A_{k_r}\} \subseteq \{A_1, \ldots, A_n\}$ que identifican de forma unívoca cada fila de la instancia de la tabla R, es decir, para dos filas distintas cualesquiera t_1, t_2 de la instancia R se cumple:

$$t_1(R.A_{k_1}) \neq t_2(R.A_{k_1}) \vee \cdots \vee t_1(R.A_{k_r}) \neq t_2(R.A_{k_r})$$

La clave primaria de una tabla R se representa como $pk(T) = \{A_{k_1}, \ldots, A_{k_r}\}$.

Las *condiciones de integridad referencial* establecen relaciones entre filas de dos tablas R_1 y R_2 con esquemas $R_1(A_1, \ldots, A_n)$ y $R_2(B_1, \ldots, B_s)$ respectivamente, mediante la definición de una o más *claves ajenas*. Un conjunto de atributos $\{A_{f_1}, \ldots, A_{f_r}\} \subseteq \{A_1, \ldots, A_n\}$ es una clave ajena de la tabla R_1 que referencia a la tabla R_2 si satisface las siguientes condiciones:

- Los atributos A_{f_i} tienen el mismo dominio que los atributos de la clave primaria de R_2, es decir, $dom(A_{f_i}) = dom(B_{k_i})$ para $i = 1, \ldots, r$, con $pk(R_2) = \{B_{k_1}, \ldots, B_{k_r}\}$.
- Dada una fila cualquiera t_1 en la instancia de tabla R_1, ha de existir una fila t_2 en la instancia de tabla R_2 tal que $t_1(R_1.A_{f_i}) = t_2(R_2.B_{k_i})$ para $i = 1, \ldots, r$.

Sea $D = (\mathcal{T}, \mathcal{C})$ un esquema de base de datos con $\mathcal{T} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$. Una *instancia de base de datos* con esquema D es un conjunto de instancias de tabla $\{d(R_1), \ldots, d(R_m)\}$ que verifica las condiciones del conjunto $\mathcal{C}$ y se denota por d.

La expresión d_s denota una *instancia de base de datos simbólica* con esquema D y representa una instancia de base de datos cuyas filas pueden contener variables lógicas. Decimos que la sustitución μ satisface d_s cuando $(d_s\mu)$ es una instancia de base de datos que verifica las condiciones del conjunto $\mathcal{C}$. La sustitución μ debe sustituir todas las variables lógicas que aparecen en d_s por valores del dominio.

3.1.2. Sintaxis de SQL

SQL [79] es un lenguaje declarativo utilizado para manipular bases de datos relacionales. Es considerado como un estándar implementado por los principales motores o sistemas de gestión de bases de datos relacionales.

Las consultas a los datos almacenados en la base de datos se realizan mediante instrucciones del tipo select. En aplicaciones de bases de datos con muchas tablas, utilizar una única instrucción de tipo select es, en muchos casos, difícil de codificar. Para estos casos, el lenguaje SQL propone la utilización de *vistas*. Las *vistas* son consultas almacenadas que pueden ser consideradas conceptualmente como tablas, aunque no tienen una copia física de los datos. En general, hablamos de *relación* tanto para referirnos a una tabla como a una vista. La sintaxis para la creación de vistas es la siguiente:

create or replace **view** V(A_1, ... , A_n) **as**
Q;

Las vistas se crean de forma dinámica proporcionando un nombre a la vista V, una consulta asociada Q expresada mediante una instrucción select válida y los nombres de sus atributos $V.A_1, \ldots, V.A_n$. Una vista se define tanto a partir de tablas de la base de datos como a partir de otras vistas previamente definidas. Podemos ampliar el concepto de esquema de base de datos D definiéndolo como una tupla de tres elementos $(\mathcal{T}, \mathcal{C}, \mathcal{V})$, con $\mathcal{T}$ y $\mathcal{C}$ definidos como hasta ahora, siendo $\mathcal{V}$ un conjunto de vistas, cada una de ellas definida usando tablas de esquemas en $\mathcal{T}$ y otras vistas en $\mathcal{V}$.

La sintaxis completa de las consultas SQL se puede encontrar en [79]. Distinguimos tres tipos de consultas:

- **Consultas básicas.** Son aquellas cuya definición contiene las secciones select, from y where. Son de la forma:

  select e_1 E_1, ..., e_n E_n
  from R_1 B_1 , ..., R_m B_m
  where C_w;

  donde R_j B_j, $(j = 1 \ldots m)$ se refiere a la relación R_j de la base de datos (tabla o vista) siendo B_j su alias, C_w es una expresión condicional que identifica las

filas que la consulta recuperará y e_i $(i = 1 \ldots n)$, una expresión. Cada expresión e_i, con su alias asociado E_i, puede contener constantes, funciones predefinidas o atributos de la forma $B_j.A$, $1 \leq j \leq m$, siendo A un atributo de R_j.

- **Consultas agrupadas.** Son aquellas consultas básicas que incluyen además secciones group by y having. Son de la forma:

  select e_1 E_1, ..., e_n E_n
  from R_1 B_1 , ..., R_m B_m
  where C_w
  group by $A'_1, \ldots, A'_k$
  having C_h;

  Este tipo de consultas permite aplicar funciones de agregación (COUNT, SUM, MAX, MIN, ...) a grupos de filas, estando los grupos basados en los valores de los atributos $A'_1, \ldots, A'_k$ especificados en la sección group by. La expresión condicional C_h de la sección having se aplica a cada grupo de filas construido por la cláusula group by.

- **Consultas compuestas.** Son aquellas consultas que combinan los resultados de dos consultas mediante operaciones de conjunto tales como union [all], intersect [all] y except [all]. La palabra reservada all indica que el resultado de la consulta debe considerarse como un multiconjunto, es decir, puede tener filas duplicadas. Estas consultas se pueden expresar de la forma:

  $$Q_1 \quad \diamondsuit \quad Q_2$$

  donde Q_1, Q_2 son consultas SQL cuyo resultado tiene el mismo número de atributos y $\diamondsuit \in \{$ union [all], intersect [all] , except [all] $\}$.

El *árbol de dependencias* de una vista V del esquema de base de datos es un árbol cuya raíz está etiquetada con el nombre de la vista V. Los hijos son los árboles de dependencia de las relaciones que aparecen en la consulta que define la vista. Las hojas del árbol se corresponden con tablas del esquema.

3.1.3. Algebra relacional extendida

El Algebra Relacional Extendida (ERA, Extended Relational Algebra) [71] es una semántica para SQL basada en el concepto de multiconjunto que permite utilizar funciones de agregación, vistas y la mayoría de las características de las consultas escritas en el lenguaje SQL. En ERA, cada relación R con esquema $\mathcal{R}$ se define como un multiconjunto de filas en $dom(\mathcal{R})$ y es considerada como una expresión relacional. Las expresiones ERA permiten definir nuevas relaciones combinando relaciones previamente definidas [66]. Para ello se utilizan operadores de conjunto y multiconjunto. A continuación introducimos algunos de estos operadores (la definición formal de cada operador se puede encontrar en [66],[71]). Supongamos que R y S son dos

multiconjuntos. Sea μ una fila tal que $|R|_\mu = n$ y $|S|_\mu = m$. Estos operadores se definen como:

- **Unión e Intersección.** La unión de R y S, es un multiconjunto $R \cup_{\mathcal{M}} S$ donde la fila μ aparece $n+m$ veces. De forma análoga, la intersección de R y S denotada por $R \cap_{\mathcal{M}} S$, es un multiconjunto donde la fila μ aparece $min(n, m)$ veces. La unión de conjuntos $R \cup S$ es un conjunto donde la fila μ aparece $min(1, n + m)$ veces y la intersección de conjuntos $R \cap S$ es un conjunto donde la fila μ aparece $min(1, n, m)$ veces.

- **Diferencia.** La diferencia de R y S, es el multiconjunto $R \setminus_{\mathcal{M}} S$ en el cual la fila μ aparece $max(0, n - m)$ veces. La diferencia de conjuntos $R \setminus S$ es el conjunto en el cual la fila μ aparece $min(1, max(0, n - m))$ veces.

- **Proyección.** La expresión $\pi_{e_1 \mapsto A_1, \ldots, e_n \mapsto A_n}(R)$ produce una nueva relación que contiene alguno de los atributos de la relación R. Por cada fila $\mu \in R$ produce una nueva fila $\nu \in dom(A_1, \ldots, A_n)$ de la forma $(e_1\mu, \ldots, e_n\mu)$ en la nueva relación.

- **Selección.** Denotada por $\sigma_C(R)$, donde C es una condición que deben satisfacer todas las filas del resultado. El operador de selección sobre multiconjuntos aplica la condición de selección a cada una de las filas del multiconjunto de forma independiente descartando las que no cumplen la condición C.

- **Producto Cartesiano.** Denotado por $R \times S$, permite combinar las filas de las dos relaciones R y S. Si μ es una fila tal que $|R|_\mu = n$ y ν es una fila tal que $|S|_\nu = m$, entonces $R \times S$ es una nueva relación donde la fila $\mu \odot \nu$ aparece $n \times m$ veces.

- **Renombramiento.** La expresión $\rho_M(R)$ produce como resultado una nueva relación M donde la fila μ aparece n veces. La expresión $\rho_{A/B}(R)$ cambia el nombre del atributo A en todas las filas de R por B.

- **Operador de agrupación.** Denotado por γ, este operador permite agrupar filas de una relación con respecto al valor de uno o varios atributos. De esta forma, las funciones de agregación se aplican a cada uno de los grupos resultantes. La aplicación de este operador se denota por $\gamma_L(R)$, donde L es una lista de elementos, cada uno de los cuales puede ser atributo de la relación R o una expresión que utiliza funciones de agregación, cada una aplicada a un atributo de la relación R.

Las relaciones SQL pueden describirse mediante expresiones ERA. Usamos la notación Φ_R para representar la expresión ERA asociada a una relación SQL, ya sea una vista, una tabla o una consulta. Así, si Q es una consulta de la forma:

$Q =$ **select** e_1 E_1, …, e_n E_n
from R_1 B_1 , …, R_m B_m
where C_w;

su expresión ERA asociada es:

$$\Phi_Q = \Pi_{e_1 \to E_1, \ldots, e_n \to E_n}(\sigma_{C_w}(S)) \quad \text{con } S = \rho_{B_1}(R_1) \times \cdots \times \rho_{B_m}(R_m)$$

Tanto las vistas como las consultas SQL dependen de otras relaciones previamente definidas. Usamos la notación $\Phi_R(R_1, \ldots, R_n)$ para indicar que la consulta R depende de las relaciones $R_1, \ldots, R_n$. Si $M_1, \ldots, M_n$ son multiconjuntos, usamos la notación $\Phi_R(M_1, \ldots, M_n)$ para indicar que la expresión Φ_R es evaluada después de substituir $R_1, \ldots, R_n$ por $M_1, \ldots, M_n$. La expresión ERA de una tabla T del esquema de base de datos coincide con el nombre de la tabla, es decir, $\Phi_T = T$.

Definición 3.1.2. *La* ***respuesta computada***[1] $\| \Phi_R \|_d$ *de la expresión* Φ_R *con respecto a la instancia de base de datos* d *se define como:*

- *Si* R *es una tabla,* $\| \Phi_R \|_d = d(R)$.
- *Si* R *es una consulta (sentencia* select *o vista) y* $R_1, \ldots, R_n$ *relaciones que aparecen en la definición de* R*, entonces:*

  $$\| \Phi_R \|_d = \Phi_R(M_1, \ldots, M_n)$$

  donde $M_i = \| \Phi_{R_i} \|_d$.

La respuesta computada de la expresión Φ_R *en la instancia* d *es el resultado de evaluar la expresión* Φ_R *después de sustituir cada nombre de relación* R_i *por su respuesta computada* $\| \Phi_{R_i} \|_d$.

Puede observarse que $\| \Phi_R \|_d$ está bien definida ya que no se permite el uso de consultas recursivas[2].

Las consultas son ejecutadas por sistemas que implementan el lenguaje SQL. Dada una consulta Q y una instancia d, denotamos por $\mathcal{SQL}(Q, d)$ al resultado producido por un sistema que implementa SQL para dicha consulta en d. Usamos la notación $\mathcal{SQL}(R, d)$ para abreviar $\mathcal{SQL}(\mathsf{select * from}\ R, d)$ cuando R es una relación. Asumimos la existencia de implementaciones correctas de SQL.

Definición 3.1.3. *Una* implementación correcta de SQL *verifica*

$$\mathcal{SQL}(Q, d) = \| \Phi_Q \|_d$$

para toda consulta Q *e instancia* d.

3.2. Generación automática de casos de prueba para vistas SQL

En esta sección introducimos un marco teórico para la generación automática de casos de prueba para sistemas de vistas correlacionadas. Los casos de prueba generados son

[1] En la publicación [33](A.3) la respuesta computada $\| \Phi_R \|_d$ con respecto a la instancia d se denota por $\langle R \rangle$.

[2] La definición de consultas recursivas se permitió a partir del estándar SQL:1999, pero al no ser soportadas por todos los sistemas, no las consideramos aquí.

diseñados siguiendo un proceso de prueba de caja blanca y se obtienen mediante un mecanismo de ejecución simbólica [80, 55, 45]. Nuestros resultados teóricos en este sentido se limitan a un subconjunto lo suficientemente representativo del lenguaje SQL, con las siguientes características:

- Asumimos que en las secciones where y having, solo pueden aparecer subconsultas de tipo existencial[3]. Otras subconsultas como aquellas de la forma ... in Q, ... any Q o ... all Q, no se contemplan de forma directa, sin embargo muchas de ellas pueden ser transformadas a otras equivalentes (como se presenta en [10]) utilizando subconsultas existenciales.
- No se contemplan las subconsultas en la sección from. En realidad esto no es una limitación ya que pueden ser reemplazadas por vistas definidas previamente.
- Tampoco contemplamos el uso del operador distinct en la sección select, pero puede ser reemplazado por consultas equivalentes utilizando agregados [9].

Se ha optado por dejar fuera de nuestro estudio otras características del lenguaje, como el operador except, operaciones de tipo join, los valores nulos o las consultas recursivas.

A continuación, definimos tres tipos diferentes de casos de prueba para consultas SQL. Cada caso de prueba independiente para una vista V en el esquema D, es una instancia válida de base de datos d que cumple ciertas condiciones derivadas de la semántica de V. Posteriormente se describe el proceso de generación de dichos casos de prueba y se enuncian los resultados teóricos relacionados con la corrección y completitud de la técnica utilizada con respecto a la semántica ERA.

3.2.1. Casos de prueba para consultas SQL

Dada una instancia de base de datos d y una vista V, definimos tres tipos de casos de prueba dependiendo del resultado de la vista V en la instancia d.

Definición 3.2.1. *Sea d una instancia de base de datos no vacía y R una consulta (sentencia* select *o una vista). Decimos que d es un* ***caso de prueba positivo*** *(Positive Test Case-PTC) para la consulta R si $\| \Phi_R \|_d \neq \emptyset$. Los casos de prueba positivos se corresponden con casos de prueba que satisfacen el criterio de cobertura de decisión positivo.*

Las instancias de base de datos vacías, no se consideran casos de prueba válidos para consultas SQL. Para que una instancia d se la considere como un caso de prueba positivo para una vista V, es necesario que la respuesta computada de V con respecto a la instancia d contenga al menos una fila, la cual actuará como testigo de un posible error en la definición de la vista V. Para que esto ocurra, se deben cumplir ciertas condiciones o reglas dependiendo del tipo de consulta Q que define la vista V. En el caso de consultas básicas, debe existir al menos una fila en el dominio de la consulta que cumpla la condición que aparece en la sección where de la consulta Q. En el caso de consultas agrupadas, debe existir al menos un grupo válido (todas sus filas

[3]Las subconsultas existenciales son de forma exists Q (o not exists Q)

verifican la condición que aparece en la sección where) que cumpla la condición que aparece en la sección having. En el caso de consultas compuestas, distinguimos casos dependiendo del tipo de operación:

- $Q = Q_1$ **union** [**all**] Q_2. La instancia d es un PTC para V si es un PTC para Q_1 o para Q_2.
- $Q = Q_1$ **intersect** [**all**] Q_2. La instancia d es un PTC para V si es un PTC tanto para Q_1 como para Q_2.

Para que una instancia d se considere un caso de prueba negativo para una vista V, es necesario que la respuesta computada de V con respecto a la instancia d sea el conjunto vacío de filas. Los casos de prueba negativos se definen transformando la consulta Q que define la vista V y posteriormente aplicando el concepto de PTC a la consulta transformada. Con este propósito usamos la notación Q_{C_w} para indicar que la consulta Q es una consulta básica, siendo C_w la condición que aparece en la sección where de Q. Análogamente, usamos la notación $Q_{(C_w,C_h)}$ para indicar que la consulta Q es una consulta agrupada siendo C_h la condición que aparece en la sección having de Q. Si Q_{C_w} es de la forma:

select e_1 E_1, ..., e_n E_n
from R_1 B_1 , ..., R_m B_m
where C_w;

$Q_{not(C_w)}$ representa la siguiente consulta transformada:

select e_1 E_1, ..., e_n E_n
from R_1 B_1 , ..., R_m B_m
where not (C_w);

Análogamente a como se han definido las consultas transformadas para las consultas básicas, dada una consulta agrupada de la forma $Q_{(C_w,C_h)}$, se definen las siguientes consultas transformadas:

$$Q_{(not(C_w),C_h)}, \quad Q_{(C_w,not(C_h))}, \quad Q_{(not(C_w),not(C_h))}$$

Definición 3.2.2. *Sea d una instancia de base de datos no vacía y V una vista definida mediante la consulta Q. Definimos* ***caso de prueba negativo*** *(Negative Test Case-NTC) distinguiendo casos según la estructura de la consulta Q:*

- *Si $Q = Q_{C_w}$, entonces decimos que d es un NTC para la vista V si d es un PTC para la consulta $Q_{not(C_w)}$.*
- *Si $Q = Q_{(C_w,C_h)}$, entonces decimos que d es un NTC para la vista V si d es un PTC para alguna de las consultas $Q_{(not(C_w),C_h)}$, $Q_{(not(C_w),not(C_h))}$ o $Q_{(C_w,not(C_h))}$.*
- *Si Q es una consulta compuesta, distinguimos dos casos:*

- *Si $Q = Q_1$ **union [all]** Q_2, entonces decimos que d es un NTC para V si es un NTC tanto para Q_1 como para Q_2.*
- *Si $Q = Q_1$ **intersect [all]** Q_2, entonces decimos que d es un NTC para V si es un NTC para Q_1 o para Q_2.*

Los casos de prueba negativos se corresponden con casos de prueba que satisfacen el criterio de cobertura de decisión negativo.

Un caso de prueba negativo para una consulta agrupada se corresponde con una instancia cuyas filas no verifican la condición que aparece en la sección where o cuyos grupos no verifican la condición que aparece en la sección having. La ventaja de definir el concepto de NTC en función de PTC es que solo es necesario estudiar el proceso de generación de casos de prueba positivos. Por supuesto, las definiciones anteriores pueden variar dependiendo del nivel de cobertura que se desee alcanzar. Por ejemplo, un NTC para vistas cuya consulta asociada es del tipo $Q_{(C_w,C_h)}$, podría definirse simplemente como un PTC para la consulta $Q_{(C_w,not(C_h))}$.

En algunos casos, una instancia d es un caso de prueba positivo y negativo al mismo tiempo, alcanzando así un nivel de cobertura de predicado (*predicate coverage* [12]) con respecto a la conjunción de las condiciones de las secciones where y having.

Definición 3.2.3. *Sea d una instancia de base de datos no vacía y V una vista. Decimos que d es un **caso de prueba positivo-negativo** (PNTC) para la vista V si d es un PTC y un NTC para V.*

Ejemplo 3.2.4. *Sea T una tabla de la base de datos con un único atributo A. Sean dos instancias d_1, d_2, tal que $d_1(T) = \{\!|\mu_1, \mu_2|\!\}$ y $d_2(T) = \{\!|\mu_1|\!\}$, con:*

$$\mu_1 = \{T.A \mapsto 5\}, \qquad \mu_2 = \{T.A \mapsto 7\}$$

Sean las vistas V y W definidas mediante las consultas Q_1 y Q_2 respectivamente, con:

$Q_1 =$ **select** A
from T
where A = 5 ;

$Q_2 =$ **select** R_1.A
from T R_1
where R_1.A=5 **and not exists** (**select** R_2.A
from T R_2
where R_2.A<>5);

La instancia d_1, es un PNTC para la vista V, el cual verifica el criterio de cobertura de predicado. Por otro lado, el conjunto $\{d_1, d_2\}$ constituye un conjunto de casos de prueba que satisface el criterio de cobertura de predicado para la vista W, siendo d_1 un NTC y d_2 un PTC. Como podemos observar, no existe una instancia d que sea un PNTC para la vista W.

Para que una instancia d de la base de datos con esquema $D = (\mathcal{T}, \mathcal{C}, \mathcal{V})$ sea válida ha de cumplir una serie de condiciones; son aquellas que se refieren a las condiciones de

integridad $\mathcal{C}$ impuestas por el esquema. Estas condiciones junto con las condiciones que se deben verificar para que d sea considerada un PTC para una vista $V \in \mathcal{V}$ se pueden modelar mediante un conjunto de fórmulas de la lógica de primer orden. Estas fórmulas sirven tanto para probar que d es un PTC para una vista V como para generar de forma automática una instancia válida d en el esquema, de forma que d sea un PTC para V.

En [114] se presenta un criterio apropiado para medir la calidad de los casos de prueba para consultas SQL denominado Full Predicate Coverage (SQLFpc). En este trabajo se define un método de transformación de la vista inicial V en n-vistas $V_1, \ldots, V_n$, de tal forma que el conjunto de casos de prueba positivos para cada una de las vistas V_i, constituye un conjunto de casos de prueba verificando el criterio anteriormente citado.

3.2.2. Proceso de generación de PTC

El proceso de generación de un PTC para la vista V en el esquema D se inicia con la creación de una *instancia de base de datos simbólica* d_s; para cada una de las tablas T del esquema de base de datos D, se crea una instancia $d_s(T)$ con un número arbitrario de filas (parametrizado por el usuario) donde el valor de cada atributo en cada una de las filas se corresponde con una variable lógica libre satisfaciendo las restricciones de dominio.

Posteriormente, para cada una de las relaciones R del esquema de base de datos D, se construye un multiconjunto $\theta(R)$ de pares de la forma (φ, μ), siendo φ una fórmula de primer orden y μ una fila en la instancia $d_s(R)$. Este multiconjunto describe las condiciones que deben cumplir cada una de las relaciones de la instancia d_s, ya sea una tabla o una vista, para que d_s constituya un PTC para la vista V. La idea intuitiva es que la instancia $d_s(R)$ contiene solo aquellas filas μ tales que las condiciones expresadas por su fórmula asociada φ se cumplen en la instancia d_s.

Una vez creado el multiconjunto $\theta(V) = \{\!|(\psi_1, \mu_1), \ldots, (\psi_n, \mu_n)|\!\}$, se traduce la fórmula:

$$\delta = (\bigvee_{i=1}^{n} \psi_i)$$

a un lenguaje de programación con restricciones específico (en nuestro caso hemos optado por SICStus Prolog). El resultado es un conjunto de restricciones sobre las variables de la instancia d_s, generando así un problema de satisfacción de restricciones [113], cuya solución (en caso de ser satisfactible) constituirá un PTC para la vista V.

Definición 3.2.5. *Sea $D = (\mathcal{T}, \mathcal{C}, \mathcal{V})$ un esquema de base de datos, d_s una instancia de base de datos simbólica con esquema D y R una relación en D. Definimos $\theta(R)$ como un multiconjunto de pares de la forma (φ, μ), siendo φ una fórmula de primer orden y μ una fila en la instancia $d_s(R)$. Este multiconjunto se define distinguiendo casos dependiendo de si R es una tabla o una vista:*

1. *Sea T una tabla del esquema de base de datos D con instancia $d_s(T) = \{\!|\mu_1, \ldots, \mu_n|\!\}$, entonces:*

$$\theta(T) = \{\!|(\varphi_1, \mu_1), \ldots, (\varphi_n, \mu_n)|\!\}$$

donde cada φ_i es una fórmula de primer orden que representa las restricciones de clave primaria y clave ajena. Si T tiene clave primaria $pk(T) = \{A_1, \ldots, A_m\}$, definimos:

$$\phi_i = (\bigwedge_{j=1, j\neq i}^{n} (\bigvee_{k=1}^{m} (\mu_i(T.A_k) \neq \mu_j(T.A_k))))$$

para $i = 1 \ldots n$. Las fórmulas $\phi_1, \ldots, \phi_n$ permiten representar las restricciones de clave primaria de la tabla T. Si la tabla T se ha definido sin clave primaria, tenemos que $\phi_i = true$ para todo $i = 1 \ldots n$. Por otro lado, si la tabla T tiene un número s de claves ajenas $fk_p(T, T_p) = \{(A_1, \ldots, A_m), (B_{p1}, \ldots, B_{pm})\}$, con $p = 1 \ldots s$ y T_p una tabla del esquema de base de datos D con instancia $d_s(T_p) = \{\!|\nu_1, \ldots, \nu_{n_2}|\!\}$, definimos:

$$\psi_{ip} = (\bigvee_{j=1}^{n_2} (\bigwedge_{k=1}^{m} (\mu_i(T.A_k) = \nu_j(T_p.B_{pk}))))$$

para $i = 1 \ldots n$ y $p = 1 \ldots s$. Las fórmulas ψ_{ip} permiten representar las restricciones de clave ajena. De esta forma, tenemos que:

$$\varphi_i = \phi_i \wedge \psi_{i1} \wedge \cdots \wedge \psi_{is}$$

2. *Sea V una vista del esquema de base de datos D definida como $V =$* create view *$V(A_1, \ldots, A_n)$* as *Q. El multiconjunto $\theta(V)$ se define distinguiendo casos en base al tipo de consulta.*

   - *Si Q es una consulta básica de la forma:*

     select $e_1\ E_1, \ldots, e_n\ E_n$ from $R_1\ B_1, \ldots, R_m\ B_m$ where C;

     entonces:

     $$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

     En primer lugar construimos el multiconjunto P de forma que para cada $(\psi_1, \nu_1) \in \theta(R_1), \ldots, (\psi_m, \nu_m) \in \theta(R_m)$ con multiplicidad $|\theta(R_1)|_{(\psi_1,\nu_1)} = n_1, \ldots, |\theta(R_m)|_{(\psi_m,\nu_m)} = n_m$ se tiene que el par $t = (\psi_1 \wedge \cdots \wedge \psi_m, {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m})$ está en P con multiplicidad $|P|_t = n_1 \times \cdots \times n_m$.
     Definimos $\theta(Q)$ a partir de P. Para cada par de la forma $(\psi, \mu) \in P$ con multiplicidad k se definen:

     - $s_Q(\mu) = \{E_1 \mapsto e_1(\mu), \ldots, E_n \mapsto e_n(\mu)\}$
     - *La fórmula lógica de primer orden $\varphi(C, \mu)$ se define como:*
       - *Si $C \equiv C_1$ and C_2, entonces $\varphi(C, \mu) = \varphi(C_1, \mu) \wedge \varphi(C_2, \mu)$*
       - *Si $C \equiv C_1$ or C_2 , entonces $\varphi(C, \mu) = \varphi(C_1, \mu) \vee \varphi(C_2, \mu)$*
       - *Si $C \equiv$ not (C_1), entonces $\varphi(C, \mu) = \neg\varphi(C_1, \mu)$*
       - *Si $C \equiv e$ con e una expresión sin subconsultas, $\varphi(C, \mu) = C\mu$*
       - *Si $C \equiv (exists\ Q_E)$, con $\theta(Q_E) = \{\!|(\psi_1, \mu_1), \ldots (\psi_k, \mu_k)|\!\}$. Entonces $\varphi(C, \mu) = (\vee_{i=1}^{k} \psi_i)$.*

Entonces, $(\psi \wedge \varphi(C,\mu), s_Q(\mu))$ *está en* $\theta(Q)$ *con* $|\theta(Q)|_{(\psi\wedge\varphi(C,\mu),s_Q(\mu))} = k$

- *Si* Q *es una consulta agrupada de la forma:*

  select $e_1\ E_1, \ldots, e_n\ E_n$ from $R_1\ B_1, \ldots, R_m\ B_m$
  where C_w group by $e'_1, \ldots, e'_k$ having C_h

  entonces:

  $$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

  Definimos $\theta(Q)$ *a partir del multiconjunto* P *construido como en el apartado anterior.*

  Para cada multiconjunto no vacío $A = \{\!|(\psi_1,\mu_1), \ldots, (\psi_j,\mu_j)|\!\} \subseteq P$ *tal que* $|A|_{(\psi_i,\mu_i)} = |P|_{(\psi_i,\mu_i)}$ *para toda tupla* $(\psi_i,\mu_i) \in A$, $1 \leq i \leq j$, *se definen:*

  - $\Pi_1(A) = \psi_1 \wedge \cdots \wedge \psi_j$,
  - $\Pi_2(A) = \{\!|\ \mu_1, \ldots, \mu_j|\!\}$,
  - $aggregate(Q,A) = group(Q, \Pi_2(A)) \wedge maximal(Q,A) \wedge \varphi(C_h, \Pi_2(A))$
  - $group(Q,A) = (\varphi(C_w,\mu_1) \wedge \cdots \wedge \varphi(C_w,\mu_j)) \wedge$
    $(\bigwedge_{i=1\ldots j}(\bigwedge_{l>i}(e'_1(\mu_i) = e'_1(\mu_l) \wedge \cdots \wedge e'_k(\mu_i) = e'_k(\mu_l))))$
  - $maximal(Q,A) = \bigwedge_{(\psi,\mu)\in P\wedge(\psi,\mu)\notin A}(\neg\psi \vee \neg group(Q, A \cup (\psi,\mu)))$

  Entonces, $t = (\Pi_1(A) \wedge aggregate(Q,A), s_Q(\Pi_2(A)))$ *está en* $\theta(Q)$ *con multiplicidad 1.*

- *Para consultas compuestas:*
  - *Si* $Q = (Q_1$ union [all] $Q_2)$, *entonces:*

    $$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

    con $E_1, \ldots, E_n$ *los nombres de los atributos en la sección* select *de las consultas* Q_1 *y* Q_2.
    - *Si* $Q = (Q_1$ union all $Q_2)$, $\theta(Q) = \theta(Q_1)\ \cup_{\mathcal{M}} \theta(Q_2)$
    - *Si* $Q = (Q_1$ union $Q_2)$,

      $$\begin{aligned}\theta(Q) = \quad & \{(\psi,\mu)\ \mid \\ & \psi = (\psi_{11} \vee \cdots \vee \psi_{1k_1}) \vee (\psi_{21} \vee \cdots \vee \psi_{2k_2}), \\ & \quad (\psi_{11},\mu) \in \theta(Q_1), \ldots, (\psi_{1k_1},\mu) \in \theta(Q_1), \\ & \quad (\psi_{21},\mu) \in \theta(Q_2), \ldots, (\psi_{2k_2},\mu) \in \theta(Q_2)\}\end{aligned}$$

  - *Si* $Q = (Q_1$intersection $Q_2)$, *entonces:*

    $$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

    con $E_1, \ldots, E_n$ *los nombres de los atributos en la sección* select *de las consultas* Q_1 *y* Q_2, *siendo* $\theta(Q)$ *de la forma:*

    $$\begin{aligned}\theta(Q) = \quad & \{(\psi,\mu)\ \mid \\ & \psi = (\psi_{11} \vee \cdots \vee \psi_{1k_1}) \wedge (\psi_{21} \vee \cdots \vee \psi_{2k_2}), \\ & \quad (\psi_{11},\mu) \in \theta(Q_1), \ldots, (\psi_{1k_1},\mu) \in \theta(Q_1), \\ & \quad (\psi_{21},\mu) \in \theta(Q_2), \ldots, (\psi_{2k_2},\mu) \in \theta(Q_2)\}\end{aligned}$$

Obsérvese que si select $e_1\ E_1,\ \ldots,\ e_n\ E_n$ *es la sección* select *de la consulta* Q, $s_Q(x)$ *representa una fila* μ *con dominio* $\{E_1, \ldots, E_n\}$ *tal que* $E_i(x) = e_i(x)$, *con* $i = 1 \ldots n$. *En el caso de que en la sección* select *no aparezcan los* E_i*'s, se asume que* $E_i = e_i$.

Según la definición 3.2.5, en el apartado (1) se generan las fórmulas asociadas a cada tabla T del esquema de base de datos D y que definen las condiciones que debe cumplir la instancia d_s para ser considerada válida, en el sentido de que verifica las condiciones de integridad de clave primaria así como las condiciones de integridad referencial. Por otro lado, en el apartado (2) se trata el caso de vistas del esquema de base de datos D. En este caso, P representa el producto cartesiano simbólico de las relaciones que aparecen en la sección from de la consulta que define la vista. El multiconjunto A representa un subconjunto de tuplas en P. La condición $group(Q, A)$ especifica las condiciones que debe cumplir el multiconjunto A para ser considerado un grupo con respecto a los atributos en la sección group by. La fórmula $maximal(Q, A)$ representa la restricción de que no exista en P otro grupo mayor, es decir, que no exista una tupla t en P, que verificando la condición expresada en la sección where de la consulta, forme grupo con A. Por último, la fórmula $aggregate(Q, A)$ permite aplicar la expresión condicional en la sección having de la consulta al grupo maximal A.

Para ilustrar la definición anterior mostramos un pequeño ejemplo:

Ejemplo 3.2.6. *Sean* T_1 *y* T_2 *dos tablas definidas en el esquema* D *con esquemas* $T_1(A, B)$ *y* $T_2(C)$ *respectivamente. Supongamos que tenemos cuatro vistas SQL* V_1, V_2, V_3 *y* V_4, *definidas en el esquema* D *como:*

create table T_1*(*A int , B char*);*
create table T_2*(*C int *);*

create view V_1*(*A_1, A_2*)* as
select $T_1'.A\ E_1$, $T_1'.B\ E_2$
from $T_1\ T_1'$
where $T_1'.A \geq 10$

create view V_2*(*A*)* as
select $T_2'.C\ E_1$
from $V_1\ V_1'$, $T_2\ T_2'$
where $V_1'.A_1 + T_2'.C = 0$

create view V_3*(*A*)* as
select*(*$V_1'.A_1$*)* E
from V_1 *as* V_1'
where exists
(select $T_2'.C\ E1$
from $T_2\ T_2'$
where $T_2'.C = V_1'.A_1$*)*

create view $V_4(A)$ as
select $V_1'.A_2\ E$
from V_1 *as* V_1'
where $V_1'.A_2 =$ *"a"*
group by $V_1'.A_2$
having sum$(V_1'.A_1) > 100$*;*

Sea $d_s = \{d_s(T_1), d_s(T_2)\}$ *una instancia simbólica, tal que:*

$$d_s(T_1) = \{\!|\mu_1, \mu_2|\!\}, \qquad d_s(T_2) = \{\!|\mu_3, \mu_4|\!\}$$

con:

$$\mu_1 = \{T_1.A \mapsto x_1, T_1.B \mapsto y_1\}$$
$$\mu_2 = \{T_1.A \mapsto x_2, T_1.B \mapsto y_2\}$$
$$\mu_3 = \{T_2.C \mapsto z_1\}$$
$$\mu_4 = \{T_2.C \mapsto z_2\}$$

Partiendo de la instancia simbólica d_s y suponiendo que no se han definido en el esquema D restricciones de clave primaria ni restricciones de clave ajena, los conjuntos $\theta(T_1)$ y $\theta(T_2)$ según la definición 3.2.5 son los siguientes:

$$\theta(T_1) = \{\!|(true, (T_1.A \mapsto x_1, T_1.B \mapsto y_1)), (true, (T_1.A \mapsto x_2, T_1.B \mapsto y_2))|\!\}$$
$$\theta(T_2) = \{\!|(true, (T_2.C \mapsto z_1)), (true, (T_2.C \mapsto z_2))|\!\}$$

A continuación construimos los conjuntos $\theta(V_1)$, $\theta(V_2)$, $\theta(V_3)$ y $\theta(V_4)$.

$$\begin{aligned}\theta(V_1) = \{\!| \quad & (x_1 \geq 10, \{V_1.A_1 \mapsto x_1, V_1.A_2 \mapsto y_1\}) \ , \\ & (x_2 \geq 10, \{V_1.A_1 \mapsto x_2, V_1.A_2 \mapsto y_2\}) \quad |\!\}\end{aligned}$$

El multiconjunto $\theta(V_1)$ está definido mediante dos tuplas indicando que el resultado de la consulta V_1 contendrá a lo sumo dos filas, cada una de las cuales se corresponde con una fila de la tabla T_1. La fórmula asociada a cada una de las filas impone la condición de que el valor del atributo $T_1.A$ debe ser mayor que 10.

$$\begin{aligned}\theta(V_2) = \{\!| \quad & (x_1 \geq 10 \wedge x_1 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & (x_1 \geq 10 \wedge x_1 + z_2 = 0, \{V_2.A \mapsto z_2\}), \\ & (x_2 \geq 10 \wedge x_2 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & (x_2 \geq 10 \wedge x_2 + z_2 = 0, \{V_2.A \mapsto z_2\}) \quad |\!\}\end{aligned}$$

El multiconjunto $\theta(V_2)$ contiene cuatro tuplas, cada una de las cuales proviene de la combinación de una tupla de V_1 y una tupla de T_2. Por ejemplo, la tupla $(x_1 \geq 10 \wedge x_1 + z_1 = 0, \{V_2.A \mapsto z_1\})$ de $\theta(V_2)$ se genera como combinación de la tupla $(x_1 \geq 10, \{V_1.A_1 \mapsto x_1, V_1.A_2 \mapsto y_1\})$ de $\theta(V_1)$ y de la tupla $(true, (T_2.C \mapsto z_1))$ de $\theta(T_2)$. Siguiendo el proceso descrito anteriormente, los multiconjuntos $\theta(V_3)$ y $\theta(V_4)$ son los siguientes:

$$\begin{aligned}\theta(V_3) = \{\!| \quad & (x_1 \geq 10 \wedge ((z_1 = x_1) \vee (z_2 = x_1)), \{V_3.A \mapsto x_1\}), \\ & (x_2 \geq 10 \wedge ((z_1 = x_2) \vee (z_2 = x_2)), \{V_3.A \mapsto x_2\}) \quad |\!\}\end{aligned}$$
$$\theta(V_4) = \{\!|(\psi_1, \{V_4.A \mapsto y_1\}), (\psi_2, \{V_4.A \mapsto y_1\}), (\psi_3, \{V_4.A \mapsto y_2\})|\!\}$$
$$\begin{aligned}\psi_1 = \quad & (x_1 \geq 10 \wedge x_2 \geq 10) \wedge (y_1 = \text{“}a\text{”} \wedge \ y_2 = \text{“}a\text{”} \wedge \ y_1 = y_2) \wedge \\ & (x_1 + x_2 > 100)\end{aligned}$$
$$\begin{aligned}\psi_2 = \quad & (x_1 \geq 10) \wedge (y_1 = \text{“}a\text{”}) \wedge \\ & (\neg(x_2 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \wedge \ y_2 = \text{“}a\text{”} \wedge \ y_1 = y_2)) \wedge (x_1 > 100)\end{aligned}$$
$$\begin{aligned}\psi_3 = \quad & (x_2 \geq 10) \wedge (y_2 > \text{“}a\text{”}) \wedge \\ & (\neg(x_1 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \wedge \ y_2 = \text{“}a\text{”} \wedge \ y_1 = y_2)) \wedge (x_2 > 100)\end{aligned}$$

Se puede observar que la vista V_4 realiza grupos sobre el atributo A_2 de la vista V_1. Como $\theta(V_1)$ contiene solo dos tuplas, $\theta(V_4)$ contiene como mucho tres tuplas, una por cada posible agrupación de las tuplas en V_1; el primer grupo se corresponde con las dos tuplas de V_1, el segundo grupo se corresponde con la primera tupla y el tercer grupo se corresponde con la segunda tupla de V_1.

El siguiente teorema enuncia las condiciones que debe cumplir una instancia d para ser un PTC para la relación R en un esquema D. Restringimos este resultado a consultas sin subconsultas debido a la limitación del Algebra Relacional Extendida para representar este tipo de consultas. El teorema 3.2.7 se corresponde con el Corolario 1 que se puede encontrar, junto con su demostración, en la publicación [33](A.3).

Teorema 3.2.7. *Sea D un esquema de base de datos, d_s una instancia simbólica. Supongamos que las vistas y consultas en D no contienen subconsultas. Sea R una relación en D tal que $\theta(R) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, y η una sustitución sobre las variables en d_s. Entonces $d_s\eta$ es un PTC para la relación R si y solo si $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

Por el teorema 3.2.7 , la instancia $d_s\eta$ con $\eta = \{x_1 \mapsto 10, y_1 \mapsto$ "a", $x_2 \mapsto 91, y_2 \mapsto$ "a"$\}$ es un PTC para la vista V_4 del ejemplo 3.2.6, ya que $\psi_1\eta = true$.

3.2.3. Implementación y prototipo

Como parte de esta tesis hemos implementado un prototipo de generador de casos de prueba siguiendo las ideas teóricas aquí presentadas, el cual se ha incluido en el sistema Datalog DES, disponible en la dirección:

`http://gpd.sip.ucm.es/yolanda/research.htm`

En esta página se da la información necesaria para la instalación, ejecución y utilización del generador.

El generador de casos de prueba parte de un esquema de base de datos D definido mediante el lenguaje SQL y una vista V para la cual se va a generar el PTC. A continuación se describe un problema de satisfacción de restricciones cuya solución, en caso de ser satisfactible, constituirá un PTC para V.

1. Creación de la *instancia de base de datos simbólica* d_s y generación de fórmulas. Para cada una de las tablas T del esquema de base de datos D, se crea una instancia $d_s(T)$ con un número arbitrario de filas. El conjunto de todas las variables que aparecen en la instancia d_s constituye las variables del problema de decisión. A continuación, siguiendo las ideas de la definición 3.2.5, se construye el multiconjunto $\theta(V) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, como se ha descrito en la sección anterior.

2. Traducción de las fórmulas a lenguaje de restricciones. Una vez generado el multiconjunto $\theta(V) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, se traduce la fórmula $\delta = (\bigvee_{i=1}^{n} \psi_i)$ al lenguaje de restricciones de SICStus Prolog. Las condiciones de dominio se pueden definir en SQL mediante la cláusula constraint o indicando el tipo de cada uno de los atributos en la definición de la tabla. En nuestro prototipo inicial se manejan dos tipos de datos: integer y string. Las restricciones sobre enteros se manejan de forma directa usando el resolutor de restricciones de dominio finito ($\mathcal{FD}$) de SICStus Prolog. Para manejar el tipo de datos string, y dado que SICStus Prolog no tiene resolutor de cadenas, construimos un diccionario

intermedio que asocia cada constante cadena que aparece en el esquema de base de datos con un valor de tipo entero. Posteriormente se utiliza el resolutor de restricciones de dominio finito ($\mathcal{FD}$) con las operaciones de igualdad y desigualdad. Otras operaciones sobre cadenas de caracteres habituales en SQL no se incluyen en el prototipo inicial. Dado que la fórmula δ se define mediante conjunciones y disyunciones, el generador usa *reificación* como una forma eficiente de implementar estas conectivas. Cada restricción atómica se reifica y se transforman las conjunciones y disyunciones en restricciones de dominio finito de la forma $B_1 * B_2 * \cdots * B_k \# = B_0$ y $B_1 + B_2 + \cdots + B_k \ \# = \ B_0$ respectivamente.

Por ejemplo, para generar un PTC para la vista V_4 del ejemplo 3.2.6 será necesario reificar las fórmulas ψ_1, ψ_2 y ψ_3. Suponiendo que la constante "a" está asociada en el diccionario intermedio al valor entero 1, la fórmula ψ_1 se transforma en el siguiente conjunto de restricciones:

$$
\begin{array}{l}
x_1 \ \# \geq \ 10 \ \# <==> \ B_{11}, \\
x_2 \ \# \geq \ 10 \ \# <==> \ B_{12}, \\
y_1 \ \# = \ 1 \ \# <==> \ B_{13}, \\
y_2 \ \# = \ 1 \ \# <==> \ B_{14}, \\
y_1 \ \# = \ y_2 \ \# <==> \ B_{15}, \\
x_1 + x_2 \ \# > \ 100 \ \# <==> \ B_{16}, \\
B_{11} * B_{12} * B_{13} * B_{14} * B_{15} * B_{16} \ \# = \ C_1
\end{array}
\qquad \text{(C1)}
$$

La fórmula ψ_2 se transforma en el siguiente conjunto de restricciones:

$$
\begin{array}{l}
x_1 \ \# \geq \ 10 \ \# <==> \ B_{21}, \\
y_1 \ \# = \ 1 \ \# <==> \ B_{22}, \\
x_2 \ \# \geq \ 10 \ \# <==> \ B_{23}, \\
y_1 \ \# = \ 1 \ \# <==> \ B_{24}, \\
y_2 \ \# = \ 1 \ \# <==> \ B_{25}, \\
y_1 \ \# = \ y_2 \ \# <==> \ B_{26}, \\
B_{24} * B_{25} * B_{26} \ \# = \ B_{27} \\
B_{23} + B_{27} \# <= 1 \# <==> B_{28}, \\
x_1 \ \# > \ 100 \ \# <==> \ B_{29}, \\
B_{21} * B_{22} * B_{28} * B_{29} \ \# = \ C_2
\end{array}
\qquad \text{(C2)}
$$

La fórmula ψ_3 se transforma en el siguiente conjunto de restricciones:

$$
\begin{array}{l}
x_2 \ \# \geq \ 10 \ \# <==> \ B_{31}, \\
y_2 \ \# > \ 1 \ \# <==> \ B_{32}, \\
x_1 \ \# \geq \ 10 \ \# <==> \ B_{33}, \\
y_1 \ \# = \ 1 \ \# <==> \ B_{34}, \\
y_2 \ \# = \ 1 \ \# <==> \ B_{35}, \\
y_1 \ \# = \ y_2 \ \# <==> \ B_{36}, \\
B_{34} * B_{35} * B_{36} \ \# = \ B_{37} \\
B_{33} + B_{37} \# <= 1 \# <==> B_{38}, \\
x_2 \ \# > \ 100 \ \# <==> \ B_{39}, \\
B_{31} * B_{32} * B_{38} * B_{39} \ \# = \ C_3
\end{array}
\qquad \text{(C3)}
$$

```
DES-SQL> CREATE OR REPLACE TABLE t1( a  int PRIMARY KEY, b char);
DES-SQL> CREATE OR REPLACE VIEW v1(a1, a2) AS
            SELECT t1.a e1, t1.b e2 FROM t1 where t1.a >= 10;
DES-SQL> CREATE OR REPLACE VIEW v4(a)  AS
            SELECT v1.a2 e1 FROM v1
            WHERE v1.a2 = 'a' GROUP BY v1.a2 having sum(v1.a1) > 100;


DES-SQL> /test_case v4 positive

Info: Test case over integers and strings:
[t1(10,a),t1(91,a)]
```

Figura 3.1: Sesión de Generación de un caso de prueba para la vista V_4 del ejemplo 3.2.6.

Cualquier solución al conjunto de restricciones formado por (C1), (C2), (C3) y la restricción $C_1 + C_2 + C_3 \ \# >= 1$ permitirá construir una instancia de la base de datos que constituye un PTC para la vista V_4.

3. Una vez modelado el problema, se ejecuta el resolutor de restricciones de SICStus Prolog. La solución a dicho problema constituye un PTC para la vista V. Por ejemplo, una solución al conjunto formado por (C1), (C2), (C3) y la restricción $C_1 + C_2 + C_3 \ \# >= 1$ es la proporcionada por la sustitución $\{x_1 \mapsto 10, y_1 \mapsto 1, x_2 \mapsto 91, y_2 \mapsto 1\}$. Partiendo de esta solución y del diccionario intermedio, se construye la instancia:

$$d(T_1) = \{|\{T_1.A \mapsto 10, T_1.B \mapsto \text{“}a\text{”}\}, \{T_1.A \mapsto 91, T_1.B \mapsto \text{“}a\text{”}\}|\}$$

En la figura 3.1 se muestra una sesión de generación de casos de prueba en el sistema DES para la vista V_4 de nuestro ejemplo. Se crean en el sistema DES la tabla `t1` y dos vistas `v1` y `v4`. Si se quiere generar un PTC para la vista `v4` basta con escribir el comando `/test_case v4 positive`. El generador de casos de prueba devuelve como resultado una instancia de la tabla `t1` con dos filas `[t1(10,a),t1(91,a)]`, cuya representación en forma de tabla es como sigue:

t1

a	b
10	a
91	a

Esta instancia de la tabla `t1` constituye, de acuerdo con el teorema 3.2.7 un PTC para nuestro ejemplo. El usuario puede configurar el tamaño mínimo Min y máximo Max de los casos de prueba. Inicialmente, el sistema intenta generar una instancia con número de filas Min. Si no es posible generarla, el número de filas se incrementa

en una unidad y se vuelve a intentar generar la instancia. El proceso se repite hasta que se alcanza el límite máximo Max o hasta que se encuentra una instancia d, siendo d un caso de prueba. En la práctica, es imposible asegurar que el sistema es capaz de generar casos de prueba cuyo tamaño no sea mayor que Max.

Para finalizar señalamos que las principales limitaciones de la versión actual de la herramienta es la falta de tratamiento de nulos [43] y la extensión del resolutor de cadenas para tratar funciones habituales como *length*, *substring*, etc. Sin embargo nos ha servido tanto para materializar las ideas teóricas aquí presentadas como para la integración con otras herramientas dedicadas a la depuración y prueba de consultas SQL. Es el caso de la depuración declarativa de consultas SQL que presentamos en la siguiente sección.

3.3. Depuración declarativa de vistas SQL

En esta sección introducimos dos propuestas de depuración de vistas SQL basadas en la comparación entre la respuesta computada por el sistema SQL que implementa ERA y la respuesta esperada por el usuario. La primera propuesta se basa en la definición de una instancia del esquema general de depuración declarativa en la que el cómputo de una consulta se representa mediante un árbol de cómputo. La segunda propuesta, aunque también está basada en depuración declarativa, no construye el árbol de cómputo de forma explícita. Surge en el intento de refinar la propuesta anterior considerando información adicional relacionada con el tipo de error. En la fase inicial, se representa el árbol de cómputo mediante un conjunto de cláusulas lógicas, dando lugar a un programa lógico. La información adicional proporcionada por el usuario se registra en forma de nuevas cláusulas con el objetivo de minimizar el número de preguntas y reducir su complejidad.

Introducimos en primer lugar los conceptos que nos permiten presentar ambas propuestas.

3.3.1. Síntomas y errores

El síntoma de error es un resultado inesperado en la consulta de una relación.

Definición 3.3.1. *Sea D un esquema de base de datos, d una instancia de base de datos con esquema D y R una relación en D. La* ***respuesta esperada*** *de la relación R con respecto a la instancia d es un multiconjunto denotado por $\mathcal{I}(R, d)$ que contiene la respuesta que el usuario espera para la consulta* `select * from R;` *en la instancia d.*

En el caso de vistas SQL, la respuesta esperada de la vista no solo depende de su semántica, sino también del contenido de las tablas en la instancia de base de datos d. Este concepto se corresponde con la idea de *interpretación pretendida* empleada habitualmente en depuración algorítmica.

courses

id	level
c1	1
c2	2
c3	3
c0	0

registration

student	course	pass
Alba	c1	0
Alba	c2	1
Anna	c1	1
Anna	c2	1
Anna	c3	0
Anna	c0	1
Carla	c0	1
James	c0	1
James	c1	1
James	c2	1
James	c3	1

allInOneCourse

student	pass
Alba	1
James	1

Figura 3.2: Instancia de la base de datos *Academy*.

La figura 3.2 muestra un ejemplo de instancia de la base de datos *Academy*. Dicha base de datos consta de tres tablas. La tabla *courses* contiene información de los cursos que ofrece una academia. Cada curso tiene un identificador y su nivel. La tabla *registration* contiene información de los estudiantes registrados en cada curso junto con información de si el estudiante ha superado el curso o no. Por último, la tabla *allInOneCourse* contiene información de los estudiantes registrados en el curso intensivo, con un indicador de si lo ha superado o no.

Por otro lado, en la figura 3.3 se define un conjunto de vistas que permiten obtener información de la base de datos. La primera vista, *standard*, completa la información de la tabla *registration* con la información del nivel de cada curso. La vista *basic* selecciona los estudiantes que han superado el curso de nivel básico (nivel 0). La vista *intensive* pretende seleccionar aquellos estudiantes que han superado el curso intensivo o aquellos que han superado los cursos de niveles 1, 2 y 3. Por último, la vista *awards* selecciona aquellos estudiantes que han cursado y superado el curso de nivel básico (nivel 0) y no se encuentran en *intensive*. Sin embargo, en la definición de la vista *intensive* hay un error. Lo veremos más adelante.

La figura 3.4 muestra la respuesta esperada de cada una de las vistas definidas en la figura 3.3. Por ejemplo, es de esperar que la vista *standard* contenga la información de los niveles de los cursos asociados a cada estudiante junto con la información de si los ha superado o no. También es de esperar que la vista *basic* contenga el nombre de los tres estudiantes que han cursado el nivel básico, que son *Anna*, *Carla* y *James*. La respuesta esperada de la vista *intensive* contiene el nombre de los estudiantes que han superado el curso intensivo. Para terminar, la respuesta esperada de la vista *awards* contiene a los estudiantes *Anna* y *Carla*. Sin embargo, la respuesta computada de la consulta `select * from awards;` contiene una única fila, la cual la podemos representar mediante el multiconjunto: $\{\!|(Carla)|\!\}$. Podemos observar que ambas respuestas, la respuesta computada y la respuesta esperada de la vista *awards*, no coinciden; se ha producido un resultado no esperado, lo que indica que existe un error. Sin embargo, no es posible asegurar que el error se encuentra en la definición de la vista *awards*. El

```
create view standard(student, level, pass) as
  select R.student, C.level, R.pass
  from courses C, registration R
  where C.id = R.course;

create view basic(student) as
  select S.student
  from standard S
  where S.level = 0 and S.pass=1;

create view intensive(student) as
  (select A.student
   from allInOneCourse A
   where A.pass=1 )
   union
  (select A1.student
   from standard A1, standard A2, standard A3
   where A1.student = A2.student )
     and A2.student = A3.student
     and A1.level = 1
     and A2.level = 2
     and A3.level = 3 );

create view awards(student) as
  (select student from basic)
  except
  (where student from intensive);
```

Figura 3.3: Vistas para la selección de estudiantes premiados.

standard

student	level	pass
Alba	1	0
Alba	2	1
Anna	0	1
Anna	1	1
Anna	2	1
Anna	3	0
Carla	0	1
James	0	1
James	1	1
James	2	1
James	3	1

basic

student
Anna
Carla
James

intensive

student
Alba
James

awards

student
Anna
Carla

Figura 3.4: Respuesta esperada de las vistas del ejemplo 3.3.

error puede venir de alguna de las relaciones que aparecen en su definición, o de las relaciones usadas por éstas y así sucesivamente.

Definición 3.3.2. *Sea D un esquema de base de datos, d una instancia de base de datos con esquema D y R una relación en D. Decimos que $\mathcal{SQL}(R, d)$ es un* ***resultado inesperado*** *para una relación* R *si $\mathcal{I}(R, d) \neq \mathcal{SQL}(\mathrm{R}, d)$.*

De la definición anterior se deduce que si $\mathcal{I}(R, d) \neq \mathcal{SQL}(R, d)$, existe una fila t tal que $|\mathcal{I}(R, d)|_t \neq |\mathcal{SQL}(R, d)|_t$. La existencia de un resultado inesperado implica la existencia de una *fila errónea* o una *fila perdida* en el resultado de una relación.

Definición 3.3.3. *Decimos que $t \in \mathcal{SQL}(\mathrm{R}, d)$ es una* ***fila errónea*** *si:*

$$|\mathcal{SQL}(\mathrm{R}, d)|_{\mathrm{t}} > 0 \ y \ |\mathcal{I}(R, d)|_{\mathrm{t}} < |\mathcal{SQL}(\mathrm{R}, d)|_{\mathrm{t}}$$

Definición 3.3.4. *Decimos que t es una* ***fila perdida*** *en $\mathcal{SQL}(\mathrm{R}, d)$ si:*

$$|\mathcal{I}(R, d)|_{\mathrm{t}} > 0 \ y \ |\mathcal{I}(R, d)|_{\mathrm{t}} > |\mathcal{SQL}(\mathrm{R}, d)|_{\mathrm{t}}$$

Por ejemplo, la respuesta esperada para la vista *awards* contiene la fila *Anna*, la cual se representa como $|\mathcal{I}(\mathsf{awards}, d)|_{(\mathsf{Anna})} = 1$. Sin embargo, la respuesta computada no incluye esta fila: $|\mathcal{SQL}(\mathsf{awards}, d)|_{(\mathsf{Anna})} = 0$. Por tanto, podemos decir que *(' Anna')* es una fila perdida en la vista *awards*.

Para definir el concepto de relación errónea, definimos previamente un concepto auxiliar, el de *respuesta inferida*.

Definición 3.3.5. *Sea D un esquema de base de datos, d una instancia de D y R una relación definida en D. La* ***respuesta inferida*** *de R con respecto a la instancia d y que denotamos por $\mathcal{E}(R, d)$, se define como:*

1. *Si R es una tabla, $\mathcal{E}(R, d) = d(R)$*

2. *Si R es una vista, $\mathcal{E}(R,d) = \mathcal{E}(Q,d)$, siendo Q la consulta que define la vista R.*

3. *Si R es una consulta y $R_1, \ldots, R_n$ son las relaciones que aparecen en la definición de R, entonces $\mathcal{E}(R,d) = \Phi_R(I_1, \ldots, I_n)$ donde $I_i = \mathcal{I}(R_i, d)$ para $i = 1 \ldots n$. Es decir, $\mathcal{E}(R,d)$ es el resultado de evaluar la expresión Φ_R después de sustituir cada nombre de relación R_i en Φ_R por su respuesta esperada $\mathcal{I}(R_i, d)$ para $i = 1 \ldots n$.*

En el caso de tablas, la respuesta inferida coincide con su instancia. En el caso de vistas, la respuesta inferida es la respuesta inferida de la consulta que define la vista. En el caso de consultas, la respuesta inferida se corresponde con la respuesta computada por el sistema partiendo de la respuesta esperada de las relaciones R_i que aparecen en la consulta.

La diferencia entre la respuesta esperada $\mathcal{I}(R,d)$ y la respuesta inferida $\mathcal{E}(R,d)$ de la relación R con respecto a la instancia d permite definir el concepto de relación errónea, como aquella relación que produce un resultado inesperado aún suponiendo que todas las relaciones de las que depende producen un resultado esperado:

Definición 3.3.6. *Sea D un esquema de base de datos, d una instancia de D y R una relación definida en D. Decimos que R es una* ***relación errónea*** *si $\mathcal{I}(R,d) \neq \mathcal{E}(R,d)$. Decimos que R es* ***correcta*** *en otro caso.*

Por ejemplo, si consideramos la instancia d de la figura 3.2, las relaciones definidas en la figura 3.3 y asumimos que todas las tablas contienen respuestas esperadas[4], es decir, $\mathcal{I}(T,d) = d(T)$ para toda tabla T del esquema de base de datos D, entonces la respuesta inferida de la vista *intensive* es la misma que su respuesta computada $\| \Phi_{intensive} \|_d$:

$$\mathcal{E}(intensive, d) = \Phi_{intensive}(\mathcal{I}(allInOneCourse, d), \mathcal{I}(standard, d)) = \\ \{\!| (Alba), (Anna), (James) |\!\}$$

Este resultado es diferente de la respuesta esperada de la vista *intensive* con respecto a la misma instancia (Figura 3.4), por lo que según la definición 3.3.6 la relación *intensive* es errónea. Su definición correcta es:

[4]Podemos observar que en la figura 3.4 no hemos incluido la respuesta esperada de las instancias de las tablas, asumiendo de forma implícita que el contenido actual de las tablas es el esperado por el usuario. En cualquier caso, también es posible detectar errores en las instancias de las tablas del esquema.

```
create view intensive(student) as
  (select A.student
   from allInOneCourse A
   where A.pass=1 )
   union
  (select A1.student
   from standard A1, standard A2, standard A3
   where A1.student = A2.student
     and A2.student = A3.student
     and A1.level = 1 and A1.pass=1
     and A2.level = 2 and A2.pass=1
     and A3.level = 3 and A3.pass=1 );
```

en lugar de la definición usada en la figura 3.3. Este error es la causa de que el resultado de la vista *awards* sea un resultado inesperado.

La definición 3.3.6 aclara el concepto fundamental de relación errónea. Por contra, no parece práctico usar directamente esta definición en el proceso de depuración, ya que para indicar que la vista R es errónea, se necesitaría comparar $\mathcal{I}(R,d)$ y $\mathcal{E}(R,d)$. Para obtener $\mathcal{E}(R,d)$, nuestra herramienta necesitaría la respuesta esperada $\mathcal{I}(R_i,d)$ para cada relación R_i que aparece en la definición de R, la cual es solo conocida por el usuario. Sin embargo, no parece realista asumir que el usuario conoce el resultado de $\mathcal{E}(R,d)$ para poder determinar si coincide con $\mathcal{I}(R,d)$. Una técnica que requiere tal cantidad de información por parte del usuario resulta muy poco práctica. Así, en lugar de utilizar la definición 3.3.6 en el proceso de depuración, utilizamos el siguiente resultado:

Teorema 3.3.7. *Sea D un esquema de base de datos y d una instancia de D. Sea R una vista definida en el esquema D y $R_1,\ldots,R_n$ las relaciones que aparecen en la consulta que define R. Supongamos que $\mathcal{I}(R_i,d) = \mathcal{SQL}(R_i,d)$ para $i = 1\ldots n$. Entonces, $\mathcal{I}(R,d) \neq \mathcal{SQL}(R,d)$ si y solo si $\mathcal{I}(R,d) \neq \mathcal{E}(R,d)$.*

El teorema permite solventar el problema de tener que preguntar directamente por la respuesta inferida. Sin embargo persiste el problema de tener que preguntar por la respuesta esperada de las relaciones. Este segundo problema se solventará, al menos en parte, en la sección 3.3.3. Mientras tanto, en la siguiente sección, completamos el esquema de depuración basado en las ideas vistas hasta ahora.

3.3.2. Depuración declarativa de vistas SQL con árboles de cómputo

En esta sección definimos una estructura en forma de árbol para representar el mecanismo de cómputo de una consulta SQL. El árbol de cómputo de una consulta se corresponde con su árbol de dependencias sintácticas completado con las respuestas obtenidas por el sistema con respecto a la instancia actual. Cada nodo del árbol está asociado a una relación, ya sea una tabla o una vista. En el caso de que un nodo esté asociado a una vista, éste tendrá tantos hijos como relaciones aparezcan en la definición de la vista.

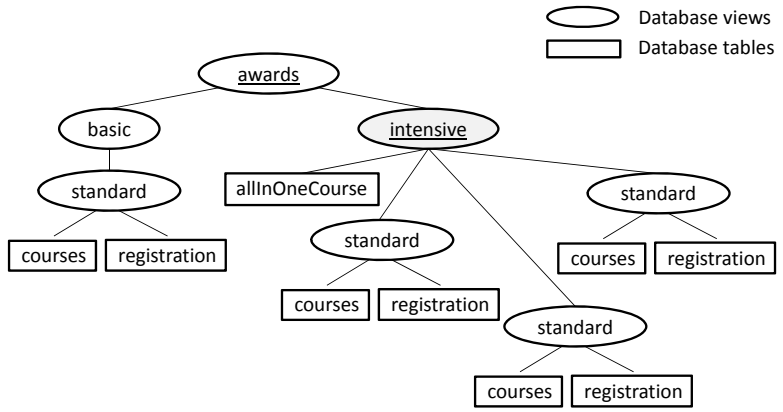


Figura 3.5: Árbol de cómputo de la vista *awards*.

Definición 3.3.8. *Sea D un esquema de base de datos y $\mathcal{V}$ el conjunto de vistas definidas en D. Sea d una instancia de base de datos con esquema D y R una relación definida en D. El* ***árbol de cómputo*** *de la relación R con respecto a la instancia d, denotado por $CT(R, d)$, se define como sigue:*

- *La raíz del árbol es $(R \mapsto \mathcal{SQL}(R, d))$.*
- *Para cada nodo del árbol $N = (R' \mapsto \mathcal{SQL}(R', d))$, se tiene:*
  - *Si R' es una tabla, el nodo N no tiene hijos, se trata de una hoja.*
  - *Si R' es una vista, los hijos de N son los árboles de cómputo de las relaciones que aparecen en la consulta que define la vista R'.*

Aunque en la Definition 3.3.8 se incluye la respuesta computada de las vistas con respecto a la instancia d como parte de los nodos, esta información no afecta a la estructura del árbol. En la práctica, y para conseguir una implementación más eficiente en términos de memoria, solo se calcula cuando el depurador la necesita. Con esta simplificación, el árbol de cómputo de una vista se corresponde con su árbol de dependencias sintácticas en el esquema. En la figura 3.5 mostramos el árbol de cómputo asociado a la vista *awards*.

Una vez construido el árbol de cómputo, el depurador recorre dicho árbol etiquetando los nodos como válidos o no válidos.

Definición 3.3.9. *Sea $T = CT(R, d)$ un árbol de cómputo y $N = (R' \mapsto \mathcal{SQL}(R', d))$ un nodo del árbol T. Decimos que N es un* ***nodo válido*** *si $\mathcal{SQL}(R', d) = \mathcal{I}(R', d)$. Por último, decimos que N es un* ***nodo crítico*** *si N es un nodo no válido y todos sus hijos son válidos.*

En el árbol de cómputo asociado a la vista *awards* de la figura 3.5, los nodos no válidos son los correspondientes a las vistas *awards* e *intensive*. La respuesta esperada

de la vista *awards*, según la figura 3.4, contiene a los estudiantes *Anna* y *Carla*, mientras que la respuesta computada de la consulta select * from awards; contiene únicamente a *Anna*. Lo mismo ocurre con la vista *intensive*; su respuesta esperada según la figura 3.4, contiene a los estudiantes *Alba* y *James*, mientras que la respuesta computada de la consulta select * from intensive; contiene únicamente a los estudiantes *Alba*, *Anna* y *James*. El único nodo crítico del árbol es el correspondiente a la vista *intensive*. En este caso, una fila errónea en la vista *intensive* produce una fila perdida en la vista *awards*.

En general, el proceso de depuración comienza cuando el usuario detecta un resultado inesperado para una vista V. El depurador construye el árbol de cómputo T de la vista V cuyo nodo raíz es no válido y lo recorre realizando preguntas al usuario acerca de la validez del resto de los nodos. Si un nodo contiene una respuesta inesperada, el depurador lo marca como no válido y válido en caso contrario. El proceso finaliza cuando el depurador encuentra un nodo crítico, un nodo no válido con todos sus hijos válidos. En nuestro contexto, un nodo crítico en el árbol se corresponde con una instancia de tabla con datos erróneos o una vista definida de forma incorrecta.

En la publicación [36](A.4) se prueba la consistencia de la técnica utilizada y se demuestra el siguiente resultado de corrección y completitud:

Teorema 3.3.10. *Sea d una instancia del esquema de base de datos D, V una vista definida en D y T el árbol de cómputo de la vista V con respecto a la instancia d. Si la raíz de T es un nodo no válido, entonces:*

- *Completitud. T contiene un nodo crítico.*
- *Corrección. Todo nodo crítico en T se corresponde con una relación errónea.*

Sesión de depuración

Esta propuesta de depuración de vistas SQL basada en árboles de cómputo ha sido implementada en el sistema DES (Datalog Educational System) [101, 100]. La implementación y las instrucciones de uso se encuentran accesibles en la dirección:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/Des`

A continuación mostramos una sesión de depuración de la vista *awards*. El script *academy.sql*, que puede encontrarse en el siguiente enlace:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/Des/academy.sql`

contiene la definición de la base de datos *academy* y el poblado de las tablas. El comando `/process academy.sql` permite cargar la base de datos *academy* en el sistema. Posteriormente, es posible realizar la consulta *select * from awards;*.

```
DES-SQL> /process academy.sql
Info: Batch file processed.
```

standard

student	level	pass
Alba	1	0
Alba	2	1
Anna	0	1
Anna	1	1
Anna	2	1
Anna	3	0
Carla	0	1
James	0	1
James	1	1
James	2	1
James	3	1

basic

student
Anna
Carla
James

intensive

student
Alba
Anna
James

awards

student
Carla

Figura 3.6: Respuesta computada por el sistema.

```
DES-SQL> select * from awards

answer(awards.student) ->
{
  answer('Carla')
}
Info: 1 tuple computed.
```

En la figura 3.6 se muestra la respuesta computada por el sistema de cada una de las vistas de la figura 3.3.

El resultado de la consulta *awards* es un resultado inesperado para el usuario. El comando `/debug_sql` seguido de la vista a depurar se utiliza para comenzar la depuración de dicha vista.

```
DES-SQL> /debug_sql awards
```

El depurador construye el árbol de cómputo de la vista *awards* de la figura 3.5. Posteriormente, se recorre dicho árbol produciéndose la siguiente sesión de depuración:

```
Info: Debugging view 'intensive'.
{
  intensive('Alba'),
  intensive('Anna'),
```

```
  intensive('James')
}
Info: 3 tuples computed.
Input: Is this view valid? (y/n/a) [y]: n
Info: Debugging view 'standard'.
{
  standard('Alba',1,0),
  standard('Alba',2,1),
  standard('Anna',0,1),
  standard('Anna',1,1),
  standard('Anna',2,1),
  standard('Anna',3,0),
  standard('Carla',0,1),
  standard('James',0,1),
  standard('James',1,1),
  standard('James',2,1),
  standard('James',3,1)
}
Info: 11 tuples computed.
Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found: intensive/1.
```

En esta sesión de depuración, la primera pregunta que realiza el depurador es acerca de la validez de la vista *intensive*. La respuesta computada por el sistema no coincide con lo que el usuario espera, así que responde `'n'` y el nodo asociado queda marcado como no válido. En este caso, el depurador visita el primer hijo, asociado a la vista *standard*. La respuesta computada de la vista *standard* es válida y el usuario responde `'y'`. El depurador marca como válidos los cuatro nodos del árbol asociados a la vista *standard*. En este punto, el depurador encuentra un nodo no válido en el árbol con todos sus hijos válidos; el nodo asociado a la vista *intensive* es un nodo crítico. En este ejemplo, el depurador ha necesitado realizar dos preguntas al usuario para encontrar un nodo crítico y por tanto la fuente del error.

Como se puede observar, en esta sesión de depuración se ha evitado preguntar al usuario acerca de la validez de la instancia de las tablas de la bases de datos, confiando en que dichas instancias no son erróneas. Se trata de los nodos hoja del árbol de cómputo correspondientes a las tablas *courses*, *registration* y *allInOneCourse*. No obstante, el depurador ofrece la posibilidad de tratar estos nodos como todos los demás, es decir, no dar por sentado que no hay errores en la instancia de las tablas y que sea el usuario el que proporcione esa información. Para depurar la vista *awards* asumiendo que las instancias de tablas pueden contener datos erróneos se utiliza el siguiente comando:

```
DES-SQL> /debug_sql awards trust_tables(no)
```

Aunque en este ejemplo el depurador utiliza una estrategia *Divide & Query* [105] para recorrer el árbol de cómputo, se podrían implementar otras estrategias para localizar nodos críticos, como las estudiadas en [106]. La estrategia *Divide & Query* [105] proporciona una mayor eficiencia que el resto, en el sentido de que se minimiza el número de nodos visitados antes de encontrar un nodo crítico en el árbol, y por tanto, se minimiza el número de preguntas al usuario. En realidad, el número de preguntas no es un problema serio ya que incluso en sistemas grandes de vistas, que no suelen alcanzar las 100 vistas, el depurador encuentra el error con un número de preguntas mucho menor. El motivo es que en cada paso de depuración se van descartando partes del árbol, reduciéndose así el número de preguntas. Este problema se da en lenguajes como Java, dónde los árboles suelen tener miles de nodos, y en este caso se utilizan estrategias más avanzadas. La complejidad de las preguntas si puede resultar un problema, ya que es posible encontrarse con situaciones en las que el depurador solicita al usuario decidir acerca de la validez de instancias con miles de filas. En la siguiente sección se propone una mejora del proceso de depuración que soluciona en muchos casos este problema.

3.3.3. Depuración de respuestas perdidas e incorrectas

En esta sección pretendemos mejorar la técnica de depuración presentada en la sección 3.3.2 permitiendo incorporar, en el proceso de depuración, información acerca del tipo de error (fila perdida o fila errónea) que se ha producido. Esta información es utilizada durante el proceso de depuración con el objetivo de focalizar la búsqueda del error hacia aquellas partes de las consultas que podrían ser el origen del error, es decir, aquellas que han producido/perdido la fila errónea/perdida. Como consecuencia de lo anterior, se reduce el número de preguntas al usuario antes de localizar la fuente del error y se simplifican las preguntas que el depurador realiza en el sentido de que son más fáciles de contestar, al reducirse el número de filas que se deben considerar [90].

El ejemplo de la sección anterior, en el que la vista *awards* produce un resultado inesperado, permite motivar esta propuesta. Supongamos que durante el proceso de depuración de la vista *awards*, el usuario indica que la fila *('Anna')* es una fila perdida en el resultado de la vista *awards*. Al estar esta definida mediante la operación except, este error se puede producir por tres causas: (a) la vista *awards* está definida de forma incorrecta, (b) la fila *('Anna')* es también una fila perdida en el resultado de la vista *basic*, (c) la fila *('Anna')* es una fila errónea en el resultado de la vista *intensive*. En este caso particular, antes de realizar ninguna pregunta al usuario, el depurador podría comprobar que la fila *Anna* está contenida en la respuesta computada de la vista *intensive* y en la respuesta computada de la vista *basic*. Por tanto, solo puede ocurrir que la vista *awards* esté definida de forma incorrecta o que la fila *('Anna')* sea errónea en *intensive*. Como consecuencia, se evitaría preguntar sobre el contenido de la vista *basic* como se hacía en la sesión de depuración de la sección 3.3.2. Por otro lado, en lugar de preguntar por el contenido completo de la vista *intensive*, se podría preguntar acerca de la validez de las filas en las que interviene *Anna* en *intensive*.

Aunque la técnica que aquí presentamos está basada en depuración declarativa, no se utiliza de forma explícita el árbol de cómputo asociado a una vista en el proceso de depuración. Las relaciones existentes entre los nodos del árbol y las respuestas

Código 1 debug(V)

```
Entrada: V: Nombre de una vista
Salida: Lista de relaciones erróneas
 1: A := askOracle(all V)
 2: if A ≡ no or A ≡ missing(t) or A ≡ wrong(t)
 3:    Valid := true
 4:    P := initialSetOfClauses(V, A)
 5:    while getBuggy(P)=[ ] do
 6:       LE := getUnsolvedEnquiries(P)
 7:       E := chooseEnquire(LE)
 8:       A := askOracle(E)
 9:       Valid := checkAnswer(A)
10:       if Valid   P := P ∪ processAnswer(E,A)
11:    end while
12:    L := getBuggy(P)
13: else
14:    L := [ ]
15: end if
16: return L
```

proporcionadas por el usuario se representan mediante un conjunto de cláusulas lógicas que dan lugar a un programa lógico. Los átomos del cuerpo de las cláusulas se corresponden con preguntas que el usuario debe contestar para detectar una relación incorrecta. El programa lógico resultante se ejecuta seleccionando en cada paso un átomo del cuerpo de las cláusulas sin resolver que da lugar a una pregunta al usuario. El proceso se repite hasta que el depurador encuentra una relación errónea en el sentido de la definición 3.3.6.

A continuación presentamos el algoritmo que define nuestra técnica de depuración.

Algoritmo de depuración

La función *debug* (Código 1) describe el esquema general del algoritmo. Cuando el resultado de una vista V no es el esperado, el depurador ejecuta la función *debug* con dicha vista como único parámetro de entrada. La salida de la función es una lista de relaciones de la base de datos consideradas erróneas.

Nuestro algoritmo utiliza algunas funciones auxiliares cuyo código no se incluye aquí por su simplicidad. Las funciones *getSelect* y *getFrom* devuelven las diferentes secciones de una consulta SQL. Si una consulta Q no tiene sección group by, la función *getGroupBy(Q)* devuelve una lista vacía de atributos, por lo que la expresión booleana *getGroupBy(Q)*=[] toma el valor *true*. La función *getRelations(R)* devuelve la lista de todas las relaciones que aparecen en la definición de R y puede ser aplicada a consultas, a tablas y a vistas. Si R es una tabla, entonces *getRelations(R)* = $\{R\}$. Si R es una consulta, entonces *getRelations(R)* es un conjunto de relaciones que aparecen en la definición de la consulta. Si R es una vista, entonces *getRelations(R)* = *getRela-*

```
buggy(awards)      :- state(all(awards),nonvalid),
                      state(all(basic),valid),
                      state(all(intensive),valid).

buggy(basic)       :- state(all(basic),nonvalid),
                      state(all(standard),valid).

buggy(intensive)   :- state(all(intensive),nonvalid),
                      state(all(allInOneCourse),valid), state(all(standard),valid).
    ...                  ...
```

Figura 3.7: Lista inicial de cláusulas que constituye el programa lógico P asociado a la depuración de la vista *awards*.

tions(Q), siendo Q la consulta que define la vista R. La función *generateUndefined(R)* genera una nueva fila cuya aridad es el número de atributos de la relación R (tabla o vista) y que contiene valores indefinidos $(\perp, \ldots, \perp)$. La función *checkAnswer(A)* devuelve el valor *true* si el parámetro de entrada A es de la forma *yes*, *no*, *missing(t)* o *wrong(t)*, devolviendo el valor *false* en otro caso.

La primera pregunta que realiza el depurador al usuario es acerca del tipo de error que se ha producido (Código 1, Linea 1). Como alternativas, la respuesta A del usuario puede ser simplemente *no*, para indicar que la respuesta computada de la vista V es inesperada, o puede ser una respuesta que proporcione algo más de información, como por ejemplo *wrong(t)* o *missing(t)*, indicando que la fila t es errónea o perdida respectivamente. Posteriormente, el depurador llama a la función *initialSetofClauses*(Código 1, Linea 4) para generar la lista inicial de cláusulas de Horn que representa el árbol de cómputo de la vista a depurar. Esta lista de cláusulas constituye un programa lógico P que irá creciendo con nuevas cláusulas a medida que se avanza en el proceso de depuración.

En nuestro ejemplo, el proceso de depuración se inicia con la llamada *debug(awards)*. El depurador realiza la primera pregunta al usuario acerca del tipo de error. El usuario indica que se ha perdido la fila *('Anna')*, en cuyo caso el parámetro A toma el valor inicial *missing(('Anna'))*. La figura 3.7 muestra (parcialmente) la lista inicial de cláusulas que representa el árbol de cómputo de la vista *awards*. Como se puede observar, existe una correspondencia directa entre estas cláusulas y el árbol de cómputo de la figura 3.5.

El propósito del bucle principal de la función *debug* (Código 1, Líneas 5-11) es añadir información al programa lógico P en forma de nuevas cláusulas, hasta que se pueda inferir una vista errónea. La función *getBuggy* devuelve una lista con todas las relaciones R tal que el objetivo *buggy(R)* puede ser probado a partir del programa lógico P. Los átomos en el cuerpo de las cláusulas del programa P representan *cuestiones* que simbolizan preguntas que el depurador puede hacer al usuario. En cada iteración del bucle se selecciona una cuestión que no ha sido resuelta aún (Líneas 6-7). En la Línea 8 el depurador realiza una pregunta al usuario y la respuesta se procesa

Código 2 initialSetofClauses(V, A)

Entrada: V: Nombre de una vista, A: respuesta del usuario
Salida: Conjunto de cláusulas de Horn

```
1: P := ∅
2: P := initialize(V)
3: P := P ∪ processAnswer((all V), A)
4: return P
```

initialize(R)
Entrada: R: nombre de una relación
Salida: Conjunto de cláusulas de Horn

```
1: P := createBuggyClause(R)
2: for each R_i in getRelations(R) do
3:    P := P ∪ initialize(R_i)
4: end for
5: return P
```

createBuggyClause(R)
Entrada: R: nombre de una relación
Salida: Una cláusula de Horn

```
1: [R_1, ..., R_n] := getRelations(R)
2: return { buggy(R)← state((all R), nonvalid),
                      state((all R_1), valid), ..., state((all R_n), valid)). }
```

en la línea 10.

En la Línea 4 del Código 1 se llama a la función *initialSetofClauses*. El primer parámetro de la función es la vista inicial V que ha producido la respuesta inesperada y el segundo parámetro A representa la información que proporciona el usuario. En este punto, el valor de A es uno de los siguientes: *no* (indicando que la respuesta es inesperada para el usuario), *wrong(t)* (indicando que el resultado de la vista contiene una fila t errónea) o *missing(t)* (indicando que se esperaba una fila t en el resultado de la vista). La salida de esta función es un conjunto de cláusulas (a las que llamaremos cláusulas *buggy*) que representan las relaciones lógicas que definen las posibles relaciones erróneas del esquema de base de datos. Inicialmente se crea un conjunto vacío de cláusulas y se llama a la función *initialize* (Código 2, Línea 2). Esta función recorre recursivamente todas las relaciones que intervienen en la definición de la vista inicial V, llamando a la función *createBuggyClause* con V como parámetro de entrada. La función *createBuggyClause* añade una nueva cláusula buggy que establece las condiciones que se debe cumplir para afirmar que la vista V es incorrecta: V debe ser *no válida* y todas las relaciones que aparecen en su definición deben ser *válidas*.

En el programa P aparecen dos tipos de predicados. El predicado *buggy* y el predi-

cado *state*. El predicado *buggy* define cuándo la relación que aparece como parámetro es errónea. En el programa P los átomos buggy son siempre términos Prolog. Así, si se puede probar el término *buggy(R)* a partir del programa P, es posible afirmar que la relación R es errónea en el sentido de la definición 3.3.6.

En la Línea 3 del Código 2, la función *processAnswer* incorpora al programa P la información proporcionada por el usuario mediante el predicado *state*. El predicado *state* permite representar la información proporcionada por el usuario acerca de la validez/no-validez de las cuestiones que simbolizan las preguntas que puede hacer el depurador. El primer parámetro del predicado *state* representa una cuestión E que simboliza una pregunta y el segundo parámetro representa la respuesta A del usuario a dicha pregunta.

La siguiente definición describe las posibles cuestiones, las preguntas asociadas con sus posibles respuestas y una medida $\mathcal{C}$ de la complejidad de las preguntas. Como puede observarse, en esta definición y en lo sucesivo, utilizaremos la notación $\mathcal{SQL}$(R) en lugar de $\mathcal{SQL}$(R, d).

Definición 3.3.11. *Sean* R *y* R' *dos relaciones definidas en el esquema D, d una instancia de base de datos con esquema D y* s *una fila con el mismo esquema que la relación* R. *Todas las preguntas que puede formular el depurador se pueden representar mediante una de las siguientes cuestiones:* (all R), (s $\in$ R) *o* (R'$\subseteq$ R). *Cada cuestión* E *simboliza una pregunta específica y tiene asociado un conjunto de respuestas posibles y una complejidad $\mathcal{C}(E)$:*

- *Sea* E $\equiv$ (all R) *y* S = $\mathcal{SQL}$(R). *La cuestión* E *simboliza la pregunta* "Is S the intended answer for R?", *(*"¿Es S la respuesta esperada de la relación R?"*). La respuesta asociada a la cuestión* E *puede ser* yes *o* no. *En el caso de que la respuesta del usuario sea* no*, el depurador pregunta acerca del tipo de error, permitiendo al usuario indicar si hay una fila t errónea o perdida en el resultado de la relación* R. *Si el usuario es capaz de proporcionar esta información, la respuesta asociada a la cuestión* E *cambia de* no *a* wrong(t) *o* missing(t) *dependiendo del tipo de error. La complejidad se define como $\mathcal{C}(E) = |S|$, donde $|S|$ representa el número de filas del multiconjunto S.*

- *Sea* E $\equiv$ (R'$\subseteq$ R) *y* S = $\mathcal{SQL}$(R'). *La cuestión* E *simboliza la pregunta* "Is S included in the intended answer for R?" *(*"¿Está S incluido en la respuesta esperada de la relación R?"*). Como en el caso anterior, la respuesta asociada a la cuestión* E *puede ser* yes *o* no*. En el caso de que la respuesta del usuario sea* no*, el depurador permite al usuario señalar una fila errónea* t $\in S$*. En este caso, la respuesta asociada a la cuestión* E *cambia de* no *a* wrong(t)*. La complejidad se define como en el caso anterior $\mathcal{C}(E) = |S|$.*

- *Sea* E $\equiv$ (s $\in$ R). *La cuestión* E *simboliza la pregunta* "Does the intended answer for R include a tuple s?" *(* "¿Está la fila s incluida en en la respuesta esperada de la relación R?"*). La respuesta asociada a la cuestión* E *puede ser* yes *o* no*. En este caso el depurador no solicita más información al usuario. En este caso $\mathcal{C}(E) = 1$, ya que esta cuestión involucra una única fila.*

En el caso de que el resultado de una relación contenga una fila errónea, el usuario puede indicar cuál es. En el caso de que el usuario detecte que falta una fila en el

Código 3 processAnswer(E,A)

Entrada: E: cuestión, A: respuesta asociada a la cuestión
Salida: Conjunto de nuevas cláusulas

```
1: if A ≡ yes
2:    P := {state(E,valid).}
3: else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t)
4:    P := {state(E,nonvalid).}
5: end if
6: if E ≡ (s ∈ R) and A ≡ yes
7:    P:= P ∪ processAnswer((all R),missing(s))
8: else if E ≡ (V ⊆ R) and (A ≡ wrong(s) or A ≡ no)
9:    P:= P ∪ processAnswer((all R), A)
10: else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t))
11:    Q := SQL query defining V
12:    P := P ∪ slice(V,Q,A)
13: end if
14: return P
```

resultado de una relación, el depurador permite al usuario registrar información del valor de uno o varios de los atributos de la fila que falta, construyéndose una fila parcial o total. La respuesta del usuario *yes* está asociada al estado *valid*, mientras que la respuesta *no* está asociada al estado *nonvalid*.

Si el programa lógico P contiene al menos un hecho Prolog de la forma *state(e, valid)* o *state(e, nonvalid)*, decimos que el átomo *state(e,s)* está resuelto. La función *getUnsolvedEnquiries* (Código 1, Línea 6) devuelve una lista con las cuestiones asociadas a los átomos sin resolver que aparecen en el cuerpo de las cláusulas del programa P. La función *chooseEnquiry* (Código 1, Línea 7) elige una de esas cuestiones E en base a algún criterio predefinido. En nuestra implementación, el criterio utilizado se basa en la complejidad de E, seleccionando la cuestión con menor complejidad, es decir con menor valor de $\mathcal{C}(E)$. En cualquier caso, es posible definir y utilizar otros criterios sin que afecte a los resultados teóricos. Una vez que la cuestión ha sido elegida, se llama a la función *askOracle* (Código 1, Línea 8) formulando una pregunta al usuario y registrando su respuesta.

La función *processAnswer* (llamada en la Línea 10 del Código 1) se encarga por un lado de registrar la respuesta del usuario con respecto a la cuestión E en forma de hechos Prolog y por otro lado de procesar dicha respuesta añadiendo nuevas cláusulas al programa lógico P. Estas nuevas cláusulas permiten al depurador realizar preguntas más fáciles de contestar para el usuario ya que los átomos que aparecen en el cuerpo de estas cláusulas implican cuestiones de menor complejidad. Además de esta ventaja, las nuevas cláusulas permiten inferir que una relación es errónea realizando un número menor de preguntas al usuario. Esto se debe a que la función *processAnswer* analiza la estructura de las consultas de forma que las nuevas cláusulas no involucren preguntas al usuario acerca de relaciones que no pueden ser la causa del error.

El código de la función *processAnswer* se encuentra descrito en Código 2. Las pri-

meras líneas (1-5) se encargan de registrar la respuesta A asociada a la cuestión E proporcionada por el usuario. En nuestro ejemplo se añade el hecho *state(all(awards), nonvalid)* al programa. El resto del código distingue casos dependiendo de la estructura de E y la respuesta asociada A.

- Si la cuestión es de la forma *(s ∈ R)* y la respuesta es *yes*, significa que $s \in \mathcal{I}(R)$. La cuestión *(s ∈ R)* está asociada a un átomo de la forma *state((s ∈ I(R)), nonvalid)* en el cuerpo de una cláusula añadida al programa P por la función *missingBasic*. Esto se produce cuando el depurador comprueba que la fila s no encaja con ninguna de las filas en la respuesta computada de la relación R (Code 4, line 8). Entonces, es posible afirmar que s es una fila perdida en la relación R, llamándose a la función *processAnswer* de forma recursiva con los parámetros *(all R)* y *missing(s)*.

- Si la cuestión es de la forma *(R'⊆ R)* y la respuesta es *wrong(s)* o *no* , es posible asegurar que la fila s es también errónea en la relación R (Línea 9), llamándose a la función *processAnswer* de forma recursiva con los parámetros *(all R)* y *wrong(s)*.

- Si la cuestión es de la forma *(all V)* y la respuesta es *wrong(t)* o *missing(t)*, se llama a la función *slice* (Línea 12). Esta función analiza la consulta Q para producir, si es posible, nuevas cláusulas que permitirán al depurador detectar relaciones erróneas realizando preguntas más simples al usuario. La función *slice* se encuentra definida en el Código 3.

Código 4 slice(V,Q,A)

Entrada: V: Nombre de una vista, Q: consulta SQL, A: respuesta
Salida: un conjunto de cláusulas

```
1: P := ∅;    S= SQL(Q);    S1= SQL(Q1);    S2= SQL(Q2)
2: if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
      (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2)
3:     S1= SQL(Q1);    S2= SQL(Q2)
4:   if |S1|t = |S|t   P:= P ∪ slice(V, Q1, A)
5:   if |S2|t = |S|t   P:= P ∪ slice(V, Q2, A)
6: else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2
7:     S1= SQL(Q1);    S2= SQL(Q2)
8:   if |S1|t = |S|t   P:= P ∪ slice(V, Q1, A)
9:   if Q ≡ Q1 except Q2 and t ∈⊥ S2   P :=P∪ slice(V,Q2,wrong(t))
10: else if basic(Q) and groupBy(Q)=[]
11:   if A ≡ missing(t)   P := P ∪ missingBasic(V, Q, t)
12:   else if A ≡ wrong(t)   P := P ∪ wrongBasic(V, Q, t)
13: end if
14: return P
```

La función *slice* recibe como parámetros de entrada una vista V, una subconsulta Q y la respuesta A. Inicialmente, Q es la consulta que define la vista V y A es la

respuesta proporcionada por el usuario, pero esta situación puede ir cambiando en las llamadas recursivas. A continuación mostramos en qué casos la función *slice* genera nuevas cláusulas y con qué estructura.

1. La vista V está definida por la consulta $\mathrm{Q} \equiv \mathrm{Q}_1$ intersect [all] Q_2 y el usuario indica que en el resultado de la vista V falta una fila t, es decir $|\mathcal{I}(V)|_t > |\mathcal{SQL}(V)|_t$.

   En este caso, si la subconsulta Q_1 produce tantas copias de t como la vista V, es posible afirmar que la subconsulta Q_1 es la causa del escaso número de copias de la fila t en el resultado de la vista V (Línea 4) . Esto permite inferir que la vista V es incorrecta preguntando al usuario únicamente acerca de la validez de las relaciones que aparecen en la subconsulta Q_1 obviando aquellas preguntas al usuario que involucren las relaciones que aparecen en la subconsulta Q_2. Y análogamente para la subconsulta Q_2 (Línea 5).

2. La vista V está definida por la consulta $\mathrm{Q} \equiv \mathrm{Q}_1$ union [all] Q_2 y el usuario indica que el resultado de la vista V contiene una fila errónea t, es decir $|\mathcal{I}(V)|_t < |\mathcal{SQL}(V)|_t$.

   En este caso, si la consulta Q_1 produce tantas copias de t como la vista V, es posible afirmar que la subconsulta Q_1 es la causa del excesivo número de copias de la fila t en el resultado de la vista V. Esto permite inferir que la vista V es incorrecta preguntando al usuario únicamente acerca de la validez de las relaciones que aparecen en la subconsulta Q_1 obviando aquellas preguntas al usuario que involucren las relaciones que aparecen en la subconsulta Q_2. Y análogamente para la subconsulta Q_2.

3. La vista V está definida por la consulta $\mathrm{Q} \equiv \mathrm{Q}_1$ except [all] Q_2 y el usuario indica que en el resultado de la vista V falta una fila t, es decir $|\mathcal{I}(V)|_t > |\mathcal{SQL}(V)|_t$.

   En este caso (Línea 6), si la consulta Q_1 produce tantas copias de t como la vista V (Línea 8), es posible afirmar que la subconsulta Q_1 es la responsable del escaso número de copias de la fila t en el resultado de la vista V. Como en los casos anteriores, es posible inferir que la relación V es errónea preguntando únicamente por la validez de las relaciones involucradas en la consulta Q_1. Además, para el caso particular de $\mathrm{Q} \equiv \mathrm{Q}_1$ except Q_2, es decir que el resultado de la vista V se defina como la diferencia de conjuntos (Línea 9), si la fila t está en el resultado de la consulta Q_2, es posible afirmar que dicha consulta es la causa del escaso número de copias de t en V. Como en los casos anteriores, es posible inferir que la relación V es errónea preguntando únicamente por la validez de las relaciones involucradas en la consulta Q_2. Observar que la llamada recursiva cambia la respuesta de *missing(t)* a *wrong(t)*.

4. La vista V está definida mediante una consulta básica y el usuario indica que el resultado de la vista V contiene una fila t errónea/perdida.

   En este caso, se llama a las funciones *missingBasic* o *wrongBasic* dependiendo de la forma de A. Las funciones *missingBasic* y *wrongBasic* pueden añadir

nuevas cláusulas que permiten al sistema inferir relaciones erróneas realizando preguntas al usuario más fáciles de contestar.

La función *missingBasic*, descrita en el Código 4, es llamada (Código 3, Línea 11) cuando el parámetro A es *missing(t)*. Los parámetros de entrada son la vista

Código 5 missingBasic(V,Q,t)

Entrada: V: nombre de una vista , Q: consulta, t: fila
Salida: Conjunto de nuevas cláusulas

```
 1: P := ∅;    S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
 2: if t ∉⊥ S
 3:   for (R AS S) in (getFrom(Q)) do
 4:     s = generateUndefined(R)
 5:     for i=1 to length(getSelect(Q)) do
 6:       if t(i) ≠⊥ and member(getSelect(Q),i) = S.A, A attrib.,  s.A = t(i)
 7:     end for
 8:     if s ∉⊥ SQL(R)
 9:       P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
10:     end if
11:   end for
12: end if
13: return  P
```

V, una consulta Q, y la fila perdida t. Observar que la consulta Q es en general una componente de la consulta que define la vista V. Para cada relación R con alias S que aparece en la sección from de la consulta Q, se construye una fila s tal que $t(S.A) = s(R.A)$ con $t(S.A) \neq \perp$, siendo $S.A$ una expresión de atributo definida en la sección select de la consulta Q (Líneas 5 - 7). Posteriormente, si la respuesta computada de la relación R no contiene ninguna fila que encaje con s en sus atributos definidos (Línea 8) es posible afirmar que no es posible obtener la fila t en el resultado de V a partir de la relación R. En este caso se añade al programa P una nueva cláusula buggy (Línea 9) indicando que si el usuario responde *no* a la pregunta *"Does the intended answer for R include a tuple matching s?"* (*"¿Está la fila s incluida en la respuesta esperado de la relación R?"*), entonces la vista V es una relación errónea.

La implementación de la función *wrongBasic* se encuentra definida en el Código 5. Los parámetros de entrada son la vista V, un consulta Q, y la fila t. En la línea 1, se crea un conjunto vacío de cláusulas. En la línea 2, la variable F representa la lista de todas las relaciones que aparecen en la sección from de la consulta Q. Para cada relación $R_i \in F$ (Líneas 4 - 7), se define una vista auxiliar V_i en el esquema de base de datos mediante la función *relevantTuples* (Línea 6). El resultado de esta vista auxiliar contiene solo aquellas filas de la relación R_i que contribuyen a producir la fila errónea t en V. Finalmente, se crea una nueva cláusula buggy para la vista V (Línea 8) indicando que la relación V es errónea si la respuesta del usuario a la pregunta asociada a cada una de las cuestiones del tipo $V_i \subseteq \mathrm{R}_i$ es *yes* para $i \in \{1 \ldots n\}$.

Código 6 wrongBasic(V,Q,t)

Entrada: V: el nombre de una vista, Q: una consulta, t: una fila
Salida: Conjunto de nuevas cláusulas.

```
1: P := ∅
2: F := getFrom(Q)
3: N := length(F)
4: for i=1 to N do
5:    R_i as S_i := member(F,i)
6:    relevantTuples(R_i,S_i,V_i, Q, t)
7: end for
8: P := P ∪ { (buggy(V) ← state((V_1 ⊆ R_1), valid), ..., state((V_n ⊆ R_n), valid).) }
9: return P
```

Código 7 relevantTuples(R_i,R',V,Q,t)

Entrada: R_i: relation, R': alias,
V: new view name, Q: Query, t: tuple
Salida: A new view in the database schema

```
1: Let A_1, ..., A_n be the attributes defining R_i
2: SQL(create view V as
     (select R_i.A_1, ..., R_i.A_n from R_i)
        intersect all
     (select R'.A_1, ..., R'.A_n from getFrom(Q)
     where getWhere(Q) and eqTups(t,getSelect(Q))))
```

eqTups(t,s)
Entrada: t,s : tuples
Salida: SQL condition

```
1: C := true
2: for i=1 to length(t) do
3:    if t(i) ≠ ⊥
4:     C:= C AND t(i)=s(i)
5: end for
6: return C
```

En [37](B.1) puede encontrarse el código del resto de funciones relevantes y la demostración de los siguientes resultados acerca de la corrección y completitud de esta propuesta.

Teorema 3.3.12. *Corrección.*
Sea R una relación y L la lista devuelta por la llamada debug(R) *(definida en el Código 1). Si el usuario responde correctamente a las preguntas planteadas por el depurador, entonces toda relación contenida en L es errónea (de acuerdo con la definición 3.3.6).*

Teorema 3.3.13. *Completitud.*
Sea R una relación y A el valor devuelto por la función askOracle(all R) *en la línea 1 del Código 1. Si A es de la forma* no, wrong(t) *o* missing(t)*, entonces la llamada* debug(R) *(definida en el Código 1) devuelve una lista L de relaciones no vacía.*

Sesión de depuración

El algoritmo que hemos descrito para la depuración de vistas SQL ha sido implementada en el sistema DES (Datalog Educational System) [101, 100]. La implementación y las instrucciones de uso se encuentran accesibles en la dirección:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/DesSQL`

A continuación mostramos una sesión de depuración para la vista *awards* de nuestro ejemplo conductor. El depurador comienza cuando el usuario detecta que el estudiante *'Anna'* no se encuentra en la lista de estudiantes producida por *awards*. El comando `/debug_sql` seguido de la vista a depurar se utiliza para comenzar la depuración de dicha vista.

```
DES-SQL> /debug_sql awards

Info: Debugging view 'awards'.
{
  1 - awards('Carla')
}
Input: Is this the expected answer for view 'awards'?
                          (y/n/m/mT/w/wN/a/h) [n]: m'Anna'

Info: Debugging view 'intensive'.
Input: Should 'intensive' include a tuple of the form 'Anna'?
                          (y/n/a) [y]: n

Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form
                                              'Anna,1,1'?
                          (y/n/a) [y]: y

Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form
                                              'Anna,2,1'?
                          (y/n/a) [y]: y

Info: Debugging view 'standard'.
Input: Should 'standard' include a tuple of the form
                                              'Anna,3,0'?
                          (y/n/a) [y]: y

Info: Buggy relation found: intensive
```

En esta sesión de depuración queremos destacar la simplicidad de las preguntas que realiza el depurador. En primer lugar, el usuario indica que *('Anna')* es una fila perdida en la vista *awards* (*m'Anna'*). Posteriormente indica que la vista *intensive* no debería incluir la fila *('Anna')*. El resto de las preguntas son muy sencillas de contestar para el usuario ya que son acerca de la validez de filas con *'Anna'* en la vista *standard*. El usuario responde que las tres filas son válidas en la vista *standard*, por lo que el depurador finaliza el proceso marcando *intensive* como una vista errónea.

Cuando el usuario indica que *('Anna')* es una fila errónea, el depurador ejecuta la función *processAnswer(all(awards),missing(('Anna')))*, la cual llama a la función *slice(awards,* Q_1 except Q_2*, missing(('Anna')))* (Código 2, Línea 12). El depurador

comprueba que Q_2 produce *('Anna')* (Código 3, Linea 9), llamando recursivamente a *slice(awards, Q_2, wrong(('Anna')))* con $Q_2 \equiv$ select student from intensive. Como la consulta Q_2 es una consulta básica, el depurador llama a la función *wrongBasic(awards, Q_2, ('Anna'))* (Código 3, Línea 12). La función *wrongBasic* crea una vista auxiliar que selecciona aquellas filas de *intensive* que producen la fila errónea *('Anna')* (función *relevantTuples* en el Código 7):

```
create view intensive_slice(student) as
    ( select * from intensive )
     intersect all
    ( select * from intensive I where I.student ='Anna')
```

Finalmente, se añade la siguiente cláusula al programa P (Código 5, Línea 8):

```
buggy(awards) :- state(subset(intensive_slice,intensive),valid).
```

donde subset(intensive_slice,intensive) representa la cuestión $E \equiv$*(intensive_slice* $\subseteq$ *intensive*).

El depurador selecciona el átomo del cuerpo de esta cláusula, ya que la vista *intensive_slice* contiene una única fila, planteando la segunda pregunta al usuario. La llamada a la función *askOracle* devuelve *wrong(('Anna'))* ya que el usuario responde *'no'*. Después se llama a la función *processAnswer(subset(intensive_slice,intensive), wrong(('Anna')))* la cual llama recursivamente a *processAnswer(all(intensive),wrong(('Anna')))*. La siguiente llamada es *slice(intensive, Q, wrong(('Anna')))*, siendo $Q \equiv Q_3$ union Q_4 la definición de la vista *intensive* (Figura 3.3). El depurador comprueba que solo la consulta Q_4 produce la fila *('Anna')* llamando a la función *slice(intensive, Q_4, wrong(('Anna')))*. Posteriormente se produce la llamada *wrongBasic(intensive, Q_4, ('Anna'))* ya que la consulta Q_4 es una consulta básica. La función *relevantTuples* es llamada tres veces, una por cada aparición de la vista *standard* en la sección from de Q_4, creándose tres nuevas vistas auxiliares:

```
create view standard_slice_1(student, level, pass) as
  ( select R.student, C.level, R.pass from standard as R )
  intersect all
  ( select A1.student, A1.level, A1.pass
      from standard as A1, standard as A2, standard as A3
      where (A1.student = A2.student
          and A2.student = A3.student
          and A1.level = 1 and A2.level = 2 and A3.level = 3)
          and A1.student = 'Anna');
```

```
create view standard_slice_2(student, level, pass) as
  ( select R.student, C.level, R.pass from standard as R )
 intersect all
  ( select A2.student, A2.level, A2.pass
      from standard as A1, standard as A2, standard as A3
      where (A1.student = A2.student
          and A2.student = A3.student
          and A1.level = 1 and A2.level = 2 and A3.level = 3)
          and A1.student = 'Anna');
```

```
create view standard_slice_3(student, level, pass) as
  ( select R.student, C.level, R.pass from standard as R )
 intersect all
  ( select A3.student, A3.level, A3.pass
      from standard as A1, standard as A2, standard as A3
      where (A1.student = A2.student
          and A2.student = A3.student
          and A1.level = 1 and A2.level = 2 and A3.level = 3)
          and A1.student = 'Anna');
```

y añadiéndose la siguiente cláusula al programa P (Código 5, Línea 8).

```
buggy(intensive) :-  state(subset(standard_slice_1,standard),valid),
                     state(subset(standard_slice_2,standard),valid),
                     state(subset(standard_slice_3,standard),valid).
```

Posteriormente, el depurador selecciona cada uno de estos átomos dando lugar a las tres últimas preguntas, para las que el usuario responde *yes*. Finalmente, el algoritmo termina en el momento en que el objetivo buggy(intensive) tiene éxito con respecto al programa P.

3.4. Conclusiones

En este capítulo se han estudiado dos de las actividades más importantes dentro del proceso se prueba de software escrito en lenguaje SQL; se trata de la generación de casos de prueba y la depuración de consultas. Se ha estudiado el caso particular y a la vez muy habitual de las consultas correlacionadas, lo que supone una novedad con respecto a trabajos relacionados. Se trata de consultas que acceden tanto a tablas como a otras vistas previamente definidas en la base de datos.

El proceso de generación de casos de prueba para una consulta SQL no es una tarea fácil, ya que no solo es necesario tener en cuenta la semántica de la consulta, sino que en él intervienen otros muchos factores representados en el esquema, como son

las condiciones de integridad, las dependencias entre relaciones, etc. Como principal aportación, y en base a todos estos factores, se define un algoritmo para la construcción de un conjunto de fórmulas de primer orden para la consulta bajo prueba, las cuales representan las condiciones que debe cumplir una instancia de la base de datos para ser considerada un caso de prueba con cobertura de predicado. Posteriormente dichas fórmulas son traducidas a un programa lógico con restricciones (CLP) y a un objetivo inicial. Las sustituciones de variables obtenidas al resolver este objetivo representan una instancia de la base de datos que constituye un caso de prueba.

Las ideas teóricas aquí presentadas se plasman en un prototipo inicial de generador de casos de prueba desarrollado en el sistema DES. En este prototipo, las fórmulas previamente generadas son traducidas al lenguaje de restricciones de dominios finitos de SICStus Prolog. Se tratan las características más habituales en las consultas SQL, permitiendo operaciones de UNION e INTERSECCIÓN, consultas correlacionadas, subconsultas en la sección WHERE y consultas agrupadas. La eficiencia del prototipo no depende tanto del tamaño de la instancia, ni del número de vistas involucradas en el proceso de generación, sino principalmente de la complejidad de las fórmulas generadas y la potencia del resolutor utilizado. Hemos podido constatar que la eficiencia decrece cuando se trata de consultas agrupadas, ya que en este caso, y dado el gran número de grupos posibles, se crea un problema combinatorio para el resolutor de restricciones de SICStus Prolog.

El trabajo futuro más interesante iría en esta línea, por un lado en la simplificación de las fórmulas generadas y por otro en la exploración de otros resolutores y su cooperación, como por ejemplo los resolutores de Dominio Finito y Racionales de IBM ILOG [78] o el resolutor de restricciones Gecode [103].

En cuanto a problema de la depuración de consultas SQL, se presenta un marco teórico para localizar errores en la definición de vistas basada en depuración declarativa. En una primera aproximación, se define una instancia del esquema general de depuración declarativa aplicada al caso de vistas SQL. Se presenta un depurador basado en el recorrido de un árbol que representa el cómputo de la vista SQL que se desea depurar. Los cómputos intermedios, representados por nodos en el árbol, se corresponden con relaciones de la base de datos. La validez de dichos cómputos está determinada por un oráculo externo, que puede ser el usuario o una especificación fiable que contenga la versión correcta de parte de las vistas definidas en el sistema. El proceso de depuración finaliza cuando se encuentra un nodo crítico en el árbol de cómputo, es decir, cuando se encuentra un nodo no válido con hijos válidos. Como resultados teóricos, se prueba la corrección y la completitud con respecto al Algebra Relacional Extendida.

En general, la eficiencia del método se puede medir en base al número de preguntas que se han de formular al usuario antes de localizar el error, junto con la complejidad de las preguntas, en el sentido del número de filas que hay que considerar para determinar la validez de los nodos. En el caso de SQL y a diferencia de lo que ocurre en otros paradigmas, el número de preguntas no es un problema ya que incluso las consultas más complejas se resuelven con un número pequeño de vistas correlacionadas, lo que implica un tamaño del árbol reducido. Además, la estrategia Divide & Query requiere una media de $\log_2 n$ preguntas [106], donde n representa el número de nodos del árbol de cómputo. Por ejemplo, si el árbol contiene 100 nodos (en el

caso de tratarse de la definición de una vista muy compleja), se requiere una media de siete preguntas al oráculo antes de localizar la vista errónea.

Sin embargo la complejidad de las preguntas sí resulta un problema en nuestro marco; dado el gran tamaño que pueden tener las instancias de la base de datos, las preguntas que el usuario debe contestar pueden involucrar demasiadas filas haciendo el método impracticable. Aunque la generación de casos de prueba y su integración con el depurador es una ayuda en este sentido, se presenta además en este capítulo una segunda alternativa de depuración que solventa en gran medida este problema. Se trata de un depurador declarativo que refina la técnica anterior con capacidad para tratar la información que pueda proporcionar el usuario acerca del tipo de error. El depurador se presenta mediante un algoritmo que realizando un análisis de la consulta a depurar y el tipo de error indicado por el usuario (*respuesta perdida* o *respuesta errónea*) es capaz de seguir la traza de las tuplas producidas por cómputos intermedios y que son relevantes a la hora de determinar la fuente del error. De esta forma, las preguntas acerca del resultado de una relación son más sencillas de contestar ya que el usuario solo necesita considerar un subconjunto de tuplas de la relación en lugar de la instancia completa. Aunque el estudio de la procedencia de la información aplicado al resultado de consultas ya se ha estudiado en otros trabajos [68, 53], ninguno de ellos trata el caso de las respuestas perdidas, por lo que este trabajo supone una novedad en este sentido.

Ambas propuestas de depuración se han plasmado en un prototipo de depurador declarativo desarrollado en el sistema DES, lo que nos ha permitido comparar y probar la utilidad de las técnicas.

En esta línea, un posible trabajo futuro consistiría en el estudio de otras técnicas que permitan reducir aún más la complejidad de las preguntas. Por ejemplo, se podrían usar resolutores CHR (Constraint Handling Rules) [64] para generar de forma automática nuevas preguntas durante el proceso de depuración a partir de la información proporcionada por el usuario y la información inferida por el depurador.

Publicaciones asociadas

(A.3) **Applying Constraint Logic Programming to SQL Test Case Generation**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In Tenth International Symposium on Functional and Logic Programming (FLOPS 2010), volume 6009 of *Lecture Notes in Computer Science*, pages 191–206, Sendai, Japan, April 19–21 2010. Springer-Verlag.

→ **Página** 156

(A.4) **Algorithmic Debugging of SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In Perspectives of Systems Informatics: 8th International Andrei Ershov Memorial Conference, (PSI 2011), volume 7162 of *Lecture Notes in Computer Science*, pages 77–85, Novosibirsk, Russia, June 27 – July 1, 2011. Springer-Verlag.

→ **Página** 172

(A.7) **Declarative Debugging of Wrong and Missing Answers for SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012), volume 7294 of *Lecture Notes in Computer Science*, pages 73–87, Kobe, Japan, May 23–25, 2012. Springer-Verlag.

→ **Página** 211

(B.1) **Declarative Debugging of Wrong and Missing Answers for SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

Technical report SIC-3-11. Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain, 2011. Versión Extendida con demostraciones y contenido extra de "*Declarative Debugging of Wrong and Missing Answers for SQL Views*, (FLOPS 2012)".

→ **Página** 260

4 | Bases de datos semiestructuradas: XQuery

En este capítulo se presenta una extensión del lenguaje lógico-funcional $\mathcal{TOY}$ que permite realizar consultas XPath y XQuery sobre documentos XML. El uso de las características propias del lenguaje $\mathcal{TOY}$, como son el *no-determinismo*, *patrones de orden superior*, *variables lógicas* y *generación-prueba*, facilitan el preprocesamiento, la traza y la depuración de consultas, así como la generación de documentos XML como casos de prueba.
En cuanto a la estructura de este capítulo, en la sección 4.1 se introducen brevemente las características del lenguaje $\mathcal{TOY}$ y se propone una representación de los documentos XML mediante tipos de datos. La sección 4.2 está dedicada a la representación del lenguaje XPath en $\mathcal{TOY}$ mediante funciones de orden superior, y es en las Secciones 4.3 y 4.4 donde se presenta una implementación que permite la generación de casos de prueba y la depuración de consultas XPath. El resto del capítulo está dedicado al lenguaje XQuery. En la sección 4.5 se presenta una implementación de XQuery puramente declarativa, representando las consultas mediante tipos de datos en lugar de funciones de orden superior. Esta propuesta nos permite probar de forma sencilla resultados de corrección y completitud con respecto a la semántica operacional de XQuery. La generación de casos de prueba para consultas XQuery se discute en la sección 4.6. Finalmente, en la sección 4.7 presentamos nuestras conclusiones.

4.1. Representación de documentos XML en $\mathcal{TOY}$

Comenzamos esta sección con una breve introducción a las características y capacidades del lenguaje lógico-funcional $\mathcal{TOY}$. Posteriormente, presentamos una representación de documentos XML en este lenguaje.

4.1.1. Lenguaje $\mathcal{TOY}$

$\mathcal{TOY}$ [84] es un lenguaje lógico-funcional perezoso con una sintaxis funcional inspirada en Haskell [77]. En general, un programa $\mathcal{TOY}$ está formado por definiciones

de tipos de datos, alias de tipo, funciones, predicados y operadores infijos. Una *expresión* (total) en $\mathcal{TOY}$ $e \in Exp$ sigue la sintaxis $e ::= X \mid h \mid (e\ e')$ donde X es una variable y h es un símbolo de función o una constructora. Las expresiones de la forma $(e\ e')$ representan la aplicación de la expresión e (actuando como función) a la expresión e' (actuando como argumento). Los *patrones* (totales) $t \in Pat \subset Exp$ pueden ser definidos como $t ::= X \mid c\ t_1 \ldots t_m \mid f\ t_1 \ldots t_m$ donde X representa una variable, c una constructora de aridad m, y f un símbolo de función de aridad n con $n > m$, mientras que t_i son patrones, $0 \leq i \leq m$.
Una función f en $\mathcal{TOY}$ se define mediante reglas con la siguiente sintaxis:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{cabeza}} = \underbrace{r}_{\text{cuerpo}} \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condiciones}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{definiciones locales}}$$

donde f es el nombre de una función, e_i, u_i y r son expresiones, las cuales pueden contener variables extra y t_i, s_i son patrones. La idea general es que la expresión $(f\ v_1 \ldots v_n)$ se reduce a la expresión $r\theta$, con θ una sustitución, si se cumple:

- Cada v_i se reduce a un patrón a_i, $i = 1 \ldots n$, tal que las expresiones $(f\ t_1 \ldots t_n)$ y $(f\ a_1 \ldots a_n)$ son unificables con u.m.g. θ,
- $u_i\theta$ se reduce al patrón $s_i\theta$ para $i = 1 \ldots m$, y
- se satisfacen las condiciones $e_i\theta$ para $i = 1 \ldots k$.

$\mathcal{TOY}$ permite usar notación infija para las funciones binarias. Los operadores asociativos por la izquierda (derecha resp.) se declaran con la palabra reservada `infixl` (`infixr` resp.). Por ejemplo, los operadores /\ y \/ representan la conjunción y la disyunción respectivamente:

```
infixr 30 /\                infixr 30 \/
false /\ X = false          true \/ X = true
true /\ X = X               false \/ X = X
```

En las declaraciones de operadores infijos se indica la prioridad del operador (representa la precedencia del operador con respecto al resto de operadores). Por ejemplo, la siguiente declaración:

```
infixr 45 ?
X ? _Y = X
_X ? Y = Y
```

indica que `?` es un operador infijo asociativo por la derecha con prioridad 45. Este operador representa la elección no determinista. Los objetivos en $\mathcal{TOY}$ son secuencias

de igualdades estrictas. Los cómputos en $\mathcal{TOY}$ comienzan cuando el usuario escribe un objetivo. Por ejemplo:

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

Durante el cómputo de este objetivo, $\mathcal{TOY}$ busca valores de la variable lógica R que satisfacen la igualdad estricta `1 ? 2 ? 3 ? 4 == R`, produciendo cuatro posibles respuestas {`R` $\mapsto$ `1` }, {`R` $\mapsto$ `2` }, {`R` $\mapsto$ `3` }, y {`R` $\mapsto$ `4` }. La siguiente función es una extensión del operador `?` para listas:

```
member [X|Xs] = X ? member Xs
```

En este caso, el objetivo `member [1,2,3,4] == R` produce las mismas respuestas que el objetivo `1 ? 2 ? 3 ? 4 == R`.

Aunque $\mathcal{TOY}$ es un lenguaje tipado, no es necesario hacer explícito el tipo ya que puede ser inferido por el sistema. Por ejemplo, la declaración del tipo de la función `member` se escribe de la siguiente forma:

```
member :: [A] -> A
member [X|Xs] = X ? member Xs
```

indicando que `member` toma una lista de elementos de tipo `A`, y devuelve un valor de tipo `A`. Como es habitual en programación lógico-funcional, $\mathcal{TOY}$ permite aplicaciones parciales en expresiones y variables de orden superior. Por ejemplo, la función que devuelve el n-ésimo elemento de una lista se escribiría como una función de aridad 2 que se aplica cuando recibe dos argumentos, un entero y una lista:

```
nth :: int -> [A] -> A
nth N [X|Xs]  = if N==1 then X else nth (N-1) Xs
```

Por tanto, el siguiente objetivo es válido:

```
Toy> nth 1 == R1, R1 ["hello","friends"] == R2
```

produciendo la respuesta `{ R1` $\mapsto$ `(nth 1), R2` $\mapsto$ `"hello"}`, donde la variable, `R1` se liga a la aplicación parcial `nth 1`. El tipo de la variable `R1` es `([A] -> A)`, por lo que se dice que es una variable de *orden superior*. Una característica de $\mathcal{TOY}$ es que admite *patrones de orden superior*, es decir, admite como patrones expresiones en las que pueden aparecer funciones aplicadas parcialmente. Por ejemplo, se podría escribir la regla:

```
first (nth N)  = N==1
```

ya que `nth N` es un patrón de orden superior. Sin embargo, la regla `first (nth 1 [2]) = true` no es válida, dado que `(nth 1 [2])` es una expresión reducible y no es un patrón válido.

Al igual que en programación lógica, $\mathcal{TOY}$ permite el uso de variables lógicas como parámetros. Por ejemplo, la función `length` que calcula el número de elementos de una lista puede definirse como:

```
length [] = 0
length [X|Xs] = 1 + length Xs
```

siendo posible escribir el objetivo `length [1,2,3] == R` (cuyo resultado es `R` $\rightarrow$ `3`). Sin embargo, la función puede ser usada para generar una lista de elementos de una determinada dimensión, como por ejemplo `length L == 3`. Este objetivo produce la respuesta `R` $\rightarrow$ `[ _A, _B, _C ]`. Esta característica es típica de los lenguajes lógico funcionales la cual se conoce como *generate and test*.

La *lógica de reescritura condicional basada en constructoras* (CRWL según las siglas de su nombre en inglés Constructor-based conditional ReWriting Logic) [88] es un marco semántico que proporciona un cálculo para computar los valores a los que se puede reducir una expresión. CRWL es una semántica apropiada para lenguajes lógico funcionales perezosos soportando indeterminismo y funciones no estrictas. Este cálculo se define mediante cinco reglas de inferencia (ver figura 4.1). La regla (BT) indica que cualquier expresión puede ser reducida al valor indefinido $\bot$. La regla (RR) establece la reflexividad entre variables, mientras que (DC) permite la descomposición de términos. La regla (JN) (join) permite probar igualdades estrictas, y la regla (FA) permite la aplicación de funciones.
En toda regla de inferencia, $e, e_i \in Exp_\bot$ representan expresiones parciales y $t_i, t, s \in Pat_\bot$ representan patrones parciales. La expresión $[P]_\bot$ en la regla de inferencia *FA* representa el conjunto $\{(l \rightarrow r \Leftarrow C)\theta \mid (l \rightarrow r \Leftarrow C) \in \mathcal{P},\ \theta \in Subst_\bot\}$ de instancias parciales de las reglas del programa $\mathcal{P}$. Estas instancias se utilizan en el cómputo del patrón parcial t como aproximación de la llamada a función $f\ \overline{e}_n$.
En esta semántica las declaraciones locales de la forma $a = b$ expresadas en

BT	$e \rightarrow \bot$	
RR	$X \rightarrow X$	with $X \in Var$
DC	$\dfrac{e_1 \rightarrow t_1 \ \ldots \ e_m \rightarrow t_m}{h\ \overline{e}_m \rightarrow h\ \overline{t}_m}$	$h\ \overline{t}_m \in Pat_\bot$
JN	$\dfrac{e \rightarrow t \quad e' \rightarrow t}{e \ == \ e'}$	$t \in Pat$ (total pattern)
FA	$\dfrac{e_1 \rightarrow t_1 \ \ldots \ e_n \rightarrow t_n \quad C \quad r\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \ \rightarrow t}$ if $(f\ \overline{t}_n \rightarrow r \Leftarrow C) \in [\mathcal{P}]_\bot, t \neq \bot$	

Figura 4.1: Calculo semántico CRWL

```
data xmlNode   = xmlText    string
               | xmlComment string
               | xmlTag     string [xmlAttribute] [xmlNode]
data xmlAttribute = xmlAtt     string string
type xml       = xmlNode
```

Figura 4.2: Representación de documentos XML en $\mathcal{TOY}$.

$\mathcal{TOY}$ mediante la palabra reservada `where` son parte de la condición C como *instrucciones de aproximación* de la forma $b \rightarrow a$.

Escribiremos $\mathcal{P} \vdash \varphi$ para expresar que la sentencia φ es demostrable en CRWL con respecto al programa $\mathcal{P}$.

4.1.2. Representación de XML en $\mathcal{TOY}$

Para representar documentos XML en $\mathcal{TOY}$ usamos definiciones de tipos de datos y alias de tipo como se muestra en la figura 4.2. Consideramos tres tipos de nodos: nodo elemento (tags), nodo texto y nodo comentario. Cada uno de estos nodos se representa con el tipo de dato `xmlNode` definido mediante una constructora cuyos argumentos representan la información del nodo en cuestión. Por ejemplo, la constructora `xmlTag` permite representar los nodos de tipo elemento. Su primer argumento es de tipo `string` y simboliza el nombre del elemento, una lista de atributos como segundo argumento y la lista de nodos hijos como tercer argumento. El tipo de datos `xmlAttribute` se define mediante la constructora `xmlAtt` de aridad dos. El primer argumento simboliza el nombre del atributo y el segundo argumento simboliza su valor. El alias de tipo `xml` permite renombrar el tipo de dato `xmlNode`.

Definimos las primitivas `load_xml_file` y `write_xml_file` en $\mathcal{TOY}$ para cargar

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
xmlTag "root" [xmlAtt  "version"  "1.0"] [
xmlTag "food"  [][
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "watermelon"],
    xmlTag "price" [][xmlText "32" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "oranges" ],
    xmlTag "variety" [][xmlText "navel"],
    xmlTag "price" [][xmlText "74" ]
  ],
  xmlTag "item" [xmlAtt "type" "vegetable"][
    xmlTag "name" [][xmlText "onions" ],
    xmlTag "price" [][xmlText "55" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"] [
    xmlTag "name" [][xmlText "strawberries"],
    xmlTag "variety" [][xmlText "alpine"],
    xmlTag "price" [][xmlText "210" ]
  ]
 ]]
```

Figura 4.3: Documento "food.xml"(izquierda) y su representación en $\mathcal{TOY}$ (derecha).

y guardar documentos XML de forma automática. En este trabajo no se comprueba la validez de los documentos XML con respecto a un esquema o una DTD [119], sino que se asume que dichos documentos están bien formados. La primitiva `load_xml_file` se aplica a un documento XML y devuelve su representación en $\mathcal{TOY}$ como un valor de tipo `document`. En la figura 4.3 se presenta un ejemplo de documento XML ("food.xml") y su representación en $\mathcal{TOY}$.

En $\mathcal{TOY}$ todos los documentos XML comienzan con un nodo elemento llamado `root` que permite agrupar fragmentos de documentos XML. En el caso de que el documento inicial contenga un único nodo `N` en el nivel más externo, el nodo `root` puede ser eliminado definiendo la siguiente función `load_doc`:

```
load_doc F = N <== load_xml_file F == xmlTag "root" [xmlAtt "version"
"1.0"] [N]
```

donde `F` es el nombre del fichero que contiene el documento XML. La igualdad estricta `==` en la condición fuerza la evaluación de la expresión `load_xml_file F`, la cual tiene éxito si el resultado es de la forma `xmlTag "root" [xmlAtt "version" "1.0"] [N]` para algún `N`, en cuyo caso se devuelve como resultado el valor de `N`.

XPath	= `doc(fileName) /` *Relative*
Relative	= *Step*$_1$ `/` ... `/` *Step*$_n$ | *Relative* | *Relative*
Step	= *Axis* :: *Test* | *Axis* :: *Test* [Filter]
Axis	= `self` | *ForwardAxis* | *ReverseAxis*
ForwardAxis	= `child` | `descendant` | `descendant-or-self` | ...
ReverseAxis	= `parent` | `ancestor` | `ancestor-or-self` | ...
Test	= `node()` | *name* | `text()` | `comment()` | *

Figura 4.4: Gramática de un subconjunto del lenguaje XPath.

4.2. Consultas XPath en $\mathcal{TOY}$

XPath [50] es un lenguaje funcional tipado que permite localizar y recuperar distintas partes de un documento XML mediante la definición de caminos en el árbol. En la figura 4.4 se presenta la gramática que define un subconjunto del lenguaje XPath con el que podemos expresar la mayoría de las consultas. La descripción completa de la gramática de XPath se puede encontrar en [50]. Una expresión XPath es una secuencia de pasos separados por el combinador `/` que definen un camino o ruta que avanza por el árbol XML. El primer paso es de la forma `doc(fileName)` fijando así el nodo inicial a la raíz del documento contenido en el fichero `fileName`. Cada paso produce un conjunto de nodos, cada uno de los cuales es usado como nodo de entrada por el siguiente paso. Cada paso (*Step*) en una ruta puede tener un eje (*Axis*), un test (*Test*) y cero o más predicados y se evalúa con respecto al conjunto de nodos producidos por el paso anterior. El combinador `::` se utiliza para separar un eje de su correspondiente test. El eje de un paso especifica la dirección del camino, pudiendo ser esta ascendente o descendente en el árbol. Un eje puede tomar el valor `self` (para referirse al nodo actual), `child` (para referirse a los hijos del nodo actual), `parent` (padre del nodo actual), etc. Los test son utilizados para incluir o excluir nodos dentro de un eje, ya sea por nombre o por tipo. El resultado son todos los nodos que cumplen el test. Un predicado o filtro permite restringir el conjunto de nodos seleccionados por un eje a aquellos que cumplen cierta condición. De esta forma, las siguientes expresiones son consultas XPath válidas sobre el documento `"food.xml"` de la figura 4.3:

```
(C1) : doc("food.xml")/child::food/child::item/child::name
(C2) : doc("food.xml")/child::food/child::item[child::variety]
```

El resultado de una consulta XPath es un conjunto de nodos del documento XML que se está consultando.

4.2.1. Representación de XPath en $\mathcal{TOY}$

La representación de las consultas XPath en $\mathcal{TOY}$ se realiza mediante funciones de orden superior conectadas mediante operadores de orden superior. De esta forma, es posible considerar las expresiones XPath como código ejecutable (cuando se aplican a un documento XML) o como estructuras de datos, cuando se las considera como patrones de orden superior. Esta representación resulta muy beneficiosa ya que permite el preprocesamiento, la traza y la depuración de consultas XPath.

El tipo de una expresión XPath en un lenguaje funcional podría definirse de la forma `type xPath = xml ->[xml]`, indicando que la expresión XPath se aplica a un documento XML produciendo una lista o secuencia de nodos XML. Este es el enfoque considerando tanto en programación lógica [4] como en programación funcional [73]. En el lenguaje $\mathcal{TOY}$, dada su naturaleza no determinista, es posible generar cada resultado de forma independiente, por lo que una expresión XPath se define como una función donde tanto el dato de entrada como el resultado que se produce es un fragmento de un documento XML. En $\mathcal{TOY}$ definimos el tipo de una expresión XPath de la forma:

```
type xPath = xml ->xml
```

Para aplicar una expresión XPath a un documento XML en $\mathcal{TOY}$, se define el siguiente operador infijo:

```
infix  20 <--
(<--)::string  -> xPath ->  xml
Doc <-- Q = Q (load_xml_file Doc)
```

Este operador juega en $\mathcal{TOY}$ el papel de `doc` en XPath. Tiene como argumentos de entrada la consulta XPath `Q` y el nombre del fichero XML `Doc`. Este operador aplica la consulta `Q` al documento XML contenido en el fichero `Doc`.

Definimos los combinadores `/` y `::` de XPath como composición de funciones en $\mathcal{TOY}$ como se muestra a continuación:

```
infixr 55 .::.
(.::.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)

infixr 40 ./.
(./.) :: xPath -> xPath -> xPath
(F ./. G) X = G (F X)
```

```
self :: xPath
self X = X

child :: xPath
child (xmlTag _Name _Attr L) = member L

descendant :: xPath
descendant X = Y ? descendant Y <== Y == child X

descendant_or_self :: xPath
descendant_or_self = self ? descendant
```

Figura 4.5: Definición de los ejes básicos de XPath en $\mathcal{TOY}$.

Aunque ambas funciones se definen igual, la función `./.` se utiliza en $\mathcal{TOY}$ para conectar los diferentes pasos de una expresión XPath, mientras que la función `.::.` se utiliza para definir un paso combinando un eje y un test. Sería posible utilizar una única función para implementar los dos combinadores, pero preferimos diferenciarlos para mantener una sintaxis lo más parecida posible a XPath. En primer lugar se aplica la expresión XPath (`F`) al nodo contexto (representado por la variable `X`) y posteriormente se aplica la expresión XPath (`G`) al resultado.

La definición de los ejes básicos de XPath en $\mathcal{TOY}$ se encuentra en la figura 4.5. El eje `self` se define como la función identidad devolviendo como resultado el nodo actual. El eje `child` se aplica únicamente a nodos de tipo elemento[1], los cuales se representan en $\mathcal{TOY}$ como términos mediante la constructora `xmlTag`. La función `child` usa la función no determinista `member` devolviendo como resultado todos los hijos del nodo actual.

Los ejes `descendant`[2] y `descendant-or-self`[3] se definen a partir de los ejes definidos anteriormente. El eje `descendant` se define como una regla condicional. La igualdad estricta en la condición fuerza la evaluación de la expresión `child X`. Si `child X` tiene éxito con resultado `Y`, el resultado de `descendant X` es la elección no determinista `Y ? descendant Y`. La condición se utiliza para evitar que la función `descendant` realice infinitas llamadas recursivas cuando se aplica a un nodo sin hijos. La función `descendant-or-self` se define de forma natural a partir de las funciones `self` y `descendant` sin necesidad de incluir el fragmento de XML como parámetro de entrada.

En la figura 4.6 se definen algunos de los posibles tests de XPath. El test `node()` se representa en $\mathcal{TOY}$ mediante la función identidad `nodeT`. Por ejemplo, la siguiente consulta XPath devuelve todos los nodos del documento XML `"food.xml"` de la figura 4.3:

[1] En XML solo los nodos de tipo elemento tienen hijos.
[2] El eje `descendant` se refiere a los descendientes del nodo contexto.
[3] El eje `descendant-or-self` se refiere al nodo contexto y a todos sus descendientes.

```
nodeT :: xPath
nodeT X = X

nameT :: string ->xPath
nameT S (xmlTag S Att L) = xmlTag S Att L

textT :: string ->xPath
textT S (xmlText S) = xmlText S

commentT :: string ->xPath
commentT S (xmlComment S) = xmlComment S

elem :: xPath
elem = nameT _
```

Figura 4.6: Definición de tests en $\mathcal{TOY}$.

XPath → `doc("food.xml")/descendant-or-self::node()`
$\mathcal{TOY}$ → `("food.xml" <-- descendant_or_self.::.nodeT) == R`

La primera línea se corresponde con la sintaxis habitual de XPath y la segunda linea se corresponde con la consulta equivalente escrita en $\mathcal{TOY}$. La única diferencia es que el sistema $\mathcal{TOY}$ devuelve en la variable `R` un único nodo cada vez. Si el usuario necesita obtener todas las soluciones, como es habitual en los sistemas que implementan XPath, es necesario utilizar la primitiva `collect`. La siguiente consulta usa la primitiva `collect` produciendo un único resultado que agrupa en una lista todos los nodos del documento `"food.xml"`:

```
Toy> collect("food.xml" <-- descendant_or_self.::.nodeT) == R
     { R -> [ (xmlTag  "root" [ (xmlAtt  "version"  "1.0") ]
                [ (xmlTag  "food" []
                  [ (xmlTag  "item" [ (xmlAtt  "type"  "fruit")]
                ...
                ...
                (xmlText  "strawberries"),
                (xmlText  "alpine"),
                (xmlText  "210") ] }

sol.1, more solutions (y/n/d/a) [y]?
      no
```

El test *name* de XPath se representa en $\mathcal{TOY}$ mediante la función `nameT`. Esta función comprueba si el nodo contexto es un nodo elemento de nombre `S`, en cuyo caso se devuelve el propio nodo contexto. En otro caso, la función falla. La siguiente expresión XPath utiliza este test:

```
XPath → child::food/child::item
TOY → child.::.nameT "food"./.child.::.nameT "item"
```

Los tests `text()` y `comment()` se implementan en $\mathcal{TOY}$ con las funciones `textT` y `commentT` respectivamente. Para implementar estas funciones en $\mathcal{TOY}$ usamos una variable lógica para recuperar el valor del texto asociado al nodo. Por ejemplo si queremos conocer el precio de todos los productos contenidos en el documento, escribiremos la consulta siguiente:

```
XPath → child::food/child::item/child::price/child::text()
TOY → child.::.nameT "food"./.child.::.nameT "item"./.
       child.::.nameT "price"./.child.::.textT P
```

La variable lógica `P` se instanciará con los valores de los nodos de tipo texto que son hijos de nodos de tipo elemento denominados `"price"`. En XPath la expresión `child::*` devuelve todos los nodos de tipo elemento que son hijos del nodo contexto. El test `*` se representa en $\mathcal{TOY}$ mediante la función `elem`.

Las expresiones XPath se pueden escribir de forma más compacta mediante una sintaxis abreviada. Por ejemplo, es posible expresar consultas sin necesidad de escribir el eje `child` en los pasos en los que este eje aparece seguido del nombre. Por ejemplo, la expresión `child::food/child::price/child::item` puede escribirse de forma abreviada como `food/price/item`. En $\mathcal{TOY}$, al ser un lenguaje tipado, no es posible hacer lo mismo ya que la función `./.` se aplica a expresiones de tipo XPath y no a cadenas de texto. Sin embargo, es posible definir nuevos operadores que transformen cadenas de caracteres en expresiones de tipo XPath.

```
name :: string -> xPath
name S = child.::.(nameT S)
```

Así, la expresión XPath anterior se escribe en $\mathcal{TOY}$ de forma abreviada como `name "food"./.name "item"./.name "price"`. La misma idea se aplica a los tests `commentT` y `textT` ([35](A.5),[34]).

De forma opcional, un test en XPath puede incluir un predicado o filtro. En $\mathcal{TOY}$ definimos el operador `.#` para conectar un filtro con su correspondiente expresión XPath:

```
infixr  60 .#
(.#) :: xPath -> xPath -> xPath
(Q .# F)  X =  if F Y == _ then Y  where Y = Q X
```

En esta definición se comprueba que el nodo contexto `Y` verifica el filtro `F`, siendo `Y` el resultado de aplicar la consulta `Q` al nodo contexto `X`. La función devuelve como resultado el nodo `Y` en el caso de que la expresión `F Y == _` no falle.

En cuanto a los diferentes tipos de filtros, el lenguaje XPath admite muchas posibilidades. En esta tesis se tratan únicamente filtros representados mediante expresiones XPath y filtros de posición [34]. Un filtro que resulta interesante es el que selecciona atributos de nombre `S` con un cierto valor `V`. En XPath se utiliza el símbolo `@` el cual se representa en $\mathcal{TOY}$ con el operador `@=` definido como:

```
(@=) :: string -> string -> xPath
(@=) S V X = if (xmlAtt S V == member Attr) then X
             where (xmlTag _Name Attr _L) = X
```

Este filtro comprueba que el nodo contexto `X` tenga un atributo `S` con valor `V`, en cuyo caso lo devuelve como resultado.

4.2.2. Representación en $\mathcal{TOY}$ de los ejes `parent` y `ancestor`

Mediante los ejes definidos en la figura 4.5 es posible definir caminos que recorren el documento XML de forma descendente. Sin embargo XPath permite referirnos al padre o a los antecesores del nodo contexto mediante los ejes `parent`, `ancestor` y `ancestor-or-self`. La implementación de estos ejes no es trivial ya que la función `xPath` recibe como entrada los fragmentos del documento XML que satisfacen los pasos previos y que se corresponden con subárboles del documento XML inicial. De esta forma no es posible recuperar el padre o el antecesor del nodo contexto. Sin embargo, gracias a los patrones de orden superior de $\mathcal{TOY}$ es posible considerar las expresiones XPath como términos lo que permite examinar y transformar consultas XPath antes y durante su evaluación. A continuación mostramos cómo utilizar esta característica de $\mathcal{TOY}$ para introducir los ejes `parent` y `ancestor`. En primer lugar definimos unas reglas para transformar una consulta XPath en otra equivalente que no contiene ejes `parent` y `ancestor` [34]. La función `transform` se define como la función identidad excepto en el caso de encontrar un eje `parent` o `ancestor`:

(P_1)	child::T_1/S/parent::T_2	$\equiv$ self::T_2[child::T_1/S]
(P_2)	descendant::T_1/S/parent::T_2	$\equiv$ self::T_2[child::T_1/S]
(P_3)	descendant::T_1/S/parent::T_2	$\equiv$ descendant::T_2[child::T_1/S]
(A_1)	S_1/S_2/ancestor::T	$\Rightarrow$ S_1/self::T[S_2]
(A_2)	descendant::T_1/ancestor::T_2	$\Rightarrow$ descendant::T_2[descendant::T_1]

Figura 4.7: Reglas de sustitución de los ejes `parent` y `ancestor` por filtros.

```
transform::xPath -> xPath -> xPath
transform X (self.::.T)       =  X ./.(self.::.T)
transform X (child.::.T)      =  X ./.(child.::.T)
transform X (descendant.::.T) =  X ./.(descendant.::.T)
transform X (parent.::.T)     = delParent  X T
transform X (ancestor.::.T)   = removeAncestor  X T
```

La variable `X` representa la parte de la expresión XPath que ha sido procesada mientras que el segundo argumento representa el siguiente paso de la consulta. La función `transform` llama a la función `delParent` y `removeAncestor` cuando se encuentran los ejes `parent` y `ancestor` respectivamente.

En la figura 4.7 se definen las reglas que permiten reemplazar los ejes `parent` y `ancestor` por filtros (de forma análoga a como se hace en [93, 94]). Los tests T_1 y T_2 pueden contener filtros. S es una secuencia (posiblemente vacía) de pasos que usa solo el eje `self` y S_1 y S_2 son secuencias de pasos tal que S_2 no es vacía y su primer paso no es de la forma `self`::T.

Por ejemplo, la regla (P_1) permite transformar la expresión `child::variety/parent::node()` en la expresión equivalente `self::node()[child::variety]`.

Las reglas $(P_1), (P_2)$ y (P_3) se implementan en $\mathcal{TOY}$ mediante la función `delParent` usando patrones de orden superior:

```
delParent::  xPath -> xPath ->xPath
delParent (X./.self.::.T1) T2 =
                        addFilter (delParent X T2) (self.::.T1)
delParent (X./.child.::.T1) T2      =
                        X ./. self .::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 =
                        X ./. self .::.(T2.#(child.::.T1))
delParent (X./.descendant.::.T1) T2 =
                        X ./. descendant .::. (T2.#(child.::.T1))
```

```
addFilter:: xPath->xPath->xPath
addFilter (X./.A.::.(T.#F)) G = X ./. (A.::. (T.# (F ./. G)))
```

Ahora, para poder aplicar una expresión XPath a un documento XML en $\mathcal{TOY}$, es necesario redefinir el operador `<--` para realizar un preprocesamiento de la consulta XPath antes de aplicarla al documento XML:

```
Doc <-- Q =  (preprocess Q) (load_xml_file Doc).
```

La función `preprocess` se define como:

```
preprocess :: xPath -> xPath
preprocess A = rev (foldl transform id  A)

foldl::(xPath->xPath->xPath)->xPath->xPath->xPath
foldl F Z (A.::.T) = F Z (A.::.T)
foldl F Z (G ./. H) = foldl F (F Z G) H

rev::xPath -> xPath
rev (A.::.B) = A.::.B
rev (F./.G) = rev' F G
rev' (A.::.B) G = (A.::.B) ./. G
rev' (X ./. Y) G = rev' X (Y./. G)
```

La función `preprocess` aplica la función `transform` a la consulta usando una versión de la conocida función `fold` de programación funcional para listas, pero en este caso aplicada a una consulta XPath y el paso identidad como valores iniciales. El resultado es una secuencia de pasos asociados por la izquierda de la forma $(S_1./.S_2)./.S_3$. La función `rev`, análoga a la función *reverse* usada en programación funcional para listas, se encarga de agrupar por la derecha la secuencia anterior.

El resultado de una consulta XPath puede contener errores. Puede ocurrir que siendo sintácticamente correcta devuelva un nodo del árbol XML que el usuario no esperaba, por lo que nos encontramos ante un caso de respuesta errónea, o que por el contrario, la consulta no sea capaz de devolver un nodo concreto del árbol XML, lo que sería un caso de respuesta perdida. El origen del error puede estar en el propio documento XML, pero dada la estructura tan compleja de los documentos, resulta muy habitual que el error se encuentre en la definición de la consulta XPath. Las siguientes secciones se dedican a tratar el problema de la generación de casos de prueba y la depuración.

4.3. Generación de casos de prueba para consultas XPath

En algunos casos puede resultar útil disponer de casos de prueba para probar las consultas, es decir, documentos XML de pequeño tamaño de tal forma que al aplicar una consulta XPath se obtenga algún resultado. De esta forma, la comparación de la estructura del documento original con la estructura del caso de prueba, puede ayudar a localizar el error.

Por ejemplo, la siguiente consulta XPath:

```
Toy> "food.xml" <-- (name "food" ./. name "item" ./. name "type" ./.
                     child.::.textT "navel")  == R
```

falla en $\mathcal{TOY}$ y no devuelve ningún resultado. El problema de la generación de casos de prueba para consultas XPath se resuelve en $\mathcal{TOY}$ simplemente lanzando el objetivo:

```
Toy> (name "food" ./. name "item" ./. name "type" ./.
       child.::.textT "navel") X  == _
```

produciéndose el siguiente resultado:

```
{ X -> (xmlTag _A _B
         [ (xmlTag  "food" _C
             [ (xmlTag  "item" _D
                 [ (xmlTag  "type" _E
                   [ (xmlText  "navel") | _F ]) | _G
                ]) | _H
             ]) | _I
        ])
```

La variable `X` representa un caso de prueba para la consulta `(name "food"./. name "item" ./. name "type" ./.child.::.textT "navel")`. Para formatear la respuesta y que sea más legible por el usuario definimos la función `generateTC` de la siguiente forma:

```
generateTC:: xPath -> string -> bool
generateTC Q S = if (Q X == _) then  write_xml_file X S
```

La función `generateTC` recibe como parámetros una expresión XPath `Q` y el nombre del fichero `S` donde se grabará el caso de prueba generado. La primitiva `write_xml_file` escribe el documento XML representado por la variable `X` en el fichero de nombre `S`. De esta forma, el objetivo:

```
Toy> generateTC (name "food" ./. name "item" ./. name "type"
             ./. child.::.textT "navel") "tc.xml" == R
```

crea un fichero con nombre `"tc.xml"` que contiene el siguiente documento XML:

```
<food>
    <item>
       <type>navel</type>
   </item>
</food>
```

La primitiva `write_xml_file` sustituye las variables lógicas que representan listas de elementos por listas vacías. Así, se genera un caso de prueba de tamaño reducido.

Si se compara el documento XML `"food.xml"` de la figura 4.3 con el caso de prueba generado, es fácil ver que `"type"` es un atributo de los elementos `"item"`, y no un hijo, lo que explica que la consulta no devuelva ningún resultado.

4.4. Depuración de consultas XPath

Como ya hemos visto anteriormente, las expresiones XPath en $\mathcal{TOY}$ pueden ser manipuladas fácilmente, lo que permite trazar y depurar consultas XPath para localizar errores.

4.4.1. Tratamiento de resultados erróneos

Supongamos que la siguiente consulta sobre el documento de la figura 4.8:

```
Toy> "bib.xml" <-- ( name "bib" ./. name "book" ./.
                     name "author" ./. name "last" ) == R
```

```
<?xml version='1.0'?>
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

Figura 4.8: Documento XML: bib.xml.

produce el siguiente resultado no esperado por el usuario:

```
R -> (tag "last" [] [ (txt "Abiteboul") ])
```

En este caso, si el origen del error está en el documento XML, puede ser sencillo encontrarlo simplemente realizando una búsqueda de la cadena *Abiteboul* en el documento "bib.xml". En otro caso, puede ser útil localizar el nodo erróneo en el árbol y posteriormente realizar una traza en sentido inverso del camino en el que se encuentra dicho nodo hasta encontrar el primer paso de la consulta XPath que ha seleccionado un nodo no esperado y que por tanto constituye el origen del error.

Implementamos en $\mathcal{TOY}$ la función `wrong` para conocer el resultado de los pasos intermedios especificados en la expresión XPath. La función `wrong` se encarga de calcular la traza, mientras que la función `generateTrace` genera un fichero con extensión `xml` que almacena un documento XML con la información de dicha traza.

```
wrong (A.::.B) I O = [((A.::.B), I, O)] <== (A.::.B) I == O
wrong (A./.B) I O  = [(A, I, O1 ) | wrong B O1 O]
                     <== A I == O1, B O1 == O
```

La función `wrong` recibe como argumentos la consulta XPath, un nodo del documento XML (inicialmente será el documento completo) y el nodo resultado de aplicar la consulta al documento. Como puede observarse, la segunda regla usa la característica *generate and test* típica de los lenguajes lógico funcionales para generar posibles valores de la variable `O1` (nodos producidos por la expresión XPath `A`) tal que la consulta `B` aplicada a `O1` produzca el nodo `O`. Como resultado, la función produce una lista con tantos elementos como pasos describa la consulta XPath original, donde cada paso está asociado a un nodo de entrada (al que se le aplica el paso) y otro de salida (el resultado de aplicar dicho paso al nodo de entrada).

El resultado de la función `wrong` es difícil de manejar por el usuario por lo que hemos implementado la función `writeTrace` para representar el resultado de `wrong` mediante un documento XML que puede ser almacenado para su posterior manipulación.

```
traceStep (Step,I,O) = xmlTag "step" [] [ xmlTag "query" []
                       [xmlText (show Step)],
                        xmlTag "input" [] [I],
                        xmlTag "output" [] [O] ]

generateTrace L = xmlTag "root" [] (map traceStep (rev L))

writeTrace XPath InputFile WrongOutput OutputFile =
      write_xml_file (
           generateTrace (
              wrong XPath (
                 load_xml_file InputFile)
              WrongOutput))
           OutputFile

show (A.::.B)    = (show A)++".::."++(show B)
show nodeT       = "nodeT"
...
show child      = "child"
...
```

La function `traceStep` genera un nodo XML de tipo elemento denominado `step` con la información de un paso concreto en la evaluación de la consulta: el paso, su nodo de entrada y el de salida. La función `generateTrace` aplica la función `traceStep` a una lista de pasos. Esta función usa las funciones típicas de programación funcional `map` y `rev`. La función `rev` se usa para asegurar que el último paso de la consulta inicial es el primero en el documento y así poder seguir la traza en orden inverso. Finalmente, la función `writeTrace` combina las funciones anteriores para generar y grabar en un fichero la información de la traza.

El siguiente objetivo genera un documento XML con toda la información que el usuario necesita para realizar la traza de la consulta y lo graba como un documento de texto llamado "trace.xml":

```
Toy> writeTrace (name "bib" ./. name "book" ./. name "author" ./.
                 name "last" ) "bib.xml"
                 (tag  "last" [] [txt  "Abiteboul"])
                 "trace.xml"
```

A continuación mostramos parcialmente el contenido del documento `"trace.xml"` generado. Como se puede observar, los dos pasos mostrados con la etiqueta `<step>` se corresponden con el último y el penúltimo paso de la consulta XPath.

```
<step>
 <query>child.::.nameT last</query>
 <input>
  <author>
   <last>Abiteboul</last>
   <first>Serge</first>
  </author>
 </input>
 <output>
  <last>Abiteboul</last>
 </output>
</step>

<step>
 <query>child.::.nameT author</query>
 <input>
  <book year="2000">
   <title>Data on the Web</title>
   <author>
    <last>Abiteboul</last>
    <first>Serge</first>
   </author>
   <author>
```

```
    <last>Buneman</last>
    <first>Peter</first>
   </author>
   <author>
    <last>Suciu</last>
    <first>Dan</first>
   </author>
   <publisher>Morgan Kaufmann Publishers</publisher>
   <price>39.95</price>
  </book>
 </input>
 <output>
  <author>
   <last>Abiteboul</last>
   <first>Serge</first>
  </author>
 </output>
</step>
...
```

4.4.2. Tratamiento de resultados incompletos

En algunos casos las consultas XPath describen un camino del árbol que no existe, y por tanto no son capaces de seleccionar nodos del árbol. Este es un error muy común (y difícil de detectar) cuando la estructura del documento XML que se desea consultar es compleja y tiene una gran cantidad de nodos.

Por ejemplo, la consulta:

```
   "food.xml" <-- name "food"./. name "item" ./. name "type" ./.
child.::.textT "navel" == R
```

falla en $\mathcal{TOY}$ y no devuelve ningún resultado.

Como se puede observar en el documento de la figura 4.3 el camino especificado por la consulta anterior no existe ya que no existe ningún nodo elemento en el documento `"food.xml"` cuyo nombre sea `type`. Tendría más sentido escribir la consulta:

```
   "food.xml" <-- name "food"./. name "item" ./. name "variety"./.
child.::.textT "navel" == R
```

cuyo resultado es: `R ->(xmlText "navel")`. Este tipo de errores no se pueden tratar siguiendo la traza inversa descrita en la sección anterior, ya que no se dispone de camino válido dentro del árbol. Para manejar este tipo de errores proponemos buscar la secuencia de pasos en el camino definido por la consulta XPath que hace que la consulta seleccione algún nodo. La idea es aplicar la consulta inicial sin el último paso.

Si devuelve algún nodo, el error está en el último paso. Si falla, aplicar la consulta sin los dos últimos pasos y así sucesivamente. Siguiendo este proceso encontramos el primer paso en la consulta que devuelve una secuencia vacía de nodos, y que es normalmente la causa del error. Este proceso se implementa mediante la función `missing`:

```
missing  (A.::.B) R = (A.::.B)
missing (X ./. Y) R = if (collect (X R) == []) then X
                      else missing Y (X R)
```

Si lanzamos el siguiente objetivo en $\mathcal{TOY}$:

```
Toy> missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
    { R -> child .::. (nameT  "type") }
```

La respuesta indica que el tercer paso de la consulta es la posible causa del error. Además, en algunos casos, como por ejemplo en el caso del test *name*, es posible indicar al usuario cuál debería ser el paso correcto. Para ello redefinimos la función `missing`:

```
missing (A.::.B) R =  guess (A.::.B) self R

missing (Step ./. Y) R = if (collect (Step R) == [])
                         then guess Step Y R
                         else missing Y (Step R)

guess Step Y R = if Step==(A.::.nameT B)
                 then if (StepBis ./. Y) R == _
                      then ( Step, "Substitute "++B++" by "++C )
                      else (Step, "No suggestion")
                 else (Step, "No suggestion")
    where StepBis = (A.::.nameT C)
```

En este caso, el resultado de la función `missing` es un par donde el primer elemento es un paso de la consulta (como en la primera versión de la función) y el segundo elemento una sugerencia acerca del cambio que habría que hacer para corregir el error. De esta forma, el objetivo:

```
Toy>  missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
{ R -> (child .::. (nameT  "type"),  "Substitute type by variety") }
```

devuelve como resultado el paso `(child .::. (nameT "type")` como posible causa del error e indica que la consulta XPath inicial devolvería algún nodo del árbol si se sustituyera la cadena `"type"` por la cadena `"variety"`.

4.5. Consultas XQuery en $\mathcal{TOY}$

Las expresiones XQuery permiten consultar documentos XML con una sintaxis algo más compleja que XPath, permitiendo escribir condiciones, agrupar resultados y generar nuevos fragmentos XML entre otras muchas características [121, 46]. Lo visto hasta ahora es útil para XPath, pero las estructuras de XQuery no se representan tan fácilmente como funciones de orden superior. En [9](A.8) presentamos un intento en este sentido, donde se puede encontrar una implementación del lenguaje XQuery en $\mathcal{TOY}$ basada en la definición de la construcción *for-let-where-return* mediante funciones $\mathcal{TOY}$. Al igual que ocurría con las expresiones XPath, las expresiones XQuery pueden ser ejecutadas en el sistema $\mathcal{TOY}$ produciendo como resultado fragmentos de documentos XML de forma no determinista. En esta propuesta se permiten instrucciones de tipo *let* usando la meta-primitiva *collect*. Sin embargo en este documento de tesis presentamos una implementación, tanto para XPath como para XQuery, puramente declarativa, lo que nos permitirá probar que la extensión es correcta y completa con respecto a la semántica de XQuery.

4.5.1. XQuery y su semántica operacional

En [22] se presenta un subconjunto del lenguaje XQuery que permite escribir expresiones XQuery mediante instrucciones del tipo *for*, *let* y *where/if*. En esta tesis consideramos un subconjunto de XQ al que llamamos SXQ cuya sintaxis se describe en la figura 4.9. En esta gramática, a representa el nombre de una etiqueta. Las diferencias de SXQ con respecto a XQ son:

1. XQ permite el uso de variables como nombre de etiquetas usando el constructor *lab($x)*. Inicialmente SXQ no lo permite, pero puede ser extendida fácilmente para soportar esta característica.

2. XQ permite encerrar una consulta Q entre etiquetas de la forma $\langle a\rangle Q\langle/a\rangle$. SXQ permite representar la etiqueta vacía y la etiqueta que contiene variables u otras etiquetas.

3. XQ admite como válida la consulta vacía ( ), lo que permite representar expresiones de la forma $\langle a\rangle\langle/a\rangle$. Aunque SXQ no permite la consulta vacía, la expresión anterior se puede construir mediante el constructor *tag*.

$$
\begin{array}{rcl}
query & ::= & query\ query \mid tag \\
 & \mid & var \mid var/axis :: test \\
 & \mid & \textbf{for}\ var\ \textbf{in}\ query\ \textbf{return}\ query \\
 & \mid & \textbf{if}\ cond\ \textbf{then}\ query \\
 & & \\
cond & ::= & var = var \mid query \\
tag & ::= & \langle a\rangle\ \langle/a\rangle \mid \langle a\rangle var \ldots var \langle/a\rangle \mid \langle a\rangle tag \langle/a\rangle \\
 & & \\
axis & ::= & \textbf{self} \mid \textbf{child} \mid \textbf{descendant} \mid \textbf{dos} \mid \ldots
\end{array}
$$

Figura 4.9: Sintaxis de SXQ, una versión simplificada de XQ.

Aunque la construcción *let* no forma parte de la gramática XQ, puede ser simulada usando instrucciones de tipo *for* encerradas entre etiquetas. En el caso de SXQ, al no permitir el uso de consultas diferentes de variables entre etiquetas, no es posible expresar construcciones de tipo *let*. La causa de esta limitación viene impuesta por la esencia no determinista de $\mathcal{TOY}$, ya que las expresiones *let* reúnen todos los resultados de una consulta en lugar de producirlos de forma individual usando no determinismo. A pesar de todas estas limitaciones, el lenguaje SXQ permite escribir un amplio abanico de consultas como muestra el siguiente ejemplo:

Ejemplo 4.5.1. *El documento "bib.xml" de la figura 4.8 contiene información acerca de libros y el documento "reviews.xml" de la figura 4.10 contiene información adicional de algunos de esos libros (documentos tomados de [118]). La siguiente consulta XQuery selecciona las revisiones de los libros que aparecen en "bib.xml".*

```
for $b in doc("bib.xml")/bib/book,
    $r in doc("reviews.xml")/reviews/entry
where $b/title = $r/title
for $booktitle in $r/title,
    $revtext in $r/review
return <rev> $booktitle  $revtext </rev>
```

La consulta anterior se puede escribir con sintaxis SXQ de la siguiente manera:

```
   for $x3 in $x1/child::bib return
   for $x4 in $x3/child::book  return
   for $x5 in $x2/child::reviews return
   for $x6 in $x5/child::entry  return
   for $x7 in $x4/child::title return
   for $x8 in $x6/child::title return
   if ($x7 = $x8) then
      for $x9 in $x6/child::title return
        for $x10 in $x6/child::review return <rev> $x9  $x10 </rev>
```

Como puede observarse, las expresiones doc("bib.xml") y doc("reviews.xml") han sido sustituidas por las variables `$x1` y `$x2` respectivamente. Ambas variables son libres en la consulta, siendo el valor de cada una de ellas el documento XML contenido en el fichemo XML asociado.

```
<?xml version='1.0'?>
<reviews>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
               A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
    <entry>
        <title>TCP/IP Illustrated</title>
        <price>65.95</price>
        <review>
               One of the best books on TCP/IP.
        </review>
    </entry>
</reviews>
```

Figura 4.10: Documento XML: reviews.xml.

La sintaxis descrita en la figura 4.9 no permite escribir consultas que contengan las instrucciones *for*, *where* y *return* habituales en XQuery, sin embargo es fácil definir un algoritmo que transforme una consulta XQuery que use *for*, *where* y *return* en otra consulta que siga la sintaxis SXQ. La idea es sustituir las referencias a documentos XML por variables nuevas e indexadas $\$x_1, \$x_2, \ldots$, que cada instrucción *for* vaya seguida de un *return*, las instrucciones *where* se transformen en *ifs* y las instrucciones XPath con más de dos pasos se descompongan en expresiones de un único paso aplicado a una variable nueva introducida por expresiones de tipo *for*.

A continuación describimos el concepto de variable libre en una consulta SXQ.

Definición 4.5.2. *Sea Q una consulta SXQ. El conjunto de variables libres de Q, denotado como free(Q), se define como el conjunto de variables no introducidas por sentencias* for*.*

Podemos asumir que el acceso a los documentos XML se realiza a través de las variables indexadas $\$x_1, \$x_2, \ldots$ del conjunto *free*(Q).

En [22] se presenta la semántica de XQ . Cada documento XML se representa como un *árbol de datos*, siendo un *bosque de datos* una secuencia de árboles de datos. Un *bosque indexado* es una tupla de dos elementos, un bosque de datos y una lista de nodos del bosque de datos.

A partir de esta semántica, en la figura 4.11 introducimos la semántica operacional de una expresión SXQ α con un máximo de k variables libres. La función $[\![\alpha]\!]_k$ toma

como parámetros de entrada un bosque de datos $\mathcal{F}$ y una $k-$tupla de nodos del bosque. Dicha función devuelve un bosque indexado. Cada una de las variables $\$x_i$, $i = 1 \ldots k$ está ligada al nodo n_i de la $k-$tupla de nodos del bosque de entrada. Esta semántica, a diferencia de la semántica para XQ presentada en [22], no incluye reglas para el constructor *lab*, ni para la consulta vacía ( ). Sin embargo, incluye una regla para el caso de las consultas representadas como una secuencia de variables dentro de una etiqueta.

XQ₁	$[\![\alpha\ \beta]\!]_k(\mathcal{F}, \overline{e})$	:=	$[\![\alpha]\!]_k(\mathcal{F}, \overline{e}) \uplus [\![\beta]\!]_k(\mathcal{F}, \overline{e})$
XQ₂	$[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(\mathcal{F}, \overline{e})$	:=	let $(\mathcal{F}', \overline{l}) = [\![\alpha]\!]_k(\mathcal{F}, \overline{e})$ in $\biguplus_{1 \leq i \leq \lvert\overline{l}\rvert} [\![\beta]\!]_{k+1}(\mathcal{F}', \overline{e} \cdot \overline{l}_i)$
XQ₃	$[\![\$x_i]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$	:=	$(\mathcal{F}, [t_i])$
XQ₄	$[\![\$x_i/\chi :: \nu]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$	:=	$(\mathcal{F}, \text{list of nodes } v \text{ such that } \chi^{\mathcal{F}}(t_i, v) \text{ and node } v \text{ has label } \nu \text{ in order } <_{doc}^{tree(t_i)})$
XQ₅	$[\![\text{if } \phi \text{ then } \alpha]\!]_k(\mathcal{F}, \overline{e})$	:=	if $\pi_2([\![\phi]\!]_k(\mathcal{F}, \overline{e})) \neq [\ ]$ then $[\![\alpha]\!]_k(\mathcal{F}, \overline{e})$ else $(\mathcal{F}, [\ ])$
XQ₆	$[\![\text{if } \$x_i = \$x_j \text{ then } \alpha]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$	:=	if $t_i = t_j$ then $[\![\alpha]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$ else $(\mathcal{F}, [\ ])$
XQ₇	$[\![\langle a\rangle\ \langle/a\rangle]\!]_k(\mathcal{F}, \overline{e})$	:=	$construct(a, (\mathcal{F}, [\ ]))$
XQ₈	$[\![\langle a\rangle\ tag\ \langle/a\rangle]\!]_k(\mathcal{F}, \overline{e})$	:=	$construct(a, [\![tag]\!]_k(\mathcal{F}, \overline{e}))$
XQ₉	$[\![\langle a\rangle\ \$x_i\ \ldots\ \$x_j\ \langle/a\rangle]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$	:=	$construct(a, (\mathcal{F}, [t_i, \ldots, t_j]))$

Figura 4.11: Semántica de SXQ .

El operador $construct(a, (\mathcal{F}, [w_1 ... w_n]))$, permite construir un nuevo árbol donde a es un nombre de etiqueta, $\mathcal{F}$ es un bosque de datos y la lista $[w_1 \ldots w_n]$ es una lista de nodos en $\mathcal{F}$. El operador *construct* devuelve el bosque indexado $(\mathcal{F} \cup T', [root(T')])$, donde T' es un árbol de datos cuyo nodo raíz $root(T')$ está etiquetado con a, y cada uno de los hijos de la raíz es cada uno de los subárboles de $\mathcal{F}$ cuya raíz se corresponde con el nodo w_i. La unión $\biguplus$ de dos bosques indexados $(\mathcal{F}_1, \overline{l}_1)$ y $, (\mathcal{F}_2, \overline{l}_2)$ devuelve un bosque indexado $(\mathcal{F}_1 \cup \mathcal{F}_2, \overline{l})$, donde $\overline{l}$ es la lista resultado de concatenar las listas de nodos $\overline{l}_1$ y $\overline{l}_2$.

Estas reglas constituyen un sistema de reescritura de términos (TRS [16]) terminante y confluente, donde cada regla define un paso de reducción simple. El símbolo $:=^*$ representa la clausura reflexiva y transitiva de $:=$. Esta semántica no modela la función document() de XQuery. En su lugar se asume la existencia de una o más variables iniciales ligadas a un nodo del bosque de datos inicial. Así, dada una consulta SXQ Q con $free(Q) = \{\$x_1, \ldots, \$x_k\}$, esta semántica evalúa la consulta Q partiendo de la expresión $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$, donde el bosque de datos inicial $\mathcal{F}$ es un bosque que contiene k documentos XML de entrada representados como árboles de datos como se explica en [22]. Cada variable $\$x_i$ en $free(Q)$ representa un documento XML y está ligada a un nodo del bosque de datos $\mathcal{F}$. La secuencia de nodos $t_1, \ldots, t_k$ de $\mathcal{F}$ se corresponde con la secuencia de nodos ligados a las variables $\{\$x_1, \ldots, \$x_k\}$. El resultado de la evaluación de una consulta es un bosque indexado de la forma

$(\mathcal{F}', [e_1, \ldots, e_m])$. La lista de los $m-$nodos de $\mathcal{F}'$ representan fragmentos de documentos XML.

4.5.2. Implementación de XQuery en $\mathcal{TOY}$

La implementación de XQuery en $\mathcal{TOY}$ usa los siguientes tipos de datos:

```
data sxq  = xfor xml sxq sxq | xif cond sxq | xmlExp xml |
                    xp path | comp sxq sxq
data cond = xml := xml | cond sxq
data path = var xml | xml :/ xPath | doc string xPath
```

El tipo de dato `sxq` permite representar consultas siguiendo la sintaxis de la figura 4.9. Como puede observarse, la variable introducida por la instrucción *for* tiene tipo `xml`, garantizando así que el valor de la variable es siempre de este tipo. Definimos en $\mathcal{TOY}$ la primitiva `parse_xquery` para representar una consulta SXQ mediante un término de tipo `sxq`. Dicha primitiva recibe como parámetro de entrada una consulta escrita siguiendo la sintaxis SXQ y devuelve otra consulta representada mediante un término $\mathcal{TOY}$. Por ejemplo, para representar en $\mathcal{TOY}$ la consulta SXQ del ejemplo 4.5.1 escribimos el siguiente objetivo:

```
Toy> parse_xquery  "for $x3 in $x1/child::bib return
                    for $x4 in .....  <rev> $x9  $x10 </rev>" == R
```

Como resultado, la función produce el siguiente término $\mathcal{TOY}$:

```
R --> xfor X3 (xp ( X1 :/ (child .::. (nameT "bib"))))
    (xfor X4 (xp ( X3 :/ (child .::.(nameT "book"))))
    (xfor X5 (xp ( X2 :/ (child .::.(nameT "reviews"))))
    (xfor X6 (xp ( X5 :/ (child .::.(nameT "entry"))))
    (xfor X7 (xp ( X4 :/ (child .::.(nameT "title"))))
    (xfor X8 (xp ( X6 :/ (child .::.(nameT "title"))))
    (xif  ( xp ( var X7 ) :=  xp ( var X8) )
    (xfor X9 (xp ( X6 :/ (child .::.(nameT "title"))))
    (xfor X10 (xp ( X6 :/ (child .::.(nameT "review"))))
       (xmlExp (xmlTag "rev" [] [X9, X10])))))))))))
```

Sin pérdida de la generalidad, la primitiva `parse_xquery` permite que el parámetro de entrada sea una consulta sin variables libres en el sentido de la definición 4.5.2.

```
sxq (xp E)              = sxqPath E
sxq (xmlExp X)          = X
sxq (comp Q1 Q2)        = sxq Q1
sxq (comp Q1 Q2)        = sxq Q2
sxq (xfor X Q1 Q2)      = sxq Q2 <== X== sxq Q1
sxq (xif (Q1:=Q2) Q3)   = sxq Q3 <== sxq Q1  == sxq Q2
sxq (xif (cond Q1) Q2) = sxq Q2 <== sxq Q1  == _

sxqPath (var X)  = X
sxqPath (X :/ S) = S X
sxqPath (doc F S) = S (load_xml_file F)
```

Figura 4.12: Intérprete de SXQ en $\mathcal{TOY}$.

Para conseguirlo basta con reemplazar cada una de las variables por el correpondiente fichero XML. Por ejemplo, las variables $\$x_1$ y $\$x_2$ pueden sustituirse por las cadenas doc("bib.xml") y doc("reviews.xml") respectivamente.

Para evaluar consultas SXQ en el sistema $\mathcal{TOY}$, definimos la función `sxq`, cuyas reglas se muestran en la figura 4.12. La función `sxq` distingue casos dependiendo de la forma de la consulta SXQ a evaluar. Si la consulta es una expresión XPath, se usa la función auxiliar `sxqPath`, cuyas reglas aparecen también en la figura 4.12. Si la consulta es una expresión XML, se devuelve esa misma expresión como resultado. Si tenemos dos consultas como argumentos de la constructora `comp`, la función `sxq` devuelve como resultado la evaluación de cualquiera de las dos de forma no determinista. La instrucción `for`, representada por la constructora `xfor` fuerza la evaluación de la consulta `Q1` ligando la variable `X` al resultado. El resultado de la evaluación de la instrucción `for` es el resultado de evaluar la consulta `Q2`.

El resultado de la consulta del ejemplo 4.5.1 se obtiene escribiendo el objetivo:

```
Toy> sxq (parse_xquery  "for $x1 in doc(\"bib.xml\")/child::bib return
                    for $x2 in .....  <rev> $x5  $x6 </rev>" ) == R
```

cuyo resultado es:

```
R -> (xmlTag  "rev" []
      [ (xmlTag  "title" []
           [ (xmlText  "TCP/IP Illustrated") ]),
        (xmlTag  "review" []
           [ (xmlText  "One of the best books on TCP/IP.") ]) ])
      Elapsed time: 63 ms.
```

```
   more solutions (y/n/d/a) [y]?
R -> (xmlTag  "rev" []
      [ (xmlTag  "title" []
           [ (xmlText  "Advanced Programming in the Unix environment") ]),
        xmlTag  "review" []
           [ (xmlText  "A clear and detailed discussion of UNIX programming.
        ") ]) ])
      Elapsed time: 46 ms.
   more solutions (y/n/d/a) [y]?
R -> (xmlTag  "rev" []
      [ (xmlTag  "title" []
           [ (xmlText  "Data on the Web") ]),
        (xmlTag  "review" []
           [ (xmlText  "A very good discussion of semi-structured database
              systems and XML.
        ") ]) ])
   more solutions (y/n/d/a) [y]?
      no
```

Ahora, si escribimos:

```
Toy> collect(sxq q1) == L,
     Sol == xmlTag "root" [] L ,
     write_xml_file Sol "salida.xml" == R
```

Se genera un fichero con nombre `"salida.xml"` con el siguiente contenido:

```
<rev>
 <title>TCP/IP Illustrated</title>
 <review>One of the best books on TCP/IP.</review>
</rev>
<rev>
 <title>Advanced Programming in the Unix environment</title>
 <review>A clear and detailed discussion of UNIX programming.</review>
</rev>
<rev>
 <title>Data on the Web</title>
 <review>A very good discussion of ... and XML.</review>
</rev>
```

A continuación enunciamos dos resultados acerca de la corrección y completitud de esta propuesta con respecto a la semántica operacional de XQuery definida en la

figura 4.11. Ambos resultados se pueden encontrar, junto con su demostración, en [7](B.2).

Teorema 4.5.3. *Sea P el programa $\mathcal{TOY}$ de la figura 4.12. Sea Q una consulta escrita mediante sintaxis SXQ con free$(Q) = \{\$x_1, \ldots, \$x_m\}$. Sea* `Q` *la representación de Q como un tipo de dato $\mathcal{TOY}$,* `t` *un patrón $\mathcal{TOY}$, y θ una sustitución tal que dom(θ) = free(Q) y $\mathcal{P} \vdash$ (*`sxq Q`θ `== t`*). Entonces, para todo bosque $\mathcal{F}$ que contenga los nodos $t_1, \ldots, t_m$ con $t_1 = \theta(x_1), \ldots, t_m = \theta(x_m)$, existe un bosque indexado $(\mathcal{F}', L')$ tal que:*

$$[\![Q]\!]_m(\mathcal{F}, t_1, \ldots, t_m) :=^* (\mathcal{F}', L')$$

verificando que $t \in L'$.

En el teorema 4.5.3, los nodos $t_1, \ldots, t_m$ representan los documentos XML de entrada de la consulta Q escrita en sintaxis SXQ, y nos enuncia que si `t` es el resultado de la consulta equivalente escrita en $\mathcal{TOY}$, entonces dicho resultado es uno de los devueltos por la consulta Q.

El teorema 4.5.4 añade que para todos los valores devueltos por la consulta Q escrita mediante sintaxis SXQ existe una prueba en el sistema $\mathcal{TOY}$.

Teorema 4.5.4. *Sea P el programa $\mathcal{TOY}$ de la figura 4.12. Sea Q una consulta escrita mediante sintaxis SXQ con free$(Q) = \{\$x_1, \ldots, \$x_k\}$. Sea* `Q` *la representación de Q como un tipo de dato $\mathcal{TOY}$. Supongamos que $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L)$ para algún $\mathcal{F}, \mathcal{F}', t_1, \ldots, t_k, L$. Entonces, para todo $t \in L$, existe una sustitución θ tal que $\theta(\$x_i) = t_i$ para todo $\$x_i \in$ free(Q) y una prueba CRWL para $\mathcal{P} \vdash$ (*`sxq Q`θ `== t`*).*

La demostración de ambos resultados se basa en la comparación entre la semántica declarativa CRWL de $\mathcal{TOY}$ definida en [70] y la semántica operacional de XQuery.

4.6. Casos de prueba para consultas XQuery

La implementación de XQuery en $\mathcal{TOY}$ permite generar fácilmente casos de prueba para consultas XQuery. Al igual que comentábamos en la sección 4.3, se trata de documentos XML de pequeño tamaño de tal forma que al consultar dicho documento mediante una expresión XQuery se produce algún resultado. De esta forma, el documento original y el caso de prueba pueden ser comparados, lo que facilita la detección de errores.

El proceso de generación de casos de prueba para consultas XQuery se describe como sigue:

1. Aplicamos la primitiva `parse_xquery` para transformar la consulta SXQ Q en el término $\mathcal{TOY}$ `Q'`.

2. Sean $F_1, \ldots, F_k$ los nombres de los documentos XML que aparecen en la consulta `Q'`.

3. Sea `Q"` la consulta obtenida después de sustituir cada expresión de la forma `doc(`F_i`)` por una nueva variable lógica D_i, con $i = 1 \ldots k$.

```
prepareTC (xp E)             = (xp E',L)
                               where (E',L) = prepareTCPath E
prepareTC (xmlExp X)         = (xmlExp X, [])
prepareTC (comp Q1 Q2)       = (comp Q1' Q2', L1++L2)
                                where (Q1',L1) = prepareTC Q1
                                      (Q2',L2) = prepareTC Q2
prepareTC (xfor X Q1 Q2)     = (xfor X Q1' Q2', L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
prepareTC (xif (Q1:=Q2) Q3) = (xif (Q1':=Q2') Q3',L1++(L2++L3))
                                where (Q1',L1) = prepareTC Q1
                                      (Q2',L2) = prepareTC Q2
                                      (Q3',L3) = prepareTC Q3
prepareTC (xif (cond Q1) Q2) = (xif (cond Q1) Q2, L1++L2)
                                where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2

prepareTCPath (var X)   = (var X, [])
prepareTCPath (X :/ S)  = (X :/ S, [])
prepareTCPath (doc F S) = (A :/ S, [write_xml_file A ("tc"++F)])
```

Figura 4.13: Reglas de la función `prepareTC`.

4. Sea `"expected.xml"` el documento XML que contiene una posible respuesta para la consulta `Q'`.

5. Finalmente, escribir el objetivo:

```
Toy> sxq Q'' == load_doc "expected.xml",
                write_xml_file D1 F1',
                ... ,
                write_xml_file Dk Fk'
```

La idea es que el objetivo indicado en el paso 5 tiene éxito cuando encuentra valores para las variables lógicas D_i, con $i = 1 \ldots k$, que satisfacen la igualdad estricta. Como resultado, cada variable lógica D_i contiene un documento XML, de tal forma que la consulta `Q"` devuelve como resultado la respuesta esperada y contenida en el fichero "expected.xml". Cada uno de los documentos puede ser grabado como un fichero XML. El conjunto de los documentos generados constituyen un caso de prueba para la consulta XQuery Q.

Para automatizar el proceso definimos en $\mathcal{TOY}$ la función `generateTC` cuyos parámetros de entrada son una consulta XQuery Q y el nombre del fichero XML que contiene la respuesta esperada para la consulta.

```
generateTC Q F = true <==  sxq Qtc == load_doc F, L==_
               where (Qtc,L) = prepareTC (parse_xquery Q)
```

La función `generateTC` usa la función `prepareTC` descrita en la figura 4.13 encargada de realizar los pasos 2-4 del proceso descrito anteriormente. En el caso de que no existan valores para todas las variables D_i, la función `generateTC` podría entrar en un bucle infinito.

Ejemplo 4.6.1. *La siguiente consulta XQuery Q es errónea y no devuelve ningún resultado:*

```
for $b in doc("bib.xml")/bib/book,
       $r in doc("reviews.xml")/reviews/entry,
       where $b/title = $r/title
       for $booktitle in $r/title,
           $revtext in $r/review,
           $publisher in $r/publisher
       return  <rev> $booktitle $publisher $revtext </rev>
```

El error se encuentra en la expresión `$r/publisher`*; la variable* `$r` *debería ser* `$b`*. Supongamos que el usuario no detecta el error en la consulta y escribe el siguiente documento XML "expected.xml", el cual representa la respuesta esperada por el usuario para la consulta:*

```
<rev>
 <title>Some title</title>
 <review>The review</review>
 <publisher>Publisher</publisher>
</rev>
```

Si escribimos el siguiente objetivo en $\mathcal{TOY}$:

```
Toy>  generateTC Q "expected.xml" == R
```

el sistema genera los siguientes documentos XML, los cuales constituyen un caso de prueba.

```
% bibtc.xml
<bib>
 <book>
  <title>Some title</title>
 </book>
</bib>
```

```
% revtc.xml
<reviews>
 <entry>
  <title>Some title</title>
  <review>The review </review>
  <publisher>Publisher</publisher>
 </entry>
</reviews>
```

Comparando el caso de prueba generado "revtc.xml" con el documento "reviews.xml" observamos que el elemento publisher no aparece en el documento "reviews.xml". Sin embargo, en la consulta se busca el elemento publisher en el documento "reviews.xml" en lugar de en el documento "bib.xml", lo cual es un error.

4.7. Conclusiones

En este capítulo, dedicado a la generación de casos de prueba y a la depuración de consultas XQuery, hemos seguido una línea distinta al caso de las bases de datos SQL, optando por representar el lenguaje XQuery en el sistema lógico-funcional $\mathcal{TOY}$. El esfuerzo de representación se ve compensado por la sencillez con la que se resuelven la generación de casos de prueba y la depuración en el nuevo marco. Se presentan dos implementaciones diferentes de los lenguajes XPath y XQuery en el lenguaje lógico-funcional $\mathcal{TOY}$. En primer lugar se presenta una implementación de XPath donde los constructores básicos se definen mediante funciones $\mathcal{TOY}$ aplicadas a términos XML. Aunque XPath ya ha sido incorporado en otros lenguajes, la ventaja de esta representación con respecto a otras propuestas es que permite aplicar características propias del lenguaje $\mathcal{TOY}$ para generar casos de prueba en forma de documentos XML y para depurar respuestas erróneas y respuestas perdidas. En el caso de respuestas erróneas, se propone una implementación que permite obtener la secuencia de los pasos intermedios que representan el camino desde la raíz del documento hasta el nodo erróneo, lo que facilita la localización de la causa del error. En el caso de respuestas perdidas, se propone una implementación que analiza la secuencia de pasos que define la consulta XPath con la propósito de localizar el primer paso de la consulta que no devuelve ningún resultado, y que normalmente es la causa del error. Un posible trabajo futuro podría consistir en la definición de un depurador declarativo para consultas XPath basado en estas ideas tan sencillas, o incluso su incorporación a librerías de XPath ya existentes.

Por otra parte, y en lo que respecta a XQuery, nos encontramos con que XQuery permite definir estructuras más complejas que XPath, las cuales no son fácilmente representables mediante funciones de orden superior. Como alternativa a la representación de XPath por medio de funciones $\mathcal{TOY}$, proponemos una representación puramente declarativa de un subconjunto de los lenguajes XPath y XQuery repre-

sentando las consultas mediante tipos de datos. Esta propuesta tiene la ventaja con respecto a la anterior de que nos permite probar de forma sencilla resultados de corrección y completitud de la implementación en $\mathcal{TOY}$ con respecto a la semántica operacional de XQuery. Actualmente esta propuesta no incluye la directiva *let* tan habitual en XQuery y se deja como un posible trabajo futuro. La inclusión de *let* en nuestra propuesta implicaría introducir en la semántica CRWL la primitiva *collect*, cuestión no trivial por la interacción de la recolección de respuestas con la semántica del indeterminismo adoptada en CRWL y la evaluación perezosa. En XQuery hemos obtenido casos de prueba aprovechando las características de "generación y prueba"propias de los lenguajes lógico-funcionales. Sin embargo, esta sencilla técnica tiene la desventaja de que no se puede asegurar que siempre se obtenga un caso de prueba, ya que en ocasiones el mecanismo de resolución entra en un bucle infinito, produciendo documentos XML que incluyen cada vez más copias de un elemento que a continuación es rechazado en la parte de prueba. Una búsqueda exhaustiva de casos de prueba requeriría el empleo de una técnica de generación de casos de prueba similar a la empleada en el caso de SQL.

Aunque hemos visto distintos mecanismos para la depuración de consultas XPath, no contemplamos en este documento la depuración de consultas XQuery. En este sentido y puesto que $\mathcal{TOY}$ ya incorpora un depurador declarativo [42], pensamos que éste se podría utilizar como base para el desarrollo de un nuevo depurador añadiendo simplemente un interfaz que permitiera mostrar los resultados en un formato cómodo para los usuarios de XQuery. Sin embargo, al igual que ocurría en el caso de las bases de datos relacionales, el tamaño de los resultados intermedios es muy elevado, por lo que de nuevo, una solución más realista al problema de la depuración de consultas XQuery pasaría por la aplicación de técnicas similares a las del capítulo anterior.

Publicaciones asociadas

(A.5) **Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 13th International Symposium of Practical Aspects of Declarative Languages (PADL 2011), volume 6539 of *Lecture Notes in Computer Science*, pages 145–159, Austin, Texas, USA, January 24–25, 2011. Springer-Verlag.

→ **Página** 181

(A.6) **A Declarative Embedding of XQuery in a Functional-Logic Language**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 21st International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2011), volume 7225 of *Lecture Notes in Computer Science*, pages 42–56, Odense, Denmark, July 18–20, 2011. Springer-Verlag.

→ **Página** 196

(B.2) **Embedding XQuery in Toy**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

Technical Report SIC-04-11. Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain, 2011. Versión Extendida con contenido extra de "*A Declarative Embedding of XQuery in a Functional-Logic Language*, (LOPSTR 2011)".

→ **Página** 300

(A.8) **XQuery in the Functional-Logic Language Toy**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), volume 6816 of *Lecture Notes in Computer Science*, pages 35–51, Odense, Denmark, July 19, 2011. Springer-Verlag.

→ **Página** 226

(A.9) **XPath Query Processing in a Functional-Logic Language**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In XI Jornadas sobre Programación y Lenguajes, a Coruña, Spain, September 5–7 , 2011. Editors: Puri Arenas, Victor Gulías and Pablo Nogueira. Volume 282 of Electronic Notes in Theoretical Computer Science, pages 31–45, 2012.

→ **Página** 243

5 | Conclusiones

Actualmente, existe una presencia notable de las bases de datos en todo tipo de aplicaciones, siendo el modelo de base de datos relacional uno de los más conocidos y extendidos, tanto en el mundo académico como en el mundo empresarial. Son una parte fundamental de todas las organizaciones, pues en ellas se almacena información crucial para el buen funcionamiento de las mismas. En lo que respecta a las bases de datos deductivas y dado el gran volumen de información que manejan las aplicaciones, en los últimos años se ha producido un resurgimiento de lenguajes como Datalog, que es usado como una herramienta sencilla y muy potente para la realización de consultas. Por otro lado, y con la creciente popularidad de los servicios Web, XML se ha convertido prácticamente de facto en el estándar para la transmisión de datos y una alternativa a las ya conocidas bases de datos relacionales.

Sin embargo, se han constatado que los sistemas gestores de bases de datos disponibles no incluyen apenas herramientas de *testing*, y cuando lo hacen son muy dependientes del mecanismo de ejecución y muy poco intuitivas para los desarrolladores. En esta tesis se ha diseñado y desarrollado una serie de técnicas que facilitan la detección y diagnosis de errores en el campo de las bases de datos, y en particular en la definición de las consultas. Para ello se han aplicado técnicas similares ya que, a pesar de sus diferencias, los distintos lenguajes de acceso a los datos comparten elementos comunes, como es su carácter declarativo, en el sentido de estar alejados de los detalles del SGBD, o la capacidad para definir consultas combinando el resultado de subconsultas previamente definidas. Esto resulta muy positivo, ya que las mismas optimizaciones pueden resultar útiles para los distintos lenguajes.

Se han seguido dos puntos vista: la generación automática de casos de prueba y la depuración declarativa. Hemos dividido y presentado este trabajo en tres bloques, cada uno de los cuales desarrolla un marco teórico para un tipo diferente de bases de datos.

En el caso de la generación automática de casos de prueba, nos hemos centrado en la generación de casos de prueba mediante la técnica de ejecución simbólica para el caso de consultas SQL definidas a partir de vistas. Partiendo de un esquema de base de datos, un conjunto de vistas correlacionadas y un conjunto de condiciones de integridad, se construye un conjunto de fórmulas que serán traducidas posteriormente a un lenguaje de restricciones, lo que constituye un problema de satisfacción de restricciones. La solución a dicho problema, en el caso de que sea satisfactible, representa un caso de prueba. Este mismo esquema, aunque no discutido en este texto,

sería aplicable al caso particular de la generación de casos de prueba para el caso de Datalog. En el caso de consultas XQuery, hemos optado por aprovechar la inmersión en el lenguaje $\mathcal{TOY}$ para obtener casos de prueba directamente a partir de la técnica conocida como "generación y prueba", natural en estos lenguajes. De todas formas y como trabajo futuro sería interesante plantear una técnica basada en ejecución simbólica también para XQuery. Aunque menos directa, esta solución tendría la ventaja de evitar los casos de no terminación que se encuentran en la propuesta actual.

Desde el punto de vista de la depuración, tanto para el caso de las bases de datos deductivas como para el caso de las bases de datos relacionales, se ha definido una instancia del modelo general de depuración declarativa y se ha aplicado técnicas que permiten mejorar la eficiencia y practicabilidad del método presentado. En este sentido y como trabajo futuro, sería interesante estudiar la posibilidad de aplicar otras técnicas, como es el análisis de la procedencia de los resultados de las consultas (*tracking provenance information*) [68, 53] o el uso de programas CHR [64]. En el caso de XML y como para el caso de la generación automática de casos de prueba, se han aprovechado las características y particularidades de la inmersión de XPath en el lenguaje $\mathcal{TOY}$, como son el uso de variables y las funciones de orden superior, para ofrecer facilidades a la hora de depurar consultas y localizar errores. Como trabajo futuro se podría profundizar en esta línea, por ejemplo se podrían utilizar las soluciones propuestas para el tratamiento de respuestas perdidas y respuestas incorrectas en la definición de un depurador declarativo, así como su incorporación en cualquiera de las librerías de XPath ya existentes.

Referencias

[1] Abiteboul, S., Bourhis, P. y Marinoiu, B. Efficient Maintenance Techniques for Views over Active Documents. En *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, páginas 1076–1087. ACM Press, New York, NY 10036, USA, 2009.

[2] Abiteboul, S., Hull, R. y Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Agrawal, H. y Horgan, J. R. Dynamic Program Slicing. *ACM SIGPLAN Notices*, vol. 25, páginas 246–256, 1990.

[4] Almendros-Jiménez, J. M. An Encoding of XQuery in Prolog. En *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies*, vol. 5679 de *XSym '09*, páginas 145–155. Springer-Verlag, Berlin, Heidelberg, 2009.

[5] Almendros-Jiménez, J. M., Becerra-Terón, A. y Enciso-Baños, F. J. Querying XML documents in Logic Programming. *Journal of Theory and Practice of Logic Programming*, vol. 8(3), páginas 323–361, 2008.

[6] Almendros-Jiménez, J. M., Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. A Declarative Embedding of XQuery in a Functional-Logic Language. En *LOPSTR 2011*, vol. 7225 de *Lecture Notes in Computer Science*, páginas 42–56. Springer-Verlag, Berlin, Heidelberg, 2011.

[7] Almendros-Jiménez, J. M., Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Embedding of XQuery in Toy. Informe Técnico SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid, 2011. `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-4-11.pdf`.

[8] Almendros-Jiménez, J. M., Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. XPath Query Processing in a Functional-Logic Language. En *XI Jornadas sobre Programación y Lenguajes, PROLE2011 (SISTEDES)*, páginas 31–45. 2011.

[9] Almendros-Jiménez, J. M., Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. XQuery in the Functional-Logic Language Toy. En *Functional and*

Constraint Logic Programming (editado por H. Kuchen), vol. 6816 de *Lecture Notes in Computer Science*, páginas 35–51. Springer-Verlag, Berlin, Heidelberg, 2011.

[10] Almendros-Jimenez, J. M., Silva, J. y Tamarit, S. XQuery Optimization Based on Program Slicing. En *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, páginas 1525–1534. ACM Press, New York, NY 10036, USA, 2011.

[11] Altova. XQuery Debugger. 2013. `http://www.altova.com/es/xmlspy/xquery-debugger.html`.

[12] Ammann, P. y Offutt, J. *Introduction to Software Testing*. Cambridge University Press, 2008.

[13] ApexSQL Debug. ApexSQL Debug . 2011. `http://www.apexsql.com/sql_tools_debug.aspx/`.

[14] Apt, K. R., Blair, H. A. y Walker, A. Towards a Theory of Declarative Knowledge. En *Foundations of Deductive Databases and Logic Programming* (editado por J. Minker), páginas 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[15] Atanassow, F., Clarke, D. y Jeuring, J. UUXML: A Type-Preserving XML Schema Haskell Data Binding. En *Proceedings of Practical Aspects of Declarative Languages*, vol. 3057, páginas 71–85. Springer-Verlag, Berlin, Heidelberg, 2004.

[16] Baader, F. y Nipkow, T. *Term Rewriting and All That*. Cambridge University Press, 1999.

[17] Bancilhon, F., Maier, D., Sagiv, Y. y Ullman, J. D. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). En *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, páginas 1–15. ACM Press, New York, NY 10036, USA, 1986.

[18] Baral, C. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY 10036, USA, 2003.

[19] Baumgartner, R., Flesca, S. y Gottlob, G. The Elog Web Extraction Language. En *Proceedings of the Artificial Intelligence on Logic for Programming*, LPAR '01, páginas 548–560. Springer-Verlag, London, UK, 2001.

[20] Beeri, C. y Ramakrishnan, R. On the Power of Magic. En *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, páginas 269–284. 1987.

[21] Benchmark Factory for Databases. Benchmark Factory for Databases. 2012. `http://www.quest.com/benchmark-factory/`.

[22] Benedikt, M. y Koch, C. From XQuery to Relational Logics. *ACM Transactions on Database Systems (TODS)*, vol. 34, páginas 25:1–25:48, 2009.

[23] Benzaken, V., Castagna, G. y Frish, A. CDuce: an XML-centric general-purpose language. En *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, páginas 51–63. ACM Press, New York, NY 10036, USA, 2005.

[24] Benzaken, V., Castagna, G. y Miachon, C. A Full Pattern-based Paradigm for XML Query Processing. En *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, vol. 3350 de *Lecture Notes in Computer Science*, páginas 235–252. Springer-Verlag, Heidelberg,Germany, 2005.

[25] Binnig, C., Kossmann, D. y Lo, E. Reverse Query Processing. En *IEEE 23rd International Conference on Data Engineering*, páginas 506–515. 2007.

[26] Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. y Teubner, J. MonetDB/XQuery: a fast XQuery Processor Powered by a Relational Engine. En *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, páginas 479–490. ACM New York, NY, USA, 2006.

[27] Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. y Teubner, J. Pathfinder: XQuery - The Relational Way. En *Proceedings of the International Conference on Very Large Databases*, páginas 1322–1325. ACM Press, New York, NY 10036, USA, 2005.

[28] Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H. y Woltran, S. Debugging ASP programs by means of ASP. En *Proceedings of the 9th international conference on Logic programming and nonmonotonic reasoning*, LPNMR'07, páginas 31–43. Springer-Verlag, Berlin, Heidelberg, 2007.

[29] Bry, F. y Schaffert, S. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. En *Proceedings of Web, Web-Services, and Database Systems*, vol. 2593 de *Lecture Notes in Computer Science*, páginas 295–310. Springer-Verlag, Berlin, Heidelberg, 2002.

[30] Caballero, R. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. En *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, páginas 8–13. ACM Press, New York, NY 10036, USA, 2005.

[31] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. A New Proposal for Debugging Datalog Programs. *Electronic Notes in Theoretical Computer Science*, vol. 216, páginas 79–92, 2008.

[32] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. A Theoretical Framework for the Declarative Debugging of Datalog Programs. En *SDKB 2008*, vol. 4925 de *Lecture Notes in Computer Science*, páginas 143–159. Springer-Verlag, Berlin, Heidelberg, 2008.

[33] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Applying Constraint Logic Programming to SQL Test Case Generation. En *Proceedings of International Symposium on Functional and Logic Programming (FLOPS'10)*, vol. 6009 de *Lecture Notes in Computer Science*, páginas 191–206. Springer-Verlag, Berlin, Heidelberg, 2010.

[34] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Integrating XPath with the Functional-Logic Language Toy (Extended Version). Informe Técnico SIC-05/10, Facultad de Informática, Universidad Complutense de Madrid, 2010. `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-5-10.pdf`.

[35] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Integrating XPath with the Functional-Logic Language Toy. En *Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages*, vol. 6539 de *PADL'11*, páginas 145–159. Springer-Verlag, Berlin, Heidelberg, 2011.

[36] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Algorithmic Debugging of SQL Views. En *Proceedings of the 8th International Andrei Ershov Memorial Conference, PSI 2011*, vol. 7162 de *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2012.

[37] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Declarative Debugging of Wrong and Missing Answers for SQL Views. En *Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012)*, vol. 7294 de *Lecture Notes in Computer Science*, páginas 73–87. Springer-Verlag, Berlin, Heidelberg, 2012.

[38] Caballero, R., García-Ruiz, Y. y Sáenz-Pérez, F. Declarative Debugging of Wrong and Missing Answers for SQL Views. Informe Técnico SIC-10/12, Facultad de Informática, Universidad Complutense de Madrid, 2012. `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/flops2012TR.pdf`.

[39] Caballero, R., Hermanns, C. y Kuchen, H. Algorithmic Debugging of Java Programs. *Electronics Notes in Theorical Computer Science*, vol. 177, páginas 75–89, 2007.

[40] Caballero, R., López-Fraguas, F. y Rodríguez-Artalejo, M. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. En *Proceedings of FLOPS'01*, número 2024 en Lecture Notes in Computer Science, páginas 170–184. Springer-Verlag, Berlin, Heidelberg, 2001.

[41] Caballero, R. y Rodríguez-Artalejo, M. A Declarative Debugging System for Lazy Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, vol. 64, páginas 113–175, 2002.

[42] Caballero, R. y Rodríguez-Artalejo, M. DDT: a Declarative Debugging Tool for Functional-Logic Languages. En *Proceedings of the 7th International*

Symposium on Functional and Logic Programming (FLOPS'04), Lecture Notes in Computer Science, páginas 70–84. Springer-Verlag, Berlin, Heidelberg, 2004.

[43] Caballero, R., Stuckey, P. J. y Tenorio-Fornés, A. Finite type extensions in constraint programming. En *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, páginas 217–227. ACM, New York, NY, USA, 2013.

[44] Cabeza, D. y Hermenegildo, M. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, vol. 1(3), páginas 251–282, 2001.

[45] Cadar, C. y Sen, K. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, vol. 56(2), páginas 82–90, 2013.

[46] Chamberlin, D., Robie, J., Dyck, M. y Snelson, J. XQuery 3.0: An XML Query Language. 2013. `http://www.w3.org/TR/xquery-30/`.

[47] Chandra, A. K. y Harel, D. Horn Clauses Queries and Generalizations. *The Journal of Logic Programming*, vol. 2(1), páginas 1–15, 1985.

[48] Chang, C. y Keisler, H. Model theory. En *Studies in Logic and Foundations of Mathematics 73*. North-Holland, 1973.

[49] Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I. y Weyuker, E. J. An agenda for testing relational database applications: Research articles. *Software Testing, Verification and Reliability*, vol. 14, páginas 17–44, 2004.

[50] Clark, J. XML Path Language (XPath) 2.0. 2007. `http://www.w3.org/TR/xpath20/`.

[51] Codd, E. Relational Completeness of Data Base Sublanguages. En *Data base Systems* (editado por Rustin), Courant Computer Science Symposia Series 6. Englewood Cliffs, N.J. Prentice-Hall, 1972.

[52] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, vol. 13, páginas 377–387, 1970.

[53] Cui, Y., Widom, J. y Wiener, J. L. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, vol. 25, páginas 179–227, 2000.

[54] Date, C. J. y Darwen, H. *A Guide to SQL Standard, 4th Edition*. Addison-Wesley, 1997.

[55] Degrave, F., Schrijvers, T. y Vanhoof, W. Towards a Framework for Constraint-based Test Case Generation. En *Proceedings of the 19th International Conference on Logic-Based Program Synthesis and Transformation*, LOPSTR'09, páginas 128–142. Springer-Verlag, Berlin, Heidelberg, 2010.

[56] DeMillo, R. A. y Offutt, A. J. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, vol. 17(9), páginas 900–910, 1991.

[57] Dietrich, S. W. Extension Tables: Memo Relations in Logic Programming. En *Proceedings of the 4th IEEE Symposium on Logic Programming*, páginas 264–272. 1987.

[58] Embarcadero Technologies on the Web. Rapid SQL Developer Debugger. 2011. `http://docs.embarcadero.com/products/rapid_sql/`.

[59] Emer, M. C. F. P., Vergilio, S. R. y Jino, M. A Testing Approach for XML Schemas. En *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, páginas 57–62. IEEE Computer Society, Washington, DC, USA, 2005.

[60] EMS Data Generator. EMS Data Generator. 2012. `http://download2.sqlmanager.net/download/datasheets/products/datagenerator/en/datagenerator.pdf`.

[61] Fegaras, L. HXQ: A Compiler from XQuery to Haskell. 2010.

[62] Fegaras, L. Propagating Updates Through XML Views Using Lineage Tracing. En *IEEE 26th International Conference on Data Engineering (ICDE)*, páginas 309 –320. 2010.

[63] Ferrand, G. Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method. *The Journal of Logic Programming*, vol. 4(3), páginas 177–198, 1987.

[64] Frühwirth, T. *Constraint Handling Rules*. Cambridge University Press, New York, NY 10036, USA, 2009.

[65] Gallaire, H. y Minker, J. Logic and Databases: a Deductive Approach. *ACM Computing Surveys*, vol. 16(2), páginas 153–185, 1984.

[66] Garcia-Molina, H., Ullman, J. D. y Widom, J. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[67] Gelfond, M. y Lifschitz, V. The Stable Model Semantics for Logic Programming. En *Proceedings of the 5th International Conference on Logic Programming* (editado por R. A. Kowalski y K. Bowen), páginas 1070–1080. The MIT Press, Cambridge, Massachusetts, 1988.

[68] Glavic, B. y Alonso, G. Provenance for Nested Subqueries. En *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, páginas 982–993. ACM Press, New York, NY 10036, USA, 2009.

[69] Gogolla, M. A note on the translation of SQL to tuple calculus. *SIGMOD Record*, vol. 19(1), páginas 18–22, 1990.

[70] González-Moreno, J., Hortalá-González, M., López-Fraguas, F. y Rodríguez-Artalejo, M. A Rewriting Logic for Declarative Programming. En *Proceedings of the European Symposium on Programming (ESOP'96)*, vol. 1058 de *Lecture Notes in Computer Science*, páginas 156–172. Springer, 1996.

[71] Grefen, P. W. y de, R. A. B. A Multi-Set Extended Relational Algebra: a Formal Approach to a Practical Issue. En *10th International Conference on Data Engineering, ICDE 1994*, páginas 80–88. IEEE, 1994.

[72] Grinev, M. y Pleshachkov, P. Rewriting-Based Optimization for XQuery Transformational Queries. En *Proceedings of the 9th International Database Engineering & Application Symposium*, páginas 163–174. IEEE Computer Society, 2005.

[73] Guerra, R., Jeuring, J. y Swierstra, S. D. Generic validation in an XPath-Haskell data binding. En *Proceedings Plan-X*. 2005.

[74] Hanus, M. Curry: An Integrated Functional Logic Language (version 0.8 of april 15, 2003). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[75] Hanus, M. Declarative Processing of Semistructured Web Data. En *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)* (editado por J. Gallagher y M. Gelfond), vol. 11 de *Leibniz International Proceedings in Informatics (LIPIcs)*, páginas 198–208. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2011.

[76] Hosoya, H. y Pierce, B. C. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, vol. 3(2), páginas 117–148, 2003.

[77] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. y Peterson, J. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, vol. 27(5), páginas 1–164, 1992.

[78] IBM. ILOG CP 1.4. 2009. `http://www.ilog.com/products/cp/`.

[79] International Organization for Standardization. *ISO/IEC 9075:1992: Information technology — Database languages — SQL* — . International Organization for Standardization, 1992.

[80] King, J. C. Symbolic Execution and Program Testing. *Communications of the ACM*, vol. 19(7), páginas 385–394, 1976.

[81] Koch, C. On the role of composition in XQuery. En *6th International Workshop on the Web and Databases (WebDB 2005)*. Baltimore, MD, 2005.

[82] Lee, S. C. L. y Offutt, J. Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis. En *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE '01, página 200. IEEE Computer Society, Washington, DC, USA, 2001.

[83] Lloyd, J. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[84] López-Fraguas, F. J. y Hernández, J. S. $\mathcal{TOY}$: A Multiparadigm Declarative System. En *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, páginas 244–247. Springer-Verlag, London, UK, 1999.

[85] Maier, D. *The Theory of Relational Databases*. Computer Science Press, 1983.

[86] Marian, A. y Simeon, J. Projecting XML Documents. En *Proceedings of International Conference on Very Large Databases*, páginas 213–224. Morgan Kaufmann, Burlington, USA, 2003.

[87] May, W. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, vol. 4(3), páginas 239–287, 2004.

[88] Moreno, J. C. G., Hortalá-González, M. T., López-Fraguas, F. J. y Rodríguez-Artalejo, M. An Approach to Declarative Programming Based on a Rewriting Logic. *The Journal of Logic Programming*, vol. 40(1), páginas 47–87, 1999.

[89] Moreno, J. C. G., Hortalá-González, M. T. y Rodríguez-Artalejo, M. A Higher Order Rewriting Logic for Functional Logic Programming. En *ICLP*, páginas 153–167. 1997.

[90] Naish, L. Declarative Diagnosis of Missing Answers. *New Generation Computing*, vol. 10(3), páginas 255–286, 1992.

[91] Naish, L. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, vol. 3, 1997.

[92] Nilsson, H. How to Look Busy while Being as Lazy as Ever: The Implementation of a Lazy Functional Debugger. *Journal of Functional Programming*, vol. 11(6), páginas 629–671, 2001.

[93] Olteanu, D. SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, páginas 934–949, 2007.

[94] Olteanu, D., Meuss, H., Furche, T. y Bry, F. Symmetry in XPath. Informe Técnico Research Report PMS-FB-2001-16, Computer Science Institute, Munich, Germany, 2001.

[95] Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edición, 2001.

[96] Ramakrishnan, R. y Ullman, J. A Survey of Research on Deductive Database Systems. *The Journal of Logic Programming*, vol. 23(2), páginas 125–149, 1993.

[97] de la Riva, C., Suárez-Cabal, M. J. y Tuya, J. Constraint-based Test Database Generation for SQL Queries. En *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, páginas 67–74. ACM Press, New York, NY 10036, USA, 2010.

[98] Ronen, R. y Shmueli, O. Evaluation of datalog extended with an XPath predicate. En *9th ACM International Workshop on Web Information and Data Management (WIDM 2007)*, páginas 9–16. 2007.

[99] Russo, F. y Sancassani, M. A Declarative Debugging Environment for DATALOG. En *Proceedings of the First Russian Conference on Logic Programming*, páginas 433–441. Springer-Verlag, London, UK, 1992.

[100] Sáenz-Pérez, F. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science*, vol. 271, páginas 63–78, 2011.

[101] Sáenz-Pérez, F. Datalog Educational System. User's Manual version 3.2. 2013. Available from `http://des.sourceforge.net/`.

[102] Schaffert, S. y Bry, F. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. Informe Técnico PMS-FB-2002-11, Computer Science Institute, Munich, Germany, 2002.

[103] Schulte, C., Lagerkvist, M. Z. y Tack, G. Gecode. 2012. `http://www.gecode.org/`.

[104] Seipel, D. Processing XML-Documents in Prolog. En *Procs. of the Workshop on Logic Programming 2002*. Technische Universität Dresden, Dresden, Germany, 2002.

[105] Shapiro, E. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.

[106] Silva, J. A Comparative Study of Algorithmic Debugging Strategies. En *Proceedings of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006*, páginas 134–140. 2007.

[107] SQL Ultimate Debugger. SQL Ultimate Debuggerr. 2011. `http://www.sqlsolutions.com/products/sql-ultimate-debugger/index.html`.

[108] Sterling, L. y Shapiro, E. *The art of Prolog: advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1986.

[109] Sulzmann, M. y Lu, K. Z. Xhaskell — adding regular expression types to haskell. En *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, páginas 75–92. Springer-Verlag, Berlin, Heidelberg, 2008.

[110] Syrjänen, T. Debugging Inconsistent Answer Set Programs. En *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning*, páginas 77–84. Lake District, UK, 2006.

[111] Tessier, A. y Ferrand, G. Declarative Diagnosis in the CLP Scheme. En *Analysis and Visualization Tools for Constraint Programming*, número 1870 en Lecture Notes in Computer Science, capítulo 5, páginas 151–174. Springer-Verlag, Berlin, Heidelberg, 2000.

[112] Thiemann, P. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, vol. 12(4&5), páginas 435–468, 2002.

[113] Tsang, E. P. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[114] Tuya, J., Suárez-Cabal, M. J. y de la Riva, C. Full predicate coverage for testing sql database queries. *Software Testing, Verification and Reliability*, vol. 20, páginas 237–288, 2010.

[115] Ullman, J. *Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1995.

[116] Ullman, J. D. *Principles of Database and Knowledge-base Systems, Vol. I*. Computer Science Press, Inc., New York, NY 10036, USA, 1988.

[117] W3C. Extensible Markup Language (XML). 2007. `http://www.w3.org/XML/`.

[118] W3C. XML Query Use Cases. 2007. `http://www.w3.org/TR/xquery-use-cases/`.

[119] W3C. XML Schema 1.1. 2010. `http://www.w3.org/XML/Schema`.

[120] Wallace, M. y Runciman, C. Haskell and XML: generic combinators or type-based translation? *ACM SIGPLAN Notices*, vol. 34(9), páginas 148–159, 1999.

[121] Walmsley, P. *XQuery*. O'Reilly Media, Inc., 2007.

[122] Wielemaker, J. SWI-Prolog SGML/XML Parser, Version 2.0.5. Informe técnico, Human Computer-Studies (HCS), University of Amsterdam, 2005.

[123] Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S. y Zicari, R. *Advanced Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[124] Zhang, J., Xu, C. y Cheung, S. C. Automatic Generation of Database Instances for White-box Testing. En *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, páginas 161–165. IEEE Computer Society, Washington, DC, USA, 2001.

A | Publicaciones

(A.1) **A New Proposal for Debugging Datalog Programs**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2007), volume 216 of *Electronic Notes in Theoretical Computer Science*, pages 79–92, Paris, France, June 25 2007.

→ **Página** 125

(A.2) **A Theoretical Framework for the Declarative Debugging of Datalog Programs**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In 3rd International Workshop on Semantics in Data and Knowledge Bases (SDKB 2008), volume 4925 of *Lecture Notes in Computer Science*, pages 143–159, Nantes, France, March 29, 2008. Springer-Verlag.

→ **Página** 139

(A.3) **Applying Constraint Logic Programming to SQL Test Case Generation**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In Tenth International Symposium on Functional and Logic Programming (FLOPS 2010), volume 6009 of *Lecture Notes in Computer Science*, pages 191–206, Sendai, Japan, April 19–21 2010. Springer-Verlag.

→ **Página** 156

(A.4) **Algorithmic Debugging of SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In Perspectives of Systems Informatics: 8th International Andrei Ershov Memorial Conference, (PSI 2011), volume 7162 of *Lecture Notes in Computer Science*, pages 77–85, Novosibirsk, Russia, June 27 – July 1, 2011. Springer-Verlag.

→ **Página** 172

(A.5) **Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$**

Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 13th International Symposium of Practical Aspects of Declarative Languages (PADL 2011), volume 6539 of *Lecture Notes in Computer Science*, pages 145–159, Austin, Texas, USA, January 24–25, 2011. Springer-Verlag.

→ **Página** 181

(A.6) **A Declarative Embedding of XQuery in a Functional-Logic Language**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 21st International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2011), volume 7225 of *Lecture Notes in Computer Science*, pages 42–56, Odense, Denmark, July 18–20, 2011. Springer-Verlag.

→ **Página** 196

(A.7) **Declarative Debugging of Wrong and Missing Answers for SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012), volume 7294 of *Lecture Notes in Computer Science*, pages 73–87, Kobe, Japan, May 23–25, 2012. Springer-Verlag.

→ **Página** 211

(A.8) **XQuery in the Functional-Logic Language Toy**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011), volume 6816 of *Lecture Notes in Computer Science*, pages 35–51, Odense, Denmark, July 19, 2011. Springer-Verlag.

→ **Página** 226

(A.9) **XPath Query Processing in a Functional-Logic Language**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

In XI Jornadas sobre Programación y Lenguajes, a Coruña, Spain, September 5–7 , 2011. Editors: Puri Arenas, Victor Gulías and Pablo Nogueira. Volume 282 of Electronic Notes in Theoretical Computer Science, pages 31–45, 2012.

→ **Página** 243

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 216 (2008) 79–92
www.elsevier.com/locate/entcs

A New Proposal for Debugging Datalog Programs[1]

R. Caballero[a,2] Y. García-Ruiz[a,3] F. Sáenz-Pérez[b,4]

[a] *Departamento de Sistemas Informáticos y Computación*
Universidad Complutense de Madrid
Madrid, Spain

[b] *Departamento de Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid
Madrid, Spain

Abstract

In this paper, we propose to apply declarative debugging to Datalog programs. Our approach relies on program semantics rather than on the computation mechanism. The debugging process starts when the user detects an unexpected answer. By asking questions about the intended semantics, the debugger looks for incorrect program relations. While usual declarative debuggers for other languages are based on computation trees, we show that graphs are more convenient structures for representing Datalog computations. The theoretical framework is complemented by the implementation of a debugger for the deductive database system DES, a publicly available open-source project.

Keywords: Declarative Debugging, Datalog Programs.

1 Introduction

The declarative programming paradigm is targeted to raise the semantic level of programs, therefore isolating them from the computation model. Thus, programmers are intended to focus on a higher semantic level rather than on the level corresponding to the underlying computation procedures.

Deductive database languages such as Datalog [9], which inherit the declarative nature of the Logic Programming language Prolog [15], increase the gap between the program semantics and the computations because the computation model of Datalog is much more intricate than that of Prolog. The Prolog computation model is

[1] This work has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.
[2] Email: rafa@sip.ucm.es
[3] Email: ygarciar@fdi.ucm.es
[4] Email: fernan@sip.ucm.es

doi:10.1016/j.entcs.2008.06.035

based on the SLD resolution principle [7], which deals with SLD computation trees, whereas Datalog computation model is based on a number of proposals, ranging from interpreters [17] to compilation to Prolog using magic sets [3]. The first proposal needs some sort of fixpoint computations to solve queries, which is not the computation model the user might have in mind. The second proposal makes things even worse, in the sense that the program is transformed before applying SLD resolution. This semantic gap between program semantics and program execution makes debugging Datalog programs a hard task if one tries to use existing tools for debugging in a quite different level the user thinks about (for instance, using a trace debugger in the level of the transformed program).

Our approach to debug Datalog programs is anchored to the semantic level, which is a natural requirement every user imposes to development systems. We propose a novel way of applying *declarative debugging*, also called Algorithmic Debugging (a term first coined in the logic programming field by E.H. Shapiro [12]) to Datalog programs, allowing to debug queries and diagnose *missing* (an expected tuple is not computed) as well as *wrong* (a given computed tuple should not be computed) answers with the same tool.

What a Datalog programmer would find useful is to catch program rules or relations which are responsible for a mismatch between the intended semantics of a query and its actual computed semantics. Our system, by means of a question-answering procedure which starts when the user detects an unexpected answer for some query, looks for those errors pointing to the program fragment responsible for the incorrectness. For this procedure, we propose to use *computation graphs* as a novel data structure for declarative debugging of Datalog programs. We find that these graphs are more convenient for modeling program computations, instead of computation trees, which have been typically used in declarative debuggers for other languages (e.g., Prolog [12], Java [5] and $\mathcal{T}oy$ [4]).

With this aim we have implemented a working prototype for DES [11], a Datalog system publicly available as an open-source project which was released in 2004, and it has been mainly used in several universities for teaching deductive databases since then. The current version with debugging capabilities can be downloaded and tested on almost any platform.

The few existing proposals for debugging Datalog programs are usually based on "imperative" debugging, that try to follow the computation model to find bugs. These proposals are mainly based on forests of proof trees [2,18,14], which makes debugging a trace-based task not so amenable to users. To our knowledge, the very first work on this setting is due to [10], but a variant of SLD resolution is used by the user to look for program errors, therefore imposing to traverse at least as many trees as particular answers are obtained for any query. In a database framework, where the answer can contain many individual values, this makes the task of debugging quite cumbersome. In our setting, we deal with the set of values for a query as a single entity, therefore reducing the complexity of the debugging task.

This paper is organized as follows. Section 2 introduces some definitions. Our proposal of computation graph is defined in 3. Section 4 discusses how the com-

putation graph can be used for debugging. Section 5 presents the debugger tool integrated in the system DES. Finally, Section 6 summarizes conclusions from this work and points out some future work.

2 Datalog Preliminaries

Definitions for Datalog mainly come from the field of Logic Programming. In this section, we only introduce the concepts which are needed in our setting, referring the reader to [7] for a more general presentation of Logic Programming.

We consider (recursive) Datalog programs with stratified negation [1,17], i.e., normal logic programs without function symbols. Stratification is imposed to ensure a clear semantics when negation is involved, and function symbols are not allowed in order to guarantee termination of computations, a natural requirement with respect to a database user.

A *term* is either a variable or a constant symbol. An *atom* is $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and t_i are terms, $1 \leq i \leq n$, which can also be written as $p(\bar{t}_n)$. A *literal* is either an atom or a negated atom. A *positive* literal is an atom, and a *negative* literal is a negated atom. A negated atom is syntactically constructed as $not(A)$, where A is an atom. The atom contained in a literal L will be denoted as *atom(L)*. A *rule* R is an expression of the form $A : -L_1, \ldots, L_n$, where A is an atom and L_i are literals. All of the variables in a rule are assumed to be universally quantified. Concerning the rule R, A is referred to as the *head* of R, $L_1, \ldots, L_n$ as its *body*, and L_i as *subgoals*. Commas in bodies stand for conjunctions. A *fact* is a rule with empty body and *ground* head. The symbol $: -$ is usually dropped in this case. A *Datalog program* is a finite set of Datalog facts and rules. In order to fit with database notation, the term *relation* is used *in lieu* of *predicate*. A database *relation* is, therefore, a set of rules with the same predicate symbol and arity. If a relation is only defined with facts, it is called an *extensional-database* relation (EDB), whereas it is otherwise called an *intensional-database* relation (IDB). A *query* (term preferred in a deductive database context) or *goal* (term preferred in a logic programming context) is a literal (i.e., an atom or a negated atom) which can be solved by a Datalog system with respect to a given program. Analogously to literals, we say that a *positive* query is an atom, and a *negative* query is a negated atom. In contrast to facts, queries may contain variables.

Substitutions are defined as usual in logic programming. *Subst* denotes the set of all the substitutions. We also assume the existence of a composition operation between substitutions defined in the usual way and fulfilling the property $(s\theta)\sigma = s(\sigma \cdot \theta)$ for all $\sigma, \theta \in Subst$. Two formulae φ, φ' are *variants* if $\varphi = \varphi'\theta$ with θ a renaming. We use the notation *fresh(φ)* to represent a renaming of the formula φ which replaces all its variables by new variables.

Datalog programs resemble Prolog programs as the program (adapted from [19]) in Figure 1 suggests, which will be used as a running example for the rest of the paper. In Datalog programs, variables start with uppercase letters whereas constants start with lowercase letters (e.g., `X` and `nil`, respectively, in the example).

```
% Pairs of non-consecutive elements in br
between(X,Z) :-       br(X), br(Y), br(Z), X<Y, Y<Z.

% Consecutive elements in the sequence, starting at nil
next(X,Y) :-          br(X), br(Y), X<Y, not(between(X,Y)).
next(nil,X) :-        br(X), not(has_preceding(X)).

% Values having preceding values in the sequence
has_preceding(X) :-   br(X), br(Y), Y>X. % error: it should be Y<X

% Values in an even position of the sequence, including nil
even(nil).
even(Y) :-            odd(X), next(X,Y).

% Values in an odd position of the sequence
odd(Y) :-             even(X), next(X,Y).

% Succeeds if the cardinality of the sequence is even
br_is_even :-         even(X), not(next(X,Y)).

% Succeeds if the cardinality of the sequence is odd
br_is_odd :-          odd(X), not(next(X,Y)).

% Sequence
br(a).
br(b).
```

Fig. 1. Example of Datalog program

Predicate symbols start with lowercase letters (e.g., `between`). Code remarks start with `%` and apply up to the end of line.

The example program is intended to compute the parity of a given base relation `br(X)`, i.e., it can determine whether the number of elements in the relation (cardinality) is even or odd by means of the relations `br_is_even`, and `br_is_odd`, respectively. The relation `next` defines an ascending chain of elements in `br` based on their textual ordering, where the first link of the chain connects the distinguished node `nil` to the first element in `br`. The relations `even` and `odd` define the even, resp. odd, elements in the chain. Finally the relation `has_preceding` defines the elements in `br` such that there are previous elements to a given one (the first element in the chain has no preceding elements). The rule defining this relation includes an intended error (fourth rule in the example) which will be used in forthcoming sections to show how it is catched by the declarative debugger. The symbol $<$ denotes a built-in relation checking if some element is less than another w.r.t. the predefined term ordering. Observe that relations `br_is_even`, and `br_is_odd` are not *range restricted* because variable Y occurs only in a negative literal, and that therefore the program does not fulfill the usual *safety conditions* [17]. In fact, our setting does not enforces the use of safe programs. Only the stratification requirement is needed for the correctness of the debugging technique.

The semantics (i.e., the "meaning") of a Datalog program can be given by either the model-theoretic, proof-theoretic or fixpoint semantics. We focus on the

model-theoretic semantics. In particular, we consider *Herbrand interpretations* and *Herbrand models*, i.e., Herbrand interpretations that make every Herbrand instance of the program rules logically true formulae.

Given a Herbrand model $\mathcal{M}$, we define the meaning of a query Q in the context of a program P as the set:

$$Q_{\mathcal{M}} = \{Q\theta \in \mathcal{M}\}$$

with $\theta \in Subst$. We call $Q_{\mathcal{M}}$ the *answer* for Q w.r.t. P.

In Logic Programming languages such as Prolog, the least Herbrand model cannot be actually computed, because programs in general are non-terminating. Also the use of *negation as failure* contributes to the lack of completeness of Prolog computations. However, due to the use of stratified programs in Datalog, the existence of a so-called *supported model* $\mathcal{M}$ is ensured [1]. $\mathcal{M}$ is a minimal Herbrand model of program P that can be actually computed by a Datalog system. Thus, we assume that our Datalog system implementation yields the value $Q_{\mathcal{M}}$ for any query Q.

Additionally, we use the term *intended interpretation* represented by $\mathcal{I}$, to denote the model the user has in mind for the program. The *intended* answer for a query Q is accordingly defined as:

$$Q_{\mathcal{I}} = \{Q\theta \in \mathcal{I}\}$$

Then, the user can focus on queries and compare the intended interpretation to the minimal Herbrand model actually computed. We therefore speak of *validity* of a computed query w.r.t. the intended model of a program when:

$$Q_{\mathcal{M}} = Q_{\mathcal{I}}$$

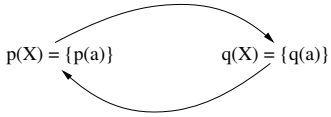
If given a program P we find some query Q s.t. $Q_{\mathcal{M}} \neq Q_{\mathcal{I}}$, we have that P is an *incorrect program*, which must include one or more *incorrectly defined relations*.

3 Computation Graphs

In this section, we define a suitable structure for representing Datalog computations. Usually in logic programming languages such as Prolog, the computations are represented through some tree structure such as the SLD-tree [7]. In the case of Datalog, we claim that a tree is not a convenient structure due to the different treatment of recursive programs. For instance consider the program:

```
p(a).
r(b).
p(X) :- q(X),r(X).
q(X) :- p(X).
```

In Prolog, the SLD-tree for the goal `p(X)` will contain an infinite branch, representing a non-terminating computation. However in Datalog the same goal is terminating and returns the finite answer: {`p(a)`} because the computation mechanism detects the repetition of the subgoal `p(x)` and avoids the infinite loop. Thus, our computation structure must represent finitely these situations, which can be achieved by using a graph:



The graph contains the two subgoals occurred during the computation together with their respective answers. It also indicates that `p(X)` and `q(X)` are mutually dependent. We will call such graph the *computation graph* for the goal w.r.t. the program. Observe that this graph is different from the *predicate dependency graph* [19] of the Datalog program, which show the connections between the relations from a static point of view. In the example, the dependence graph will include a vertex for the relation `r` connected to the vertex for `p`. However, our computation graph depends on the initial goal and the subgoals that occur during the computation and hence does not include any vertex for `r`.

The computation graph (CG in short) is a directed graph which only has one connected component, and that can contain cycles in the case of queries involving recursive predicates.

Each vertex of the CG contains all the information necessary for detecting its validity w.r.t. the intended model of the program. Hence the information stored at each vertex of the CG is the following:

- The query Q.
- The answer for the query.

Now we describe formally how the debugger builds a CG.

Definition 3.1 (**Computation Graph**) Given a Datalog program P and a query Q the computation graph of Q w.r.t. P is defined as the value *cg(P,Q)* defined as follows:

cg(P,Q)

Input:

- P: a stratified Datalog program.
- Q: a program query. It must be either of the form $p(\bar{a}_n)$ or $not(p(\bar{a}_n))$, with $p(\bar{a}_n)$ an atom and p a relation defined in P.

Output: a directed graph (V, E), where V is a set of vertices of the form $(Q = Q_{\mathcal{M}})$ and $E = \{(u, v) | u, v \in V\}$ is a set of edges.

Steps:

1 The first vertex v of the graph is associated to the initial query Q and is defined as $v := (p(\bar{a}_n) = \Pi)$, with $\Pi := p(\bar{a}_n)_{\mathcal{M}}$. The set Π can be obtained directly using the system to obtain the answer for the query $p(\bar{a}_n)$.

2 $V := \{v\}$, $E := \emptyset$.

3 Let A be an auxiliary set containing the vertices that must be unfolded in order to build the graph. Initially $A := \{v\}$.

4 While $A \neq \emptyset$ do:
   i) Select any vertex u in A. $A := A \setminus \{u\}$. The vertex u must be of the form $(q(\bar{b}_k) = \Pi_q)$ for some q, Π_q, $\bar{b}_k$.
   ii) Consider all the rules (disregarding facts) defining q: $R_{q_1}, \ldots, R_{q_s}$. For each R_{q_i}, $1 \leq i \leq s$:
      a) N := *newVertices*($q(\bar{b}_k)$,*fresh*(R_{q_i}))
      b) For each new vertex $(Q = Q_{\mathcal{M}}) \in N$ check whether exists already a vertex $(Q' = Q'_{\mathcal{M}}) \in V$ such that Q and Q' are variants. There are two possibilities:
         - There exists such $(Q' = Q'_{\mathcal{M}})$. Then, $E := E \cup \{(u, (Q' = Q'_{\mathcal{M}}))\}$. That is, if the vertex already exists we simply add a new edge from u.
         - Otherwise, $V := V \cup \{(Q = Q_{\mathcal{M}})\}$, $A := A \cup \{(Q = Q_{\mathcal{M}})\}$, and $E := E \cup \{(u, (Q = Q_{\mathcal{M}}))\}$.

newVertices(A,R)

Input:

- A: An atom of the form $q(\bar{b}_k)$.
- R: A program rule of the form $q_i(\bar{t}_n) :- L_1, \ldots, L_m$.

Output: a set S of vertices.

Steps:

1 If $\theta := m.g.u.(q(\bar{b}_k), q_i(\bar{t}_n))$ does not exist return $\emptyset$.

2 Otherwise, for each literal L_j in the right-hand side of rule R consider the next two possibilities:
   i) $j = 1$. Then $v_1 := (atom(L_1)\theta = \Pi_1)$, where $\Pi_1 := (atom(L_1)\theta)_{\mathcal{M}}$. $S := \{v_1\}$. v_1 is the vertex associated with the first literal L_1 .
   ii) If $j > 1$, let $\delta_1, \ldots, \delta_r$ be all the substitutions such that:

$$L_h\theta\delta_1 \in \mathcal{M}, \ldots, L_h\theta\delta_r \in \mathcal{M} \quad , \; h = 1, \ldots, j-1$$

This means that the first $j-1$ literals have succeeded for each substitution $\theta\delta_1, \ldots, \theta\delta_r$. Then the following new r vertices associated with the literal L_j are defined

$$v_{j_1} := (\ atom(L_j)\theta\delta_1 = \Pi_{j_1}) \qquad \ldots \qquad v_{j_r} := (\ atom(L_j)\theta\delta_r = \Pi_{j_r})$$

with $\Pi_{j_1} := (atom(L_j)\theta\delta_1)_{\mathcal{M}}, \ \ldots, \ \Pi_{j_r} := (atom(L_j)\theta\delta_r)_{\mathcal{M}}$.

Finally set $S := S \cup \{v_{j_1}, \ldots, v_{j_r}\}$, stating that we have created r new nodes.

End of Definition

Observe that if a relation p is only defined by facts (i.e., $s = 0$ at step 4.*ii*) of the *cg* algorithm), the CG only contains one vertex, and has no edges. A CG of the query `br_is_even` w.r.t the program in Figure 1 is presented in Figure 2 (ignore the bounded and colored vertices at the moment). For instance the vertex `even(X) = {`

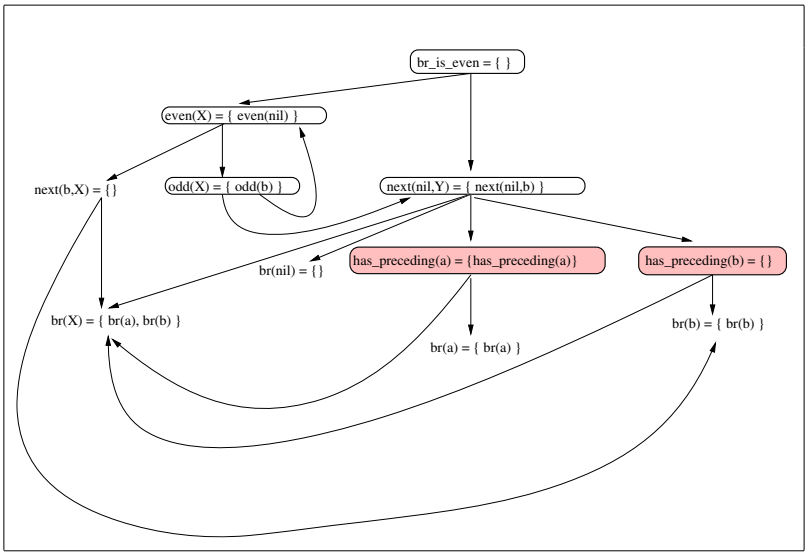


Fig. 2. Example of Computation Graph

`even(nil) }` includes the answer `even(nil)` for the query `even(X)`. The relation `even` is defined through one fact and one rule. The fact does not appear in the figure (they do not create new subgoals), and the rule yields two new vertices, one for each literal in its right-hand side. The first one, namely `odd(X)` , includes an answer with one result. This result has an associated substitution $\delta_1 = \{X \mapsto b\}$. The second vertex corresponds to `next(X,Y)`δ_1, which produces an empty answer. Notice that there is no edge from vertex `next(b,X) = { }` to any vertex corresponding to the relation `between`, although `between` occurs in the rhs of the first rule defining `next`. The reason is that there is not exist a substitution δ such that

$$br(b)\delta \in \mathcal{M} \ \wedge \ br(Y)\delta \in \mathcal{M} \ \wedge \ (b < Y)\delta \in \mathcal{M}$$

i.e. the first three literals for this clause do not succeed for any substitution δ. Also observe that there is no vertex corresponding to the symbol $<$ in the graph, because it is a primitive built-in relation and it is hence assumed correct by the debugger.

4 Declarative Debugging with CG's

In this section, we show how the computation graph is used by the debugger in order to detect incorrectly defined relations.

The debugging process starts when the user finds some unexpected answer for a query, i.e., some *initial symptom*. For instance, in the case of the program in Figure 1, the user expects that the answer for the query `br_is_even` should be

{`br_is_even`}, because the relation `br` contains two elements: `a` and `b`. However, the answer returned by the system is { }, which means that the corresponding goal was unsuccessful. Therefore, the user will start the debugger. The debugger proceeds by following the stages:

(i) First the system ensures that the computation for the goal is up to date. Then, it generates a suitable *computation graph* that represents the computation. In the case of our running example and the query `br_is_even`, the CG is displayed in Figure 2. This first phase is automatically performed by the tool.

(ii) Second, the CG obtained in the previous phase is traversed asking to the user about the validity of some vertices looking for a *buggy* vertex. A vertex is called buggy when it is non-valid but all its immediate descendants are valid. A buggy vertex always corresponds to an incorrectly defined relation, which is pointed out by the debugger as the cause of the error.

Therefore, the debugger will ask the user questions about the validity of certain vertices of the CG w.r.t. to the intended interpretation $\mathcal{I}$ of the program P. For instance, the intended interpretation of the program of the Figure 2 is:

$$\mathcal{I} = \{ \quad \textit{br\_is\_even, even(nil), even(b), odd(a), next(nil,a), next(a,b),}$$
$$\textit{has\_preceding(b), br(a), br(b)} \ \}$$

The debugger assumes that the user knows whether an instance of a query is in $\mathcal{I}$, i.e., that the user can determine the answer for any query. For instance, a possible question could be *Is odd(X)={odd(b)} valid?* The question must be understood as *Is {odd(b)} the expected answer for the query odd(X)?*. The answer will be `no` because the expected answer for a query *odd(X)* is {*odd(a)*}, because *a* is the only element of the sequence *a,b* that is in an odd position.

We can distinguish two reasons for detecting that a computed answer $Q_{\mathcal{M}}$ for a query Q is incorrect:

(i) There exists $\sigma \in \mathit{Subst}$ s.t. $Q\sigma \in \mathcal{M}$ but $Q\sigma \notin \mathcal{I}$. Then $Q_{\mathcal{M}}$ is called a *wrong answer*. For instance, the answer {*has_preceding(a)*} for the query *has_preceding(a)* in the CG of Figure 2 contains a wrong answer, because *a* has no preceding value in the intended interpretation.

(ii) There exists $\sigma \in \mathit{Subst}$ s.t. $Q\sigma \in \mathcal{I}$ but $Q\sigma \notin \mathcal{M}$. Then $Q_{\mathcal{M}}$ is called a *missing answer*. For instance, the vertex *even(X)={even(nil)}* in the Figure 2 contains a missing answer because for $\sigma = \{X \mapsto b\}$ we have $\mathit{even}(X)\sigma = \mathit{even}(b)$, and $\mathit{even}(b) \in \mathcal{I}$ but $\mathit{even}(b) \notin \mathcal{M}$.

Observe that the two errors can exist at the same time: $\mathit{odd}(X) = \{\mathit{odd}(b)\}$ is both a wrong answer ($\mathit{odd}(b)$ should not be in the answer) and a missing answer ($\mathit{odd}(a)$ should be in the answer). Declarative debuggers usually require the user to distinguish both types of errors in order to initiate the debugging process. These types of errors can require even different types of different computation structures. An advantage of our approach is that this distinction is not needed, and that the same structure, the CG is valid for both types of errors.

In fact, our approach does not match the general scheme proposed by L. Naish in [8] for declarative debugging, because it is based on a graph rather than on a tree. However, some of the basic results are still valid. For instance the *correctness* of the technique:

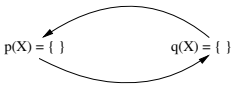
Theorem 4.1 *Any buggy vertex in a CG corresponds to an incorrectly defined relation.*

The informal reasoning behind this result is easy: since a buggy vertex is an incorrect vertex, this means that it contains an incorrect answer for its associated query. Also, in the graph construction algorithm, it can be checked that the immediate descendants of the vertex are the subqueries whose result were needed to produce such incorrect answers. But, if the subqueries returned correct values, the error must come from the relation itself, which is therefore incorrectly defined.

In the CG of the Figure 2 the incorrect vertices are surrounded by a box, while the two buggy vertices are contained in a colored box. Both buggy vertices correspond to the relation `has_preceding`, which corresponds to the only incorrectly defined relation of the program of Figure 1.

Another nice property of the general scheme based on computation trees is completeness: every computation tree with an incorrect node contains a buggy node. Unfortunately, this result does not hold in our setting. Consider for instance the program:

```
p(X) :- q(X).
q(X) :- p(X).
```



The CG, displayed at the right of the program will contain two vertices, both displaying the empty answer. Imagine also that either p or q is an incorrectly defined relation, because the user forgot to include in the program either the fact *p(a)* or *q(a)*. In either of theses cases $\mathcal{I} = \{p(a), q(a)\}$. Then, we will have a CG with two incorrect vertices and with no buggy vertex. In this case, our debugger will not point out any relation, but a *set of connected relations* as the cause of the error. Fortunately, this situation that leads to less informative diagnosis, is not usual in common Datalog programs.

5 Implementation

The Datalog Educational System (DES) is an open-source free Prolog-based implementation of a basic deductive database with stratified negation with Datalog as a query language. The system is implemented on top of Prolog and can be used from several Prolog interpreters (Ciao Prolog, GNU Prolog, SICStus Prolog, and SWI Prolog) running on several operating systems (OSs). Moreover, executables for several OSs (Windows 98 and later, and SunOS/Solaris) are also provided. It was aimed to have a simple, interactive, multiplatform, and affordable system for students and researchers. DES 1.3.0 is the current release, which enjoys full recursive evaluation with memoization techniques and stratified negation. DES is

implemented with the seminar ideas found in [6,16], that deal with termination issues of Prolog programs. These ideas have been already used in the deductive database community. Our implementation uses the concept of *extension table* for achieving a top-down driven bottom-up approach. In its current form, it can be seen as an extension of the work in [6] in the sense that, in addition, we deal with negation and undefined (although incomplete) information (cfr. [11] for further details about undefinedness). Once a query is computed, the extension table holds the computed meaning of the program restricted to the query, i.e., only the meanings of needed relations for computing the meaning of the query are computed.

We have implemented a debugger tool in the DES system based on the ideas presented in previous sections. Next, we describe some of its features.

As we have seen in Section 4, the debugging process consists of two phases. During the first phase, the tool builds a CG for the initial query Q w.r.t. the program P. This phase, in turn, can be divided into two parts:

(i) The debugger uses the system DES in order to produce the *extension table* for Q w.r.t. P.

(ii) The CG is built following the description given in Section 3. The answers Π at each vertex are obtained from the extension table. This corresponds to assuming that the system computes the supported Herbrand model $\mathcal{M}$. As we explained in Section 1, this is possible due to the requirements of stratification imposed to our Datalog programs.

The second phase consists of traversing the CG in order to find either a buggy vertex or a set of related incorrect vertices. The vertex associated to the initial query Q is marked automatically as *non-valid* by the debugger. The rest of the vertices are marked initially as *unknown*. In order to minimize the number of questions asked by a declarative debugger, several traversing strategies have been studied [4,13]. However, these strategies are only valid for declarative debuggers based on trees and not on graphs and new strategies are still to be investigated for this structure. Nevertheless, the currently implemented strategy already contains some ideas of how to minimize the number of questions in a CG:

- It firstly asks about the validity of vertices that are not part of a cycle, in order to find a buggy vertex if it exists. Only when this is no longer possible the vertices that are part of cycles are visited.
- Each time the user indicates that a vertex ($Q = \Pi$) is valid, i.e., the validity of the answer Π for the subquery Q is ensured, the tool changes to *valid* all the vertices with associated queries of the form $Q\theta$, with $\theta \in Subst$.
- Each time the user indicates that a vertex ($Q = \Pi$) is non-valid, the tool changes to *non-valid* all the vertices with associated queries Q', with $Q = Q'\theta$, $\theta \in Subst$.

The two last items help to reduce the number of questions deducing automatically the validity/non-validity of some vertices from the validity/non-validity of others. For instance, in Figure 2, the validity of the vertices containing *br(a)*={*br(a)*} and *br(nil)*={} can be deduced automatically from the validity of the vertex *br(X)*=

{*br(b),br(a)* }. The soundness of these deductions is established by the following proposition:

Proposition 5.1 *Let Q be a query, let $Q_{\mathcal{I}}$ be the answer of Q w.r.t. the intended interpretation $\mathcal{I}$, and let θ be a ground substitution, $\theta \in Subst$. Then:*

(i) *If $Q = Q_{\mathcal{I}}$ is valid, then $Q\theta = Q\theta_{\mathcal{I}}$ is valid.*

(ii) *If $Q = Q'\theta$ for some Q' and $Q = Q_{\mathcal{I}}$ is non-valid, then $Q' = Q'_{\mathcal{I}}$ is non-valid.*

Proof.

(i) If $Q = Q_{\mathcal{I}}$ is valid, then for all $\sigma \in Subst$, $Q\sigma \in \mathcal{I} \Longleftrightarrow Q\sigma \in \mathcal{M}$. Thus, for all σ', $(Q\theta)\sigma' \in \mathcal{I} \Longleftrightarrow (Q\theta)\sigma' \in \mathcal{M}$ simply considering $\sigma = (\sigma' \cdot \theta)$.

(ii) If $Q = Q'\theta$ for some Q' and $Q = Q_{\mathcal{I}}$ is non-valid, then there exists $\sigma \in Subst$ s.t. one of the two possibilities hold:
   (a) $Q\sigma \in \mathcal{M}$ but $Q\sigma \notin \mathcal{I}$.
   (b) $Q\sigma \in \mathcal{I}$ but $Q\sigma \notin \mathcal{M}$.
   Hence, $Q_{\mathcal{M}}$ is either a wrong or a missing answer. By defining $\sigma' = (\sigma \cdot \theta)$ we have that $Q'\sigma'$ is also a wrong or missing answer.

□

A debugger session for the query `br_is_even` of our running example:

```
DES> /debug br_is_even
Debugger started ...
Is br(b) = {br(b)}  valid(v)/non-valid(n) [v]? v
Is has_preceding(b) = {}  valid(v)/non-valid(n) [v]? n
Is br(X) = {br(b),br(a)}  valid(v)/non-valid(n) [v]? v

! Error in relation:  has_preceding/1
! Witness query: has_preceding(b) = { }
```

In this particular case, only three questions are necessary to find out that the relation *has_preceding* is incorrectly defined.

6 Conclusions and Future Work

In the previous sections we have presented a framework for the declarative debugging of Datalog programs. The proposed technique finds incorrect relation definitions in Datalog programs by comparing the results of the computations to the intended interpretation of each relation, which is assumed to be known by the user. Thus, our technique relies on the program semantics for debugging, disregarding the implementation issues. In Datalog this is not only an advantage; it is almost a necessity. The Datalog computations are based on operational features or program transformations that make the execution very difficult to follow and understand using the normal trace facilities included in logic languages such as Prolog.

We have also defined a suitable structure for representing the computations, the computation graphs. This represents a novelty w.r.t. the traditional presentation of

declarative debugging, which is based on trees rather than on graphs. Nevertheless, declarative debugging using computation graphs lacks some of the nice properties of usual declarative debugging such as completeness. We have shown that indeed in some Datalog programs is not possible to point out a single relation as the cause of an unexpected computation result, and that in those programs the debugger can only detect sets of mutually dependent relations as possible error sources, meaning that one or more of these relations are incorrectly defined. However, these situations are not common in practice.

It is important to emphasize that the debugger can be used for diagnosing errors starting either from a wrong or from a missing answer. Since Datalog programs are terminating, we can claim that the presented technique covers *all* the possible errors that produce unexpected answers in Datalog programs. This makes a difference w.r.t. other declarative debuggers that are limited to a particular kind of errors (i.e., only missing or only wrong answers). The ideas have been implemented in a working prototype included as part of the Datalog system DES.

As future work we plan to represent graphically the computation graph. This will help the user to find the error more easily, inspecting the graph and choosing freely the more convenient vertices to start the debugging process. Another improvement can be obtained by allowing the user to provide more informative answers. For instance, if the debugger knows that an answer is not only non-valid but *wrong*, i.e., it contains an unexpected atom, it can use this information to skip some questions; in particular, the questions involving children with empty answers, which are always valid w.r.t. wrong answers.

References

[1] Apt, K. R., H. A. Blair and A. Walker, *Towards a theory of declarative knowledge* (1988), pp. 89–148.

[2] Arora, T., R. Ramakrishnan, W. G. Roth, P. Seshadri and D. Srivastava, *Explaining program execution in deductive systems*, in: *Deductive and Object-Oriented Databases*, 1993, pp. 101–119.

[3] Beeri, C. and R. Ramakrishnan, *On the power of magic*, 1987, pp. 269–284.

[4] Caballero, R., *A declarative debugger of incorrect answers for constraint functional-logic programs*, in: *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming* (2005), pp. 8–13.

[5] Caballero, R., C. Hermanns and H. Kuchen, *Algorithmic Debugging of Java Programs*, Electronics Notes in Theorical Computer Science (2007), in Press.

[6] Dietrich, S. W., *Extension tables: Memo relations in logic programming.*, in: *SLP*, 1987, pp. 264–272.

[7] Lloyd, J., "Foundations of Logic Programming," Springer Verlag, 1984.

[8] Naish, L., *A Declarative Debugging Scheme*, Journal of Functional and Logic Programming **3** (1997).

[9] Ramakrishnan, R. and J. Ullman, *A survey of research on Deductive Databases*, The Journal of Logic Programming **23** (1993), pp. 125–149.

[10] Russo, F. and M. Sancassani, *A declarative debugging environment for Datalog*, in: *Proceedings of the First Russian Conference on Logic Programming* (1992), pp. 433–441.

[11] Sáenz-Pérez, F., *Datalog Educational System. User's Manual*, Technical Report 139-04, Facultad de Informática, Universidad Complutense de Madrid (2004), `http://des.sourceforge.net/`.

[12] Shapiro, E., "Algorithmic Program Debugging," ACM Distiguished Dissertation, MIT Press, 1982.

[13] Silva, J., *A Comparative Study of Algorithmic Debugging Strategies*, in: *Proc. of International Symposium on Logic-based Program Synthesis and Transformation LOPSTR 2006*, 2007, pp. 134–140.

[14] Specht, G., *Generating explanation trees even for negations in deductive database systems*, in: *Proceedings of the 5th Workshop on Logic Programming Environments*, Vancouver, Canada, 1993.

[15] Sterling, L. and E. Shapiro, "The art of Prolog: advanced programming techniques," MIT Press, Cambridge, MA, USA, 1986.

[16] Tamaki, H. and T. Sato, *Old resolution with tabulation*, in: *Proceedings on Third international conference on logic programming* (1986), pp. 84–98.

[17] Ullman, J., "Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies)," Computer Science Press, 1995.

[18] Wieland, C., *Two explanation facilities for the deductive database management system DeDEx.*, in: H. Kangassalo, editor, *ER* (1990), pp. 189–203.

[19] Zaniolo, C., S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian and R. Zicari, "Advanced Database Systems," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

A Theoretical Framework for the Declarative Debugging of Datalog Programs

R. Caballero[1], Y. García-Ruiz[1], and F. Sáenz-Pérez[2,⋆]

[1] Departamento de Sistemas Informáticos y Computación,
[2] Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
rafa@sip.ucm.es, ygarciar@fdi.ucm.es, fernan@sip.ucm.es

Abstract. The logic programming language Datalog has been extensively researched as a query language for deductive databases. Although similar to Prolog, the Datalog operational mechanisms are more intricate, leading to computations quite hard to debug by traditional approaches. In this paper, we present a theoretical framework for debugging Datalog programs based on the ideas of declarative debugging. In our setting, a debugging session starts when the user detects an unexpected answer for some query, and ends with the debugger pointing to either an erroneous predicate or to a set of mutually recursive predicates as the cause of the unexpected answer. Instead of representing the computations by means of trees, as usual in declarative debugging, we propose graphs as a more convenient structure in the case of Datalog, proving formally the soundness and completeness of the debugging technique. We also present a debugging tool implemented in the publicly available deductive database system DES following this theoretical framework.

1 Introduction

Deductive databases rely on logic programming based query languages. Although not very well-known out of the academic institutions, some of their concepts are used in today relational databases to support advanced features of more recent SQL standards, and even implemented in major systems (e.g., the linear recursion provided in IBM's DB2 following the SQL-99 standard). A successful language for deductive databases has been Datalog [1], which allows users writing more expressive queries than relational databases. Relations and queries in Datalog are considered from a model-theoretic point of view, that is, thinking of relations as sets, and the language itself as a tool for manipulating sets and obtaining answer sets.

Raising the abstraction level generally implies a more complex computation mechanism acting as a black-box hidden from the user. Although this leads to more expressive programs, it also makes query debugging a very difficult

⋆ The authors have been partially supported by the Spanish National Project MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM(S-0505/TIC/0407).

K.-D. Schewe and B. Thalheim (Eds.): SDKB 2008, LNCS 4925, pp. 143–159, 2008.

process. An operational semantics oriented debugger is not helpful in this context, since the underlying computational mechanism is not directly related to the model-theoretic approach, but to implementation techniques such as *magic sets* [2] or *tabling* [3]. The few existing proposals for debugging specifically Datalog programs are usually based on "imperative" debugging, that try to follow the computation model to find bugs. These proposals are mainly based on forests of proof trees [4,5,6], which makes debugging a trace based task not so amenable to users. The first work related to the declarative debugging of Datalog programs is due to [7], but a variant of SLD resolution is used in order to look for program errors, imposing to traverse at least as many trees as particular answers are obtained for any query.

In the more general setting of *answer set programming* [8], there have been several proposals for diagnosing program errors in the last few years. In [9] a technique for detecting *conflict sets* is proposed. The paper explains how this approach can be used for detecting missing answers. Our proposal is limited to a more particular type of programs, namely stratified programs, but it can be applied for diagnosing not only missing but also *wrong* answers. In [10] the authors propose a technique that transforms programs into other programs with answer sets including debugging-relevant information about the original programs. This approach can be seen as a different, complementary view of the debugging technique described here.

In [11] we proposed a novel way of applying declarative debugging (also called *algorithmic debugging*, a term first coined in the logic programming field by E.H. Shapiro [12]), to Datalog programs. In that work, we introduced the notion of *computation graphs* (shortly *CG*s) as a suitable structure for representing and debugging Datalog computations. One of the virtues of declarative debugging is that it allows theoretical reasoning about the adequacy of the proposal. This paper addresses this task, proving formally the soundness and completeness of the debugging technique. We also present a prototype based in these ideas and included as part of a publicly available Datalog system DES [13].

The next section introduces the theoretical background needed for proving the properties of the debugger. Section 3 presents the concept of computation graph and proves several properties of *CG*s, while Section 4 includes the soundness and completeness results. Section 5 is devoted to discuss some implementation issues. Finally, Section 6 summarizes the work and presents the conclusions.

2 Datalog Programs

In this section, we introduce the syntax and semantics of Datalog programs and define the different types of errors that can occur in our setting. Although there are different proposals for this language, we will restrict our presentation to the language features included in the system DES [13]. Observe that the setting for Datalog presented here is a subsumed by the more general framework of *Answer Set Programming* [8].

2.1 Datalog Syntax

We consider (recursive) Datalog programs [14,15], i.e., normal logic programs without function symbols. In our setting, *terms* are either variables or constant symbols and *atoms* are of the form $p(t_1, \ldots, t_n)$, with p an n-ary predicate symbol and t_i terms for each $1 \leq i \leq n$. The notation $t_1, \ldots, t_n$ will be usually abbreviated as $\bar{t}_n$. A *positive* literal is an atom, and a *negative* literal is a negated atom. A negated atom is syntactically constructed as $not(A)$, where A is an atom. The atom contained in a literal L will be denoted as *atom(L)*. The set of variables of any formula F will be denoted as $var(F)$. A formula F is *ground* when $var(F) = \emptyset$.

A *rule* (or *clause* in the logic programming context) R has the form $p(\bar{t}_n)$: − $l_1, \ldots, l_m$ representing the first order logic formula $p(\bar{t}_n) \leftarrow l_1 \wedge \ldots \wedge l_m$, where l_i are literals for $i = 1 \ldots m$, and $m \geq 0$. The left-hand side atom $p(\bar{t}_n)$ will be referred to as the *head* of R, the right-hand side $l_1, \ldots, l_n$ as the *body* of R, and the literals l_i as *subqueries*. The variables occurring only in the body $l_1 \wedge \ldots \wedge l_m$ are assumed to be existentially quantified and the rest universally quantified. We require that $vars(H) \subseteq vars(B)$ for every program rule $H :- B$. A *fact* is a rule with empty body and ground head. The symbol :− is dropped in this case. The *definition of a relation* (or *predicate*) p in a program P consists of all the program rules with p in the head. A *query* (or *goal*) is a literal.

We consider stratified negation, a form of negation introduced in the context of deductive databases in [16]. A program P is called *stratified* if there is a partition $\{P_1, \ldots, P_n\}$ of P s.t. for $i = 1 \ldots n$:

1. If a relation symbol occurs in a positive literal of the body of any rule in P_i then its definition is contained in $\cup_{j \leq i} P_j$.
2. If a relation symbol occurs in a negative literal of the body of any rule in P_i then its definition is contained in $\cup_{j < i} P_j$.

We call each P_i a *stratum*. For instance, consider the Datalog program of Figure 1. We can check that the program is stratified by defining two strata: P_1 containing the rules for `star`, `orbits` and `intermediate`, and P_2 containing the rule for `planet`.

```
star(sun).
orbits(earth, sun).
orbits(moon, earth).
orbits(X,Y)       :- orbits(X,Z), orbits(Z,Y).
planet(X)         :- orbits(X,Y), star(Y), not(intermediate(X,Y)).
intermediate(X,Y) :- orbits(X,Y), orbits(Z,Y).
```

Fig. 1. A (buggy) Datalog Program

2.2 Program Models

We consider *Herbrand interpretations* and *Herbrand models*, i.e., Herbrand interpretations that make every Herbrand instance of the program rules logically true formulae. An *instance* of a formula is the result of applying the substitution θ to a formula F. We use the notation $F\theta$ instead of $\theta(F)$ for representing instances. The set $Subst$ represents the set of all the possible substitutions. Often, we will be interested in *ground instances of a rule*, assuming implicitly that every rule is renamed with new variables each time it is selected. The composition operation between substitutions is defined in the usual way and fulfilling the property $(F\sigma)\theta = F(\sigma \cdot \theta)$ for all $\sigma, \theta \in Subst$. Two formulae φ, φ' are *variants* if $\varphi = \varphi'\theta$, where θ is a renaming, i.e., a bijection among variables. We say that $\sigma \in Subst$ is an *instance* of $\theta \in Subst$ when $\sigma = \theta\mu$, with μ some substitution. In this case, we write $\sigma \geq \theta$.

Given a Herbrand interpretation I for a the Datalog program P, we use the notation $I \models F$ to indicate that the formula F is true in I. The *meaning of a query* Q w.r.t. the interpretation I, denoted by Q_I, is the set of ground instances $Q\theta$ s.t. $I \models Q\theta$. If Q is an atom, then an equivalent definition is $Q_I = \{Q\theta \mid Q\theta \in I \text{ for some } \theta \in Subst\}$.

In logic programming without negation, the existence of a *least Herbrand model* for every program P is ensured, and it can be obtained as the least fixed point of a closure operator T_P, which is defined over any interpretation I as:

$$A \in T_P(I) \text{ iff for some rule } (H :\!- B) \in P, I \models B\theta \text{ and } A = H\theta$$

In these conditions, the least Herbrand model is defined as $T_P \uparrow \omega(\emptyset)$, i.e., as the fixed point obtained when iterating the operator starting at the empty interpretation. In general, however, the existence of the least Herbrand model is not ensured in programs using negation. Fortunately, due to the use of stratified programs in Datalog, the existence of a so-called *standard model*, which we will represent also as $\mathcal{M}$, is in any case ensured [14]. Given a program P stratified by the partition $\{P_1, \ldots, P_k\}$, we define the sets $M_0 = \emptyset$, $M_1 = T_{P_1} \uparrow \omega(M_0)$, $\ldots$, $M_k = T_{P_k} \uparrow \omega(M_k - 1)$. Then, the standard model of P is defined as $\mathcal{M} = M_k$. The standard model verifies the following properties (the proofs can be found in [14]):

Proposition 1. *Let P be a program stratified by the partition $\{P_1, \ldots, P_k\}$. Then:*

1. *$\mathcal{M}$ is a minimal model.*
2. *$\mathcal{M}$ is supported, i.e., for all $p(\bar{s}_n) \in \mathcal{M}$ there exists an* associated program rule *$(H :\!- B) \in P$ and an* associated substitution *$\theta \in Subst$ such that $p(\bar{s}_n) = H\theta$, $\mathcal{M} \models B\theta$ and $B\theta$ ground (due to our safety condition, $var(H) \subseteq var(B)$, which means that $H\theta$ is also ground).*
3. *Conversely, if there is some $(H :\!- B) \in P$, $\theta \in Subst$ s.t. $\mathcal{M} \models B\theta$, then $\mathcal{M} \models H\theta$.*
4. *$\mathcal{M}$ is independent of the stratification.*

5. The following chain of inclusions holds:

$$\begin{array}{llll} \emptyset = M_0 & \subseteq T_{P_1}(M_0) & \subseteq T^2_{P_1}(M_0) & \subseteq \ldots \subseteq M_1 \\ M_1 & \subseteq T_{P_2}(M_1) & \subseteq T^2_{P_2}(M_2) & \subseteq \ldots \subseteq M_2 \\ & & \ldots & \\ M_{k-1} & \subseteq T_{P_k}(M_{k-1}) & \subseteq T^2_{P_k}(M_{k-1}) & \subseteq \ldots \subseteq M_k = \mathcal{M} \end{array}$$

Since functions are not allowed in Datalog, the standard model is finite and it can be actually computed. In fact, the deductive database systems such as DES are implemented to obtain the values $Q_{\mathcal{M}}$ for every query Q. Thus, $Q_{\mathcal{M}}$ will be referred to as the *answer* to Q. From now on, we assume that the Datalog system supporting the debugger verifies this condition, which is a reasonable requirement in the context of Datalog. This is different from the general setting of logic languages such as Prolog, even if we restrict to the case of Prolog programs without functions in the signature. For instance, consider the following dummy program:

```
p(X) :- q(X).      q(X) :- p(X).
```

The program is valid both in Prolog and in Datalog. However, the goal (resp. query) `p(X)` shows the difference between the two settings: In Prolog, it leads to a non-terminating computation, whereas in Datalog it succeeds with the answer $\{\}$, meaning that no ground instance of `p(X)` can be deduced from the program. Our selected system DES computes the answer to a query following a top-down approach, so that only the relevant information to obtain $Q_{\mathcal{M}}$ is computed in order to increase the efficiency of the computation.

The concept of standard model above is generalized by that of *stable model* [17], which can be applied also to non-stratified programs. However, in this work we restrict our semantics to stratified programs because this is a requirement of several Datalog systems.

2.3 Correct and Incorrect Programs

We use the term *intended interpretation*, denoted by $\mathcal{I}$, to denote the Herbrand model the user has in mind for the program. If $\mathcal{M} = \mathcal{I}$, we say that the program is *well-defined*, and if $\mathcal{M} \neq \mathcal{I}$ we say that the program is *buggy*. Declarative debugging assumes that the user focus on query answers for comparing the intended interpretation to the standard Herbrand model actually computed. Thus, we say that $Q_{\mathcal{M}}$ is an *unexpected answer* for a query Q if $Q_{\mathcal{M}} \neq Q_{\mathcal{I}}$. An unexpected answer can be either a *wrong answer*, when there is some $Q\theta \in Q_{\mathcal{M}}$ s.t. $Q\theta \notin Q_{\mathcal{I}}$, or a *missing answer*, when there is $Q\theta \in Q_{\mathcal{I}}$ s.t. $Q\theta \notin Q_{\mathcal{M}}$. In the first case, $Q\theta$ is a *wrong instance*, while in the second one $Q\theta$ is a *missing instance*. Observe that an unexpected answer can be both missing and wrong at the same time. The next proposition indicates that an unexpected answer to a positive query implies an unexpected answer to its negation.

Proposition 2. *Let P be a program containing at least one constant, $\mathcal{I}$ its intended model and Q a positive query. Then, $Q_{\mathcal{M}}$ is a missing answer for Q iff*

$(\neg Q)_{\mathcal{M}}$ is a wrong answer for $\neg Q$, and $Q_{\mathcal{M}}$ is a wrong answer for Q iff $(\neg Q)_{\mathcal{M}}$ is a missing answer for $\neg Q$.

Proof. Straightforward from the definition of meaning of a query w.r.t. an interpretation, since $Q_I \cap (\neg Q)_I = \emptyset$ in every interpretation I. Then, $p(\bar{t}_n) \notin Q_{\mathcal{M}}$ and $p(\bar{t}_n) \in Q_{\mathcal{I}}$, i.e., if $p(\bar{t}_n)$ is a missing instance and $Q_{\mathcal{M}}$ is a missing answer, iff $p(\bar{t}_n) \in (\neg Q)_{\mathcal{M}}$), $p(\bar{t}_n) \notin (\neg Q)_{\mathcal{I}}$, i.e., $p(\bar{t}_n)$ is a wrong instance and $(\neg Q)_{\mathcal{M}}$ is a wrong answer for $\neg Q$. Analogous for the other case. □

An unexpected answer indicates that the program is erroneous, and it will be considered as the initial symptom for a user to start the debugging process. The two usual causes of errors considered in the declarative debugging of logic programs are *wrong* and *incomplete* relations:

Definition 1 (Wrong Relation). *Let P be a Datalog program. We say that $p \in P$ is a **wrong relation** w.r.t. $\mathcal{I}$ if there exist a rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and a substitution θ such that $\mathcal{I} \models l_i\theta$, $i = 1 \ldots m$ and $\mathcal{I} \not\models p(\bar{t}_n)\theta$.*

Definition 2 (Incomplete Relation). *Let P be a Datalog program. We say that $p \in P$ is an **incomplete relation** w.r.t. $\mathcal{I}$ if there exists an atom $p(\bar{s}_n)\theta$ s.t. $\mathcal{I} \models p(\bar{s}_n)\theta$ and, for each rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ and substitution θ', either $p(\bar{t}_n)\theta' \neq p(\bar{s}_n)\theta$ or $\mathcal{I} \not\models l_i\theta'$ for some l_i, $1 \leq i \leq m$.*

In Datalog we also need to consider another possible cause of errors, namely the *incomplete set of relations*. This concept depends on the auxiliary definition of uncovered set of atoms.

Definition 3 (Uncovered Set of Atoms). *Let P be a Datalog program and $\mathcal{I}$ an intended interpretation for P. Let U be a set of atoms s.t. $\mathcal{I} \models p(\bar{s}_n)$ for each $p(\bar{s}_n) \in U$. We say that U is an **uncovered set of atoms** if for every rule $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and substitution θ s.t.:*

- *$p(\bar{t}_n)\theta \in U$,*
- *$\mathcal{I} \models l_i\theta$ for $i = 1 \ldots m$*

there is some $l_j\theta \in U$, $1 \leq j \leq m$, with l_j a positive literal.

Now, we are ready for defining the third kind of error, which generalizes the idea of incomplete relation:

Definition 4 (Incomplete Set of Relations). *Let P be a Datalog program and S a set of relations defined in P. We say that S is an **incomplete set of relations** in P iff exists an uncovered set of atoms U s.t. for each relation $p \in S$, $p(\bar{t}_n) \in U$ for some $t_1, \ldots, t_n$.*

To the best of our knowledge, this error has not been considered in the literature about Datalog debugging so far, but it is necessary for correctly diagnosing Datalog programs. Consider again the program `p(X):- q(X). q(X):-p(X).` with the intended interpretation $I = \{p(a), q(a)\}$ and the query `p(X)`. The computed

answer {} is a missing answer with `p(a)` as missing instance. However, neither of the two relations is incomplete, because their rules can produce the values `p(a)`, `q(a)` by means of the instance given by the substitution $\theta = \{X \mapsto a\}$. So, $U = \{p(a), q(a)\}$ is an uncovered set of atoms and hence $S = \{p, q\}$ is an incomplete set of relations.

We say that a relation is *buggy* when it is wrong, incomplete or member of an incomplete set of relations, and that it is well-defined otherwise. Observe that, due to the use of negation, a wrong answer does not correspond always to a wrong relation. For instance, in the following program:

```
p(X) :-  r(X), not(q(X)).
% missing q(a).
r(a).
```

with intended interpretation $\mathcal{I} = \{q(a), r(a)\}$ the query `p(X)` produces the wrong answer {`p(a)`} but there is no wrong relation in the program and instead there is an incomplete relation (q).

As an example, consider the program of Figure 1. This program defines a relation `orbits` by two facts and a rule establishing the transitive closure of the relation. A relation `star` is defined by one fact and indicates that the sun is a star. The relation `intermediate` is defined in terms of `orbits`, relating two bodies `X` and `Y` whenever there is some intermediate body between them. Finally, `planet` is defined as a body `X` that orbits directly a star `Y`, without any other body in between. However, a mistake has been introduced in the program: The underlined `Y` in the rule for `intermediate` should be `Z`. As a consequence, the query `planet(X)` yields the missing answer {} (assuming that the atom `planet(earth)` is in $\mathcal{I}$). In the next section, we will show how such errors can be detected by using declarative debugging based on computation graphs.

3 Computation Graphs

In this section, we define a structure for representing Datalog computations and prove their adequacy for declarative debugging.

3.1 Graph Terminology

We consider finite *directed graphs* $G = (V, E)$, where V is a finite set of vertices and E a finite set of directed edges, $E \subseteq V \times V$. Often, we use the notation $v \in G$ instead of $v \in V$ and $(u, v) \in G$ instead of $(u, v) \in E$. Given any vertex $u \in G$ we say that $v \in G$ is a *successor* of u in G if $(u, v) \in G$, which we represent by the notation $succ_G(u, v)$.

Given $G = (V, E)$, we say that $G' = (V', E')$ is a *subgraph* of G if G' is a graph s.t. $V' \subseteq V$ and $E' \subseteq E$. A particular case of subgraph is the *subgraph generated* from a subset of vertices $V' \subseteq V$. This subgraph is of the form $G' = (V', E')$, where $E' = \{(u, v) \in G \mid u, v \in V'\}$.

In a directed graph, the *output degree* of a vertex $v \in G$ is the cardinal of the set $\{u \in G \mid (v,u) \in G\}$ and it is represented by $gr^+_G(v)$. Analogously, the value $gr^-_G(v) = \mid \{u \in G \mid (u,v) \in G\} \mid$ represents the *input degree* of v. These concepts can be naturally extended to subgraphs by defining $gr^+_G(G') = \mid \{(u,v) \in G \mid u \in G', v \notin G'\} \mid$, $gr^-_G(G') = \mid \{(v,u) \in G \mid u \in G', v \notin G'\} \mid$. We remove the subindex G in gr^+_G, gr^-_G whenever the reference to the graph considered cannot be ambiguous in the context.

A sequence of vertices $u_1, u_2, \ldots, u_n$ of G such that $(u_i, u_{i+1}) \in G$ for all $i = 1 \ldots n-1$ are called a *walk* from u_1 to u_n. A walk s.t. $u_1 = u_n$ is called a *circuit*. A walk with no repeated vertices except maybe the first and the last vertex is called a *path*. If indeed $u_1 = u_n$ the path is called a *cycle*, i.e., a cycle is a special case of circuit with exactly one vertex repeated. The notation $path_G(u,v)$ represents a path starting at u and ending in v in some graph G.

A directed graph G is called *strongly connected* if, for every pair of vertices $u, v \in G$, there is a path from u to v and a path from v to u. The strongly connected components of a directed graph are its maximal strongly connected subgraphs, and they form a partition of G.

3.2 Datalog Computation Graphs

The *computation graph* (*CG* in short) for a query Q w.r.t. a program P is a directed graph $G = (V, E)$ such that each vertex V is of the form $(Q', Q'_{\mathcal{M}})$, where Q' is a subquery produced during the computation, and $Q'_{\mathcal{M}}$ is the computed answer for Q'. The next definition includes the construction of a computation graph. Observe that the answers of the subqueries are not relevant for the graph structure and, therefore, they are included as part of the vertices in a last step.

Definition 5 *(**Computation Graph**). Let P be a Datalog program and Q a query either of the form $p(\bar{a}_n)$ or $not(p(\bar{a}_n))$. The computation graph for Q w.r.t. P is represented by a pair (V, E) of vertices and edges defined as follows:*

The construction of the graph uses an auxiliary set A for containing the vertices that must be expanded in order to complete the graph.

1. *Put $V = A = \{p(\bar{a}_n)\}$ and $E = \emptyset$.*

2. *While $A \neq \emptyset$ do:*
   (a) *Select a vertex u in A with query $q(\bar{b}_n)$. $A = A \setminus \{u\}$.*
   (b) *For each rule R defining q, $R = (q(\bar{t}_n) :- l_1, \ldots, l_m)$ with $m > 0$, such that there exists $\theta = mgu(\bar{t}_n, \bar{b}_n)$, the debugger creates a set S of new vertices. Initially, we define $S = \emptyset$ and include new vertices associated to each literal l_i, $i = 1 \ldots m$ as follows:*
      i. *$i = 1$, a new vertex is included: $S = S \cup \{atom(l_1)\theta\}$.*
      ii. *$i > 1$. We consider the literal l_i. For each set of substitutions $\{\sigma_1, \ldots, \sigma_i\}$ with $dom(\sigma_1 \cdot \ldots \cdot \sigma_{i-1}) \subseteq var(l_1) \cup \cdots \cup var(l_i)$ such that for every $1 < j \leq i$:*
         - *$atom(l_{j-1})(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}) \in S$, and*

– $l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_j) \in (l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}))_{\mathcal{M}}$

include a new vertex in S*:*

$$S = S \cup \{atom(l_i)(\sigma_1 \cdot \ldots \cdot \sigma_i)\}$$

(c) *For each vertex* $v \in S$*, test whether there exists already a vertex* $v' \in V$ *such that* v *and* v' *are variants (i.e., there is a variable renaming). There are two possibilities:*
   – *There is such a vertex* v'*. Then,* $E = E \cup \{(u, v')\}$*. That is, if the vertex already exists, we simply add a new edge from the selected vertex* u *to* v'*.*
   – *Otherwise,* $V = V \cup \{v\}$*,* $A = A \cup \{v\}$*, and* $E = E \cup \{(u, v)\}$*.*

3. *Complete the vertices including the computed answer* $Q_{\mathcal{M}}$ *of every subquery* Q*.*

End of Definition

We will use the notation $[Q = Q_{\mathcal{M}_A}]$ for representing the content of the vertices. The values $Q_{\mathcal{M}_A}$ included at step 3 can be obtained from the underlying deductive database system by submitting each Q. The vertex is *valid* if $Q_{\mathcal{M}_A}$ is the expected answer for Q, and *invalid* otherwise.

Figure 2 shows the *CG* for the query `planet(X)` w.r.t. the program of Figure 1. The first vertex included in the graph at step 1 corresponds to `planet(X)`.

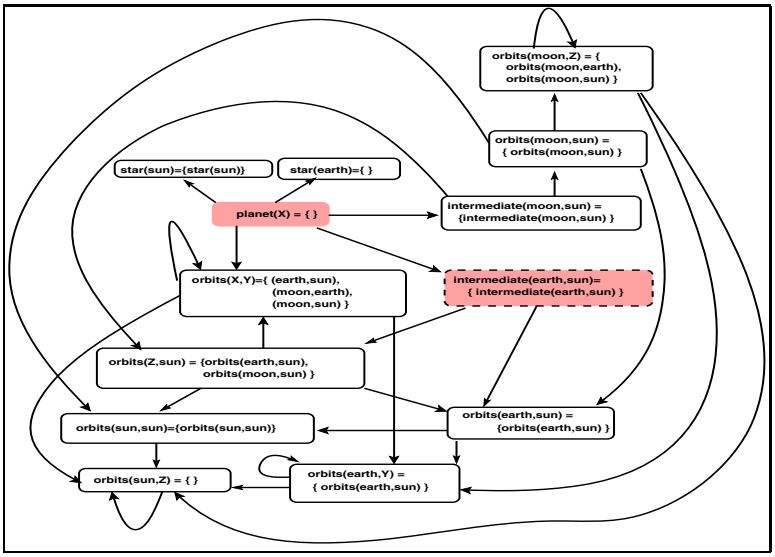


Fig. 2. CG for the Query `planet(X)` w.r.t. the Program of Figure 1

From this vertex and by using the only program rule for `planet`, four new vertices are added, the first one corresponding to the first literal `orbits(X,Y)`. Since two values of `Y` satisfy this subquery, namely `Y=sun` and `Y=earth`, the definition introduces two new vertices for the next literal `star(Y)`, `star(sun)` and `star(earth)`. The last one produces the empty answer, but `star(sun)` succeeds. Then, the last literal in the rule, `not(intermediate(X,Y))`, yields vertices for the two values of `X` and the only value of `Y` that satisfies the two previous literals. Observe, however, that the vertices for this literal are introduced in the graph without the negation, i.e., the CG will contain only subqueries for atoms. This simplifies the questions asked to the user during the navigation phase, and can be done without affecting the correctness of the technique because the validity of the positive literal implies the validity of its negation, and the other way round (although the type of associated error changes, see Proposition 2). The rest of the vertices of the example graph are built expanding the successors of `planet(X)` and repeating the process until no more vertices can be added.

The termination of the process is guaranteed because in our setting the signature is finite and the CG cannot have two occurrences of the same vertex due to step 2c, which introduces edges between existing vertices instead of creating new ones when possible.

The next proposition relates the elements of the computed answer stored at a vertex u with the immediate successors of u and vice versa.

Proposition 3. *Let $u = [Q = Q_{\mathcal{M}}]$ be a vertex in the computation graph G of some query w.r.t. a program P. Let $p(\bar{s}_n)$ be an instance of Q. Then $p(\bar{s}_n) \in Q_{\mathcal{M}}$ iff there exist a rule variant $p(\bar{t}_n) \ :- \ l_1, \ldots, l_m$ and a substitution θ such that among the successors of u in G there are vertices of the form $[atom(l_i)\sigma_i = A_i]$ with $\theta \geq \sigma_i$ for each $i = 1 \ldots m$.*

Proof. First, we suppose that $p(\bar{s}_n) \in Q_{\mathcal{M}}$. Let $(p(\bar{t}_n) \ :- \ l_1, \ldots, l_m) \in P$ and $\theta \in Subst$ be respectively the associated rule and the associated substitution to $p(\bar{s}_n)$, as defined in Proposition 1, item 2. Then, by this proposition, $p(\bar{s}_n) = p(\bar{t}_n)\theta$, which implies the existence of the $mgu(p(\bar{t}_n), p(\bar{s}_n))$ because we always consider rule variants and, hence, $var(p(\bar{s}_n)) \cap var(p(\bar{t}_n)) = \emptyset$. Then, the algorithm of Definition 5, item 2b, ensures that this program rule produces new vertices, successors of u in G. We check by induction on the number of literals in the body rule, m, that these vertices are of the form $[atom(l_i)\sigma_i = A_i]$ with $\theta \geq \sigma_i$ for $i = 1 \ldots m$. If $m = 0$, the result holds trivially. If $m = 1$, then there is a successor of u of the form $[atom(l_1)\theta = A_1]$ (item 2(b)i of Definition 5). For the inductive case $m > 1$, we assume that there is already a successor of u of the form $[atom(l_{m-1})\sigma = A_{m-1}]$, $\theta \geq \sigma$, i.e., $\theta = \sigma \cdot \sigma_m$ for some substitution σ_m. By the graph construction algorithm, σ must be of the form $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_{m-1}$. By Proposition 1, item 2, $M \models l_{m-1}\theta$, i.e., $l_{m-1}\theta \in A_{m-1}$ (by the same Proposition 1, $l_{m-1}\theta$ is ground, and therefore must be part of the computed answer for $l_{m-1}\sigma$). Hence, $l_{m-1}(\sigma_1 \cdot \ldots \cdot \sigma_m) \in A_{m-1}$. In these conditions, the algorithm of Definition 5 includes a new successor of u with the form $atom(l_m)(\sigma_1 \cdot \ldots \cdot \sigma_m)$.

Conversely, if there exists a program rule, a substitution, and successor vertices as the proposition indicates, then it can be proved by a similar reasoning

that $M \models (l_1, \ldots, l_m)\theta$, and then, Proposition 1, item 3, ensures that $p(\bar{s}_n) = p(\bar{t}_n)\theta$ verifies $\mathcal{M} \models p(\bar{s}_n)$. In particular, if $p(\bar{s}_n)$ is ground, this means that $p(\bar{s}_n) \in \mathcal{M}$. □

The relation among a vertex and its descendants also relates the validity of them, as the following proposition states:

Proposition 4. *Let G be a computation graph and $u = [p(\bar{s}_n) = A]$ be an invalid vertex of G such that p is a well-defined relation. Then, u has some invalid successor v in G.*

Proof. If the vertex u is invalid, then A is either a wrong or a missing answer for $p(\bar{s}_n)$, which means that it contains either a wrong or a missing instance.

Suppose that $p(\bar{s}_n)\theta$ is a wrong instance for some $\theta \in subst$. Since $p(\bar{s}_n)\theta \in (p(\bar{s}_n))_{\mathcal{M}}$, by Proposition 1, there exists some associated program rule $R \in P$ and substitution θ' s.t. $(R)\theta' = (p(\bar{t}_n) \ :- l_1, \ldots l_m)\theta'$, with $\mathcal{M} \models l_i\theta'$ for all $i = 1 \ldots m$ and $p(\bar{t}_n)\theta' = p(\bar{s}_n)\theta$. From Proposition 3, it can be deduced that there are successor vertices of u of the form $[atom(l_i)\sigma_i = A_i]$ for all $i = 1 \ldots m$, with $\theta' \geq \sigma_i$. Assume that all these vertices are valid. Then, for each $i = 1 \ldots m$ we can ensure the validity of $l_i\theta'$ because:

- If l_i is a positive literal, from the validity of the answer for $atom(l_i)\sigma_i$ we obtain the validity of the more particular $atom(l_i)\theta'$ (the validity of a formula entails the validity of its instances).
- If l_i is a negative literal, from the validity of the answer for $atom(l_i)\sigma_i$ we obtain the validity of the answer for $atom(l_i)\theta'$, and from this, the validity of the answer for $l_i\theta'$ (as a consequence of Proposition 2).

Then, we have that $\mathcal{M} \models (l_1, \ldots, l_m)\theta'$, but $\mathcal{M} \not\models p(\bar{t}_n)\theta'$, i.e., $(R)\theta'$ is a wrong instance. But this is not possible because p is well-defined. Therefore, some of the successors of u must be invalid.

The proof is analogous in the case of a missing answer. □

3.3 Buggy Vertices and Buggy Circuits

In the traditional declarative debugging scheme [18] based on trees, program errors correspond to *buggy nodes*. In our setting, we also need the concept of buggy node, here called *buggy vertex*, but in addition our computation graphs can include *buggy circuits*:

Definition 6 (Buggy Circuit). *Let $CG = (V, A)$ be a computation graph. We define a* buggy circuit *as a circuit $W = v_1 \ldots v_n$ s.t. for all $1 \leq i \leq n$:*

1. *v_i is invalid.*
2. *If $(v_i, u) \in A$ and u is invalid then $u \in W$.*

Definition 7 (Buggy Vertex). *A vertex is called buggy when it is invalid but all its successors are valid.*

The next result proves that a computation graph corresponding to an initial error symptom, i.e., including some invalid vertex, contains either a buggy circuit or a buggy vertex.

Proposition 5. *Let G be a computation graph containing an invalid vertex. Then, G contains either a buggy vertex or a buggy circuit.*

Proof. Let G be the computation graph and $u \in G$ an invalid vertex. From G, we obtain a new graph G' by including all the invalid vertices reachable from u. More formally, G' is the subgraph of G generated by the set of vertices

$$\{v \in G \mid \text{ there is a path } \Pi = path_G(u, v) \text{ and } w \text{ invalid for every } w \in \Pi\}$$

Now, we consider the set S of strongly connected components in G',

$$S = \{C \mid C \text{ is a strongly connected component of } G'\}$$

The cardinality of S is finite since G' is finite. Then, there must exist $C \in S$ such that $gr^+_{G'}(C) = 0$. Moreover, for all $u \in C$, $succ_G(u, u')$ means that either $u' \in C$ or u' is valid because $u' \notin C$, u' invalid, would imply $gr^+_{G'}(C) > 0$. Observe also that, by the construction of G', every $u \in C$ is invalid. Then:

- If C contains a single vertex u, then u is a buggy vertex in G.
- If C contains more than a vertex, then all its vertices form a buggy circuit in G. □

4 Soundness and Completeness

The debugging process we propose can be summarized as follows:

1. The user finds out an unexpected answer for some query Q w.r.t. some program P.
2. The debugger builds the computation graph G for Q w.r.t. P.
3. The graph is traversed, asking questions to the user about the validity of some vertices until a buggy vertex or a buggy circuit has been found.
4. If a buggy vertex is found, its associated relation is pointed out as buggy. If instead a buggy circuit is found, the set of relations involved in the circuit are shown to the user indicating that at least one of them is buggy or that the set is incomplete.

Now, we must check that the technique is reliable, i.e., that it is both sound and complete. First we need some auxiliary lemmata.

Lemma 1. *Let G be a computation graph for some query Q w.r.t. a program P, and let $C = u_1, \ldots, u_k$, with $u_k = u_1$ be a circuit in G. Then, all the u_i are of the form $[Q_i = Q_{i\mathcal{M}}]$ with Q_i associated to a positive literal in its corresponding program rule for $i = 1 \ldots k - 1$.*

Proof. It can be proved that every relation occurring in some Q_i depends recursively on itself. This means that Q_i cannot occur negatively in a clause because this would mean than P is not stratified (see Lemma 1 in [14]). □

Lemma 2. *Let $v = [p(\bar{s}_n) = \dots]$ be a vertex of some CG G obtained w.r.t. some program P with standard model $\mathcal{M}$. Let $p(\bar{t}_n) :- l_1, \dots, l_m$ be a rule in P, and θ s.t. $p(\bar{t}_n)\theta = p(\bar{s}_n)\theta$, and that $\mathcal{M} \models l_1\theta, \dots, l_k\theta$ for some $1 \leq k \leq m$. Then, v has children vertices in G of the form $[atom(l_i)\theta_i = \dots]$ for $i = 1 \dots k+1$, with $\theta \geq \theta_i$.*

Proof. The proof corresponds to that of Proposition 3, but considering only the first $k+1$ literals of the program rule. □

Observe that theoretically the debugger could be applied to any computation graph even if there is no initial wrong or missing answer. The following soundness result ensures that in any case it will behave correctly.

Proposition 6 (Soundness). *Let P be a Datalog program, Q be a query and G be the computation graph for Q w.r.t. P. Then:*

1. *Every buggy node in G is associated to a buggy relation.*
2. *Every buggy circuit in G contains either a vertex with an associated buggy relation or an incomplete set of relations.*

Proof

1. Suppose that G contains a buggy vertex $u \equiv [q(\bar{t}_n) = S]$. By definition of buggy vertex, all the immediate descendants of u are valid. Since vertex u is invalid, by Proposition 4, the relation q cannot be well-defined.
2. Suppose that G contains a buggy circuit $C \equiv u_1, \dots, u_n$ with $u_n = u_1$ and each u_i of the form $[A_i = S_i]$ for $i = 1 \dots n-1$. We consider two possibilities:
   (a) At least one of the vertices in the circuit contains a wrong answer. Let S be the set of the wrong atom instances contained in the circuit:

$$S = \{B \in S_i \wedge B \notin \mathcal{I} \mid \text{ for some } 1 \leq i < n\}$$

   Obviously, $S \subseteq \mathcal{M}$ and $S \cap \mathcal{I} = \emptyset$. Now, we consider a stratification $\{P_1, \dots, P_k\}$ of the program P and the sequence of Herbrand interpretations starting from $\emptyset$ and ending in $\mathcal{M}$ defined in item 5 of Proposition 1. We single out the first interpretation in this sequence including some element of S. Such interpretation must be of the form $T_{P_i}(I)$, with I the previous interpretation in the sequence and $1 \leq i \leq k$. Let $p(\bar{s}_n)$ be an element of $T_{P_i}(I) \cap S$. By definition of T_P, there exists a substitution θ and a program rule $(p(\bar{t}_n) :- l_1, \dots, l_m) \in P$ s.t. $p(\bar{s}_n) = p(\bar{t}_n)\theta$ and that $I \models l_i\theta$ for every $i = 1 \dots m$. By Proposition 3, each l_i, $i = 1 \dots m$, has some associated vertex V' successor of V in the CG with V' of the form $[atom(l_i)\sigma = \dots]$ with σ more general than θ. We distinguish two possibilities:

- V' is out of the circuit. Then, by the definition of buggy circuit V' is valid w.r.t. $\mathcal{I}$, which means all the instances of $l_i\theta$ are also valid w.r.t. $\mathcal{I}$. This is true independently of whether l_i is positive or negative because the validity of the answer for a query implies the validity of the answer for its negation in our setting.
- V' is in the circuit. Then, l_i is positive due to Lemma 1, and by construction, all the instances of $l_i\theta$ included in $\mathcal{I}$ are valid w.r.t. $\mathcal{I}$.

In any case, $\mathcal{I} \models l_i\theta$ for every $i = 1 \ldots m$ but $p(\bar{t}_n)\theta \notin \mathcal{I}$ and hence p is an *incorrect relation*.

(b) If none of the vertices in the buggy circuit contains a wrong answer, then every vertex contains a missing answer.
Put

$$S = \{A_i\sigma \in \mathcal{I}, A_i\sigma \notin S_i \mid \text{for some } 1 \leq i < k\}$$

i.e., S is the set of missing instances in the circuit. Next, we check that S is an uncovered set of atoms, which means that the relations in the buggy circuit form an incomplete set of relations. Let $A_j\sigma \in S$ be an atom of S with $1 \leq j \leq k$, $(p(\bar{t}_n) :- l_1, \ldots, l_m) \in P$ be a program rule, and $\theta \in Subst$ such that:
- $p(\bar{t}_n)\theta = A_j\sigma$,
- $\mathcal{I} \models l_i\theta$ for $i = 1 \ldots m$

There must exist at least one $l_i\theta \notin \mathcal{M}$, $1 \leq i \leq m$, otherwise $A_j\sigma$ would be in $\mathcal{M}$. Let r be the least index, $1 \leq r \leq m$, s.t. $l_r\theta \notin \mathcal{M}$. By Lemma 2, there is a successor of $[A_j = S_j]$ in G of the form $w = [l_r\theta' = S_r]$ with $\theta \geq \theta'$. Then, $l_r\theta$ is a missing answer for w, i.e., it is an invalid vertex (it is easy to prove that, if $l\theta$ has a missing answer, then $l\theta'$ has a missing answer for every θ' s.t. $\theta \geq \theta'$). This implies that $w \in C$, and hence l_r is a positive literal (by Lemma 1), $l_i\theta \in S$, and S is uncovered. □

After the soundness result, it remains to prove that the technique is complete:

Proposition 7 (Completeness). *Let P be a Datalog program and Q be a query with answer $Q_{\mathcal{M}}$ unexpected. Then, the computation graph G for Q w.r.t. P contains either a buggy node or a buggy circuit.*

Proof. By the construction of the computation graph, G contains a vertex for $[atom(Q) = atom(Q)_{\mathcal{M}}]$. If Q is positive, then $Q = atom(Q)$ and the vertex is of the form $[Q = Q_{\mathcal{M}}]$. Then, by hypothesis, $Q_{\mathcal{M}}$ is unexpected, and therefore the vertex is invalid. If Q is negative and it has an unexpected answer, it is straightforward to check that $atom(Q)$ also produces an unexpected answer and hence $[atom(Q) = atom(Q)_{\mathcal{M}}]$ is also invalid. Then, the result is a direct consequence of Proposition 5.

5 Implementation

The theoretical ideas explained so far have been implemented in a debugger included as part of the Datalog system DES [13]. The *CG* is built after the user

has detected some unexpected answer. The values $(Q, Q_{\mathcal{M}_A})$ are stored along the computation and can be accessed afterwards without repeating the computation, thus increasing the efficiency of the graph construction.

A novelty of our approach is that it allows the user to choose working either at clause level or at predicate level, depending on the grade of precision that the user needs, and its knowledge of the intended interpretation $\mathcal{I}$. At predicate level, the debugger is able to find a buggy relation or an incomplete set of relations. At clause level, the debugger can provide additional information, namely the rule which is the cause of error.

For instance, next is the debugging session at predicate level for the query `planet(X)` w.r.t. our running example:

```
DES> /debug planet(X) p

Info: Starting debugger...

Is orbits(sun,sun) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(earth,Y) = {orbits(earth,sun)}
                      valid(v)/invalid(n)/abort(a) [v]? v
Is intermediate(earth,sun) = {intermediate(earth,sun)}
                      valid(v)/invalid(n)/abort(a) [v]? n
Is orbits(sun,Y) = {}  valid(v)/invalid(n)/abort(a) [v]? v
Is orbits(X,sun) = {orbits(earth,sun),orbits(moon,sun)}
                         valid(v)/invalid(n)/abort(a) [v]? v

Error in relation: intermediate/2
Witness query:
      intermediate(earth,sun) = {intermediate(earth,sun)}
```

The first question asks whether the query `orbits(sun,sun)` is expected to fail, i.e., it yields no answer. This is the case because we do not consider the sun orbiting around itself. The answer to the second question is also `valid` because the earth orbits only the sun in our intended model. But the answer to the next question is `invalid`, since the query `intermediate(earth,sun)` should fail because the earth orbits directly the sun. The next two answers are `valid`, and with this information the debugger determines that there is a buggy node in the *CG* corresponding to the relation `intermediate/2`, which is therefore buggy. The *witness query* shows the instance that contains the unexpected instance. This information can be useful for locating the bug.

In order to minimize the number of questions asked to the user, the tool relies on a navigation strategy similar to the *divide & query* presented in [12] for deciding which vertex is selected at each step. In other paradigms it has been shown that this strategy requires an average of $\log_2 n$ questions to find the bug [19], with n the number of nodes in the computation tree. Our experiments confirms that this is also the case when the CGs are in fact trees, i.e., they do not contain cycles, which occurs very often. In the case of graphs containing cycles the results also show this tendency, although a more extensive number of experiments is still needed.

6 Conclusions and Future Work

We have applied declarative debugging to Datalog programs. The debugger detects incorrect fragments of code starting from an unexpected answer. In order to find the bug, the tool requires the help of the user as an external oracle answering questions about the validity of the results obtained for some subqueries. We have proved formally the completeness and soundness of the technique, thus proposing a solid foundations for the debugging of Datalog programs. During the theoretical study, we have found that the traditional errors considered usually in logic programming are not enough in the case of Datalog where a new kind of error, the incomplete sets of predicates, can occur.

The theoretical ideas have been set in practice by developing a declarative debugger for the Datalog system DES. The debugger allows diagnosing both missing and wrong answers, which constitute all the possible errors symptoms of a Datalog program. Although a more extensive workbench is needed, the preliminary experiments are encouraging about the usability of the tool. The debugger allows to detect readily errors which otherwise would take considerable time. This is particularly important for the DES system, which has been developed with educational purposes. By using the debugger, the students can find the errors in a program by considering only its declarative meaning and disregarding operational issues.

From the point of view of efficiency, the results are also quite satisfactory. The particular characteristics of DES make all the information necessary for producing the graph available after each computation. The answers to each subquery, therefore, are not actually computed in order to build the graph but simply pointed to. This greatly speeds up the graph construction and keeps small the size of the graph even for large computations.

As future work, we consider the possibility of allowing more elaborated answers from the user. For instance, indicating that a vertex is not only invalid but also that it contains a wrong answer. The identification of such an answer can greatly reduce the number of questions. Another task is to develop and compare different navigation strategies for minimizing the number of questions needed for finding the bug.

References

1. Ramakrishnan, R., Ullman, J.: A survey of research on Deductive Databases. The Journal of Logic Programming 23(2), 125–149 (1993)
2. Beeri, C., Ramakrishnan, R.: On the power of magic. In: Proceedings of the Sixth ACM Symposium on Principles of Database Systems, pp. 269–284 (1987)
3. Dietrich, S.W.: Extension tables: Memo relations in logic programming. In: SLP, pp. 264–272 (1987)
4. Arora, T., Ramakrishnan, R., Roth, W.G., Seshadri, P., Srivastava, D.: Explaining program execution in deductive systems. In: Deductive and Object-Oriented Databases, pp. 101–119 (1993)
5. Wieland, C.: Two explanation facilities for the deductive database management system DeDEx. In: Kangassalo, H. (ed.) ER 1990, pp. 189–203, ER Institute (1990)

6. Specht, G.: Generating explanation trees even for negations in deductive database systems. In: Proceedings of the 5th Workshop on Logic Programming Environments, Vancouver, Canada (1993)
7. Russo, F., Sancassani, M.: A declarative debugging environment for Datalog. In: Proceedings of the First Russian Conference on Logic Programming, pp. 433–441. Springer, London (1992)
8. Baral, C.: Knowledge representation, reasoning, and declarative problem solving. Cambridge University Press, Cambridge (2003)
9. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning, Lake District, UK, pp. 77–84 (May 2006)
10. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP Programs by Means of ASP. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 31–43. Springer, Heidelberg (2007)
11. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A new proposal for debugging datalog programs. In: 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2007) (June 2007)
12. Shapiro, E.: Algorithmic Program Debugging. In: ACM Distiguished Dissertation. MIT Press, Cambridge (1982)
13. Sáenz-Pérez, F.: Datalog Educational System. User's Manual. Technical Report 139-04, Facultad de Informática, Universidad Complutense de Madrid (2004), `http://des.sourceforge.net/`
14. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of deductive databases and logic programming, pp. 89–148. Morgan Kaufmann Publishers Inc., San Francisco (1988)
15. Ullman, J.: Database and Knowledge-Base Systems Vols. I (Classical Database Systems) and II (The New Technologies). Computer Science Press (1995)
16. Chandra, A.K., Harel, D.: Horn clauses queries and generalizations. J. Log. Program. 2(1), 1–15 (1985)
17. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K. (eds.) Proceedings of the Fifth International Conference on Logic Programming, pp. 1070–1080. MIT Press, Cambridge (1988)
18. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
19. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: WCFLP 2005: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming, pp. 8–13. ACM Press, New York (2005)

Applying Constraint Logic Programming to SQL Test Case Generation*

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{rafa,fernan}@sip.ucm.es, ygarciar@fdi.ucm.es

Abstract. We present a general framework for generating SQL query test cases using Constraint Logic Programming. Given a database schema and a SQL view defined in terms of other views and schema tables, our technique generates automatically a set of finite domain constraints whose solutions constitute the test database instances. The soundness and correctness of the technique w.r.t. the semantics of Extended Relational Algebra is proved. Our setting has been implemented in an available tool covering a wide range of SQL queries, including views, subqueries, aggregates and set operations.

1 Introduction

Checking the correctness of a piece of software is generally a labor-intensive and time-consuming work. In the case of the declarative relational database language SQL [17] this task becomes especially painful due to the size of actual databases; it is usual to find select queries involving thousands of database rows, and reducing the size of the databases for testing is not a trivial task. The situation becomes worse when we consider correlated views. Thus, generating test database instances to show the possible presence of faults during unit testing has become an important task. Much effort has been devoted to studying and improving the different possible coverage criteria for SQL queries (see [21,1] for a general discussion, [3,18] for the particular case of SQL). However, the common situation of queries defined through correlated views had not yet been considered.

In this work we address the problem of generating test cases for checking correlated SQL queries. A set of related views is transformed into a constraint satisfiability problem whose solution provides an instance of the database which will constitute a test case. This technique is known as constraint-based test data generation [7], and has already been applied to SQL basic queries [20]. Other recent works [2] use RQP (Reverse Query Processing) to generate different database instances for a given query and a result of that query. In [6] the problem of generating database test cases in the context of Java programs interacting with relational databases, focusing on the relation between SQL queries and

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S-0505/TIC/0407, S2009TIC-1465 and UCM-BSCH-GR58/08-910502

M. Blume, N. Kobayashi, and G. Vidal (Eds.): FLOPS 2010, LNCS 6009, pp. 191–206, 2010.

program values. The contributions of our work w.r.t. previous related proposals are twofold:

First, as mentioned above, the previous works focus on a single SQL query instead of considering the more usual case of a set of correlated views. Observe that the problem of test case generation for views cannot be reduced to solving the problem for each query separately. For instance, consider the two simple views, that assume the existence of a table t with one integer attribute a:

```
create view v2(c) as select v1.b from v1 where v1.b>5;
create view v1(b) as select t.a  from t  where t.a>8;
```

A *positive* test case for $v2$ considering its query as a non-correlated query could consist of a single row for $v1$ containing for instance $v1.b = 6$, since $6 > 5$ and therefore this row fulfils the $v2$ condition. However, 6 is not a possible value for $v1.b$ because $v1.b$ can contain only numbers greater than 8. Therefore the connection between the two views must be taken into account (a valid positive test case would be for instance a single row in t with $t.a = 9$).

Second, we present a formal definition of the algorithm for defining the constraints. This definition allows us to prove the soundness and (weak) completeness of the technique with respect to the Extended Relational Algebra [9].

The next section presents the basis of our SQL setting. Section 3 introduces the concept of positive and negative test cases, while Section 4 defines the constraints whose solution will constitute our test cases. This section also introduces the main theoretical result, which is proven in Appendix A. Section 5 discusses the prototype implementation and Section 6 presents the conclusions.

2 Representing SQL Queries

The first formal semantics for relational databases were based on the concept of set (e.g. relational algebra, tuple calculus [4]). However these formalisms are incomplete with respect to the treatment of non-relational features such as repeated rows and aggregates, which are part of practical languages such as SQL. Therefore, other semantics based on multisets [5], also known in this context as *bags*, have been proposed. In this paper we adopt the point of view of the *Extended Relational Algebra* [12,9]. We start by defining the concepts of database schemas and instances but with a Logic Programming flavor. In particular the database instance rows will be considered logic substitutions of attributes by values.

2.1 Relational Database Schemas and Instances

A *table schema* is of the form $T(A_1, \ldots, A_n)$, with T the table name and A_i attribute names for $i = 1 \ldots n$. We will refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type (*integer*, *string*, ...) represented by $type(T.A)$. An *instance* of a table schema $T(A_1, \ldots, A_n)$ will be represented as a finite multiset of functions (called rows) $\{\!|\mu_1, \mu_2, ..., \mu_m|\!\}$

such that $dom(\mu_i) = \{T.A_1, \ldots, T.A_n\}$, and $\mu_i(T.A_j) \in type(T.A_j)$ for every $i = 1, \ldots, m$, $j = 1, \ldots, n$. Observe that we qualify the attribute names in the domain by table names. This is done because in general we will be interested in rows that combine attributes from different tables, usually as result of cartesian products. In the following, it will be useful to consider each attribute $T.A_i$ in $dom(\mu)$ as a logic variable, and μ as a logic substitution.

The concatenation of two rows μ_1, μ_2 with disjoint domain is defined as the union of both functions represented as $\mu_1 \odot \mu_2$. Given a row μ and an expression e we use the notation $e\mu$ to represent the value obtained applying the substitution μ to e. Analogously, let S be a multiset of rows $\{\!|\mu_1, \ldots, \mu_n|\!\}$ and let e be an expression. Then $(e)S$ represents the result of replacing each attribute $T.A$ occurring in an *aggregate subexpression* of e by the multiset $\{\!|\mu_1(T.A), \ldots, \mu_n(T.A)|\!\}$. The attributes $T.B$ not occurring in aggregate subexpressions of e must take the same value for every $\mu_i \in S$, and are replaced by such value. For instance, let $e = sum(T.A)+T.B$ and $S = \{\!|\mu_1, \mu_2, \mu_3|\!\}$ with $\mu_1 = \{T.A \mapsto 2, T.B \mapsto 5\}$, $\mu_2 = \{T.A \mapsto 3, T.B \mapsto 5\}$, $\mu_3 = \{T.A \mapsto 4, T.B \mapsto 5\}$. Then $(e)S = sum(\{\!|2, 3, 4|\!\}) + 5$. If $dom(\mu) = \{T.A_1, \ldots, T.A_n\}$ and $\nu = \{U.A_1 \mapsto T.A_1, \ldots, U.A_n \mapsto T.A_n\}$ (i.e., ν is a table renaming) we will use the notation μ^U to represent the substitution composition $\nu \circ \mu$. The previous concepts for substitutions can be extended to multisets of rows in a natural way. For instance, given the multiset of rows S and the row μ, $S\mu$ represents the application of μ to each member of the multiset.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{C}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables, $\mathcal{C}$ a finite set of database constrains and $\mathcal{V}$ a finite set of views (defined below). In this paper we consider only *primary key* and *foreign key* constraints, defined as traditionally in relational databases (see Subsection 4.1). A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$ verifying $\mathcal{C}$ (thus we only consider *valid instances*). To represent the instance of a table T in d we will use the notation $d(T)$. A *symbolic database instance* d_s is a database instance whose rows can contain logical variables. We say that d_s is satisfied by a substitution μ when $(d_s\mu)$ is a database instance. μ must substitute all the logic variables in d_s by domain values.

2.2 Extended Relational Algebra and SQL Queries

Next we present the basics of Extended Relational Algebra (ERA from now on) [12,9] which will be used as semantics of our framework. There are other approaches for defining SQL semantics such as [14], but we have chosen ERA because it provides an *operational* semantics very suitable for proving the correctness of our technique. Let R and S be multisets. Let μ be any row occurring n times in R and m times in S. Then ERA consists of the following operations:

- Unions and intersections. The union of R and S, is a multiset $R \cup S$ in which the row μ occurs $n + m$ times. Analogously $R \cap S$, the intersection of R and S, is a multiset in which the row μ occurs $min(n, m)$ times.

- Projection. The expression $\pi_{e_1 \mapsto A_1, \ldots, e_n \mapsto A_n}(R)$ produces a new relation producing for each row $\mu \in R$ a new row $\{A_1 \mapsto e_1\mu, \ldots, A_n \mapsto e_n\mu\}$. The resulting multiset has the same number of rows as R.
- Selection. Denoted by $\sigma_C(R)$, where C is the condition that must be satisfied for all rows in the result. The selection operator on multisets applies the selection condition to each row occurring in the multiset independently.
- Cartesian products. Denoted as $R \times S$, each row in the first relation is paired with each row in the second relation.
- Renaming. The expression $\rho_S(R)$ changes the name of the relation R to S and the expression $\rho_{A/B}(R)$ changes the name of the attribute A of R to B.
- Aggregate operators. These operators are used to aggregate the values in one column of a relation. Here we consider sum, avg, min, max and count.
- Grouping operator. Denoted by γ, this operator allows us to consider the rows of a relation in groups, corresponding to the value of one or more attributes and to aggregate only within each group. This operation is denoted by $\gamma_L(R)$, where L is a list of elements, each one either a grouping attribute, that is, an attribute of the relation R to which the γ is applied, or an aggregate operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregate in the result, an arrow and a new name are appended to the aggregate. It is worth observing that $\gamma_L(R)$ will contain one row for each maximal group, i.e., for each group not strictly contained in a larger group.

A relational database can be consulted by using queries and views defined over other views and queries. Queries are select SQL statements. In our setting we allow three kind of queries:

- *Basic queries* of the form:

  $$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w;$$

  with R_j tables or views for $j = 1 \ldots m$, e_i, $i = 1 \ldots n$ expressions involving constants, predefined functions and attributes of the form $B_j.A$, $1 \leq j \leq m$, and A an attribute of R_j. The meaning of any query Q in ERA is denoted $\langle Q \rangle$. In the case of basic queries is

  $$\langle Q \rangle = \Pi_{e_1 \to E_1, \ldots, e_n \to E_n}(\sigma_{C_w}(R))$$

  where $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$.
- *Aggregate queries*, including group by and having clauses:

  $$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ A'_1, \ldots, A'_k\ \mathsf{having}\ C_h;$$

  In this case, the equivalent ERA expression is the following:

  $$\langle Q \rangle = \Pi_{e'_1 \to E_1, \ldots, e'_n \to E_n}(\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$$

  where $L = \{A'_1, \ldots, A'_k, u_1 \mapsto U_1, \ldots, u_l \mapsto U_l\}$, R defined as in the previous case, u_i, $1 \leq i \leq l$ the aggregate expressions occurring either in the select or in the having clauses, U_i new attribute names, e'_j, $j = 1 \ldots n$ the result of replacing each occurrence of u_i by U_i in e_j and analogously for C'_h.

– *Set queries* of the form $Q = V_1$ {union, intersection} V_2;
with V_1, V_2 views (defined below) with the same attribute names. The meaning of set queries in ERA is represented by $\cup$ and $\cap$ multiset operators for union and intersection, respectively: $\langle Q \rangle = \langle V_1 \rangle$ $\{\cup, \cap\}$ $\langle V_2 \rangle$

In order to simplify our framework we assume queries such that:

– Without loss of generality we assume that the where and having clauses only contain existential subqueries of the form exists Q (or not exists Q). It has been shown that other subqueries of the form ... in Q, ... any Q or ... all Q can be translated into equivalent subqueries with exists and not exists (see for instance [10]). Analogously, subqueries occurring in arithmetic expressions can be transformed into exists subqueries.
– The from clause does not contain subqueries. This is not a limitation since all the subqueries in the from clause can be replaced by views.
– We also do not allow the use of the distinct operator in the select clause. It is well-known that queries using this operator can be replaced by equivalent aggregate queries without distinct. In the language of ERA, this means that the operator δ for eliminating duplicates –not used here– is a particular case of the aggregate operator γ (see [9]).
– Our setting does not allow: recursive queries, the minus operator, join operations, and null values. All these features, excepting the recursive queries, can be integrated in our setting, although they have not been considered here for simplicity.

We also need to consider the concept of *views*, which can be thought of as new tables created dynamically from existing ones by using a query and allowing the renaming of attributes.

The general form of a view is: create view V(A_1, ..., A_n) as Q, with Q a query and $V.A_1, \ldots V.A_n$ the name of the view attributes. Its meaning is defined as: $\langle V \rangle = \Pi_{E_1 \to V.A_1, \ldots, E_n \to V.A_n} \langle Q \rangle$, with $E_1, \ldots, E_n$ the attribute names of the select clause in Q. In general, we will use the name *relation* to refer to either a table or a view. The semantics of a table T in a given instance d is defined simply as its rows: $\langle T \rangle = d(T)$. A view query can depend on the tables of the schema and also on previously defined views (no recursion between views is allowed). Thus, the *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in the from clause of its query. This concept can be easily extended to queries by assuming some arbitrary name labeling the root node, and to tables, where the dependency tree will be a single node labeled by the table name.

3 SQL Test Cases

In the previous section we have defined an operational semantics for SQL. Now we are ready for defining the concept of test case for SQL. We distinguish between positive and negative test cases:

Definition 1. *We say that a non-empty database instance d is a* positive test case *(PTC) for a view V when $\langle V \rangle \neq \emptyset$.*

Observe that our definition excludes implicitly the empty instances, which will be considered as neither positive nor negative test cases. We require that the (positive or negative) test case contains at least one row that will act as witness of the possible error in the view definition. The overall idea is that we consider d a PTC for a view when the corresponding query answer is not empty. In a basic query this means that at least one tuple in the query domain satisfies the where condition. In the case of aggregate queries, a PTC will require finding a valid aggregate verifying the having condition, which in turn implies that all its rows verify the where condition. If the query is a set query, then the ranges are handled according to the set operation involved.

The negative test cases (NTC) are defined by modifying the initial queries and then applying the concept of positive test case. With this purpose we use the notation Q_{C_w} and $Q_{(C_w,C_h)}$ to indicate that C_w is the where condition in Q and C_h is the having condition in Q (when Q is an aggregate query). If Q_{C_w} is of the form select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where C_w; then the notation $Q_{not(C_w)}$ represents select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where not(C_w); and analogously for $Q_{(C_w,C_h)}$ and $Q_{(not(C_w),C_h)}$, $Q_{(C_w,not(C_h))}$, and $Q_{(not(C_w),not(C_h))}$. For instance, in the case of basic query, we expect that a NTC will contain some row in the domain of the view not verifying the where condition:

Definition 2. *We say that a database instance d is a NTC for a view V with associated basic query Q_{C_w} when d is a PTC for $Q_{not(C_w)}$.*

In queries containing aggregate functions, the negative case corresponds either to a tuple that does not satisfy the where condition, or to an aggregate not satisfying the having condition:

Definition 3. *We say that a database instance d is a NTC for a view V with associated aggregate query $Q_{(C_w,C_h)}$ if it is a PTC for either $Q_{(not(C_w),C_h)}$, $Q_{(C_w,not(C_h))}$, or $Q_{(not(C_w),not(C_h))}$.*

Next is the definition of negative test cases for set queries:

Definition 4. *We say that a database instance d is a NTC for a view with query defined by:*

- *A query union of Q_1, Q_2, if d is a NTC for both Q_1 and Q_2.*
- *A query intersection of Q_1, Q_2, if d is a NTC for either Q_1 or Q_2.*

The main advantage of defining NTCs in terms of PTCs is that only a positive test case generator must be implemented. The previous definitions are somehow arbitrary depending on the coverage. For instance the NTCs for views with aggregate queries $Q_{(C_w,C_h)}$ could be defined simply as the PTCs for $Q_{(not(C_w),not(C_h))}$.

It is possible to obtain a test case which is both positive and negative at the same time thus achieving *predicate coverage* with respect to the where and having conditions (in the sense of [1]). We will call these tests PNTCs. For instance, for the query select A from T where A=5; with T a table with a single attribute A, the test case d s.t. $d(T) = \{\!|\mu_1, \mu_2|\!\}$ with $\mu_1 = \{T.A \mapsto 5\}$, $\mu_2 = \{T.A \mapsto X\}$, X any value different from 5, is a PNTC. However, this is not always possible. For instance, the query select R_1.A from T R_1 where R_1.A=5 and not exists (select R_2.A from T R_2 where R_2.A<>5); allows both PTCs and NTCs but no PNTC. Our tool will try to generate a PNTC for a view first, but if it is not possible it will try to obtain a PTC and a NTC separately.

4 Generating Constraints

The main goal of this paper is to use Constraint Logic Programming for generating test cases for SQL views. The process can be summarized as follows:

1. First, create a *symbolic database instance*. Each table will contain an arbitrary number of rows, and each attribute value in each row will correspond to a fresh logic variable with its associated domain integrity constraints.
2. Establish the constraints corresponding to the integrity of the database schema: primary and foreign keys.
3. Represent the problem of obtaining a test case as a constraint satisfaction problem.

Next, we explain in detail phases 2 and 3.

4.1 Primary and Foreign Keys

Given a relation R with primary key $pk(R) = \{A_1, \ldots, A_m\}$ and a symbolic instance d such that $d(R) = \{\!|\mu_1, \ldots, \mu_n|\!\}$, we check that d satisfies $pk(R)$ by establishing the following constraint:

$$\bigwedge_{i=1}^{n} \Big(\bigwedge_{j=i+1}^{n} \Big(\bigvee_{k=1}^{m} (\mu_i(R.A_k) \neq \mu_j(R.A_k)) \Big) \Big)$$

that is, different rows must contain different values for the primary key. Given two relations R_1, R_2 and an instance d such that $d(R_1) = \{\!|\mu_1, \ldots, \mu_{n_1}|\!\}$, $d(R_2) = \{\!|\nu_1, \ldots, \nu_{n_2}|\!\}$ a *foreign key* from R_1 referencing R_2, denoted by $fk(R_1, R_2) = \{(A_1, \ldots, A_m), (B_1, \ldots, B_m)\}$, indicates that for each row μ in R_1 there is a row ν in R_2 such that $(A_1\mu, \ldots, A_m\mu) = (B_1\nu, \ldots, B_m\nu)$. Foreign keys are represented with the following constraints:

$$\bigwedge_{i=1}^{n_1} \Big(\bigvee_{j=1}^{n_2} \Big(\bigwedge_{k=1}^{m} \big(\mu_i(R_1.A_k) = \nu_j(R_2.B_k) \big) \Big) \Big)$$

4.2 SQL Test Cases as a Constraint Satisfaction Problem

Now we are ready for describing the technique supporting our implementation. First we introduce the two following auxiliary operations over multisets:

Definition 5. *Let $A = \{\!|(a_1, b_1), \ldots, (a_n, b_n)|\!\}$. Then we define the operations Π_1 and Π_2 as follows: $\Pi_1(A) = \{\!|a_1, \ldots, a_n|\!\}$, $\Pi_2(A) = \{\!|b_1, \ldots, b_n|\!\}$.*

The following definition will associate a first order formula to every possible row of a relation. The idea is that the row will be in the relation instance iff the formula is satisfied.

Definition 6. *Let D be a database schema and d a database instance. We define $\theta(R)$ for every relation R in D as a multiset of pairs (ψ, u) with ψ a first order formula, and u a row. This multiset is defined as follows:*

1. *For every table T in D such that $d(T) = \{\!|\mu_1, \ldots, \mu_n|\!\}$:*

$$\theta(T) = \{\!|(true, \mu_1), \ldots, (true, \mu_n)|\!\}$$

2. *For every view $V =$ create view $V(A_1, \ldots, A_n)$ as Q,*

$$\theta(V) = \theta(Q)\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$$

   with $E_1, \ldots, E_n$ the attribute names in the select *clause of Q.*
3. *If Q is a basic query of the form:*

$$\text{select } e_1\ E_1, \ldots, e_n\ E_n \text{ from } R_1\ B_1, \ldots, R_m\ B_m \text{ where } C_w;$$

   Then:

$$\theta(Q) = \{\!|(\psi_1 \wedge \ldots \wedge \psi_m \wedge \varphi(C_w, \mu),\ s_Q(\mu)) \mid (\psi_1, \nu_1) \in \theta(R_1), \ldots, (\psi_m, \nu_m) \in \theta(R_m), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\!\}$$

   with $s_Q(\mu) = \{E_1 \mapsto (e_1\mu), \ldots, E_n \mapsto (e_n\mu)\}$, and the first order formula $\varphi(C, \mu)$ is defined as

   - *if C does not contain subqueries, $\varphi(C, \mu) = C'\mu$, with C' obtained from C by replacing every occurrence of* and *by $\wedge$,* or *by $\vee$, and* not *by $\neg$.*
   - *if C does contain subqueries, let* Q= (exists Q_E) *be an outermost existential subquery in C, with $\theta(Q_E) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$. Let C' be the result of replacing Q by* true *in C. Then $\varphi(C, \mu) = (\vee_{i=1}^{n} \psi_i) \wedge \varphi(C', \mu)$.*

4. *For set queries:*
   - *$\theta(V_1$ union $V_2) = \theta(V_1) \cup \theta(V_2)$ with $\cup$ the multiset union.*
   - *$(\psi, \mu) \in \theta(V_1$ intersection $V_2)$ with cardinality k iff $(\psi_1, \mu) \in \theta(V_1)$ with cardinality k_1, $(\psi_2, \mu) \in \theta(V_2)$ with cardinality k_2, $k = min(k_1, k_2)$ and $\psi = \psi_1 \wedge \psi_2$.*

5. If Q includes aggregates, then it is of the form:

$$\begin{array}{l}\mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1, \ldots, R_m\ B_m \\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ e'_1, \ldots, e'_k\ \mathsf{having}\ C_h\end{array}$$

Then we define:

$$\begin{array}{rl} P = & \{\!|(\psi, \mu)\ \mid\ (\psi_1, \nu_1), \ldots, (\psi_m, \nu_m) \in (\theta(R_1) \times \ldots \times \theta(R_m)) \\ & \qquad \psi = \psi_1 \wedge \ldots \wedge \psi_m,\ \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}\ |\!\} \\ \theta(Q) = & \{\!|(\bigwedge(\Pi_1(A)) \wedge aggregate(Q, A),\ s_Q(\Pi_2(A)))\ |\ A \subseteq P\ |\!\}\end{array}$$

$$\begin{array}{l} aggregate(Q, A) = group(Q,\Pi_2(A)) \wedge maximal(Q,A) \wedge \varphi(C_h, \Pi_2(A)) \\ group(Q, S) = (\bigwedge\{\!|\varphi(C_w, \mu)\ \mid \mu \in S|\!\}\ ) \wedge \\ \qquad\qquad (\bigwedge\{\!|\ ((e'_1)\nu_1 = (e'_1)\nu_2 \wedge \ldots \wedge (e'_k)\nu_1 = (e'_k)\nu_2)\ \mid\ \nu_1, \nu_2 \in S|\!\}\ ) \\ maximal(Q, A) = \bigwedge\{\!|\ (\neg\psi \vee \neg group(Q, \Pi_2(A) \cup \{\!|\mu|\!\})\ \mid\ (\psi, \mu) \in (P - A)\ |\!\}\end{array}$$

Observe that the notation $s_Q(x)$ with Q a query is a shorthand for the row μ with domain $\{E_1, \ldots, E_n\}$ such that $(E_i)x = (e_i)x$, with $i = 1 \ldots n$, with select $e_1\ E_1, \ldots, e_n\ E_n$ the select clause of Q. If E_i's are omitted in the query, it is assumed that $E_i = e_i$.

Example 1. Let V_1 , V_2 , V_3 and V_4 be four SQL views defined as:

```
create view V1(A1, A2) as
        select T1'.A E1, T1'.B E2
        from T1 T1'
        where T1'.A ≥ 10

create view V2(A) as
        select T2'.C E1
        from V1 V1', T2 T2'
        where V1'.A1 + T2'.C = 0

create view V3(A) as
        select(V1'.A1) E
        from V1 V1'
        where exists
              (select T2'.C E1
               from T2 T2'
               where T2'.C = V1'.A1)

create view V4(A) as
        select V1'.A2 E
        from V1 V1'
        where V1'.A2 = "a"
        group by V1'.A2
        having sum(V1'.A1) > 100;
```

Suppose table T_1 has the attributes A, B while table T_2 has only one attribute C. Consider the following symbolic database instances $d(T_1) = \{\!|\mu_1, \mu_2|\!\}$ and $d(T_2) = \{\!|\mu_3, \mu_4|\!\}$ with: $\mu_1 = \{T_1.A \mapsto x_1, T_1.B \mapsto y_1\}$, $\mu_2 = \{T_1.A \mapsto x_2, T_1.B \mapsto y_2\}$ and $\mu_3 = \{T_2.C \mapsto z_1\}$, $\mu_4 = \{T_2.C \mapsto z_2\}$. Then:

$$\theta(T_1) = \{\!|(true, \mu_1), (true, \mu_2)|\!\}, \quad \theta(T_2) = \{\!|(true, \mu_3), (true, \mu_4)|\!\}$$

$$\begin{array}{rl} \theta(V_1) = & \{\!|\ (x_1 \geq 10, \{V_1.A_1 \mapsto x_1, V_1.A_2 \mapsto y_1\})\ , \\ & \quad (x_2 \geq 10, \{V_1.A_1 \mapsto x_2, V_1.A_2 \mapsto y_2\})\ \ |\!\} \\ \theta(V_2) = & \{\!|\ (x_1 \geq 10 \wedge x_1 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & \quad (x_1 \geq 10 \wedge x_1 + z_2 = 0, \{V_2.A \mapsto z_2\}), \\ & \quad (x_2 \geq 10 \wedge x_2 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\ & \quad (x_2 \geq 10 \wedge x_2 + z_2 = 0, \{V_2.A \mapsto z_2\})\ \ |\!\}\end{array}$$

$$\theta(V_3) = \{\!| \ (x_1 \geq 10 \wedge ((z_1 = x_1) \vee (z_2 = x_1)), \{V_3.A \mapsto x_1\}),$$
$$(x_2 \geq 10 \wedge ((z_1 = x_2) \vee (z_2 = x_2)), \{V_3.A \mapsto x_2\}) \ |\!\}$$
$$\theta(V_4) = \{\!|(\psi_1, \{V_4.A \mapsto y_1\}), (\psi_2, \{V_4.A \mapsto y_1\}), (\psi_3, \{V_4.A \mapsto y_2\})|\!\}$$
$$\psi_1 = (x_1 \geq 10 \wedge x_2 \geq 10) \wedge (y_1 = \text{“}a\text{”} \ \wedge \ y_2 = \text{“}a\text{”} \ \wedge \ y_1 = y_2) \wedge$$
$$(x_1 + x_2 > 100)$$
$$\psi_2 = (x_1 \geq 10) \wedge (y_1 = \text{“}a\text{”}) \wedge$$
$$(\neg(x_2 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \ \wedge \ y_2 = \text{“}a\text{”} \ \wedge \ y_1 = y_2)) \wedge (x_1 > 100)$$
$$\psi_3 = (x_2 \geq 10) \wedge (y_2 > \text{“}a\text{”}) \wedge$$
$$(\neg(x_1 \geq 10) \vee \neg(y_1 = \text{“}a\text{”} \ \wedge \ y_2 = \text{“}a\text{”} \ \wedge \ y_1 = y_2)) \wedge (x_2 > 100)$$

For instance observe that V_4 has an aggregate query with a *group by* over V_1. Since $\theta(V_1)$ contains 2 tuples, $\theta(V_4)$ contains three possible tuples, one for each possible group in V_1: the first group containing the two rows in V_1, the second corresponding only to the first row, and the third possibility a group containing only the second row in V_1.

The following result and its corollary represent the main result of this paper, stating the soundness and completeness of our proposal:

Theorem 1. *Let D be a database schema and d a database instance. Assume that the views and queries in D do not include subqueries. Let R be a relation in D. Then $\mu \in \langle R \rangle$ with cardinality k iff $(\mu, true) \in \theta(R)$ with cardinality k.*

Proof. See Appendix A.

The restriction to queries without subqueries is due to the limitations of ERA. The following corollary contains the idea for generating constraints that will yield the PTCs:

Corollary 1. *Let D be a database schema and d_s a symbolic database instance. Assume that the views and queries in D do not include subqueries. Let R be a relation in D such that $\theta(R) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, and η a substitution satisfying d_s. Then $d_s\eta$ is a PTC for R iff $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

Proof. Straightforward from Theorem 1: $(\bigvee_{i=1}^{n} \psi_i)\eta = true$ iff there is some ψ_i with $1 \leq i \leq n$ such that $\psi_i\eta = true$ iff $(\mu_i\eta) \in \langle R \rangle$ iff $\langle R \rangle \neq \emptyset$.

5 Implementation and Prototype

In this section, we comment on some aspects of our implementation and show a system session with actual results of the test case generator.

Our test case generator is bundled as a component of the Datalog deductive database system DES [15]. The input of the tool consists of:

- A database schema D defined by means of SQL language, i.e., a finite set of tables $\mathcal{T}$, constraints $\mathcal{C}$ and views $\mathcal{V}$, as well as the integrity constraints for the columns (primary and foreign keys).
- A SQL view V for which the test case is to be generated.

DES [15] is implemented in Prolog and includes a SQL parser for queries and views, and a type inference system for SQL views. In this way we benefit from the DES facilities for dealing with SQL and at the same time we can exploit the constraint solving features available in current Prolog implementations. As a first step, we have chosen SICStus Prolog as a suitable platform for our development (although others will be handled in a near future).

As explained in Section 4, we do need constraints that include a mix of conjunctions and disjunctions. We use *reification* to achieve an efficient implementation of these connectives. Thus, we reify every atomic constraint and transform conjunctions and disjunctions of constraints into finite domain constraints of the form $B_1 * \ldots * B_k \geq B_0$, and $B_1 + \ldots + B_k \geq B_0$, respectively. B_0 allows a compact form to state the truth or falsity of these constraints.

Apart from the constraints indicated in Section 4 we also need to consider *domain integrity constraints*, the constraints that restrict the given set of values a table attribute can take. These values are represented by a built-in datatype, e.g., string, integer, and float. On the one hand, types in SQL are *declared* in create table statements. In addition, further domain constraints can be declared, which can be seen as subtype declarations, as the *column constraint* $A > 0$, where A is a table attribute with numeric type. On the other hand, types are *inferred* for views.

Up to now, we support integer and string datatypes by using the finite domain ($\mathcal{FD}$) constraint system available in SICStus Prolog. Although with a few changes this can also be easily mapped to Ciao Prolog, GNU Prolog and SWI-Prolog. Posting our constraints over integers to the underlying $\mathcal{FD}$ constraint solver is straightforward. In the case of string constraints we map each different string constant in the SQL statements to a unique integer, allowing equality and disequality ($\mathcal{FD}$) constraints. This mapping is stored in a dictionary before posting constraints for generating the test cases. Then, string constants are replaced by integer keys in the involved views. Generation and solving of constraints describing the test cases in the integer domain follows. Before displaying the instanced result involving only integers, the string constants are recovered back by looking for these integer keys in the dictionary. If some key values are not in the dictionary they must correspond to new strings. The tool generates new string constants for these values. Our treatment is only valid for equalities and disequalities, and it does not cover other common string operations such as the concatenation or the *LIKE* operator which will require a string constraint solver (see [8] for a discussion on solving string constraints involving the *LIKE* operator).

Our tool allows the user to choose the type of test case to be generated, either PTC, or NTC or both PNTC for any view V previously defined in D. The output is a database instance d of a database schema D such that d is a test case for the given view V with as few entries as possible.

For instance, consider the following session:

```
DES-SQL> CREATE OR REPLACE TABLE t(a INT PRIMARY KEY, b INT);
DES-SQL> CREATE OR REPLACE VIEW u(a1, a2) AS SELECT a, b
         FROM t WHERE a >= 10;
DES-SQL> CREATE OR REPLACE VIEW v(a) AS SELECT a2 FROM u
         WHERE a2 = 88 GROUP BY a2 HAVING SUM(a1) > 0;
```

Then, test cases (both positive and negative) for the view `v` can be obtained via the following command:

```
DES-SQL> /test_case v
Info: Test Case over integers:
[t([[1000,88],[999,1000]])]
```

Here, we get the PNTC `[t([[1000,88],[999,1000]])]`. If it is not possible to find a PNTC, the tool would try to generate a PTC and a NTC separately.

Observe that in practice our system cannot reach completeness, but only weak completeness module the size of the tables of the instance. That is, our system will find a PTC if it is possible to construct it with all the tables containing a number of rows less than an arbitrary number. By default the system starts trying to define PTCs with the number of rows limited to 2. If it is not possible, the number of rows is increased. The process is repeated stopping either when a PTC is found or when an upper bound is reached (by default 10). Both the lower and the upper limits are user configurable.

6 Conclusions and Future Work

We have presented a novel technique for generating finite domain constraints whose solutions correspond to test cases for SQL relations. Similar ideas have been suggested in other works but, to the best of our knowledge, not for views, which corresponds to more realistic applications. We have formally defined the algorithm for producing the constraints, and have proved the soundness and weak completeness of the approach with respect to the operational semantics of Extended Relational Algebra. Another novelty of our approach is that we allow the use of string values in the query definitions. Although constraint systems over other domains, as reals or rationals, are available, we have not used them in our current work. However, they can be straightforwardly implemented. In addition, enumerated types (available in object-oriented SQL extensions) could also be included, following a similar approach to the one taken for strings.

Our setting includes primary and foreign keys, existential subqueries, unions, intersections, and aggregate queries, and can be extended to cover other SQL features not included in this paper. For instance, null values can be considered by defining an extra *null table* T_{null} containing the logic variables that are null, and taking into account this table when evaluating expressions. For instance, a condition $T.A = T'.B$ will be translated into $(T.A = T'.B) \wedge (T.A \notin T_{null}) \wedge (T'.B \notin T_{null})$.

Dealing with recursive queries is more involved. One possibility could be translating the SQL views into a logic language like Prolog, and then use a technique for generating test cases for this language [11]. However, aggregate queries are not easily transformed into Prolog queries, and thus this approach will only be useful for non-aggregate queries.

It is well-known that the problem of finding complete sets of test cases is in general undecidable [1]. Different coverage criteria have been defined (see [1] for a survey) in order to define test cases that are complete at least w.r.t. some desired property. In this work, we have considered a simple criterion for SQL queries, namely the *predicate coverage criterium*. However, it has been shown [19] that other coverage criteria can be reduced to predicate coverage by using suitable query transformations. For instance, if we look for a set of test cases covering every atomic condition in the where clause of a query Q, we could apply our tool to a set of queries, each one containing a where clause containing only one of the atomic conditions occurring in Q.

A SICStus Prolog prototype implementing these ideas has been reported in this paper, which can be downloaded and tested (binaries provided for both Windows and Linux OSs) from http://gpd.sip.ucm.es/yolanda/research.htm. To allow performance comparisons and make the sources for different Prolog platforms available, an immediate work is the port to Ciao, GNU Prolog and SWI-Prolog.

Although test case generation is a time consuming problem, the efficiency of our prototype is reasonable, finding in a few seconds TCs for views with dependence trees of about ten nodes and with a number of rows limited to seven for every table. The main efficiency problem comes from aggregate queries, where the combinatorial problem of selecting the aggregates can be too complex for the solver. To improve this point, even when efficiency of the SICStus constraint solver is acknowledged, there are more powerful solvers in the market. In particular, we plan to test the industrial, more efficient $\mathcal{FD}$ and $\mathcal{R}$ IBM ILOG solvers [13], which allow to handle bigger problems at a faster rate than SICStus solvers. Also, another striking state-of-the-art, free, and open-source $\mathcal{FD}$ solver library to be tested is Gecode [16].

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008)
2. Binnig, C., Kossmann, D., Lo, E.: Towards automatic test database generation. IEEE Data Eng. Bull. 31(1), 28–35 (2008)
3. Cabal, M.J.S., Tuya, J.: Using an SQL coverage measurement for testing database applications. In: Taylor, R.N., Dwyer, M.B. (eds.) SIGSOFT FSE, pp. 253–262. ACM, New York (2004)
4. Codd, E.: Relational Completeness of Data Base Sublanguages. In: Rustin, R. (ed.) Data base Systems. Courant Computer Science Symposia Series 6, Prentice-Hall, Englewood Cliffs (1972)

5. Dayal, U., Goodman, N., Katz, R.H.: An extended relational algebra with control over duplicate elimination. In: PODS 1982: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems, pp. 117–123. ACM, New York (1982)
6. Degrave, F., Schrijvers, T., Vanhoof, W.: Automatic generation of test inputs for mercury, pp. 71–86 (2009)
7. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17(9), 900–910 (1991)
8. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA 2007: Proceedings of the 2007 international symposium on Software testing and analysis, pp. 151–162. ACM, New York (2007)
9. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall PTR, Upper Saddle River (2008)
10. Gogolla, M.: A note on the translation of SQL to tuple calculus. SIGMOD Record 19(1), 18–22 (1990)
11. Gómez-Zamalloa, M., Albert, E., Puebla, G.: On the generation of test data for prolog by partial evaluation. CoRR, abs/0903.2199 (2009)
12. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra: a formal approach to a practical issue. In: 10th International Conference on Data Engineering, pp. 80–88. IEEE, Los Alamitos (1994)
13. ILOG CP 1.4, `http://www.ilog.com/products/cp/`
14. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of SQL queries. ACM Trans. Database Syst. 16(3), 513–534 (1991)
15. Sáenz-Pérez, F.: Datalog educational system. user's manual version 1.7.0. Technical report, Faculty of Computer Science, UCM (November 2009), `http://des.sourceforge.net/`
16. Schulte, C., Lagerkvist, M.Z., Tack, G.: Gecode, `http://www.gecode.org/`
17. SQL, ISO/IEC 9075:1992, third edn. (1992)
18. Suárez-Cabal, M., Tuya, J.: Structural coverage criteria for testing SQL queries. Journal of Universal Computer Science 15(3), 584–619 (2009)
19. Tuya, J., Suárez-Cabal, M.J., de la Riva, C.: Full predicate coverage for testing SQL database queries. Software Testing, Verification and Reliability (2009) (to be published)
20. Zhang, J., Xu, C., Cheung, S.C.: Automatic generation of database instances for white-box testing. In: COMPSAC, pp. 161–165. IEEE Computer Society, Los Alamitos (2001)
21. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29, 366–427 (1997)

A Proof of Theorem 1

In this Appendix we include the proof of our main theoretical result. The theorem establishes a bijective mapping between the rows obtained by applying the ERA semantics to a relation R defined on an schema with instance d and the tuples $(true, \sigma)$ in $\theta(R)$ (see Definition 6). Before proving the result we introduce an auxiliary Lemma:

Lemma 1. *Let D be a database schema and d a database instance, $R_1, \ldots, R_m$ relations verifying Theorem 1, $B_1, \ldots, B_m$ attribute names, and R an expression*

in ERA defined as $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$. *Let* P *be a multiset defined as*

$$P = \{\!|(\psi, \mu) \mid (\psi_1, \nu_1), \ldots, (\psi_m, \nu_m) \in (\theta(R_1) \times \ldots \times \theta(R_m)) \\ \psi = \psi_1 \wedge \ldots \wedge \psi_m,\ \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m} |\!\}$$

Then $\mu \in R$ *with cardinality* k *iff* $(true, \mu) \in P$ *with cardinality* k.

Proof. $(true, \mu) \in P$ with cardinality k iff there are pairs $(\psi_i, \nu_i) \in \theta(R_i)$ with cardinality c_i for $i = 1 \ldots m$ such that $k = c_1 \times \ldots \times c_m$ and $\mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}$. From the conditions of P we have that $\psi = true$ iff $\psi_i = true$ for $i = 1 \ldots m$. By hypothesis $(true, \nu_i) \in \theta(R_i)$ with cardinality c_i iff $\nu_i \in R_i$ with cardinality c_i for $i = 1 \ldots m$, iff $({\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}) \in (\rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m))$ with cardinality $c_1 \times \ldots \times c_m$, i.e., $\mu \in R$ with cardinality k.

Next we prove the Theorem by induction on the number of nodes of the dependence tree for R. If this number is 1 (basis) then R is a table T, $\langle T \rangle = d(T)$, and the result is an easy consequence of Definition 6 item 1. If the dependence tree contains at least two nodes (inductive case) R cannot be a table. We distinguish cases depending on the form of R:

- R *aggregate query.* Then Q is of the form

$$\mathsf{Q} = \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m \\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ A'_1, \ldots, A'_k\ \mathsf{having}\ C_h$$

Then $\langle Q \rangle = \Pi_{e'_1 \to E_1, \ldots, e'_n \to E_n}(\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$, with $R = \rho_{B_1}(R_1) \times \ldots \times \rho_{B_m}(R_m)$, $L = \{A'_1, \ldots, A'_k, u_1 \mapsto U_1, \ldots, u_l \mapsto U_l\}$, u_i the aggregate expressions occurring either in the select or in the having clauses for $i = 1 \ldots l$, U_i new attribute names for $i = 1 \ldots l$, e'_j the result of replacing each occurrence of u_i in e_j, $1 \leq j \leq n$ by U_i and analogously for C'_h. From Definition 6, item 5 we have

$$\theta(Q) = \{\!|(\bigwedge(\Pi_1(A)) \wedge aggregate(Q, A),\ s_Q(\Pi_2(A))) \mid A \subseteq P |\!\}$$

Let $\mu \in \langle Q \rangle$ with cardinality k. Then there are rows $\nu_1, \ldots, \nu_r$ such that μ is of the form $\mu = (\nu_i)\{E_1 \mapsto e'_1, \ldots, E_n \mapsto e'_n\}$, $1 \leq i \leq r$ with $\nu_i \in (\sigma_{C'_h}(\gamma_L(\sigma_{C_w}(R))))$, with cardinality c_i for $i = 1 \ldots r$ and $k = c_1 + \ldots + c_r$. From the definition of γ we have that the c_i occurrences of ν_i for $i = 1 \ldots r$ correspond to the existence of c_i maximal aggregates $S_i^j \subseteq \sigma_{C_w}(R)$, $j = 1 \ldots c_i$.

Observe for every $\eta \in S_i^j$ we have that the cardinality of η in S_i^j and in R is the same because S_i^j is maximal. Then from Lemma 1 we have that the set $A_i^j = \{\!|(true, \eta) \mid \eta \in S_i^j|\!\}$ verifies $A_i^j \subseteq P$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$. Then it is immediate that $\bigwedge(\Pi_1(A_i^j)) = true$ and that $s_Q(\Pi_2(A_i^j)) = s_Q(S_i) = \{E_1 \mapsto ((e_1)S_i^j), \ldots, E_n \mapsto ((e_n)S_i^j)\} = (\nu_i)\{E_1 \mapsto e'_1, \ldots, E_n \mapsto e'_n\} = \mu$. Then we have that $(true \wedge aggregate(Q, A_i^j),\ \mu) \in \theta(Q)$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$. It remains to check that $aggregate(Q, A_i^j) = true$, i.e., that

- $group(Q, \Pi_2(A_i^j)) = true$. $\Pi_2(A_i^j) = S_i^j$ and the definition of group requires that all the rows in S_i^j verify the where condition and that every row takes

the same values for the grouping attributes. The first requirement is a consequence of $S_i^j \subseteq \sigma_{C_w}(R)$, while the second one holds because we are assuming that the multiset S_i^j was selected has a valid group by the operator γ.
- $maximal(Q, A_i^j) = true$. The auxiliary definition *maximal* indicates that no other element of the form $(true, \mu')$ from P can be included in S_i^j verifying that we still have the same values for the grouping attributes and μ' verifying the where condition. This is true because if there were such $(true, \mu') \in P - A_i^j$, then by Lemma 1 μ' will be in R and S_i^j will not be maximal in $\sigma_{C_w}(R)$ as required by γ.
- $\varphi(C_h, \Pi_2(A_i^j)) = true$. Observe that $\varphi(C_h, \Pi_2(A_i^j)) = \varphi(C_h, S_i^j)$, and that in the absence of subqueries φ only checks that the S_i^j verify the having condition C_h, which is true because ν_i verifies C_h'.

Then we have $(true,\ \mu) \in \theta(Q)$ for $i = 1 \ldots r$, $j = 1 \ldots c_i$ and thus $(true,\ \mu) \in \theta(Q)$ with cardinality k .
The converse result, i.e., assuming $(true, \mu) \in \theta(Q)$ with cardinality k and proving that then $\mu \in\in \langle Q \rangle$ with cardinality k, is analogous.

- R *basic query.* Similar to the previous case.
- $R = V_1$ *union* V_2. Then $\langle R \rangle = \langle V_1 \rangle \cup \langle V_2 \rangle$, $\theta(R) = \theta(V_1) \cup \theta(V_2)$ and the result follows by induction hypothesis since V_1, V_2 are children of R in its dependence tree.
- $R = V_1$ *intersection* V_2 . Then

$$\begin{aligned} \langle R \rangle &= \langle V_1 \rangle \cap \langle V_2 \rangle \\ \theta(R) &= \{\!|(\psi_1 \wedge \psi_2 \wedge \nu_1 = \nu_2,\ \nu_1) \mid \ (\psi_1, \nu_1) \in \theta(V_1), (\psi_2, \nu_2) \in \theta(V_2)|\!\} \end{aligned}$$

Then $\mu \in \langle R \rangle$ with cardinality k iff $\mu \in \langle V_1 \rangle$ and $\mu \in \langle V_2 \rangle$ with cardinalities k_1, k_2 respectively and $k = min(k_1, k_2)$. By the induction hypothesis $(true, \mu) \in \theta(V_1)$ with cardinality k_1, $(true, \mu) \in \theta(V_2)$ with cardinality k_2 and this happens iff $(true, \mu) \in \theta(R)$.
- R *is a view* V *with associated query* Q. Then $\langle V \rangle = \Pi_{E_1 \to V.A_1, \ldots, E_n \to V.A_n} \langle Q \rangle$ and $\theta(V) = \theta(Q)\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$ with $E_1, \ldots, E_n$ the attribute names of the select clause in Q. We have proved above that $\mu \in \langle Q \rangle$ iff $(true, \mu) \in \theta(Q)$ with the same cardinality. Now observe that for every $\mu \in \langle Q \rangle$ applying the projection $\Pi_{E_1 \to A_1, \ldots, E_n \to A_n}$ produces a renaming of its domain $E_1, \ldots, E_n$ to $A_1, \ldots A_n$, and that this is the same as $\mu\{V.A_1 \mapsto E_1, \ldots, V.A_n \mapsto E_n\}$.

Algorithmic Debugging of SQL Views*

Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid, Spain
{rafa,fernan}@sip.ucm.es, ygarciar@fdi.ucm.es

Abstract. We present a general framework for debugging systems of correlated SQL views. The debugger locates an erroneous view by navigating a suitable computation tree. This tree contains the computed answer associated with every intermediate relation, asking the user whether this answer is expected or not. The correctness and completeness of the technique is proven formally, using a general definition of SQL operational semantics. The theoretical ideas have been implemented in an available tool which includes the possibility of employing trusted specifications for reducing the number of questions asked to the user.

1 Introduction

SQL [12] is the *de facto* standard language for querying and updating relational databases. Its declarative nature and its high-abstraction level allows the user to easily define complex operations that could require hundreds of lines programmed in a general purpose language. In the case of relational queries, the language introduces the possibility of querying the database directly using a *select* statement. However, in realistic applications, queries can become too complex to be coded in a single statement and are generally defined using *views*. Views can be considered in essence as virtual tables. They are defined by a *select* statement that can rely on the database tables as well as in other previously defined views. Thus, views become the basic components of SQL queries.

As in other programming paradigms, views can have bugs which produce unexpected results. However, we cannot infer that a view is buggy only because it returns an unexpected result. Maybe it is correct but receives erroneous input data from the other views or tables it depends on. There are very few tools for helping the user to detect the cause of these errors; so, debugging becomes a labor-intensive and time-consuming task in the case of queries defined by means of several intermediate views. The main reason for this lack of tools is that the usual *trace* debuggers used in other paradigms are not available here due to the high abstraction level of the language. A *select* statement is internally translated into a sequence of low level operations that constitute the *execution plan* of the

* This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

E. Clarke, I. Virbitskaite, and A. Voronkov (Eds.): PSI 2011, LNCS 7162, pp. 77–85, 2012.

query. Relating these operations to the original query is very hard, and debugging the execution plan step by step will be of little help. In this paper, we propose a theoretical framework for debugging SQL views based on *declarative debugging*, also known as *algorithmic debugging* [11]. This technique has been employed successfully in (constraint) logic programming [11], functional programming [9], functional-logic programming [2], and in deductive database languages [1]. The overall idea of declarative debugging [7] can be explained briefly as follows:

- The process starts with an initial error symptom, which in our case corresponds to the unexpected result of a user-defined view.
- The debugger automatically builds a tree representing the computation. Each node of the tree corresponds to an intermediate computation with its result. The children of a node are those nodes obtained from the subcomputations needed for obtaining the parent result. In our case, nodes will represent the computation of a relation R together with its answer. Children correspond to the computation of views and tables occurring in R if it is a view.
- The tree is *navigated*. An external oracle, usually the user, compares the computed result in each node with the *intended* interpretation of the associated relation. When a node contains the expected result, it is marked as *valid*, otherwise it is marked as *nonvalid*.
- The navigation phase ends when a nonvalid node with valid children is found. Such node is called a *buggy node*, and corresponds to an incorrect piece of code. In our case, the debugger will end pointing out either an erroneously defined view, or a table containing a nonvalid instance.

Our goal is to present a declarative debugging framework for SQL views, showing that it can be implemented in a realistic, scalable debugging tool.

We have implemented our debugging proposal in the Datalog Educational System (DES [10]), which makes it possible for Datalog and SQL to coexist as query languages for the same database. The current implementation of our proposal for debugging SQL views and instructions to use it can be downloaded from `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/Des`.

2 SQL Semantics

The first formal semantics for relational databases based on the concept of set (e.g., relational algebra, tuple calculus [3]) were incomplete with respect to the treatment of non-relational features such as repeated rows and aggregates, which are part of practical languages such as SQL. Therefore, other semantics, most of them based on multisets [4], have been proposed. In our framework we will use the *Extended Relational Algebra* [6,5]. We start by defining the concepts of database schemas and instances.

A *table schema* is of the form $T(A_1, \ldots, A_n)$, with T being the table name and A_i the attribute names for $i = 1 \ldots n$. We will refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type (*integer*, *string*, ...). An *instance* of a table schema $T(A_1, \ldots, A_n)$ is determined by its particular

Owner

id	name
1	Mark Costas
2	Helen Kaye
3	Robin Scott
4	Tom Cohen

Pet

code	name	species
100	Wilma	dog
101	Kitty	cat
102	Wilma	cat
103	Lucky	dog
104	Rocky	dog
105	Oreo	cat
106	Cecile	turtle
107	Chelsea	dog

PetOwner

id	code
1	100
1	101
2	102
2	103
3	104
3	105
4	106
4	107

Fig. 1. All Pets Club database instance

rows. Each row contains values of the correct type for each attribute in the table schema. *Views* can be thought of as new tables created dynamically from existing ones by using a SQL query. The general syntax of a SQL view is: create view $\mathsf{V}(A_1, \ldots, A_n)$ as Q, with Q a SQL *select* statement, and $V.A_1, \ldots, V.A_n$ the names of the view attributes. In general, we will use the name *relation* to refer to either a table or a view (observe that the mathematical concept of relation is defined over sets, but in our setting we define relations among multisets). A *database schema* D is a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of table schemas and $\mathcal{V}$ a finite set of view definitions. Although database schemas also include constraints such as primary keys, they are not relevant to our setting.

A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$. To represent the instance of a table T in d we will use the notation $d(T)$.

The syntax of SQL queries can be found in [12]. The *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in its query. The next example defines a particular database schema that will be used in the rest of the paper as a running example.

Example 1. The *Dog and Cat Club* annual dinner is going to take place in a few weeks, and the organizing committee is preparing the guest list. Each year they browse the database of the *All Pets Club* looking for people that own at least one cat and one dog. Owners come to the dinner with all their cats and dogs. However, two additional constraints have been introduced this year:

- People owning more than 5 animals are not allowed (the dinner would become too noisy).
- No animals sharing the same name are allowed at the party. This means that if two different people have a cat or dog sharing the same name neither of them will be invited. This severe restriction follows after last year's incident, when someone cried *Tiger* and dozens of pets started running without control.

Figure 1 shows the *All Pets Club* database instance. It consists of three tables: *Owner*, *Pet*, and *PetOwner* which relates each owner with its pets. Primary keys are shown underlined. Figure 2 contains the views for selecting the

```
create or replace view AnimalOwner(id,aname,species) as
 select O.id, P.name, P.species
 from Owner O, Pet P, PetOwner PO
 where O.id = PO.id and P.code = PO.code;

create or replace view LessThan6(id) as
 select id from AnimalOwner
 where species='cat' or species='dog'
 group by id having count(*)<6;

create or replace view CatsAndDogsOwner(id,aname) as
 select AO1.id,AO1.aname
 from AnimalOwner AO1, AnimalOwner AO2
 where AO1.id = AO2.id and AO1.species='dog'
       and AO2.species='cat';

create or replace view NoCommonName(id) as
 select id from CatsAndDogsOwner
 except
 select B.id from CatsAndDogsOwner A, CatsAndDogsOwner B
                  where A.id <> B.id
                  and A.aname = B.aname;

create or replace view Guest(id,name) as
 select id, name
 from Owner natural inner join NoCommonName
            natural inner join LessThan6;
```

Fig. 2. Views for selecting dinner guests

dinner guests. The first view is *AnimalOwner*, which obtains all the tuples (*id*,*aname*,*species*) such that *id* is the owner of an animal of name *aname* of species *species*. *LessThan6* returns the identifiers of the owners with less than six cats and dogs. *CatsAndDogsOwner* returns pairs (*id*,*aname*) where *id* is the identifier of the owner of either a cat or a dog with name *aname*, such that *id* owns both cats and dogs. *NoCommonName* is defined by removing owners sharing pet names from the total list of cats and dog owners. Finally, the main view is *Guest*, which selects those owners that share no pet name with another owner (view *NoCommonName*) and that have less than six cats and dogs (view *LessThan6*). However, these views contain a bug that will become apparent in the next sections.

The Extended Relational Algebra (ERA from now on) [6] is an operational SQL Semantics allowing aggregates, views and most of the common features of SQL queries. The main characteristics of ERA are:

1. The table instances and the result of evaluating queries/views are multisets, (it is also possible to consider *lists* instead of multisets if we consider relevant the order among rows in a query result).
2. ERA expressions define new relations by combining previously defined relations using multiset operators (see [5] for a formal definition of each operator).
3. We use Φ_R to represent a SQL query or view R as an ERA expression, as explained in [5]. Since a query/view depends on previously defined relations, sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on $R_1, \ldots, R_n$. If $M_1, \ldots, M_n$ are multisets we use the notation $\Phi_R(M_1, \ldots, M_n)$ to indicate that the expression Φ_R is evaluated after substituting $R_1, \ldots, R_n$ by $M_1, \ldots, M_n$.
4. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table.
5. The computed answer of Φ_R with respect to some schema instance d will be denoted by $\| \Phi_R \|_d$, where
   - If R is a database table, $\| \Phi_R \|_d = d(R)$.
   - If R is a database view or a query and $R_1, \ldots, R_n$ the relations defined in R then $\| \Phi_R \|_d = \Phi_R(\| \Phi_{R_1} \|_d, \ldots, \| \Phi_{R_n} \|_d)$.

Observe that $\| \Phi_R \|_d$ is well defined since mutually recursive view definitions are not allowed[1]. We assume that $\| \Phi_R \|_d$ actually corresponds to the answer obtained by a correct SQL implementation, i.e., that the available SQL systems implement ERA. In fact our proposal is valid for any semantics that associate a formula Φ_R to any relation R and allow the recursive definition of computed answer of item 5 above.

3 Declarative Debugging Framework

In this section, we assume a set of SQL views $\mathcal{V} = \{V_1, \ldots, V_n\}$ such that for some $1 \leq i \leq n$, and for some database instance d, V_i has produced an unexpected result in some SQL system. We also assume that this SQL system implements the ERA operational semantics of previous section. Our debugging technique will be based on the comparison between the answers by a SQL system implementing the ERA semantics, and the oracle intended answers. Next, we define the concept of intended answer for schema relations.

Definition 1. Intended Answers for Schema Relations
Let D be a database schema, d an instance of D, and R a relation defined in D. The intended answer for R w.r.t. d, is a multiset denoted as $\mathcal{I}(R, d)$ containing the answer that the user expects for the query select * from R; *in the instance d.*

The intended answer depends not only on the view semantics but also on the contents of the tables in the instance d. This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging. Figure 3

[1] Recursive views are allowed in the SQL:1999 standard but they are not supported in all the systems, and they are not considered here.

AnimalOwner

id	aname	species
1	Wilma	dog
1	Kitty	cat
2	Wilma	cat
2	Lucky	dog
3	Rocky	dog
3	Oreo	cat
4	Cecile	turtle
4	Chelsea	dog

LessThan6

id
1
2
3
4

CatsAndDogsOwner

id	aname
1	Wilma
1	Kitty
2	Wilma
2	Lucky
3	Oreo
3	Rocky

NoCommonName

id
3

Guest

id	name
3	Robin Scott

Fig. 3. Intended answer for the views in Example 1

contains the intended answer for each view defined in Figure 2. For instance, it is expected that *AnimalOwner* will identify each owner *id* with the names and species of his pets. It is also expected than *LessThan6* will contain the *id* of all four owners, since all of them have less than six cats and dogs. The intended answer for view *CatsAndDogsOwner* contains the *id* and *name* attributes of those entries in *AnimalOwner* corresponding to owners with at least one dog and one cat, and removing pets different from cats and dogs. View *NoCommonName* is expected to contain only one row for owner with identifier 3. The reason is that both owners 1 and 2 share a pet name (*Wilma*). Finally, the only expected *Guest* will be the owner with identifier 3, *Robin Scott*. If now we try the query `select * from Guest`; in a SQL system, we obtain a computed answer representing the multiset $\{\!|(1, Mark\ Costas), (2, Helen\ Kaye), (3, Robin\ Scott)|\!\}$. This computed answer is different from the intended answer for *Guest*, and indicates that there is some error. However, we cannot ensure that the error is in the query for *Guest*, because the error can come from any of the relations in its *from* clause. And also from the relations used by these relations, and so on. In order to define the key concept of erroneous relation it will be useful to define the auxiliary concept of *inferred answer*.

Definition 2. Inferred Answers
Let D be a database schema, d an instance of D, and R a relation in D. The inferred answer for R, with respect to d, $\mathcal{E}(R, d)$, is defined as

1. *If R is a table, $\mathcal{E}(R, d) = d(R)$.*
2. *If R is a view, $\mathcal{E}(R, d) = \Phi_R(\mathcal{I}(R_1, d), \ldots, \mathcal{I}(R_n, d))$ with $R_1, \ldots, R_n$ the relations occurring in R.*

Thus, in the case of tables, the inferred answer is just its table instance. In the case of a view V, the inferred answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of V contain the intended answers. For instance, consider Example 1 and the instance d of Figure 1. Assume that all the tables contain the intended answers, i.e., for every table T, $\mathcal{I}(T, d) = d(T)$. Then the inferred answer for view *CatsAndDogsOwner* is the same as its computed answer

$\| \, CatsAndDogsOwner \, \|_d$:

$$\mathcal{E}(CatsAndDogsOwner, d) = \Phi_{CatsAndDogsOwner}(\mathcal{I}(AnimalOwner, d)) = \{\| \, (1, Wilma), (2, Lucky), (3, Rocky) \|\}$$

However, this result is different from the intended answer for this view (Fig. 3). A discrepancy between $\mathcal{I}(R, d)$ and $\mathcal{E}(R, d)$ shows that R does not compute its intended answer, even assuming that all the relations it depends on contain their intended answers. Such relation is erroneous:

Definition 3. Erroneous Relation
Let D be a database schema, d an instance of D, an R a relation defined in D. We say that R is an erroneous relation when $\mathcal{I}(R, d) \neq \mathcal{E}(R, d)$.

Definition 3 clarifies the fundamental concept of erroneous relation. However, it cannot be used directly for defining a practical debugging tool, because in order to point out a view V as erroneous, it would require comparing $\mathcal{I}(V, d)$ and $\mathcal{E}(V, d)$. By Definition 2, to obtain $\mathcal{E}(V, d)$, the tool will need the intended answer $\mathcal{I}(R, d)$ for every R occurring in the query defining V. But $\mathcal{I}(R, d)$ is only known by the user, who should provide this information during the debugging process. Obviously, a technique requiring such amount of information would be rejected by most of the users. Instead, we will require from the oracle only to answer questions of the form *'Is the computed answer (...) the intended answer for view V?'* Thus, the declarative debugger will compare the computed answer –obtained from the SQL system– and the intended answer –known by the oracle– In a first phase, the debugger builds a *computation tree* for the main view. The definition of this structure is the following:

Definition 4. Computation Trees
Let D be a database schema with views $\mathcal{V}$, d an instance of D, and R a relation defined in D. The computation tree $CT(R, d)$ associated with R w.r.t. d is defined as follows:

- *The root of $CT(R, d)$ is $(R \mapsto\| \, \Phi_R \, \|_d)$.*
- *For any node $N = (R' \mapsto\| \, \Phi_{R'} \, \|_d)$ in $CT(R, d)$:*
  - *If R' is a table, then N has no children.*
  - *If R' is a view, the children of N will correspond to the CTs for the relations occurring in the query associated with R'.*

In practice, the nodes in the computation tree correspond to the syntactic dependency tree of the main SQL view, with the children at each node corresponding to the relations occurring in the definition of the corresponding view. After building the computation tree, the debugger will *navigate* the tree, asking the oracle about the validity of some nodes:

Definition 5. Valid, Nonvalid and Buggy Nodes
Let $T = CT(R, d)$ be a computation tree, and $N = (R' \mapsto\| \, \Phi_{R'} \, \|_d)$ a node in T. We say that N is valid *when $\| \, \Phi_{R'} \, \|_d = \mathcal{I}(R', d)$,* nonvalid *when $\| \, \Phi_{R'} \, \|_d \neq \mathcal{I}(R', d)$, and* buggy *when N is nonvalid and all its children in T are valid.*

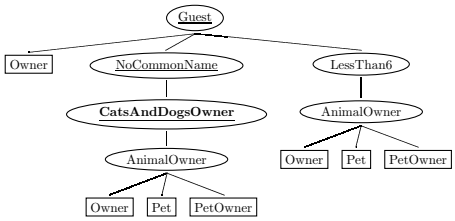


Fig. 4. Computation tree for view Guest

The goal of the debugger will be to locate buggy nodes. The next theorem shows that a computation tree with a nonvalid root always contains a buggy node, and that every buggy node corresponds to an erroneous relation.

Theorem 1. *Let d be an instance of a database schema D, V a view defined in D, and T a computation tree for V w.r.t. d. If the root of T is nonvalid then:*

- *Completeness. T contains a buggy node.*
- *Soundness. Every buggy node in T corresponds to an erroneous relation.*

The debugging process starts when the user finds a view V returning an unexpected result. The debugger builds the computation tree for V, which has a nonvalid root as required by the theorem. Figure 4 shows the computation tree for our running example (after removing the repeated children). Nonvalid nodes are underlined, and the only buggy node (in bold face) corresponds to view *CatsAndDogsOwner*.

4 Conclusions

In this paper, we propose using algorithmic debugging for finding errors in systems involving several SQL views. To the best of our knowledge, it is the first time that a debugging tool of these characteristics has been proposed. The debugger is based on the navigation of a suitable computation tree corresponding to some view returning some unexpected result. The validity of the nodes in the tree is determined by an external oracle, which can be either the user, or a trusted specification containing a correct version of part of the views in the system. The debugger ends when a buggy node, i.e., a nonvalid node with valid children, is found. We prove formally that every buggy node corresponds to an erroneous relation, and that every computation tree with a nonvalid root contains some buggy node. Although the results are established in the context of

the Extended Relational Algebra, they can be easily extended to other possible SQL semantics, such as the Extended Three Valued Predicate Calculus [8].

The technique is easy to implement, obtaining an efficient, platform-independent and scalable debugger without much effort. The tool is very intuitive, because it automates what usually is done when an unexpected answer is found in a system with several views: check the relations in the *from* clause, and if some of them return an unexpected answer, repeat the process. Automating this process is of great help, especially when the tool includes additional features as advanced navigation strategies, or the possibility of using trusted specifications. We have successfully implemented our proposal in the existing, widely-used DES system.

As future work, we plan the development of a graphical interface, which can be very helpful for inspecting the computation tree providing information about the node validity. It will also be useful to consider individual wrong tuples in the unexpected results and study its provenance[13], for a fine-grain error detection.

References

1. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Theoretical Framework for the Declarative Debugging of Datalog Programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)
2. Caballero, R., López-Fraguas, F., Rodríguez-Artalejo, M.: Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 170–184. Springer, Heidelberg (2001)
3. Codd, E.: Relational Completeness of Data Base Sublanguages. In: Rustin (ed.) Data Base Systems. Courant Computer Science Symposia Series, vol. 6. Prentice-Hall, Englewood Cliffs (1972)
4. Dayal, U., Goodman, N., Katz, R.H.: An extended relational algebra with control over duplicate elimination. In: PODS 1982: Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp. 117–123. ACM, New York (1982)
5. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall PTR, Upper Saddle River (2008)
6. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra: a formal approach to a practical issue. In: 10th International Conference on Data Engineering, pp. 80–88. IEEE (1994)
7. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
8. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of SQL queries. ACM Trans. Database Syst. 16(3), 513–534 (1991)
9. Nilsson, H.: How to look busy while being lazy as ever: The implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
10. Sáenz-Pérez, F.: DES: A Deductive Database System. In: Spanish Conference on Programming and Computer Languages (September 2010) (in Press)
11. Shapiro, E.: Algorithmic Program Debugging. In: ACM Distiguished Dissertation. MIT Press (1982)
12. SQL, ISO/IEC 9075:1992, third edition (1992)
13. Vansummeren, S., Cheney, J.: Recording provenance for sql queries and updates. IEEE Data Eng. Bull. 30(4), 29–37 (2007)

Integrating XPath with the Functional-Logic Language Toy

Rafael Caballero[1], Yolanda García-Ruiz[1], and Fernando Sáenz-Pérez[2,⋆]

[1] Departamento de Sistemas Informáticos y Computación,
[2] Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain

Abstract. This paper presents a programming framework for incorporating XPath queries into the functional-logic language $\mathcal{TOY}$. The proposal exploits the language characteristics, including non-determinism, logic variables, and higher-order functions and patterns. Our setting covers a wide range of standard XPath axes and tests. In particular reverse axes are implemented thanks to the double nature of XPath queries, which are both higher-order functions and data terms in our setting. The combination of these two different worlds, the functional-logic paradigm and the XML query language XPath, is very enriching for both of them. From the point of view of functional-logic programming, the language is now able to deal with XML documents in a very simple way. From the point of view of XPath, our approach presents several nice properties as the generation of XML test-cases for XPath queries, which can be useful for finding bugs in erroneous queries.

Keywords: Functional-Logic Programming, Non-Deterministic Functions, XPath Queries, Higher-Order Patterns.

1 Introduction

In the last few years the Extensible Markup Language XML [12] has become the *de facto* standard for the exchange of different types of data. Thus, querying XML documents from different languages as become a convenient feature. XQuery [14,15] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [13], a domain-specific language that has become part of general-purpose languages. Although less expressive than XQuery, the simplicity of XPath makes it a perfect tool for many types of queries. In this paper, we address the task of incorporating XPath into the functional-logic system $\mathcal{TOY}$ [8]. The usual approach for integrating XPath in an existing programming language first represents the XPath query by means of some suitable data type, and then employs some evaluator which takes the XPath query and the XML document as inputs, and produces the desired result

⋆ This work has been supported by the Spanish projects TIN2008-06622-C03-01, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502.

R. Rocha and J. Launchbury (Eds.): PADL 2011, LNCS 6539, pp. 145–159, 2011.

as output. However, in functional and functional-logic languages, a different approach is possible: XPath queries can be represented by higher-order functions connected by higher-order combinators. Using this approach, an XPath query becomes at the same time implementation (code) and representation (data term). In this paper we follow this idea, which has been used in the past, for instance for defining parsers in functional and functional-logic languages [3,7].

The specific characteristics of functional-logic languages match perfectly the nature of XPath queries:

- *Non-deterministic functions* are used to nicely represent the evaluation of an XPath query, which consists of fragments of the input XML document.
- *Logic variables* are employed for instance when obtaining the contents of XPath text nodes. Also, they play an important role when defining XML test-cases for XPath queries, one of the most appealing features of our setting.
- By defining rules with *higher-order patterns* XPath queries become truly first-class citizens in our setting. This allows us to define the transformation for introducing reverse axes as `parent` or checking that the query is constructed using XPath standard components.

The rest of the paper is organized as follows. Section 2.1 briefly introduces the functional-language $\mathcal{TOY}$ and the XPath subset considered in this work. Section 3 defines the basic components of XPath queries in $\mathcal{TOY}$. Section 4 shows how XML test-cases for XPath queries can be readily generated, while Section 5 takes advantage of higher-order patterns for introducing some improvements in our framework. Finally, Section 6 presents some conclusions.

2 Preliminaries

Next we introduce briefly the functional-logic language $\mathcal{TOY}$ and the subset of XPath that we intend to integrate with $\mathcal{TOY}$.

2.1 The Functional-Logic Language $\mathcal{TOY}$

All the examples in this paper are written in the concrete syntax of the lazy functional-logic language $\mathcal{TOY}$ [8], but most of the code can be easily adapted to other similar languages as Curry [5]. We start explaining a possible representation of basic XML documents in $\mathcal{TOY}$. A $\mathcal{TOY}$ program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$, as illustrated next:

```
data node      = txt    string
               | comment string
               | tag     string [attribute] [node]
data attribute = att     string string
type xml       = node
```

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
tag "root" [att "version" "1.0"] [
tag "food" [] [
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "watermelon"],
     tag "price" [] [txt "32"]
  ],
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "oranges"],
     tag "variety" [] [txt "navel"],
     tag "price" [] [txt "74"]
  ],
  tag "item" [att "type" "vegetable"] [
     tag "name" [] [txt "onions"],
     tag "price" [] [txt "55"]
  ],
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "strawberries"],
     tag "variety" [] [txt "alpine"],
     tag "price" [] [txt "210"]
  ]
 ]]
```

Fig. 1. XML example (left) and its representation in $\mathcal{TOY}$ (right)

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Notice that in this paper we do not consider the adequacy of the document to its underlying *Schema* definition [11]. This task has been addressed in functional programming defining regular expression types [10]. However, in this work we assume well-formed input XML documents.

The $\mathcal{TOY}$ primitive `load_xml_file` loads an XML file returning its representation as a value of type `node`. Figure 1 shows an example of XML file and its representation in $\mathcal{TOY}$.

Each rule for a function f has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \quad \text{where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns. The overall idea is that a function call $(f\ e_1 \ldots e_n)$ returns an instance $r\theta$ of r, if:

- Each e_i can be reduced to some pattern a_i, $i = 1 \ldots n$, such that $(f\ t_1 \ldots t_n)$ and $(f\ a_1 \ldots a_n)$ are unifiable with most general unifier θ, and
- $u_i\theta$ can be reduced to pattern $s_i\theta$ for each $i = 1 \ldots m$.

In $\mathcal{TOY}$, variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase. Infix operators are also allowed as particular case of program functions. Consider for instance the definitions:

```
infixr 30 /\                infixr 30 \/            infixr 45 ?
false /\ X = false          true \/ X = true        X ? _Y = X
true /\ X = X               false \/ X = X          _X ? Y = Y
```

The /\ and \/ operators represent the standard conjunction and disjunction, respectively, while `?` represents the non-deterministic choice. For instance the infix declaration `infixr 45 ?` indicates that `?` is an infix operator that associates to the right (the r in `infixr`) and that its priority is 35. The priority is used to assume precedences in the case of expressions involving different operators. Computations in $\mathcal{TOY}$ start when the user inputs some goal as

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

This goal asks $\mathcal{TOY}$ for values of the logical variable R that make true the (strict) equality `1 ? 2 ? 3 ? 4 == R`. This goal yields four different answers {`R` $\mapsto$ `1` }, {`R` $\mapsto$ `2` }, {`R` $\mapsto$ `3` }, and {`R` $\mapsto$ `4` }. The next function extends the choice operator to lists: `member [X|Xs] = X ? member Xs`. For instance, the goal `member [1,2,3,4] == R` has the same four answers that were obtained by trying `1 ? 2 ? 3 ? 4 == R`. $\mathcal{TOY}$ is a *typed* language. Types do not need to be annotated explicitly by the user, they are inferred by the system, which rejects ill-typed expressions. However, function type declarations can also be made explicit by the user, which improves the clarity of the program and helps to detect some bugs at compile time. For instance, a function type declaration is: `member :: [A] -> A` which indicates that `member` takes a list of elements of type A, and returns a value which must be also of type A. As usual in functional programming languages, $\mathcal{TOY}$ allows partial applications in expressions and higher order parameters like `apply F X = F X`.

A particularity of $\mathcal{TOY}$ is that partial applications with pattern parameters are also valid patterns. They are called *higher-order patterns*. For instance, a program rule like `foo (apply member) = true` is valid, although `foo (apply member []) = true` is not because `apply member []` is a reducible expression and not a valid pattern. Higher-order variables and patterns play an important role in our setting. Functional-logic programming share with logic programming the possibility of using logic variables as parameters. For instance, the goal `member L == 3` asks for lists containing the value `3`. The first solution is `L -> [ 3 | _A ]`, which indicates that `L` can be a list starting by `3` and followed by any list (represented by the anonymous variable `_A`). The second answer is `L -> [ _A, 3 | _B ]`, indicating that `3` can be the second element of the list as

well. In this way a (potentially) infinite number of answers can be obtained. The possibility of generating values for the parameters is employed for generating test-cases in Section 4.

2.2 The XML Query Language XPath

XPath is a typed functional language. We consider XPath queries of the form (a complete description of XPath 2.0 can be found at [13]):

XPath = `doc`(file) `/` *Relative*
Relative = $Step_1$ `/` ... `/` $Step_n$ | *Relative* `|` *Relative*
Step = *Axis* :: *Test* | *Axis* :: *Test*`[`*XPath*`]`
Axis = `self` | *ForwardAxis* | *ReverseAxis*
ForwardAxis = `child` | `descendant` | `descendant-or-self` | ...
ReverseAxis = `parent` | `ancestor` | `ancestor-or-self` | ...
Test = `node()` | *name* | `text()` | `comment()` | *

The grammar above specifies a subset of the XPath language, enough for representing easily most XPath queries. There are other axes that can be used in XPath, as `following-sibling`, but according to [15], implementations are not required to support them. *Absolute* XPath location paths start with `doc`(*file*), which loads the XML *file*, and sets the context node to the root, followed by a relative location path. A relative location path can be either a sequence of steps or two relative location paths combined by the disjunction operator `|`. Each step takes as starting node the context node, and it is composed by an *axis* that changes the context node, and by a test that returns only those nodes satisfying the test. Tests can be *kind tests* as `comment()` which holds for comment nodes, or *name tests* which check the name of the node. A special kind test is `*` which holds for *element* nodes. For instance, the query:

```
doc("food.xml")/child::food/
               child::item[child::name/child::text()="onions"]/
               child::price/child::text()
```

returns the price of onions in file `"food.xml"`. Assuming the XML document of Figure 1, this query returns in a XQuery/XPath system the value `"55"`. Observe the presence of the filter `[child::name/child::text()="onions"]`. Filters select some context nodes that verify certain conditions. In this case it means that we select all the element nodes `item` such that they have a children element with tag `name` containing a text `"onions"`. However, filters do not change the context node, that is, the `item` node verifying the filter is kept as context after the step. The rest of the location path navigates to the children of the `item` node with tag `price`, returning its text value. XPath allows also abbreviated forms. For instance the previous query can be written as:

```
doc("food.xml")/food/item[name="onions"]/price/text()
```

3 XPath Queries in $\mathcal{TOY}$

In this section we present the basis of our setting, including the type for XPath queries, the step combinators, tests and forward axes. Reverse axes are considered in Section 5.

3.1 The Type xPath

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type for `XPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [1] and also the usual in functional programming [4]. However, in our case we take advantage of the non-deterministic nature of our language, returning each result individually and avoiding the introduction of lists. We define an XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result: `type xPath = xml -> xml.`

3.2 Loading XML Documents and Combining XPath Queries

In order to apply an XPath expression to a particular document, we use the following infix operator definition:

```
(<--) :: string  -> xPath -> xml     S <-- Q = Q (load_xml_file S)
```

The input arguments of this operator are the string `S` representing the file name and an XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator plays in $\mathcal{TOY}$ the role of `doc` in XPath.

Next, we define the XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively. In $\mathcal{TOY}$, these symbols are defined simply as function composition:

```
infixr 55 .::.                    infixr 40 ./.
(.::.) :: xPath -> xPath -> xPath (./.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)           (F ./. G) X = G (F X)
```

We use the function operator names `.::.` and `./.` because `::` and `/` are already defined in $\mathcal{TOY}$. The variable `X` represents the input XML fragment (the context node). The rules specify how the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Observe that due to the precedence and associativity, an expression like: `A.::.B ./. C.::.D ./. E.::.F` is understood by $\mathcal{TOY}$ as: `(A.::.B) ./. ((C.::.D) ./. (E.::.F))` . The disjunction operator `|` of XPath is is represented in $\mathcal{TOY}$ simply by the choice operator `?` defined in Subsection 2.1.

3.3 Basic Axes and Tests

Figure 2 shows the representation in $\mathcal{TOY}$ of some basic axes. The first one is `self`, which returns the context node. In our setting, it corresponds simply to the

```
self,child,descendant :: xPath
descendant_or_self :: xPath
self X = X
child (tag _ _ L) = member L
descendant X = child X
descendant X = if child X == Y
               then descendant Y
descendant_or_self =
               self ? descendant
```

```
nodeT,elem :: xPath
nameT,textT,commentT::string->xPath
nodeT X = X
nameT S (tag S Att L ) = tag S Att L
textT S (txt S) = txt S
commentT S (comment S) = comment S
elem = nameT _
```

Fig. 2. XPath axes and tests in $\mathcal{TOY}$

identity function. A more interesting axis is `child` which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that these nodes correspond in $\mathcal{TOY}$ representation to terms rooted by constructor `tag`. Once `child` has been defined, `descendant` is just a generalization. The first rule for this function specifies that `child` must be used once, while the second rule corresponds to two or more applications of `child`. In this rule, the `if` statement is employed to ensure that `child` succeeds applied to the input XML fragment, thus avoiding possibly infinite recursive calls. Finally, the definition of axis `descendant-or-self` is straightforward. The first test defined in Figure 2 is `nodeT`, which corresponds to *node()* in the usual XPath syntax. This test is simply the identity. For instance, here is the XPath expression that returns all the nodes in an XML document, together with its $\mathcal{TOY}$ equivalent:

XPath → `doc("food.xml")/descendant-or-self::node()`
$\mathcal{TOY}$ → `("food.xml" <-- descendant_or_self.::.nodeT)==R`

The only difference is that the $\mathcal{TOY}$ expression returns one result at a time in the variable `R`, asking the user if more results are needed. If the user wishes to obtain all the solutions at a time, as usual in XPath evaluators, then it is enough to use the primitive `collect`. For instance, the answer to the $\mathcal{TOY}$ goal:

```
Toy> collect ("food.xml" <-- descendant_or_self.::.nodeT) == R
```

produces a single answer, with `R` instantiated to a list whose elements are the nodes in `"food.xml"`. The *name* test checks if the context node is an element with a certain name `S`. The test either returns as output the same XML fragment received as input, or fails. An example of a relative location path using this test:

XPath → `child::food/child::item`
$\mathcal{TOY}$ → `child.::.nameT "food"./.child.::.nameT "item"`

Notice that the expression in $\mathcal{TOY}$ is longer in length due to the presence of the identifier `nameT`, which is not required in XPath. In the next subsection

we see how this situation improves when introducing abbreviated forms. Other useful tests are `textT` and `commentT`, which correspond to *text()* and *comment()*, respectively, in XPath. In the case of $\mathcal{TOY}$, the text (respectively comment) string is obtained by means of a logic variable as, for instance, in:

```
XPath → child::food/child::item/child::price/child::text()
TOY   → child.::.nameT "food"./.child.::.nameT "item" ./.
        child.::.nameT "price"./.child.::.textT P
```

The logic variable `P` obtains the prices contained in the example document. Finally, the text `elem` represents in $\mathcal{TOY}$ the XPath test `*` which is satisfied only for *element* nodes. Notice in its definition (cf. Figure 2) the use of the anonymous variable `_` in its right-hand side indicating that any tag name is accepted.

3.4 Abbreviations

A number of abbreviations are used frequently in XPath expressions. The most important abbreviation is that `child::` can be omitted from a location step. This is usually done when `child::` is followed by a name test. Thus, the query `child::food/child::price/child::item` becomes simply `food/price/item`. In $\mathcal{TOY}$ we cannot do that directly because we are in a typed language and the combinator `./.` expects xPath expressions and not strings. However, we can introduce a similar abbreviation by defining new unitary operators `name` and `text`, which transform strings in XPath expressions:

```
name :: string -> xPath      name S = child.::.(nameT S)
```

An example:

```
XPath → food/item/price
TOY   → name "food"./.name "item"./.name "price"
```

The same idea can be applied to `commentT` and `textT`. Another XPath abbreviation is `//` which stands for `/descendant-or-self::node()/`. In $\mathcal{TOY}$:

```
infixr 30 .//.
(.//.) ::  xPath -> xPath -> xPath
A .//. B = append A (descendant_or_self .::. nodeT ./. B)
append :: xPath -> xPath -> xPath
append (A.::.B) C = (A.::.B) ./. C
append (X ./.Y) C = X ./. (append Y  C)
```

Notice that a new function `append` is used for concatenating the XPath expressions. This function is analogous to the well-known `append` for lists, but defined over `xPath` terms. This is our first example of the usefulness of higher-order patterns since for instance pattern `(A.::.B)` has type `xPath`, i.e. `xml -> xml`.

The next example uses both `name`, `.//.` and the disjunction operator, asking for all the elements with name either "price" or "variety":

```
XPath → food//(price | variety)
TOY   → name "food".//.(name "price" ? name "variety")
```

Another possible improvement is to define a new version of `./.` whose left-hand side is an XML name (a string):

```
infixr 35 /.
(/.) :: string -> xPath -> xPath        S /. X = name S ./. X
```

For instance:

```
XPath → food/item/price/text()
TOY   → "food"/."item"/."price"/.text P
```

Now the queries in XPath and in $\mathcal{TOY}$ look quite similar. In XPath we obtain the output: `32 74 55 210`, while in $\mathcal{TOY}$ we get the associated four solutions: `P` $\mapsto$ `32`, `P` $\mapsto$ `74`, `P` $\mapsto$ `55`, and `P` $\mapsto$ `210`.

3.5 Filters

Optionally, XPath tests can include a predicate or filter. Filters in XPath are enclosed between square brackets. In $\mathcal{TOY}$, they are enclosed between round brackets and connected to its associated XPath expression by the operator `.#`:

```
infixr  60 .#
(.#) :: xPath -> xPath -> xPath
(Q .# F)  X =  if F Y == _ then Y  where Y = Q X
```

This definition can be understood as follows: first the query `Q` is applied to the context node `X`, returning a new context node `Y`. Then the `if` condition checks whether `Y` satisfies the filter `F`, simply by checking that `F Y` does not fail, which means that it returns some value represented by the anonymous variable in `F Y == _`. Although XPath filter predicates allow several possibilities, in this presentation we restrict to XPath expressions. As in the previous subsection, it is convenient to define a version of `.#` accepting strings instead of XPath queries:

```
infixr 60 #
(#) :: string -> xPath -> xPath   S # F = child.::.(nameT S) .# F
```

Filters in XPath are defined usually by means of comparison operators, as `=` or `>`. For instance, the following XPath query asks for the price of watermelons: `food/item[name="watermelon"]/price`. The expression `name="watermelon"` means: check whether the context node has a children `name`, which has a children text `watermelon`. In $\mathcal{TOY}$ we can mimic this behavior by defining:

```
(.=) :: string -> string -> xPath
(.=) A B  = (A /. text B)
```

This operator takes as input parameters both sides of the equality, represented by the strings `A` and `B`, and the input XML context `X`. The strict equality with anonymous variable at the right-hand side is used to check whether `A` has a text child `B` in the XPath context `X`. An example of application of this operator:

XPath → `food/item[name="onions"]`
$\mathcal{TOY}$ → `"food"/."item"#("name".="onions")`

The same approach can be used for other operators, as `>`. Filters selecting attributes with certain values are of particular interest, and are represented in XPath by symbol `@`. In $\mathcal{TOY}$ they are represented by the operator `@=`:

```
(@=) :: xmlName -> xmlName -> xPath
(@=) S V X = if (xmlAtt S V == member Attr) then X
             where (xmlTag _Name Attr _L) = X
```

This filter checks if the attribute `S` of the context element takes the value `V`. The next example shows the items of type `fruit`:

XPath → `food/item[@type="fruit"]`
$\mathcal{TOY}$ → `"food"/."item"#("type"@="fruit")`

Or course, other comparison operators as `@>` can be defined analogously. As $\mathcal{TOY}$ is a typed language, several versions of the operators would be needed for the different involved types (strings, numbers, ...).

4 Generating Test-Cases for XPath Expressions

Suppose that we wish to know the price of onions as stored in our XML document. According to the previous section, we can write in $\mathcal{TOY}$:

```
Toy>("food.xml" <--
     name "food"./."item"#("type"@="onions")./.name "price" ) == R
```

The goal returns no answer, but we know that `"food.xml"` includes the price of onions. Where is the error? Sometimes it is useful to have a test-case, i.e., an XML file which contains some answer for the query. Comparing the test-case and the original XML document can help to find the error. In our setting, such test-cases are obtained for free. For instance, we can submit the goal:

```
Toy>(name "food"./."item"#("type"@="onions")./.name "price") X== _
```

asking for an XML document `X` such that the query succeeds. The anonymous variable at the right-hand side of the strict equality indicates that we are not interested in the output. However, the answer is difficult to read and understand:

```
X -> tag _A _B [tag  "food" _C [
     tag  "item" [att  "type"  "onions" | _D ]
                 [tag  "price" _E _F | _G ] | _H ] | _I ]
```

The logic variables indicate that replacing them by any valid XML fragment produces a valid XML test-case for the query. In particular, in the case of lists, they indicate that other elements can be added, and the smaller test-case corresponds to substituting these variables by the empty list. In order to enhance the readability of the result we define a function:

```
generateTC :: xPath -> string -> bool
generateTC F S = if (F X == _) then write_xml_file X S
```

This function receives the XPath expression and the file name `S` as input parameters, looks for an XML test-case `X`, and writes it to the file using the primitive `write_xml_file`. The goal:

```
Toy> generateTC (name "food"./."item"#("type"@="onions")./.
                 name "price") "tc.xml" == R
```

produces the following XML file "tc.xml":

```
<food>
 <item type="onions">
  <price />
 </item>
</food>
```

It is worth noticing that the primitive has replaced the logic variables by empty elements. Comparing this file and our example `"food.xml"`, we see that "onions" is not an attribute, but a child node. Therefore, the correct query should be:

```
Toy> ("food.xml"<--
      name "food"./."item"#("name".="onions")./.name "price")==R
```

which returns the answer: `R` $\rightarrow$ `tag "price" [] [text "55"]`.

5 Higher Order Patterns

The possibility of employing higher order patterns in $\mathcal{TOY}$ allows the user to consider XPath queries as truly data terms. Queries can be examined and modified before and during its evaluation, as any constructed term. In this section, we take advantage of this feature in two ways. First, we define a function that checks if an XPath query follows the XPath standard. Then, we apply a transformation similar to those described in [9] for introducing the reverse axis `parent`.

5.1 Validating XPath Queries

So far, we have described several different tests and axes that can be combined for defining XPath queries. Moreover, our setting allows the user to define their own combinators, axes and tests, or to use the existing ones in a non-standard way. For instance, the query `nodeT.::.child` is allowed, although it does not follows the XPath grammar (it should be `child.::.nodeT`, first the axis and then the test). The reason is that the expression is well-typed from the point of view of a $\mathcal{TOY}$ expression. Although in principle such unusual queries can work and even be useful in some cases, it is convenient to define a function that indicates whether a query conforms to the XPath standard or not. However, in the previous sections we have defined many different abbreviations. Should we consider all of them for detecting standard queries? Fortunately, the answer is 'no'. It is enough to recognize the few basic axes and tests, because the abbreviations are automatically reduced to these basis forms during computations. For instance, the goal `Toy> ("name" /. text T) == R` yields:

```
R ->  child.::.nameT "name" ./. child.::.textT T
```

Now we are ready to define the function `standard` using higher-order patterns:

```
standard,step,test:: xPath -> bool
simpleTest,axis::xPath -> bool

standard A             = step A
standard (A ./. B)     = step A /\ standard B
step (Axis.::.Test)    = (axis Axis) /\ (test Test)
axis A                 = (A==child)\/(A==self)\/(A==descendant)
test A                 = simpleTest A
test (A .# B)          = simpleTest A /\ standard B
simpleTest nodeT       = true
simpleTest (nameT S)   = true
simpleTest (textT S)   = true
simpleTest (commentT S)= true
```

Function `standard` succeeds if the query is either a single step of several steps combined by the operator `(./.)`. Steps are defined by an axis and a test connected by `(.::.)`. Finally, the definition of functions `test`, `simpleTest` and `axis` is self-explanatory. For instance, the goal: `Toy> standard ("food" /. name "item")` produces the answer `yes`, but `standard (nodeT.::.child)` produces the answer `no`, meaning that the query is not standard.

5.2 Reverse Axes

The queries defined so far only use forward axes such as `descendant` or `child`. However, in XPath reverse axes such as `parent` are also allowed. Implementing these axes is not trivial in our approach, since each `xPath` function receives

```
delParent :: xPath -> xPath ->xPath
delParent (X./.self.::.T1) T2 = addFilter (delParent X T2) (self.::.T1)
delParent (X./.child.::.T1) T2 = X./.self.::.(T2.#(child.::.T1)
delParent (X./.descendant.::.T1) T2= X./.self.::.T2.#(child.::.T1)
delParent (X./.descendant.::.T1) T2= X./.descendant.::.T2.#(child.::.T1)

preprocess :: xPath -> xPath
preprocess A = rev (foldl transform (self.::.nodeT) A)

foldl :: (xPath -> xPath -> xPath) -> xPath -> xPath -> xPath
foldl F Z (A.::.T) = F Z (A.::.T)
foldl F Z (G ./. H) = foldl F (F Z G) H

transform :: xPath -> xPath -> xPath
transform X (self.::.T) = X ./.(self.::.T)
transform X (child.::.T) = X ./.(child.::.T)
transform X (descendant.::.T) = X ./.(descendant.::.T)
transform X (parent.::.T) = delParent X T

addFilter :: xPath -> xPath -> xPath
addFilter (X./.A.::.(T.#F)) G = X ./. (A.::. (T.# (F ./. G)))

rev :: xPath -> xPath
rev (A.::.B) = A.::.B
rev (F./.G) = rev' F G
rev' (A.::.B) G = (A.::.B) ./. G
rev' (X ./. Y) G = rev' X (Y./. G)
```

Fig. 3. Preprocessing `parent` axis

as input the fragments of the XML document that satisfied the previous steps. These fragments corresponds to a subtree of the XML document and thus it is not possible to obtain the `parent` of the current XML fragment. A possible solution is to include the whole XML document and a representation of the path leading to the context node as input parameters, following by instance the ideas in [6]. Nevertheless, this complicates the implementation, and the simple definitions of the previous sections would be no longer valid. An alternative is to preprocess the query, replacing the reverse axes by predicate filters including forward axes, as shown in [9]. For the sake of space we only include the rules for removing `parent` outside filter predicates, although the same approach can be extended to `parent` in filter predicates, to `ancestor`, and to `following-sibling`.

(P_1) $\mathsf{child}::T_1/\mathsf{S}/\mathsf{parent}::T_2 \equiv \mathsf{self}::T_2[\mathsf{child}::T_1/\mathsf{S}]$
(P_2) $\mathsf{descendant}::T_1/\mathsf{S}/\mathsf{parent}::T_2 \equiv \mathsf{self}::T_2[\mathsf{child}::T_1/\mathsf{S}]$
(P_3) $\mathsf{descendant}::T_1/\mathsf{S}/\mathsf{parent}::T_2 \equiv \mathsf{descendant}::T_2[\mathsf{child}::T_1/\mathsf{S}]$

where T_1 and T_2 are tests that optionally can include filters, and S is a (possibly empty) sequence of steps using the `self` axis. For instance the relative location path `child::variety/parent::node()` is transformed by (P_1) into the equivalent expression `self::node()[child::variety]`. The equations are

implemented in $\mathcal{TOY}$ through the program rules for `delParent` which can be found in Figure 3. The first program rule is used for skipping the sequence S, while the three following rules resemble closely $(P_1), (P_2), (P_3)$ when S is the empty sequence. In order to apply this function, we change the definition of the operator `<--`, which now preprocesses the query before applying it to the XML document: `S <-- F = (preprocess F) (load_xml_file S)`. Then we define an initial version of `parent` that indicates that it fails without preprocessing:

```
parent::xPath            parent S =  if false then S
```

Function `preprocess` uses a version of the well-known catamorphism `fold` acting over XPath queries to apply a function `transform` to each individual steps, which in turn employs `delParent` as auxiliary function. The result is obtained with the steps associated to the left, as in $(S_1$`./.`$S_2)$`./.`S_3. This is corrected by function `rev` which is the analogous to the reverse function used in functional program for lists. All this code is possible thanks to the use of higher-order patterns. The next example looks for nodes having at least one "variety" child.

```
XPath → doc("food.xml")/food//variety/parent::node()
TOY   → name "food".//."variety"/.parent.::.nodeT
```

6 Conclusions

We have shown how the declarative nature of the XML query language XPath fits in a very natural way in functional-logic languages. XPath queries are represented in this setting by non-deterministic higher-order expressions, thus becoming first-class citizens of the language that can be readily extended and adapted by the programmer. In the case of the functional-logic language $\mathcal{TOY}$, the possibility of using higher-order patterns make this affirmation even more valid, since XPath expressions manipulated directly as data terms. The result is enriching for both XPath and $\mathcal{TOY}$ users:

- For the users of the functional-logic $\mathcal{TOY}$ the advantage is clear: they can use XPath queries in their programs in a natural way. The queries are written in $\mathcal{TOY}$ and thus using them requires little effort. Moreover, since the combinators, tests and axes are written in $\mathcal{TOY}$ they can be freely modified and extended. The situation can be analogous to the introduction of parsers in functional [7] and functional-logic languages [3].
- From the point of view of the XPath apprentices, the tool can be useful, specially if they have some previous knowledge of declarative languages. The possibility of generating test-cases for XPath queries is an easy and powerful tool that can be very helpful for understanding the basics of XPath.
- The framework can also be interesting for designers of XPath environments, because it allows the users to easily define prototypes of new features such as new combinators or functions.

Our proposal also contains some drawbacks that deserve to be discussed. First of all, the syntax of the queries resembles quite closely XPath, but the differences can be confusing at first. However, in our experience this difficulty is soon overcome by practice, and in any case is easy to write a parser converting standard XPath format to the format explained in this paper. Another difficulty arises from the implementation of features using the position of the node in the sequence. This features can be introduced in our non-deterministic setting, but only using some impure primitive like `collect` that bundles in a list the results of a non-deterministic expression. The problem with this impure primitive is that cannot deal with logic variables, which can be a problem for instance for the generation of test-cases.

A description of how to download and install the $\mathcal{TOY}$ system including the source code of the XPath library, and a description of some extensions like the `ancestor` axis, position filters, and more, can be found at [2].

References

1. Almendros-Jiménez, J.M.: An Encoding of XQuery in Prolog. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 145–155. Springer, Heidelberg (2009)
2. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Integrating XPath with the Functional-Logic Language Toy (Extended Version). Technical Report SIP-05/10, Facultad de Informática, Universidad Complutense de Madrid (2010), `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-5-10.pdf`
3. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective on parsing. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
4. Guerra, R., Jeuring, J., Swierstra, S.D.: Generic validation in an XPath-Haskell data binding. In: Proceedings Plan-X (2005)
5. Hanus, M.: Curry: An Integrated Functional Logic Language (version 0.8.2) (March 28, 2006), `http://www.informatik.uni-kiel.de/~mh/curry/` (2003)
6. Huet, G.: The zipper. J. Funct. Program. 7(5), 549–554 (1997)
7. Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. 8(4), 437–444 (1998)
8. López-Fraguas, F.J., Hernández, J.S.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
9. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
10. Sulzmann, M., Lu, K.Z.: Xhaskell — adding regular expression types to haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
11. W3C. XML Schema 1.1
12. W3C. Extensible Markup Language, XML (2007)
13. W3C. XML Path Language (XPath) 2.0 (2007)
14. W3C. XQuery 1.0: An XML Query Language (2007)
15. Walmsley, P.: XQuery. O'Reilly Media, Inc., Sebastopol (2007)

A Declarative Embedding of XQuery in a Functional-Logic Language*

Jesús M. Almendros-Jiménez[1], Rafael Caballero[2], Yolanda García-Ruiz[2], and Fernando Sáenz-Pérez[3]

[1] Dpto. Lenguajes y Computación, Universidad de Almería, Spain
[2] Dpto. de Sistemas Informáticos y Computación, UCM, Spain
[3] Dpto. de Ingeniería del Software e Inteligencia Artificial, UCM, Spain
jalmen@ual.es, {rafa,fernan}@sip.ucm.es , ygarciar@fdi.ucm.es

Abstract. This paper addresses the problem of integrating a fragment of XQuery, a language for querying XML documents, into the functional-logic language $\mathcal{TOY}$. The queries are evaluated by an interpreter, and the declarative nature of the proposal allows us to prove correctness and completeness with respect to the semantics of the subset of XQuery considered. The different fragments of XML that can be produced by XQuery expressions are obtained using the non-deterministic features of functional-logic languages. As an application of this proposal we show how the typical *generate and test* techniques of logic languages can be used for generating test-cases for XQuery expressions.

1 Introduction

XQuery has been defined as a query language for finding and extracting information from XML [15] documents. Originally designed to meet the challenges of large-scale electronic publishing, XML also plays an important role in the exchange of a wide variety of data on the Web and elsewhere. For this reason many modern languages include libraries or encodings of XQuery, including logic programming [1] and functional programming [6]. In this paper we consider the introduction of a simple subset of XQuery [18,20] into the functional-logic language $\mathcal{TOY}$ [11].

One of the key aspects of declarative languages is the emphasis they pose on the logic semantics underpinning declarative computations. This is important for reasoning about computations, proving properties of the programs or applying declarative techniques such as abstract interpretation, partial evaluation or algorithmic debugging [14]. There are two different declarative alternatives that can be chosen for incorporating XML into a (declarative) language:

1. Use a domain-specific language and take advantage of the specific features of the host language. This is the approach taken in [9], where a rule-based

* Work partially supported by the Spanish projects STAMP TIN2008-06622-C03-01, DECLARAWEB TIN2008-06622-C03-03, Prometidos-CM S2009TIC-1465 and GPD UCM-BSCH-GR58/08-910502.

G. Vidal (Ed.): LOPSTR 2011, LNCS 7225, pp. 42–56, 2012.

language for processing semi-structured data that is implemented and embedded into the functional-logic language Curry, and also in [13] for the case of logic programming.
2. Consider an existing query language such as XQuery, and embed a fragment of the language in the host language, in this case $\mathcal{TOY}$. This is the approach considered in this paper.

Thus, our goal is to include XQuery using the purely declarative features of the host languages. This allows us to prove that the semantics of the fragment of XQuery has been correctly included in $\mathcal{TOY}$. To the best of our knowledge, it is the first time a fragment of XQuery has been encoded in a functional-logic language. A first step in this direction was proposed in [5], where XPath [16] expressions were introduced in $\mathcal{TOY}$. XPath is a subset of XQuery that allows navigating and returning fragments of documents in a similar way as the path expressions used in the *chdir* command of many operating systems. The contributions of this paper with respect to [5] are:
- The setting has been extended to deal with a simple fragment of XQuery, including *for* statements for traversing XML sequences, *if/where* conditions, and the possibility of returning XML elements as results. Some basic XQuery constructions such as *let* statements are not considered, but we think that the proposal is powerful enough for representing many interesting queries.
- The soundness of the approach is formally proved, checking that the semantics of the fragment of XQuery is correctly represented in $\mathcal{TOY}$.

Next section introduces the fragment of XQuery considered and a suitable operational semantics for evaluating queries. The language $\mathcal{TOY}$ and its semantics are presented in Section 3. Section 4 includes the interpreter that performs the evaluation of simple XQuery expressions in $\mathcal{TOY}$. The theoretical results establishing the soundness of the approach with respect to the operational semantics of Section 2 are presented in Section 4.1. Section 5 explains the automatic generation of test cases for simple XQuery expressions. Finally, Section 6 concludes summarizing the results and proposing future work.

An extended version of the paper including proofs of the theoretical results can be found at [2].

2 XQuery and Its Operational Semantics

XQuery allows the user to query several documents, applying join conditions, generating new XML fragments, and using many other features [18,20]. The syntax and semantics of the language are quite complex [19], and thus only a small subset of the language is usually considered. The next subsection introduces the fragment of XQuery considered in this paper.

2.1 The Subset SXQ

In [4] a declarative subset of XQuery, called XQ, is presented. This subset is a core language for XQuery expressions consisting of *for*, *let* and *where/if* statements.

$$
\begin{aligned}
query ::= & \ (\ ) \mid query\ query \mid tag \\
& \mid doc(File) \mid doc(File)/axis :: \nu \mid var \mid var/axis :: \nu \\
& \mid \texttt{for}\ var\ \texttt{in}\ query\ \texttt{return}\ query \\
& \mid \texttt{if}\ cond\ \texttt{then}\ query \\
cond ::= & \ var{=}var \mid query \\
tag ::= & \ \langle a \rangle\ var \ldots var\ \langle /a \rangle \mid \langle a \rangle\ tag\ \langle /a \rangle
\end{aligned}
$$

Fig. 1. Syntax of SXQ, a simplified version of XQ

In this paper we consider a simplified version of XQ which we call SXQ and whose syntax can be found in Figure 1. where *axis* can be one of *child*, *self*, *descendant* or *dos* (i.e. descendant or self), and ν is a node test. The differences of SXQ with respect to XQ are:

1. XQ includes the possibility of using variables as tag names using a constructor *lab($x)*.
2. XQ permits enclosing any query Q between tag labels $\langle a \rangle Q \langle /a \rangle$. SXQ only admits either variables or other tags inside a tag.

Our setting can be easily extended to support the *lab($x)* feature, but we omit this case for the sake of simplicity in this presentation. The second restriction is more severe: although *let*s are not part of XQ, they could be simulated using *for* statements inside tags. In our case, forbidding other queries different from variables inside tag structures imply that our core language cannot represent *let* expressions. This limitation is due to the non-deterministic essence of our embedding, since a *let* expression means collecting all the results of a query instead of producing them separately using non-determinism. In spite of these limitations, the language SXQ is still useful for solving many common queries as the following example shows.

Example 1. Consider an XML file *"bib.xml"* containing data about books, and another file *"reviews.xml"* containing reviews for some of these books (see [17], sample data 1.1.2 and 1.1.4 to check the structure of these documents and an example). Then we can list the reviews corresponding to books in *"bib.xml"* as follows:

```
for $b in doc("bib.xml")/bib/book,
    $r in doc("reviews.xml")/reviews/entry
where $b/title = $r/title
for $booktitle in $r/title,
    $revtext in $r/review
return <rev> $booktitle  $revtext </rev>
```

The variable $\$b$ takes the value of the different books, and $\$r$ the different reviews. The *where* condition ensures that only reviews corresponding to the book are considered. Finally, the last two variables are only employed to obtain

the book title and the text of the review, the two values that are returned as output of the query by the *return* statement.

It can be argued that the code of this example does not follow the syntax of Figure 1. While this is true, it is very easy to define an algorithm that converts a query formed by *for*, *where* and *return* statements into a SXQ query (as long as it only includes variables inside tags, as stated above). The idea is simply to convert the *where* into *if*s, following each *for* by a *return*, and decomposing XPath expressions including several steps into several *for* expressions by introducing a new auxiliary variable and each one consisting of a single step.

Example 2. The query of Example 1 using SXQ syntax:

```
for $x1 in doc("bib.xml")/child::bib return
for $x2 in $x1/child::book  return
for $x3 in doc("reviews.xml")/child::reviews return
for $x4 in $x3/entry  return
if ($x2/title = $x4/title) then
    for $x5 in $x4/title return
     for $x6 in $x4/review return  <rev> $x5  $x6 </rev>
```

We end this subsection with a few definitions that are useful for the rest of the paper. The set of variables in a query Q is represented as $Var(Q)$. Given a query Q, we use the notation $Q_{|p}$ for representing the subquery Q' that can be found in Q at position p. Positions are defined as usual in syntax trees:

Definition 1. *Given a query Q and a position p, $Q_{|p}$ is defined as follows:*

$$\begin{array}{ll}
Q_{|\varepsilon} & = Q \\
(Q_1\ Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{for var in } Q_1 \texttt{ return } Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{if } Q_1 \texttt{ then } Q_2)_{|(i\cdot p)} & = (Q_i)_{|p}\ \ i \in \{1,2\} \\
(\texttt{if var=var then } Q_1)_{|(1\cdot p)} & = (Q_1)_{|p}
\end{array}$$

Hence the position of a subquery is the path in the syntax tree represented as the concatenation of children positions $p_1 \cdot p_2 \ldots \cdot p_n$. For every position p, $\varepsilon \cdot p = p \cdot \varepsilon = p$. In general $Q_{|p}$ is not a proper SXQ query, since it can contain *free variables*, which are variables defined previously in *for* statements in Q. The set of variables of Q that are *relevant* for $Q_{|p}$ is the subset of $Var(Q)$ that can appear free in any subquery at position p. This set, denoted as $Rel(Q,p)$ is defined recursively as follows:

Definition 2. *Given a query Q, and a position p, $Rel(Q,p)$ is defined as:*

1. $\emptyset$, *if* $p = \varepsilon$.
2. $Rel(Q_1,p')$, *if* $Q \equiv Q_1\ Q_2$, $p = 1 \cdot p'$.
3. $Rel(Q_2,p')$, *if* $Q \equiv Q_1\ Q_2$, $p = 2 \cdot p'$.
4. $Rel(Q_1,p')$, *if* $Q \equiv$ `for` var `in` Q_1 `return` Q_2, $p = 1 \cdot p'$.
5. $\{\text{var}\} \cup Rel(Q_2,p')$, *if* $Q \equiv$ `for` var `in` Q_1 `return` Q_2, $p = 2 \cdot p'$.
6. $Rel(Q_1,p')$, *if* $Q \equiv$ `if` Q_1 `then` Q_2, $p = 1 \cdot p'$.

7. $Rel(Q_2, p')$, *if* $Q \equiv$ `if` Q_1 `then` Q_2, $p = 2 \cdot p'$.

Observe that cases $Q \equiv ()$, $Q \equiv tag$, $Q \equiv \text{var}$, $Q \equiv \text{var}/\chi :: \nu$, *and* $\text{var} = \text{var}$ *correspond to* $p \equiv \varepsilon$.

Without loss of generality we assume that all the relevant variables for a given position are indexed starting from 1 at the outer level. We also assume that every `for` statement introduces a new variable. A query like `for X in ((for Y in ...) (for Y in ...)) ...` is then renamed to an equivalent query of the form `for` X_1 `in ((for` X_2 `in ...) (for` X_3 `in ...)) ...` (notice that the two Y variables occurred in different scopes).

2.2 XQ Operational Semantics

Figure 2 introduces the operational semantics of XQ that can be found in [4]. The only difference with respect to the semantics of this paper is that there is no rule for the constructor *lab*, for the sake of simplicity.

As explained in [4], the previous semantics defines the denotation of an XQ expression Q with k relevant variables, under a graph-like representation of a data forest $\mathcal{F}$, and a list of indexes $\overline{e}$ in $\mathcal{F}$, denoted by $[\![Q]\!]_k(\mathcal{F}, \overline{e})$. In particular, each relevant variable $\$x_i$ of Q has as value the tree of $\mathcal{F}$ indexed at position e_i. $\chi^{\mathcal{F}}(e_i, v)$ is a boolean function that returns *true* whenever v is the subtree of $\mathcal{F}$ indexed at position e_i. The operator $construct(a, (\mathcal{F}, [w_1 ... w_n]))$, denotes the construction of a new tree, where a is a label, $\mathcal{F}$ is a data forest, and $[w_1 \dots w_n]$ is a list of nodes in $\mathcal{F}$. When applied, *construct* returns an indexed forest $(\mathcal{F} \cup T', [root(T')])$, where T' is a tree with domain a new set of nodes, whose root is labeled with a, and with the subtree rooted at the i-th (in sibling order) child of $\text{root}(T')$ being an isomorphic copy of the subtree rooted by w_i in $\mathcal{F}$. The symbol $\biguplus$ used in the rules takes two indexed forests $(\mathcal{F}_1, l_1), (\mathcal{F}_2, l_2)$ and returns an indexed forest $(\mathcal{F}_1 \cup \mathcal{F}_2, l)$, where $l = l_1 \cdot l_2$. Finally, $tree(e_i)$ denotes the maximal tree within the input forest that contains the node e_i, hence $<_{doc}^{tree(e_i)}$ is the document order on the tree containing e_i.

$$
\begin{aligned}
[\![(\,)]\!]_k(\mathcal{F}, \overline{e}) &= (\mathcal{F}, [\,]) \\
[\![Q_1\ Q_2]\!]_k(\mathcal{F}, \overline{e}) &= [\![Q_1]\!]_k(\mathcal{F}, \overline{e}) \uplus [\![Q_2]\!]_k(\mathcal{F}, \overline{e}) \\
[\![\texttt{for } \$x_{k+1} \texttt{in } Q_1 \texttt{ return } Q_2]\!]_k(\mathcal{F}, \overline{e}) &= \text{let } (\mathcal{F}', \overline{l}) = [\![Q_1]\!]_k(\mathcal{F}, \overline{e}) \text{ in} \\
&\qquad \biguplus_{1 \le i \le |\overline{l}|} [\![Q_2]\!]_{k+1}(\mathcal{F}', \overline{e} \cdot l_i) \\
[\![\$x_i]\!]_k(\mathcal{F}, [e_1, \dots, e_k]) &= (\mathcal{F}, [e_i]) \\
[\![\$x_i/\chi :: \nu]\!]_k(\mathcal{F}, [e_1, \dots, e_k]) &= (\mathcal{F}, \text{list of nodes } v \text{ such that } \chi^{\mathcal{F}}(e_i, v) \text{ and} \\
&\qquad \text{label name of } v = \nu \text{ in order } <_{doc}^{tree(e_i)}) \\
[\![\texttt{if } C \texttt{ then } Q_1]\!]_k(\mathcal{F}, \overline{e}) &= \text{if } \pi_2([\![C]\!]_k(\mathcal{F}, \overline{e})) \neq [\,] \text{ then } [\![Q_1]\!]_k(\mathcal{F}, \overline{e}) \\
&\qquad \text{else } (\mathcal{F}, [\,]) \\
[\![\$x_i = \$x_j]\!]_k(\mathcal{F}, [e_1, \dots, e_k]) &= \text{if } e_i = e_j \text{ then } construct(yes, (\mathcal{F}, [\,])) \\
&\qquad \text{else } (\mathcal{F}, [\,])
\end{aligned}
$$

Fig. 2. Semantics of Core XQuery

Without loss of generality this semantics assumes that all the variables relevant for a subquery are numbered consecutively starting by 1 as in Example 2. It also assumes that the documents appear explicitly in the query. That is, in Example 2 we must suppose that instead of *doc("bib.xml")* we have the XML corresponding to this document. Of course this is not feasible in practice, but simplifies the theoretical setting and it is assumed in the rest of the paper.

These semantic rules constitute a term rewriting system (TRS in short, see [3]), with each rule defining a single reduction step. The symbol $:=^*$ represents the reflexive and transitive closure of $:=$ as usual. The TRS is terminating and confluent (the rules are not overlapping). Normal forms have the shape $(\mathcal{F}, e_1, \ldots, e_n)$ where $\mathcal{F}$ is a forest of XML fragments, and e_i are nodes in $\mathcal{F}$, meaning that the query returns the XML fragments (**indexed by**) $e_1, \ldots, e_n$. The semantics evaluates a query starting with the expression $[\![Q]\!]_0(\emptyset, ())$. Along intermediate steps, expressions of the form $[\![Q']\!]_k(\mathcal{F}, \overline{e}_k)$ are obtained. The idea is that Q' is a subquery of Q with k relevant variables (which can occur free in Q'), that must take the values $\overline{e}_k$. The next result formalizes these ideas.

Proposition 1. *Let Q be a SXQ query. Suppose that*

$$[\![Q]\!]_0(\emptyset, ()) :=^* [\![Q']\!]_n(\mathcal{F}, \overline{e}_n)$$

Then:

- *Q' is a subquery of Q, that is, $Q' = Q_p$ for some p.*
- *$Rel(Q, p) = \{X_1, \ldots, X_n\}$.*
- *Let S be the set of free variables in Q'. Then $S \subset Rel(Q, p)$.*
- *$[\![Q']\!]_n(\mathcal{F}, \overline{e}_n) = [\![Q'\theta]\!]_0(\emptyset, ())$, with $\theta = \{X_1 \mapsto e_1, \ldots, X_n \mapsto e_n\}$*

Proof. Straightforward from Definition 2, and from the XQ semantic rules of Figure 2.

A more detailed discussion about this semantics and its properties can be found in [4].

3 $\mathcal{TOY}$ and Its Semantics

A $\mathcal{TOY}$ [11] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of *partial expressions* in $\mathcal{TOY}$ $e \in Exp_\perp$ is $e ::= \perp \mid X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). Similarly, the syntax of *partial patterns* $t \in Pat_\perp \subset Exp_\perp$ can be defined as $t ::= \perp \mid X \mid c\ t_1 \ldots t_m \mid f\ t_1 \ldots t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, being t_i partial patterns for all $1 \leq i \leq m$. Each rule for a function f in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{C_1, \ldots, C_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where u_i and r are expressions (that can contain new extra variables), C_j are strict equalities, and t_i, s_i are patterns. In $\mathcal{TOY}$, variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$:

```
data node      = txt    string
               | comment string
               | tag     string [attribute] [node]
data attribute = att     string string
type xml       = node
```

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation.

$\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience all the documents are started with a dummy node `root`. This is useful for grouping several XML fragments. If the file contains only one node `N` at the outer level, the `root` node is unnecessary, and can be removed using this simple function:

```
load_doc F = N <== load_xml_file F == xmlTag "root" [] [N]
```

where `F` is the name of the file containing the document. Observe that the strict equality `==` in the condition forces the evaluation of `load_xml_file F` and succeeds if the result has the form `xmlTag "root" [] [N]` for some `N`. If this is the case, `N` is returned.

The constructor-based ReWriting Logic (CRWL) [7] has been proposed as a suitable declarative semantics for functional-logic programming with lazy nondeterministic functions. The calculus is defined by five inference rules (see Figure 3): (BT) that indicates that any expression can be approximated by bottom, (RR) that establishes the reflexivity over variables, the decomposition rule (DC), the (JN) (join) rule that indicates how to prove strict equalities, and the function application rule (FA). In every inference rule, $r, e_i, a_j \in Exp_\perp$ are partial expressions and $t, t_k \in Pat_\perp$ are partial patterns. The notation $[P]_\perp$ of the inference rule FA represents the set $\{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in P,\ \theta \in Subst_\perp\}$ of partial instances of the rules in program P ($Subst_\perp$ represents the set of partial

BT $e \to \bot$

RR $X \to X$ with $X \in Var$

DC $\dfrac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$ $h\ \overline{t}_m \in Pat_\bot$

JN $\dfrac{e \to t \quad e' \to t}{e \ == \ e'}$ $t \in Pat$ (total pattern)

FA $\dfrac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \ \to t}$
if $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot, t \neq \bot$

Fig. 3. CRWL Semantic Calculus

substitutions that replace variables by partial terms). The most complex inference rule is FA (Function Application), which formalizes the steps for computing a *partial pattern* t as approximation of a function call $f\ \overline{e}_n$:

1. Obtain partial patterns t_i as suitable approximations of the arguments e_i.
2. Apply a program rule $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot$, verify the condition C, and check that t approximates the right-hand side r.

In this semantic notation, local declarations $a = b$ introduced in $\mathcal{TOY}$ syntax by the reserved word `where` are part of the condition C as approximation statements of the form $b \to a$.
The semantics in $\mathcal{TOY}$ allows introducing non-deterministic functions, such as the following function `member` that returns all the elements in a list:

```
member:: [A]  -> A
member  [X | Xs] =  X
member  [X | Xs] =  member Xs
```

Another example of $\mathcal{TOY}$ function is the definition of the infix operator `.:::.` for XPath expressions (the operator `::` in XPath syntax):

```
(.:::.) :: (A -> B) -> (B -> C) -> (A -> C)
(F .:::. G) X = G (F X)
```

As the examples show, $\mathcal{TOY}$ is a typed language. However the type declaration is optional and in the rest of the paper they are omitted for the sake of simplicity. *Goals* in $\mathcal{TOY}$ are sequences of strict equalities. A strict equality $e_1 == e_2$ holds (inference JN) if both e_1 and e_2 can be reduced to the same total pattern t. For instance, the goal `member [1,2,3,4] == R` yields four answers, the four values for `R` that make the equality true: $\{\texttt{R} \mapsto \texttt{1}\}, \ldots, \{\texttt{R} \mapsto \texttt{4}\}$.

4 Transforming SXQ into $\mathcal{TOY}$

In order to represent SXQ queries in $\mathcal{TOY}$ we use some auxiliary datatypes:

```
type xPath = xml-> xml

data sxq  = xfor xml sxq sxq | xif cond sxq | xmlExp xml |
                    xp path | comp sxq sxq
data cond = xml := xml | cond sxq
data path = var xml | xml :/ xPath | doc string xPath
```

The structure of the datatype `sxq` allows representing any SXQ query (see SXQ syntax in Figure 1). It is worth noticing that a variable introduced by a *for* statement has type `xml`, indicating that the variable always contains a value of this type. $\mathcal{TOY}$ includes a primitive `parse_xquery` that translates any SXQ expression into its corresponding representation as a term of this datatype, as the next example shows:

Example 3. The translation of the SXQ query of Example 2 into the datatype `sxq` produces the following $\mathcal{TOY}$ data term:

```
Toy> parse_xquery  "for $x1 in doc(\"bib.xml\")/child::bib return
          for $x2 in .....  <rev> $x5  $x6 </rev>" == R
yes
{R --> xfor X1 (xp (doc "bib.xml" (child .::. (nameT "bib"))))
    (xfor X2 (xp ( X1 :/ (child .::.(nameT "book"))))
    (xfor X3 (xp (doc "reviews.xml" (child  .::. (nameT "reviews"))))
    (xfor X4 (xp ( X3 :/ (child .::.(nameT "entry"))))
    (xif  ((xp(X2 :/ (child .::.(nameT "title")))) :=
             (xp(X4 :/ (child .::.(nameT "title")))))
    (xfor X5 (xp ( X4 :/ (child .::.(nameT "title"))))
    (xfor X6 (xp ( X4 :/ (child .::.(nameT "review"))))
       (xmlExp (xmlTag "rev" [] [X5,X6])))))))
 }
```

The interpreter assumes the existence of the infix operator `.::.` that connects axes and tests to build steps, defined as the sequence of applications in Section 3.

The rules of the $\mathcal{TOY}$ interpreter that processes SXQ queries can be found in Figure 4. The main function is `sxq`, which distinguishes cases depending of the form of the query. If it is an XPath expression then the auxiliary function `sxqPath` is used. If the query is an XML expression, the expression is just returned (this is safe thanks to our constraint of allowing only variables inside XML expressions). If we have two queries (`comp` construct), the result of evaluating any of them is returned using non-determinism. The `for` statement (`xfor` construct) forces the evaluation of the query `Q1` and binds the variable `X` to the result. Then the result query `Q2` is evaluated. The case of the `if` statement is analogous. The XPath subset considered includes tests for attributes (`attr`), label names (`nameT`), general elements (`nodeT`) and text nodes (`textT`). It also includes the axes `self`, `child`, `descendant` and `dos`. Observe that we do not

```
sxq (xp E)              = sxqPath E
sxq (xmlExp X)          = X
sxq (comp Q1 Q2)        = sxq Q1
sxq (comp Q1 Q2)        = sxq Q2
sxq (xfor X Q1 Q2)      = sxq Q2 <== X== sxq Q1
sxq (xif (Q1:=Q2) Q3)   = sxq Q3 <== sxq Q1  == sxq Q2
sxq (xif (cond Q1) Q2) = sxq Q2 <== sxq Q1  == _

sxqPath (var X)  = X
sxqPath (X :/ S) = S X
sxqPath (doc F S) = S (load_xml_file F)

%%%% XPATH %%%%%%%
attr A (xmlTag S Attr L ) = xmlText T <== member Attr == xmlAtt A T
nameT S (xmlTag S Attr L ) = xmlTag S Attr L
nodeT X = X
textT (xmlText S) = xmlText S
commentT S (xmlComment S) = xmlComment S

self X = X
child (xmlTag _Name _Attr L) = member L
descendant X = child X
descendant X = descendant Y <== child X == Y
dos = self
dos = descendant
```

Fig. 4. $\mathcal{TOY}$ transformation rules for SXQ

include reverse axes like `ancestor` because they can be replaced by expressions including forward axes, as shown in [12]. Other constructions such as filters can be easily included (see [5]). The next example uses the interpreter to obtain the answers for the query of our running example.

Example 4. The goal `sxq (parse_xquery "for....")) == R` applies the interpreter of Figure 4 to the code of Example 2 (assuming that the string after `parse_xquery` is the query in Example 2), and returns the $\mathcal{TOY}$ representation of the expected results:

```
<rev>
 <title>TCP/IP Illustrated</title>
 <review>  One of the best books on TCP/IP.  </review>
</rev>
...
```

4.1 Soundness of the Transformation

One of the goals of this paper is to ensure that the embedding is semantically correct and complete. This section introduces the theoretical results establishing these properties. If V is a set of indexed variables of the form $\{X_1, \ldots, X_n\}$ we

use the notation $\theta(V)$ to indicate the sequence $\theta(X_1), \ldots, \theta(X_n)$. In these results it is implicitly assumed that there is a bijective mapping f from XML format to the datatype `xml` in $\mathcal{TOY}$. Also, variables in XQuery $\$x_i$ are assumed to be represented in $\mathcal{TOY}$ as X_i and conversely. However, in order to simplify the presentation, we omit the explicit mention to f and to f^{-1}.

Lemma 1. *Let P be a $\mathcal{TOY}$ program, Q' an SXQ query, and Q, p such that $Q \equiv Q'_{|p}$. Define $V = Rel(Q', p)$ (see Definition 2), and $k = |V|$. Let θ be a substitution such that $\mathcal{P} \vdash (\texttt{sxq}\ Q\theta == t)$ for some pattern t.*
Then $[\![Q]\!]_k(\mathcal{F}, [\theta(V)]) :=^* (\mathcal{F}', L)$, for some forests $\mathcal{F}, \mathcal{F}'$ and with L verifying $t \in L$.

The theorem that establishes the correctness of the approach is an easy consequence of the Lemma.

Theorem 1. *Let P be the $\mathcal{TOY}$ program of Figure 4, Q an SXQ query, t a $\mathcal{TOY}$ pattern, and θ a substitution such that $\mathcal{P} \vdash (\texttt{sxq}\ Q\theta == t)$ for some θ. Then $[\![Q]\!]_0(\emptyset, []) :=^* (\mathcal{F}, L)$, for some forest $\mathcal{F}$, and L verifying $t \in L$.*

Proof. In Lemma 1 consider the position $p \equiv \varepsilon$. Then $Q' \equiv Q$, $V = \emptyset$ and $k = 0$. Without loss of generality we can restrict in the conclusion to $\mathcal{F} = \emptyset$, because $\theta(V) = \emptyset$ and therefore $\mathcal{F}$ is not used during the rewriting process. Then the conclusion of the theorem is the conclusion of the lemma.

Thus, our approach is correct. The next Lemma allows us to prove that it is also complete, in the sense that the $\mathcal{TOY}$ program can produce every answer obtained by the XQ operational semantics.

Lemma 2. *Let $\mathcal{P}$ be the $\mathcal{TOY}$ program of Figure 4. Let Q' be a SXQ query and Q, p such that $Q \equiv Q'_{|p}$. Define $V = Rel(Q', p)$ (see Definition 2) and $k = |V|$.*
Suppose that $[\![Q]\!]_k(\mathcal{F}, \overline{e}_k) :=^* (\mathcal{F}', \overline{a}_n)$ for some $\mathcal{F}$, $\mathcal{F}'$, $\overline{e}_k$, $\overline{a}_n$.
Then, for every a_j, $1 \leq j \leq n$, there is a substitution θ such that $\theta(X_i) = e_i$ for $X_i \in V$ and a CRWL-proof proving $\mathcal{P} \vdash \texttt{sxq}\ Q\theta == a_j$.

As in the case of correctness, the completeness theorem is just a particular case of the Lemma:

Theorem 2. *Let $\mathcal{P}$ be the $\mathcal{TOY}$ program of Figure 4. Let Q be a SXQ query and suppose that $[\![Q]\!]_k(\emptyset, []) :=^* (\mathcal{F}, \overline{a}_n)$ for some $\mathcal{F}, \overline{a}_n$. Then for every a_j, $1 \leq j \leq n$, there is $\mathcal{P} \vdash (\texttt{sxq}\ Q)\theta == a_j$ for some substitution θ.*

Proof. As in Theorem 1, suppose $p \equiv \varepsilon$ and thus $Q' \equiv Q$. Then $V = \emptyset$ and $k = 0$. Then, if $[\![Q]\!]_0(\emptyset, \emptyset) :=^* (\mathcal{F}, \overline{a}_n)$ it is easy to check that $[\![Q]\!]_0(\mathcal{F}', \emptyset) :=^* (\mathcal{F}, \overline{a}_n)$ for any $\mathcal{F}'$. Then the conclusion of the lemma is the same as the conclusion of the Theorem.

The proofs of Lemmata 1 and 2 can be found in [2].

5 Application: Test Case Generation

In this section we show how an embedding of SXQ into $\mathcal{TOY}$ can be used for obtaining test-cases for the queries. For instance, consider the erroneous query of the next example.

Example 5. Suppose that the user also wants to include the publisher of the book among the data obtained in Example 1. The following query tries to obtain this information:

```
Q = for $b in doc("bib.xml")/bib/book,
        $r in doc("reviews.xml")/reviews/entry,
        where $b/title = $r/title
        for $booktitle in $r/title,
            $revtext in $r/review,
            $publisher in $r/publisher
        return  <rev> $booktitle $publisher $revtext </rev>
```

However, there is an error in this query, because in `$r/publisher` the variable `$r` should be `$b`, since the publisher is in the document *"bib.xml"*, not in *"reviews.xml"*. The user does not notice that there is an error, tries the query (in $\mathcal{TOY}$ or in any XQuery interpreter) and receives an empty answer.

In order to check whether a query is erroneous, or even to help finding the error, it is sometimes useful to have test-cases, i.e., XML files which can produce some answer for the query. Then the test-cases and the original XML documents can be compared, and this can help finding the error. In our setting, such test-cases are obtained for free, thanks to the generate and test capabilities of logic programming. The general process can be described as follows:

1. Let `Q'` be the translation `parse_xquery Q` of query `Q` into $\mathcal{TOY}$.
2. Let $F_1, \ldots, F_k$ be the names of the XML documents occurring in `Q'`. That is, for each F_i, $1 \leq i \leq k$, there is an occurrence of an expression of the form `load_xml_file(`F_i`)` in `Q'` (which corresponds to expressions `doc(`F_i`)` in `Q`). Let `Q''` be the result of replacing each `doc(`F_i`)` expression by a new variable D_i, for $i = 1 \ldots k$.
3. Let ``expected.xml'' be a document containing an expected answer for the query `Q`.
4. Let `E` the expression `Q''==load_doc ``expected.xml''`.
5. Try the goal
   `G` $\equiv$ `E, write_xml_file` D_1 F'_1`, ..., write_xml_file` D_k F'_k

The idea is that the goal `G` looks for values of the logic variables D_i fulfilling the strict equality. The result is that after solving this goal, the D_i variables contain XML documents that can produce the expected answer for this query. Then each document is saved into a new file with name F'_i. For instance F'_i can consist of the original name F_i preceded by some suitable prefix *tc*. The process can be automatized, and the result is the code of Figure 5.

```
prepareTC (xp E)            = (xp E',L)
                              where (E',L) = prepareTCPath E
prepareTC (xmlExp X)        = (xmlExp X, [])
prepareTC (comp Q1 Q2)      = (comp Q1' Q2', L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
prepareTC (xfor X Q1 Q2)    = (xfor X Q1' Q2', L1++L2)
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2
prepareTC (xif (Q1:=Q2) Q3) = (xif (Q1':=Q2') Q3',L1++(L2++L3))
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
                                     (Q3',L3) = prepareTC Q3
prepareTC (xif (cond Q1) Q2) = (xif (cond Q1) Q2, L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2

prepareTCPath (var X)   = (var X, [])
prepareTCPath (X :/ S)  = (X :/ S, [])
prepareTCPath (doc F S) = (A :/ S, [write_xml_file A ("tc"++F)])

generateTC Q F = true <==  sxq Qtc == load_doc F, L==_
              where (Qtc,L) = prepareTC Q
```

Fig. 5. $\mathcal{TOY}$ transformation rules for SXQ

The code uses the list concatenation operator ++ which is defined in $\mathcal{TOY}$ as usual in functional languages such as Haskell. It is worth observing that if there are no test-case documents that can produce the expected result for the query, the call to `generateTC` will loop. The next example shows the generation of test-cases for the wrong query of Example 5.

Example 6. Consider the query of Example 5 , and suppose the user writes the following document "`expected.xml`":

```
<rev>
 <title>Some title</title>
 <review>The review</review>
 <publisher>Publisher</publisher>
</rev>
```

This is a possible expected answer for the query. Now we can try the goal:

```
Toy> Q == parse_xquery "for....", R == generateTC Q "expected.xml"
```

The first strict equality parses the query, and the second one generates the XML documents which constitute the test cases. In this example the test-cases obtained are:

```
% bibtc.xml
<bib>
 <book>
  <title>Some title</title>
 </book>
</bib>
```

```
% revtc.xml
<reviews>
 <entry>
  <title>Some title</title>
  <review>The review </review>
  <publisher>Publisher</publisher>
 </entry>
</reviews>
```

By comparing the test-case "`revtc.xml`" with the file "`reviews.xml`" we observe that the publisher is not in the reviews. Then it is easy to check that in the query the publisher is obtained from the reviews instead of from the *bib* document, and that this constitutes the error.

6 Conclusions

The paper shows the embedding of a fragment of the XQuery language for querying XML documents into the functional-logic language $\mathcal{TOY}$. Although only a small subset of XQuery consisting of *for*, *where/if* and *return* statements has been considered, the users of $\mathcal{TOY}$ can now perform simple queries such as *join* operations. The formal definition of the embedding allows us to prove the soundness of the approach with respect to the operational semantics of XQuery. The proposal respects the declarative nature of $\mathcal{TOY}$, exploiting its non-deterministic nature for obtaining the different results produced by XQuery expressions. An advantage of this approach with respect to the use of lists usually employed in functional languages is that our embedding allows the user to generate test-cases automatically when possible, which is useful for testing the query, or even for helping to find the error in the query. An extended version of this paper, including the proofs of the theoretical results and more detailed explanations about how to install $\mathcal{TOY}$ and run the prototype can be found in [2].

The most obvious future work would be introducing the *let* statement, which presents two novelties. The first is that they are *lazy*, that is, they are not evaluated if they are not required by the result. This part is easy to fulfill since we are in a lazy language. In particular, they could be introduced as local definitions (*where* statements in $\mathcal{TOY}$). The second novelty is more difficult to capture, and it is that the variables introduced by *let* represent an XML sequence. The natural representation in $\mathcal{TOY}$ would be a list, but the non-deterministic nature of our proposal does not allow us to collect all the results provided by an expression in a declarative way. A possible idea would be to use the functional-logic language Curry [8] and its encapsulated-search [10], or even the non-declarative *collect* primitive included in $\mathcal{TOY}$. In any case, this will imply a different theoretical framework and new proofs for the results. A different line for future work is the use of test cases for finding the error in the query using some variation of declarative debugging [14] that could be applied to this setting.

References

1. Almendros-Jiménez, J.M.: An Encoding of XQuery in Prolog. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 145–155. Springer, Heidelberg (2009)
2. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Declarative Embedding of XQuery in a Functional-Logic Language. Technical Report SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid (2011), http://gpd.sip.ucm.es/rafa/xquery/
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1999)
4. Benedikt, M., Koch, C.: From XQuery to relational logics. ACM Trans. Database Syst. 34, 25:1–25:48 (2009)
5. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Integrating XPath with the Functional-Logic Language Toy. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 145–159. Springer, Heidelberg (2011)
6. Fegaras, L.: Propagating updates through XML views using lineage tracing. In: IEEE 26th International Conference on Data Engineering (ICDE), pp. 309–320 (March 2010)
7. González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M.: A Rewriting Logic for Declarative Programming. In: Riis Nielson, H. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 156–172. Springer, Heidelberg (1996)
8. Hanus, M.: Curry: An Integrated Functional Logic Language (version 0.8.2 March 28, 2006) (2003), http://www.informatik.uni-kiel.de/~mh/curry/
9. Hanus, M.: Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel (2011)
10. Hanus, M., Steiner, F.: Controlling Search in Declarative Programs. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 374–390. Springer, Heidelberg (1998)
11. Fraguas, F.J.L., Hernández, J.S.: $\mathcal{TOY}$: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
12. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking Forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
13. Seipel, D., Baumeister, J., Hopfner, M.: Declaratively Querying and Visualizing Knowledge Bases in XML. In: Seipel, D., Hanus, M., Geske, U., Bartenstein, O. (eds.) INAP/WLP 2004. LNCS (LNAI), vol. 3392, pp. 16–31. Springer, Heidelberg (2005)
14. Shapiro, E.: Algorithmic Program Debugging. ACM Distiguished Dissertation. MIT Press (1982)
15. W3C. Extensible Markup Language (XML) (2007)
16. W3C. XML Path Language (XPath) 2.0 (2007)
17. W3C. XML Query Use Cases (2007), http://www.w3.org/TR/xquery-use-cases/
18. W3C. XQuery 1.0: An XML Query Language (2007)
19. W3C. XQuery 1.0 and XPath 2.0 Formal Semantics, 2nd edn. (2010), http://www.w3.org/TR/xquery-semantics/
20. Walmsley, P.: XQuery. O'Reilly Media, Inc. (2007)

Declarative Debugging of Wrong and Missing Answers for SQL Views*

Rafael Caballero[1], Yolanda García-Ruiz[1], and Fernando Sáenz-Pérez[2]

[1] Departamento de Sistemas Informáticos y Computación
[2] Dept. de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
{rafa,fernan}@sip.ucm.es, ygarciar@fdi.ucm.es

Abstract. This paper presents a debugging technique for diagnosing errors in SQL views. The debugger allows the user to specify the error type, indicating if there is either a missing answer (a tuple was expected but it is not in the result) or a wrong answer (the result contains an unexpected tuple). This information is employed for slicing the associated queries, keeping only those parts that might be the cause of the error. The validity of the results produced by sliced queries is easier to determine, thus facilitating the location of the error. Although based on the ideas of declarative debugging, the proposed technique does not use computation trees explicitly. Instead, the logical relations among the nodes of the trees are represented by logical clauses that also contain the information extracted from the specific questions provided by the user. The atoms in the body of the clauses correspond to questions that the user must answer in order to detect an incorrect relation. The resulting logic program is executed by selecting at each step the unsolved atom that yields the simplest question, repeating the process until an erroneous relation is detected. Soundness and completeness results are provided. The theoretical ideas have been implemented in a working prototype included in the Datalog system DES.

1 Introduction

SQL (Structured Query Language [18]) is a language employed by relational database management systems. In particular, the SQL select statement is used for querying data from databases. Realistic database applications often contain a large number of tables, and in many cases, queries become too complex to be coded by means of a single select statement. In these cases, SQL allows the user to define *views*. A SQL view can be considered as a virtual table, whose content is obtained executing its associated SQL select query. View queries can rely on previously defined views, as well as on database tables. Thus, complex queries can be decomposed into sets of correlated views. As in other programming paradigms, views can have bugs. However, we cannot infer that a view is

* Work partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502).

T. Schrijvers and P. Thiemann (Eds.): FLOPS 2012, LNCS 7294, pp. 73–87, 2012.

incorrectly defined when it computes an unexpected result, because it might be receiving erroneous input data from other database tables or views. Given the high-abstraction level of SQL, usual techniques like trace debugging are difficult to apply. Some tools as [2,13] allow the user to trace and analyze the stored SQL procedures and user defined functions, but they are of little help when debugging systems of correlated views. *Declarative Debugging*, also known as *algorithmic debugging*, is a technique applied successfully in (constraint) logic programming [16], functional programming [12], functional-logic programming [5], and in deductive database languages [3]. The technique can be described as a general debugging schema [11] which starts when an *initial error symptom* is detected by the user, which in our case corresponds to an unexpected result produced by a view. The debugger automatically builds a tree representing the erroneous computation. In SQL, each node in the tree contains information about both a relation, which is a table or a view, and its associated computed result. The root of the tree corresponds to the initial view. The children of a node correspond to the relations (tables or views) occurring in the definition of its associated query. After building the tree, it is *navigated* by the debugger, asking to the user about the validity of some nodes. When a node contains the expected result, it is marked as *valid*, and otherwise it is marked as *nonvalid*. The goal of the debugger is to locate a *buggy node*, which is a nonvalid node with valid children. It can be proved that each buggy node in the tree corresponds to either an erroneously defined view, or to a database table containing erroneous data. A debugger based on these ideas was presented in [4]. The main criticism that can be leveled at this proposal is that it can be difficult for the user to check the validity of the results. Indeed, even very complex database queries usually are defined by a small number of views, but the results returned by these views can contain hundreds or thousands of tuples. The problem can be easily understood by considering the following example:

Example 1. The loyalty program of an academy awards an intensive course for students that satisfy the following constraints:
- The student has completed the basic level course (level = 0).
- The student has not completed an intensive course.
- To complete an intensive course, a student must either pass the *all in one* course, or the three initial level courses (levels 1, 2 and 3).

The database schema includes three tables: *courses(id,level)* contains information about the standard courses, including their identifier and the course level; *registration(student,course,pass)* indicates that the *student* is in the *course*, with *pass* taking the value *true* if the course has been successfully completed; and the table *allInOneCourse(student,pass)* contains information about students registered in a special intensive course, with *pass* playing the same role as in *registration*. Figure 1 contains the SQL views selecting the award candidates. The first view is *standard*, which completes the information included in the table *Registration* with the course level. The view *basic* selects those *standard* students that have passed a basic level course (level 0). View *intensive* defines as intensive students those in the *allInOneCourse* table, together with the students that have

```
create or replace view standard(student, level, pass) as
   select R.student, C.level, R.pass
   from courses C, registration R
   where C.id = R.course;

create or replace view basic(student) as
   select S.student
   from standard S
   where S.level = 0 and S.pass;

create or replace view intensive(student) as
   (select A.student from allInOneCourse A  where A.pass)
   union
   (select a1.student
    from standard A1, standard A2, standard A3
    where A1.student = A2.student and A2.student = A3.student
          and
          a1.level = 1  and a2.level = 2 and a3.level = 3);

create or replace view awards(student) as
  select student from  basic
  where student not in (select student from  intensive);
```

Fig. 1. Views for selecting award winner students

completed the three initial levels. However, this view definition is erroneous: we have forgotten to check that the courses have been completed (flag *pass*). Finally, the main view *awards* selects the students in the basic but not in the intensive courses. Suppose that we try the query select * from awards;, and that in the result we notice that the student *Anna* is missing. We know that *Anna* completed the basic course, and that although she registered in the three initial levels, she did not complete one of them, and hence she is not an intensive student. Thus, the result obtained by this query is nonvalid. A standard declarative debugger using for instance a top-down strategy [17], would ask first about the validity of the contents of *basic*, because it is the first child of *awards*. But suppose that *basic* contains hundreds of tuples, among them one tuple for *Anna*; in order to answer that *basic* is valid, the user must check that *all* the tuples in the result are the expected ones, and that there is no missing tuple. Obviously, the question about the validity of *basic* becomes practically impossible to answer.

The main goal of this paper is to overcome or at least to reduce this drawback. This is done by asking for more specific information from the user. The questions are now of the type "Is there a missing answer (that is, a tuple is expected but it is not there) or a wrong answer (an unexpected tuple is included in the result)?" With this information, the debugger can:
- Reduce the number of questions directed at the user. Our technique considers only those relations producing/losing the wrong/missing tuple. In the example, the debugger checks that *Anna* is in *intensive*. This means that either *awards* is

erroneous or *Anna* is wrong in *intensive*. Consequently, the debugger disregards *basic* as a possible error source, reducing the number of questions.
- The questions directed at the user about the validity in the children nodes can be simplified. For instance, the debugger only considers those tuples that are needed to produce the wrong or missing answer in the parent. In the example, the tool would ask if *Anna* was expected in *intensive*, without asking for the validity of the rest of the tuples in this view.

Another novelty of our approach is that we represent the computation tree using Horn clauses, which allows us to include the information obtained from the user during the session. This leads to a more flexible and powerful framework for declarative debugging that can now be combined with other diagnosis techniques. We have implemented these ideas in the system DES [14,15].

Next section presents some basic concepts used in the rest of the paper. Section 3 introduces the debugging algorithm that constitutes the main contribution of our paper, including the theoretical results supporting the proposal. The implementation is discussed in Section 4. Finally, Section 5 presents the conclusions and proposes future work.

2 Preliminaries

This section introduces some basic concepts about databases, interpretations and types of errors which are used in the rest of the paper. A *table schema* has the form $T(A_1, \ldots, A_n)$, with T being the table name and A_i the attribute names for $i = 1 \ldots n$. We refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type. An *instance* of a table schema $T(A_1, \ldots, A_n)$ is determined by its particular *tuples*. Each tuple contains values of the correct type for each attribute in the table schema. The notation t_i represents the i-th element in the tuple. In our setting, *partial* tuples are tuples that might contain the special symbol $\perp$ in some of their components. The set of defined positions of a partial tuple s, *def(s)*, is defined by $p \in \mathit{def(s)} \Leftrightarrow s_p \neq \perp$. Tuples s with $\mathit{def(s)} = \emptyset$ are *total* tuples. Membership with partial tuples is defined as follows: if s is a partial tuple, and S a set of total tuples with the same arity as s, we say that $s \in S$ if there is a tuple $u \in S$ such that $u_p = s_p$ for every $p \in (\mathit{def(s)} \cap \mathit{def(u)})$. Otherwise we say that $s \notin S$.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables and $\mathcal{V}$ a finite set of views. *Views* can be thought of as new tables created dynamically from existing ones by using a SQL query. The general syntax of a SQL view is: create view $\mathsf{V}(A_1, \ldots, A_n)$ as Q, with Q a query and $V.A_1, \ldots V.A_n$ the names of the view attributes. A *database instance* d of a database schema is a set of table instances, one for each table in $\mathcal{T}$. The notation $d(T)$ represents the instance of a table T in d. The *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in its query. Figure 2 shows the dependency tree for our running example. In general, the name *relation* refers to either a table or a view. The syntax of SQL queries can be found in [18]. We distinguish between *basic queries* and *compound*

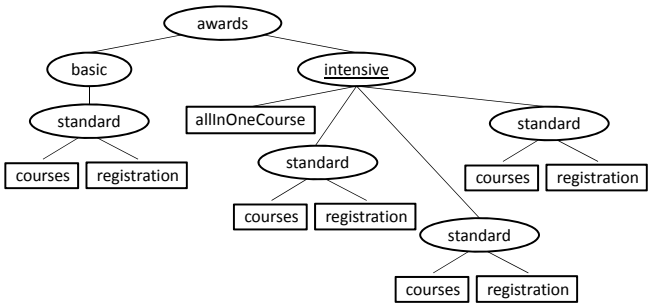


Fig. 2. Example of Computation Tree

queries. A basic query Q contains both select and from sections in its definition with the optional where, group by and having sections. For instance, the query associated to the view *standard* in the example of Figure 1 is a basic query. A compound query Q combines the results of two queries Q_1 and Q_2 by means of set operators union [all], except [all] or intersect [all] (the keyword all indicates that the result is a multiset). For convenience, our debugger transforms basic queries into compound queries when necessary. We also assume that the queries defining views do not contain subqueries. Translating queries into equivalent definitions without subqueries is a well-known transformation (see for instance [6]). For instance, the query defining view *awards* in the Figure 1 is transformed into:

```
select student from basic
except
select student from intensive;
```

The semantics of SQL assumed in this paper is given by the Extended Relational Algebra (ERA) [10], an operational semantics allowing aggregates, views, and most of the common features of SQL queries. Each relation R is defined as a multiset of tuples. The notation $|R|_t$ refers to the number of occurrences of the tuple t in the relation R, and Φ_R represents the ERA expression associated to a SQL query or view R, as explained in [8]. A query/view usually depends on previously defined relations, and sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on $R_1, \ldots, R_n$. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table. The *computed answer* of an ERA expression Φ_R with respect to some schema instance d is denoted by $\parallel \Phi_R \parallel_d$, where:

- If R is a database table, $\parallel \Phi_R \parallel_d = d(R)$.
- If R is a database view or a query and $R_1, \ldots, R_n$ the relations defined in R, then $\parallel \Phi_R \parallel_d = \Phi_R(\parallel \Phi_{R_1} \parallel_d, \ldots, \parallel \Phi_{R_n} \parallel_d)$.

The parameter d indicating the database instance is omitted in the rest of the presentation whenever is clear from the context.

Queries are executed by SQL systems. The answer for a query Q in an implementation is represented by $\mathcal{SQL}(Q)$. The notation $\mathcal{SQL}(R)$ abbreviates $\mathcal{SQL}(\mathsf{select\ *\ from}\ R)$. In particular, we assume in this paper the existence of *correct* SQL implementations. A *correct* SQL implementation verifies that $\mathcal{SQL}(Q) = \parallel \Phi_Q \parallel$ for every query Q. In the rest of the paper, D represents the database schema, d the current instance of D, and R a relation defined in D. We assume that the user can check if the computed answer for a relation matches its intended answer. The *intended answer* for a relation R w.r.t. d, is a multiset denoted as $\mathcal{I}(R)$ containing the answer that the user expects for the query $\mathsf{select\ *\ from\ R}$ in the instance d. This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging. We say that $\mathcal{SQL}(R)$ is an *unexpected answer* for a query R if $\mathcal{I}(R) \neq \mathcal{SQL}(R)$. An unexpected answer can contain either a *wrong tuple*, when there is some tuple t in $\mathcal{SQL}(R)$ s.t. $|\mathcal{I}(R)|_t < |\mathcal{SQL}(R)|_t$, or a *missing tuple*, when there is some tuple t in $\mathcal{I}(R)$ s.t. $|\mathcal{I}(R)|_t > |\mathcal{SQL}(R)|_t$. For instance, the intended answer for *awards* contains *Anna* once, which is represented as $|\mathcal{I}(\mathsf{awards})|_{(\mathsf{Anna})} = 1$. However, the computed answer does not include this tuple: $|\mathcal{SQL}(\mathsf{awards})|_{(\mathsf{Anna})} = 0$. Thus, *('Anna')* is a missing tuple for *awards*. In order to define the key concept of erroneous relation we need the following auxiliary concept. Let R be either a query or a relation. The *expectable* answer for R w.r.t. d, $\mathcal{E}(R)$, is defined as:

1. If R is a table, $\mathcal{E}(R) = d(R)$, with d the database schema instance.
2. If R is a view, then $\mathcal{E}(R) = \mathcal{E}(Q)$, with Q the query defining R.
3. If R is a query $\mathcal{E}(R) = \Phi_R(\mathcal{I}(R_1), \ldots, \mathcal{I}(R_n))$ with $R_1, \ldots, R_n$ the relations occurring in R.

Thus, in the case of a table, the expectable answer is its instance. In the case of a view V, the expectable answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of V contain the intended answers. Then, $\mathcal{I}(R) \neq \mathcal{E}(R)$ indicates that R does not compute its intended answer, even assuming that all the relations it depends on contain their intended answers. Such relation is called *erroneous*. In our running example, the real cause of the missing answer for the view *awards* is the erroneous definition of the view *intensive*.

3 Debugging Algorithm

In this section we present the algorithm that defines our debugging technique, describing the purpose of each function. Although the process is based on the ideas of declarative debugging, this proposal does not use computation trees explicitly. Instead, our debugger represents computation trees by means of Horn clauses, denoted as $\mathsf{H} \leftarrow \mathsf{C}_1, \ldots, \mathsf{C}_n$, where the comma represents the conjunction, and $\mathsf{H}, \mathsf{C}_1, \ldots, \mathsf{C}_n$ are positive atoms. As usual, a *fact* H stands for the clause $\mathsf{H} \leftarrow \mathsf{true}$. Next, we describe the functions that define the algorithm, although the code of some basic auxiliary functions is omitted for the sake of space. This is the case of *getSelect*, *getFrom*, *getWhere*, and *getGroupBy* which return the different

Code 1. debug(V)

```
Input: V: view name
Output: A list of buggy views
 1: A := askOracle(all V)
 2: P := initialSetOfClauses(V, A)
 3: while getBuggy(P)=[] do
 4:    LE := getUnsolvedEnquiries(P)
 5:    E := chooseEnquire(LE)
 6:    A := askOracle(E)
 7:    P := P ∪ processAnswer(E,A)
 8: end while
 9: return (getBuggy(P))
```

sections of a SQL query. In *getFrom*, we assume that every relations has an alias. The result is a sequence of elements of the form *R as R'*. A Boolean expression like *getGroupBy(Q)*=[] is satisfied if the query Q has no group by section. Function *getRelations(R)* returns the set of relations involved in R. It can be applied to queries, tables and views: if R is a table, then *getRelations(R)* = $\{R\}$, if R is a query, then *getRelations(R)* is the set of relations occurring in the definition of the query, and if R is a view, then *getRelations(R)* = *getRelations(Q)*, with Q the query defining R. The function *generateUndefined(R)* generates a tuple whose arity is the number of attributes in R containing only undefined values $(\perp, \ldots, \perp)$.

The general schema of the algorithm is summarized in the code of function *debug* (Code 1). The debugger is started by the user when an unexpected answer is obtained as computed answer for some SQL view V. In our running example, the debugger is started with the call *debug(awards)*. Then, the algorithm asks the user about the type of error (line 1). The answer A can be simply *valid*, *nonvalid*, or a more detailed explanation of the error, like *wrong(t)* or *missing(t)*, indicating that t is a wrong or missing tuple respectively. In our example, A takes the initial value *missing(('Anna'))*. During the debugging process, variable P keeps a list of Horn clauses representing a logic program. The initial list of clauses P is generated by the function *initialSetofClauses* (line 2). The purpose of the main loop (lines 3-8) is to add information to the program P, until a buggy view can be inferred. The function *getBuggy* returns the list of all the relations R such that *buggy(R)* can be proven w.r.t. the logic program P. The clauses in P contain enquiries that might imply questions to the user. Each iteration of the loop represents the election of an enquiry in a body atom whose validity has not been established yet (lines 4-5). Then, an enquiry about the result of the query is asked to the user (line 6). Finally, the answer is processed (line 7). Next, we explain in detail each part of this main algorithm.

Code 2 corresponds to the initialization process of line 2 from Code 1. The function *initialSetofClauses* gets as first input parameter the initial view V. This view has returned an unexpected answer, and the input parameter A contains the explanation. The output of this function is a set of clauses representing the

Code 2. initialSetOfClauses(V,A)

Input: V: view name, A: answer
Output: A set of clauses

```
1: P := ∅
2: P := initialize(V)
3: P := P ∪ processAnswer((all V), A)
4: return P
```

initialize(R)
Input: R: relation
Output: A set of clauses

```
1: P := createBuggyClause(R)
2: for each R_i in getRelations(R) do
3:     P := P ∪ initialize(R_i)
4: end for
5: return P
```

createBuggyClause(V)
Input: V: view name
Output: A Horn clause

```
1: [R_1, ..., R_n] := getRelations(V)
2: return { buggy(V)← state((all V), nonvalid),
                      state((all R_1), valid), ..., state((all R_n), valid)). }
```

logic relations that define possible buggy relations with predicate *buggy*. Initially it creates the empty set of clauses and then it calls the function *initialize* (line 2), a function that traverses recursively all the relations involved in the definition of the initial view V, calling *createBuggyClause* with V as input parameter. *createBuggyClause* adds a new clause indicating the enquiries that must hold in order to consider V as incorrect: it must be nonvalid, and all the relations it depends on must be valid. Next is part of the initial set of clauses generated for the running example of this paper:

```
buggy(awards)    :- state(all(awards),nonvalid),
                    state(all(basic),valid), state(all(intensive),valid).
buggy(basic)     :- state(all(basic),nonvalid), state(all(standard),valid).
buggy(intensive) :- state(all(intensive),nonvalid),
                    state(all(allInOneCourse),valid), state(all(standard),valid).
...
```

The correlation between these clauses and the dependency tree is straightforward. Finally, in line 3, function *processAnswer* incorporates the information that can be extracted from A into the program P. The information about the validity/nonvalidity of the results associated to enquiries is represented in our setting with predicate *state*. The first parameter is an enquiry E, and the second one can be either *valid* or *nonvalid*. Enquiries can be of any of the following forms: *(all R)*, *(s ∈ R)*, or *(R' ⊆ R)* with R, R' relations, and s a tuple with the same schema as relation R. Each enquiry E corresponds to a specific question with a possible set of answers and an associated complexity $\mathcal{C}(E)$:
- If $E \equiv$ *(all R)*. Let $\mathsf{S} = \mathcal{SQL}(\mathsf{R})$. The associated question asked to the user is *"Is S the intended answer for R?"* The answer can be either *yes* or *no*. In the case of *no*, the user is asked about the type of the error, *missing* or *wrong*, giving the possibility of providing a witness tuple t. If the user provides this

Code 3. processAnswer(E,A)

```
Input: E: enquiry, A: answer obtained for the enquiry
Output: A set of new clauses
 1: if A ≡ yes then
 2:    P := {state(E,valid).}
 3: else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
 4:    P := {state(E,nonvalid).}
 5: end if
 6: if E ≡ (s ∈ R) then
 7:    if (s ∈ SQL(R) and A ≡ no) then
 8:       P:= P ∪ processAnswer((all R),wrong(s))
 9:    else if (s∉ SQL(R) and A ≡ yes) then
10:       P:= P ∪ processAnswer((all R),missing(s))
11:    end if
12: else if E ≡ (V ⊆ R) and (A ≡ wrong(s)) then
13:    P:= P ∪ processAnswer((all R), A)
14: else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t)) then
15:    Q := SQL query defining V
16:    P := P ∪ slice(V,Q,A)
17: end if
18: return P
```

information, the answer is changed to *missing(t)* or *wrong(t)*, depending on the type of the error. We define $\mathcal{C}(E) = |S|$, with $|S|$ the number of tuples in S.

-If $E \equiv$ *(R' $\subseteq$ R)*. Let $S = \mathcal{SQL}$*(R')*. Then the associated question is *"Is S included in the intended answer for R?"* As in the previous case the answer allowed can be *yes* or *no*. In the case of *no*, the user can point out a wrong tuple $t \in S$ and the answer is changed to *wrong(t)*. $\mathcal{C}(E) = |S|$ as in the previous case.

- If $E \equiv$ *(s ∈ R)*. The question is *"Does the intended answer for R include a tuple s?"* The possible answer can be *yes* or *no*. No further information is required from the user. In this case $\mathcal{C}(E) = 1$, because only one tuple must be considered.

In the case of *wrong*, the user typically points to a tuple in the result R. In the case of *missing*, the tuple must be provided by the user, and in this case *partial* tuples, i.e., tuples including some undefined attributes are allowed. The answer *yes* corresponds to the state *valid*, while the answer *no* corresponds to *nonvalid*. An atom *state(q,s)* occurring in a clause body, is a *solved enquiry* if the logic program P contains at least one fact of the form *state(q, valid)* or *state(q, nonvalid)*, that is, if the enquiry has been already solved. The atom is called an *unsolved enquiry* otherwise. The function *getUnsolvedEnquiries* (see line 4 of Code 1) returns in a list all the unsolved enquiries occurring in P. The function *chooseEnquiry* (line 5, Code 1) chooses one of these enquiries according to some criteria. In our case we choose the enquiry E that implies the smaller complexity value $\mathcal{C}(E)$, although other more elaborated criteria could be defined without affecting the theoretical results supporting the technique. Once the enquiry has been chosen, Code 1 uses the function *askOracle* (line 6) in order to ask for the

associated question, returning the answer of the user. We omit the definitions of these simple functions for the sake of space.

The code of function *processAnswer* (called in line 7 of Code 1), can be found in Code 3. The first lines (1-5) introduce a new logic fact in the program with the state that corresponds to the answer A obtained for the enquiry E. In our running example, the fact *state(all(awards), nonvalid)* is added to the program. The rest of the code distinguishes several cases depending on the form of the enquiry and its associated answer. If the enquiry is of the form *(*$s \in R$*)* with answer *no* (meaning $s \notin \mathcal{I}(R)$), and the debugger checks that the tuple s is in the computed answer of the view R (line 7), then s is wrong in the relation R. In this case, the function *processAnswer* is called recursively with the enquiry *(all R)* and *wrong(s)* (line 8). If the answer is *yes* and the debugger checks that s does not belong to the computed answer of R (line 10), then s is missing in the relation R. For enquiries of the form *(*$V \subseteq R$*)* and answer *wrong(s)*, it can be ensured that s is wrong in R (line 13). If the enquiry is *(all V)* for some view V, and with an answer including either a wrong or a missing tuple, the function *slice* (line 16) is called. This function exploits the information contained in the parameter A (*missing(t)* or *wrong(t)*) for slicing the query Q in order to produce, if possible, new clauses which will allow the debugger to detect incorrect relations by asking simpler questions to the user. The implementation of *slice* can be found in Code 4. The function receives the view V, a subquery Q, and an answer A as

Code 4. slice(V,Q,A)

```
Input: V: view name, Q: query, A: answer
Output: A set of new clauses
 1: P := ∅;    S= SQL(Q);    S1= SQL(Q1);    S2= SQL(Q2)
 2: if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
       (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2) then
 3:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 4:    if |S2|t = |S|t then  P:= P ∪ slice(V, Q2, A)
 5: else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2  then
 6:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 7:    if Q ≡ Q1 except Q2 and t ∈ S2 then P :=P∪ slice(V,Q2,wrong(t))
 8: else if basic(Q) and groupBy(Q)=[] then
 9:    if A ≡ missing(t) then P := P ∪ missingBasic(V, Q, t)
10:    else if A ≡ wrong(t) then P := P ∪ wrongBasic(V, Q, t)
11: end if
12: return  P
```

parameters. Initially, Q is the query defining V, and A the user answer, but this situation can change in the recursive calls. The function distinguishes several particular cases:

- The query Q combines the results of Q_1 and Q_2 by means of either the operator union or union all, and A is *wrong(t)* (first part of line 2). Then query Q produces too many copies of t. Then, if any Q_i produces as many copies of t as Q, we can

blame Q_i as the source of the excessive number of t's in the answer for V (lines 3 and 4). The case of subqueries combined by the operator intersect [all], with $A \equiv$ *missing(t)* is analogous, but now detecting that a subquery is the cause of the scanty number of copies of t in $\mathcal{SQL}(V)$.
- The query Q is of the form Q_1 except [all] Q_2, with $A \equiv$ *missing(t)* (line 5). If the number of occurrences of t in both Q and Q_1 is the same, then t is also missing in the query Q_1 (line 6). Additionally, if query Q is of the particular form Q_1 except Q_2, which means that we are using the difference operator on sets (line 7), then if t is in the result of Q_2 it is possible to claim that the tuple t is wrong in Q_2. Observe that in this case the recursive call changes the answer from *missing(t)* to *wrong(t)*.
- If Q is defined as a basic query without group by section (line 8), then either function *missingBasic* or *wrongBasic* is called depending on the form of A.

Both *missingBasic* and *wrongBasic* can add new clauses that allow the system to infer buggy relations by posing questions which are easier to answer. Function *missingBasic*, defined in Code 5, is called (line 9 of Code 4) when A is *missing(t)*. The input parameters are the view V, a query Q, and the missing tuple t. Notice

Code 5. missingBasic(V,Q,t)

```
Input: V: view name, Q: query, t: tuple
Output: A new list of Horn clauses
 1: P := ∅;    S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
 2: if t ∉ S then
 3:    for (R AS S) in (getFrom(Q)) do
 4:       s = generateUndefined(R)
 5:       for i=1 to length(getSelect(Q)) do
 6:          if t_i ≠⊥and member(getSelect(Q),i) = S.A, A attrib., then s.A = t_i
 7:       end for
 8:       if s ∉ SQL(R) then
 9:          P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
10:       end if
11:    end for
12: end if
13: return  P
```

that Q is in general a component of the query defining V. For each relation R with alias S occurring in the from section, the function checks if R contains some tuple that might produce the attributes of the form $S.A$ occurring in the tuple t. This is done by constructing a tuple s undefined in all its components (line 4) except in those corresponding to the select attributes of the form $S.A$, which are defined in t (lines 5 - 7). If R does not contain a tuple matching s in all its defined attributes (line 8), then it is not possible to obtain the tuple t in V from R. In this case, a buggy clause is added to the program P (line 9) meaning that if the answer to the question *"Does the intended answer for R include a tuple s?"* is *no*, then V is an incorrect relation.

Code 6. wrongBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A set of clauses

```
1: P := ∅
2: F := getFrom(Q)
3: N := length(F)
4: for i=1 to N do
5:    R_i as S_i := member(F,i)
6:    relevantTuples(R_i,S_i,V_i, Q, t)
7: end for
8: P := P ∪ { (buggy(V) ← state((V_1 ⊆ R_1), valid), ..., state((V_n ⊆ R_n), valid).) }
9: return P
```

Code 7. relevantTuples(R_i,R',V,Q,t)

Input: R_i: relation, R': alias,
V: new view name, Q: Query, t: tuple
Output: A new view in the database schema

```
1: Let A_1, ..., A_n be the attributes defining R_i
2: SQL(create view V as
     (select R_i.A_1, ..., R_i.A_n from R_i)
        intersect all
     (select R'.A_1, ..., R'.A_n from getFrom(Q)
     where getWhere(Q) and eqTups(t,getSelect(Q))))
```

eqTups(t,s)

Input: t,s : tuples
Output: SQL condition

```
1: C := true
2: for i=1 to length(t) do
3:    if t_i ≠ ⊥ then
4:       C:= C AND t_i = s_i
5: end for
6: return C
```

The implementation of *wrongBasic* can be found in Code 6. The input parameters are again the view V, a query Q, and a tuple t. In line 1, this function creates an empty set of clauses. In line 2, variable F stands for the set containing all the relations in the from section of the query Q. Next, for each relation $R_i \in F$ (lines 4 - 7), a new view V_i is created in the database schema after calling the function *relevantTuples* (line 6), which is defined in Code 7. This auxiliary view contains only those tuples in relation R_i that contribute to produce the wrong tuple t in V. Finally, a new buggy clause for the view V is added to the program P (line 8) explaining that the relation V is buggy if the answer to the question associated to each enquiry of the form $V_i \subseteq R_i$ is *yes* for $i \in \{1 \ldots n\}$.

The following theoretical results guarantee that the technique is reliable.

Theorem 1. *Let* R *be a relation. Then:*
Correctness: If the call debug(R) *returns a list* L*, then all relation names contained in* L *are erroneous relations.*
Completeness: Let A be the answer obtained after the call to askOracle(all R) *in line 1 of Code 1. If A is of the form* nonvalid, wrong(t) *or* missing(t)*, then the call* debug(R) *(defined in Code 1) returns a list L containing at least one relation.*

Thus, the algorithm always stops pointing to some user view (completeness) which is incorrectly defined (correctness).

4 Implementation

The algorithm presented in Section 3 has been implemented in the Datalog Educational System (DES [14,15]). The debugger is started when the user detects that *Anna* is not among the (large) list of student names produced by view *awards*. The command `/debug_sql` starts the session:

```
1: DES-SQL> /debug_sql awards
2: Info: Debugging view 'awards': {   1 - awards('Carla'), ... }
3: Is this the expected answer for view 'awards'?  m'Anna'
4: Does the intended answer for 'intensive' include  ('Anna')        ? n
5: Does the intended answer for 'standard'  include  ('Anna',1,true) ? y
6: Does the intended answer for 'standard'  include  ('Anna',2,true) ? y
7: Does the intended answer for 'standard'  include  ('Anna',3,false)? y
8: Info: Buggy relation found: intensive
```

The user answer *m'Anna'* in line 3 indicates that *('Anna')* is missing in the view *awards*. In line 4 the user indicates that view *intensive* should not include *('Anna')*. In lines 5, 6, and 7, the debugger asks three simple questions involving the view *standard*. After checking the information for *Anna*, the user indicates that the listed tuples are correct. Then, the tool points out *intensive* as the buggy view, after only five simple questions. Observe that intermediate views can contain hundreds of thousands of tuples, but the slicing mechanism helps to focus only on the source of the error. Next, we describe briefly how these questions have been produced by the debugger.

After the user indicates that *('Anna')* is missing, the debugger executes a call *processAnswer(all(awards),missing((Anna)))*. This implies a call to *slice(awards,* Q_1 except Q_2*, missing(('Anna')))* (line 16 of Code 3). The debugger checks that Q_2 produces *('Anna')* (line 7 of Code 4), and proceeds with the recursive call *slice(awards,* Q_2*, wrong(('Anna')))* with $Q_2 \equiv$ select student from intensive. Query Q_2 is basic, and then the debugger calls *wrongBasic(awards,* Q_2*, ('Anna'))* (line 10 of Code 4)). Function *wrongBasic* creates a view that selects only those tuples from *intensive* producing the wrong tuple *('Anna')* (function *relevantTuples* in Code 7):

```
create view intensive_slice(student) as
(select * from intensive)
intersect all
(select * from intensive I where  I.student = 'Anna');
```

Finally the following buggy clause is added to the program P (line 8, Code 6):

```
buggy(awards) :- state(subset(intensive_slice,intensive),valid).
```

By enabling development listings with the command `/development on`, the logic program is also listed during debugging. The debugger chooses the only body atom in this clause as next unsolved enquiry, because it only contains one tuple. The call to *askOracle* returns *wrong(('Anna'))* (the user answers *'no'* in line 4). Then *processAnswer(subset(intensive_slice,intensive), wrong(('Anna')))* is called, which in turn calls to *processAnswer(all(intensive),wrong(('Anna')))* recursively. Next call is *slice(intensive, Q, wrong(('Anna')))*, with $Q \equiv Q_3$ union

Q_4 the query definition of *intensive* (see Figure 1). The debugger checks that only Q_4 produces *('Anna')* and calls to *slice(intensive, Q_4, wrong(('Anna')))*. Query Q_4 is basic, which implies a call to *wrongBasic(intensive, Q_4, ('Anna'))*. Then *relevantTuples* is called three times, one for each occurrence of the view *standard* in the from section of Q_4, creating new views:

```
create view standard_slicei(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A1.student, A1.level, A1.pass
    from standard as A1,  standard as A2, standard as A3
    where (A1.student = A2.student and A2.student = A3.student
       and A1.level = 1 and A2.level = 2 and A3.level = 3)
       and A1.student = 'Anna');
```

for $i = 1 \ldots 3$. Finally, the clause:

```
buggy(intensive) :- state(subset(standard_slice1,standard),valid),
                    state(subset(standard_slice2,standard),valid),
                    state(subset(standard_slice3,standard),valid).
```

is added to P (line 8, Code 6). Next, the tool selects the unsolved question with less complexity that correspond to the questions of lines 5, 6, and 7, for which the user answer *yes*. Therefore, the clause for buggy(intensive) succeeds and the algorithm finishes.

5 Conclusions

We have presented a new technique for debugging systems of SQL views. Our proposal refines the initial idea presented in [4] by taking into account information about *wrong* and *missing* answers provided by the user. Using a technique similar to dynamic slicing [1], we concentrate only in those tuples produced by the intermediate relations that are relevant for the error. This minimizes the main problem of the technique presented in [4], which was the huge number of tuples that the user must consider in order to determine the validity of the result produced by a relation. Previous works deal with the problem of tracking provenance information for query results [9,7], but to the best of our knowledge, none of them treat the case of missing tuples, which is important in our setting.

The proposed algorithm looks for particular but common error sources, like tuples missed in the from section or in *and* conditions (that is, *intersect* components in our representation). If such shortcuts are not available, or if the user only answers *yes* and *no*, then the tools works as a pure declarative debugger.

A more general contribution of the paper is the idea of representing a declarative debugging computation tree by means of a set of logic clauses. In fact, the algorithm in Code 1 can be considered a general debugging schema, because it is independent of the underlying programming paradigm. The main advantage of this representation is that it allows combining declarative debugging with other diagnosis techniques that can be also represented as logic programs. In our case,

declarative debugging and slicing cooperate for locating an erroneous relation. It would be interesting to research the combination with other techniques such as the use of assertions.

References

1. Agrawal, H., Horgan, J.R.: Dynamic program slicing. SIGPLAN Not. 25, 246–256 (1990)
2. ApexSQL Debug (2011), http://www.apexsql.com/sql_tools_debug.aspx/
3. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Theoretical Framework for the Declarative Debugging of Datalog Programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)
4. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Algorithmic Debugging of SQL Views. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) PSI 2011. LNCS, vol. 7162, pp. 77–85. Springer, Heidelberg (2012)
5. Caballero, R., López-Fraguas, F.J., Rodríguez-Artalejo, M.: Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 170–184. Springer, Heidelberg (2001)
6. Ceri, S., Gottlob, G.: Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. IEEE Trans. Softw. Eng. 11, 324–345 (1985)
7. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. 25, 179–227 (2000)
8. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall PTR, Upper Saddle River (2008)
9. Glavic, B., Alonso, G.: Provenance for nested subqueries. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT 2009, pp. 982–993. ACM, New York (2009)
10. Grefen, P.W., de By, R.A.: A multi-set extended relational algebra: a formal approach to a practical issue. In: ICDE 1994, pp. 80–88. IEEE (1994)
11. Naish, L.: A Declarative Debugging Scheme. Journal of Functional and Logic Programming 3 (1997)
12. Nilsson, H.: How to look busy while being lazy as ever: The implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
13. Rapid SQL Developer Debugger (2011), http://docs.embarcadero.com/products/rapid_sql/
14. Sáenz-Pérez, F.: Datalog Educational System v3.0 (March 2012), http://des.sourceforge.net/
15. Sáenz-Pérez, F., Caballero, R., García-Ruiz, Y.: A Deductive Database with Datalog and SQL Query Languages. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 66–73. Springer, Heidelberg (2011)
16. Shapiro, E.: Algorithmic Program Debugging. ACM Distiguished Dissertation. MIT Press (1982)
17. Silva, J.: A survey on algorithmic debugging strategies. Advances in Engineering Software 42(11), 976–991 (2011)
18. SQL, ISO/IEC 9075:1992, 3rd edn. (1992)

XQuery in the Functional-Logic Language Toy

Jesus M. Almendros-Jiménez[1], Rafael Caballero[2], Yolanda García-Ruiz[2], and Fernando Sáenz-Pérez[3,⋆]

[1] Dpto. de Lenguajes y Computación, Universidad de Almería
[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
[3] Departamento de Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
Spain

Abstract. This paper presents an encoding of the XML query language XQuery in the functional-logic language $\mathcal{TOY}$. The encoding is based on the definition of for-let-where-return constructors by means of $\mathcal{TOY}$ functions, and uses the recently proposed XPath implementation for this language as a basis. XQuery expressions can be executed in $\mathcal{TOY}$ obtaining sequences of XML elements as answers. Our setting exploits the non-deterministic nature of $\mathcal{TOY}$ by retrieving the elements of the XML tree once at a time when necessary. We show that one of the advantages of using a rewriting-based language for implementing XQuery is that it can be used for optimizing XQuery expressions by query rewriting. With this aim, XQuery expressions are converted into higher order patterns that can be analyzed and modified by $\mathcal{TOY}$ functions.

Keywords: Functional-Logic Programming, Non-Deterministic Functions, XQuery, Higher-Order Patterns.

1 Introduction

In the last few years the eXtensible Markup Language XML [33] has become a standard for the exchange of semistructured data. Thus, querying XML documents from different languages has become a convenient feature. XQuery [35,37] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [34], a domain-specific language that has become part of general-purpose languages. Recently, in [10], we have proposed an implementation of XPath in the functional-logic language $\mathcal{TOY}$ [22]. The implementation is based on the definition of XPath constructors by means of $\mathcal{TOY}$ functions. As well, XML documents are represented in $\mathcal{TOY}$ by means of terms, and the basic constructors of XPath: `child`, `self`, `descendant`, etc.

⋆ This work has been supported by the Spanish projects TIN2008-06622-C03-01, TIN2008-06622-C03-03, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502.

H. Kuchen (Ed.): WFLP 2011, LNCS 6816, pp. 35–51, 2011.

are defined as functions that apply to XML terms. The goal of this paper is to extend [10] to XQuery.

The existing XQuery implementations either use functional programming or Relational Database Management Systems (RDBMS's). In the first case, the *Galax* implementation [23] encodes XQuery into *Objective Caml*, in particular, encodes XPath. Since XQuery is a functional language (with some extensions) the main encoding is related with the type system for allowing XML documents and XPath expressions to occur in a functional expression. With this aim, a specific type system for handling XML tags, the hierarchical structure of XML, and sequences of XML items is required. In addition, XPath expressions can be implemented from this representation. There are also proposals for new languages based on functional programming rather than implementing XPath and XQuery. This is the case of *XDuce* [19] and *CDuce* [5,6], which are languages for XML data processing, using regular expression pattern matching over XML trees and subtyping as basic mechanism. There are also proposals around *Haskell* for handling XML documents, such as *HaXML* and *UUXML* [31,4,36,30]. XML types are encoded with Haskell's type classes providing a Haskell library in which XML types are encoded as algebraic datatypes. HXQ [14] is a translator from XQuery to embedded Haskell code, using the Haskell templates. HXQ stores XML documents in a relational database, and translates queries into SQL queries.

This is also followed in some RDBMS XQuery implementations: XML documents are encoded with relational tables, and XPath and XQuery with SQL. The most relevant contribution in this research line is *MonetDB/XQuery* [7]. It consists of the *Pathfinder* XQuery compiler [8] on top of the *MonetDB* RDBMS, although *Pathfinder* can be deployed on top of any RDBMS. MonetDB/XQuery encodes the XML tree structure in a relational table following a pre/post order traversal of the tree (with some variant). XPath can be implemented from such table-based representation, and XQuery by encoding *flwor* expressions into the *relational algebra*, extended with the so-called *loop-lifted staircase join*.

There are also proposals based on logic programming. In most cases, new languages for XML processing are proposed. The *Xcerpt* project [27,9] proposes a pattern and rule-based query language for XML documents, using the so-called *query terms* including logic variables for the retrieval of XML elements. Another contribution to XML processing is the language *XPathLog* (integrated in the the *Lopix* system) [24] which is a *Datalog*-style extension for *XPath* with variable bindings. *XCentric* [13] is an approach for representing and handling XML documents by logic programs, by considering terms with functions of flexible arity and regular types. $XPath^L$ [26] is a logic language based on rules for XML processing including a specific predicate for handling *XPath* expressions in Datalog programs. *FNPath* [29] is also a proposal for using *Prolog* as a query language for XML documents. It maps XML documents to a Prolog Document Object Model (DOM), which can either consist of facts (graph notation) or a term structure (field notation). *FNPath* can evaluate XPath expressions based on that DOM. [2,3] aim to implement XQuery by means of logic programming, providing two

alternatives: a top-down and a bottom-up approaches (the latter in the line of Datalog programs). Finally, some well-known *Prolog* implementations include libraries for loading XML documents, such as *SWI-Prolog* [38] and *Ciao* [12].

In the field of functional-logic languages, [18] proposes a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry [17]. The framework is based on providing operations to describe partial matchings in the data and exploits functional patterns and set functions for the programming tasks.

In functional and functional-logic languages, a different approach is possible: XPath queries can be represented by higher-order functions connected by higher-order combinators. Using this approach, an XPath query becomes at the same time implementation (code) and representation (data term). This is the approach we have followed in our previous work [10]. In the case of XQuery, for-let-where-return constructors can be encoded in $\mathcal{TOY}$, which uses the XPath query language as a basis. XQuery expressions can be encoded by means of (first-order) functions. However, we show that we can also consider XQuery expressions as higher order patterns, in order to manipulate XQuery programs by means of $\mathcal{TOY}$. For instance, we have studied how to transform XQuery expressions into $\mathcal{TOY}$ patterns in order to optimize them. In this paper we follow this idea, which has been used in the past, for instance for defining parsers in functional and functional-logic languages [11,20]. A completely declarative proposal for integrating part of XQuery in $\mathcal{TOY}$ can be found in [1], which restricts itself to the completely declarative features of the language. This implies that the subset of XQuery considered is much narrower than the framework presented here. The advantage of restricting to the purely declarative view is that proofs of correctness and completeness are provided. In this work we take a different point of view, trying to define a more general XQuery framework although using non-purely declarative features as the (meta-)primitive `collect`. Another difference of this work is the use of higher-order patterns for rewriting queries, which was not available in [1].

The specific characteristics of functional-logic languages match perfectly the nature of XQuery queries:

- *Non-deterministic functions* are used to nicely represent the evaluation of an XPath/XQuery query, which consists of fragments of the input XML document. In addition, the `for` constructor of XQuery can be defined with non-deterministic behavior.
- *Logic variables* are employed for instance when obtaining the contents of XPath text nodes, and for solving nested XQuery expressions, capturing the non-deterministic behavior of inner `for` and XPath expressions.
- By defining rules with *higher-order patterns*, XPath/XQuery queries become truly first-class citizens in our setting. In the case of XQuery, this allows us to rewrite queries in order to be optimized. XPath can also be optimized (see [10] for more details).

The rest of the paper is organized as follows. Section 2 briefly introduces the XPath subset presented in [10]. Section 3 defines the encoding of XQuery in

$\mathcal{TOY}$. Section 4 shows how to use $\mathcal{TOY}$ for the optimization of XQuery. Finally, Section 5 presents some conclusions.

2 XPath in $\mathcal{TOY}$

This section introduces the functional-logic language $\mathcal{TOY}$ [22] and the subset of XPath that we intend to integrate with $\mathcal{TOY}$, omitting all the feaures of XPath that are supported by $\mathcal{TOY}$ but not used in this paper, such as filters, abbreviations, attributes and preprocessing of reverse axes. See [10] for a more detailed introduction to XPath in $\mathcal{TOY}$.

2.1 The Functional-Logic Language $\mathcal{TOY}$

All the examples in this paper are written in the concrete syntax of the lazy functional-logic language $\mathcal{TOY}$ [22], but most of the code can be easily adapted to other similar languages as Curry [17]. $\mathcal{TOY}$ is a lazy functional-logic language. A $\mathcal{TOY}$ program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax is similar to the functional language Haskell, except for the capitalization, which follows the approach of Prolog (variables start by uppercase, and other symbols by lowercase[1]). Each rule for a function f has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} = \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns. The overall idea is that a function call $(f\ e_1 \ldots e_n)$ returns an instance $r\theta$ of r, if:

- Each e_i can be reduced to some pattern a_i, $i = 1 \ldots n$, such that $(f\ t_1 \ldots t_n)$ and $(f\ a_1 \ldots a_n)$ are unifiable with most general unifier θ, and
- $u_i\theta$ can be reduced to pattern $s_i\theta$ for each $i = 1 \ldots m$.

Infix operators are also allowed as particular case of program functions. Consider for instance the definitions:

```
infixr 30 /\                infixr 30 \/            infixr 45 ?
false /\ X = false          true \/ X = true        X ? _Y = X
true /\ X = X               false \/ X = X          _X ? Y = Y
```

The /\ and \/ operators represent the standard conjunction and disjunction, respectively, while `?` represents the non-deterministic choice. For instance the infix declaration `infixr 45 ?` indicates that `?` is an infix operator that associates to the right (the r in `infixr`) and that its priority is 35. The priority is used to assume precedences in the case of expressions involving different operators. Computations in $\mathcal{TOY}$ start when the user inputs some goal as

[1] Also, only variables are allowed to start that way. If another identifier has to start with uppercase or underscore, it must be delimited between single quotes.

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

This goal asks $\mathcal{TOY}$ for values of the logical variable R that make true the (strict) equality `1 ? 2 ? 3 ? 4 == R`. This goal yields four different answers `{R` $\mapsto$ `1 }`, `{R` $\mapsto$ `2 }`, `{R` $\mapsto$ `3 }`, and `{R` $\mapsto$ `4 }`. The next function extends the choice operator to lists: `member [X|Xs] = X ? member Xs`. For instance, the goal `member [1,2,3,4] == R` has the same four answers that were obtained by trying `1 ? 2 ? 3 ? 4 == R`.

$\mathcal{TOY}$ is a *typed* language. Types do not need to be annotated explicitly by the user, they are inferred by the system, which rejects ill-typed expressions. However, function type declarations can also be made explicit by the user, which improves the clarity of the program and helps to detect some bugs at compile time. For instance, a function type declaration is: `member :: [A] -> A` which indicates that `member` takes a list of elements of type `A`, and returns a value which must be also of type `A`. As usual in functional programming languages, $\mathcal{TOY}$ allows partial applications in expressions and higher order parameters like `apply F X = F X`. Consider for instance the function that returns the n-th value in a list:

```
nth :: int -> [A] -> A
nth N [X|Xs]  = if N==1 then X else nth (N-1) Xs
```

This function has program arity 2, which means that the program rule is applied when it receives `nth 1 == R1, R1 ["hello","friends"] == R2` and produces the answer `{ R1` $\mapsto$ `(nth 1), R2` $\mapsto$ `"hello" }`. In this solution, `R1` is bound to the partial application `nth 1`. Observe that `R1` has type `([A] -> A)`, and thus it is a *higher-order* variable. Applying `R1` to a list of strings like in the second part of the goal `R1 ["hello","friends"] == R2` 'triggers' the use of the program rule for `nth`. A particularity of $\mathcal{TOY}$ is that partial applications with pattern parameters are also valid patterns. They are called *higher-order patterns*. For instance, a program rule like `foo (apply member) = true` is valid, although `foo (apply member []) = true` is not because `apply member []` is a reducible expression and not a valid pattern. For instance, one could define a function like: `first (nth N) = N==1` because `nth N` is a higher-order pattern. However, a program rule like: `foo (nth 1 [2]) = true` is not valid, because `(nth 1 [2])` is reducible and thus it is not a valid pattern. Higher-order variables and patterns play an important role in our setting.

2.2 Representing XPath Queries

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$, as illustrated next:

```
data xmlNode      = txt     string
                  | comment string
                  | xmlTag  string [xmlAttribute] [xmlNode]
data xmlAttribute = att     string string
```

```
type xml          = xmlNode
type xPath        = xml -> xml
```

Data type `xmlNode` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, comments, and tags (element nodes), each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `xmlTag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. Data type `xmlAttribute` contains the name of the attribute and its value (both of type `string`). Type alias `xml` is a renaming of the data type `xmlNode`. Finally, type alias `xPath` is defined as a function from nodes to nodes, and is the type of XPath constructors. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Notice that in $\mathcal{TOY}$ we do not still consider the adequacy of the document to its underlying *Schema* definition [32]. This task has been addressed in functional programming defining regular expression types [30]. However, we assume well-formed input XML documents. In order to import XML documents, the $\mathcal{TOY}$ primitive `load_xml_file` loads an XML file returning its representation as a value of type `xmlNode`. Figure 1 shows an example of XML file and its representation in $\mathcal{TOY}$.

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in $\mathcal{TOY}$ for `xPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [2] and also the usual in functional programming [16]. However,

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
xmlTag "root" [att "version" "1.0"] [
xmlTag "food" [] [
  xmlTag "item" [att "type" "fruit"] [
    xmlTag "name" [] [txt "watermelon"],
    xmlTag "price" [] [txt "32"]
  ],
  xmlTag "item" [att "type" "fruit"] [
    xmlTag "name" [] [txt "oranges"],
    xmlTag "variety" [] [txt "navel"],
    xmlTag "price" [] [txt "74"]
  ],
  xmlTag "item" [att "type" "vegetable"][
    xmlTag "name" [] [txt "onions"],
    xmlTag "price" [] [txt "55"]
  ],
  xmlTag "item" [att "type" "fruit"] [
    xmlTag "name" [] [txt "strawberries"],
    xmlTag "variety" [] [txt "alpine"],
    xmlTag "price" [] [txt "210"]
  ]
 ]]
```

Fig. 1. XML example (left) and its representation in $\mathcal{TOY}$ (right)

in our case we take advantage of the non-deterministic nature of our language, returning each result individually. We define an XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result: `type xPath = xml -> xml`. In order to apply an XPath expression to a particular document, we use the following infix operator definition:

```
(<--) :: string -> xPath -> xml     S <-- Q = Q (load_xml_file S)
```

The input arguments of this operator are a string `S` representing the file name and an XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator plays in $\mathcal{TOY}$ the role of `doc` in XPath. The XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively, are defined in $\mathcal{TOY}$ as function composition:

```
infixr 55 .::.                    infixr 40 ./.
(.::.) :: xPath -> xPath -> xPath (./.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)           (F ./. G) X = G (F X)
```

We use the function operator names `.::.` and `./.` because `::` and `/` are already defined in $\mathcal{TOY}$. Also notice that their definitions are the same. Indeed, we could use a single operator for representing both combinators, but we decided to do this way for maintaining a similar syntax for XPath practitioners, more accustomed to use such symbols. In addition, we do not check for the "appropriate" use of such operators and either rely on the provided automatic translation by the parser or left to the user. The variable `X` represents the input XML fragment (the context node). The rules specify how the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Figure 2 shows the $\mathcal{TOY}$ definition of XPath main axes and tests. node. In our setting, it corresponds simply to the identity function. A more interesting axis is `child`, which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that in the $\mathcal{TOY}$ representation these nodes correspond to terms rooted by constructor `xmlTag`. Once `child` has been defined, `descendant` and `descendant-or-self` are just generalizations. The first rule for this function specifies that `child` must be used once, while the second rule corresponds to two or more applications of `child`. In this rule,

```
self,child,descendant :: xPath
descendant_or_self :: xPath
self X = X
child (tag _ _ L) = member L
descendant X = child X
descendant X = if child X == Y
               then descendant Y
descendant_or_self =
               self ? descendant
```

```
nodeT,elem :: xPath
nameT,textT,commentT::string->xPath
nodeT X = X
nameT S (xmlTag S Att L) =
         xmlTag S Att L
textT S (txt S) = txt S
commentT S (comment S) = comment S
elem = nameT _
```

Fig. 2. XPath axes and tests in $\mathcal{TOY}$

the `if` statement is employed to ensure that `child` succeeds when applied to the input XML fragment, thus avoiding possibly infinite recursive calls. Finally, the definition of axis `descendant-or-self` is straightforward. Observe that the XML input argument is not necessary in this natural definition. With respect to test nodes, the first test defined in Figure 2 is `nodeT`, which corresponds to `node()` in the usual XPath syntax. This test is simply the identity. For instance, here is the XPath expression that returns all the nodes in an XML document, together with its $\mathcal{TOY}$ equivalent:

```
XPath → doc("food.xml")/descendant-or-self::node()
TOY   → ("food.xml" <- descendant_or_self.::.nodeT)==R
```

The only difference is that the $\mathcal{TOY}$ expression returns one result at a time in the variable `R`, asking the user if more results are needed. If the user wishes to obtain all the solutions at a time, as usual in XPath evaluators, then it is enough to use the primitive `collect`. For instance, the answer to the $\mathcal{TOY}$ goal:

```
Toy> collect ("food.xml" <-- descendant_or_self.::.nodeT) == R
```

produces a single answer, with `R` instantiated to a list whose elements are the nodes in `"food.xml"`. XPath abbreviated syntax allows the programmer to omit the axis `child::` from a location step when it is followed by a name. Thus, the query `child::food/child::price/child::item` simply `food/price/item`. In $\mathcal{TOY}$ we cannot do that directly because we are in a typed language and the combinator `./.` expects xPath expressions and not strings. However, we can introduce a similar abbreviation by defining new unitary operators `name` (and similarly `text`), which transform strings into XPath expressions:

```
name :: string -> xPath
name S = child.::.(nameT S)
```

So, we can write in $\mathcal{TOY}$ `name "food"./.name "item"./.name "price"`.

Other tests as `nameT` and `textT` *select* fragments of the XML input, which can be returned in a logical variable, as in:

```
XPath → child::food/child::item/child::price/child::text()
TOY   → child.::.nameT "food"./.child.::.nameT "item" ./.
        child.::.nameT "price"./.child.::.textT P
```

The logic variable `P` obtains the prices contained in the example document. Another XPath useful abbreviation is `//` which stands for the unabbreviated expression `/descendant-or-self::node()/`. In $\mathcal{TOY}$, we can define:

```
infixr 30 .//.
(.//.) ::  xPath -> xPath -> xPath
A .//. B = append A (descendant_or_self .::. nodeT ./. B)
append :: xPath -> xPath -> xPath
append (A.::.B) C = (A.::.B) ./. C
append (X ./.Y) C = X ./. (append Y  C)
```

Notice that a new function `append` is used for concatenating XPath expressions. This function is analogous to the well-known `append` for lists, but defined over `xPath` terms. This is our first example of the usefulness of higher-order patterns since for instance pattern `(A.::.B)` has type `xPath`, i.e., `xml -> xml`.

3 XQuery in $\mathcal{TOY}$

Now, we are in a position to define the proposed extension to XQuery. Firstly, the subset of XQuery expressions handled in our setting is presented (XQuery is a richer language than the fragment presented here):

XQuery ::= XPath | $Var | XQuery/XPath* |
let $Var **:=** XQuery [**where** BXQuery] **return** XQuery |
for $Var **in** XQuery [**where** BXQuery] **return** XQuery |
<tag> XQuery < /tag>
BXQuery ::= XQuery | XQuery=XQuery

Basically, the XQuery fragment handled in $\mathcal{TOY}$ allows building new XML documents employing new tags, and the traversal of XML documents by means of the `for` construction. XQuery variables are used in `for` and `let` expressions and can occur in the built documents and XPath expressions. It is worth observing that XPath can be applied to XQuery expressions, that is, for instance, XPath can be applied to the result of a `for` expression. Therefore, such XPath expressions are not rooted by documents (they are denoted by XPath*). In order to encode XQuery in $\mathcal{TOY}$ we define a new type:

```
type xQuery = [xml]
```

In Section 2, XPath has been represented as functions from xml nodes to xml nodes. However, XQuery expressions are defined as sequences of xml nodes represented in $\mathcal{TOY}$ by lists. This does not imply that our approach returns the answers enclosed in lists, it still uses non-determinism for evaluating `for` and XPath expressions. We define functions for representing `for-let-where-return` expressions as follows. Firstly, `let` and `for` expressions are defined as:

```
xLet :: xQuery -> xQuery -> xQuery
xLet X [Y]       = if X == collect Y then X
xLet X (X1:X2:L) = if X == (X1:X2:L) then X

xFor :: xQuery -> xQuery -> xQuery
xFor X [Y]       = if X == [Y] then X
xFor X (X1:X2:L) = if X == [member (X1:X2:L)] then X
```

`xLet` uses `collect` for capturing the elements of `Y` in a list, whereas `xFor` retrieves non deterministically the elements of `Expr` in unitary lists. It fits well, for instance, when `Y` is an XPath expression in $\mathcal{TOY}$. The definition of `for` relies on the non-deterministic function `member` defined in Section 2. Now $\mathcal{TOY}$ goals like `xFor X ("food.xml" <$- name "food" ./. name "item")==R` or `xLet X`

`("food.xml" <$- name "food" ./. name "item")==R` can be tried. Let us remark that XPath expressions have been modified in XQuery as follows. A new operator `<$-` is defined in terms of `<-`:

```
infixr 35 <$--
(<$--) :: string -> xPath -> xQuery
(<$--) Doc Path = [(<--) Doc Path]
```

The function `<$-` returns (non deterministically) unitary lists with the elements of the given document in the corresponding path. Therefore, XPath and `for` expressions have the same behavior in the $\mathcal{TOY}$ implementation of XQuery. In other words, `(<$-)` serves for type conversion from XPath to XQuery. Now, we can define `where` and `return` as follows:

```
infixr 35 `xWhere`
(`xWhere`) :: xQuery -> bool -> xQuery
(`xWhere`) X Y = if Y then X

infixr 35 `xReturn`
(`xReturn`) :: xQuery -> xQuery -> xQuery
(`xReturn`) X Y = if X == _ then Y
```

The definition of `xWhere` is straightforward: the query `X` is returned if the condition `Y` can be satisfied. The `if` statement in `xReturn` forces the evaluation of `X`. The anonymous variable (`_`) can be read as *if the query* `X` *does not fail, then return* `Y`. With these definitions, we can simulate many XQuery expressions in $\mathcal{TOY}$. However, there are two elements still to be added. XPath expressions can now be rooted by XQuery expressions. Thus, we add a new function:

```
infixr 35 <$
(<$) :: xQuery -> xPath -> xQuery
(<$) [Y] Path = [Path Y]
(<$) (X:Y:L) Path = map Path (X:Y:L)
```

The first argument is an XPath variable or, more generally, an XQuery expression. The XPath expression represented by variable `Path` is applied to all the values produced by the XQuery expression. According to the commented behavior, XQuery expressions can be unitary lists (`for`'s and XPath's) and non-unitary lists (`let`'s). The `xmlTag` constructor is also converted into a function `xmlTagX`:

```
xmlTagX :: string -> [xmlAttribute] -> xQuery -> xQuery
xmlTagX Name Attributes [Expr] =
     if Y == collect Expr then [xmlTag Name Attributes Y]
xmlTagX Name Attributes (X:Y:L) = [xmlTag Name Attributes (X:Y:L)]
```

Basically, this conversion is required to apply `collect` when either a `for` or an XPath expression provides the elements enclosed in an XML tag. With the previous definitions, $\mathcal{TOY}$ accepts the following query:

```
R == xmlTagX "names" []
         (xLet X ("food.xml" <$-- name "food")
         `xReturn`
         xmlTagX "result" [] (X <$ (name "item"./.name "name")))
```

which simulates the query:

```
<names>
let $x:=doc("food.xml")/food return
<result> { $x/item/name } </result>
</names>
```

and outcomes the following answer:

```
{R -> [xmlTag "names" []
         [xmlTag "result" [] [
          xmlTag "name" [] [xmlText "watermelon" ],
          xmlTag "name" [] [xmlText "oranges" ],
          xmlTag "name" [] [xmlText "onions" ],
          xmlTag "name" [] [xmlText "strawberries" ]]]] }
```

It is worth noticing that $\mathcal{TOY}$ shows not only the binding for `R`, but also for the variable `X`. If we are interested in the query without the values of the variables, we can introduce a function containing the code:

```
query = xmlTagX "names" []
         (xLet X ("food.xml" <$-- name "food")
         `xReturn`
         xmlTagX "result" [] (X <$ (name "item"./.name "name")))
```

and try the goal `query == R` to get the same result. In the case of `for` expressions, we can write:

```
query2 = xFor Y
        (xFor X ("food.xml" <$-- name "food")
              `xReturn` (X <$ (name "item" ./. name "name")))
        `xReturn` Y
```

which simulates the following query:

```
for $Y in
(for $X in doc("food.xml")/food return $X/item/name)
return $Y
```

The following $\mathcal{TOY}$ query returns four answers, once at a time, due to the use of non-determinism in the `for` expression:

```
Toy> query2== X
      { X -> [xmlTag "name" [] [xmlText "watermelon"]] }
      { X -> [xmlTag "name" [] [xmlText "oranges"]] }
      { X -> [xmlTag "name" [] [xmlText "onions"]] }
      { X -> [xmlTag "name" [] [xmlText "strawberries"]] }
```

4 XQuery Optimization in $\mathcal{TOY}$

In this section we present one of the advantages of using $\mathcal{TOY}$ for running XQuery expressions. In [10] we have shown that XPath queries can be preprocessed by replacing the reverse axes by predicate filters including forward axes, as shown in [25]. In the case of XQuery, one of the optimizations to be achieved is to avoid XPath expressions at outermost positions. Here is an example of optimization. Consider the following query:

```
exam = xFor X
            (xFor Y ("food.xml" <$-- name "food" ./. name "item")
            `xReturn`
            (xmlTagX "elem" []
                (xFor Z (Y <$ name "name")
                    `xReturn` (xmlTagX "ids" [] Z))))
        `xReturn`
            (X <$ ((name "ids") ./. (name "name")))
```

In such a query, `(X <$ ((name "ids") ./. (name "name"))` is an XPath expression applied to an XML term constructed by the same query. By removing outermost XPath expressions, we can optimize XQuery expressions. In general, a place for optimization are nested XQuery expressions [21,15]. In our case, we argue that XPath can be statically applied to XQuery expressions. The optimization comes from the fact that unnecessary XML terms can be built at run-time, and that removing them improves memory consumption. We observe in the previous query that `"elem"` and `"ids"` tags are useless, once we retrieve `"name"` from the original file. Therefore, the previous query can be rewritten into a more simpler and equivalent one:

```
examo = ("food.xml" <$-- name "food"./.name "item"./.name "name")
```

4.1 XQuery as Higher Order Patterns

In order to proceed with optimizations, we follow the same approach as in XPath. In [10] we have used the representation of XPath expressions for optimizing. As it was commented before, XPath operators are higher order operators, and then we can take advantage of the $\mathcal{TOY}$ facilities for using higher order patterns to rewrite them. This is not the case, however, for XQuery expressions in $\mathcal{TOY}$ because, for instance, `xFor` and `xLet` are always applied to two arguments, and therefore constitute reducible expressions, not higher-order patterns. In order to convert XQuery-$\mathcal{TOY}$ expressions into higher-order patterns, we propose a redefinition of the functions adding a dummy argument. Then, XQuery constructors can be redefined as follows:

```
yLet :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yLet X Y _ = xLet (X _)  (Y _)
```

```
yFor :: (A -> xQuery) -> (A -> xQuery) -> A -> xQuery
yFor X Y _ = xFor (X _)  (Y _)
```

The anonymous variable plays a role similar to the `quote` operator in Lisp [28]. In our case the expressions will become reducible when any extra argument is provided. In the meanwhile it can be considered as a data term, and as such it can be analyzed and modified. In the definitions above, `yLet` is reduced to `xLet` when such extra argument is provided. The two arguments `X` and `Y` also need their extra variable to become reducible. A variable is a special case, which has to be converted into a function (`xvar`):

```
xvar :: xQuery -> A -> xQuery
xvar X _ = X
```

Now, a given query can be rewritten as a higher order pattern. For instance, the previous `exam` can be represented as follows:

```
xexam = yFor (xvar X)
       (yFor (xvar Y) ("food.xml" <$$-- name "food"./.name "item")
              `yReturn`
                 (xmlTagY "elem" []
                    (yFor (xvar Z) ((xvar Y) <$$ (name "name"))
                       `yReturn` (xmlTagY "ids" [] (xvar Z)) )))
        `yReturn`
        ((xvar X) <$$ ((name "ids") ./. name "name"))
```

The query can be executed in $\mathcal{TOY}$ just providing any additional argument, in this case an anonymous variable:

```
Toy> xexam _ == R
      { R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
      { R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
      { R -> [xmlTag "name" [] [xmlText "onions" ] ] }
      { R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }
```

If the extra argument `_` is omitted, then the variable `R` is bound to the XQuery code `yFor (xvar X) (...name "name"))`. This behavior allows us to inspect and modify the query in the next subsection.

4.2 XQuery Transformations

Now, we would like to show how to rewrite XQuery expressions in order to optimize them. We have defined a set of transformation rules for removing outermost XPath expressions, when possible. Let us remark that correctness of the transformation rules, that is, preserving equivalence, is out of the scope of this paper. An example of (a subset of) the transformation rules is:

```
reduce ((yFor (xvar Z) E) `yReturn` (xvar Z)) = E
reduce ((xmlTagY N A E) <$$ P)   = reduce_xml (xmlTagY N A E) P
reduce_xml (xmlTagY N A E) P     = reduce_xmlPath E P
reduce_xmlPath (xmlTagY N A E) P =
                       if P == (name N) ./. P2
                       then reduce_xmlPath E P2
                       else ((xmlTagY N A E) <$$ P)
...
```

The first reduction rule removes the unnecessary `for` expressions that define a variable `Z` taking a value `E` only to return `Z`. The second rule removes XPath expressions that traverse elements built in the same query. For instance, an expression of the form `$X/a/b`, with `$X` of the form `<a>E</a>` is reduced to `$Y/b` with `$Y/b` and `$X` taking the value `E` (this transformation is performed by function `reduce_xmlPath`). The optimizer can be defined as the fixpoint of function `reduce`:

```
optimize :: (A -> xQuery) -> (A -> xQuery)
optimize X = iterate reduce X

iterate :: (A -> A) -> A -> A
iterate G X = if Y == X then Y else iterate G Y
              where Y = (G X)
```

For instance, the running example is optimized as follows:

```
Toy> optimize xexam == X
      { X -> (<$$-- "food.xml" child .::. (nameT "food") ./.
       child .::. (nameT "item") ./. child .::. (nameT "name")) }
```

Finally, an XQuery expression is executed (with optimizations) in $\mathcal{TOY}$ by calling the function `run`, which is defined as:

```
run :: (A -> xQuery) -> xQuery
run X = (optimize X) _
```

By using `run`, $\mathcal{TOY}$ obtains the same four answers as with the original query:

```
Toy> run xexam == X
      { R -> [xmlTag "name" [] [xmlText "watermelon" ] ] }
      { R -> [xmlTag "name" [] [xmlText "oranges" ] ] }
      { R -> [xmlTag "name" [] [xmlText "onions" ] ] }
      { R -> [xmlTag "name" [] [xmlText "strawberries" ] ] }
```

In order to analyze the performance of the optimization, the next table compares the elapsed time for the query running on $\mathcal{TOY}$ before and after the optimization, with respect to different sizes for file `"food.xml"`.

Items	*Initial Query*	*Optimized Query*	*Speed-up*
1,000	1.9	0.4	4.8
2,000	3.7	0.8	9.3
4,000	7.4	1.7	4.4
8,000	18.1	3.9	4.6
16,000	36.0	7.8	4.6

The first column indicates the number of `item` elements included in `"food.xml"`, the second and third column display the time in seconds required by the original and the optimized query, respectively, and the last column displays the speed-up of the optimized code. In order to force the queries to find all the answers, the submitted goals are `(exam == R, false)` and `(run xexam == R, false)`, corresponding to the initial and the optimized query, respectively. The atom `false` after the first atomic subgoal always fails, forcing the reevaluation until no more solutions exist. As can be seen in the table, in this experiment the optimized query is above 4.5 times faster in the average than the initial one. In other experiments (for instance, replacing `for` by `let` in this example) the difference can be noticeable also in terms of memory, since the system runs out of memory computing the query before optimization, but works fine with the optimized query. Of course, more extensive benchmarks would be needed to assess this preliminary results. However, the purpose of this paper is not to propose or to evaluate XQuery optimizations, but to show how they can be easily incorporated and tested in our framework.

5 Conclusions

We have shown how the declarative nature of the XML query language XQuery fits in a very natural way in functional-logic languages. Our setting fruitfully combines the collection of results required by XQuery `let` statements and the use of individual values as required by `for` statements and XPath expressions. For the users of the functional-logic $\mathcal{TOY}$, the advantage is clear: they can use queries very similar to XQuery in their programs. Although adapting to the $\mathcal{TOY}$ syntax can be hard at first, we think that the queries are close enough to their equivalents in native XQuery. However, we would like to go further by providing a parser from XQuery standard syntax to the equivalent $\mathcal{TOY}$ expressions.

From the point of view of the XQuery apprentices, the tool can be useful, specially if they have some previous knowledge of declarative languages. The possibility of testing query optimizations can be very helpful. The paper shows a technique based on the use of additional dummy variables for converting queries in higher-order patters. A similar idea would be to use a data type for representing the query and then a parser/interpreter for evaluating this data type. However, we think that the approach considered here has a higher abstraction level, since the queries can not only be analyzed, they can also be computed by simply providing an additional argument. Finally, the framework can also

be interesting for designers of XQuery environments, because it allows users to easily define prototypes of new features such as new combinators and functions.

A version of the $\mathcal{TOY}$ system including the examples of this paper can be downloaded from `http://gpd.sip.ucm.es/rafa/wflp2011/toyxquery.rar`

References

1. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Declarative Embedding of XQuery in a Functional-Logic Language. Technical Report SIC-04/11, Facultad de Informática, Universidad Complutense de Madrid (2011), `http://gpd.sip.ucm.es/rafa/xquery/`
2. Almendros-Jiménez, J.M.: An Encoding of XQuery in Prolog. In: Bellahsène, Z., Hunt, E., Rys, M., Unland, R. (eds.) XSym 2009. LNCS, vol. 5679, pp. 145–155. Springer, Heidelberg (2009)
3. Almendros-Jiménez, J.M., Becerra-Terón, A., Enciso-Baños, F.J.: Querying XML documents in logic programming. Journal of Theory and Practice of Logic Programming 8(3), 323–361 (2008)
4. Atanassow, F., Clarke, D., Jeuring, J.: UUXML: A type-preserving XML schema–haskell data binding. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 71–85. Springer, Heidelberg (2004)
5. Benzaken, V., Castagna, G., Frish, A.: CDuce: an XML-centric general-purpose language. In: Proc. of the ACM SIGPLAN International Conference on Functional Programming, pp. 51–63. ACM Press, New York (2005)
6. Benzaken, V., Castagna, G., Miachon, C.: A Full Pattern-Based Paradigm for XML Query Processing. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 235–252. Springer, Heidelberg (2005)
7. Boncz, P., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 479–490. ACM Press, New York (2006)
8. Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: Pathfinder: XQuery - The Relational Way. In: Proc. of the International Conference on Very Large Databases, pp. 1322–1325. ACM Press, New York (2005)
9. Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In: Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R. (eds.) NODe-WS 2002. LNCS, vol. 2593, pp. 295–310. Springer, Heidelberg (2003)
10. Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: Integrating XPath with the Functional-Logic Language Toy. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 145–159. Springer, Heidelberg (2011)
11. Caballero, R., López-Fraguas, F.: A functional-logic perspective on parsing. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
12. Cabeza, D., Hermenegildo, M.: Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. Theory and Practice of Logic Programming 1(3), 251–282 (2001)
13. Coelho, J., Florido, M.: XCentric: logic programming for XML processing. In: WIDM 2007: Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management, pp. 1–8. ACM Press, New York (2007)
14. Fegaras, L.: HXQ: A Compiler from XQuery to Haskell (2010)

15. Grinev, M., Pleshachkov, P.: Rewriting-based optimization for XQuery transformational queries. In: 9th International Database Engineering and Application Symposium, IDEAS 2005, pp. 163–174. IEEE Computer Society, Los Alamitos (2005)
16. Guerra, R., Jeuring, J., Swierstra, S.D.: Generic validation in an XPath-Haskell data binding. In: Proceedings Plan-X (2005)
17. Hanus, M.: Curry: An Integrated Functional Logic Language (2003), `http://www.informatik.uni-kiel.de/~mh/curry/` (version 0.8.2 March 28, 2006)
18. Hanus, M.: Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel (2011)
19. Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)
20. Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. 8(4), 437–444 (1998)
21. Koch, C.: On the role of composition in XQuery. In: Proc. WebDB (2005)
22. Fraguas, F.J.L., Hernández, J.S.: TOY: A Multiparadigm Declarative System. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 244–247. Springer, Heidelberg (1999)
23. Marian, A., Simeon, J.: Projecting XML Documents. In: Proc. of International Conference on Very Large Databases, pp. 213–224. Morgan Kaufmann, Burlington (2003)
24. May, W.: XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. Theory and Practice of Logic Programming 4(3), 239–287 (2004)
25. Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking forward. In: Chaudhri, A.B., Unland, R., Djeraba, C., Lindner, W. (eds.) EDBT 2002. LNCS, vol. 2490, pp. 109–127. Springer, Heidelberg (2002)
26. Ronen, R., Shmueli, O.: Evaluation of datalog extended with an XPath predicate. In: Proceedings of the 9th Annual ACM International Workshop on Web Information and Data Management, pp. 9–16. ACM, New York (2007)
27. Schaffert, S., Bry, F.: A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In: Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web. CEUR Workshop Proceedings, vol. 60, p. 22 (2002)
28. Seibel, P.: Practical Common Lisp. Apress (2004)
29. Seipel, D.: Processing XML-Documents in Prolog. In: Procs. of the Workshop on Logic Programming 2002, p. 15. Technische Universität Dresden, Dresden (2002)
30. Sulzmann, M., Lu, K.Z.: XHaskell – adding regular expression types to haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 75–92. Springer, Heidelberg (2008)
31. Thiemann, P.: A typed representation for HTML and XML documents in Haskell. Journal of Functional Programming 12(4&5), 435–468 (2002)
32. W3C. XML Schema 1.1
33. W3C. Extensible Markup Language, XML (2007)
34. W3C. XML Path Language (XPath) 2.0 (2007)
35. W3C. XQuery 1.0: An XML Query Language (2007)
36. Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the International Conference on Functional Programming, pp. 148–159. ACM Press, New York (1999)
37. Walmsley, P.: XQuery. O'Reilly Media, Inc., Sebastopol (2007)
38. Wielemaker, J.: SWI-Prolog SGML/XML Parser, Version 2.0.5. Technical report, Human Computer-Studies (HCS), University of Amsterdam (March 2005)

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 282 (2012) 19–34

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

XPath Query Processing in a Functional-Logic Language

J.M. Almendros-Jiménez[a,1], R. Caballero[b,1], Y. García-Ruiz [b,1], F. Sáenz-Pérez [c,1],

[a] *Dpto. de Lenguajes y Computación*
Universidad de Almería, Spain

[b] *Dpto. de Sistemas Informáticos y Computación*
Universidad Complutense de Madrid, Spain

[c] *Dpto. de Ingeniería del Software e Inteligencia Artificial*
Universidad Complutense de Madrid, Spain

Abstract

XPath is a well-known query language for finding and extracting information from XML documents. This paper shows how the characteristics of this domain-specific language fits very well into the functional-logic paradigm. The proposed framework allows the user to write XPath-like queries as first-class citizens of the functional-logic language $\mathcal{TOY}$, using higher-order combinators for constructing the queries and non-determinism in order to obtain the different answers that XPath queries can return. The result is a very good example of cross-fertilization of two different areas. In the case of $\mathcal{TOY}$, the users can now integrate XML queries in their programs without using any external library or ad hoc interface. In the case of XPath, the use of higher-order patterns allow us to define functions for easily processing the queries. In particular, the paper shows how to trace and debug erroneous queries, and how to detect that a query is a refinement of another query, which can be useful for improving the efficiency of query processing.

Keywords: Functional-Logic Programming, Non-Deterministic Functions, XPath, Higher-Order Patterns

[1] This work has been supported by the Spanish projects STAMP (TIN2008-06622-C03-01, TIN2008-06622-C03-03), S-0505/TIC/0407, Prometidos-CM (S2009TIC-1465), and GPD (UCM-BSCH-GR35/10-A-910502)

doi:10.1016/j.entcs.2011.12.003

1 Introduction

In the last years the eXtensible Markup Language XML [18] has become the *de facto* standard for exchanging structured data in plain text files. This was the key for its success as data structures are revealed and therefore they are readily available for its processing (even with usual text editors if one wishes to manually edit them). Structured data means that new, more involved access methods must be devised. XQuery [21,23] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [19], a domain-specific language that has become part of general-purpose languages. Although less expressive than XQuery, the simplicity of XPath makes it a perfect tool for many types of queries. Due to its acknowledged importance, XML and its query languages have been embodied in many applications as in database management systems, which include native support for XML data and documents both in data representations and query languages (e.g., Oracle and SQL Server). Some of them extend SQL to include support for XQuery, so that results from XML queries can be used by the more declarative SQL language in the context of a database, making possible to share relational and XML data sources.

Many general-purpose declarative programming languages include support for XPath and XQuery. In the functional programming area, works about Haskell can be found in [17,2,22,16]. There are also proposals based on logic programming as [15,14,3,12,7,13]. In the field of functional-logic languages, [10] proposes a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry [9]. Recently, in [5], we have proposed an implementation of XPath in the functional-logic language $\mathcal{TOY}$ [11], where a XPath query becomes at the same time implementation (code) and representation (data term). XML documents are represented in this proposal by means of data terms, and the basic constructors of XPath: `child`, `self`, `descendant`, etc. are defined as non-deterministic higher-order functions that can be applied to XML terms.

This paper continues this work, showing that XPath fits very well into the functional-logic paradigm. The proposed framework allows the user to write XPath-like queries as first-class citizens of the functional-logic language $\mathcal{TOY}$. To this end, higher-order combinators are used for constructing queries, and we take advantage of non-determinism in order to obtain the different answers that XPath queries may return. The result is a very good example of cross-fertilization of two different areas:

- In the case of $\mathcal{TOY}$ (introduced in Section 2), users can now integrate XML queries in their programs without using any external library or *ad hoc*

interface.

- In the case of XPath (introduced in Section 3), the use of higher-order patterns allows us to define functions for processing easily queries.

Justification for the embedding is also provided in this paper in two forms, which constitute our main contributions: First, we show in Section 4 how to debug and trace erroneous queries. Second, we apply the idea of *compensation* [24] in the field of relational databases to our scheme. This refers to the ability of reusing previous cached query results for enhancing query solving performance (Section 5). Finally, in Section 6 we draw some conclusions and point out some future work.

2 The Functional-Logic Language $\mathcal{TOY}$

A $\mathcal{TOY}$ [11] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of (total) *expressions* in $\mathcal{TOY}$ $e \in Exp$ is $e ::= X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). Similarly, the syntax of (total) *patterns* $t \in Pat \subset Exp$ can be defined as $t ::= X \mid c\, t_1 \ldots t_m \mid f\, t_1 \ldots t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are patterns for all $1 \leq i \leq m$.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$:

```
data node      = txt     string
               | comment string
               | tag     string [attribute] [node]
data attribute = att     string string
type xml       = node
```

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, the constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation. Figure 1 in next page shows an XML document and its representation

in $\mathcal{TOY}$.

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
tag "root" [att "version" "1.0"] [
tag "food" [] [
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "watermelon"],
     tag "price" [] [txt "32"]
  ],
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "oranges"],
     tag "variety" [] [txt "navel"],
     tag "price" [] [txt "74"]
  ],
  tag "item" [att "type" "vegetable"] [
     tag "name" [] [txt "onions"],
     tag "price" [] [txt "55"]
  ],
  tag "item" [att "type" "fruit"] [
     tag "name" [] [txt "strawberries"],
     tag "variety" [] [txt "alpine"],
     tag "price" [] [txt "210"]
  ]
 ]]
```

Fig. 1. XML example (left) and its representation in $\mathcal{TOY}$ (right)

Each rule for a function f in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where e_i, u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns.

In $\mathcal{TOY}$ variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase. $\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience all the documents are started with a dummy node `root`. This is useful for grouping several XML fragments. If the file contains only one node `N` at the outer level, `root` can be removed defining the following simple function:

```
load_doc F = N <== load_xml_file F == tag "root" [att "version"
"1.0"] [N]
```

where `F` is the name of the file containing the document. Observe that the strict equality `==` in the condition forces the evaluation of `load_xml_file F` and succeeds if the result has the form `tag "root" [att "version" "1.0"] [N]` for some `N`. If this is the case, `N` is returned.

3 Representing XPath Queries

This section introduces the subset of XPath that we intend to integrate with $\mathcal{TOY}$, omitting all the features of XPath that are supported by $\mathcal{TOY}$ but not used in this paper, such as preprocessing of reverse axes. See [5,4] for a more detailed introduction to XPath in $\mathcal{TOY}$.

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in a functional language for `xPath` could be `type xPath = xml -> [xml]` meaning that a list or sequence of results is obtained. This is the approach considered in [1] and also the usual in functional programming [8]. However, in our case we take advantage of the non-deterministic nature of our language, returning each result individually. We define a XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result: `type xPath = xml -> xml`. In order to apply a XPath expression to a particular document, we use the following infix operator definition:

```
infix  20 <--
(<--)::string  -> xPath -> xml
S <-- Q = Q (load_xml_file S)
```

The input arguments of this operator are a string `S` representing the file name and a XPath query `Q`. The function applies `Q` to the XML document contained in file `S`. This operator plays in $\mathcal{TOY}$ the role of `doc` in XPath. The XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively, are defined in $\mathcal{TOY}$ as function composition:

```
infixr 55 .::.                          infixr 40 ./.
(.::.) :: xPath -> xPath -> xPath       (./.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)                 (F ./. G) X = G (F X)
```

The function operator names `.::.` and `./.` are employed because the standard XPath separators `::` and `/` are already defined in $\mathcal{TOY}$ with a different meaning. Notice that the two definitions are the same since they stand for the application of a XPath expression to another XPath expression and return also a XPath expression, although they are intended to be applied to different fragments of XPath: `./.` for steps and `.::.` for combining axes and tests producing steps. Indeed, we would use a single operator for representing both combinators, but we decided to do this way for maintaining a similar syntax for XPath practitioners, more accustomed to use such symbols. In addition, we do not check for the "appropriate" use of such operators and either rely on the provided automatic translation by the parser or left to the user. The variable `X` represents the input XML fragment (the context node). The rules specify how

```
self,child,descendant ::  xPath
descendant_or_self ::  xPath
self X = X
child (tag _ _ L) = member L
descendant X = child X
descendant X =
       if child X == Y
       then descendant Y
descendant_or_self =
       self ?  descendant
```

```
nodeT,elem ::  xPath
nameT,textT,commentT ::
         string->xPath
nodeT X = X
nameT S (tag S Att L) =
         tag S Att L
textT S (txt S) = txt S
commentT S (comment S) =
         comment S
elem = nameT _
```

Fig. 2. XPath axes and tests in $\mathcal{TOY}$

the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Figure 2 shows the $\mathcal{TOY}$ definition of XPath main axes and tests. The first one is `self`, which returns the context node. In our setting, it corresponds simply to the identity function. A more interesting axis is `child` which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that in the $\mathcal{TOY}$ representation these nodes correspond to terms rooted by constructor `tag`. Once `child` has been defined, `descendant` and `descendant-or-self` are just generalizations. The first rule for this function specifies that `child` must be used once, while the second rule corresponds to two or more applications of `child`. In this rule, the `if` statement is employed to ensure that `child` succeeds when applied to the input XML fragment, thus avoiding possibly infinite recursive calls. Finally, the definition of axis `descendant-or-self` is straightforward. Observe that the XML input argument is not necessary in this natural definition. With respect to test nodes, the first test defined in Figure 2 is `nodeT`, which corresponds to `node()` in the usual XPath syntax. This test is simply the identity. For instance, here is the XPath expression that returns all the nodes in an XML document, together with its $\mathcal{TOY}$ equivalent:

XPath → `doc("food.xml")/descendant-or-self::node()`

$\mathcal{TOY}$ → `("food.xml" <-- descendant_or_self.::.nodeT) == R`

The only difference is that the $\mathcal{TOY}$ expression returns one result at a time in the variable `R`, asking the user if more results are needed. If the user wishes to obtain all the solutions at a time, as usual in XPath evaluators, then it is enough to use the primitive `collect`. For instance, the answer to:

```
Toy> collect ("food.xml" <-- descendant_or_self.::.nodeT) == R
```

produces a single answer, with `R` instantiated to a list whose elements are the nodes in `"food.xml"`. XPath abbreviated syntax allows the programmer to omit the axis `child::` from a location step when it is followed by a name. Thus, the query `child::food/child::item/child::price` becomes in XPath simply as `food/item/price`. In $\mathcal{TOY}$ we cannot do that directly because we are in a typed language and the combinator `./.` expects XPath expressions and not strings. However, we can introduce a similar abbreviation by defining new unitary operators `name` (and similarly `text`), which transform strings into XPath expressions:

```
name :: string -> xPath          name S = child.::.(nameT S)
```

So, we can write in $\mathcal{TOY}$ `name "food"./.name "item"./.name "price"`. Other tests as `nameT` and `textT` *select* fragments of the XML input, which can be returned in a logical variable, as in:

XPath → `child::food/child::item/child::price/child::text()`

$\mathcal{TOY}$ → `child.::.nameT "food"./.child.::.nameT "item" ./.`

`child.::.nameT "price"./.child.::.textT P`

The logic variable `P` obtains the prices contained in the example document. Another XPath abbreviation is `//` which stands for `/descendant-or-self::node()/`. In $\mathcal{TOY}$, we can define:

```
infixr 30 .//.
(.//.) ::  xPath -> xPath -> xPath
A .//. B = append A (descendant_or_self .::. nodeT ./. B)
append :: xPath -> xPath -> xPath
append (A.::.B) C = (A.::.B) ./. C
append (X ./.Y) C = X ./. (append Y  C)
```

Notice that a new function `append` is used for concatenating XPath expressions. This function is analogous to the well-known `append` for lists, but defined over `xPath` terms. This is our first example of the usefulness of higher-order patterns since for instance pattern `(A.::.B)` has type `xPath`, i.e., `xml -> xml`.

Optionally, XPath tests can include a predicate or filter. Filters in XPath are enclosed between square brackets. In $\mathcal{TOY}$, they are enclosed between round brackets and connected to its associated XPath expression by the operator `.#`:

```
infixr  60 .#
(.#) :: xPath -> xPath -> xPath
(Q .# F)  X =  if F Y == _ then Y  where Y = Q X
```

This definition can be understood as follows: first the query `Q` is applied to

the context node `X`, returning a new context node `Y`. Then the `if` condition checks whether `Y` satisfies the filter `F`, simply by checking that `F Y` does not fail, which means that it returns some value represented by the anonymous variable in `F Y == _`. Although XPath filter predicates allow several possibilities, in this presentation we restrict to XPath expressions. Multiple predicates can be chained together to filter items as with the `and` operator, which can be formulated as follows:

```
infixr 60 /&
(X /& Y) Z = if X Z==_ then Y Z
```

4 Debugging XPath Queries

One of the most appealing features of our setting is that XPath queries can be manipulated. In this section we use this feature for tracing and debugging queries. We distinguish two types of possible errors in a XPath Query depending on the erroneous result produced: *wrong* queries when the query returns an unexpected result, and *missing* when the query does not produce some expected result. We present a different proposal depending on the error.

4.1 Wrong XPath Queries

Consider for instance the goal `("bib.xml" <-- name "bib" ./. name "book" ./. name "author" ./. name "last" ) == R` and suppose that it produces the unexpected answer `R -> (tag "last" [] [ (txt "Abiteboul") ])` (see [20], sample data 1.1.2 and 1.1.4 to check the structure of these documents and an example). If the error is just some misspelling of the author's last name it is easy to look for the wrong information in the document in order to correct the error. However in some situations, and in particular when dealing with complicated, large documents, the error can be in the XPath query, that has selected an erroneous path in the document. In these cases it is also useful to find the answer in the document and then trace back the XPath query until the error is found. Observe that the erroneous answer can be just one of the produced answers (in the example the query can produce many other, expected, answers), and that we are interested only in those portions of the document that produce this unexpected result.

In $\mathcal{TOY}$ we can obtain each intermediate step with its associated answer by defining a suitable function `wrong` that receives three arguments: the query, its input (initially the whole document) and the unexpected output (initially the unexpected answer). The implementation is straightforward:

```
wrong (A.::.B) I O = [((A.::.B), I, O)] <== (A.::.B) I == O
wrong (A./.B) I O  = [(A, I, O1 ) | wrong B O1 O]
```

```
<== A I == O1, B O1 == O
```

In the case of a single step `(A.::.B)` the first rule checks that indeed the step applied to the input produces the output returning the three elements. In the case of two or more steps, the second rule looks for the value `O1` produced by the single step `A` such that the rest of the query `B` applied to `O1` produces the erroneous result `O`. The variable `O1` is a new logic variable, and that the code uses the *generate and test* feature typical of functional languages. The function `wrong` produces a list where each step is associated with its input and its output. However, using `wrong` directly produces a verbose, difficult to understand output due to the representation of XML elements as a data terms in $\mathcal{TOY}$. This can be improved by building a new XML document containing all the information and saving it to a file using the primitive `write_xml_file`:

```
traceStep (Step,I,O) = tag "step" [] [ tag "query" []
                      [txt (show Step)],
                       tag "input" [] [I],
                       tag "output" [] [O] ]

generateTrace L = tag "root" [] (map traceStep (rev L))

writeTrace XPath InputFile WrongOutput OutputFile =
      write_xml_file (
           generateTrace (
              wrong XPath (
                 load_xml_file InputFile)
              WrongOutput))
           OutputFile

show (A.::.B)    = (show A)++".::."++(show B)
show nodeT       = "nodeT"
...
show child      = "child"
...
```

The first function `traceStep` generates an XML element `step` containing the information associated with a XPath step: the step combinator, its input, and its output. This function uses the auxiliary function `show` to obtain the string representation of the step (only part of the code of this function is displayed). Then function `generateTrace` applies `traceStep` to a list of steps. It uses the functions `map` and `rev` whose definition is the same as in functional programming. In particular, `rev` is employed to ensure that the last step of the query is the first in the document, which is convenient for tracing back the result. Finally `writeTrace` combines the previous functions. It receives four parameters: the query, the name of the initial document, the unexpected XML fragment we intend to trace, and the name of the output

file where the result is saved. Now we can try the goal:

```
Toy> writeTrace (name "bib" ./. name "book" ./. name "author" ./.
                  name "last" ) "bib.xml"
                (tag  "last" [] [txt  "Abiteboul"])
                "trace.xml"
```

In this case the symbol == is not used which indicates to $\mathcal{TOY}$ that the result of this expression must be true. After the goal is solved we can consult the document "trace.xml":

```
<step>
 <query>child.::.nameT last</query>
 <input>
  <author>
   <last>Abiteboul</last>
   <first>Serge</first>
  </author>
 </input>
 <output>
  <last>Abiteboul</last>
 </output>
</step>

<step>
 <query>child.::.nameT author</query>
 <input>
  <book year="2000">
   <title>Data on the Web</title>
   <author>
    <last>Abiteboul</last>
```

For the sake of space only the first step and part of the second step is displayed above, although the document contains all the information that allows the user to trace the query.

4.2 Missing XPath Queries

Sometimes a XPath query produces no answer, although some result was expected. For instance the goal `"food.xml" <-- name "food" ./. name "item" ./. name "type" ./. child.::.textT "navel" == R` simply fails in $\mathcal{TOY}$. The reason is that the user wrote `type` where it should be `variety`. This error is very common, and the source of the error can be difficult to detect. Observe that the previous idea of tracing the result back cannot be applied because there is no result to trace. For these situations we propose trying the XPath query without the last step, then if it fails without the last two steps and so on. The idea is to find the first step that produces an empty result, because it is usually the source of the error.

```
missing  (A.::.B) R = (A.::.B)
missing (X ./. Y) R = if (collect (X R) == []) then X
                      else missing Y (X R)
```

The previous definition of function `missing` relies on the primitive `collect` that accumulates all the results of a function call in a list. Therefore `collect (X R) == []` means that `X` applied to the input XML fragment `R` fails. Now we can try the goal:

```
Toy> missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
    { R -> child .::. (nameT  "type") }
```

The answer indicates that the step `child .::. (nameT "type")` is the possible source of the error. Thus this simply function is useful, but we can do better. Instead of simply returning the erroneous step we can try to *guess* how the error can be corrected. In the case of *name* tests as the example the error is usually the same erroneous string has been used. Replacing the string by a logic variable such that the query now succeeds can help to find the error. Therefore, we implement a second version of `missing`:

```
missing (A.::.B) R =  guess (A.::.B) self R

missing (Step ./. Y) R = if (collect (Step R) == [])
                         then guess Step Y R
                         else missing Y (Step R)

guess Step Y R = if Step==(A.::.nameT B)
                 then if (StepBis ./. Y) R == _
                      then ( Step, "Substitute "++B++" by "++C )
                      else (Step, "No suggestion")
                 else (Step, "No suggestion")
    where StepBis = (A.::.nameT C)
```

In this case `missing` returns a pair. The first element of the pair is the same as in the first version, and the second element is a suggestion produced by function `guess`. This function first checks if the `Step` is of the form `(A.::.nameT B)`. If this is the case, it replaces the name `B` by a new variable `C` and uses the condition in the second `if` statement (`(StepBis ./. Y) R == _`) to check if `C` can take any value such that the query does not fail. If such value for `C` is found the returned string proposes replacing `B` by `C`. Otherwise `"No suggestion"` is returned. Since function `guess` requires an argument `Y` with the rest of the XPath query, the basis case of `missing` (first rule) uses `self` to represent the identity query. Now we can try

```
Toy>  missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
{ R -> (child .::. (nameT  "type"),  "Substitute type by variety") }
```

Thus, the debugger finds the erroneous step and proposes the correct solution.

5 Compensation

Now, we would like to show how to take advantage of our implementation based on a functional-logic language for preprocessing queries. For instance, one of the well-known procedures in databases is the so-called *compensation of views*. Let us suppose the case in which a certain user queries the XML database, obtaining a certain answer. Such answer is stored by the system (we can suppose that the answer is locally cached). Later, the user wants to refine the query. A suitable database manager should be able to compute the answer of the refined query from the previous (cached) answer in order to improve answer time and performance.

For instance, let V be the query `"food.xml" <--name "food" ./. name "item"`. Such query retrieves the items from the file "food.xml". Suppose the user wants to refine the first query with a new query P defined as `"food.xml" <-- name "food" ./. name "item".#(name "variety")`. The second query can be answered from the first one, since the user requests the items for which variety is specified. In other words, the second answer is a piece of the first answer. Therefore, the second query can be computed from a new query P' defined as `name "item".#(name "variety")` applied to the answer of the first query. In such a case, P' is called the *compensation* of V w.r.t. P. Compensation of XPath queries has been studied by some authors [24], including the case of multiple views [6]. There are two related problems to compensation: the existence of the compensation and to find a minimal compensation.

Following [24], the compensation to a unique view can be computed by defining a certain *concatenation* operator between queries. Such concatenation operator is defined as follows. Firstly, we need to consider the representation of XPaths by the so-called *tree patterns*. Such tree patterns are tree based representation of XPath expressions in which nodes are labels and edges are axes. We can restrict XPath expressions to the same case as [24], the so-called $XP^{\{/,//,*,[]\}}$ fragment, in which XPath expressions are built from label tests, child axes (`./.`), descendant axes (`.//.`), branches (`.#`) and wildcards (`descendant.::.nodeT`). For instance, let us suppose V to be the XPath expression `name "a" .# name "c" .//. name "b" .# name "f"`, which can be represented as in Figure 3 (b). Black nodes represent the *output node* of the XPath expression, that is, the tag(s) of the answer to the given XPath expression. Now, suppose the XPath expression P' defined as `descendant.::.nodeT .# name "e" .//. name "f"`, and represented by a tree pattern in Figure 3

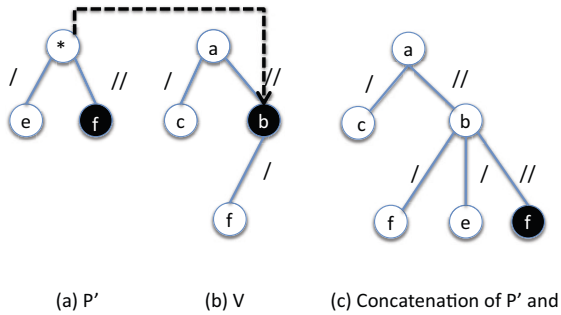


Fig. 3. Concatenation Operator

(a). Then the concatenation operator, denoted by $P' \oplus V$, is defined as in Figure 3 (c), representing `name "a" .# name "c" .//. name "b" .# (name "f" /& name "e") .//. name "f"`.

Basically, given two patterns P' and V, the concatenation of P' and V is constructed by merging the root of P' and the output node of V into one node. The root of V becomes the root of $P' \oplus V$, and the output node of P' becomes the output node of $P' \oplus V$. The merged node has as children both the children of the output node of V and the children of P'. When the nodes to be merged have different labels (for instance, '*' and b of Figure 3), the node assumes the more restrictive label (i.e., b), except when they have different label tests, which means that concatenation is not possible.

Following [24], the problem of finding a compensation of V w.r.t. P is equivalent to finding P' such that $P' \oplus V$ is equal to P. In order to compute in $\mathcal{TOY}$ the compensation of a given XPath expression we have defined the concatenation operator as a $\mathcal{TOY}$ function called `concat`, and the compensation can be defined as: `compensation V P = if concat P' V == P then P'` . It exploits the use of logic variables in $\mathcal{TOY}$, by computing the compensation of a given XPath expression. The function `concat` uses the representation of XPath by means of higher order patterns. The main rules are:

```
concat ((child .::. nameT N) ./. P)
            (child .::. nameT N) = (child .::. nameT N) ./. P
concat P' (P ./. Q) = P ./. (concat P' Q)
concat P' (G .# F) =
              if F==S then (if Fl == self then Rt ./. S
                                else (Rt .# Fl) ./. S)
                       else (if Fl == self then (Rt .# F) ./. S
                                else (Rt .# (F /& Fl))./. S)
              where R = concat P' G
                    S = children R
```

```
                    Fl = filter R
                    Rt = root R
```

The first rule handles the merged node, assigning the children of the first argument to the second argument whenever they have the same label test. Second and third rules are recursive rules for handling steps and filters. Third rule accumulates filters by means of the XPath operator `/&`. The auxiliary functions `children`, `filter` and `root` compute the children and filter of the root, and the root of a XPath expression, respectively. Let us now see an example of use. For instance, defining:

```
v = name "food" ./. name "item"
p = name "food" ./. name "item" .# (name "variety")
```

```
Toy> compensation v p == P'
{ P' -> (child.::.(nameT "item")).#(child.::.(nameT "variety")) }
{ P' -> (descendant.::.nodeT).#(child.::.(nameT "variety")) }
```

We can see that we obtain two answers corresponding to `/item[variety]` and `/*[variety]` in the standard XPath syntax. In summary, we are able to use the logic features of $\mathcal{TOY}$ for building the compensation of a certain view. Currently, $\mathcal{TOY}$ is able to solve one of the problems related to compensations: to find them. $\mathcal{TOY}$ offers as output a set of compensations when they exist, otherwise fails. To find a minimal compensation is considered as future work of this implementation. Moreover, we would like to extend our implementation to cover with compensations to multiple views.

6 Conclusions

This paper shows how the framework for introducing XPath in the functional-logic language $\mathcal{TOY}$ allows us to define readily simple but powerful applications such as preprocessing, tracing and debugging. The implementation makes use of the main features of the language including:
- *Non-determinism* for defining easily the XPath framework, and also in the trace of wrong answers of Section 4, where only those parts of the computation that produce a particular fragment of the answer are required.
- *Higher-order patterns*. This feature allows us to consider $\mathcal{TOY}$ expressions that are not yet reducible as patterns. This means in our case that XPath expressions can be considered at the same time executable code (when applied to an input XML document), or data structures when considered as higher-order patterns. This powerful characteristic of the language is heavily used in our proposals of sections 4 and 5.
- *Logic variables*, specially when used in *generate and test* expressions are very suitable for obtaining the values of intermediate computations, and in our case

also for guessing values in the debugger of missing answers.

Summarizing, we consider that the declarative nature of XPath matches very well the characteristics of functional-logic languages. This benefits both paradigms: the functional-logic language can include easily XPath queries without using any external library, and XPath practitioners can implement easily functions that manipulate the queries in order to devise prototypes of XPath tools such as optimizers or debuggers.

A $\mathcal{TOY}$ implementation that includes the XPath primitives for loading and saving XML documents and most of the examples of this paper can be downloaded from: `http://gpd.sip.ucm.es/rafa/xpath/toyxpath.rar`. As future work, we plan to consider exploiting the same $\mathcal{TOY}$ characteristics for optimizing XQuery expressions.

References

[1] J. M. Almendros-Jiménez. An Encoding of XQuery in Prolog. In *XSym '09: Proceedings of the 6th International XML Database Symposium on Database and XML Technologies*, pages 145–155, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema Haskell Data Binding. In *Proc. of Practical Aspects of Declarative Languages*, pages 71–85, Heidelberg, Germany, 2004. Springer LNCS 3057.

[3] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *Proc. of Web, Web-Services, and Database Systems*, pages 295–310, Heidelberg, Germany, 2002. Springer LNCS 2593.

[4] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$ (Extended Version). Technical Report SIP-05/10, Facultad de Informática, Universidad Complutense de Madrid, 2010. `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-5-10.pdf`.

[5] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 145–159, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *Proceedings of the International Workshop on Web and Databases*, 2008.

[7] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *WIDM '07: Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 1–8, NY, USA, 2007. ACM Press.

[8] R. Guerra, J. Jeuring, and S. D. Swierstra. Generic validation in an XPath-Haskell data binding. In *Proceedings Plan-X*, 2005.

[9] M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8.2 march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[10] M. Hanus. Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel, 2011.

[11] F. J. López-Fraguas and J. S. Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 244–247, London, UK, 1999. Springer-Verlag.

[12] W. May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 4(3):239–287, 2004.

[13] R. Ronen and O. Shmueli. Evaluation of datalog extended with an XPath predicate. In *Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 9–16. ACM New York, NY, USA, 2007.

[14] S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, page 22 pages, Aachen, Germany, 2002. CEUR Workshop Proceedings 60.

[15] D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *In Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management*, pages 140–151. Springer, 2004.

[16] M. Sulzmann and K. Z. Lu. Xhaskell — adding regular expression types to haskell. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 75–92, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, 2002.

[18] W3C. Extensible Markup Language (XML), 2007.

[19] W3C. XML Path Language (XPath) 2.0, 2007.

[20] W3C. XML Query Use Cases, 2007. http://www.w3.org/TR/xquery-use-cases/.

[21] W3C. XQuery 1.0: An XML Query Language, 2007.

[22] M. Wallace and C. Runciman. Haskell and XML: generic combinators or type-based translation? *SIGPLAN Not.*, 34(9):148–159, 1999.

[23] P. Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

[24] W. Xu and Z. Özsoyoglu. Rewriting XPath queries using materialized views. In *Proceedings of the 31st international conference on Very large data bases*, pages 121–132. VLDB Endowment, 2005.

B | Otros recursos

(B.1) **Declarative Debugging of Wrong and Missing Answers for SQL Views**
Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

Technical report SIC-3-11. Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain, 2011. Versión Extendida con demostraciones y contenido extra de "*Declarative Debugging of Wrong and Missing Answers for SQL Views*, (FLOPS 2012)".

→ **Página** 260

(B.2) **Embedding XQuery in Toy**
Jesús Manuel Almendros-Jiménez, Rafael Caballero, Yolanda García-Ruiz and Fernando Sáenz-Pérez

Technical Report SIC-04-11. Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain, 2011. Versión Extendida con contenido extra de "*A Declarative Embedding of XQuery in a Functional-Logic Language*, (LOPSTR 2011)".

→ **Página** 300

Declarative Debugging of Wrong and Missing Answers for SQL Views

Rafael Caballero[†], Yolanda García-Ruiz[†] and Fernando Sáenz-Pérez[‡]

Technical Report SIC-10-12

[†]Dpto. de Sistemas Informáticos y Computación, UCM, Spain
[‡]Dpto. de Ingeniería del Software e Inteligencia Artificial, UCM, Spain
{rafa,fernan}@sip.ucm.es , ygarciar@fdi.ucm.es

April 2011

Abstract. This report presents a debugging technique for diagnosing errors in SQL views. The debugger allows the user to specify the error type, indicating if there is either a missing answer (a tuple was expected but it is not in the result) or a wrong answer (the result contains an unexpected tuple). This information is employed for slicing the associated queries, keeping only those parts that might be the cause of the error. The validity of the results produced by sliced queries is easier to determine, thus facilitating the location of the error. Although based on the ideas of declarative debugging, the proposed technique does not use computation trees explicitly. Instead, the logical relations among the nodes of the trees are represented by logic clauses that also contain the information extracted from the specific questions provided by the user. The atoms in the body of the clauses correspond to questions that the user must answer in order to detect an incorrect relation. The resulting logic program is executed by selecting at each step the unsolved atom that yields the simplest question, repeating the process until an erroneous relation is detected. Soundness and completeness results are provided. The theoretical ideas have been implemented in a working prototype included in the Datalog system DES.

1 Introduction

SQL (Structured Query Language [20]) is a language employed by relational database management systems. In particular, the SQL select statement is used for querying data from databases. Realistic database applications often contain a large number of tables, and in many cases, queries become too complex to be coded by means of a single select statement. In these cases, SQL allows the user to define *views*. A SQL view can be considered as a virtual table, whose content is obtained executing its associated SQL select query. View queries can rely on previously defined views, as well as on database tables. Thus, complex queries can be decomposed into sets of correlated views. As in other programming paradigms, views can have bugs. However, we cannot infer that a view is incorrectly defined when it computes an unexpected result, because it might be receiving erroneous input data from the other database tables or views. Given the high-abstraction level of SQL, usual techniques like trace debugging are difficult to apply. Some tools like [2,15] allow the user to trace and analyze the stored SQL procedures and user defined functions, but they are of little help when debugging systems of correlated views. *Declarative Debugging*, also known as *algorithmic debugging*, is a technique applied successfully in (constraint) logic programming [18], functional programming [14], functional-logic programming [6], and in deductive database languages [3]. The technique can be described as a general debugging schema [13] which starts when an *initial error symptom* is detected by the user. In the context of SQL views, the initial symptom corresponds to an unexpected result produced by a view. The debugger automatically builds a tree representing the erroneous computation that has produced the symptom. In SQL, each node in the tree contains information about both a relation, which is a table or a view, and its associated computed result. The root of the tree corresponds to the query that produced the initial symptom. The children of each node correspond to the relations (tables or views) occurring in the definition of its associated query. After building the tree, it is *navigated* by the debugger, asking to the user about the validity of some nodes. When a node contains the expected result, it is marked as *valid*, and otherwise it is marked as *nonvalid*. The goal of the debugger is to locate a *buggy node*, which is a nonvalid node with valid children. It can be proved that each buggy node in the tree corresponds to either an erroneously defined view, or to a database table containing erroneous data. A debugger based on these ideas was presented in [4]. The main criticism that can be leveled at this proposal is that it can be difficult for the user to check the validity of the results. Indeed, the results returned by SQL views can contain hundreds or thousands of tuples. The problem can be easily understood by considering the following example:

Example 1. The loyalty program of an academy awards an intensive course for students that satisfy the following constraints:
- The student has completed the basic level course (level = 0).
- The student has not completed an intensive course.

```
create or replace view standard(student, level, pass) as
   select R.student, C.level, R.pass
   from courses C, registration R
   where C.id = R.course;

create or replace view basic(student) as
   select S.student
   from standard S
   where S.level = 0 and S.pass;

create or replace view intensive(student) as
   (select A1.student from allInOneCourse A1  where A1.pass)
   union
   (select A1.student
    from standard A1, standard A2, standard A3
    where A1.student = A2.student and A2.student = A3.student
          and
          A1.level = 1  and A2.level = 2 and A3.level = 3);

create or replace view awards(student) as
  select student from  basic
  where student not in (select student from  intensive);
```

Fig. 1. Views for selecting award winner students

- To complete an intensive course, a student must either pass the *all in one* course, or the three initial level courses (levels 1, 2 and 3).

The database schema includes three tables: *courses(id,level)* contains information about the standard courses, including their identifier and the course level; *registration(student,course,pass)* indicates that the *student* is in the *course*, with *pass* taking the value *true* if the course has been successfully completed; and the table *allInOneCourse(student,pass)* contains information about students registered in a special intensive course, with *pass* playing the same role as in *registration*. Figure 1 contains the SQL views selecting the award candidates. The first view is *standard*, which completes the information included in the table *registration* with the course level. The view *basic* selects those *standard* students that have passed a basic level course (level 0). View *intensive* defines as intensive students those in the *allInOneCourse* table, together with the students that have completed the three initial levels. However, this view definition is erroneous: we have forgotten to check that the courses have been completed (flag *pass*). Finally, the main view *awards* selects those students who are registered in the basic but not in the intensive courses. Suppose that we try the query select * from awards;, and that in the result we notice that the student *Anna* is missing. We know that *Anna* completed the basic course, and that although she registered in the three initial levels she did not complete one of them, and hence she is not an

intensive student. Thus, the result obtained by this query is nonvalid. A standard declarative debugger using for instance a top-down strategy [19], would ask first about the validity of the contents of *basic*, because it is the first child of *awards*. But suppose that *basic* contains hundreds of tuples, among them one tuple for *Anna*; in order to answer that *basic* is valid, the user must check that *all* the tuples in the result are the expected ones, and that there is no missing tuple. Obviously, the question about the validity of *basic* becomes practically impossible to answer.

The main goal of this report is to overcome or at least to reduce this drawback. This is done by asking for more specific information from the user. The questions are now of the type "Is there a missing answer (that is, a tuple is expected but it is not there) or a wrong answer (an unexpected tuple is included in the result)?" The answer provided by the user can be either "Yes" or "No". In the case or "No" the user can point out a wrong tuple or a missing tuple. In the example, the user indicates that *Anna* is missing in *awards*. With this information, the debugger can:

- Reduce the number of questions directed at the user. Our technique considers only those relations producing/losing the wrong/missing tuple. In the example, the debugger checks that the result of *awards* depends on the content of *basic* and *intensive* and also that there is a tuple for *Anna* in both of them (*intensive* and *basic*). This means that either the view *awards* is erroneous or the tuple for *Anna* in *intensive* is wrong. Consequently the debugger disregards *basic* as a possible error source, reducing the number of questions.
- The questions directed at the user about the validity in the children nodes can be simplified. For instance, the debugger only considers those tuples that are needed to produce the wrong or missing answer in the parent. In the example, the tool would ask if *Anna* was expected in *intensive*, without considering the validity of the rest of the tuples in this view.

Another novelty of our approach is that the computation tree is represented using Horn clauses, which allows us to include the information obtained from the user during the session. This leads to a more flexible and powerful framework for declarative debugging that can now be combined with other diagnosis techniques. We have implemented these ideas in the system DES [16].

The next Section presents some basic concepts used in the rest of the report. Section 3 introduces the debugging algorithm that constitutes the main contribution of this work. Section 4 proves the theoretical results supporting our technique. The implementation is discussed in Section 5. Finally, Section 6 presents the conclusions and proposes future work.

2 Declarative Debugging in SQL: a first approach

In this Section, we summarize the main results of [4] by describing the basic concepts of declarative debugging applied to SQL views. This first approach to debug SQL views will be refine in Section 3.

2.1 Basic Concepts of Relational Databases

A *relation schema* $\mathcal{R}$ consists of a list of attributes $(A_1, \ldots, A_n)$. Each attribute A_i has an associated domain (*integer*, *string*, ...) denoted as $dom(A_i)$.

The domain of $\mathcal{R}$ is defined as $dom(\mathcal{R}) = dom(A_1) \times \cdots \times dom(A_n)$. A *relation* or *relation instance* R of relation schema $\mathcal{R}$ is a multiset of elements in $dom(\mathcal{R})$.

A *tuple* t of schema $\mathcal{R}$ is an element in $dom(\mathcal{R})$. Duplicate tuples are allowed in a relation. The multiplicity of a tuple t in the relation instance R is denoted as $|R|_t$. A tuple t is an element of relation R if its multiplicity is greater than zero. In this case, we say that $t \in R$. If the multiplicity of t in R is zero, we say that $t \notin R$.

Each tuple t in the relation instance R can be considered as a function such that $dom(t) = \{A_1, \ldots, A_n\}$, with $t(A_i)$ the value of the attribute A_i in t. The number of attributes of t is denoted as $length(t)$. In general, we will be interested in tuples that combine attributes from different relation instances, usually as result of cartesian products. In this case, we qualify the attributes names by relation names using the notation $t(R.A_i)$ instead of $t(A_i)$.

The concatenation of two tuples t, s with disjoint domain is defined as the union of both functions represented as $t \odot s$. Given a tuple t and an arithmetic expression e defined on the attributes in $dom(t)$, we use the notation $e(t)$ to represent the value obtained applying the substitution t to e. Given a secuence $l = (e_1, \ldots, e_m)$ of arithmetic expressions defined on the attributes in $dom(t)$ with $m > 1$, the projection $\pi_l(t)$ is defined as a new tuple of the form $(e_1(t), \ldots, e_m(t))$.

The notation t_i represents the i-th position in the tuple. In our setting, *partial* tuples are tuples that might contain the special symbol $\perp$ in some of their components and *total* in other case. The set of defined positions of a partial tuple t, $def(t)$, is defined by $p \in def(t) \Leftrightarrow t_p \neq \perp$. We say that $t =_\perp t'$ if $length(t) = length(t')$ and $t_p = t'_p$ for every $p \in (def(t) \cap def(t'))$. Membership with partial tuples is defined as follows: if t is a partial tuple, and S a set of tuples with the same number of positions as t, we say that $t \in_\perp S$ if there is a tuple $t' \in S$ such that $t =_\perp t'$. Otherwise we say that $t \notin_\perp S$.

In SQL, all data are stored and accessed via relations of a particular relation schema. Relations that store data are called *tables*. Other relations do not store data, but are computed by applying relational operations to other relations. These relations are called *views* or *queries*. In implementations, views can be thought of as new tables created dynamically from existing ones by using a SQL queries. The general syntax of a SQL view is: create view $\mathsf{V}(A_1, \ldots, A_n)$ as Q, with Q a query and $A_1, \ldots, A_n$ the names of the view attributes.

Consider a *database schema* D as a tuple $(\mathcal{T}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables and $\mathcal{V}$ is a finite set of views. A *database instance* d of a database schema D is a set of table instances T, with T in $\mathcal{T}$. Sometimes, we use the notation $d(T)$ to refers to the relation instance T in d.

The syntax of SQL queries can be found in [20]. We distinguish between *basic queries* and *compound queries*. A basic query Q contains both select and from

sections in its definition with the optional where, group by and having sections. For instance, the query associated to the view *standard* in the example of Figure 1 is a basic query. A compound query Q combines the results of two queries Q_1 and Q_2 by means of set operators union [all], except [all][1] or intersect [all] (the keyword all indicates that the result is a multiset). We assume that the queries defining views do not contain subqueries. Translating queries into equivalent definitions without subqueries is a well-known transformation (see for instance [7]). For convenience, our debugger transforms basic queries into compound queries when necessary. For instance, the query defining view *awards* in the Figure 1 is transformed into the following query:

```
select student from basic
except
select student from intensive;
```

The semantics of SQL assumed in this report is given by the Extended Relational Algebra (ERA) [12], an operational semantics allowing aggregates, views, and most of the common features of SQL queries.

In ERA, each relation R of relation schema $\mathcal{R}$ is defined as a multiset of tuples in $dom(\mathcal{R})$ and it is considered as a relational expression. Similar to the notation for multiset relations, the multiplicity of a tuple t in a multiset expression R is denoted as $|R|_t$. Multisets can be denoted as a collection of individual tuples t, possibly containing duplicates, or as a set of pairs $(t, |R|_t)$ without duplicates. For convenience, next definitions refer to multisets denoted as a set of pairs $(t, |R|_t)$. ERA defines new relations by means multiset expressions. These expressions combine previously defined relations using set and multiset operators.

Next we introduce some of the operators needed to understand our work (see [10,12] for a formal definition of each operator). Let M_1 and M_2 be relational expressions of relation schema $\mathcal{R}$ and let M_3 be a relational expression of relation schema $\mathcal{R}'$. Then the following operations are relational expressions:

- The *union* $M_1 \cup_{\mathcal{M}} M_2$ collects the elements of M_1 and M_2 into a new multiset of tuples in $dom(\mathcal{R})$:

$$M_1 \cup_{\mathcal{M}} M_2 = \{(t, |M_1|_t + |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *intersection* $M_1 \cap_{\mathcal{M}} M_2$ produces a new multiset of tuples in $dom(\mathcal{R})$ consisting of the elements that are both in M_1 and M_2:

$$M_1 \cap_{\mathcal{M}} M_2 = \{(t, min(|M_1|_t, |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *difference* $M_1 \setminus_{\mathcal{M}} M_2$, "subtract" the contents of M_2 from the contents of M_1 into a new multiset of tuples in $dom(\mathcal{R})$:

$$M_1 \setminus_{\mathcal{M}} M_2 = \{(t, max(0, |M_1|_t - |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

- The *cartesian product* $M_1 \times M_3$, forms the cartesian product of the elements of M_1 and M_3 producing the new multiset of tuples in $dom(\mathcal{R}) \odot dom(\mathcal{R}')$:

$$M_1 \times M_3 = \{(t \odot t', M_1(t) \cdot M_3(t')) \mid t \in dom(\mathcal{R}),\ t' \in dom(\mathcal{R}')\}$$

[1] The Oracle database systems uses MINUS instead of the SQL standard EXCEPT.

– The *selection* $\sigma_\varphi(M_1)$ selects from a multiset that meet a condition φ defined on individual tuples in $dom(\mathcal{R})$, producing the following new multiset of tuples in $dom(\mathcal{R})$:

$$\sigma_\varphi(M_1) = \{(t, |M_1|_t) \mid t \in dom(\mathcal{R}) \wedge \varphi(t)\} \cup \{(t, 0) \mid t \in dom(\mathcal{R}) \wedge \neg\varphi(t)\}$$

In this case, φ can be seen as a function from $dom(\mathcal{R})$ into the boolean domain.

– The *projection* $\Pi_l(M_1)$ projects a multiset M_1 on the elements in the sequence l, where l can contain attributes from M_1 as well as arithmetic expressions defined on the attributes from M_1. These arithmetic expressions can be seen as functions from $\mathcal{R}$ into a basic domain. The result is a new multiset with schema $\Pi_l(\mathcal{R})$:

$$\Pi_l(M_1) = \{(s, \Sigma_{\varphi(t')}(|M_1|_{t'})) \mid t \in dom(\mathcal{R})\}$$

where $s = \pi_l(t)$ and $\varphi(t') \equiv t' \in dom(\mathcal{R}) \wedge \pi_l(t') = \pi_l(t)$. The summation $\Sigma_{\varphi(y)}(|M|_y)$ is to be interpreted as the sum of $|M|_y$ for all y satisfying the condition $\varphi(y)$.

– The *renaming* expression $\rho_R(M_1)$ returns a new multiset R with schema $\mathcal{R}$:

$$R = \{(t, |M_1|_t) \mid t \in dom(\mathcal{R})\}$$

The expression $\rho_{R(B_1/A_1,\ldots,B_n/A_n)}(M_1)$ returns a new multiset R with schema $(B_1, \ldots, B_n)$:

$$R = \{(s_t, |M_1|_t) \mid t \in dom(\mathcal{R})\}$$

where $dom(s_t) = \{B_1, \ldots, B_n\}$, $dom(B_i) = dom(A_i)$ and $s_t(B_i) = t(A_i)$
The rename operator is used to give a name as well as a new schema to relational expressions.

In our setting we allow the set operators union, intersection and difference. However the result of these operations is considered as a multiset.

– The union $M_1 \cup M_2$ collects the elements of M_1 and M_2 into a new multiset with schema $\mathcal{R}$:

$$M_1 \cup M_2 = \{(t, min(1, |M_1|_t + |M_2|_t)) \mid t \in dom(\mathcal{R})\}$$

– The intersection $M_1 \cup M_2$ produces a multiset with schema $\mathcal{R}$ consisting of the elements that are both in M_1 and M_2:

$$M_1 \cap M_2 = \{(t, min(1, |M_1|_t, |M_2|_t) \mid t \in dom(\mathcal{R})\}$$

– The *difference* $M_1 \setminus M_2$, "subtract" the contents of M_2 from the contents of M_1 into a new multi-set with schema $\mathcal{R}$:

$$\begin{aligned} M_1 \setminus M_2 = &\{(t, 1) \mid t \in dom(\mathcal{R}) \wedge |M_1|_t > 0 \wedge |M_2|_t = 0\} \cup \\ &\{(t, 0) \mid t \in dom(\mathcal{R}) \wedge |M_2|_t > 0\} \end{aligned}$$

In the rest of the report, we consider multisets denoted as a collection of individual tuples t, possibly containing duplicates.

We use the notation Φ_R for representing the ERA expression associated to a SQL relation R (table, query or view). For instance, in the case of SQL queries, the select, from and where sections correspond to the projection operation, cartesian product operation and selection condition of ERA respectively. The other sections such as group by and having corresponds to an aggregate expression in ERA as shown in [12]. The SQL set and multiset operators union [all], except [all] and intersect [all] correspond to the set and multiset operations in ERA. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table. SQL views are represented by means the rename operator of ERA.

A query/view usually depends on previously defined relations, and sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on relations $R_1, \ldots, R_n$.

Example 2. The ERA expressions associated with the views defined in Figure 1 are:

$$\Phi_{standard} = \rho_{standard(student/R.student,level/C.level,pass/R.pass)}(\\ \Pi_{R.student,C.level,R.pass}(\sigma_{C.id=R.course}(\rho_C(courses) \times \rho_R(registration))))$$

$$\Phi_{basic} = \rho_{basic(student/S.student)}(\ \Pi_{S.student}(\ \sigma_{S.level=0\ \wedge\ S.pass}(\ \rho_S(standard\ )))\ )$$

$$\Phi_{intensive} = \rho_{intensive(student/A_1.student)}(\\ (\ \Pi_{A_1.student}(\ \sigma_{A_1.pass}(\ \rho_{A_1}(allInOneCourse\ )))\ ) \\ \cup \\ (\ \Pi_{A_1.student}(\sigma_{Cond}(\rho_{A_1}(standard) \times \rho_{A_2}(standard) \times \rho_{A_3}(standard)))))$$

$$\text{where } Cond \equiv (\ A_1.student = A_2.student\ \wedge\ A_2.student = A_3.student \\ \wedge\ A_1.level = 1\ \wedge\ A_2.level = 2\ \wedge\ A_3.level = 3\ )$$

$$\Phi_{awards} = \rho_{awards}(\ \Pi_{student}(basic) \setminus\ \Pi_{student}(intensive)\ )$$

Definition 1. *The* computed answer *of an ERA expression Φ_R with respect to some schema instance d is denoted by $\| \Phi_R \|_d$, where:*

- *If R is a database table, $\| \Phi_R \|_d = R$.*
- *If R is a database view or a query and R depends on the relations $R_1, \ldots, R_n$, then $\| \Phi_R \|_d = \Phi_R(M_1, \ldots, M_n)$, where $M_i = \| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.*

meaning that the computed answer of an expression Φ_R in the instance d is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its computed answer $\| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.

The parameter d indicating the database instance is omitted in the rest of the presentation whenever it is clear from the context.

Queries are executed by SQL systems. The answer for a query Q in an implementation is represented by $\mathcal{SQL}(Q)$. The notation $\mathcal{SQL}(R)$ abbreviates $\mathcal{SQL}(\mathsf{select\ *\ from}\ R)$. In particular, we assume in this report the existence of *correct* SQL implementations.

Definition 2. *A* correct *SQL implementation verifies that* $\mathcal{SQL}(Q) = \parallel \Phi_Q \parallel$ *for every query* Q.

2.2 Declarative Debugging Framework

In the rest of the report, D represents the database schema, d the current instance of D, and R a relation in D.

We assume that the user can check if the computed answer for a relation matches its intended answer.

Definition 3. *The* intended answer *for a relation* R *w.r.t.* d*, is a multiset denoted as* $\mathcal{I}(R)$ *containing the answer that the user expects for the query* select * from R *in the instance* d.

This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging.

Definition 4. *We say that* $\mathcal{SQL}(R)$ *is an* unexpected answer *for a relation* R *if* $\mathcal{I}(\mathrm{R}) \neq \mathcal{SQL}(\mathrm{R})$.

Observe that $\mathcal{I}(R) \neq \mathcal{SQL}(R)$ means that there is some tuple t such that $|\mathcal{I}(R)|_t \neq |\mathcal{SQL}(R)|_t$. The existence of an unexpected answer implies the existence of either a *wrong tuple* or a *missing tuple*.

Definition 5. *We say that* t *is a* wrong tuple *for a relation* R *if:*

$$|\mathcal{SQL}(R)|_{\mathrm{t}} > 0 \text{ and } |\mathcal{I}(\mathrm{R})|_{\mathrm{t}} < |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$$

Definition 6. *We say that* t *is a* missing tuple *for a relation* R *if:*

$$|\mathcal{I}(\mathrm{R})|_{\mathrm{t}} > 0 \text{ and } |\mathcal{I}(\mathrm{R})|_{\mathrm{t}} > |\mathcal{SQL}(\mathrm{R})|_{\mathrm{t}}$$

For instance, the intended answer for *awards* contains *Anna* once, which is represented as $|\mathcal{I}(\mathsf{awards})|_{(\mathsf{'Anna'})} = 1$. However, the computed answer does not include this tuple: $|\mathcal{SQL}(\mathsf{awards})|_{(\mathsf{'Anna'})} = 0$. Thus, *('Anna')* is a missing tuple for *awards*. As we said in the introduction, an unexpected answer produced by a relation R does not imply that R is erroneous. In order to define the key concept of erroneous relation we need the following auxiliary concept.

Definition 7. *Let* R *be either a query or a relation. The* expectable *answer for a relation* R *w.r.t. the instance* d*,* $\mathcal{E}(R, d)$*, is defined as:*

1. *If* R *is a table,* $\mathcal{E}(R, d) = R$.
2. *If* R *is a view, then* $\mathcal{E}(R, d) = \mathcal{E}(Q, d)$*, with* Q *the query defining* R.

3. *If R is a query and $R_1, \ldots, R_n$ the relations ocurring in R, then $\mathcal{E}(R, d) = \Phi_R(I_1, \ldots, I_n)$ where $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$, meaning that $\mathcal{E}(R, d)$ is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its intended answer $\mathcal{I}(R_i)$ for $i = 1 \ldots n$.*

In the rest of the report we use $\mathcal{E}(R)$ instead of $\mathcal{E}(R, d)$ if d is clear from the context. Thus, if R is a table, the expectable answer of R is the table instance R. In the case of a view V, the expectable answer corresponds to the expectable answer for the query Q defining V. In the case of a query Q, the expectable answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of Q contain the intended answers.

A discrepancy between $\mathcal{I}(R)$ and $\mathcal{E}(R)$ indicates that R does not compute its intended answer, even assuming that all the relations it depends on correspond to their intended answers. Such relation is called *erroneous*.

Definition 8. *We say that a relation R is* erroneous *when $\mathcal{I}(R) \neq \mathcal{E}(R)$, and* correct *otherwise.*

In our running example, the intended answer for *awards* contains *Anna* once, which is represented as $|\mathcal{I}(\mathsf{awards})|_{(\text{'Anna'})} = 1$. As we said in the introduction, we know that *Anna* completed the basic course, that is the intended answer for *basic* contains *Anna* once, and that Anna is not an intensive student, that is the intended answer for *intensive* does not contain *Anna*. Then, following Definition 7, *Anna* is in $\mathcal{E}(awards)$ with multiplicity 1, which is represented as: $|\mathcal{E}(\mathsf{awards})|_{(\text{'Anna'})} = 1$. Then, following Definition 8, relation *awards* is correct. The real cause of the missing answer for the view *awards* is the erroneous definition of the view *intensive*.

Definition 8 clarifies the fundamental concept of erroneous relation. However, it cannot be used directly for defining a practical debugging tool, because in order to point out a view V as erroneous, it would require comparing $\mathcal{I}(V)$ and $\mathcal{E}(V)$. Asking about $\mathcal{E}(V)$ to the users is unrealistic; we only can assume that they know $\mathcal{I}(V)$ but not $\mathcal{E}(V)$. Thus, the debugger requires from the user only to answer questions of the form *'Is the computed answer $\{|\ldots|\}$ the intended answer for view V?'*.

The following theorems relate the concept of erroneous relation and the concept of computed answer.

Theorem 1. *Let V be a database view and $R_1, \ldots, R_n$ the relations occurring in the query defining V such that $\mathcal{I}(R_i) = \mathcal{SQL}(R_i)$ for $i = 1 \ldots n$. Then, $\mathcal{SQL}(V)$ is unexpected iff V is erroneous.*

Proof. $\mathcal{SQL}(V)$ unexpected means that $\mathcal{I}(V) \neq \mathcal{SQL}(V)$. We prove that $\mathcal{E}(V) = \mathcal{SQL}(V)$ and the result $\mathcal{I}(V) \neq \mathcal{E}(V)$ holds. Let Q the query defining V. By Definitions 7 (2 and 3):

(1) $\qquad \mathcal{E}(V) = \Phi_V(I_1, \ldots, I_n)$ where $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$

By Definitions 2 and 1,

(2) $\mathcal{SQL}(V) = \| \Phi_V \| = \Phi_V(M_1, \ldots, M_n)$, where $M_i = \| \Phi_{R_i} \|$ for $i = 1 \ldots n$

By hypothesis and Definition 2

(3) $\mathcal{I}(R_i) = \mathcal{SQL}(R_i) = \| \Phi_{R_i} \| = M_i$ for $i = 1 \ldots n$

Then substituting the multiset I_i by the multiset M_i in (1) for $i = 1 \ldots n$, we obtain that:

(4) $\mathcal{E}(V) = \Phi_V(M_1, \ldots, M_n)$

By (2 and 4) we obtain that $\mathcal{E}(V) = \mathcal{SQL}(V)$ and the result $\mathcal{I}(V) \neq \mathcal{E}(V)$ holds. Then, by Definition 8, relation V is erroneous. □

Theorem 2. *Let T be a database table. Then, $\mathcal{SQL}(T)$ is unexpected iff T is erroneous.*

Proof. By Definition 4, $\mathcal{SQL}(T)$ unexpected means that:

(5) $\mathcal{I}(T) \neq \mathcal{SQL}(T)$

By Definitions 1 and 2,

(6) $\mathcal{SQL}(T) = T$

Combining (6) and Definition 7, item 1, $\mathcal{SQL}(T) = \mathcal{E}(T)$, and thus by (5), $\mathcal{E}(T) \neq \mathcal{I}(T)$. Then, by Definition 8, table T is erroneous. □

The debugging process is based on the results in Theorems 1 and 2. We emphasize the fact that the debugger requires from the user only to answer questions about the intended answer $\mathcal{I}(R)$, which is known by him, instead of the expectable answer $\mathcal{E}(R)$. In order to locate erroneous relations, the debugger compares the computed answer –obtained from the SQL system– and the intended answer –known by the user–.

2.3 Computation trees

In our proposal, the debugging process starts when the user finds a view R returning an unexpected answer. In a first phase, the debugger builds a *computation tree* for this view R. The definition of this structure is the following:

Definition 9. *The computation tree $CT(R)$ associated with a relation R is defined as follows:*

- *The root of $CT(R)$ is $(R \mapsto \mathcal{SQL}(R))$.*
- *For any node $N = (R' \mapsto \mathcal{SQL}(R'))$ in $CT(R)$:*
  - *If R' is a table, then N has no children.*

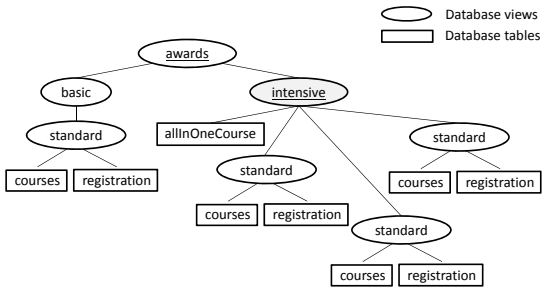


Fig. 2. Computation Tree associated with the view *awards*

- *If R' is a view, the children of N will correspond to the CTs for the relations occurring in the query associated with R'.*

Although Definition 9 includes the computed answer $\mathcal{SQL}(R)$ as part of nodes, this information is not relevant for the tree structure. In practice, this will lead to a non-efficient implementation in terms of memory usage. Instead, the computed answers are obtained from the SQL system by the debugger when needed. With this simplification, the computation tree corresponds to the dependency tree of the view R in the schema. Figure 2 shows the computation tree for our running example. After building the computation tree, the debugger will *navigate* the tree, asking the user about the validity of some nodes:

Definition 10. *Let $T = CT(R)$ be a computation tree, and $N = (R' \mapsto \mathcal{SQL}(R'))$ a node in T. We say that N is a* valid node *when $\mathcal{SQL}(R') = \mathcal{I}(R')$, a* nonvalid node *when $\mathcal{SQL}(R') \neq \mathcal{I}(R')$, and a* buggy node *when N is nonvalid and all its children in T are valid.*

The goal of the debugger is to locate buggy nodes. The next theorem shows that a computation tree with a nonvalid root always contains a buggy node, and that every buggy node corresponds to an erroneous relation.

Theorem 3. *Let V a database view and $CT(V)$ its associated computation tree. If the root of $CT(V)$ is nonvalid, then:*

- *Completeness. $CT(V)$ contains a buggy node.*
- *Soundness. Every buggy node in $CT(V)$ corresponds to an erroneous relation.*

Proof. The completeness is straightforward using induction on the number of nodes of $CT(V)$, see [13]. In order to prove the soundness, let $N = (R \mapsto \mathcal{SQL}(R))$ be a buggy node in $CT(V)$. We must prove R is erroneous. Since N is buggy, by definition 10,

$$\mathcal{SQL}(R) \neq \mathcal{I}(R) \tag{7}$$

– If R is a table, by (7) and Theorem 2, the result holds.
– If R is a view, we have that the children $N_1, \ldots, N_m$ of N correspond to the relations $R_1, \ldots, R_m$ occurring in the query associated with R. By definition 10, all of them are valid. Then, we have:

(8) $$\mathcal{I}(R_i) = \mathcal{SQL}(R_i) \text{ for } i = 1 \ldots m$$

By (7), (8) and Theorem 1, the result holds. □

2.4 Debugging session based on computation trees

Next we describe a simple debugging session based on computation trees and the result of the Theorem 3. A debugger technique following these ideas was presented in [4]. The debugger does not allow the user to point out wrong/missing tuples but it only allows the user to indicate the validity/nonvalidity of certain relations which correspond to nodes in the tree.

Suppose the user detects an unexpected result from view *awards* and he indicates to the debugger that the intended answers for table instances correspond to the actual database instance, thus trusting the instance. Then, he can type the command `/debug_sql` to debug the view *awards*, as follows:

```
DES-SQL> /debug_sql awards
```

The debugger builds internally the computation tree of the view *awards* (see Figure 2), and starts the debugging session:

```
Info: Debugging view 'basic'.
{
  basic('Anna'),
  basic('Carla'),
    ....
  basic('James')
}
Input: Is this view valid? (y/n/a) [y]: y

Info: Debugging view 'intensive'.
{
  intensive('Alba'),
  intensive('Anna'),
    ....
  intensive('James')
}

Input: Is this view valid? (y/n/a) [y]: n

Info: Debugging view 'standard'.
{
  standard('Alba',1,0),
  standard('Alba',2,1),
```

```
  standard('Anna',0,1),
  standard('Anna',1,1),
  standard('Anna',2,1),
  standard('Anna',3,0),
  standard('Carla',0,1),
  standard('James',0,1),
  standard('James',1,1),
    ...
  standard('James',3,1)
}

Input: Is this view valid? (y/n/a) [y]: y
Info: Buggy view found: intensive/1.
```

The debugger marks the root of the tree as a nonvalid node (the computed answer of the view *awards* is an unexpected answer). The first question is about the validity of *basic*. Suppose the user checks the answer produced by the SQL system as valid. In this case, the children of *basic* are not considered anymore. The second question is about the validity of the view *intensive*. The computed answer of the view *intensive* produced by the SQL system contains Anna, and the user checks this answer as nonvalid. Next, the children of *intensive* are visited. The only child of *intensive* that is not a table is *standard*. As this is checked as valid, the debugger points out node *intensive* as buggy; node *intensive* is marked as nonvalid and all its children are marked as a valid nodes. In the tree of Figure 2, nonvalid nodes are underlined and the only buggy node (a shader node) corresponds to view *intensive*. The tool has asked three questions to the user. However, note that these questions can be difficult to answer when the computed answer contains hundreds or thousands of tuples.

In the next Section we improve the debugging technique presented in this Section by allowing the user to specify if an unexpected answer contains a wrong or a missing tuple.

3 Improved Debugging Algorithm

This Section refines the ideas for debugging SQL views presented so far. Although the process is based on the ideas of declarative debugging, this proposal does not use computation trees explicitly. Instead, our debugger represents the logic inferences defining buggy nodes in computation trees by means of Horn clauses, denoted as $\mathsf{H} \leftarrow \mathsf{C}_1, \ldots, \mathsf{C}_n$, where the comma represents the conjunction, and $\mathsf{H}, \mathsf{C}_1, \ldots, \mathsf{C}_n$ are positive atoms. As usual, a *fact* H stands for the clause $\mathsf{H} \leftarrow \mathsf{true}$.

This Section starts off describing some auxiliary functions that define the debugging algorithm. Next, we present the general schema of the algorithm. This Section ends discussing how the debugger process the user answer for detecting errors.

3.1 Auxiliary functions

We describe some auxiliary functions that define the debugging algorithm, although the code of some basic auxiliary functions is omitted. Some of them are the following:

Code 1 debug(V)

```
Input: V: view name
Output: A list of buggy views
 1: A := askOracle(all V)
 2: if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
 3:    Valid := true
 4:    P := initialSetOfClauses(V, A)
 5:    while getBuggy(P)=[] do
 6:       LE := getUnsolvedEnquiries(P)
 7:       E := chooseEnquire(LE)
 8:       A := askOracle(E)
 9:       Valid := checkAnswer(A)
10:       if Valid then P := P ∪ processAnswer(E,A)
11:    end while
12:    L := getBuggy(P)
13: else
14:    L := [ ]
15: end if
16: return L
```

- Functions *getSelect* y *getWhere* return the different sections of a SQL query.
- The result of the function *getFrom* is a sequence of elements of the form *R as R'* (assuming that every relation R in the from section of a SQL query has an alias R').
- A boolean expression like *getGroupBy(Q)*=[] is satisfied if the query Q has no group by section.
- Function *getRelations(R)* returns the set of relations involved in the relation R. It can be applied to queries, tables and views:
  - If R is a table, then *getRelations(R)* = $\{R\}$.
  - If R is a query, then *getRelations(R)* is the set of relations occurring in the definition of the query.
  - If R is a view, then *getRelations(R)* = *getRelations(Q)*, with Q the query defining R.
- Function *generateUndefined(R)* generates a new tuple with length the number of attributes in the relation R containing only undefined values $(\perp, \ldots, \perp)$.
- Function *checkAnswer(A)* returns the boolean value *true* if the input parameter is of the form *yes*, *no*, *missing(t)* or *wrong(t)* and *false* in other case.

3.2 Debugging schema

The general schema of the algorithm is summarized in the code of function *debug* (Code 1). The debugger is started by the user when an unexpected answer is obtained as computed answer for some SQL view *V*. In our running example, the debugger is started with the call *debug(awards)*. Then, the algorithm asks the user about the type of error (line 1). The answer A can be simply *no*, or a more detailed explanation of the error, like *wrong(t)* or *missing(t)*, indicating that t is a wrong or missing tuple respectively. In our example, A takes the initial value *missing('Anna')*. During the debugging process, variable P keeps a list of Horn clauses representing a logic program.

Code 2 initialSetofClauses(V, A)

Input: V: view name, A: answer
Output: A set of clauses

```
1: P := ∅
2: P := initialize(V)
3: P := P ∪ processAnswer((all V), A)
4: return P
```

createBuggyClause(V)
Input: V: view name
Output: A Horn clause

```
1: [R_1, ..., R_n] := getRelations(V)
2: return { buggy(V)← state((all V), nonvalid),
                      state((all R_1), valid), ..., state((all R_n), valid)). }
```

initialize(R)
Input: R: relation
Output: A set of clauses

```
1: P := createBuggyClause(R)
2: for each R_i in getRelations(R) do
3:     P := P ∪ initialize(R_i)
4: end for
5: return P
```

The atoms in the body of the clauses represent *enquiries* that might represent questions to the user. The initial list of clauses P is generated by the function *initialSetofClauses* (line 4). This function introduces the clauses that correspond to the computation tree rooted by V (which are listed partially in Figure 3 for the running example). The purpose of the main loop (lines 5-11) is to add information to the program P, until a buggy view can be inferred. The function *getBuggy* returns the list of all the relations R such that *buggy(R)* can be proven w.r.t. the logic program P. Each iteration of the loop represents the election of an enquiry in a body atom whose validity has not been established yet (lines 6-7). Then, in line 8 the debugger asks to the user about the result of the question associated to the chosen enquiry. Finally, the answer is processed (line 10). Next, we explain in detail each part of this main algorithm.

Code 2 corresponds to the initialization process called in line 4 of Code 1. The function *initialSetofClauses* gets as first input parameter the initial view V. This view has returned an unexpected answer, and the input parameter A contains the explanation. The output of this function is a set of clauses representing the logic relations that define possible buggy relations with predicate *buggy*. Initially it creates the empty set of clauses and then it calls the function *initialize* (line 2), a function that traverses recursively all the relations involved in the definition of the initial view V, calling *createBuggyClause* with V as input parameter. *createBuggyClause* adds a new clause indicating the enquiries that must hold in order to consider V as incorrect: it must be nonvalid, and all the relations it depends on must be valid.

Figure 3 shows a partial list of initial clauses for our example. The correlation between these clauses and the computation tree of Figure 2 is straightforward. Finally, in line 3, function *processAnswer* incorporates the information that can be extracted from A into the program P.

The information about the validity/nonvalidity of the results associated to enquiries is represented in our setting by predicate *state*. The first parameter of *state* is an enquiry E, and the second one can be either *valid* or *nonvalid*. The next definition determines the possible enquiries, their associated questions and answers, and a measure $\mathcal{C}$ of the complexity of the questions:

```
buggy(awards)      :- state(all(awards),nonvalid),
                      state(all(basic),valid), state(all(intensive),valid).
buggy(basic)       :- state(all(basic),nonvalid), state(all(standard),valid).
buggy(intensive)   :- state(all(intensive),nonvalid),
                      state(all(allInOneCourse),valid), state(all(standard),valid).
    ...
buggy(courses)     :- state(all(courses),nonvalid).
buggy(registration) :- state(all(registration),nonvalid).
```

Fig. 3. Initial set of clauses for the running example

Definition 11. *Enquiries can be of any of the following forms:* (all R), (s ∈ R), *or* (R' ⊆ R), *with* R, R' *relations, and* s *a tuple with the same schema as relation* R. *Each enquiry* E *corresponds to a specific question with a possible set of answers and an associated complexity* $\mathcal{C}(E)$*:*

- *If* E ≡ (all R). *Let* $S = \mathcal{SQL}(R)$. *The associated question asked to the user is* "Is S the intended answer for R?" *The answer can be either* yes *or* no. *In the case of* no, *the user is asked about the type of the error,* missing *or* wrong, *giving the possibility of providing a witness tuple* t. *If the user provides this information, the answer is changed to* missing(t) *or* wrong(t), *depending on the type of the error. We define* $\mathcal{C}(E) = |S|$, *with* $|S|$ *the number of tuples in* S.
- *If* E ≡ (R' ⊆ R). *Let* S = $\mathcal{SQL}$(R'). *Then the associated question is* "Is S included in the intended answer for R?" *As in the previous case the answers allowed are* yes *or* no. *In the case of* no, *the user can point out a wrong tuple* t ∈ S *and the answer is changed to* wrong(t). $\mathcal{C}(E) = |S|$ *as in the previous case.*
- *If* E ≡ (s ∈ R). *Tuple* s *can be a partial tuple. The question is* "Does the intended answer for R include a tuple matching the tuple s?" *The possible answers are* yes *or* no. *No further information is required from the user. In this case* $\mathcal{C}(E) = 1$, *because only one tuple must be considered.*

In the case of *wrong* answers, the user typically points to a tuple in the result R. In the case of *missing* answers, the tuple must be provided by the user, and in this case *partial* tuples, i.e., tuples including some undefined attributes are allowed. The answer *yes* corresponds to the state *valid*, while the answer *no* corresponds to *nonvalid*. An atom *state(q,s)* occurring in the body of a clause in P implies an enquiry q. The enquiry q is a *solved enquiry* if the logic program P contains at least one fact of the form *state(q, valid)* or *state(q, nonvalid)*, that is, if the enquiry has been already solved. The enquiry q is called an *unsolved enquiry* otherwise. The function *getUnsolvedEnquiries* (see line 6 of Code 1) returns in a list all the unsolved enquiries occurring in body atoms of clauses in P. The function *chooseEnquiry* (line 7, Code 1) chooses one of these enquiries according to some predefined criteria. Our current implementation chooses the enquiry E that implies the smaller complexity value $\mathcal{C}(E)$, although other more elaborated criteria could be defined without affecting the theoretical results supporting the technique. Once the enquiry has been chosen, Code 1 uses the function *askOracle* (line 8) in order to ask to the user the associated question, returning the answer.

3.3 Processing user answers and detecting errors

Code 3 processAnswer(E,A)

Input: E: enquiry, A: answer obtained for the enquiry
Output: A set of new clauses

```
 1: if A ≡ yes then
 2:    P := {state(E,valid).}
 3: else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t) then
 4:    P := {state(E,nonvalid).}
 5: end if
 6: if E ≡ (s ∈ R) and (A ≡ yes) then
 7:    P:= P ∪ processAnswer((all R),missing(s))
 8: else if E ≡ (V ⊆ R) and (A ≡ wrong(s) or A ≡ no) then
 9:    P:= P ∪ processAnswer((all R), A)
10: else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t)) then
11:    Q := SQL query defining V
12:    P := P ∪ slice(V,Q,A)
13: end if
14: return P
```

Function *processAnswer* process the user answer distinguishing several cases depending on the form of its associated enquiry. The code of function *processAnswer* (called in line 10 of Code 1 and in line 3 of Code 2), can be found in Code 3. The first lines (1-5) introduce a new logic fact in the program with the state that corresponds to the answer A obtained for the enquiry E. In our running example, *debug(awards)* calls to *initialSetofClauses(awards, missing(('Anna')))* which calls to *processAnswer(all(awards), missing(('Anna')))* which adds the fact *state(all(awards), nonvalid)* to the program. The rest of the code distinguishes several cases depending on the form of the enquiry and its associated answer. If the enquiry is of the form *(s ∈ R)* with answer *yes* (line 6), then s is missing in the relation R. Notice, the enquiry *(s ∈ R)* is associated to a body atom of the form *state((s ∈ R), nonvalid)* of a clause added in P by the function *missingBasic* when the debugger checks that the tuple s is not in the computed answer of the view R (Code 5, line 8).

For enquiries of the form *(V ⊆ R)* and answer *wrong(s)*, it can be ensured that s is wrong in R (line 9). If the enquiry is of the form *(V ⊆ R)* with answer *no*, then R is a nonvalid relation. If the enquiry is *(all V)* for some view V, and with an answer including either a wrong or a missing tuple, the function *slice* (line 12) is called. This function exploits the information contained in the parameter A (*missing(t)* or *wrong(t)*) for slicing the query Q in order to produce, if possible, new clauses which might allow the debugger to detect incorrect relations by asking simpler questions to the user. The implementation of *slice* can be found in Code 4. The function receives the view V, a subquery Q, and an answer A as parameters. Initially, Q is the query defining V, and A the user answer, but this situation can change in the recursive calls. The function distinguishes several particular cases:

- The query Q combines the results of Q_1 and Q_2 by means of either the operator union or union all, and A is *wrong(t)* (first part of line 2). Then query Q produces too many copies of t. Then, if any Q_i produces as many copies of t as Q, we can blame Q_i as the source of the excessive number of t's in the answer for V (lines 4 and 5). The case of subqueries combined by the operator intersect [all], with A

Code 4 slice(V,Q,A)

Input: V: view name, Q: query, A: answer
Output: A set of new clauses

```
 1: P := ∅;    S= SQL(Q);
 2: if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
       (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2) then
 3:    S1= SQL(Q1);    S2= SQL(Q2);
 4:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 5:    if |S2|t = |S|t then  P:= P ∪ slice(V, Q2, A)
 6: else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2  then
 7:    S1= SQL(Q1);    S2= SQL(Q2);
 8:    if |S1|t = |S|t then  P:= P ∪ slice(V, Q1, A)
 9:    if Q ≡ Q1 except Q2 and t ∈⊥ S2 then P :=P∪ slice(V,Q2,wrong(t))
10: else if basic(Q) and groupBy(Q)=[] then
11:    if A ≡ missing(t) then P := P ∪ missingBasic(V, Q, t)
12:    else if A ≡ wrong(t) then P := P ∪ wrongBasic(V, Q, t)
13: end if
14: return  P
```

≡ *missing(t)* is analogous, but now detecting that a subquery is the cause of the scanty number of copies of t in $\mathcal{SQL}(V)$.

- The query Q is of the form Q_1 except [all] Q_2, with $A \equiv$ *missing(t)* (line 6). If the number of occurrences of t in both Q and Q_1 is the same, then t is also missing in the query Q_1 (line 8). Additionally, if query Q is of the particular form Q_1 except Q_2, which means that we are using the difference operator on sets (line 9), then if t is in the result of Q_2 it is possible to claim that the tuple t is wrong in Q_2. Observe that in this case the recursive call changes the answer from *missing(t)* to *wrong(t)*.
- If Q is defined as a basic query without group by section (line 10), then either function *missingBasic* or *wrongBasic* is called depending on the form of A.

Both *missingBasic* and *wrongBasic* can add new clauses that allow the system to infer buggy relations by posing questions which are easier to answer. Function *missingBasic*, defined in Code 5, is called (line 11 of Code 4) when A is *missing(t)*. The input parameters are the view V, a query Q, and the missing tuple t. Notice that Q is in general a component of the query defining V. For each relation R with alias S occurring in the from section of Q, the function checks if R contains some tuple that might produce the attributes of the form $S.A$ occurring in the tuple t. This is done by constructing a tuple s undefined in all its components (line 4) except in those corresponding to the select attributes of the form $S.A$, which are defined in t (lines 5 - 7). If R does not contain a tuple matching s in all its defined attributes (line 8), then it is not possible to obtain the tuple t in V from R. In this case, a buggy clause is added to the program P (line 9) meaning that if the answer to the question *"Does the intended answer for R include a tuple s?"* is *no*, then V is an incorrect relation.

The implementation of *wrongBasic* can be found in Code 6. The input parameters are again the view V, a query Q, and a tuple t. In line 1, this function creates an empty set of clauses. In line 2, variable F stands for the set containing all the relations in the from section of the query Q. Next, for each relation $R_i \in F$ (lines 4 - 7), a new view V_i is created in the database schema after calling the function *relevantTuples* (line 6),

Code 5 missingBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A new list of Horn clauses

```
 1: P := ∅;    S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
 2: if t ∉⊥ S then
 3:    for (R AS S) in (getFrom(Q)) do
 4:       s = generateUndefined(R)
 5:       for i=1 to length(getSelect(Q)) do
 6:          if t_i ≠⊥ and member(getSelect(Q),i) = S.A, A attrib., then s(R.A) = t_i
 7:       end for
 8:       if s ∉⊥ SQL(R) then
 9:          P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
10:       end if
11:    end for
12: end if
13: return P
```

Code 6 wrongBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A set of clauses

```
1: P := ∅
2: F := getFrom(Q)
3: N := length(F)
4: for i=1 to N do
5:    R_i as S_i := member(F,i)
6:    relevantTuples(R_i,S_i,V_i, Q, t)
7: end for
8: P := P ∪ { (buggy(V) ← state((V_1 ⊆ R_1), valid), ..., state((V_n ⊆ R_n), valid).) }
9: return P
```

which is defined in Code 7. This auxiliary view contains only those tuples in relation R_i that contribute to produce the wrong tuple t in V. Finally, a new buggy clause for the view V is added to the program P (line 8) explaining that the relation V is buggy if the answer to the question associated to each enquiry of the form $V_i \subseteq \mathrm{R}_i$ is *yes* for $i \in \{1 \ldots n\}$.

4 Theoretical Results

In the previous Section we have introduced the debugging algorithm, explaining the intuitive ideas supporting the technique. Now we establish formally the correctness and the completeness of the proposal. In the rest of the Setion we assume that the debugging algorithm uses a SLD-based logic system for checking the atoms that are entailed by the program contained in the variable P of Code 1. The notation $\mathsf{P} \vdash \mathsf{A}$ denotes that there is a SLD proof for A with respect to the program P.

Code 7 relevantTuples(R_i,R',V,Q,t)

Input: R_i: relation, R': alias,
V: new view name, Q: Query, t: tuple
Output: A new view in the database schema

```
1: Let A_1, ..., A_n be the attributes defining R_i
2: SQL(create view V as
     (select R'.A_1, ..., R'.A_n from R_i as R')
        intersect all
     (select R'.A_1, ..., R'.A_n from getFrom(Q)
     where getWhere(Q) and eqTups(t,getSelect(Q))))
```

eqTups(t,s)

Input: t,s : tuples
Output: SQL condition

```
1: C := true
2: for i=1 to length(t) do
3:     if t_i ≠ ⊥ then
4:         C:= C AND t_i = s_i
5: end for
6: return C
```

4.1 Correctness

We start checking the correctness of the framework.

Theorem 4. *Correctness.*
Let R be a relation and L the list returned by debug(R) *(defined in Code 1). If the user answers correctly all the questions performed by the debugger, then every relation contained in L is erroneous (according to Definition 8).*

Proof. We prove the correctness of our technique using some auxiliary results (see Figure 4).

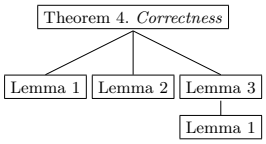


Fig. 4. The proof correctness structure

Lemma 1 proves properties of the new view created by function relevantTuples.
Lemma 2 establishes the relationship between enquiries and answers.
Lemma 3 indicates how state relates the answers obtained by the SQL system and the intended interpretation $\mathcal{I}$.

Let $\mathcal{P}$ be the logic program contained in the variable P of Code 1. Then the list L returned by debug(R) contains all the database relations A such that:

$$\mathcal{P} \vdash buggy(A) \tag{9}$$

Let A be any of the relations verifying (9). We prove that A is erroneous according to Definition 8.

The SLD inference proving *buggy(A)* must start using a clause with head *buggy(A)*. Notice A is a relation name, and therefore *buggy(A)* is a ground atom. The algorithm code introduces clauses for predicate *buggy* at three points:

1.- In Code 2 (function *createBuggyClause*, line 2). In this point, the added clause is:

– If A is a view:

$\mathsf{buggy(A)}\leftarrow \mathsf{state((all\ A),\ nonvalid)}, \mathsf{state((all}\ R_1\mathsf{),\ valid)}, \ldots, \mathsf{state((all}\ R_n\mathsf{),\ valid)}.$

where $R_1, \ldots, R_n$ are all the relations employed in the definition of the view A. Then, by Lemma 3:

(10) $\mathcal{P} \vdash \mathit{state((all\ R_i),\ valid)} \Longrightarrow \mathcal{SQL}(R_i) = \mathcal{I}(R_i)$ for $i = 1 \ldots n$

(11) $\mathcal{P} \vdash \mathit{state((all\ A),\ nonvalid)} \Longrightarrow \mathcal{SQL}(A) \neq \mathcal{I}(A)$

By (11) and Definition 4, $\mathcal{SQL}(A)$ is unexpected, which means, by (10) and Theorem 1, that A is erroneous.

– If A is a table, the added clause is:

$$\mathsf{buggy(A)}\leftarrow \mathsf{state((all\ A),\ nonvalid)}.$$

Then, by Lemma 3:

(12) $\mathcal{P} \vdash \mathit{state((all\ A),\ nonvalid)} \Longrightarrow \mathcal{SQL}(A) \neq \mathcal{I}(A)$

By (12) and Definition 4, $\mathcal{SQL}(A)$ is unexpected and the result is straightforward from Theorem 2.

2.- In Code 5, line 9, which introduces the clause:

$$(\mathsf{buggy(A)} \leftarrow \mathsf{state((s \in R),\ nonvalid)})$$

Then $\mathcal{P} \vdash \mathit{state((}\mathsf{s} \in \mathsf{R}\mathit{),\ nonvalid))}$, and:

(13) By Code 5, line 8, $s \notin_\perp \mathcal{SQL}(R)$

(14) By Lemma 3, $s \notin_\perp \mathcal{I}(R)$

The input parameters of the function *missingBasic* defined in Code 5 are the view A, a query Q and a tuple t indicating that t is missing in A. Notice that the query Q is in general a component of the query defining the view A. The function *missingBasic* is called from function *slice* (Code 4) which ensures that Q is defined by a basic query without group by section. Next we prove that $|\mathcal{E}(Q)|_t \leq |\mathcal{SQL}(Q)|_t$ and by applying Lemma 2 we conclude that A is an incorrect view.

Examining the code of the function *missingBasic* it is clear that R AS S is in the from section and the *select* section contains $m \geq 1$ values of the form $S.A_1, \ldots, S.A_m$ with tuple s of the form: $s(R.A_i) = t(S.A_i) \neq \perp$ for $i = 1 \ldots m$ (otherwise the tuple s will be completely undefined and the condition $s \notin_\perp \mathcal{SQL}(R)$ could not hold).

Therefore, the ERA expression associated to the query Q can be written as

(15) $$\Phi_Q = \Pi_{(\ldots S.A_1, \ldots, S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(R) \times \ldots))$$

for some condition C in the where section of Q. Then, by Definitions 1 and 2:

(16) $$\mathcal{SQL}(Q) = \| \Phi_Q \| = \Pi_{(\ldots S.A_1, \ldots, S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(M) \times \ldots))$$

with $M = \mathcal{SQL}(R)$. By (13), $s \notin_\perp M$, that is there is no tuple $u \in M$ such that $s =_\perp u$. Therefore, there is no tuple $u \in M$ such that $u(R.A_i) = t(S.A_i)$ for $i = 1 \ldots m$. Then:

$$t \notin_\perp \mathcal{SQL}(Q) \tag{17}$$

Applying Definition 7 to (15) we obtain:

$$\mathcal{E}(Q) = \Pi_{(\ldots S.A_1, \ldots, S.A_m \ldots)}(\sigma_C(\cdots \times \rho_S(I) \times \ldots)) \tag{18}$$

with $I = \mathcal{I}(R)$. By (14), $s \notin_\perp I$, that is there is no tuple $u' \in I$ such that $s =_\perp u'$. Therefore, there is no tuple $u' \in I$ such that $u'(R.A_i) = t(S.A_i)$ for $i = 1 \ldots m$. Then:

$$t \notin_\perp \mathcal{E}(Q) \tag{19}$$

By (17) and (19):

$$|\mathcal{E}(Q)|_t = |\mathcal{SQL}(Q)|_t = 0 \tag{20}$$

The idea is that if the relation R does not contain a tuple matching s in all its defined attributes, then it is not possible to obtain the tuple t in $\mathcal{SQL}(Q)$ from R. Taking into account that the function *missingBasic* has been called from *slice* (Code 4, line 11), with the same input parameters A, Q, and a missing tuple t, then Lemma 2 can be applied to the call *slice(A,Q,missing(t))*, and (20) implies that A is an incorrect relation.

3.- In Code 6, line 8, which introduces the clause:

$$(\mathsf{buggy(A)} \leftarrow \mathsf{state((V_1 \subseteq R_1), valid)}, \ldots, \mathsf{state((V_n \subseteq R_n), valid)})$$

where V_i is a new database view returned by function *relevantTuples* (called in Code 6, line 6). Then $\mathcal{P} \vdash$ *state(*$\mathrm{V}_i \subseteq \mathrm{R}_i$*, valid))*, for i=1 ... n.

By Lemma 3:

$$\mathcal{SQL}(V_i) \subseteq \mathcal{I}(R_i) \qquad \text{for i=1 ... n} \tag{21}$$

Analogously to the previous case, the function *wrongBasic* is called from function *slice* (Code 4, line 12). The input parameters of the function *wrongBasic* defined in Code 6 are the view A, a query component Q of the query defining the view A (Q is defined by a basic query without group by section) and a wrong tuple t. The ERA expression of Q is of the form:

$$\Phi_Q = \Pi_{(S)}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_n}(R_n))) \tag{22}$$

where S is the list of expressions in the select section, C is the condition in the where section and R_1 *AS* $S_1, \ldots, R_n$ *AS* S_n is the sequence of elements in the from section of the query Q. Using Definitions 1 and 2 we obtain the SQL computed answer

$$\mathcal{SQL}(Q) = \| \Phi_Q \| = \Pi_{(S)}(\sigma_C(\rho_{S_1}(M_1)) \times \cdots \times \rho_{S_n}(M_n))))$$

with $M_i = \mathcal{SQL}(R_i)$ for $i = 1 \ldots n$, and in particular:

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_n}(M_n))))|_t \tag{23}$$

By Lemma 1, replacing each R_i by its corresponding V_i in the query Q does not affect to the number of copies of t obtained, that is:

$$|\mathcal{SQL}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(M'_1)) \times \cdots \times \rho_{S_i}(M'_i) \times \cdots \times \rho_{S_n}(M'_n)))|_t \quad (24)$$

with $M'_i = \mathcal{SQL}(V_i)$.

By Definition 7, item 3, we obtain:

$$\mathcal{E}(Q) = \Pi_{(S)}(\sigma_C(\rho_{S_1}(I_1)) \times \cdots \times \rho_{S_n}(I_n))))$$

with $I_i = \mathcal{I}(R_i)$ for $i = 1 \ldots n$. And in particular:

$$|\mathcal{E}(Q)|_t = |\Pi_{(S)}(\sigma_C(\rho_{S_1}(I_1)) \times \cdots \times \rho_{S_n}(I_n))))|_t \quad (25)$$

It is easy to check that in an expression like (24), replacing a multiset M'_i in the cartesian product by other multiset W such that $M'_i \subseteq W$ implies at least the same tuples in the result (and possibly more, new tuples). Therefore, applying (21) to (24) and (25):

$$|\mathcal{SQL}(Q)|_t \leq |\mathcal{E}(Q)|_t \quad (26)$$

Taking into account that the function *wrongBasic* has been called from *slice* (Code 4, line 12), with the same input parameters A, Q, and a wrong tuple t, then Lemma 2 can be applied to the call *slice(A,Q,wrong(t))*, and (26) implies that A is an incorrect relation. □

Next we prove some auxiliary Lemmata. First we prove a property of the new view created by function *relevantTuples*.

Lemma 1. *After a call of the form* relevantTuples(R_i,S_i,V,Q,t), V *is a new view such that*

1. $\mathcal{SQL}(V) \subseteq \mathcal{SQL}(\mathrm{R}_i)$
2. *Let* Q′ *be the result of replacing* R_i *by* V *in* Q. *Then* $|\mathcal{SQL}(Q)|_t = |\mathcal{SQL}(Q')|_t$

Proof.

Function *relevantTuples* is called from function *wrongBasic*. The input parameters are a query Q, a (partial or total) tuple t of the form $(t_1, \ldots, t_k)$, a relation R_i occurring in the from section of the query Q and its associated alias S_i. The output is a new auxiliary view V in the database schema containing only those tuples from R_i that contribute to produce the tuple t in the result of the query Q. The definition of V can be found in Code 7. Suppose that:

- *getFrom(Q)* = R_1 *as* S_1, ..., R_m *as* S_m,
- *getWhere(Q)* = C,
- *getSelect(Q)* = e_1, ..., e_k
- C' is the SQL condition representing the logical expression: $\bigwedge_{\substack{1 \leq j \leq k \\ t_j \neq \perp}} (e_j = t_j)$

Then, the definition of V can be represented as:

(27)

```
create view V as
 (select Si.A1, ..., Si.An from Ri as Si )
     intersect all
 (select Si.A1, ..., Si.An
  from R1 as S1, ..., Ri as Si, ..., Rm as Sm
   where C and C')
```

1. Taking into account that $R_i.A_1, \ldots, R_i.A_n$ are all the attributes of R_i, the definition of V from (27) can be represented by the following ERA expression as intersection of multisets:

   (28) $$\rho_V(\rho_{S_i}(R_i) \cap_{\mathcal{M}} \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_m}(R_m))))$$

   and therefore $\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R_i)$.

2. The function *relevantTuples* is called from function *wrongBasic* (line 6, Code 6), which is called from function *slice* (Code 4, line 12). The *if* sentence in *slice* ensures that Q is a basic query without group by section. Therefore, Q must be of the form:

   ```
   select e1, ..., ek
   from R1 as S1, ..., Ri as Si, ..., Rm as Sm
   where C
   ```

   which can be represented in ERA as:

   (29) $$\Phi_Q = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_i}(R_i) \times \cdots \times \rho_{S_m}(R_m)))$$

   Let $\Phi_{Q'}$ be the ERA expression obtained by replacing R_i by V in (29):

   (30) $$\Phi_{Q'} = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(R_1) \times \cdots \times \rho_{S_i}(V) \times \cdots \times \rho_{S_m}(R_m)))$$

   Observe that only the from section needs to be modified, because the rest of the query does not include R_i but his alias S_i, and aliases are kept unaltered in $\Phi_{Q'}$. Then:
   Using Definitions 1 and 2 we obtain the SQL computed answer of Q and Q':

   (31) $$\| \Phi_Q \| = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m)))$$

   (32) $$\| \Phi_{Q'} \| = \Pi_{e_1,\ldots,e_k}(\sigma_C(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M) \times \cdots \times \rho_{S_m}(M_m)))$$

   with $M_j = \mathcal{SQL}(R_j)$ for $j = 1 \ldots m$, and $M = \mathcal{SQL}(V)$.
   Then we must prove that $|\mathcal{SQL}(Q)|_t = |\mathcal{SQL}(Q')|_t$. In other words, by Definition 2, we must prove

   $$| \| \Phi_Q \| |_t = | \| \Phi_{Q'} \| |_t$$

   Let U be the minimum multiset such that

   (33) $$U \subseteq \rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \ldots \rho_{S_m}(M_m))$$

   (34) $$| \| \Phi_Q \| |_t = |\Pi_{e_1,\ldots,e_k}(\sigma_C(U))|_t$$

(minimum here means that is if we remove from U any occurrence of any of its tuples, (34) becomes false). Since the difference between (31) and (32) is only the replacement of M_i by M, we can concentrate on this part of the tuples of U. Observe that $M_i = \mathcal{SQL}(R_i)$. Hence we define:

$$U_{R_i} = \{(u, |M_i|_u) \mid u = \pi_{S_i.A_1,\ldots,S_i.A_n}(w), w \in U\}$$

By construction of U_{R_i}, we obtain:

$$U_{R_i} \subseteq M_i = \mathcal{SQL}(R_i) \tag{35}$$

The proof is complete is we check that $\rho_V(U_{R_i}) = M$.
By construction of U:

$$U = \sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m))$$

and from this and the construction of U_{R_i}:

$$U_{R_i} = \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_i}(M_i) \times \cdots \times \rho_{S_m}(M_m)))$$

which considering that:

$$M = \rho_V(\rho_{S_i}(M_i) \cap_{\mathcal{M}} \Pi_{S_i.A_1,\ldots,S_i.A_n}(\sigma_{C \wedge C'}(\rho_{S_1}(M_1) \times \cdots \times \rho_{S_m}(M_m))))$$

yields:

$$M = \rho_V(\rho_{S_i}(M_i) \cap_{\mathcal{M}} U_{R_i})$$

and by (35) we obtain $M = \rho_V(U_{R_i})$. □

Next we prove an auxiliary result that establishes the relationship between enquiries and answers:

Lemma 2. *Let* slice(V,Q,A) *be any call to Code 4 that occurs during the execution of the debugger. Then:*

- *If* $\mathsf{A} \equiv$ *wrong(t) and* $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$, *then* V *is an incorrect view.*
- *If* $\mathsf{A} \equiv$ *missing(t) and* $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$, *then* V *is an incorrect view.*

Proof. We prove the results by induction on the number n of recursive calls to slice occurred before the current call.

If $n = 0$, then the initial call for slice corresponds to processAnswer, Code 3, line 12. This call ensures that V is a view, Q is the query defining V, and A is either missing(t) or wrong(t), where t has been pointed out as missing (respectively wrong) by the user. By definition 4, and taking into account that $\mathcal{SQL}(\mathsf{V}) = \mathcal{SQL}(\mathsf{Q})$, we have that in this first call:

- If A is wrong(t), then $|\mathcal{I}(V)|_t < |\mathcal{SQL}(Q)|_t$. Therefore, $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$ implies $|\mathcal{E}(\mathsf{Q})|_t > |\mathcal{I}(V)|_t$.
- If A is missing(t), then $|\mathcal{I}(V)|_t > |\mathcal{SQL}(Q)|_t$. Then, $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ implies $|\mathcal{E}(\mathsf{Q})|_t < |\mathcal{I}(V)|_t$.

In both cases, and considering that from Definition 7, $\mathcal{E}(\mathsf{V}) = \mathcal{E}(\mathsf{Q})$, we have that $\mathcal{E}(\mathsf{V}) \neq \mathcal{I}(\mathsf{V})$ and according to the Definition 8, the view V is erroneous.

If $n > 0$ we suppose that the result holds for the n-th call slice(V,Q,A), and we want to check that it is also valid for the n+1-th call slice(V,Q',A'). Observe that all the recursive calls occur in Code 4 and verify that they do not change the first parameter V, which is hence the same as in the initial call. The values Q' and A', might have changed with respect to the input values Q and A. By inductive hypothesis we have that this Lemma can be applied to the input values of the $n-1$ call, slice(V,Q,A). Now we check that the result can be applied also to V, Q' and A', distinguishing cases depending on the particular call:

- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 4. In this case, considering Q' as Q_1, we have:

  $$|\mathcal{SQL}(\mathsf{Q}_1)|_t = |\mathcal{SQL}(\mathsf{Q})|_t \tag{36}$$

  Then, one of the following conditions hold:
  - A ≡ missing(t) and Q ≡ Q_1 intersect Q_2.
    Let $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \cap \Phi_{\mathsf{Q}_2}$ be the ERA expression associated to the SQL query Q. By Definition 7, we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \cap \mathcal{E}(\mathsf{Q}_2)$. Therefore,

    $$|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{E}(\mathsf{Q})|_t \tag{37}$$

    If $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (36) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (37), $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$. Then, by induction hypothesis, V is incorrect.
  - A ≡ missing(t) and Q ≡ Q_1 intersect all Q_2. Analogous to the previous point. Observe that replacing the set operator $\cap$ by $\cap_{\mathcal{M}}$ does not affect to the result.
  - A ≡ wrong(t) and Q ≡ Q_1 union Q_2.
    From $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \cup \Phi_{\mathsf{Q}_2}$, and by Definition 7 we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \cup \mathcal{E}(\mathsf{Q}_2)$. Therefore,

    $$|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{E}(\mathsf{Q})|_t \tag{38}$$

    If $|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (36) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (38) $|\mathcal{E}(\mathsf{Q})|_t \geq |\mathcal{SQL}(\mathsf{Q})|_t$. By induction hypothesis we conclude that V is an incorrect view.
  - A ≡ wrong(t) and Q ≡ Q_1 union all Q_2. Analogous to the previous point. Observe that replacing the set operator $\cup$ by $\cup_{\mathcal{M}}$ does not affect to the result.
- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 5. Considering Q' as Q_2. This case is analogous to the previous case changing Q_1 by Q_2.
- slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 8. Considering Q' as Q_1 . In this case we have:

  $$|\mathcal{SQL}(\mathsf{Q}_1)|_t = |\mathcal{SQL}(\mathsf{Q})|_t \tag{39}$$

  and A ≡ missing(t) and Q ≡ Q_1 except [all] Q_2. Then, from $\Phi_{\mathsf{Q}} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$ (changing $\setminus$ by $\setminus_{\mathcal{M}}$ in the case of all), and Definition 7 we have $\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \setminus \mathcal{E}(\mathsf{Q}_2)$. Therefore

  $$|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{E}(\mathsf{Q}_1)|_t \tag{40}$$

  If $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q}_1)|_t$, by (39) we obtain that $|\mathcal{E}(\mathsf{Q}_1)|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ and by (40), $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$. Then, by induction hypothesis, V is incorrect.

– slice(V,Q,A) calls to slice(V,Q',A') in Code 4, Line 9. Considering Q' as Q_2 . In this case $\mathsf{A} \equiv \mathsf{missing(t)}$, $\mathsf{A}' \equiv \mathsf{wrong(t)}$, $\mathsf{Q} \equiv \mathsf{Q}_1$ EXCEPT Q_2, $\mathsf{t} \in_\perp \mathcal{SQL}(\mathsf{Q}_2)$.

From $\Phi_\mathsf{Q} = \Phi_{\mathsf{Q}_1} \setminus \Phi_{\mathsf{Q}_2}$, and by Definition 7 we have

$$\mathcal{E}(\mathsf{Q}) = \mathcal{E}(\mathsf{Q}_1) \setminus \mathcal{E}(\mathsf{Q}_2) \tag{41}$$

and by Definition 1,

$$\mathcal{SQL}(Q) = \| \Phi_Q \| = \| \Phi_{Q_1} \| \setminus \| \Phi_{Q_2} \| = \mathcal{SQL}(Q_1) \setminus \mathcal{SQL}(Q_2) \tag{42}$$

From $\mathsf{t} \in_\perp \mathcal{SQL}(Q_2)$, we have that $|\mathcal{SQL}(Q)|_t = 0$, which means that the induction hypothesis $|\mathcal{E}(\mathsf{Q})|_t \leq |\mathcal{SQL}(\mathsf{Q})|_t$ can be rewritten as:

$$\text{If } \mathsf{t} \notin \mathcal{E}(\mathsf{Q}), \text{ then } \mathsf{V} \text{ is incorrect} \tag{43}$$

Now observe that in this case the call to slice is slice(V,Q_2,wrong(t)). Therefore we must prove that if $|\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t$, then V is an incorrect view. $|\mathcal{E}(\mathsf{Q}_2)|_t \geq |\mathcal{SQL}(\mathsf{Q}_2)|_t$ implies in particular that $\mathsf{t} \in_\perp \mathcal{E}(\mathsf{Q}_2)$, which by (41) means that $\mathsf{t} \notin \mathcal{E}(\mathsf{Q})$. Then, (43) holds. □

The next lemma indicates how state relates the answers obtained by the SQL system and the intended interpretation $\mathcal{I}$:

Lemma 3. *Let* R *be a relation,* $\mathcal{I}$(R) *its intended answer w.r.t. the current instance, and let* $\mathcal{P}$ *be the logic program contained in the variable* P *of Code 1. Then, the following implications hold at any moment of the execution of the algorithm:*

(P.1) $\mathcal{P} \vdash$ state((all R), valid) $\Rightarrow \mathcal{SQL}(\mathrm{R}) = \mathcal{I}(\mathrm{R})$
(P.2) $\mathcal{P} \vdash$ state((all R), nonvalid) $\Rightarrow \mathcal{SQL}(\mathrm{R}) \neq \mathcal{I}(\mathrm{R})$
(P.3) $\mathcal{P} \vdash$ state((t $\in$ R), valid) $\Rightarrow$ t $\in_\perp \mathcal{I}(\mathrm{R})$
(P.4) $\mathcal{P} \vdash$ state((t $\in$ R), nonvalid) $\Rightarrow$ t $\notin_\perp \mathcal{I}(\mathrm{R})$
(P.5) $\mathcal{P} \vdash$ state(($\mathrm{R}_1 \subseteq$ R), valid) $\Rightarrow \mathcal{SQL}(\mathrm{R}_1) \subseteq \mathcal{I}(\mathrm{R})$
(P.6) $\mathcal{P} \vdash$ state(($\mathrm{R}_1 \subseteq$ R), nonvalid) $\Rightarrow \mathcal{SQL}(\mathrm{R}_1) \not\subseteq \mathcal{I}(\mathrm{R})$

Proof. Proving $\mathcal{P} \vdash$ *state(E,S)* implies that there is a fact *state(E,S)* $\in \mathcal{P}$, because *state* is defined only by facts introduced by *processAnswer(E,A)* (Code 3, lines 1-5). We distinguish cases depending on the form of the input parameters E and A received by *processAnswer*.

– $E \equiv (R_1 \subseteq R)$. Then the function *processAnswer* has been called after asking the user about the validity of the enquire E, obtaining answer A. This happens in Code 1, line 10, and corresponds to the question *"Is S included in the intended answer for R?"*, with $S = \mathcal{SQL}(R_1)$ (see Definition 11). If the function *processAnswer* introduces the fact *state(*$(R_1 \subseteq R)$*,valid)*, this implies that the answer of the user was *yes*, meaning that $\mathcal{SQL}(R_1) \subseteq \mathcal{I}(R)$ that proves the implication *(P.5)*. The function *processAnswer* introduces the fact *state(*$(R_1 \subseteq R)$*,nonvalid)* when the answer of the user is either *no* or *wrong(s)*, meaning that $\mathcal{SQL}(R_1) \not\subseteq \mathcal{I}(R)$ which proves the implication *(P.6)*.
– $E \equiv (t \in R)$. Analogously, the function *processAnswer* has been called from Code 1, line 7 after asking the user about the validity of the enquire E, obtaining answer A. In this case the answer provided by the user to the question *"Does the intended answer for R include a tuple matching the tuple t?"* can be *yes* or *no*. If the function *processAnswer* introduces the fact *state(*$(t \in R)$*,valid)*, this implies that the

answer of the user was *yes*, meaning that $t \in_\perp \mathcal{I}(R)$ proving the implication *(P.3)*. The function *processAnswer* introduces the fact *state((t ∈ R),nonvalid)* when the answer of the user is *no*, meaning that $t \notin_\perp \mathcal{I}(R)$. Thus the implication *(P.4)* holds.

– $E \equiv$ *(all R)*. This input parameter corresponds to calls obtained in two different situations:

1. As in the previous cases, when the debugger obtains the user answer to the question *"Is S the intended answer for R?"*, with $S = \mathcal{SQL}(R)$. This corresponds to a call to processAnswer either from Code 1 line 10 or from Code 2, line 3.
   - If the call is from Code 1 line 10 then the fact *state((all R),valid)* is introduced as a consequence of an answer *yes*, meaning that $\mathcal{SQL}(R) = \mathcal{I}(R)$. Thus the implication *(P.1)* holds.
   - If the call is from Code 2 line 3 then the fact *state((all R),nonvalid)* is introduced by *processAnswer* as a consequence of an answer of the form *no*, *missing(t)* or *wrong(t)*. All these cases mean that $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ (see Definition 4), and the implication *(P.2)* holds.
2. In a recursive call produced by *processAnswer*. It is easy to check that only one recursive call can occur, due to the change in the first parameter to *(all R)* (which avoids further recursive calls). That is, a first call occurs containing the answer provided by the user, and the execution of this call starts a recursive call, which does not call *processAnswer* recursively. The recursive calls are located in three points of Code 3 and all of them correspond to the implication *(P.2)*:
   - Line 7. The initial call must be *processAnswer((s ∈ R),yes)*, and this call has introduced a fact *state((s ∈ R),valid)*, which means:

     (44) $$s \in_\perp \mathcal{I}(R)$$

     Enquiries of the form *(s ∈ R)* are associated to clauses of the form:
     buggy(V) ← state((s ∈ R), nonvalid)
     which are added in P by the function *missingBasic* (Code 5) only when the debugger checks that:

     (45) $$s \notin_\perp \mathcal{SQL}(R)$$

     The recursive call is *processAnswer((all R),missing(s))*. This call introduces the fact *state((all R),nonvalid))*. Then, from (44) and (45), we obtain the result $\mathcal{SQL}(R) \neq \mathcal{I}(R)$ and the implication *(P.2)* holds.
   - Line 9. The first call must be *processAnswer((V ⊆ R), A)* with $A \equiv$ *no* or$A \equiv$ *wrong(s)*. This is one of the cases already analyzed, where A is the answer provided by the user for the enquiry *(V ⊆ R)*. Now, observe that this enquiry must correspond to the election of an atom *state((V ⊆ R), . . .)* already ocurring in the program. Such atoms are introduced in line 8 of Code 6. In this function, the parameter V corresponds to a new view created by function *relevantTuples* (Code 7), and by Lemma 1:

     (46) $$\mathcal{SQL}(V) \subseteq \mathcal{SQL}(R)$$

     This first call *processAnswer((V ⊆ R), A)* has introduced a fact *state((V ⊆ R),nonvalid)*, and we have already proved that this implies:

     (47) $$\mathcal{SQL}(V) \not\subseteq \mathcal{I}(R)$$

which, combined with (46) means:

(48) $$\mathcal{SQL}(R) \not\subseteq \mathcal{I}(R)$$

□

4.2 Completeness

Next we study the completeness of the technique.

Theorem 5. *Completeness.*
Let R be a relation, and A the answer obtained after the call to askOracle(all R) *in line 1 of Code 1. If A is of the form* no, wrong(t) *or* missing(t)*, then the call* debug(R) *(defined in Code 1) returns a list L containing at least one relation.*

Proof. We prove the Completeness of our technique using some auxiliary results (see Figure 5).

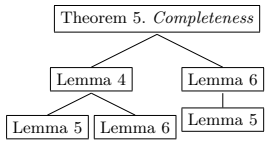


Fig. 5. The proof completeness structure

Lemma 4 proves that the program $\mathcal{P}$ contained in the variable P of Code 1 is finite.
Lemma 5 proves that the call to the function *slice* returns a finite number of clauses.
Lemma 6 proves that the while loop in Code 1 terminates in a finite number of iterations.

Let $\mathcal{P}$ be the logic program contained in the variable P of Code 1. By the structure of Code 1, the while loop only stops when there is at least one relation S such that $\mathcal{P} \vdash$ *buggy(S)*, and when this happens at least the relation S will be in the list L. Therefore, we only need to prove that the algorithm terminates in the conditions of the premises.
First we prove the termination of all calls to functions occurring in Code 1:

1. The call to *askOracle* in Code 1, lines 1 and 8 ends returning a valid value, that is a value of the form *yes*, *no*, *wrong(t)* or *missing(t)*.
2. The call to *initialize* in Code 2, line 3 ends because it traverses the computation tree top-down, and we are assuming finite computation trees (the database schema is finite and mutually recursive views are not allowed).
3. Functions *wrongBasic* and *missingBasic* always end. In both cases the body of the *for* loop is executed a finite number times because both the set *getFrom(Q)* and the set *getSelect(Q)* are finite for every query Q.

4. Function *slice* calls itself recursively traversing the structure of a query Q. Since the query definition must be finite, the recursion always ends reaching a *basic* query, and this case does not include recursive calls. This function calls to *wrongBasic* and *missingBasic* in lines 12 and 11 that by 3 always end. Then *slice* always ends.
5. Function *processAnswer* (Code 3) calls itself recursively at three points, but all these calls include a first parameter *(all R)* and observing the code we can check that these calls generate no further recursion. This function calls to *slice* in line 12, that by 4, always ends. Then *processAnswer* always ends.
6. The function *initialSetOfClauses* is called in Code 1 line 4. The input parameters are the view V to debug and a valid answer returned by the function *askOracle(all V)*. Then, the call to *initialSetOfClauses* ends by items 2 and 5.
7. The function *getBuggy* is called in Code 1 line 5 returning the list of all the relations R such that the goal *buggy(R)* can be proven w.r.t. the logic program P. This function always ends because the goal *buggy(R)* in *getBuggy(P)* is terminating. Lema 4 proves that the program P is finite. Thus, there is a maximum number of clauses *buggy* in P. And a finite, non-recursive (mutually recursive views are not allowed) and ground logic program is always terminating for any goal. This means that the goal *buggy(R)* in *getBuggy(P)* is terminating.
8. The function *getUnsolvedEnquiries* is called in Code 1 line 6. This function collects in a list LE all the unsolved enquiries e occurring in body atoms of the form *state(e,a)* of *buggy* clauses in P such that the logic program P does not contain neither a fact of the form *state(e,valid)* nor a fact of the form *state(e,nonvalid)*. By Lemma 4 there is a finite number of clauses *buggy* in P. For every atom of the form *state(e,a)* throw the goal *state(e,_)*, where the anonymous variable indicates that we do not care about the second parameter value. The goal *state(e,_)* is always terminating because predicate *state* is defined only by facts.
   Notice the returned list LE is never empty. In that case, the previous call to the function *getBuggy* would have returned a not empty list and the while loop would have stoped. As in the previous case, this function is always terminating.
9. The function *chooseEnquire* is called in Code 1 line 7 returning one of the enquiries in the list returned by the function *getUnsolvedEnquiries* according to some predefined terminating criterium.
10. Function *slice* calls itself recursively traversing the structure of a query Q. Since the query definition must be finite, the recursion always ends reaching a *basic* query, and this case does not include recursive calls.

In order to complete the proof we must check that the while loop in Code1, line 5 always terminates. By item 7, function *getBuggy(P)* ends and the goal *buggy(R)* in *getBuggy(P)* is terminating. By Lemma 6 the while loop terminates in a finite number of iterations.

□

Lemma 4. *Given a call* debug(R)*, there is a constant k such that the program* P *in Code 1 always have less than k clauses.*

Proof. Originally the number of clauses with head *buggy* is the number of nodes in the computation tree of the view to debug. During the execution of the algorithm new facts for predicate *state* are added, and also new clauses with head *buggy* are included. Let n be the number of nodes in the computation tree rooted by R. Notice that n is finite number, because views cannot be mutually recursive and we are assuming a finite database schema. Facts and clauses are added in two points in Code *debug*:

a) In Line 4. Function *initialSetOfClauses(R, A)* adds $k_1 \leq 2 \times n$ buggy clauses and one fact for the predicate *state*.

- The function *initialize(R)* (Code 2) traverses recursively the computation tree for R adding exactly n buggy clauses, one buggy clause for each node in the computation tree $CT(R)$.
- The function *processAnswer((all R), A)* (called in Code 2, line 3), adds one fact for the predicate *state* and calls to function *slice(R, Q, A)* with Q the query defining R. Let m be the number of relations occurring in the *from* clauses of the basic components occurring in the query Q. Then, by Lemma 5, the call to *slice(R,Q,A)* returns k_2 clauses, with $k_2 \leq m$. By Definition 9, $m = n - 1$.
  Then $k_1 = n + k_2 \leq 2 \times n$.
  Consider the program P after the call to *initialSetOfClauses(R, A)* (Code 1, line 4). Let L be the list of all the unsolved enquiries in P and let k_3 the number of elements in L. Notice that the cardinality of each node in the computation tree $CT(R)$ is at most m, where the cardinality of a node N is defined as the number of children of N. Then, in this point, the number of elements in L is less or equal than $k_1 \times m$ (L contains one unsolved enquiry for each atom of the form *state(q,s)* occurring in body clauses in the current program P). Therefore, $k_3 \leq k_1 \times m$.

b) In Line 10. Function *processAnswer(E, A)* adds k_4 clauses and at most two facts for the predicate *state* in each iteration of the while loop.
- Function *processAnswer(E, A)* adds at most two facts for the predicate *state* because only one recursive call can be occurs.
- Function *processAnswer((all V), A)* calls to function *slice(V, Q, A)* with Q the query defining V. Let $CT(V)$ the computation tree of V with n' nodes. $CT(V)$ is a subtree of $CT(R)$ and therefore $n' \leq n$. Let p be the number of relations occurring in the *from* clauses of the basic components occurring in the query Q. Then, by Lemma 5, the call to *slice(V,Q,A)* returns k_4 clauses, with $k_4 \leq p$. By definition 9, $p = n' - 1$. Then $k_4 \leq n' \leq n$.
  By Lemma 6 the while loop stops before k_3 iterations. After k_3 iterations the list L is empty and the function *getBuggy* returns a not empty list. Then, after $k3$ iterations, the number of facts added to the program is less than $k_3 \times 2$ and the number of buggy clauses added to the program is less than $k_3 \times k_4$.

In summary, the number of facts added to the program is less than $p_1 = 1 + (k_3 \times 2)$ and the number of buggy clauses added to the program is less than $p_2 = k_1 + (k_3 \times k_4)$.
The result holds by considering $k = p_1 + p_2$. □

Lemma 5. *Let* slice(V,Q,A) *be any call to Code 4 that occurs during the execution of the debugger. Let* $Q_1, \ldots, Q_q$ *be the basic queries occurring in the query* Q*, and let* n_i *be the number of relations occurring in the* from *clause of the basic query* Q_i*,* $i = 1, \ldots, q$*. Then:*

1. *The call to* slice(V,Q,A) *returns* k *clauses, with* $k \leq n_1 + \cdots + n_q$*.*
2. *All the clauses returned by the call to* slice(V,Q,A) *are any of the following forms:*
   - *buggy(V)* ← *state((s ∈ R), nonvalid) with* R *a relation occurring in the* from *clause of any basic query* Q_i*,* $i = 1, \ldots, q$*.*

– *buggy(V)* ← *state((*$V_1 \subseteq R_1$*), valid), . . . , state((*$V_n \subseteq R_n$*), valid) with* R_j *relations occurring in the* from *clause of any basic query* Q_i, $i = 1, \ldots, q$.

Proof. We prove the results by structural induction on the form of the query Q.

Basis: If Q is a basic query. We distinguish two cases:

- If A = missing(t), the set of clauses returned by the call *slice(V,Q,A)* is the set of clauses returned by the call to *missingBasic(V, Q, t)* (line 11 of code 4). Let R_1 AS $S_1, \ldots, R_n$ AS S_n be the list of elements returned by the function *getFrom(Q)*. Then the call *missingBasic(V, Q, t)* returns k buggy clauses of the form:
$$\mathsf{buggy(V)} \leftarrow \mathsf{state((s} \in \mathsf{R}_i\mathsf{)}, \mathsf{nonvalid)}$$
with $0 \leq k \leq n$. Then, the results 1 and 2 hold with $k \leq n$.
- If A = wrong(t), the set of clauses returned by the call *slice(V,Q,A)* is the set of clauses returned by the call to *wrongBasic(V, Q, t)* (line 12 of code 4). Let R_1 AS $S_1, \ldots, R_n$ AS S_n be the list of elements returned by the function *getFrom(Q)*. The call *wrongBasic(V, Q, t)* returns only one buggy clause (Code 6, line 8) of the form:
$$\mathsf{(buggy(V)} \leftarrow \mathsf{state((V}_1 \subseteq \mathsf{R}_1\mathsf{)}, \mathsf{valid)}, \ldots, \mathsf{state((V}_n \subseteq \mathsf{R}_n\mathsf{)}, \mathsf{valid))}$$
Then, the results 1 and 2 hold with $k = 1$.

Notice that in both cases, *slice(V,Q,A)* adds new clauses that do not imply enquiries of the form *(all R)*.

Inductive step: If Q is a compound query. Let Q_1, Q_2 its query components. In this case $q = q_1 + q_2$ where q_i is the total number of basic queries occurring in the query Q_i, $i = 1, 2$. By induction hypothesis, the call to the function *slice(V,*Q_i*,A)* returns a finite set of clauses $k_i \leq n_i$, where n_i is the total number of relations occurring in the *from* clauses of the basic queries $Q_{i1}, \ldots, Q_{iq_i}$ occurring in the query Q_i, $i = 1, 2$.
Additionally, all the clauses returned by the call to *slice(V,*Q_i*,A)* are any of the following forms:

- $\mathsf{buggy(V)} \leftarrow \mathsf{state((s} \in \mathsf{R}\mathsf{)}, \mathsf{nonvalid)}$ with R a relation occurring in the *from* clause of any basic query $Q_{i1}, \ldots, Q_{iq_i}$ occurring in the query Q_i, $i = 1, 2$.
- $\mathsf{buggy(V)} \leftarrow \mathsf{state((V}_1 \subseteq \mathsf{R}_1\mathsf{)}, \mathsf{valid)}, \ldots, \mathsf{state((V}_n \subseteq \mathsf{R}_n\mathsf{)}, \mathsf{valid)}$ with R_j the relations occurring in the *from* clause of any basic query $Q_{i1}, \ldots, Q_{iq_i}$ occurring in the query Q_i, $i = 1, 2$.

Following the Code 4, lines 2 to 9, the number of clauses k returned by the call *slice(V,Q,A)* is less than the number of clauses returned by the call to *slice(V,*Q_1*,_)* and *slice(V,*Q_2*,_)*. Then, $k \leq k_1 + k_2 < n_1 + n_2$.

□

Lemma 6. *Given a call* debug(R)*, the* while *loop in Code 1, lines 5-11 terminates in a finite number of iterations.*

Proof. A common tool for proving the termination of programs is the well-founded set, a set ordered in such a way as to admit no infinite descending sequences [9]. We use a well-founded multiset order $\succeq$ for proving the termination of the while loop in Code 1. The idea is to define an ordering $\succeq$ on finite multisets of enquiries that is induced by the ordering $\succ$ on the enquiries. We consider the following well-founded partial-ordering $\succ$ on the enquiries:

- *(all V)* $\succ$ *(s ∈ R)* for all R in $CT(V)$
- *(all V)* $\succ$ *(V' ⊆ R)* for all R in $CT(V)$
- *(V ⊆ R)* $\succ$ *(s ∈ R')* for all R' in $CT(R)$
- *(V ⊆ R)* $\succ$ *(V' ⊆ R')* for all R' in $CT(R)$ and $R \neq R'$
- *(s ∈ V)* $\succ$ *(t ∈ R)* for all R in $CT(V)$ and $R \neq V$
- *(s ∈ V)* $\succ$ *(V' ⊆ R)* for all R in $CT(V)$ and $R \neq V$

It is easy to check that there is no infinite descending chain using the ordering $\succ$ on the enquiries. The ordering $\succ$ on the enquiries induces an ordering $\succeq$ on multisets of enquiries which is defined as follows:

Let L_1 and L_2 be two multisets of enquiries. $L_1 \preceq L_2$ if for some multisets of enquiries X and Y, where X es not empty and $X \subseteq L_1$, $L_2 = (L_1 \setminus_{\mathcal{M}} X) \cup_{\mathcal{M}} Y$ and for all $y \in Y$, there exist $x \in X$ such that $x \succ y$.

In this ordering, $L_1 \preceq L_2$ if L_2 can be obtained from L_1 by replacing one or more enquiries in L_1 by any finite number of enquiries, each of which is smaller than one of the replaced enquiry. In particular, a multiset of enquiries is reduced by replacing an enquiry with zero enquiries, i.e. by deleting it.

Consider de program P after the call to *initialSetOfClauses* (Code 1, line 4). Let LE be the list of all the unsolved enquiries in P returned by the call to the function *getUnsolvedEnquiries* (Code 1, line 6). Next we prove that after each loop iteration either the while condition becomes true (a buggy node is found) and the while loop terminates or the multiset LE of the unsolved enquiries in P is reduced. This means that eventually LE will be empty, but this means that the initial set of enquiries LE are solved and a buggy node has been found.

At each iteration of the loop (Code 1, lines (5 - 11)), new clauses and facts returned by the function *processAnswer* (line 10) are added in P.

The new clauses returned by the function *processAnswer* are the clauses returned by the function *slice* called in Code 3, line 12. By Lemma 5, function *slice* returns a finite number of new clauses, and the returned clauses do not imply enquiries of the form *(all V)*.

One unsolved enquiry E in LE is selected in line 7, and the answer A provided by the user is processed calling to the function *processAnswer(E, A)* (line 10 in Code 1). Each call to the function *processAnswer* solves one or two unsolved enquiries in LE (Code 3, lines 1-5). We distinguish cases depending of the form of the enquiry E and the answer A:

- $E \equiv$ *(t ∈ V)*.
  - $A \equiv$ *no*. In this case, function *processAnswer* returns the fact *state((t ∈ V),nonvalid)* and the enquiry E is solved. Notice no more iterations are needed because the enquiry $(t \in V)$ is associated to a buggy clause in P of the form:

    $$\mathsf{buggy(R)} \leftarrow \mathsf{state((t \in V), nonvalid)}$$

    and therefore, the goal *buggy(R)* can be proved w.r.t. the program P and the while condition becomes true.
  - $A \equiv$ *yes*. In this case, function *processAnswer* returns two facts. The first one is *state((t ∈ V),valid)* and the second one is *state((all V), nonvalid)*. The last one is returned by the recursive call to the function *processAnswer* with the enquiry *(all V)* and missing(t) as parameters (line 7). Therefore at least the enquiry E is solved.
    The recursive call returns a set of clauses returned by the call to the function *slice(V, Q, missing(t))* with Q the query defining V. By Lemma 5, item 2, the

returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by replacing the enquiry *(t ∈ V)* (and possibly the enquiry *(all V)*) by the set of enquiries $E_1, \ldots E_n$ and therefore $LE \succeq LE'$.

- $E \equiv$ *(V ⊆ R)*.
  - $A \equiv no$ or $A \equiv wrong(t)$. Function *processAnswer* returns a fact of the form *state((V ⊆ R),nonvalid)* and the enquiry E is solved. The function *processAnswer* is called recursively with the enquiry *(all R)* and A as parameters (line 10). The recursive call returns a set of clauses returned by the call to the function *slice(R, Q, A)* with Q the query defining R. By Lemma 5, item 2, the returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the the goal *buggy(R)* can not be proved w.r.t. the program P, the loop body is executed again, in which case the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by replacing the enquiry *(V ⊆ R)* (and possibly the enquiry *(all R)*) by the set of enquiries $E_1, \ldots E_n$. Therefore, $LE \succeq LE'$.
  - $A \equiv yes$. Function *processAnswer* returns a fact of the form *state((V ⊆ R),valid)* and the enquiry E is solved. If the while loop does not terminates, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(V ⊆ R)* and therefore $LE \succeq LE'$.
- $E \equiv$ *(all V)*.
  - $A \equiv yes$. Function *processAnswer* returns a fact of the form *state((all V), valid)* and the enquiry E is solved. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$.
  - $A \equiv no$. In this case, function *processAnswer* returns a fact of the form *state((all V),nonvalid)* and the enquiry E is solved. As in the previous case, if the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$.
  - $A \equiv missing(t)$ or $A \equiv wrong(t)$. Function *processAnswer* returns a fact of the form *state((all V),nonvalid)* (and the enquiry E is solved) and a set of clauses returned by the call to the function *slice(V, Q, A)* with Q the query defining V. By Lemma 5, item 2, the returned clauses imply a finite set of enquiries $E_1, \ldots E_n$ such that $E \succ E_i$, $0 \leq i \leq n$. If the while condition becomes true, the while loop terminates. Otherwise, the list of unsolved enquiries LE' returned by the call to the function *getUnsolvedEnquiries* in the next loop iteration is obtained from LE by deleting the enquiry *(all V)* and therefore $LE \succeq LE'$. □

Thus, the algorithm always stops pointing to some user view (completeness) which is incorrectly defined (correctness).

5 Implementation

The algorithm presented in Section 3 has been implemented in the Datalog Educational System (DES [16,17]), which makes it possible for Datalog and SQL to coexist as query

languages for the same database. The debugger is started when the user detects that *Anna* is not among the (large) list of student names produced by view *awards*. The command `/debug_sql` starts the session:

```
1: DES-SQL> /debug_sql awards
2: Info: Debugging view 'awards': {   1 - awards('Carla'), ... }
3: Is this the expected answer for view 'awards'?  m'Anna'
4: Does the intended answer for 'intensive' include  ('Anna')        ? n
5: Does the intended answer for 'standard'  include  ('Anna',1,true) ? y
6: Does the intended answer for 'standard'  include  ('Anna',2,true) ? y
7: Does the intended answer for 'standard'  include  ('Anna',3,false)? y
8: Info: Buggy relation found: intensive
```

The user answer *m'Anna'* in line 3 indicates that *('Anna')* is missing in the view *awards*. In line 4 the user indicates that view *intensive* should not include *('Anna')*. In lines 5, 6, and 7, the debugger asks three simple questions involving the view *standard*. After checking the information for *Anna*, the user indicates that the listed tuples are correct. Then, the tool points out *intensive* as the buggy view, after only five simple questions. Observe that intermediate views can contain hundreds of thousands of tuples, but the slicing mechanism helps to focus only on the source of the error. Next, we describe briefly how these questions have been produced by the debugger.

After the user indicates that *('Anna')* is missing, the debugger executes a call *processAnswer(all(awards),missing(('Anna')))*. This implies a call to *slice(awards,* Q_1 except Q_2*, missing(('Anna')))* (line 12 of Code 3). The debugger checks that Q_2 produces *('Anna')* (line 9 of Code 4), and proceeds with the recursive call *slice(awards,* Q_2*, wrong(('Anna')))* with $Q_2 \equiv$ select student from intensive. Query Q_2 is basic, and then the debugger calls *wrongBasic(awards,* Q_2*, ('Anna'))* (line 12 of Code 4)). Function *wrongBasic* creates a view that selects only those tuples from *intensive* producing the wrong tuple *('Anna')* (function *relevantTuples* in Code 7):

```
create view intensive_slice(student) as
(select * from intensive)
intersect all
(select * from intensive I where  I.student = 'Anna');
```

Finally, the following clause is added to the program P (line 8, Code 6) where subset(intensive_slice,intensive) represents the enquiry $E \equiv$*(intensive_slice* $\subseteq$ *intensive*) :

```
buggy(awards) :- state(subset(intensive_slice,intensive),valid).
```

By enabling development listings with the command `/development on`, the logic program is also listed during debugging. The debugger chooses the only body atom in this clause as next unsolved enquiry, because it only contains one tuple. The call to *askOracle* returns *wrong(('Anna'))* (the user answers *'no'* in line 4). Then *processAnswer(subset(intensive_slice,intensive), wrong(('Anna')))* is called, which in turn calls to *processAnswer(all(intensive),wrong(('Anna')))* recursively. Next call is *slice(intensive, Q, wrong(('Anna')))*, with $Q \equiv Q_3$ union Q_4 the query definition of *intensive* (see Figure 1). The debugger checks that only Q_4 produces *('Anna')* and calls to *slice(intensive,* Q_4*, wrong(('Anna')))*. Query Q_4 is basic, which implies a call to *wrongBasic(intensive,* Q_4*, ('Anna'))*. Then *relevantTuples* is called three times, one for each occurrence of the view *standard* in the from section of Q_4, creating new views:

```
create view standard_slice_1(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A1.student, A1.level, A1.pass
    from standard as A1,  standard as A2, standard as A3
    where (A1.student = A2.student and A2.student = A3.student
       and A1.level = 1 and A2.level = 2 and A3.level = 3)
       and A1.student = 'Anna');

create view standard_slice_2(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A2.student, A2.level, A2.pass
    from standard as A1,  standard as A2, standard as A3
    where (A1.student = A2.student and A2.student = A3.student
       and A1.level = 1 and A2.level = 2 and A3.level = 3)
       and A1.student = 'Anna');

create view standard_slice_3(student,level,pass) as
   ( select R.student, R.level, R.pass from standard as R)
       intersect all
   (select  A3.student, A3.level, A3.pass
    from standard as A1,  standard as A2, standard as A3
    where (A1.student = A2.student and A2.student = A3.student
       and A1.level = 1 and A2.level = 2 and A3.level = 3)
       and A1.student = 'Anna');
```

Finally, the clause:

```
buggy(intensive) :- state(subset(standard_slice_1,standard),valid),
                    state(subset(standard_slice_2,standard),valid),
                    state(subset(standard_slice_3,standard),valid).
```

is added to P (line 8, Code 6). Next, the tool selects the unsolved question with less complexity that correspond to the questions of lines 5, 6, and 7, for which the user answer *yes*. Therefore, the clause for buggy(intensive) succeeds and the algorithm finishes pointing out *intensive* as a source of the error.

6 Conclusions

We have presented a new technique for debugging systems of SQL views. Our proposal present a declarative debugging technique and then it refines the technique by taking into account information about *wrong* and *missing* answers provided by the user. Using a technique similar to dynamic slicing [1], we concentrate only in those tuples produced by the intermediate relations that are relevant for the error. This minimizes the main problem of declarative debugging when applied directly to SQL views, namely the huge number of tuples that the user must consider in order to determine the validity of the result produced by a relation. Previous works deal with the problem of tracking provenance information for query results [11,8], but to the best of our knowledge, none of them treat the case of missing tuples, which is important in our setting. This report

extends two previous papers [4,5] which present an abbreviated version and without proofs of all the results of our setting.

The proposed algorithm looks for particular but common error sources, like tuples missed in the from section or in and conditions (that is, intersect components in our representation). If such shortcuts are not available, or if the user only answers *yes* and *no*, then the tools works as a pure declarative debugger.

A more general contribution of the report is the idea of representing a declarative debugging computation tree by means of a set of logic clauses. In fact, the algorithm in Code 1 can be considered a general debugging schema, because it is independent of the underlying programming paradigm. The main advantage of this representation is that it allows combining declarative debugging with other diagnosis techniques that can be also represented as logic programs. In our case, declarative debugging and slicing cooperate for locating an erroneous relation. It would be interesting to research the combination with other techniques such as the use of assertions.

References

1. H. Agrawal and J. R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25:246–256, June 1990.
2. ApexSQL Debug , 2011. `http://www.apexsql.com/sql_tools_debug.aspx/`.
3. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A theoretical framework for the declarative debugging of datalog programs. In *SDKB 2008*, volume 4925 of *LNCS*, pages 143–159. Springer, 2008.
4. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Proceedings of the 8th International Andrei Ershov Memorial Conference, PSI 2011*, volume 7162 of *Lecture Notes in Computer Science*, pages 77–85. Springer LNCS, 2012.
5. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Declarative Debugging of Wrong and Missing Answers for SQL Views. In *Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012)*, volume 7294 of *LNCS*, pages 73–87. Springer-Verlag, 2012.
6. R. Caballero, F. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Proc. FLOPS'01*, number 2024 in LNCS, pages 170–184. Springer, 2001.
7. S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Softw. Eng.*, 11:324–345, April 1985.
8. Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25:179–227, June 2000.
9. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, Aug. 1979.
10. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
11. B. Glavic and G. Alonso. Provenance for nested subqueries. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 982–993, New York, NY, USA, 2009. ACM.
12. P. W. Grefen and R. A. de By. A multi-set extended relational algebra: a formal approach to a practical issue. In *ICDE'94*, pages 80–88. IEEE, 1994.

13. L. Naish. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 3, 1997.
14. H. Nilsson. How to look busy while being lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
15. Rapid SQL Developer Debugger, 2011. `http://docs.embarcadero.com/products/rapid_sql/`.
16. F. Sáenz-Pérez. Datalog Educational System v2.6, October 2011. `http://des.sourceforge.net/`.
17. F. Sáenz-Pérez. DES: A Deductive Database System. *Elec. Notes on Theor. Comp. Science*, 271, 2011.
18. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
19. J. Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.
20. SQL, ISO/IEC 9075:1992, third edition, 1992.

Embedding XQuery in Toy *

Jesús Almendros-Jiménez [§], Rafael Caballero[†], Yolanda García-Ruiz[†] and Fernando Sáenz-Pérez[‡]

Technical Report SIC-04-11

[§]Dpto. Lenguajes y Computación, Universidad de Almería, Spain
[†]Dpto. de Sistemas Informáticos y Computación, UCM, Spain
[‡]Dpto. de Ingeniería del Software e Inteligencia Artificial, UCM, Spain
jalmen@ual.es, {rafa,fernan}@sip.ucm.es , ygarciar@fdi.ucm.es

April 2011

* Work partially supported by the Spanish projects STAMP TIN2008-06622-C03-01, Prometidos-CM S2009TIC-1465 and GPD UCM-BSCH-GR58/08-910502.

Abstract. This report addresses the problem of integrating a fragment of XQuery, a language for querying XML documents, into the functional-logic language $\mathcal{TOY}$. The queries are evaluated by an interpreter, and the declarative nature of the proposal allows us to prove correctness and completeness with respect to the semantics of the subset of XQuery considered. The different fragments of XML that can be produced by XQuery expressions are obtained using the non-deterministic features of functional-logic languages. As an application of this proposal we show how the typical *generate and test* techniques of logic languages can be used for generating test-cases for XQuery expressions.

1 Introduction

XQuery has been defined as a query language for finding and extracting information from XML [19] documents. Originally designed to meet the challenges of large-scale electronic publishing, XML also plays an important role in the exchange of a wide variety of data on the Web and elsewhere. For this reason many modern languages include libraries or encodings of XQuery, including logic programming [1] and functional programming [9]. In this report we consider the introduction of a simple subset of XQuery [5,21] into the functional-logic language $\mathcal{TOY}$ [14].

One of the key aspects of declarative languages is the emphasis they pose on the logic semantics underpinning declarative computations. This is important for reasoning about computations, proving properties of the programs or applying declarative techniques such as abstract interpretation [7,8], partial evaluation [13] or algorithmic debugging [18]. There are two different declarative alternatives that can be chosen for incorporating XML into a (declarative) language:

1. Use a domain-specific language and take advantage of the specific features of the host language. This is the approach taken in [11], which presents a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry, and also in [17] for the case of logic programming.
2. Consider an existing query language such as XQuery, and embed a fragment of the language in the host language, in this case $\mathcal{TOY}$. This is the approach considered in this report.

Thus, our goal is to include XQuery using the purely declarative features of the functional-logic language $\mathcal{TOY}$. Moreover, analyzing the functional-logic semantics [15] of the embedding we are able to prove that the semantics of the considered fragment of XQuery has been correctly included in $\mathcal{TOY}$. To the best of our knowledge, it is the first time a fragment of XQuery has been encoded in a functional-logic language. A first step in this direction was proposed in [4], where XPath [6] expressions were introduced in $\mathcal{TOY}$. XPath is a subset of XQuery that allows navigating and returning fragments of documents in a similar way as the path expressions used in the *chdir* command of many operating systems. The contributions of this report with respect to [4] are:

1. The setting has been extended to deal with a simple fragment of XQuery, including *for* statements for traversing XML sequences, *if/where* conditions, and the possibility of returning XML elements as results. Some basic XQuery constructions such as *let* statements are not considered, but we think that the proposal is powerful enough for representing many interesting queries.
2. The soundness of the approach is formally proved, checking that the semantics of the fragment of XQuery included in our setting is correctly represented in $\mathcal{TOY}$.

Next Chapter introduces the fragment of XQuery considered and a suitable operational semantics for evaluating queries. Then the language $\mathcal{TOY}$ and its

$$
\begin{aligned}
query &::= query\ query \mid tag \\
&\mid var \mid var/axis :: \nu \\
&\mid \textbf{for}\ var\ \textbf{in}\ query\ \textbf{return}\ query \\
&\mid \textbf{if}\ cond\ \textbf{then}\ query \\
cond &::= var = var \mid query \\
tag &::= \langle a\rangle\ \langle /a\rangle \mid \langle a\rangle var\ \ldots\ var\langle /a\rangle \mid \langle a\rangle tag\langle /a\rangle \\
axis &::= \textbf{self} \mid \textbf{child} \mid \textbf{descendant} \mid \textbf{dos}
\end{aligned}
$$

Fig. 1. Syntax of SXQ, a simplified version of XQ

semantics are presented in Chapter 3. Chapter 4 includes the interpreter that performs the evaluation of simple XQuery expressions in $\mathcal{TOY}$. The theoretical results establishing the soundness of the approach with respect to the operational semantics of Chapter 2 are presented in Section 4.1. Chapter 5 explains the automatic generation of Test Cases for simple XQuery expressions. Finally, Chapter 6 concludes summarizing the results and proposing future work.

2 XQuery and Its Operational Semantics

XQuery allows the user to query several documents, applying join conditions, generating new XML fragments, and many other features [5,21]. The syntax and semantics of the language are quite complex [20], and thus only a small subset of the language is usually considered. The next section introduces the fragment of XQuery considered in this report.

2.1 The subset SXQ

In [3] a declarative subset of XQuery, called XQ, is presented. This subset is a core language for XQuery expressions consisting of *for*, *let* and *where/if* statements. In this report we consider a simplified version of XQ, which we call SXQ and whose syntax can be found in Figure 1. In this grammar, a denotes a label and ν refers to a *label test* which is a label. The differences of SXQ with respect to XQ are:

1. XQ includes the possibility of using variables as tag names using a constructor *lab($x)*.
2. XQ permits enclosing any query Q between tag labels $\langle a\rangle Q\langle /a\rangle$. SXQ admits a start and end tag with nothing between them $\langle a\rangle\langle /a\rangle$ and either variables or other tags inside a tag.
3. XQ allows the empty query ( ) as a valid query. This allows representing expressions of the form $\langle a\rangle\langle /a\rangle$. Although SXQ does not allow empty queries, these expressions are built in SXQ by the *tag* constructor.

Our setting can be easily extended to support the *lab($x)* feature, but we omit this case for the sake of simplicity in this presentation. The second restriction

is more severe: although *let*s are not part of XQ, they could be simulated using *for* statements inside tags. In our case, forbidding other queries different from variables inside tag structures imply that our core language cannot represent *let* expressions. This limitation is due to the non-deterministic essence of our embedding, since a *let* expression means collecting all the results of a query instead of producing them separately using non-determinism. In spite of these limitations, the language SXQ is still useful for solving many common queries as the following example shows.

Example 1. Consider an XML file "bib.xml" containing data about books, and another file "reviews.xml" containing reviews for some of these books (see Appendix A). Then, we can list the reviews corresponding to books in "bib.xml" as follows:

```
for $b in doc("bib.xml")/bib/book,
    $r in doc("reviews.xml")/reviews/entry
where $b/title = $r/title
for $booktitle in $r/title,
    $revtext in $r/review
return <rev> $booktitle  $revtext </rev>
```

The variable $\$b$ takes the value of each different book, and $\$r$ represents the different reviews. The *where* condition ensures that only reviews corresponding to the book are considered. Finally, the last two variables are only employed to obtain the book title and the text of the review, the two values that are returned as output of the query by the *return* statement.

It can be argued that the code of this example does not follow the syntax of Figure 1. While this is true, it is very easy to define an algorithm that converts a query formed by *for*, *where* and *return* statements into a SXQ query (as long as it only includes variables inside tags, as stated above). The idea is simply to replace the references to XML documents by new indexed variables $\$x_1, \$x_2, \ldots$, and convert the *where* into *if*s, following each *for* by a *return*, and decomposing XPath expressions including several steps into several *for* expressions by introducing a new auxiliary variables, each one consisting of a single step.

Example 2. The query of Example 1 using SXQ syntax:

```
for $x3 in $x1/child::bib return
for $x4 in $x3/child::book  return
for $x5 in $x2/child::reviews return
for $x6 in $x5/child::entry  return
for $x7 in $x4/child::title return
for $x8 in $x6/child::title return
if ($x7 = $x8) then
   for $x9 in $x6/child::title return
     for $x10 in $x6/child::review return <rev> $x9  $x10 </rev>
```

Notice that the expressions doc("bib.xml") and doc("reviews.xml") have been substituted by the variables `$x1` and `$x2` respectively. Both variables are

free in the query and the value of each one is the XML document contained in the corresponding XML file.

The concept of set of *free* variables of a SXQ query is given by the following inductive definition:

Definition 1. *Let Q be a SXQ query. The set of free variables of Q, denoted by $free(Q)$, is defined as follows:*

- *If* $Q \equiv Q_1\ Q_2$, *then* $free(Q) := free(Q_1) \cup free(Q_2)$
- *If* $Q \equiv$ for $\$x$ in Q_1 return Q_2, *then* $free(Q) := (free(Q_1) \cup free(Q_2)) \setminus \{\$x\}$
- *If* $Q \equiv$ if $\$x_i = \$x_j$ then Q_2, *then* $free(Q) := \{\$x_i, \$x_j\} \cup free(Q_2)$
- *If* $Q \equiv$ if Q_1 then Q_2, *then* $free(Q) := free(Q_1) \cup free(Q_2)$
- *If* $Q \equiv \$x$, *then* $free(Q) := \{\$x\}$
- *If* $Q \equiv \$x/axis :: \nu$, *then* $free(Q) := \{\$x\}$
- *If* $Q \equiv \langle a \rangle \langle /a \rangle$, *then* $free(Q) := \emptyset$
- *If* $Q \equiv \langle a \rangle tag \langle /a \rangle$, *then* $free(Q) := free(tag)$
- *If* $Q \equiv \langle a \rangle \$x_i \ldots \$x_j \langle /a \rangle$, *then* $free(Q) := \{\$x_i, \ldots, \$x_j\}$

We assume that XML documents must be accessed initially via indexed variables $\$x_1, \$x_2, \ldots$ belonging to the set $free(Q)$.

Next we define a function ρ that takes a SXQ query and an index m (a positive integer) as inputs and returns a new SXQ query equivalent to Q.

Definition 2. *Let Q be a SXQ query and an index m. We define the query $\rho(Q, m)$ from Q recursively as follows:*

1. *If* $Q \equiv Q_1\ Q_2$, *then* $\rho(Q, m) := \rho(Q_1, m)\rho(Q_2, m)$
2. *If* $Q \equiv$ for $\$a$ in Q_1 return Q_2, *then*

$$\rho(Q, m) := \text{ for } \$x_{m+1} \text{ in } \rho(Q_1, m) \text{ return } \rho(Q_2', m+1)$$

   where Q_2' is the SXQ query obtained from Q_2 by replacing all occurrences of variable $\$a$ by the variable $\$x_{m+1}$.
3. *If* $Q \equiv$ if Q_1 then Q_2, *then*

$$\rho(Q, m) := \text{ if } \rho(Q_1, m) \text{ then } \rho(Q_2, m)$$

4. *If* $Q \equiv$ if $\$x_i = \$x_j$ then Q_1, *then*

$$\rho(Q, m) := \text{ if } \$x_i = \$x_j \text{ then } \rho(Q_1, m)$$

5. *In the rest of the cases,* $\rho(Q, m) := Q$.

Without loss of generality, in the rest of the report we assume that all the variables occurring in Q are renamed following the Definition 2 using index k with k the cardinality of $free(Q)$.

Example 3. Consider the XML file "bib.xml" from Example 1. Let Q be the following SXQ query:

```
for $b in for $c in $x1/child::bib return $c/child::book return
for $c in $b/child::title return
for $d in $b/child::price return  <item>  $c  $d </item>
```

Query Q selects the title and price of books in "bib.xml". The XML document "bib.xml" is accessed via the variable $\$x1$. By Definition 1, $free(Q) = \{\$x1\}$. The following query is obtained from Q by renaming all the variables occurring in Q according to Definition 2 using index $m = 1$:

```
for $x2 in for $x2 in $x1/child::bib return $x2/child::book return
for $x3 in $x2/child::title return
for $x4 in $x2/child::price return  <item>  $x3  $x4 </item>
```

Notice that the two occurrences of variable $\$x2$ correspond to different scopes.

We end this Section with a few definitions that are useful for the rest of the report. Following the syntax of Figure 1, SXQ queries can contain subqueries. In that case given a query Q, we use the notation $Q_{|p}$ for representing the subquery Q' that can be found in Q at position p. More formally, notions like positions and subqueries can be defined by induction on the structure of the query.

Definition 3. *Let Q be a SXQ query:*

1. *The set of positions of the query Q is a set $Pos(Q)$ of strings over the alphabet $\{1, 2\}$, which is inductively defined as follows:*
   - *If $Q \equiv \$x$, $Q \equiv \$x/axis :: \nu$, $Q \equiv \langle a\rangle\langle/a\rangle$ or $Q \equiv \langle a\rangle \$x_i \ldots \$x_j \langle/a\rangle$, then $Pos(Q) := \{\varepsilon\}$, where ε denotes de empty string.*
   - *If $Q \equiv \langle a\rangle tag \langle/a\rangle$, then $Pos(Q) := \{\varepsilon\} \cup \{1 \cdot p \mid p \in Pos(tag)\}$.*
   - *If $Q \equiv Q_1\ Q_2$, $Q \equiv$ for $\$x_j$ in Q_1 return Q_2 or $Q \equiv$ if Q_1 then Q_2, then $Pos(Q) := \{\varepsilon\} \cup \bigcup_{i=1}^{2} \{i \cdot p \mid p \in Pos(Q_i)\}$.*
   - *If $Q \equiv$ if $\$x_i = \$x_j$ then Q_1, then $Pos(Q) := \{\varepsilon\} \cup \{1 \cdot p \mid p \in Pos(Q_1)\}$.*

   The prefix order defined as

   $$p \leq q \text{ iff there exists } p' \text{ such that } p \cdot p' = q$$

   is a partial order on positions.
2. *For $p \in Pos(Q)$, the subquery of Q at position p, denoted by $Q_{|p}$, is defined by induction on the length of p:*

$$\begin{array}{ll} Q_{|\varepsilon} & := Q, \\ (Q_1\ Q_2)_{|i\cdot q} & = (Q_i)_{|q}, \\ (\text{for } \$x_k \text{ in } Q_1 \text{ return } Q_2)_{|i\cdot q} & := (Q_i)_{|q}, \\ (\text{if } Q_1 \text{ then } Q_2)_{|i\cdot q} & := (Q_i)_{|q}, \\ (\text{if } \$x_k = \$x_j \text{ then } Q_1)_{|1\cdot q} & := (Q_1)_{|q}, \\ (\ \langle a\rangle tag\langle/a\rangle\ )_{|1\cdot q} & := (tag)_{|q}. \end{array}$$

   Note that, for $p = i \cdot q$, $p \in Pos(Q)$ and queries of the form $\langle a\rangle tag\langle/a\rangle$ and if $\$x_k = \$x_j$ then Q_1, *implies that $i = 1$.*

Example 4. The position $p = 2{\cdot}2{\cdot}2{\cdot}2$ of the query in the Example 2 corresponds to the query:

```
for $x7 in $x4/child::title return
  for $x8 in $x6/child::title return
   if ($x7 = $x8) then
    for $x9 in $x6/child::title return
      for $x10 in $x6/child::review return <rev> $x9  $x10 </rev>
```

while the position $p = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 1$ of the same query corresponds to the query:

```
$x4/child::title
```

The set of positions of a query Q is closed under taking prefixes, i.e. if $p \in Pos(Q)$ then $q \in Pos(Q)$ for all $q \leq p$.

Next definition relates the subquery of Q at position $p \in Pos(Q)$ to the variables introduced by *for* statements in Q at positions $q \in Pos(Q)$ with $q < p$.

Definition 4. *Given a SXQ query Q and a position $p \in Pos(Q)$, the set of variables $V_{for}(Q, p)$ is defined as:*

1. $V_{for}(Q, \varepsilon) := \emptyset$
2. $V_{for}(Q_1\ Q_2\ ,\ i \cdot q) := V_{for}(Q_i, q)$
3. $V_{for}(\ \mathsf{for}\ \ \$x_k\ \mathsf{in}\ \ Q_1\ \mathsf{return}\ \ Q_2\ ,\ 1 \cdot q) := V_{for}(Q_1, q)$
4. $V_{for}(\ \mathsf{for}\ \ \$x_k\ \mathsf{in}\ \ Q_1\ \mathsf{return}\ \ Q_2\ ,\ 2 \cdot q) := \{\$x_k\} \cup V_{for}(Q_2, q)$
5. $V_{for}(\ \mathsf{if}\ \ Q_1\ \mathsf{then}\ \ Q_2\ ,\ i \cdot q) := V_{for}(Q_i, q)$
6. $V_{for}(\ \mathsf{if}\ \ \$x_k = \$x_j\ \mathsf{then}\ \ Q_1\ ,\ 1 \cdot q) := V_{for}(Q_1, q)$
7. $V_{for}(\langle a \rangle tag \langle / a \rangle\ ,\ 1 \cdot q) := \emptyset$

Next, we define the set of *relevant* variables for a query Q at position p, denoted by $Rel(Q, p)$, as the set of variables that can appear free in a query Q at position p. The next lemma introduces a basic property of $V_{for}(Q, p)$.

Lemma 1. *Let Q be a SXQ query, and p, q be strings over the alphabet $\{1, 2\}$. If $p \cdot q \in Pos(Q)$, then $V_{for}(Q, p \cdot q) = V_{for}(Q, p) \cup V_{for}(Q_{|p}, q)$.*

Proof. Notice that if $p \cdot q \in Pos(Q)$, then $p \in Pos(Q)$ and $q \in Pos(Q_{|p})$. The result can be proved by induction on the length of p according to the formal definitions given above.

- For $p = \varepsilon$, we have $Q_{|\varepsilon} = Q$. In addition $p = \varepsilon$ implies $p \cdot q = q$ and $V_{for}(Q, p \cdot q) = V_{for}(Q, q)$. By Definition 4, rule 1, $V_{for}(Q, \varepsilon) = \emptyset$. Then, $V_{for}(Q, p \cdot q) = V_{for}(Q, q) = \emptyset \cup V_{for}(Q, q) = V_{for}(Q, \varepsilon) \cup V_{for}(Q_{|\varepsilon}, q)$, which shows the result.
- Now, assume that $p = i \cdot p'$. Because $i \cdot p' \cdot q \in Pos(Q)$, the query Q is of the form:
  - $Q \equiv Q_1\ Q_2$. In this case, i can be either 1 or 2. Suppose $i = 1$. Then:
    (a) $Q_{|p} = Q_{|1 \cdot p'} = (Q_1)_{|p'}$ (by Definition 3)
    (b) $V_{for}(Q, 1 \cdot p') = V_{for}(Q_1, p')$ (by Definition 4, rule 2)

(c) $V_{for}(Q, p \cdot q) = V_{for}(Q, 1 \cdot p' \cdot q) = V_{for}(Q_1, p' \cdot q)$ (by Definition 4, rule 2)
Applying induction on (c) we obtain
(d) $V_{for}(Q_1, p' \cdot q) = V_{for}(Q_1, p') \cup V_{for}((Q_1)_{|p'}, q)$
and by (a) and (b),
(e) $V_{for}(Q_1, p' \cdot q) = V_{for}(Q, 1 \cdot p') \cup V_{for}(Q_{|1 \cdot p'}, q)$
Then, with (c) and (e), we obtain $V_{for}(Q, p{\cdot}q) = V_{for}(Q, 1{\cdot}p') \cup V_{for}(Q_{|1 \cdot p'}, q) = V_{for}(Q, p) \cup V_{for}(Q_{|p}, q)$ and the result holds.
If $i = 2$, the result can be proved similarly.

- $Q \equiv$ for $\$x_k$ in Q_1 return Q_2. If $i = 1$ the result can be proved similarly to the previous case. Now, suppose $i = 2$. Then:
  (a) $Q_{|p} = Q_{|2 \cdot p'} = (Q_2)_{|p'}$ (by Definition 3)
  (b) $V_{for}(Q, p) = V_{for}(Q, 2 \cdot p') = \{\$x_k\} \cup V_{for}(Q_2, p')$ (by Definition 4, rule 4)
  By Definition 4, rule 4, $V_{for}(Q, p \cdot q) = V_{for}(Q, 2 \cdot p' \cdot q) = \{\$x_k\} \cup V_{for}(Q_2, p' \cdot q)$, and by induction we obtain:
  (c) $V_{for}(Q, p \cdot q) = \{\$x_k\} \cup V_{for}(Q_2, p') \cup V_{for}((Q_2)_{|p'}, q)$
  Then, by (a) and (b), the result:
  $V_{for}(Q, p{\cdot}q) = V_{for}(Q, p) \cup V_{for}((Q_2)_{|p'}, q) = V_{for}(Q, p) \cup V_{for}(Q_{|p}, q)$ holds.

  The rest of the cases can be proved similarly.

Definition 5. *Given a SXQ query Q and a position $p \in Pos(Q)$, $Rel(Q, p)$ is defined as:*

$$Rel(Q, p) := free(Q) \cup V_{for}(Q, p)$$

Example 5. Let Q be the SXQ query in the Example 2 and $p = 2 \cdot 2$ a position in $Pos(Q)$. The subquery $Q_{|p}$ corresponds to the expression:

```
for $x5 in $x2/child::reviews return
 for $x6 in $x5/child::entry  return
  for $x7 in $x4/child::title return
   for $x8 in $x6/child::title return
    if ($x7 = $x8) then
     for $x9 in $x6/child::title return
       for $x10 in $x6/child::review return <rev> $x9  $x10 </rev>
```

The set of variables $free(Q)$ contains the only two variables $\{\$x_1, \$x_2\}$ and following Definition 1, $free(Q_{|p}) = \{\$x_2, \$x_4\}$. Notice variable $\$x_1 \in free(Q)$ but $\$x_1 \notin free(Q_{|p})$. By Definition 4, $V_{for}(Q, p) = \{\$x_3, \$x_4\}$ and following Definition 5, $Rel(Q, p) = \{\$x_1, \$x_2, \$x_3, \$x_4\}$.

Note that, the set $Rel(Q, p)$ collects all the free variables occurring in the query Q and all the variables introduced by *for* statements in positions q with $q < p$. Thus, the set $Rel(Q, p)$ contains all the variables that could appear free in the subquery $Q_{|p}$. Next Lemma formalizes this idea.

Lemma 2. *Let Q be a SXQ query and $p \in Pos(Q)$ be a position of the query Q. If $\$x \in free(Q_{|p})$, then $\$x \in Rel(Q, p)$.*

Proof. The result can be proved by induction on the length of p.

- For $p = \varepsilon$, we have $Q_{|\varepsilon} == Q$. By Definition 5, $Rel(Q, \varepsilon) := free(Q) \cup V_{for}(Q, \varepsilon)$, and by Definition 4, item 1, $Rel(Q, \varepsilon) := free(Q) \cup \emptyset = free(Q_{|\varepsilon})$ which shows the result.
- Now, assume that $p = i \cdot q$. Because $i \cdot q \in Pos(Q)$, the query Q is of the form:
  - $Q \equiv Q_1\ Q_2$. In this case, i can be either 1 or 2. Suppose $i = 1$. Then:
    (a) By Definition 5, $Rel(Q, 1{\cdot}q) = free(Q) \cup V_{for}(Q, 1{\cdot}q)$ and by Definition 4, item 2, $Rel(Q, 1 \cdot q) = free(Q) \cup V_{for}(Q_1, q)$.
    (b) $free(Q_{|1\cdot q}) = free(Q_{1_{|q}})$. If $x \in free(Q_{|1\cdot q})$, then $x \in free(Q_{1_{|q}})$. By induction we obtain $x \in Rel(Q_1, q)$. By Definition 5, $Rel(Q_1, q) = free(Q_1) \cup V_{for}(Q_1, q)$. Now, $x \in Rel(Q_1, q)$ implies either $x \in free(Q_1)$ or $x \in V_{for}(Q_1, q)$.
    * If $x \in free(Q_1)$, then $x \in free(Q_{|1})$. Applying induction we obtain $x \in Rel(Q, 1)$. By Definition 5, $Rel(Q, 1) = free(Q) \cup V_{for}(Q, 1) = free(Q) \cup V_{for}(Q_1, \varepsilon) = free(Q) \cup \emptyset$ (by Definition 4). Then, $x \in free(Q)$ and by (a) $x \in Rel(Q, 1 \cdot q)$ which shows the result.
    * If $x \in V_{for}(Q_1, q)$, then, by (a), $x \in Rel(Q, 1 \cdot q)$ and the result holds.

    If $i = 2$, the result can be proved similarly.

  - $Q \equiv$ for $\$x_k$ in Q_1 return Q_2. If $i = 1$ the result can be proved similarly to the previous case. Now, suppose $i = 2$. Then:
    (a) $Q_{|p} = Q_{|2\cdot q} = (Q_2)_{|q}$ (by Definition 3)
    (b) By Definition 5, $Rel(Q, 2{\cdot}q) = free(Q) \cup V_{for}(Q, 2{\cdot}q)$ and by Definition 4, item 4, $Rel(Q, 2 \cdot q) = free(Q) \cup \{\$x_i\} \cup V_{for}(Q_2, q)$.
    (c) $free(Q_{|2\cdot q}) = free(Q_{2_{|q}})$. If $x \in free(Q_{|2\cdot q})$, then $x \in free(Q_{2_{|q}})$. By induction we obtain $x \in Rel(Q_2, q)$. By Definition 5, $Rel(Q_2, q) = free(Q_2) \cup V_{for}(Q_2, q)$. Now, $x \in Rel(Q_2, q)$ implies either $x \in free(Q_2)$ or $x \in V_{for}(Q_2, q)$.
    * If $x \in free(Q_2)$, then $x \in free(Q_{|2})$. Applying induction we obtain $x \in Rel(Q, 2)$. By Definition 5, $Rel(Q, 2) = free(Q) \cup V_{for}(Q, 2) = free(Q) \cup \{\$x_i\} \cup V_{for}(Q_2, \varepsilon) = free(Q) \cup \{\$x_i\} \cup \emptyset$ (by Definition 4). Then, $\$x \in Rel(()Q, 2)$ implies either $\$x \in free(Q)$ or $\$x = \$x_i$ and by (b), the result holds.
    * If $x \in V_{for}(Q_2, q)$, then, by (b), $x \in Rel(Q, 2 \cdot q)$ and the result holds.

    The rest of the cases can be proved similarly.

Next lemma presents some properties of the set of relevant variables for a query Q at position p.

Lemma 3. *Let Q' be a SXQ query and $p \cdot i \in Pos(Q')$ be a position of the query Q' such that $Q'_{|p} = Q$ and $i \in \{1, 2\}$. Then,*

- *If $Q \equiv$ for $\$x$ in Q_1 return Q_2, then:*
  - $Rel(Q', p \cdot 1) = Rel(Q', p)$
  - $Rel(Q', p \cdot 2) = Rel(Q', p) \cup \{\$x\}$
- *For the rest of the cases, $Rel(Q', p \cdot i) = Rel(Q', p)$*

Proof. We distinguish cases depending of the form of the query Q.

- $Q \equiv Q_1\ Q_2$.
  - If $i = 1$, $Q'_{|p\cdot 1} \equiv Q_{|1} \equiv Q_1$ is an SXQ query. Note that, $p \cdot 1 \in Pos(Q')$. Then,

$$\begin{array}{ll} Rel(Q', p \cdot 1) & = \text{(by Definition 5)} \\ free(Q') \cup V_{for}(Q', p \cdot 1) & = \text{(by Lemma 1)} \\ free(Q') \cup V_{for}(Q', p) \cup V_{for}(Q'_{|p}, 1) & = \text{(by Definition 4, rule 2)} \\ free(Q') \cup V_{for}(Q', p) \cup V_{for}(Q'_{|p\cdot 1}, \varepsilon) & = \text{(by Definition 4, rule 1)} \\ free(Q') \cup V_{for}(Q', p) \cup \emptyset & = \text{(by Definition 5)} \\ Rel(Q', p) & \end{array}$$

  - If $i = 2$, the result $Rel(Q', p \cdot 1) = Rel(Q', p)$ can be proved similarly to the previous case.
- $Q \equiv$ for $\$x$ in Q_1 return Q_2. This query introduces a new variable by means of a *for* statement.
  - In the case of $i = 1$, $Q'_{|p\cdot 1} \equiv Q_1$. Then,

$$\begin{array}{ll} Rel(Q', p \cdot 1) & = \text{(by Definition 5)} \\ free(Q') \cup V_{for}(Q', p \cdot 1) & = \text{(by Lemma 1)} \\ free(Q') \cup V_{for}(Q', p) \cup V_{for}(Q'_{|p}, 1) & = \text{(by Definition 4, rule 3)} \\ free(Q') \cup V_{for}(Q', p) \cup V_{for}(Q'_{|p\cdot 1}, \varepsilon) & = \text{(by Definition 4, rule 1)} \\ free(Q') \cup V_{for}(Q', p) \cup \emptyset & = \text{(by Definition 5)} \\ Rel(Q', p) & \end{array}$$

    which shows the result.
  - In the case of $i = 2$, $Q'_{|p\cdot 2} \equiv Q_2$. Then,

$$\begin{array}{ll} Rel(Q', p \cdot 2) & = \text{(by Definition 5)} \\ free(Q') \cup V_{for}(Q', p \cdot 2) & = \text{(by Lemma 1)} \\ free(Q') \cup V_{for}(Q', p) \cup V_{for}(Q'_{|p}, 2) & = \text{(by Definition 4, rule 4)} \\ free(Q') \cup V_{for}(Q', p) \cup \{\$x\} \cup V_{for}(Q'_{|p\cdot 2}, \varepsilon) & = \text{(by Definition 4, rule 1)} \\ free(Q') \cup V_{for}(Q', p) \cup \{\$x\} \cup \emptyset & = \text{(by Definition 5)} \\ Rel(Q', p) \cup \{\$x\} & \end{array}$$

    which shows the result.

The rest of the cases are analogous to the previous cases.

2.2 XQ Operational Semantics

The semantics of XQ can be found in [3]. We will use XML documents represented as *data trees*. A *data forest* is a sequence of data trees and an *indexed forest* is a pair consisting of a data forest and a sequence of nodes in it.

Figure 2 introduces the operational semantics of an SXQ expression α with at most k free variables using a function $[\![\alpha]\!]_k$ that takes a data forest $\mathcal{F}$ and a $k-$tuple of nodes from the forest as input and returns an indexed forest. The input $k-$tuple of nodes represents an assignment of nodes to a given $k-$variables, that is, each variable $\$x_i$ is pointing to the node n_i, $1 \leq i \leq k$. The differences of the semantics of SXQ with respect to the semantics of XQ in [3] are:

- There is no rule for the constructor *lab*.
- There is no rule for the empty query represented by ().
- There is a new rule for the query represented by a sequence of variables inside a tag.

$$
\begin{array}{lrl}
\mathbf{XQ_1} & [\![\alpha\ \beta]\!]_k(\mathcal{F},\overline{e}) & := [\![\alpha]\!]_k(\mathcal{F},\overline{e}) \uplus [\![\beta]\!]_k(\mathcal{F},\overline{e}) \\
\mathbf{XQ_2} & [\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(\mathcal{F},\overline{e}) & := \text{let } (\mathcal{F}',\overline{l}) = [\![\alpha]\!]_k(\mathcal{F},\overline{e}) \text{ in} \\
 & & \quad \biguplus_{1\leq i\leq|\overline{l}|} [\![\beta]\!]_{k+1}(\mathcal{F}',\overline{e}\cdot\overline{l}_i) \\
\mathbf{XQ_3} & [\![\$x_i]\!]_k(\mathcal{F},t_1,\ldots,t_k) & := (\mathcal{F},[t_i]) \\
\mathbf{XQ_4} & [\![\$x_i/\chi :: \nu]\!]_k(\mathcal{F},t_1,\ldots,t_k) & := (\mathcal{F}, \text{list of nodes } v \text{ such that } \chi^{\mathcal{F}}(t_i,v) \text{ and} \\
 & & \quad \text{node } v \text{ has label } \nu \text{ in order } <_{doc}^{tree(t_i)}) \\
\mathbf{XQ_5} & [\![\text{if } \phi \text{ then } \alpha]\!]_k(\mathcal{F},\overline{e}) & := \text{if } \pi_2([\![\phi]\!]_k(\mathcal{F},\overline{e})) \neq [\,] \text{ then } [\![\alpha]\!]_k(\mathcal{F},\overline{e}) \\
 & & \quad \text{else } (\mathcal{F},[\,]) \\
\mathbf{XQ_6} & [\![\text{if } \$x_i = \$x_j \text{ then } \alpha]\!]_k(\mathcal{F},t_1,\ldots,t_k) & := \text{if } t_i = t_j \text{ then } [\![\alpha]\!]_k(\mathcal{F},t_1,\ldots,t_k) \\
 & & \quad \text{else } (\mathcal{F},[\,]) \\
\mathbf{XQ_7} & [\![\langle a\rangle\ \langle/a\rangle]\!]_k(\mathcal{F},\overline{e}) & := \mathit{construct}(a,(\mathcal{F},[\,])) \\
\mathbf{XQ_8} & [\![\langle a\rangle\ \mathit{tag}\ \langle/a\rangle]\!]_k(\mathcal{F},\overline{e}) & := \mathit{construct}(a,[\![\mathit{tag}]\!]_k(\mathcal{F},\overline{e})) \\
\mathbf{XQ_9} & [\![\langle a\rangle\ \$x_i\ \ldots\ \$x_j\ \langle/a\rangle]\!]_k(\mathcal{F},t_1,\ldots,t_k) & := \mathit{construct}(a,(\mathcal{F},[t_i,\ldots,t_j]))
\end{array}
$$

Fig. 2. Semantics of SXQ

This semantics makes use of some functions that construct indexed forest. The operator $\mathit{construct}(a,\ (\mathcal{F},\ [w_1 ... w_n]))$, denotes the construction of a new tree, where a is a label, $\mathcal{F}$ is a data forest, and $[w_1 \ldots w_n]$ is a list of nodes in $\mathcal{F}$. When applied, *construct* returns an indexed forest $(\mathcal{F} \uplus T', [root(T')])$, where T' is a data tree with domain a new set of nodes, whose root is labeled with a, and with the subtree rooted at the i-th (in sibling order) child of root(T') being an isomorphic copy of the subtree rooted by w_i in $\mathcal{F}$. The symbol $\biguplus$ used in the rules takes two indexed forests $(\mathcal{F}_1,\overline{l}_1),(\mathcal{F}_2,\overline{l}_2)$ where the $\mathcal{F}_i$ are a data forest and $\overline{l}_i$ are lists of nodes in $\mathcal{F}_i$, and returns an indexed forest $(\mathcal{F}_1 \cup \mathcal{F}_2,\overline{l})$, where $\overline{l}$ is the concatenation of $\overline{l}_1$ and $\overline{l}_2$.

For a data tree $\mathcal{F}$, we let the binary relation $<_{doc}^{\mathcal{F}}$ on nodes be the *document-order* on $\mathcal{F}$: the depth-first left-to-right traversal order through $\mathcal{F}$. In the semantics of $\$x_i/\chi :: \nu$ we use $tree(t_i)$ to denote the maximal tree within the input forest that contains the node t_i, hence $<_{doc}^{tree(t_i)}$ is the document-order on the tree containing t_i. $\chi^{\mathcal{F}}$ is the interpretation of the axis relation of the same name in the data forest.

These semantic rules constitute a term rewriting system (TRS in short, see [2]), with each rule defining a single reduction step. The symbol $:=^*$ represents the reflexive and transitive closure of $:=$ as usual. The TRS is terminating and confluent (the rules are not overlapping).

As explained in [3], this semantics does not model the document() function of XQuery. Instead, we assume that there exist one or more initial variables that are each bound to a node of the input forest.

Given a SXQ query Q with $free(Q) = \{\$x_1, \ldots, \$x_k\}$, the semantics evaluates a query Q starting with the expression $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$. The initial data forest $\mathcal{F}$ is a forest containing k input XML documents represented as data trees as explained in [3]. Each variable $\$x_i$ in $free(Q)$ represents an initial XML document and it is bound to a node of the input data forest $\mathcal{F}$. The sequence of nodes $t_1, \ldots, t_k$ from $\mathcal{F}$, corresponds to the nodes assigned to the variables $\{\$x_1, \ldots, \$x_k\}$. Along intermediate steps, expressions of the form $[\![Q']\!]_{k+n}(\mathcal{F}', t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+n})$ are obtained. The idea is that Q' is the subquery that can be found in Q at some position $p \in Pos(Q)$, and the set $Rel(Q, p)$ contains $k + n$ variables. The data forest $\mathcal{F}'$ is built from the input data forest $\mathcal{F}$ by adding (possible) new data trees, which are constructed by the operator *construct* representing new XML fragments.

The evaluation of a query returns as a result an indexed forest as a pair of the form $(\mathcal{F}', [e_1, \ldots, e_m])$ meaning that the query returns a sequence of $m-$nodes from $\mathcal{F}'$ representing XML fragments.

A more detailed discussion about this semantics and its properties can be found in [3].

3 $\mathcal{TOY}$ and Its Semantics

A $\mathcal{TOY}$ [14] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of (total)*expressions* in $\mathcal{TOY}$ $e \in Exp$ is $e ::= X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). Similarly, the syntax of (total)*patterns* $t \in Pat \subset Exp$ can be defined as $t ::= X \mid c\ t_1 \ldots t_m \mid f\ t_1 \ldots t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are patterns for all $1 \leq i \leq m$. The set of *partial expressions* $Exp_\perp$ is the result of incorporating the new constant (0-arity constructor) $\perp$ to Exp. This constant plays the role

of the undefined value. Similarly, the set of *partial patterns* $Pat_\perp$ is the result of incorporating the constant $\perp$ to Pat.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$:

```
data node      = txt    string
               | comment string
               | tag    string [attribute] [node]
data attribute = att    string string
type xml       = node
```

The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Of course, this list is not exhaustive, since it misses several types of XML nodes, but it is enough for this presentation.
Each rule for a function f in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \ \rightarrow \ \underbrace{r}_{\text{right-hand side}} \ \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns.
In $\mathcal{TOY}$, variable names must start with either an uppercase letter or an underscore (for anonymous variables), whereas other identifiers start with lowercase. $\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience, primitive `load_xml_file` includes a dummy tag `"root"` at the outer level. This is useful for grouping several XML fragments. If the file contains only one node `N` at the outer level, the `root` node is unnecessary, and can be removed using this simple function:

```
load_doc F = N <== load_xml_file F == xmlTag "root" [] [N]
```

where `F` is the name of the file containing the document. Observe that the strict equality `==` in the condition forces the evaluation of `load_xml_file F` and succeeds if the result has the form `xmlTag "root" [] [N]` for some `N`. If this is the case, `N` is returned.

The constructor-based ReWriting Logic (CRWL) [15] has been proposed as a suitable declarative semantics for functional-logic programming with lazy nondeterministic functions. The calculus is defined by five inference rules (see Figure 3): (BT) that indicates that any expression can be approximated by bottom, (RR) that establishes the reflexivity over variables, the decomposition rule (DC),

BT $e \to \bot$

RR $X \to X$ with $X \in Var$

DC $\dfrac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m}$ $h\ \overline{t}_m \in Pat_\bot$

JN $\dfrac{e \to t \quad e' \to t}{e \ == \ e'}$ $t \in Pat$ (total pattern)

FA $\dfrac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \ \to t}$
if $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot, t \neq \bot$

Fig. 3. CRWL Semantic Calculus

the (JN) (join) rule that indicates how to prove strict equalities, and the function application rule (FA). In every inference rule, $e, e_i \in Exp_\bot$ are partial expressions and $t_i, t, s \in Pat_\bot$ are partial patterns. The notation $[P]_\bot$ in the inference rule *FA* represents the set $\{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in P,\ \theta \in Subst_\bot\}$ of partial instances of the rules in the program P. The most complex inference rule is *FA* (Function Application), which formalizes the steps for computing a *partial pattern* t as approximation of a function call $f\ \overline{e}_n$:

1. Obtain partial patterns t_i as suitable approximations of the arguments e_i.
2. Apply a program rule $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot$, verify the condition C, and check that t approximates the right-hand side r.

In this semantic notation, local declarations $a = b$ introduced in $\mathcal{TOY}$ syntax by the reserved word `where` are represented as part of the condition C as *approximation statements* of the form $b \to a$.

The semantics in $\mathcal{TOY}$ allows introducing non-deterministic functions, such as the following function `member` that returns all the elements in a list:

```
member:: [A]  -> A
member  [X | Xs] =  X
member  [X | Xs] =  member Xs
```

Another example of $\mathcal{TOY}$ function is the definition of the infix operator `.:::` which corresponds to the function composition:

```
infixr 90 .:::
(.:::) :: (A -> B) -> (B -> C) -> (A -> C)
(F .:::. G) X = G (F X)
```

As the examples show, $\mathcal{TOY}$ is a typed language. However, the type declaration is optional and in the rest of the report they are omitted for the sake of simplicity. *Goals* in $\mathcal{TOY}$ are sequences of strict equalities. A strict equality

$e_1 == e_2$ holds (inference JN) if both e_1 and e_2 can be reduced to the same total pattern t. For instance, the goal `member [1,2,3,4] == R` yields four answers, the four values for `R` that make the equality true: $\{\texttt{R} \mapsto \texttt{1}\}, \ldots, \{\texttt{R} \mapsto \texttt{4}\}$.

The next lemma presents some easy consequences of the inference rules that are used in the proof of the main theoretical results.

Lemma 4. *Let t_1, t_2 be patterns and e be an expression. Then*

1. *If $\mathcal{P} \vdash t_1 \to t_2$, and t_2 is total, then $t_1 \equiv t_2$ (the symbol $\equiv$ is used to represent syntactic equivalence).*
2. *If $\mathcal{P} \vdash t_1 == t_2$, then $t_1 \equiv t_2$.*
3. *$\mathcal{P} \vdash e == t_1$, iff $\mathcal{P} \vdash e \to t_1$ and t_1 total.*
4. *It is always possible to prove $\mathcal{P} \vdash t_1 \to t_1$.*

Proof.

1. By structural induction on t_2. First observe that t_2 cannot contain $\bot$ because it is total, and that therefore the inference BT is never applied. If t_2 is a variable X, then the only inference applicable is RR and t_1 is also X. If $t_2 = h\ \overline{s'}_n$ for some patterns s'_i, then the only possible inference is DC, which implies that $t_1 = h\ \overline{s}_n$, and the result follows applying the inductive hypothesis to the premises.
2. The first step of the proof must consists of a JN inference rule. Thus, there is some total pattern t such that $\mathcal{P} \vdash t_1 \to t$, $\mathcal{P} \vdash t_2 \to t$. Then from the previous item, $t_1 \equiv t$, $t_2 \equiv t$, and therefore $t_1 \equiv t_2$.
3. First, assume $\mathcal{P} \vdash e == t_1$. In the premises of the JN we find $\mathcal{P} \vdash t_1 \to t$ for some total patter t. Then from the first item $t_1 \equiv t$. The other premise of the JN inference is $\mathcal{P} \vdash e \to t$, that is $\mathcal{P} \vdash e \to t_1$. Now suppose that $\mathcal{P} \vdash e \to t_1$ and t_1 total. Then we can prove $\mathcal{P} \vdash e == t_1$ taking $t \equiv t_1$ as the total pattern required by the JN inference.
4. If $t_1 \equiv \bot$ then the proof consists of a BT inference step, if it is a variable of a RR step, and if it is of the form $t_1 \equiv c\ \overline{s}_n$ of a DC step with premises $s_i \to s_i$ that can be proven in CRWL by induction hypothesis.

4 Transforming SXQ into $\mathcal{TOY}$

In order to represent SXQ queries in $\mathcal{TOY}$ we use some auxiliary datatypes:

```
type xPath = xml-> xml

data sxq  = xfor xml sxq sxq | xif cond sxq | xmlExp xml |
                    xp path | comp sxq sxq
data cond = sxq := sxq | cond sxq
data path = var xml | xml :/ xPath | doc string xPath
```

The structure of the datatype `sxq` allows representing any SXQ query (see Figure 4). It is worth noticing that a variable introduced by a *for* statement has type `xml`, indicating that the variable always contains a value of this type.

SXQ query	SXQ query in $\mathcal{TOY}$
query query	`comp sxq sxq`
for *var* **in** *query* **return** *query*	`xfor xml sxq sxq`
if *cond* **then** *query*	`xif cond sxq`
var	`xp (var xml)`
tag	`xmlExp xml`
$var/axis :: \nu$	`xp (xml :/ xPath)`
$var = var$	`sxq := sxq`

Fig. 4. Representation of SXQ queries in Figure 1 as a $\mathcal{TOY}$ terms

$\mathcal{TOY}$ includes a primitive `parse_xquery` that translates any SXQ expression into its corresponding representation as a term of this datatype, as the next example shows:

Example 6. The translation of the SXQ query of Example 2 into the datatype `sxq` produces the following $\mathcal{TOY}$ dataterm:

```
Toy> parse_xquery  "for $x3 in $x1/child::bib return
          for $x4 in .....  <rev> $x9  $x10 </rev>" == R
yes
{R --> xfor X3 (xp ( X1 :/ (child .::. (nameT "bib"))))
    (xfor X4 (xp ( X3 :/ (child .::.(nameT "book"))))
    (xfor X5 (xp ( X2 :/ (child .::.(nameT "reviews"))))
    (xfor X6 (xp ( X5 :/ (child .::.(nameT "entry"))))
    (xfor X7 (xp ( X4 :/ (child .::.(nameT "title"))))
    (xfor X8 (xp ( X6 :/ (child .::.(nameT "title"))))
    (xif  ( xp ( var X7 ) :=  xp ( var X8) )
    (xfor X9 (xp ( X6 :/ (child .::.(nameT "title"))))
    (xfor X10 (xp ( X6 :/ (child .::.(nameT "review"))))
       (xmlExp (xmlTag "rev" [] [X9, X10])))))))))))
 }
```

The primitive `parse_xquery` takes as input a SXQ expression Q and returns as ouput the query `Q` represented as a $\mathcal{TOY}$ dataterm.

Without loss of generality, in order to simplify our implementation, the primitive `parse_xquery` also allows as input queries without free variables. This is possible by replacing all the free variables in the SXQ query Q by its corresponding XML files. That is, in Example 6 instead of variables $\$x1$ and $\$x2$ we have the strings "doc(bib.xml)" and "doc(reviews.xml)" respectively.

The interpreter assumes the existence of the infix operator `.::.` that connects axes and tests to build steps (the operator `::` in XPath syntax), defined as the sequence of applications in Chapter 3.

The rules of the $\mathcal{TOY}$ interpreter that processes SXQ queries can be found in Figure 5. The main function is `sxq`, which distinguishes cases depending of the form of the query. If it is an XPath expression then the auxiliary function

```
sxq (xp E)              = sxqPath E
sxq (xmlExp X)          = X
sxq (comp Q1 Q2)        = sxq Q1
sxq (comp Q1 Q2)        = sxq Q2
sxq (xfor X Q1 Q2)      = sxq Q2 <== X== sxq Q1
sxq (xif (Q1:=Q2) Q3)   = sxq Q3 <== sxq Q1  == sxq Q2
sxq (xif (cond Q1) Q2) = sxq Q2 <== sxq Q1  == _

sxqPath (var X)  = X
sxqPath (X :/ S) = S X
sxqPath (doc F S) = S (load_xml_file F)

%%%% XPATH %%%%%%
attr A (xmlTag S Attr L ) = xmlText T <== member Attr == xmlAtt A T
nameT S (xmlTag S Attr L ) = xmlTag S Attr L
nodeT X = X
textT (xmlText S) = xmlText S
commentT S (xmlComment S) = xmlComment S

self X = X
child (xmlTag _Name _Attr L) = member L
descendant X = child X
descendant X = descendant Y <== child X == Y
dos = self
dos = descendant
```

Fig. 5. $\mathcal{TOY}$ transformation rules for SXQ

`sxqPath` is used. If the query is an XML expression, the expression is just returned (this is safe thanks to our constraint of allowing only variables inside XML expressions). If we have two queries (`comp` construct), the result of evaluating any of them is returned using non-determinism. The `for` statement (`xfor` construct) forces the evaluation of the query `Q1` and binds the variable `X` to the result. Then the result query `Q2` is evaluated. The case of the `if` statement is analogous. The XPath subset considered includes tests for attributes (`attr`), label names (`nameT`), general elements (`nodeT`), text nodes (`textT`) and comments (`commentT`). It also includes the axes `self`, `child`, `descendant` and *descendant or self* (called here `dos` as usual in XQuery). Observe that we do not include reverse axes like `ancestor` because they can be replaced by expressions including forward axes, as shown in [16,4]. Other constructions such as filters can be easily included (see [4]). The next example uses the interpreter to obtain the answers for the query of our running example.

Example 7. The goal:

```
Toy> sxq (parse_xquery "for....") == R
```

applies the interpreter of Figure 5 to the code of Example 6 (assuming that the string after `parse_xquery` is the query in Example 2), and returns the $\mathcal{TOY}$ representation of the expected results:

```
<rev>
 <title>TCP/IP Illustrated</title>
 <review>  One of the best books on TCP/IP.  </review>
</rev>
...
```

Regarding performance, the current main limitation is that the primitive `load_xml_file` cannot load documents with size beyond a few megabytes. Our experiments with these medium-size files indicate that the interpreter computes the answer in a reasonable amount of time, even for complex queries.

4.1 Soundness of the Transformation

One of the goals of this report is to ensure that the embedding is semantically correct and complete. This section introduces the theoretical results establishing these properties. If V is a set of indexed variables of the form $\{X_1, \ldots, X_n\}$ and θ a substitution on these variables, we use the notation $\theta(V)$ to indicate the sequence $\theta(X_1), \ldots, \theta(X_n)$. In the following results it is implicitly assumed that there is a bijective mapping f from XML format to the datatype `xml` in $\mathcal{TOY}$. However, in order to simplify the presentation, we omit the explicit mention to f and to its inverse f^{-1}. Also, variables in SXQ queries, with names of the form $\$x_i$ are assumed to be represented in $\mathcal{TOY}$ as X_i and conversely.

Lemma 5. *Let P be a $\mathcal{TOY}$ program, Q' be a SXQ query and `Q'` the representation of Q as a $\mathcal{TOY}$ dataterm. Let $p \in Pos(Q')$ be a position of the query Q' such that $Q'_{|_p} \equiv Q$ with $Rel(Q', p) = \{\$x_1, \ldots, \$x_k\}$ (see Definition 5). Let θ be a substitution such that $dom(\theta) = Rel(Q', p)$ and $\mathcal{P} \vdash$ (`sxq Q`θ `== t`) for some pattern `t`.*
Then, for every data forest $\mathcal{F}$, containing the list of nodes $t_1, \ldots, t_k$ with $t_1 = \theta(\$x_1), \ldots, t_k = \theta(\$x_k)$, there exists an indexed forest $(\mathcal{F}', L')$ such that:

$$[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L')$$

verifying $t \in L'$.

Proof. Observe that from $\mathcal{P} \vdash$ (`sxq Q`θ `== t`) and by Lemma 4, item 3 we have $\mathcal{P} \vdash$ (`sxq Q`$\theta \rightarrow$ `t`). Suppose $\theta = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. We prove by complete induction on the structure of Q that if $\mathcal{P} \vdash$ (`sxq Q`$\theta \rightarrow$ `t`) then for all data forest $\mathcal{F}$ containing the list of nodes $t_1, \ldots, t_k$, there exists an indexed forest $(\mathcal{F}', L')$ such that: $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L')$, verifying $t \in L'$.

- $Q \equiv Q_1\ Q_2$. The query Q is represented in $\mathcal{TOY}$ as `comp Q1 Q2`. Any proof of $\mathcal{P} \vdash$ `sxq (comp Q1 Q2)`$\theta \rightarrow$ `t` must start by a (FA) CRWL reduction step

(see Figure 3), which must use an instance either of the rule `sxq (comp Q1 Q2) = sxq Q1` or of the rule `sxq (comp Q1 Q2) = sxq Q2`. Assume the first rule is used (analogous for the second one).
Applying the rule instance `sxq (comp Q1' Q2') = sxq Q1'`, the proof of $\mathcal{P} \vdash$ `sxq (comp Q1 Q2)`$\theta \rightarrow$ `t` is of the form:

$$\frac{\texttt{(comp (Q1 Q2))}\theta \rightarrow \texttt{(comp (Q1' Q2'))}\sigma \qquad \texttt{sxq Q1'}\sigma \rightarrow \texttt{t}}{\texttt{sxq (comp (Q1 Q2))}\theta \ \rightarrow\ \texttt{t}}$$

with $\sigma = \{\texttt{Q1}' \mapsto \texttt{Q1}, \texttt{Q2}' \mapsto \texttt{Q2}\} \cdot \theta$. Then the (FA) inference step has a premise proving $\mathcal{P} \vdash$`sxq Q1`$\theta \rightarrow t$.
We check that the induction hypothesis can be applied to Q_1 verifying that it satisfies the premises of the lemma.

- $Q_1 \equiv Q_{|1}$ then $Q_1 \equiv Q'_{|p\cdot 1}$ is an SXQ query. Note that, $p \cdot 1 \in Pos(Q')$.
- By Lemma 3, $Rel(Q', p \cdot 1) = Rel(Q', p)$.

In SXQ : By $\mathbf{XQ_1}$

$$(1) \qquad [\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := [\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) \ \biguplus\ [\![Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$$

By the induction hypothesis:

$$(2) \qquad [\![Q_1]\!]_k(\mathcal{F},\, t_1, \ldots, t_k) :=^* (\mathcal{F}_1,\, L_1)$$

with $\mathcal{F}_1$ some forest containing the nodes in the list L_1, verifying $t \in L_1$. Combining (1) and (2) and considering that $:=$ is normalizing, that is:

$$[\![Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}_2, L_2)$$

for some indexed forest $(\mathcal{F}_2, L_2)$. Then:

$$[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}_1 \cup \mathcal{F}_2,\, L_1 +\!\!+ L_2)$$

with $t \in L_1$ and thus $t \in L_1 +\!\!+ L_2$.

– $Q \equiv$ for $\$x_{k+1}$ in Q_1 return Q_2. This query introduces a new variable by means of a *for* statement.
The query Q is represented in $\mathcal{TOY}$ as `xfor` $\texttt{X}_{k+1}$ `Q1 Q2`. Then any proof of $\mathcal{P} \vdash$ `sxq Q`$\theta \rightarrow$ `t` must start with a (FA) inference using a variant of the program rule: `sxq (xfor X Q1' Q2') = sxq Q2' <== X== sxq Q1'`.

$$\frac{\begin{array}{ll}(1) & \texttt{(xfor } \texttt{X}_{k+1} \texttt{ Q1 Q2)}\theta \rightarrow \texttt{(xfor X Q1' Q2')}\sigma \\ (2) & \texttt{sxq Q1'}\sigma \ ==\ \texttt{X}\sigma \\ (3) & \texttt{sxq Q2'}\sigma \rightarrow \texttt{t}\end{array}}{\texttt{(sxq (xfor } \texttt{X}_{k+1} \texttt{ Q1 Q2))}\theta \ \rightarrow\ \texttt{t}}$$

with $\sigma = \{\texttt{X} \mapsto \texttt{X}_{k+1}, \texttt{Q1}' \mapsto \texttt{Q1}, \texttt{Q2}' \mapsto \texttt{Q2}\} \cdot \theta$. This proof can be rewritten as:

$$\frac{\begin{array}{ll}(1) & \texttt{(xfor } \texttt{X}_{k+1} \texttt{ Q1 Q2)}\theta \rightarrow \texttt{(xfor X Q1' Q2')}\sigma \\ (2) & \texttt{sxq Q1}\theta \ ==\ \texttt{X}_{k+1}\theta \\ (3) & \texttt{sxq Q2}\theta \rightarrow \texttt{t}\end{array}}{\texttt{(sxq (xfor } \texttt{X}_{k+1} \texttt{ Q1 Q2))}\theta \ \rightarrow\ \texttt{t}}$$

There is a CRWL proof for the three premises. The strict equality $\texttt{sxq Q1}\theta$ == $\texttt{X}_{k+1}\theta$ holds (inference JN); there is a term `t'` such that both $\texttt{sxq Q1}\theta$ and $\texttt{X}_{k+1}\theta$ can be reduced to `t'`. The proof must be of the form:

$$\frac{\begin{array}{l}(\texttt{sxq Q1})\theta' \to \texttt{t'} \\ \texttt{X}_{k+1}\theta' \to \texttt{t'}\end{array}}{\texttt{sxq Q1}\theta \;==\; \texttt{X}_{k+1}\theta}$$

with $\theta' = \{\texttt{X}_{k+1} \mapsto \texttt{t'}\} \cdot \theta$.
Next we check that Q_1 and Q_2 verify the lemma premises, and that hence it is possible to apply the induction hypothesis to Q_1 and Q_2.

In the case of Q_1:
- $Q_1 \equiv Q'_{|p\cdot 1}$.
- By Lemma 3, $Rel(Q', p \cdot 1) = Rel(Q', p)$.

Then applying induction:

$$[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}_1, L_1)$$

with $\mathcal{F}_1$ some data forest containing all the nodes in L_1 and verifying:

$$t' \in L_1 \tag{3}$$

Notice that for all data tree $\mathcal{T} \in \mathcal{F}$, $\mathcal{T} \in \mathcal{F}_1$. Then it is posible to ensure that the data forest $\mathcal{F}_1$ contains all the nodes in the list $\{t_1, \ldots, t_k\}$.

In the case of Q_2:
- $Q_2 \equiv Q'_{|p\cdot 2}$.
- By Lemma 3, $Rel(Q', p \cdot 2) = Rel(Q', p) \cup \{\$x_{k+1}\}$.
- Additionally, in the premises of the CRWL proof there is a CRWL proof for $\mathcal{P} \vdash \texttt{sxq Q2}\theta \to \texttt{t}$. The proof must use a substitution of the form $\theta' = \{\texttt{X}_{k+1} \mapsto \texttt{t'}\} \cdot \theta$. Then, there is a CRWL proof for $\mathcal{P} \vdash \texttt{sxq Q2}\theta' \to \texttt{t}$.

Then applying induction:

$$[\![Q_2]\!]_{k+1}(\mathcal{F}_1, t_1, \ldots, t_k, t') :=^* (\mathcal{F}_2, L_2), \quad t \in L_2 \tag{4}$$

In SXQ , by $\mathbf{XQ_2}$:

$[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := \biguplus_{1 \le j \le |L_1|} [\![Q_2]\!]_{k+1}(\mathcal{F}_1, t_1, \ldots, t_k, l_j) = (\mathcal{F}', L')$
with:
$[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}_1, L_1)$, with l_j the l_j-th node in L_1.
Then from (3) t' is one of these l_j, and from (4) $t \in L_2$, and thus $t \in L'$.

– $Q \equiv \$x_i$. The query Q is such that $Q'_{|p} \equiv Q$ for $p \in Pos(Q')$. By Definition 1, $free(Q'_{|p}) = \{\$x_i\}$ and by Lemma 2, $\$x_i \in Rel(Q', p) = \{\$x_1, \ldots, \$x_k\}$. The representation of this query in $\mathcal{TOY}$ will be `xp (var X_i)`. Any proof for $\mathcal{P} \vdash \texttt{sxq (xp (var X}_i\texttt{))}\theta \to \texttt{t}$ must start with a (FA) inference using a variant of the program rule `sxq (xp E) = sxqPath E`. Therefore this inference has a premise proving $\mathcal{P} \vdash \texttt{sxqPath (var X}_i\texttt{)}\theta \to t$.

This proof must use again the (FA) inference, this time applying the rule `sxqPath (var X) = X`. Therefore in the proof of this statement we find a proof for $\mathcal{P} \vdash \mathtt{X}_i\theta \to t$, which by Lemma 4 implies that $\theta(X_i) \equiv t$.
In SXQ . Applying $\mathbf{XQ_3}$,

$$[\![\$x_i]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := (\mathcal{F}, [t_i])$$

with $\theta(x_i) = t_i$. Then $t_i = t$ and the result holds.

– $Q \equiv \$x_i/axis :: \nu$. The query Q is such that $Q'_{|p} \equiv Q$ for $p \in Pos(Q')$. By Definition 1, $free(Q'_{|p}) = \{\$x_i\}$ and by Lemma 2, $\$x_i \in Rel(Q', p) = \{\$x_1, \ldots, \$x_k\}$.
We check the case where the axis is `child` and the test a node name (the proof is analogous for the rest of axes and tests). In this case the representation in $\mathcal{TOY}$ of the query Q is: `xp` $\mathtt{X}_i$ `:/ child .::. (nameT "name")`. From the premise $\mathcal{P} \vdash$ `sxq` $\mathtt{Q}\theta$ `==` `t` and by Lemma 4 there is a CRWL proof for $\mathcal{P} \vdash$ `sxq` $\mathtt{Q}\theta \to$ `t`.
The proof must start applying a (FA) inference rule of CRWL, applying the rule `sxq (xp E) = sxqPath E` (see Figure 5). The step must be of the form:

$$\frac{\begin{array}{l}\mathtt{X}_i\theta \texttt{ :/ child .::. (nameT } \textit{name}\texttt{)} \to \mathtt{X}_i\theta \texttt{ :/ child .::. (nameT } \textit{name}\texttt{)} \\ \texttt{sxqPath (}\mathtt{X}_i\theta \texttt{ :/ child .::. (nameT } \textit{name}\texttt{))} \to \texttt{t}\end{array}}{\texttt{sxq } \mathtt{Q}\theta \to \texttt{t}}$$

with $\sigma = \{E \mapsto \mathtt{X}_i \texttt{ :/ child .::. (nameT "name")}\}\theta$.
The second premise must have a proof starting with a (FA) inference applying the second rule of `sxqPath` (that is, `sxqPath (X :/ S) = S X`, see Figure 5), and with an instance given by the substitution $\sigma = \{\mathtt{X} \mapsto \mathtt{X}_i\theta,\ S \mapsto$ `child .::. (nameT` $\textit{name}$`)`$\}$:

$$\frac{\begin{array}{l}\mathtt{X}_i\theta \to \mathtt{X}_i\theta \\ \texttt{child .::. (nameT } \textit{name}\texttt{)} \to \texttt{child .::. (nameT } \textit{name}\texttt{)} \\ \texttt{(child .::. (nameT } \textit{name}\texttt{))}\ \mathtt{X}_i\theta \to \texttt{t}\end{array}}{\texttt{sxqPath (}\mathtt{X}_i\theta \texttt{ :/ child .::. (nameT } \textit{name}\texttt{))} \to \texttt{t}}$$

The two first premises correspond to the pattern matching of parameters. The third premise is the reduction of the right-hand side, and must apply once more an FA inference, this time using the rule for the infix operator `.::.`. The used rule is `(F .::. G) X = G (F X)`, see Section 3, with instance

$$\sigma = \{\mathtt{F} \mapsto \texttt{child}, \mathtt{G} \mapsto \texttt{nameT } \textit{name}, \mathtt{X} \mapsto \mathtt{X}_i\theta\}$$

The inference must be of the form:

$$\frac{\begin{array}{l}\texttt{child} \to \texttt{child} \\ \texttt{nameT } \textit{name} \to \texttt{nameT } \textit{name} \\ \mathtt{X}_i\theta \to \mathtt{X}_i\theta \\ \texttt{(nameT } \textit{name}\texttt{) (child } \mathtt{X}_i\theta\texttt{)} \to \texttt{t}\end{array}}{\texttt{(child .::. (nameT } \textit{name}\texttt{))}\ \mathtt{X}_i\theta \to \texttt{t}}$$

Now the proof of `(child .::. (nameT name))`$\mathtt{X}_i\theta \to \mathtt{t}$ corresponds to an application of function `nameT` with two arguments: *name* and `(child` $\mathtt{X}_i\theta$`)`. The program rule for `nameT` is `nameT S (xmlTag S Attr L ) = xmlTag S Attr L`. In order to apply this rule the second argument of `nameT`, `(child` $\mathtt{X}_i\theta$`)`, must be reduced to a pattern of the form `xmlTag name Attr L`. Therefore the substitution must be $\sigma = \{\mathtt{S} \mapsto \mathit{name}\}$. Then the FA step is of the form:

$$\frac{\begin{array}{l}\mathit{name} \to \mathit{name} \\ \mathtt{child}\ \mathtt{X}_i\theta \to \mathtt{xmlTag}\ \mathit{name}\ \mathtt{Attr\ L} \\ \mathtt{xmlTag}\ \mathit{name}\ \mathtt{Attr\ L} \to \mathtt{t}\end{array}}{\mathtt{(nameT}\ \mathit{name}\mathtt{)\ (child}\ \mathtt{X}_i\theta\mathtt{)} \to \mathtt{t}}$$

Since `t` is a total pattern, from Lemma 4 applied to the third premise we have $\mathtt{t} \equiv$ `xmlTag name Attr L`, that is, `t` is the representation in $\mathcal{TOY}$ of an XML element with label *name*. The second premise implies a proof for `child` $\mathtt{X}_i\theta \to$ `xmlTag name Attr L` in CRWL. Again the rule FA is applied, this time using the program rule `child (xmlTag Name' Attr' L') = member L'` (the variables have been renamed). The instance must use a substitution of the form $\sigma = \{\mathit{Name}' \mapsto \mathtt{A}, \mathit{Attr}' \mapsto \mathtt{B}$ for some patterns `A` and `B`. The FA step must be of the form:

$$\frac{\begin{array}{l}\mathtt{X}_i\theta \to \mathtt{xmlTag\ A\ B\ L'} \\ \mathtt{member\ L'} \to \mathtt{xmlTag}\ \mathit{name}\ \mathtt{Attr\ L}\end{array}}{\mathtt{child}\ \mathtt{X}_i\theta \to \mathtt{xmlTag}\ \mathit{name}\ \mathtt{Attr\ L}}$$

It is easy to prove that `member L'` returns all the members in `L'` (by induction on the length of `L'`). Therefore:

1. $\mathtt{X}_i\theta$ is a value of the form `xmlTag A B L'` for some values `A`, `B` and $\mathtt{L}'$.
2. $\mathtt{t} \equiv$ `xmlTag name Attr L` is in $\mathtt{L}'$, which means that `t` is a child of $\mathtt{X}_i\theta$ with label *name*.

In SXQ : Observe that $\$x_i \in \mathit{Rel}(Q', p)$. Applying $\mathbf{XQ_4}$ we have that for all data forest $\mathcal{F}$, containing the list of nodes $t_1, \ldots, t_k$,

$$[\![\$x_i/\mathrm{child} :: \mathit{name}]\!]_k(\mathcal{F}, t_1, \ldots, t_k) = (\mathcal{F}, L'')$$

Then we prove that $t \in L''$. This holds because by $\mathbf{XQ_4}$, L'' is the list of nodes v such that [1]:

i) $\mathrm{child}^{\mathcal{F}}(t_i, v)$. The children of $t_i =$ `xmlTag A B L'` are the elements of L'. Then $t \in L'$, and therefore it satisfies this condition.

ii) Label `name` of $(v) = \mathit{name}$. The label of t is *name*.

Therefore $t \in L''$ as indicated in the lemma.

The rest of the cases are analogous to the previous cases.

[1] The condition about the order in the nodes in $\mathbf{XQ_4}$ is not included because it has no effect in the result.

The theorem that establishes the correctness of the approach is an easy consequence of the previous Lemma.

Theorem 1. *Let P be the $\mathcal{TOY}$ program of Figure 5, Q a SXQ query with $free(Q) = \{\$x_1, \ldots, \$x_m\}$. Let* `Q` *be the representation of Q as a $\mathcal{TOY}$ dataterm according to the table in Figure 4,* `t` *be a $\mathcal{TOY}$ pattern, and θ a substitution such that $dom(\theta) = free(Q)$ and $\mathcal{P} \vdash$* `(sxq Q`θ `== t)`*. Then, for all data forest $\mathcal{F}$ containing the nodes $t_1, \ldots, t_m$ with $t_1 = \theta(x_1), \ldots, t_m = \theta(x_m)$, there exists an indexed forest $(\mathcal{F}', L')$ such that:*

$$[\![Q]\!]_m(\mathcal{F}, t_1, \ldots, t_m) :=^* (\mathcal{F}', L')$$

verifying $t \in L'$.

Proof. In Lemma 5 consider the position $p \equiv \varepsilon$. Then $Q' \equiv Q$, $Rel(Q, p) = free(Q) \cup V_{for}(Q, \varepsilon) = free(Q) \cup \emptyset = \{\$x_1, \ldots, \$x_m\}$. Then, the conclusion of the theorem is the conclusion of the lemma.

Thus, our approach is correct. The next Lemma allows us to prove that it is also complete, in the sense that the $\mathcal{TOY}$ program can produce every answer obtained by the SXQ operational semantics.

Lemma 6. *Let P be the $\mathcal{TOY}$ program of Figure 5. Let Q' be a SXQ query and p a position in $Pos(Q')$ such that $Q \equiv Q'_{|p}$ and $Rel(Q', p) = \{\$x_1, \ldots, \$x_k\}$. Suppose that $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L)$ for some $\mathcal{F}$, $\mathcal{F}'$, $t_1, \ldots, t_k$, L. Then, for every $t \in L$, there is a substitution θ such that $\theta(\$x_i) = t_i$ for all $\$x_i \in Rel(Q', p)$ and a CRWL-proof proving $\mathcal{P} \vdash$* `sxq Q`θ `== t`*.*

Proof. Due to the Lemma 4 it is enough to prove that $\mathcal{P} \vdash$ `sxq Q`$\theta \rightarrow$ `t` by complete induction on the structure of Q.

- $Q \equiv Q_1\ Q_2$.
  In this case, by $\mathbf{XQ_1}$,
  $[\![Q_1 Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := [\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) \uplus [\![Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$
  $:=^* (\mathcal{F}_1, L_1) \uplus (\mathcal{F}_2, L_2)$
  Then t is either in L_1 or in L_2.
  - If t in L_1. Then we consider the reduction $[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}_1, L_1)$. The set of variables of Q' that are relevant for Q_1 is denoted by $Rel(Q', p \cdot 1)$, and by Lemma 3 $Rel(Q', p \cdot 1) = Rel(Q', p)$. For all $\$x_i$ in $Rel(Q', p \cdot 1)$, $\theta(\$x_i) = t_i$, and by induction hypothesis, for every $t \in L_1$, there is a CRWL-proof proving $\mathcal{P} \vdash$ `sxq Q1`$\theta \rightarrow$ `t`. Now, applying a variant of the the third rule of `sxq` (for instance, `sxq (comp Q1' Q2') = sxq Q1'`, see Figure 5), there is a CRWL proof for $\mathcal{P} \vdash$ `sxq (comp (Q1 Q2))`$\theta \rightarrow$ `t` using the substitution $\sigma = \{\texttt{Q1}' \mapsto \texttt{Q1}, \texttt{Q2}' \mapsto \texttt{Q2}\} \cdot \theta$ to obtain the rule instance. The first step of the proof is:

$$\frac{\texttt{(comp (Q1 Q2))}\theta \rightarrow \texttt{(comp (Q1' Q2'))}\sigma \qquad \texttt{sxq Q1'}\sigma \rightarrow \texttt{t}}{\texttt{sxq (comp (Q1 Q2))}\theta \ \rightarrow\ \texttt{t}}$$

The CRWL-proof of the first premise is obtained from Lemma 4 since both sides are the same term due to the definition of σ. The second premise is the result we have obtained by induction hypothesis since `sxq Q1'`σ = `sxq Q1`θ.

- If t in L_2. Analogously to the previous case, the induction hypothesis can be applied to Q_2, concluding that for every $t \in L_2$, there is some CRWL-proof proving $\mathcal{P} \vdash$ `sxq Q2` $\theta \rightarrow$ `t`, and hence for $\mathcal{P} \vdash$ `sxq (comp (Q1 Q2))`$\theta \rightarrow$ `t` using the fourth rule of `sxq` (Figure 5).

In both cases for all $t \in L_1 \cup L_2$, there is a CRWL-proof proving

$$\mathcal{P} \vdash \texttt{sxq (comp (Q1 Q2))}\theta \rightarrow \texttt{t}$$

which proves the result.

– $Q \equiv$ for $\$x_{k+1}$ in Q_1 return Q_2.
In this case, by $\mathbf{XQ_2}$,

$$[\![\text{for } \$x_{k+1} \text{ in } Q_1 \text{ return } Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := \biguplus_{1 \leq i \leq m} [\![Q_2]\!]_{k+1}(\mathcal{F}', t_1, \ldots, t_k, l_i)$$
$$:=^* (\mathcal{F}_1 \cup \cdots \cup \mathcal{F}_m, L_1 + + \cdots + + L_m)$$

where $[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := (\mathcal{F}', [l_1, \ldots l_m])$.
If $L_1 + + \cdots + + L_m = [\,]$ the result trivially holds. In other case, consider any $t \in L_1 + + \cdots + + L_m$
Now, we check the induction hypothesis can be applied to both Q_1 and Q_2.

- The set of variables of Q' that are relevant for Q_1 is denoted by $Rel(Q', p \cdot 1)$, and by Lemma 3, $Rel(Q', p) = Rel(Q', p \cdot 1)$.
  By induction hypothesis there is a CRWL-proof proving $\mathcal{P} \vdash$ `sxq Q1` $\theta_1 \rightarrow l_r$ for some substitution θ_1 such that $\theta_1(\$x_i) = t_i$, for $t_i = 1 \ldots k$, with $\$x_i \in Rel(Q', p \cdot 1)$.
- Consider the reduction $[\![Q_2]\!]_{k+1}(\mathcal{F}', t_1, \ldots, t_k, l_r) :=^* (\mathcal{F}_r, L_r)$. In this case $Rel(Q', p \cdot 2)$ be the set of variables of Q' that are relevant for Q_2. Then, by Lemma 3, $Rel(Q', p \cdot 2) = Rel(Q', p) \cup \{\$x_{k+1}\}$.
  By induction hypothesis, for all $t \in L_r$, there is a CRWL-proof proving $\mathcal{P} \vdash$ `sxq Q2`$\theta_2 \rightarrow$ `t` for some θ_2 such that $\theta_2(\$x_j) = \theta_1(\$x_j) = t_j$, $1 \leq j \leq k$ and $\theta_2(\$x_{k+1}) = l_r$. Then we can define $\theta = \theta_1 \cup \theta_2$ without ambiguity, because $dom(\theta_1) \cap dom(\theta_2) = Rel(Q', p)$ and $\theta_2(\$x) = \theta_1(\$x)$ for every $\$x \in Rel(Q', p)$.
  Now we can use a variant of the the fifth rule of `sxq` (see Figure 5) such as `sxq (xfor X Q1' Q2') = sxq Q2' <== X== sxq Q1'`, and a substitution $\sigma = \{\texttt{X} \mapsto \texttt{X}_{k+1}, \texttt{Q1}' \mapsto \texttt{Q1}, \texttt{Q2}' \mapsto \texttt{Q2}\} \cdot \theta$, and build a CRWL-proof for $\mathcal{P} \vdash$ (`sxq (xfor` $\texttt{X}_{k+1}$ `Q1 Q2)`)$\theta \rightarrow$ `t` starting with a (FA) inference of the form:

$$\frac{\begin{array}{l}(\texttt{xfor X}_{k+1}\ \texttt{Q1 Q2})\theta \rightarrow (\texttt{xfor X Q1' Q2'})\sigma \\ \texttt{sxq Q1'}\sigma\ ==\ \texttt{X}\sigma \\ (\texttt{sxq Q2'})\sigma \rightarrow\ \texttt{t}\end{array}}{(\texttt{sxq (xfor X}_{k+1}\ \texttt{Q1 Q2}))\theta\ \rightarrow\ \texttt{t}}$$

which can be rewritten as

$$\frac{\begin{array}{ll}(1) & (\texttt{xfor X}_{k+1}\ \texttt{Q1 Q2})\theta_2 \rightarrow (\texttt{xfor X}_{k+1}\ \texttt{Q1 Q2})\theta_2 \\ (2) & \texttt{sxq Q1}\theta_1\ ==\ \texttt{X}_{k+1}\theta_2 \\ (3) & \texttt{sxq Q2}\theta_2 \rightarrow\ \texttt{t}\end{array}}{(\texttt{sxq (xfor X}_{k+1}\texttt{Q1 Q2))}\theta_2\ \rightarrow\ \texttt{t}}$$

taking into account the definition of σ and θ.

Now we check that the three premises can be proven in CRWL.

1. $\mathcal{P} \vdash (\texttt{xfor}\ X_{k+1}\ Q_1\ Q_2)\theta_2 \rightarrow (\texttt{xfor}\ X_{k+1}\ Q_1\ Q_2)\theta_2$. Holds by Lemma 4.
2. $\mathcal{P} \vdash \texttt{sxq}\ Q_1\theta_1\ ==\ X_{k+1}\theta_2$. Considering that $X_{k+1}\theta_2 = l_r$, and by Lemma 4, we must find a proof for $\mathcal{P} \vdash \texttt{sxq}\ Q_1\theta_1\ \rightarrow\ l_r$, and such proof exists by induction hypothesis.
3. $\mathcal{P} \vdash \texttt{sxq}Q_2\theta_2 \rightarrow\ \texttt{t}$. Holds by induction hypothesis.

Observe that in fact θ_2 contains also $\$x_{k+1}$ in its domain, with $\$x_{k+1} \notin Rel(Q', p)$, but it still verifies the requirements of the Lemma, because $\theta_2(\$x_i) = t_i$ for $\$x_i \in Rel(Q', p)$.

– $Q \equiv \texttt{if}\ Q_1\texttt{then}\ Q_2$. In $\mathcal{TOY}$: `xif (cond Q1 ) Q2` .

In this case, by $\mathbf{XQ_5}$,

$$[\![\text{if } Q_1 \text{ then } Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := \begin{array}{l}\text{if } \pi_2([\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k)) \neq [\,] \\ \quad \text{then } [\![Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) \\ \quad \text{else } (\mathcal{F}, [\,])\end{array}$$

We distinguish two cases:

- $[\![\text{if } Q_1 \text{ then } Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}, [\,])$
  In this case, $[\![\text{if } Q_1 \text{ then } Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) = (\mathcal{F}, [\,])$. Therefore, the result trivially holds.
- $[\![\text{if } Q_1 \text{ then } Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := [\![Q_2]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L')$.
  In this case, the condition Q_1 returns some result, that is, $[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}'', L'')$. Let t be any value in L'. Then we prove that $\mathcal{P} \vdash (\texttt{sxq Q})\theta \rightarrow \texttt{t}$ for some θ with $\theta(\$x_i) = t_i$ for every $1 \leq i \leq k$. By Lemma 3, $Rel(Q', p) = Rel(Q', p \cdot 1)$ and $Rel(Q', p) = Rel(Q', p \cdot 2)$.
  Hence the induction hypothesis can be applied to both Q_1 and Q_2.
  - For $t' \in L''$, there ia a substitution θ_1 such that $\theta_1(\$x_i) = t_i$ for $\$x_i \in Rel(Q', p)$, and a CRWL proof proving $\mathcal{P} \vdash (\texttt{sxq Q1})\theta_1 \rightarrow \texttt{t}'$.
  - For $t \in L'$, there is a a substitution θ_2 such that $\theta_2(\$x_i) = t_i$ for $\$x_i \in Rel(Q', p)$, and a CRWL proof proving $\mathcal{P} \vdash (\texttt{sxq Q2})\theta_2 \rightarrow \texttt{t}$.

  Observe that we are assuming that each query introduces new variable names. Then we can define $\theta = \theta_1 \cup \theta_2$ without ambiguity. Now, applying a variant of the seventh rule of `sxq` (for instance `sxq (xif (cond Q1') Q2') = sxq Q2' <== sxq Q1' == A` , see Figure 5), and defining a substitution $\sigma = \{Q_1' \mapsto Q_1, Q_2' \mapsto Q_2, A \mapsto t'\} \cdot \theta$ (in fact A can be bound to any $t' \in L''$), we can build a CRWL proof for $\mathcal{P} \vdash (\texttt{sxq Q})\theta \rightarrow \texttt{t}$:

$$\frac{\begin{array}{l}(\texttt{xif (cond Q1) Q2})\theta \rightarrow (\texttt{xif (cond Q1') Q2'})\sigma \\ (\texttt{sxq Q1'})\sigma == A\sigma \\ (\texttt{sxq Q2'})\sigma \rightarrow \texttt{t}\end{array}}{(\texttt{sxq xif cond Q1 Q2})\theta \rightarrow\ \texttt{t}}$$

Taking into account the definition of σ the previous (FA) step can be rewritten as:

$$\frac{\begin{array}{l}(\texttt{xif (cond Q1) Q2})\theta \rightarrow (\texttt{xif (cond Q1) Q2})\theta \\ \texttt{sxq Q1}\theta_1 == \texttt{t}' \\ \texttt{sxq Q2}\theta_2 \rightarrow \texttt{t}\end{array}}{(\texttt{sxq xif (cond Q1) Q2 })\theta \rightarrow \texttt{t}}$$

In the first premise we have the same term at left-hand side and at right-hand side, and the existence of the proof is ensured by Lemma 4. The same Lemma indicates that proving $\texttt{sxq Q1}\theta_1 == \texttt{t}'$ is equivalent to proving $\texttt{sxq Q1}\theta_1 \rightarrow \texttt{t}'$, and we have seen that it holds by induction hypothesis. The same happens with the third premise.

– $Q \equiv \texttt{if } (\$x_i := \$x_j)\texttt{then } Q_1$.
In $\mathcal{TOY}$: `xif (xp (var X_i) := xp (var X_j)) Q1`.

In this case, by $\mathbf{XQ_6}$,

$$[\![\text{if } (\$x_i := \$x_j) \text{ then } Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := \begin{cases} [\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) & \text{if } t_i = t_j \\ (\mathcal{F}, [\,]) & \text{e.o.c} \end{cases}$$

If $t_i \neq t_j$ the result holds. Thus we assume that $t_i = t_j$, and that:

$$[\![Q_1]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L')$$

Let t be any element of L'. We must prove that $\mathcal{P} \vdash \texttt{sxq Q}\theta \rightarrow \texttt{t}$ for some substitution θ such that $\theta(\$x_i) = t_i$ for every $\$x_i \in Rel(Q', p)$. In first place it is possible to check that $\mathcal{P} \vdash (\texttt{sxq (xp (var X}_i\texttt{))} == \texttt{sxq (xp (var X}_j\texttt{)))}\theta$ with θ such that $\theta(\$x_i) = t_i$, $\theta(\$x_j) = t_j$.

$$\frac{\dfrac{(\texttt{X}_i)\theta \rightarrow \texttt{t}'}{(\texttt{sxqPath (var X}_i\theta\texttt{))} \rightarrow \texttt{t}'}}{\texttt{sxq (xp (var X}_i\theta\texttt{))} \rightarrow \texttt{t}'} \qquad \frac{\dfrac{(\texttt{X}_j)\theta \rightarrow \texttt{t}'}{(\texttt{sxqPath (var X}_j\theta\texttt{))} \rightarrow \texttt{t}'}}{(\texttt{sxq (xp (var X}_j\theta\texttt{))} \rightarrow \texttt{t}'}}{\texttt{sxq (xp (var X}_i\theta\texttt{))} == \texttt{sxq (xp (var X}_j\theta\texttt{))}}$$

With $t' \equiv t_i \equiv t_j$, using the inference rules (JN), (FA) using the first rule of `sxq`, (FA) using the first rule of `sxqPath` and finally proving the premises on top applying the Lemma 4.
By Lemma 3, $Rel(Q', p{\cdot}1) = Rel(Q', p)$, and the induction hypothesis can be applied to Q_1 as in the previous case. Now, applying the sixth rule of `sxq` (`sxq (xif (Q2 := Q3) Q1) = sxq Q1 <== sxq Q2 == sxq Q3`, see Figure 5), it is possible to find a CRWL proof for $\mathcal{P} \vdash (\texttt{sxq Q})\theta \rightarrow \texttt{t}$ (the details are similar to the previous cases).

– $Q \equiv \$x_i$.
In this case, $[\![\$x_i]\!]_k(\mathcal{F}, t_1, \ldots, t_k) := (\mathcal{F}, [t_i])$ by $\mathbf{XQ_3}$. The query Q is such that $Q'_{|p} \equiv Q$ for $p \in Pos(Q')$. Then, $\$x_i \in Rel(Q', p) = \{\$x_1, \ldots, \$x_k\}$.

Then $\$x_i \in Rel(Q, p)$. θ must be a substitution such that $(\$x_i)\theta = t_i$. The representation in $\mathcal{TOY}$ of $\$x_i$ is `sxqPath (var Xi)`, see Figure 5), and applying the first rule of `sxqPath`, (`sxqPath (var X) = X`, see Figure 5), we can build a CRWL proof for $\mathcal{P} \vdash$ (`sxq Q`)$\theta \to t_i$ with instance $\sigma = \{X \to t_i\}$.

$$\frac{(\texttt{varX}_i\theta) \to (\texttt{varX}\sigma) \qquad (\texttt{X}_i)\sigma \to\ t_i}{\texttt{sxqPath}\ (\texttt{varX}_i)\theta \to\ t_i}$$

Applying the definition of θ and σ in the premises we have:

$$\frac{(\texttt{varX}_i\theta) \to (\texttt{varX}\sigma) \qquad (\texttt{X}_i)\sigma \to\ t_i}{\texttt{sxqPath}\ (\texttt{varX}_i)\theta \to\ t_i}$$

and all the premises are consequence of Lemma 4

– $Q \equiv \$x_i/axis :: \nu$. The proof in this case is very similar to the corresponding case in Lemma 5 which can be in fact read as an *if and only if* proof.

As in the case of correctness, the completeness theorem is just a particular case of the Lemma:

Theorem 2. *Let P be the $\mathcal{TOY}$ program of Figure 5. Let Q be a SXQ query with $free(Q) = \{\$x_1, \ldots, \$x_k\}$ and suppose that $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L)$ for some $\mathcal{F}$, $\mathcal{F}', t_1, \ldots, t_k, L$. Then, for every $t \in L$, there is a substitution θ such that $\theta(\$x_i) = t_i$ for all $\$x_i \in free(Q)$ and a CRWL-proof proving $\mathcal{P} \vdash sxq\ Q\theta == $ `t`.*

Proof. In Lemma 6, consider $p \equiv \varepsilon$ and thus $Q' \equiv Q$. Then $Rel(Q, \varepsilon) = free(Q) \cup V_{for}(Q, \varepsilon) = free(Q)$. Then the conclusion of the lemma is the same as the conclusion of the Theorem.

5 Application: Test Case Generation

In this chapter we show how the embedding of SXQ in $\mathcal{TOY}$ can be used for obtaining test-cases for the queries. For instance, consider the erroneous query of the next example.

Example 8. Suppose that the user also wants to include the publisher of the book among the data obtained in Example 1. The following query tries to obtain this information:

```
Q = for $b in doc("bib.xml")/bib/book,
        $r in doc("reviews.xml")/reviews/entry,
        where $b/title = $r/title
        for $booktitle in $r/title,
            $revtext in $r/review,
            $publisher in $r/publisher
        return  <rev> $booktitle $publisher $revtext </rev>
```

However, there is an error in this query, because in the expression `$r/publisher` the variable `$r` should be `$b`, since the publisher is in the document "bib.xml", not in "reviews.xml". The user does not notice that there is an error, tries the query (in $\mathcal{TOY}$ or in any XQuery interpreter) and receives an empty answer.

In order to check whether a query is erroneous, or even to help finding the error, it is sometimes useful to have test-cases, i.e., XML files which can produce some answer for the query. Then the test-cases and the original XML documents can be compared, and this can help finding the error. In our setting, such test-cases are obtained for free, thanks to the generate and test capabilities of logic programming. The general process can be described as follows:

1. Let `Q'` be the translation of the SXQ query Q as a $\mathcal{TOY}$ dataterm by means the primitive `parse_xquery`.
2. Let $F_1, \ldots, F_k$ be the names of the XML documents occurring in `Q'`.
3. Let `Q"` be the result of replacing each expression of the form `doc(`F_i`)` by a new variable D_i, for $i = 1 \ldots k$.
4. Let `"expected.xml"` be a document containing an expected answer for the query `Q`.
5. Try the following goal:

```
Toy> sxq Q'' == load_doc "expected.xml",
                write_xml_file D1 F1',
                ... ,
                write_xml_file Dk Fk'
```

The idea is that the goal above looks for values of the logic variables D_i fulfilling the strict equality. The result is that after solving this goal, the D_i variables contain XML documents that can produce the expected answer for this query. Then each document is saved into a new file with name F_i'. For instance F_i' can consist of the original name F_i preceded by some suitable prefix *tc*. The process can be automatized, and the result is the code of Figure 6.

The code uses the list concatenation operator `++` which is defined in $\mathcal{TOY}$ as usual in functional languages such as Haskell. It is worth observing that if there are no test-case documents that can produce the expected result for the query, the call to `generateTC` will loop. The next example shows the generation of test-cases for the wrong query of Example 8.

Example 9. Consider the query of Example 8 , and suppose the user writes the following document "expected.xml":

```
<rev>
 <title>Some title</title>
 <review>The review</review>
 <publisher>Publisher</publisher>
</rev>
```

This is a possible expected answer for the query. Now we can try the goal:

```
prepareTC (xp E)            = (xp E',L)
                              where (E',L) = prepareTCPath E
prepareTC (xmlExp X)        = (xmlExp X, [])
prepareTC (comp Q1 Q2)      = (comp Q1' Q2', L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
prepareTC (xfor X Q1 Q2)    = (xfor X Q1' Q2', L1++L2)
                              where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2
prepareTC (xif (Q1:=Q2) Q3) = (xif (Q1':=Q2') Q3',L1++(L2++L3))
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
                                     (Q3',L3) = prepareTC Q3
prepareTC (xif (cond Q1) Q2) = (xif (cond Q1) Q2, L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                    (Q2',L2) = prepareTC Q2

prepareTCPath (var X)   = (var X, [])
prepareTCPath (X :/ S)  = (X :/ S, [])
prepareTCPath (doc F S) = (A :/ S, [write_xml_file A ("tc"++F)])

generateTC Q F = true <==  sxq Qtc == load_doc F, L==_
              where (Qtc,L) = prepareTC Q
```

Fig. 6. $\mathcal{TOY}$ test case generation rules for SXQ

```
Toy> Q == parse_xquery "for....", R == generateTC Q "expected.xml"
```

The first strict equality parses the query, and the second one generates the XML documents which constitute the test cases. In this example the test-cases obtained are:

```
% bibtc.xml
<bib>
 <book>
  <title>Some title</title>
 </book>
</bib>
```

```
% revtc.xml
<reviews>
 <entry>
  <title>Some title</title>
  <review>The review </review>
  <publisher>Publisher</publisher>
 </entry>
</reviews>
```

By comparing the test-case "revtc.xml" with the file "reviews.xml" (see Appendix A) we observe that the publisher is not part of the structure defined for reviews. Then, it is easy to check that in the query the publisher is obtained from the reviews instead of from the *bib* document, and that this constitutes the error.

6 Conclusions

The report shows the embedding of a fragment of the XQuery language for querying XML documents in the functional-logic language $\mathcal{TOY}$. Although only a small subset of XQuery consisting only of *for*, *where/if* and *return* statements has been considered, the users of $\mathcal{TOY}$ can now perform simple queries typical of database queries such as *join* operations. The embedding has respected the declarative nature of $\mathcal{TOY}$, and we have provided the soundness of the approach with respect to the operational semantics of XQuery. From the point of view of XQuery the results are also encouraging. The embedding allows the user to generate test-cases automatically when possible, which is useful for testing the query, or even for helping to find the error in the query.

The most obvious future work would be introducing *let* statements, which presents two novelties. The first is that they are *lazy*, that is, they are not evaluated if they are not required by the result. This part is easy to fulfill since we are in a lazy language. In particular, they could be introduced as local definitions (*where* statements in $\mathcal{TOY}$).

The second novelty is more difficult to capture, and it is that the variables introduced by *let* represent an XML sequence. The natural representation in $\mathcal{TOY}$ would be a list, but the non-deterministic nature of our proposal does not allow us to collect all the results provided by an expression in a declarative way. A possible idea would be to use the functional-logic Curry [10] and its encapsulated-search [12], or even the non-declarative *collect* primitive included in $\mathcal{TOY}$. In any case, this will imply a different theoretical framework and new proofs for the results. A different line for future work is the use of test cases for finding the error in the query using some variation of declarative debugging [18] that would be applied to this setting.

References

1. J. Almendros-Jiménez. An Encoding of XQuery in Prolog. In Z. Bellahsène, E. Hunt, M. Rys, and R. Unland, editors, *Database and XML Technologies*, volume 5679 of *Lecture Notes in Computer Science*, pages 145–155. Springer Berlin / Heidelberg, 2009.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1999.
3. M. Benedikt and C. Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34:25:1–25:48, December 2009.
4. R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language $\mathcal{TOY}$. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, volume 6539 of *PADL'11*, pages 145–159, Berlin, Heidelberg, 2011. Springer-Verlag.
5. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language, 2010. `http://www.w3.org/TR/xquery/`.
6. J. Clark. XML Path Language (XPath) 2.0, 2010. `http://www.w3.org/TR/xpath20/`.

7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).
9. L. Fegaras. Propagating updates through XML views using lineage tracing. In *IEEE 26th International Conference on Data Engineering (ICDE)*, pages 309 – 320, march 2010.
10. M. Hanus. Curry: An Integrated Functional Logic Language (version 0.8.2 march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.
11. M. Hanus. Declarative processing of semistructured web data. In J. Gallagher and M. Gelfond, editors, *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 198–208, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
12. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALPŠ98*, pages 374–390. Springer LNCS, 1998.
13. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
14. F. J. López-Fraguas and J. S. Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *RTA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 244–247, London, UK, 1999. Springer-Verlag.
15. J. C. G. Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.
16. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management*, pages 109–127. Springer, 2002.
17. D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *In Proc. Int. Conf. on Applications of Declarative Programming and Knowledge Management*, pages 140–151. Springer, 2004.
18. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
19. W3C. Extensible Markup Language (XML), 2007.
20. W3C. XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition), 2010. `http://www.w3.org/TR/xquery-semantics/`.
21. P. Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

A Examples of XML documents

```
% bib.xml

<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

```
% reviews.xml

<reviews>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
               A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
    <entry>
        <title>TCP/IP Illustrated</title>
        <price>65.95</price>
        <review>
               One of the best books on TCP/IP.
        </review>
    </entry>
</reviews>
```

C | Summary

The purpose of this thesis is to design and develop techniques for detecting and diagnosing errors in the field of databases, and in particular in the case of queries to databases. In order to help in the task of detecting errors, we develop techniques for automatically generating test cases. These test cases are considered as valid database instances which allow to the user easily checking the correctness of the query results. We propose to apply techniques related to declarative debugging for diagnosing errors. These techniques are based on the exploration of a suitable structure that represent the erroneous computation. This structure contains information about the final result and also the information about all intermediate results. For locating the cause of the error, the debugger asks to an oracle about the expected results.

In the field of databases, we have focused on deductive databases, relational databases and semistructured databases. The main contributions of this thesis can be summarized as follows:

In Section C.1 we apply declarative debugging to Datalog programs. The debugger detects incorrect fragments of code starting from an unexpected answer. During the theoretical study of Datalog, its semantics and its computation mechanism, we have found that the traditional errors considered usually in logic programming are not enough in the case of Datalog where a new kind of error, the incomplete sets of predicates, can occur. Due to the set-oriented nature of Datalog computations, the declarative debugging schema used traditionally in logic programs is not appropriate for Datalog Programs. Therefore, we define a new instance based on a suitable structure for representing the computation mechanism of Datalog. This is a novelty w.r.t. others works related to declarative debugging of logic programs. In particular we have found that recursive computations (and thus computations associated to incomplete sets of predicates) are not easily represented by computations trees, the structure employed usually in declarative debugging. Thus, we propose to use graphs for representing the computations. In our proposed computation graphs incomplete set of predicates are represented naturally, given raise to *buggy circuits*.

The theoretical ideas propose solid foundations for the debugging of Datalog programs and have been set in practice by developing a declarative debugger for the Datalog system DES. The debugger allows diagnosing both missing and wrong answers, which constitute all the possible errors symptoms of a Datalog program.

In Section C.2 we discuss the problem of checking correlated SQL queries. Firstly we present a general framework for generating SQL query test cases using Constraint

Logic Programming. Given a database schema and a SQL view defined in terms of other views and schema tables, our technique generates automatically a set of finite domain constraints whose solutions constitute the test database instances. We have formally defined the algorithm for producing the constraints, and we have proven the soundness and correctness of the technique w.r.t. the semantics of Extended Relational Algebra. Regarding the coverage criteria for testing SQL queries, we have considered a simple criterion namely the *predicate coverage criterium*. A prototype of generator has been implemented in an available tool covering a wide range of SQL queries, including views, subqueries, aggregates and set operations.

Secondly, we propose using algorithmic debugging for finding errors in systems involving several SQL views and we present two techniques: The first one is based on the navigation of a suitable computation tree corresponding to some view returning some unexpected result. This tree contains the computed answer associated with every intermediate relation, asking the user whether this answer is expected or not. The debugger ends when a buggy node, i.e., a nonvalid node with valid children, is found. We prove formally that every buggy node corresponds to an erroneous relation, and that every computation tree with a nonvalid root contains some buggy node. The results are established in the context of the Extended Relational Algebra. The main criticism that can be leveled at this proposal is that it can be difficult for the user to check the validity of the intermediate results. Indeed, the results returned by SQL views can contain hundreds or thousands of tuples.

The second technique for debugging SQL views refines the first one by taking into account a more detailed information; the process allows the user to provide information about the type of the error and the debugger is guided by this information. Using a technique similar to dynamic slicing, we concentrate only in those tuples produced by the intermediate relations that are relevant for the error. The proposed technique does not use computation trees explicitly. Instead, the logical relations among the nodes of the tree are represented by logic clauses that also contain the information extracted from the specific questions provided by the user. We use a dynamic Prolog program in order to represent both the computation tree and the user information. The atoms in the body of the clauses correspond to questions that the user must answer in order to detect an incorrect relation. The resulting logic program is executed by selecting at each step the unsolved atom that yields the simplest question, repeating the process until an erroneous relation is detected. We have proven the correctness and the completeness of the proposal.

Both proposals have been implemented in a working prototype included in the Datalog system DES.

In Section C.3 we show a framework that allows the user testing XPath and XQuery queries. In this case, we have chosen to represent both languages in the funcional-logic language $\mathcal{TOY}$. The effort for embedding XPath and XQuery in $\mathcal{TOY}$ is rewarded due to the simplicity for generating test cases automatically when possible, which is useful for testing the query, or even for helping to find the error in the query. In the case of XPath, queries are represented in this setting by non-deterministic higher-order expressions, thus becoming first-class citizens of the language that can be readily extended and adapted by the programmer. The possibility of including *higher-order* patterns in Toy allow us to consider expressions that are

not yet reducible as patterns. This means in our case that XPath expressions can be considered at the same time executable code (when applied to an input XML document), or data structures when considered as higher-order patterns. This powerful characteristic of the language is heavily used in our proposals for debugging wrong and missing answers. The use of *non-deterministic* functions in Toy allow us to define easily the XPath framework, and are also useful for tracing wrong answers, where only those parts of the computation that produce a particular fragment of the answer are required. The use of *logic variables*, specially when used in *generate and test* expressions are very suitable for obtaining the values of intermediate computations, and in our case also for guessing values in the debugging of missing answers.

In the case of XQuery, although only a small subset of XQuery consisting of *for*, *where/if* and *return* statements has been considered, the users of $\mathcal{TOY}$ can now perform simple queries typical of database queries such as *join* operations. The embedding has respected the declarative nature of $\mathcal{TOY}$, and we have provided the soundness of the approach with respect to the operational semantics of XQuery.

C.1. Deductive databases

The declarative programming paradigm is targeted to raise the semantic level of programs, therefore isolating them from the computation model. Thus, programmers are intended to focus on a higher semantic level rather than on the level corresponding to the underlying computation procedures.

Deductive database languages such as Datalog [96], which inherit the declarative nature of the Logic Programming language Prolog [108], increase the gap between the program semantics and the computations because the computation model of Datalog is much more intricate than that of Prolog. Prolog computation model is based on the SLD resolution principle [83], which deals with SLD computation trees, whereas Datalog computation model is based on a number of proposals, ranging from interpreters [115] to compilation to Prolog using magic sets [20]. This semantic gap between program semantics and program execution makes debugging Datalog programs a hard task if one tries to use existing tools for debugging in a quite different level the user thinks about (for instance, using a trace debugger in the level of the transformed program).

Our approach to debug Datalog programs is anchored to the semantic level, which is a natural requirement every user imposes to development systems. In our setting, we deal with the set of values for a query as a single entity, therefore reducing the complexity of the debugging task. We propose a novel way of applying *declarative debugging*, also called Algorithmic Debugging [105] to Datalog programs, allowing to debug queries and diagnose *missing* (an expected tuple is not computed) as well as *wrong* (a given computed tuple should not be computed) answers with the same tool. Although the general schema of declarative debugging is valid for any language, it results particularly useful when applied to declarative languages [30, 63, 92] where the computations are difficult to debug using traditional approaches such as the program step-by-step computation.

What a Datalog programmer would find useful is to catch program rules or rela-

tions which are responsible for a mismatch between the intended semantics of a query and its actual computed semantics. Our system, by means of a question-answering procedure which starts when the user detects an unexpected answer for some query, looks for those errors pointing to the program fragment responsible for the incorrectness. For this procedure, we propose to use *computation graphs* as a novel data structure for declarative debugging of Datalog programs. We find that these graphs are more convenient for modeling program computations, instead of computation trees, which have been typically used in declarative debuggers for other languages (e.g., Prolog [105], Java [39] and $\mathcal{T}oy$ [30]).

Next Section introduces basic information about the syntax and semantics for Datalog Programs and defines the different types of errors that are discussed in our proposal.

C.1.1. Datalog Programs

Definitions for Datalog mainly come from the field of Logic Programming. We consider (recursive) Datalog programs with stratified negation [14, 115], i.e., normal logic programs without function symbols. Stratification is imposed to ensure a clear semantics when negation is involved, and function symbols are not allowed in order to guarantee termination of computations, a natural requirement with respect to a database user.

A *term* is either a variable or a constant symbol. An *atom* is $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and t_i are terms, $1 \leq i \leq n$, which can also be written as $p(\bar{t}_n)$. A *literal* is either an atom or a negated atom. A *positive* literal is an atom, and a *negative* literal is a negated atom. A negated atom is syntactically constructed as $not(A)$, where A is an atom. The atom contained in a literal L will be denoted as *atom(L)*. A *rule* R is an expression of the form $A :- L_1, \ldots, L_n$, where A is an atom and L_i are literals. All of the variables in a rule are assumed to be universally quantified. Concerning the rule R, A is referred to as the *head* of R, $L_1, \ldots, L_n$ as its *body*, and L_i as *subgoals*. Commas in bodies stand for conjunctions. A *fact* is a rule with empty body and *ground* head. The symbol $:-$ is usually dropped in this case. A *Datalog program* is a finite set of Datalog facts and rules. In order to fit with database notation, the term *relation* is used *in lieu* of *predicate*. A database *relation* is, therefore, a set of rules with the same predicate symbol and arity. A *query* (term preferred in a deductive database context) or *goal* (term preferred in a logic programming context) is a literal (i.e., an atom or a negated atom) which can be solved by a Datalog system with respect to a given program. Analogous to literals, we say that a *positive* query is an atom, and a *negative* query is a negated atom. In contrast to facts, queries may contain variables.

Datalog programs resemble Prolog programs as the program in Figure C.1 suggests. Our setting does not enforces the use of safe programs. Only the stratification requirement is needed for the correctness of the debugging technique.

We consider *Herbrand interpretations* and *Herbrand models* [48], i.e., Herbrand interpretations that make every Herbrand instance of the program rules logically true formulae.

```
star(sun).
orbits(earth, sun).
orbits(moon, earth).
orbits(X,Y):- orbits(X,Z), orbits(Z,Y).
intermediate(X,Y) :- orbits(X,Y), orbits(Z,Y).
planet(X)         :- orbits(X,Y), star(Y),
                     not(intermediate(X,Y)).
```

Figure C.1: A (Buggy) Datalog Program.

An *instance* of a formula is the result of applying the substitution θ to a formula F. We use the notation $F\theta$ for representing instances. The set $Subst$ represents the set of all the possible substitutions. Often, we will be interested in *ground instances of a rule*, assuming implicitly that every rule is renamed with new variables each time it is selected.

Given a Herbrand interpretation I for a the Datalog program P, we use the notation $I \models F$ to indicate that the formula F is true in I. The *meaning of a query* Q w.r.t. the interpretation I, denoted by Q_I, is the set of ground instances $Q\theta$ s.t. $I \models Q\theta$. That is:

$$Q_I = \{Q\theta \mid Q\theta \in I \text{ for some } \theta \in Subst\}$$

In logic programming without negation, the existence of a *least Herbrand model* for every program P is ensured. In general, however, the existence of the least Herbrand model is not ensured in programs using negation. Fortunately, in the case of Datalog the existence of a so-called *standard model*, which we will represent also as $\mathcal{M}$, is in any case ensured [14]. The concept of standard model is generalized by that of *stable model* [67], which can be applied also to non-stratified programs.

Since functions are not allowed in Datalog, the standard model is finite and it can be actually computed. In fact, the deductive database systems are implemented to obtain the values $Q_\mathcal{M}$ for every query Q. Thus, $Q_\mathcal{M}$ will be referred to as the *answer* to Q. From now on, we assume that the Datalog system verifies this condition, which is a reasonable requirement in the context of Datalog. This is different from the general setting of logic languages such as Prolog, even if we restrict to the case of Prolog programs without functions in the signature.

Additionally, we use the term *intended interpretation*, denoted by $\mathcal{I}$, to denote the Herbrand model the user has in mind for the program. If $\mathcal{M} = \mathcal{I}$, we say that the program is *well-defined*, and if $\mathcal{M} \neq \mathcal{I}$ we say that the program is *buggy*. Declarative debugging assumes that the user focus on query answers for comparing the intended interpretation to the standard Herbrand model actually computed. Thus, we say that $Q_\mathcal{M}$ is an *unexpected answer* for a query Q if $Q_\mathcal{M} \neq Q_\mathcal{I}$. An unexpected answer can be either a *wrong answer*, when there is some $Q\theta \in Q_\mathcal{M}$ s.t. $Q\theta \notin Q_\mathcal{I}$, or a *missing answer*, when there is $Q\theta \in Q_\mathcal{I}$ s.t. $Q\theta \notin Q_\mathcal{M}$. In the first case, $Q\theta$ is a *wrong instance*, while in the second one $Q\theta$ is a *missing instance*. Observe that an unexpected answer can be both missing and wrong at the same time. The next proposition indicates

that an unexpected answer to a positive query implies an unexpected answer to its negation.

Proposition C.1.1. *Let P be a program containing at least one constant, $\mathcal{I}$ its intended model and Q a positive query. Then, $Q_{\mathcal{M}}$ is a missing answer for Q iff $(\neg Q)_{\mathcal{M}}$ is a wrong answer for $\neg Q$, and $Q_{\mathcal{M}}$ is a wrong answer for Q iff $(\neg Q)_{\mathcal{M}}$ is a missing answer for $\neg Q$.*

An unexpected answer indicates that the program is erroneous, and it will be considered as the initial symptom for a user to start the debugging process. The two usual causes of errors considered in the declarative debugging of logic programs are *wrong* and *incomplete* relations:

Definition C.1.2. *Let P be a Datalog program. We say that $p \in P$ is a **wrong relation** w.r.t. $\mathcal{I}$ if there exist a rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and a substitution θ such that $\mathcal{I} \models l_i\theta$, $i = 1 \ldots m$ and $\mathcal{I} \not\models p(\bar{t}_n)\theta$.*

Definition C.1.3. *Let P be a Datalog program. We say that $p \in P$ is an **incomplete relation** w.r.t. $\mathcal{I}$ if there exists an atom $p(\bar{s}_n)\theta$ s.t. $\mathcal{I} \models p(\bar{s}_n)\theta$ and, for each rule variant $p(\bar{t}_n) :- l_1, \ldots, l_m$ and substitution θ', either $p(\bar{t}_n)\theta' \neq p(\bar{s}_n)\theta$ or $\mathcal{I} \not\models l_i\theta'$ for some l_i, $1 \leq i \leq m$.*

In Datalog we also need to consider another possible cause of errors, namely the *incomplete set of relations*. This concept depends on the auxiliary definition of uncovered set of atoms.

Definition C.1.4. *Let P be a Datalog program and $\mathcal{I}$ an intended interpretation for P. Let U be a set of atoms s.t. $\mathcal{I} \models p(\bar{s}_n)$ for each $p(\bar{s}_n) \in U$. We say that U is an **uncovered set of atoms** if for every rule $p(\bar{t}_n) :- l_1, \ldots, l_m$ in P and substitution θ s.t.:*

- *$p(\bar{t}_n)\theta \in U$,*
- *$\mathcal{I} \models l_i\theta$ for $i = 1 \ldots m$*

there is some $l_j\theta \in U$, $1 \leq j \leq m$, with l_j a positive literal.

Now, we are ready for defining the third kind of error, which generalizes the idea of incomplete relation:

Definition C.1.5. *Let P be a Datalog program and S a set of relations defined in P. We say that S is an **incomplete set of relations** in P iff exists an uncovered set of atoms U s.t. for each relation $p \in S$, $p(\bar{t}_n) \in U$ for some $t_1, \ldots, t_n$.*

To the best of our knowledge, this error has not been considered in the literature about Datalog debugging so far, but it is necessary for correctly diagnosing Datalog programs. Consider the program `p(X):- q(X). q(X):-p(X).` with the intended interpretation $I = \{p(a), q(a)\}$ and the query `p(X)`. The computed answer $\{\}$ is a missing answer with `p(a)` as missing instance. However, neither of the two relations is incomplete, because their rules can produce the values `p(a)`, `q(a)` by means of the

instance given by the substitution $\theta = \{X \mapsto a\}$. So, $U = \{p(a), q(a)\}$ is an uncovered set of atoms and hence $S = \{p, q\}$ is an incomplete set of relations.

We say that a relation is *buggy* when it is wrong, incomplete or member of an incomplete set of relations, and that it is well-defined otherwise. Observe that, due to the use of negation, a wrong answer does not correspond always to a wrong relation. For instance, in the following program:

```
p(X) :-  r(X), not(q(X)).
r(a).
```

with intended interpretation $\mathcal{I} = \{q(a), r(a)\}$ the query `p(X)` produces the wrong answer `{p(a)}` but there is no wrong relation in the program and instead there is an incomplete relation (q).

As an example, consider the program of Figure C.1. This program defines a relation `orbits` by two facts and a rule establishing the transitive closure of the relation.

A relation `star` is defined by one fact and indicates that the sun is a star. The relation `intermediate` is defined in terms of `orbits`, relating two bodies `X` and `Y` whenever there is some intermediate body between them. Finally, `planet` is defined as a body `X` that orbits directly a star `Y`, without any other body in between. However, a mistake has been introduced in the program: The underlined `Y` in the rule for `intermediate` should be `Z`. As a consequence, the query `planet(X)` yields the missing answer {} (assuming that the atom `planet(earth)` is in $\mathcal{I}$). In the next Section, we will show how such errors can be detected by using declarative debugging based on computation graphs.

C.1.2. Computation Graphs

In this Section, we define a suitable structure for representing Datalog computations. Usually in logic programming languages such as Prolog, the computations are represented through some tree structure such as the SLD-tree [83]. In the case of Datalog, we claim that a tree is not a convenient structure due to the different treatment of recursive programs.

For instance consider the program in Figure C.2. In Prolog, the SLD-tree for the goal `p(X)` will contain an infinite branch, representing a non-terminating computation. However in Datalog the same goal is terminating and returns the finite answer: `{p(a)}` because the computation mechanism detects the repetition of the subgoal `p(x)` and avoids the infinite loop. Thus, our computation structure must represent finitely these situations, which can be achieved by using a graph.

The graph in Figure C.2 contains the two subgoals occurred during the computation together with their respective answers. It also indicates that `p(X)` and `q(X)` are mutually dependent. We will call such graph the *computation graph* for the goal w.r.t. the program. Observe that this graph is different from the *predicate dependency graph* [123] of the Datalog program, which show the connections between the relations from a static point of view.

```
p(a).
r(b).
p(X) :- q(X),r(X).
q(X) :- p(X).
```

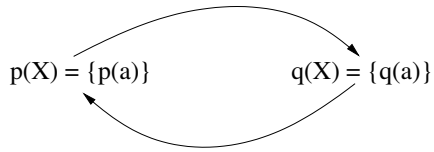


Figure C.2: Computation in Datalog for the goal `p(X)` w.r.t. the program in left.

The *computation graph* (*CG* in short) for a query Q w.r.t. a program P is a directed graph $G = (V, E)$ such that each vertex V is of the form $[Q' = Q'_{\mathcal{M}}]$, where Q' is a subquery produced during the computation, and $Q'_{\mathcal{M}}$ is the computed answer for Q'. The next definition includes the construction of a computation graph.

Definition C.1.6. *Let P be a Datalog program and Q a query either of the form $p(\bar{a}_n)$ or not$(p(\bar{a}_n))$. The computation graph for Q w.r.t. P is represented by a pair (V, E) of vertices and edges defined as follows:*

The construction of the graph uses an auxiliary set A for containing the vertices that must be expanded in order to complete the graph.

1. *Put $V = A = \{p(\bar{a}_n)\}$ and $E = \emptyset$.*

2. *While $A \neq \emptyset$ do:*

   a) *Select a vertex u in A with query $q(\bar{b}_n)$. $A = A \setminus \{u\}$.*

   b) *For each rule R defining q, $R = (q(\bar{t}_n) :- l_1, \ldots, l_m)$ with $m > 0$, such that there exists $\theta = mgu(\bar{t}_n, \bar{b}_n)$, the debugger creates a set S of new vertices. Initially, we define $S = \emptyset$ and include new vertices associated to each literal l_i, $i = 1 \ldots m$ as follows:*

      1) *$i = 1$, a new vertex is included: $S = S \cup \{atom(l_1)\theta\}$.*

      2) *$i > 1$. We consider the literal l_i. For each set of substitutions $\{\sigma_1, \ldots, \sigma_i\}$ with $dom(\sigma_1 \cdot \ldots \cdot \sigma_{i-1}) \subseteq var(l_1) \cup \cdots \cup var(l_i)$ such that for every $1 < j \leq i$:*

         - *$atom(l_{j-1})(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}) \in S$, and*
         - *$l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_j) \in (l_{j-1}(\sigma_1 \cdot \ldots \cdot \sigma_{j-1}))_{\mathcal{M}}$*

         include a new vertex in S:

$$S = S \cup \{atom(l_i)(\sigma_1 \cdot \ldots \cdot \sigma_i)\}$$

   c) *For each vertex $v \in S$, test whether there exists already a vertex $v' \in V$ such that v and v' are variants (i.e., there is a variable renaming). There are two possibilities:*

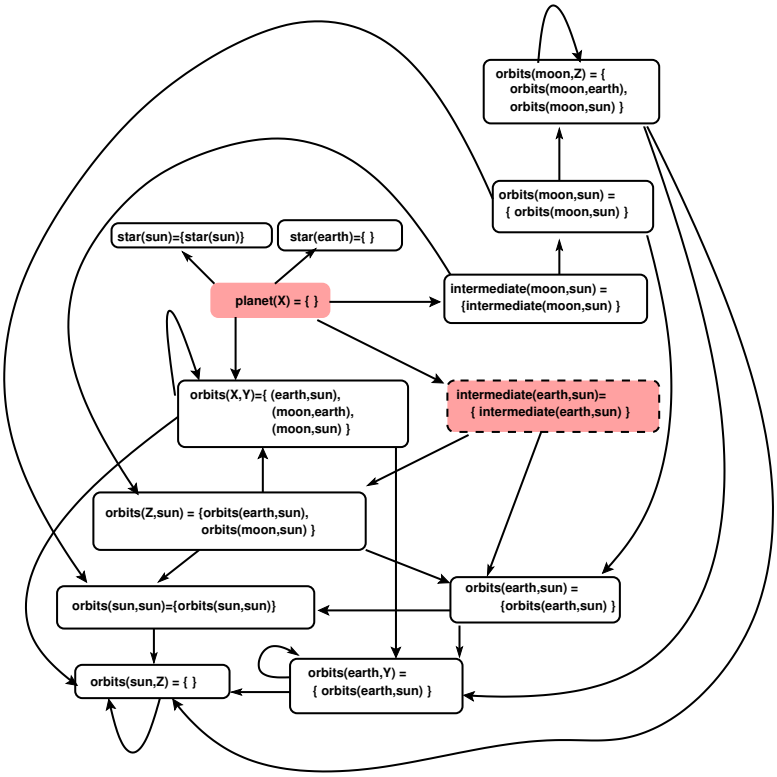


Figure C.3: CG for the Query `planet(X)` w.r.t. the Program of Figure C.1.

- *There is such a vertex v'. Then, $E = E \cup \{(u, v')\}$. That is, if the vertex already exists, we simply add a new edge from the selected vertex u to v'.*
- *Otherwise, $V = V \cup \{v\}$, $A = A \cup \{v\}$, and $E = E \cup \{(u, v)\}$.*

3. Complete the vertices including the computed answer $Q_{\mathcal{M}}$ of every subquery Q.

The values $Q_{\mathcal{M}}$ included at step 3 can be obtained from the underlying deductive database system by submitting each Q. The termination of the process is guaranteed because in our setting the signature is finite and the *CG* cannot have two occurrences of the same vertex due to step $2c$, which introduces edges between existing vertices instead of creating new ones when possible.

Figure C.3 shows the *CG* for the query `planet(X)` w.r.t. the program of Figure C.1. The first vertex included in the graph at step 1 corresponds to `planet(X)`. From this vertex and by using the only program rule for `planet`, four new vertices are added, the first one corresponding to the first literal `orbits(X,Y)`. Since two values

of `Y` satisfy this subquery, namely `Y=sun` and `Y=earth`, the definition introduces two new vertices for the next literal `star(Y)`, `star(sun)` and `star(earth)`. The last one produces the empty answer, but `star(sun)` succeeds. Then, the last literal in the rule, `not(intermediate(X,Y))`, yields vertices for the two values of `X` and the only value of `Y` that satisfies the two previous literals. Observe, however, that the vertices for this literal are introduced in the graph without the negation, i.e., the CG will contain only subqueries for atoms. This simplifies the questions asked to the user during the navigation phase, and can be done without affecting the correctness of the technique because the validity of the positive literal implies the validity of its negation, and the other way round (although the type of associated error changes, see Proposition C.1.1). The rest of the vertices of the example graph are built expanding the successors of `planet(X)` and repeating the process until no more vertices can be added.

C.1.3. Declarative Debugging with CG's

In the traditional declarative debugging scheme [91] based on trees, program errors correspond to *buggy nodes*. In our setting, we also need the concept of buggy node, here called *buggy vertex*, but in addition our computation graphs can include *buggy circuits*. Given a computation graph $CG = (V, A)$, we define a **buggy circuit** as a circuit $W = v_1 \dots v_n$ s.t. for all $1 \leq i \leq n$:

1. v_i is invalid.
2. If $(v_i, u) \in A$ and u is invalid then $u \in W$.

A **buggy vertex** is an invalid vertex but all its successors are valid. The next result states that a computation graph corresponding to an initial error symptom, i.e., including some invalid vertex, contains either a buggy circuit or a buggy vertex.

Proposition C.1.7. *Let G be a computation graph containing an invalid vertex. Then, G contains either a buggy vertex or a buggy circuit.*

The debugging process we propose can be summarized as follows: Firstly the user finds out an unexpected answer for some query Q w.r.t. some program P. The debugger builds the computation graph G for Q w.r.t. P. Then, the graph is traversed, asking questions to the user about the validity of some vertices until a buggy vertex or a buggy circuit has been found. If a buggy vertex is found, its associated relation is pointed out as buggy. If instead a buggy circuit is found, the set of relations involved in the circuit are shown to the user indicating that at least one of them is buggy or that the set is incomplete.

In the paper [32](A.2) we check that the technique is reliable. Observe that theoretically the debugger could be applied to any computation graph even if there is no initial wrong or missing answer. The following soundness and completeness results ensure that in any case it will behave correctly.

Proposition C.1.8 (Soundness)**.** *Let P be a Datalog program, Q be a query and G be the computation graph for Q w.r.t. P. Then:*

1. *Every buggy node in G is associated to a buggy relation.*

2. *Every buggy circuit in G contains either a vertex with an associated buggy relation or an incomplete set of relations.*

Proposition C.1.9 (Completeness)**.** *Let P be a Datalog program and Q be a query with answer $Q_{\mathcal{M}}$ unexpected. Then, the computation graph G for Q w.r.t. P contains either a buggy node or a buggy circuit.*

As part of this thesis, this theoretical ideas have been implemented in a debugger included as part of the Datalog system DES [101], which is available in:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/DD`

The *CG* is built after the user has detected some unexpected answer. The values $[Q, Q_{\mathcal{M}}]$ are stored along the computation and can be accessed afterwards without repeating the computation, thus increasing the efficiency of the graph construction. A novelty of our approach is that it allows the user to choose working either at clause level or at predicate level, depending on the grade of precision that the user needs, and its knowledge of the intended interpretation $\mathcal{I}$. At predicate level, the debugger is able to find a buggy relation or an incomplete set of relations. At clause level, the debugger can provide additional information, namely the rule which is the cause of error.

In order to minimize the number of questions asked to the user, the tool relies on a navigation strategy similar to the *divide & query* presented in [105] for deciding which vertex is selected at each step. In other paradigms it has been shown that this strategy requires an average of $\log_2 n$ questions to find the bug [30], with n the number of nodes in the computation tree. Our experiments confirms that this is also the case when the CGs are in fact trees, i.e., they do not contain cycles, which occurs very often. In the case of graphs containing cycles the results also show this tendency, although a more extensive number of experiments is still needed.

A more detailed explanations about this proposal for debugging Datalog programs, including examples and the proofs of the theoretical results can be found in [31, 32].

C.2. Relational databases

SQL (Structured Query Language [79]) is a language employed by relational database management systems. In particular, the SQL select statement is used for querying data from databases. Realistic database applications often contain a large number of tables, and in many cases, queries become too complex to be coded by means of a single select statement. In these cases, SQL allows the user to define *views*. View queries can rely on previously defined views, as well as on database tables. As in other programming paradigms, views can have bugs and checking their correctness becomes especially painful due to the size of actual databases; it is usual to find select queries and views involving thousands of database rows, and reducing the size of the databases for testing is not a trivial task.

In this Section we discuss the problem of checking correlated SQL queries. Firstly we present a general framework for generating SQL query test cases using Constraint

Logic Programming. Given a database schema and a SQL view defined in terms of other views and schema tables, our technique generates automatically a set of finite domain constraints whose solutions constitute the test database instances. As a theoretical result, we prove the soundness and correctness of the technique w.r.t. the semantics of Extended Relational Algebra. Next we present a debugging technique for diagnosing errors in SQL views based in declarative debugging. The debugger starts with an initial symptom, which in our case corresponds to the unexpected result of a user-defined view. The debugger allows the user to specify the error type, indicating if there is either a missing answer (a tuple was expected but it is not in the result) or a wrong answer (the result contains an unexpected tuple). This information is employed for locating the cause of the error.

Next Section starts by defining basic information about relational databases.

C.2.1. Basic Concepts of Relational Databases

A *relation schema* $\mathcal{R}$ consists of a list of attributes $(A_1, \ldots, A_n)$. Each attribute A_i has an associated domain denoted as $dom(A_i)$. The domain of $\mathcal{R}$ is defined as $dom(\mathcal{R}) = dom(A_1) \times \cdots \times dom(A_n)$. A *relation* or *relation instance* R of relation schema $\mathcal{R}$ is a multiset of elements in $dom(\mathcal{R})$. A *tuple* t of schema $\mathcal{R}$ is an element in $dom(\mathcal{R})$. Duplicate tuples are allowed in a relation. The multiplicity of a tuple t in the relation instance R is denoted as $|R|_t$. A tuple t is an element of relation R if its multiplicity is greater than zero. In this case, we say that $t \in R$. If the multiplicity of t in R is zero, we say that $t \notin R$.

Each tuple t in the relation instance R can be considered as a function such that $dom(t) = \{A_1, \ldots, A_n\}$, with $t(R.A_i)$ the value of the attribute A_i in t. The concatenation of two tuples t, s with disjoint domain is defined as the union of both functions represented as $t \odot s$.

Given a tuple t and an arithmetic expression e defined on the attributes in $dom(t)$, we use the notation $e(t)$ to represent the value obtained applying the substitution t to e. Analogously, let S be a multiset of rows $\{|\mu_1, \ldots, \mu_n|\}$ and let e be an expression. Then $e(S)$ represents the result of replacing each attribute $T.A$ occurring in an *aggregate subexpression* of e by the multiset $\{|\mu_1(T.A), \ldots, \mu_n(T.A)|\}$. The attributes $T.B$ not occurring in aggregate subexpressions of e must take the same value for every $\mu_i \in S$, and are replaced by such value.

In SQL, all data are stored and accessed via relations of a particular relation schema. Relations that store data are called *tables*. Other relations do not store data, but are computed by applying relational operations to other relations. These relations are called *views* or *queries*. In implementations, views can be thought of as new tables created dynamically from existing ones by using a SQL queries. The general syntax of a SQL view is: create view V(A_1, ..., A_n) as Q, with Q a query and $A_1, \ldots, A_n$ the names of the view attributes.

A *database schema* D is a tuple $(\mathcal{T}, \mathcal{C}, \mathcal{V})$, where $\mathcal{T}$ is a finite set of tables, $\mathcal{C}$ a finite set of database constrains and $\mathcal{V}$ a finite set of views. We consider only *primary key* and *foreign key* constraints, defined as traditionally in relational databases. A

database instance d of a database schema is a set of table instances, one for each table in $\mathcal{T}$ verifying $\mathcal{C}$ (thus we only consider *valid instances*). To represent the instance of a table T in d we will use the notation $d(T)$. A *symbolic database instance* d_s is a database instance whose rows can contain logical variables. We say that d_s is satisfied by a substitution μ when $(d_s\mu)$ is a database instance. μ must substitute all the logic variables in d_s by domain values.

The syntax of SQL queries can be found in [79]. We distinguish between *basic queries*, *aggregate queries* and *set queries*.

- *Basic queries* of the form:

  $$\mathsf{Q} = \quad \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w;$$

  with R_j tables or views for $j = 1 \ldots m$, e_i, $i = 1 \ldots n$ expressions involving constants, predefined functions and attributes of the form $B_j.A$, $1 \leq j \leq m$, and A an attribute of R_j.

- *Aggregate queries*, including group by and having clauses:

  $$\mathsf{Q} = \quad \mathsf{select}\ e_1\ E_1, \ldots, e_n\ E_n\ \mathsf{from}\ R_1\ B_1\ , \ldots, R_m\ B_m\ \mathsf{where}\ C_w\ \mathsf{group\ by}\ A'_1, \ldots, A'_k\ \mathsf{having}\ C_h;$$

  with A'_i attributes of the form $B_j.A$, $1 \leq j \leq m$, and A an attribute of R_j.

- *Set queries* of the form $Q = Q_1$ {union [all], except [all], intersect [all]} Q_2;

A set query Q combines the results of two queries Q_1 and Q_2 by means of set operators union [all], except [all][1] or intersect [all] (the keyword all indicates that the result is a multiset).

The semantics of SQL assumed in this framework is given by the Extended Relational Algebra (ERA) [71], an operational semantics allowing aggregates, views, and most of the common features of SQL queries. In ERA, each relation R of relation schema $\mathcal{R}$ is defined as a multiset of tuples in $dom(\mathcal{R})$ and it is considered as a relational expression. ERA defines new relations by means multiset expressions. These expressions combine previously defined relations using set and multiset operators (see [66, 71] for a formal definition of each operator).

We use the notation Φ_R for representing the ERA expression associated to a SQL relation R (table, query or view). For instance, in the case of SQL queries, the select, from and where sections correspond to the projection operation, cartesian product operation and selection condition of ERA respectively. The other sections such as group by and having corresponds to an aggregate expression in ERA as shown in [71]. The SQL set and multiset operators union [all], except [all] and intersect [all] correspond to the set and multiset operations in ERA. Tables are denoted by their names, that is, $\Phi_T = T$ if T is a table. A query/view usually depends on previously defined relations, and sometimes it will be useful to write $\Phi_R(R_1, \ldots, R_n)$ indicating that R depends on relations $R_1, \ldots, R_n$.

The *dependency tree* of any view V in the schema is a tree with V labeling the root, and its children the dependency trees of the relations occurring in the from

[1] The Oracle database systems uses MINUS instead of the SQL standard EXCEPT.

clause of its query. This concept can be easily extended to queries by assuming some arbitrary name labeling the root node, and to tables, where the dependency tree will be a single node labeled by the table name.

The *computed answer* of an ERA expression Φ_R with respect to some schema instance d is denoted by $\| \Phi_R \|_d$, where:

- If R is a database table, $\| \Phi_R \|_d = d(R)$.
- If R is a database view or a query and R depends on the relations $R_1, \ldots, R_n$, then $\| \Phi_R \|_d = \Phi_R(M_1, \ldots, M_n)$, where $M_i = \| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.

meaning that the computed answer of an expression Φ_R in the instance d is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its computed answer $\| \Phi_{R_i} \|_d$ for $i = 1 \ldots n$.

Queries are executed by SQL systems. The answer for a query Q in an implementation is represented by $\mathcal{SQL}(Q,d)$. The notation $\mathcal{SQL}(R,d)$ abbreviates $\mathcal{SQL}$(select * from R,d). In particular, we assume in this report the existence of *correct* SQL implementations. A *correct* SQL implementation verifies that $\mathcal{SQL}(Q,d) = \| \Phi_Q \|_d$ for every query Q.

C.2.2. SQL Test Cases

In order to simplify our framework we assume queries such that:

- The where and having clauses only contain existential subqueries of the form exists Q (or not exists Q). It has been shown that other subqueries of the form ... in Q, ... any Q or ... all Q can be translated into equivalent subqueries with exists and not exists (see for instance [69]). Analogously, subqueries occurring in arithmetic expressions can be transformed into exists subqueries.
- The from clause does not contain subqueries. This is not a limitation since all the subqueries in the from clause can be replaced by views.
- We also do not allow the use of the distinct operator in the select clause. It is well-known that queries using this operator can be replaced by equivalent aggregate queries without distinct.
- Our setting does not allow: recursive queries, the minus operator, join operations, and null values. All these features, excepting the recursive queries, can be integrated in our setting, although they have not been considered here for simplicity.

We distinguish between positive and negative test cases. We say that a non-empty database instance d is a *positive test case* (PTC) for a view V when $\| \Phi_R \|_d \neq \emptyset$. We require that the (positive or negative) test case contains at least one row that will act as witness of the possible error in the view definition. The overall idea is that we consider d a PTC for a view when the corresponding query answer is not empty. In a basic query this means that at least one tuple in the query domain satisfies the where condition. In the case of aggregate queries, a PTC will require finding a valid

aggregate verifying the having condition, which in turn implies that all its rows verify the where condition. If the query is a set query, then the ranges are handled according to the set operation involved.

The negative test cases (NTC) are defined by modifying the initial queries and then applying the concept of positive test case. The main advantage is that only a positive test case generator must be implemented. With this purpose we use the notation Q_{C_w} and $Q_{(C_w,C_h)}$ to indicate that C_w is the where condition in Q and C_h is the having condition in Q (when Q is an aggregate query). If Q_{C_w} is of the form select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where C_w; then the notation $Q_{not(C_w)}$ represents select e_1 E_1, ..., e_n E_n from R_1 B_1 , ..., R_m B_m where not(C_w); and analogously for $Q_{(C_w,C_h)}$ and $Q_{(not(C_w),C_h)}$, $Q_{(C_w,not(C_h))}$, and $Q_{(not(C_w),not(C_h))}$. For instance, in the case of basic query, we expect that a NTC will contain some row in the domain of the view not verifying the where condition. Then, we say that a database instance d is a NTC for a view V with associated basic query Q_{C_w} when d is a PTC for $Q_{not(C_w)}$.

It is possible to obtain a test case which is both positive and negative at the same time thus achieving *predicate coverage* with respect to the where and having conditions (in the sense of [12]). We will call these tests PNTCs.

C.2.2.1. Generating PTC

The input of our test case generator consists of:

- A database schema D defined by means of SQL language, i.e., a finite set of tables $\mathcal{T}$, constraints $\mathcal{C}$ and views $\mathcal{V}$, as well as the integrity constraints for the columns (primary and foreign keys).
- A SQL view V for which the test case is to be generated.

The generating process begins by building a *symbolic database instance* $d_s(T)$ for each table T in D. Each table will contain an arbitrary number of rows, and each attribute value in each row will correspond to a fresh logic variable with its associated domain integrity constraints. Next, for each relation R in D, we define the multiset $\theta(R)$ as a multiset of pairs (ψ, u) with ψ a first order formula, and u a row in $d_s(R)$. The idea is that the row u will be in the relation instance $d_s(R)$ iff the formula ψ is satisfied.

From the multiset $\theta(V) = \{\!|(\psi_1, u_1), \ldots, (\psi_n, u_n)|\!\}$, formula:

$$\delta = (\bigvee_{i=1}^{n} \psi_i)$$

is written into a specific constraints programming language. Then, a constraint satisfaction problem (CSP) is obtained. Solutions satisfying this formula δ can be computed using existing techniques and tools available in the CSP community (we have chossen SICStus Prolog). These solutions constitute a PTC for the view V.

Next we define formally the multiset $\theta(R)$:

Definición C.2.1. *Let $D = (\mathcal{T}, \mathcal{C}, \mathcal{V})$ a database schema, d_s a symbolic database instance and R a relation in D. We define $\theta(R)$ as a multiset of the form (φ, μ), with φ a first order formula and μ a row in $d_s(R)$. This multiset is defined as follows:*

1. *Let T a table in D such that $d_s(T) = \{\!|\mu_1, \ldots, \mu_n|\!\}$, then:*

$$\theta(T) = \{\!|(\varphi_1, \mu_1), \ldots, (\varphi_n, \mu_n)|\!\}$$

with φ_i a first order formula representing the primary key and foreign key constraints. If $pk(T) = \{A_1, \ldots, A_m\}$ is a primary key, we define:

$$\phi_i = (\bigwedge_{j=1, j\neq i}^{n} (\bigvee_{k=1}^{m} (\mu_i(T.A_k) \neq \mu_j(T.A_k)) \ ) \ )$$

for $i = 1 \ldots n$. Formulas $\phi_1, \ldots, \phi_n$ represent the primary key constraints of T. If T has s foreign keys of the form $fk_p(T, T_p) = \{(A_1, \ldots, A_m), (B_{p1}, \ldots, B_{pm})\}$, with $p = 1 \ldots s$ and T_p a table in D and $d_s(T_p) = \{\!|\nu_1, \ldots, \nu_{n_2}|\!\}$, then we define:

$$\psi_{ip} = \ (\bigvee_{j=1}^{n_2} (\bigwedge_{k=1}^{m} (\ \mu_i(T.A_k) = \nu_j(T_p.B_{pk}) \ ) \ ) \ )$$

for $i = 1 \ldots n$ and $p = 1 \ldots s$. Formulas ψ_{ip} represent the foreign key constraints of T Then,

$$\varphi_i = \phi_i \wedge \psi_{i1} \wedge \cdots \wedge \psi_{is}$$

2. *Let V a view in D defined as $V =$* create view *$V(A_1, \ldots, A_n)$* as *Q. The multiset $\theta(V)$ is defined as:*

   - *If Q is a basic query:*

     select $e_1\ E_1, \ldots, e_n\ E_n$ from $R_1\ B_1, \ldots, R_m\ B_m$ where C;

     then:

$$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

     Firstly, we build the multiset P. For each $(\psi_1, \nu_1) \in \theta(R_1), \ldots, (\psi_m, \nu_m) \in \theta(R_m)$ with $|\theta(R_1)|_{(\psi_1, \nu_1)} = n_1, \ldots, \ |\theta(R_m)|_{(\psi_m, \nu_m)} = n_m$ the tuple $t = (\psi_1 \wedge \cdots \wedge \psi_m, {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m})$ is in P with $|P|_t = n_1 \times \cdots \times n_m$.
     $\theta(Q)$ is defined from P. For each tuple of the form $(\psi, \mu) \in P$ with multiplicity k, we define:

     - *$s_Q(\mu) = \{E_1 \mapsto e_1(\mu), \ldots, E_n \mapsto e_n(\mu)\}$*
     - *Formula $\varphi(C, \mu)$ is defined: as*
       - *If $C \equiv C_1$ and C_2, then $\varphi(C, \mu) = \varphi(C_1, \mu) \wedge \varphi(C_2, \mu)$*
       - *If $C \equiv C_1$ or C_2 , then $\varphi(C, \mu) = \varphi(C_1, \mu) \vee \varphi(C_2, \mu)$*
       - *If $C \equiv$ not (C_1), then $\varphi(C, \mu) = \neg\varphi(C_1, \mu)$*
       - *If $C \equiv e$ with e an expression without subqueries, $\varphi(C, \mu) = C\mu$*

- *If* $C \equiv (exists\ Q_E)$, *with* $\theta(Q_E) = \{\!|(\psi_1, \mu_1), \ldots (\psi_k, \mu_k)|\!\}$. *Then* $\varphi(C, \mu) = (\vee_{i=1}^{k} \psi_i)$.

Then, $(\psi \wedge \varphi(C, \mu), s_Q(\mu))$ *is in* $\theta(Q)$ *con* $|\theta(Q)|_{(\psi \wedge \varphi(C,\mu), s_Q(\mu))} = k$

- *If* Q *is of the form:*

  select $e_1\ E_1, \ldots, e_n\ E_n$ from $R_1\ B_1, \ldots, R_m\ B_m$
  where C_w group by $e'_1, \ldots, e'_k$ having C_h

  then:

$$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

  $\theta(Q)$ *is defined from* P. *For each non-empty multiset* $A = \{\!|(\psi_1, \mu_1), \ldots, (\psi_j, \mu_j)|\!\} \subseteq P$ *such that* $|A|_{(\psi_i,\mu_i)} = |P|_{(\psi_i,\mu_i)}$ *for any tuple* $(\psi_i, \mu_i) \in A$, $1 \leq i \leq j$, *we define:*

  - $\Pi_1(A) = \psi_1 \wedge \cdots \wedge \psi_j$,
  - $\Pi_2(A) = \{\!|\ \mu_1, \ldots, \mu_j|\!\}$,
  - $aggregate(Q, A) = group(Q, \Pi_2(A)) \wedge maximal(Q,A) \wedge \varphi(C_h, \Pi_2(A))$
  - $group(Q, A) = (\varphi(C_w, \mu_1) \wedge \cdots \wedge \varphi(C_w, \mu_j)) \wedge (\bigwedge_{i=1\ldots j}(\bigwedge_{l>i}(e'_1(\mu_i) = e'_1(\mu_l) \wedge \cdots \wedge e'_k(\mu_i) = e'_k(\mu_l))))$
  - $maximal(Q, A) = \bigwedge_{(\psi,\mu) \in P \wedge (\psi,\mu) \notin A}(\neg\psi \vee \neg group(Q, A \cup (\psi, \mu)))$

  Then, $t = (\Pi_1(A) \wedge aggregate(Q, A), s_Q(\Pi_2(A)))$ *is in* $\theta(Q)$ *with multiplicity 1.*

- *For set queries:*
  - *If* $Q = (Q_1$ union [all] $Q_2)$, *then:*

$$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

    with $E_1, \ldots, E_n$ *the names of the attributes in the* select *clause of the queries* Q_1 *y* Q_2.
    - *If* $Q = (Q_1$ union all $Q_2)$, $\theta(Q) = \theta(Q_1)\ \cup_{\mathcal{M}} \theta(Q_2)$
    - *If* $Q = (Q_1$ union $Q_2)$,

$$\begin{aligned}\theta(Q) = \quad & \{(\psi, \mu)\ \mid \\ & \psi = (\psi_{11} \vee \cdots \vee \psi_{1k_1}) \vee (\psi_{21} \vee \cdots \vee \psi_{2k_2}), \\ & (\psi_{11}, \mu) \in \theta(Q_1), \ldots, (\psi_{1k_1}, \mu) \in \theta(Q_1), \\ & (\psi_{21}, \mu) \in \theta(Q_2), \ldots, (\psi_{2k_2}, \mu) \in \theta(Q_2)\}\end{aligned}$$

  - *If* $Q = (Q_1$ intersection $Q_2)$, *then:*

$$\theta(V) = \theta(Q)\{E_1 \mapsto V.A_1, \ldots, E_n \mapsto V.A_n\}$$

    with $E_1, \ldots, E_n$ *the names of the attributes in the* select *clause of the queries* Q_1 *y* Q_2.

$$\begin{aligned}\theta(Q) = \quad & \{(\psi, \mu)\ \mid \\ & \psi = (\psi_{11} \vee \cdots \vee \psi_{1k_1}) \wedge (\psi_{21} \vee \cdots \vee \psi_{2k_2}), \\ & (\psi_{11}, \mu) \in \theta(Q_1), \ldots, (\psi_{1k_1}, \mu) \in \theta(Q_1), \\ & (\psi_{21}, \mu) \in \theta(Q_2), \ldots, (\psi_{2k_2}, \mu) \in \theta(Q_2)\}\end{aligned}$$

Observe that the notation $s_Q(x)$ with Q a query is a shorthand for the row μ with domain $\{E_1, \ldots, E_n\}$ such that $E_i(x) = e_i(x)$, with $i = 1 \ldots n$, with select e_1 E_1, $\ldots$, e_n E_n *the* select *clause of Q. If E_i's are omitted in the query, it is assumed that $E_i = e_i$.*

Following example illustrates the previous definition. Let V_1 , V_2 be two SQL views defined as:

create view $V_1(A_1, A_2)$ as
select $T_1'.A$ E_1, $T_1'.B$ E_2
from T_1 T_1'
where $T_1'.A \geq 10$

create view $V_2(A)$ as
select $T_2'.C$ E_1
from V_1 V_1', T_2 T_2'
where $V_1'.A_1 + T_2'.C = 0$

Suppose table T_1 has the attributes A, B while table T_2 has only one attribute C. Consider the following symbolic database instances $d(T_1) = \{\!|\mu_1, \mu_2|\!\}$ and $d(T_2) = \{\!|\mu_3, \mu_4|\!\}$ with: $\mu_1 = \{T_1.A \mapsto x_1, T_1.B \mapsto y_1\}$, $\mu_2 = \{T_1.A \mapsto x_2, T_1.B \mapsto y_2\}$ and $\mu_3 = \{T_2.C \mapsto z_1\}$, $\mu_4 = \{T_2.C \mapsto z_2\}$. Then:

$$\theta(T_1) = \{\!|(true, \mu_1), (true, \mu_2)|\!\}, \quad \theta(T_2) = \{\!|(true, \mu_3), (true, \mu_4)|\!\}$$

$$\begin{aligned}
\theta(V_1) = \{\!| \quad & (x_1 \geq 10, \{V_1.A_1 \mapsto x_1, V_1.A_2 \mapsto y_1\}) \ , \\
& (x_2 \geq 10, \{V_1.A_1 \mapsto x_2, V_1.A_2 \mapsto y_2\}) \ \ |\!\} \\
\theta(V_2) = \{\!| \quad & (x_1 \geq 10 \wedge x_1 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\
& (x_1 \geq 10 \wedge x_1 + z_2 = 0, \{V_2.A \mapsto z_2\}), \\
& (x_2 \geq 10 \wedge x_2 + z_1 = 0, \{V_2.A \mapsto z_1\}), \\
& (x_2 \geq 10 \wedge x_2 + z_2 = 0, \{V_2.A \mapsto z_2\}) \ \ |\!\}
\end{aligned}$$

The following theorem contains the idea for generating constraints that will yield the PTCs:

Teorema C.2.2. *Let D be a database schema and d_s a symbolic database instance. Assume that the views and queries in D do not include subqueries. Let R be a relation in D such that $\theta(R) = \{\!|(\psi_1, \mu_1), \ldots (\psi_n, \mu_n)|\!\}$, and η a substitution satisfying d_s. Then $d_s\eta$ is a PTC for R iff $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

A SICStus Prolog prototype implementing these ideas can be downloaded and tested from:

`https://gpd.sip.ucm.es/yolanda/research.htm`.

Our test case generator is bundled as a component of the Datalog deductive database system DES [101, 100].

Up to now, we support integer and string datatypes by using the finite domain ($\mathcal{FD}$) constraint system available in SICStus Prolog. Posting our constraints over integers to the underlying $\mathcal{FD}$ constraint solver is straightforward. Although constraint systems over other domains, as reals or rationals, are available, we have not used them in our current work. However, they can be straightforwardly implemented. In addition, enumerated types (available in object-oriented SQL extensions) could also be included, following a similar approach to the one taken for strings.

```
DES-SQL> CREATE OR REPLACE TABLE t1( a  int PRIMARY KEY, b int);
DES-SQL> CREATE OR REPLACE TABLE t2( c  int PRIMARY KEY);
DES-SQL> CREATE OR REPLACE VIEW v1(a1, a2) AS
            SELECT t1.a e1, t1.b e2 FROM t1 where t1.a >= 10;
DES-SQL> CREATE OR REPLACE VIEW v2(a)  AS
            SELECT t2.c e1 FROM v1, t2 where v1.a1 + t2.c = 0;

DES-SQL> /test_case v2 positive

Info: Test case over integers and strings:
[t1(12,-12),t2(-12)]
```

Figure C.4: System session.

The user can choose the type of test case to be generated, either PTC, or NTC or both PNTC for any view V previously defined in D. The output is a database instance d of a database schema D such that d is a test case for the given view V with as few entries as possible.

Figure C.4 shows a system session, which firstly consists of creating tables and views. Relations `t1`, `t2` , `v1` and `v2` are created in the system DES. Then, PTC for the view `v2` can be obtained via the command `/test_case v2 positive`.

The obtained instance `[t1(12,-12),t2(-12)]` constitute, by Theorem C.2.2, a PTC for view `v2`.

It is well-known that the problem of finding complete sets of test cases is in general undecidable [12]. Different coverage criteria have been defined (see [12] for a survey) in order to define test cases that are complete at least w.r.t. some desired property. In this work, we have considered a simple criterion for SQL queries, namely the *predicate coverage criterium*. However, it has been shown [114] that other coverage criteria can be reduced to predicate coverage by using suitable query transformations.

In practice our system cannot reach completeness, but only weak completeness module the size of the tables of the instance. That is, our system will find a PTC if it is possible to construct it with all the tables containing a number of rows less than an arbitrary number. If it is not possible, the number of rows is increased. The process is repeated stopping either when a PTC is found or when an upper bound is reached. Both the lower and the upper limits are user configurable.

Although test case generation is a time consuming problem, the efficiency of our prototype is reasonable, finding in a few seconds test cases for views with dependence trees of about ten nodes and with a number of rows limited to seven for every table. The main efficiency problem comes from aggregate queries, where the combinatorial problem of selecting the aggregates can be too complex for the solver.

An more detailed explanations about this proposal for generating SQL views test cases using Constraint Logic Programming, including examples and the proofs of the theoretical results can be found in [33].

This theoretical ideas can be integrated into tools for debugging SQL queries. It is the case of declarative debugging of SQL queries presented in next Section.

C.2.3. Declarative Debugging of SQL views

Given the high-abstraction level of SQL, usual techniques like trace debugging are difficult to apply. *Declarative Debugging*, is a technique that can be described as a general debugging schema [91] which starts when an *initial error symptom* is detected by the user. In the context of SQL views, the initial symptom corresponds to an unexpected result produced by a view. The debugger automatically builds a tree representing the erroneous computation that has produced the symptom. In SQL, each node in the tree contains information about both a relation, which is a table or a view, and its associated computed result. The root of the tree corresponds to the query that produced the initial symptom. The children of each node correspond to the relations (tables or views) occurring in the definition of its associated query. After building the tree, it is *navigated* by the debugger, asking to the user about the validity of some nodes. When a node contains the expected result, it is marked as *valid*, and otherwise it is marked as *nonvalid*. The goal of the debugger is to locate a *buggy node*, which is a nonvalid node with valid children. It can be proved that each buggy node in the tree corresponds to either an erroneously defined view, or to a database table containing erroneous data. A debugger based on these ideas is presented in [36](A.4). The main criticism that can be leveled at this proposal is that it can be difficult for the user to check the validity of the results. Indeed, the results returned by SQL views can contain hundreds or thousands of tuples. In [37](A.7) we present a new technique for debugging SQL views based in the ideas of declarative debugging in order to overcome or at least to reduce this drawback. This is done by asking for more specific information from the user. The questions are now of the type "Is there a missing answer (that is, a tuple is expected but it is not there) or a wrong answer (an unexpected tuple is included in the result)?" The answer provided by the user can be either "Yes" or "No". In the case or "No" the user can point out a wrong tuple or a missing tuple. With this information, the debugger can reduce the number of questions directed at the user. Our technique considers only those relations producing/losing the wrong/missing tuple. Additionally, the questions directed at the user about the validity in the children nodes can be simplified. For instance, the debugger only considers those tuples that are needed to produce the wrong or missing answer in the parent.

In this proposal, the computation tree is represented using Horn clauses, which allows us to include the information obtained from the user during the session.

In Section C.2.3.2 we present our first proposal for debugging sql views, and in Section C.2.3.3 we present a declarative debugging technique that refines the previous one. In next Section we describe the main concepts of declarative debugging applied to SQL views.

C.2.3.1. Declarative debugging framework

In our context, we assume that the user can check if the computed answer for a relation matches its intended answer. The *intended answer* for a relation R w.r.t. d, is a multiset denoted as $\mathcal{I}(R,d)$ containing the answer that the user expects for the query select * from R in the instance d. This concept corresponds to the idea of *intended interpretations* employed usually in algorithmic debugging. We say that $\mathcal{SQL}(R,d)$ is an *unexpected answer* for a relation R if $\mathcal{I}(R,d) \neq \mathcal{SQL}(R,d)$. Observe that $\mathcal{I}(R,d) \neq \mathcal{SQL}(R,d)$ means that there is some tuple t such that $|\mathcal{I}(R,d)|_t \neq |\mathcal{SQL}(R,d)|_t$. The existence of an unexpected answer implies the existence of either a *wrong tuple* or a *missing tuple*. We say that t is a *wrong tuple* for a relation R if:

$$|\mathcal{SQL}(R,d)|_t > 0 \text{ and } |\mathcal{I}(R,d)|_t < |\mathcal{SQL}(R,d)|_t$$

We say that t is a *missing tuple* for a relation R if:

$$|\mathcal{I}(R,d)|_t > 0 \text{ and } |\mathcal{I}(R,d)|_t > |\mathcal{SQL}(R,d)|_t$$

An unexpected answer produced by a relation R does not imply that R is erroneous. In order to define the key concept of erroneous relation we need the auxiliary concept of *expectable* answer. The *expectable* answer for a relation R w.r.t. the instance d is denoted as $\mathcal{E}(R,d)$. If R is a table, $\mathcal{E}(R,d) = R$ and $\mathcal{E}(R,d) = \mathcal{E}(Q,d)$ if R is a view, with Q the query defining R. If R is a query and $R_1,\ldots,R_n$ the relations occurring in R, then $\mathcal{E}(R,d) = \Phi_R(I_1,\ldots,I_n)$ where $I_i = \mathcal{I}(R_i,d)$ for $i = 1\ldots n$, meaning that $\mathcal{E}(R,d)$ is the result of evaluating the expression Φ_R after substituting each relation name R_i in Φ_R by its intended answer $\mathcal{I}(R_i,d)$ for $i = 1\ldots n$. Thus, if R is a table, the expectable answer of R is the table instance R. In the case of a view V, the expectable answer corresponds to the expectable answer for the query Q defining V. In the case of a query Q, the expectable answer corresponds to the computed result that would be obtained assuming that all the relations R_i occurring in the definition of Q contain the intended answers.

A discrepancy between $\mathcal{I}(R,d)$ and $\mathcal{E}(R,d)$ indicates that R does not compute its intended answer, even assuming that all the relations it depends on correspond to their intended answers. Such relation is called *erroneous*. We say that a relation R is *erroneous* when $\mathcal{I}(R,d) \neq \mathcal{E}(R,d)$, and *correct* otherwise. Definition of *erroneous* relation cannot be used directly for defining a practical debugging tool, because in order to point out a view V as erroneous, it would require comparing $\mathcal{I}(V,d)$ and $\mathcal{E}(V,d)$. Asking about $\mathcal{E}(V,d)$ to the users is unrealistic; we only can assume that they know $\mathcal{I}(V,d)$ but not $\mathcal{E}(V,d)$. The following result relate the concept of erroneous relation and the concept of computed answer, which is used in our debugger.

Teorema C.2.3. *Let V be a database view and $R_1,\ldots,R_n$ the relations occurring in the query defining V such that $\mathcal{I}(R_i,d) = \mathcal{SQL}(R_i,d)$ for $i = 1\ldots n$. Then, $\mathcal{SQL}(V,d)$ is unexpected iff V is erroneous.*

The debugging process is based on the result in Theorem C.2.3. We emphasize the fact that the debugger requires from the user only to answer questions about the intended answer $\mathcal{I}(R,d)$, which is known by him, instead of the expectable answer

$\mathcal{E}(R, d)$. In order to locate erroneous relations, the debugger compares the computed answer –obtained from the SQL system– and the intended answer –known by the user–.

C.2.3.2. Declarative Debugging of SQL views with computation trees

The debugging process starts when the user finds a view R returning an unexpected answer. In a first phase, the debugger builds a *computation tree* for this view R. The definition of this structure is the following:

Definición C.2.4. *The computation tree $CT(R)$ associated with a relation R is defined as follows:*

- *The root of $CT(R)$ is $(R \mapsto \mathcal{SQL}(R, d))$.*
- *For any node $N = (R' \mapsto \mathcal{SQL}(R', d))$ in $CT(R)$:*
  - *If R' is a table, then N has no children.*
  - *If R' is a view, the children of N will correspond to the CTs for the relations occurring in the query associated with R'.*

Although Definition C.2.4 includes the computed answer $\mathcal{SQL}(R, d)$ as part of nodes, this information is not relevant for the tree structure. In practice, this will lead to a non-efficient implementation in terms of memory usage. Instead, the computed answers are obtained from the SQL system by the debugger when needed. With this simplification, the computation tree corresponds to the dependency tree of the view R in the schema. After building the computation tree, the debugger will *navigate* the tree, asking the user about the validity of some nodes. Let $CT(R)$ be a computation tree, and $N = (R' \mapsto \mathcal{SQL}(R', d))$ a node in $CT(R)$. We say that N is a *valid node* when $\mathcal{SQL}(R', d) = \mathcal{I}(R', d)$, a *nonvalid node* when $\mathcal{SQL}(R', d) \neq \mathcal{I}(R', d)$, and a *buggy node* when N is nonvalid and all its children in $CT(R)$ are valid.

The goal of the debugger is to locate buggy nodes. The next theorem shows that a computation tree with a nonvalid root always contains a buggy node, and that every buggy node corresponds to an erroneous relation.

Teorema C.2.5. *Let V a database view and $CT(V)$ its associated computation tree. If the root of $CT(V)$ is nonvalid, then:*

- *Completeness. $CT(V)$ contains a buggy node.*
- *Soundness. Every buggy node in $CT(V)$ corresponds to an erroneous relation.*

A debugger technique following these ideas was presented in [36]. The debugger does not allow the user to point out wrong/missing tuples but it only allows the user to indicate the validity/nonvalidity of certain relations which correspond to nodes in the tree. However, these questions can be difficult to answer when the computed answer contains hundreds or thousands of tuples. In the next Section we improve the debugging technique presented in this Section by allowing the user to specify if an unexpected answer contains a wrong or a missing tuple.

Code 1 debug(V)

```
Input: V: view name
Output: A list of buggy views
 1: A := askOracle(all V)
 2: if A ≡ no or A ≡ missing(t) or A ≡ wrong(t)
 3:    Valid := true
 4:    P := initialSetOfClauses(V, A)
 5:    while getBuggy(P)=[] do
 6:       LE := getUnsolvedEnquiries(P)
 7:       E := chooseEnquire(LE)
 8:       A := askOracle(E)
 9:       Valid := checkAnswer(A)
10:       if Valid   P := P ∪ processAnswer(E,A)
11:    end while
12:    L := getBuggy(P)
13: else
14:    L := [ ]
15: end if
16: return L
```

C.2.3.3. Improved Debugging Algorithm

This Section refines the ideas for debugging SQL views presented so far. Although the process is based on the ideas of declarative debugging, this proposal does not use computation trees explicitly. Instead, our debugger represents the logic inferences defining buggy nodes in computation trees by means of Horn clauses.

The general schema of the algorithm is summarized in the code of function *debug* (Code 1). The code of some basic auxiliary functions can be found in [37]. The debugger is started by the user when an unexpected answer is obtained as computed answer for some SQL view V. Then, the algorithm asks the user about the type of error (line 1). The answer A can be simply *no*, or a more detailed explanation of the error, like *wrong(t)* or *missing(t)*, indicating that t is a wrong or missing tuple respectively. During the debugging process, variable P keeps a list of Horn clauses representing a logic program. The atoms in the body of the clauses represent *enquiries* that might represent questions to the user. The initial list of clauses P is generated by the function *initialSetofClauses* (line 4). This function introduces the clauses that correspond to the computation tree rooted by V. The purpose of the main loop (lines 5-11) is to add information to the program P, until a buggy view can be inferred. The function *getBuggy* returns the list of all the relations R such that *buggy(R)* can be proven w.r.t. the logic program P. Each iteration of the loop represents the election of an enquiry in a body atom whose validity has not been established yet (lines 6-7). Then, in line 8 the debugger asks to the user about the result of the question associated to the chosen enquiry. Finally, the answer is processed (line 10).

The function *initialSetofClauses* gets as first input parameter the initial view V. This view has returned an unexpected answer, and the input parameter A contains

the explanation. The output of this function is a set of clauses representing the logic relations that define possible buggy relations with predicate *buggy*. This clauses are of the form:

```
buggy(R1)   :- state(all(R1),nonvalid),
               state(all(R2),valid), state(all(R3),valid).
```

with R1, R2, R3 relations, and R1 defined from R2 and R3. The information about the validity/nonvalidity of the results associated to enquiries is represented in our setting by predicate *state*. The first parameter of *state* is an enquiry E, and the second one can be either *valid* or *nonvalid*. Enquiries can be of any of the following forms: *(all R)*, *(s ∈ R)*, or *(R'⊆ R)*, with *R*, *R'* relations, and *s* a tuple with the same schema as relation *R*. Each enquiry E corresponds to a specific question with a possible set of answers and an associated complexity $\mathcal{C}(E)$:

- If $E \equiv$ *(all R)*. Let $S = \mathcal{SQL}(R)$[2].The associated question asked to the user is *"Is S the intended answer for R?"* The answer can be either *yes* or *no*. In the case of *no*, the user is asked about the type of the error, *missing* or *wrong*, giving the possibility of providing a witness tuple t. If the user provides this information, the answer is changed to *missing(t)* or *wrong(t)*, depending on the type of the error. We define $\mathcal{C}(E) = |S|$, with $|S|$ the number of tuples in S.

- If $E \equiv$ *(R'⊆ R)*. Let $S = \mathcal{SQL}(R')$. Then the associated question is *"Is S included in the intended answer for R?"* As in the previous case the answers allowed are *yes* or *no*. In the case of *no*, the user can point out a wrong tuple $t \in S$ and the answer is changed to *wrong(t)*. $\mathcal{C}(E) = |S|$ as in the previous case.

- If $E \equiv$ *(s ∈ R)*. Tuple *s* can be a partial tuple. The question is *"Does the intended answer for R include a tuple matching the tuple s?"* The possible answers are *yes* or *no*. No further information is required from the user. In this case $\mathcal{C}(E) = 1$, because only one tuple must be considered.

In the case of *wrong* answers, the user typically points to a tuple in the result *R*. In the case of *missing* answers, the tuple must be provided by the user, and in this case *partial* tuples are allowed. In our setting, *partial* tuples are tuples that might contain the special symbol $\perp$ in some of their components and *total* in other case. The answer *yes* corresponds to the state *valid*, while the answer *no* corresponds to *nonvalid*. An atom *state(q,s)* occurring in the body of a clause in P implies an enquiry *q*. The enquiry *q* is a *solved enquiry* if the logic program P contains at least one fact of the form *state(q, valid)* or *state(q, nonvalid)*, that is, if the enquiry has been already solved. The enquiry *q* is called an *unsolved enquiry* otherwise. The function *getUnsolvedEnquiries* (see line 6 of Code 1) returns in a list all the unsolved enquiries occurring in body atoms of clauses in P. The function *chooseEnquiry* (line 7, Code 1) chooses one of these enquiries according to some predefined criteria. Our current implementation chooses the enquiry E that implies the smaller complexity value $\mathcal{C}(E)$, although other

[2]We use the notation $\mathcal{SQL}$(R) for representing $\mathcal{SQL}$(R,d). The parameter d is omitted from now on for simplicity

Code 2 processAnswer(E,A)

Input: E: enquiry, A: answer obtained for the enquiry
Output: A set of new clauses

```
1: if A ≡ yes
2:    P := {state(E,valid).}
3: else if A ≡ no or A ≡ missing(t) or A ≡ wrong(t)
4:    P := {state(E,nonvalid).}
5: end if
6: if E ≡ (s ∈ R) and (A ≡ yes)
7:    P:= P ∪ processAnswer((all R),missing(s))
8: else if E ≡ (V ⊆ R) and (A ≡ wrong(s) or A ≡ no)
9:    P:= P ∪ processAnswer((all R), A)
10: else if E ≡ (all V) with V a view and (A ≡ missing(t) or A ≡ wrong(t))
11:    Q := SQL query defining V
12:    P := P ∪ slice(V,Q,A)
13: end if
14: return P
```

more elaborated criteria could be defined without affecting the theoretical results supporting the technique. Once the enquiry has been chosen, Code 1 uses the function *askOracle* (line 8) in order to ask to the user the associated question, returning the answer.

Function *processAnswer* can be found in Code 2. This function process the user answer distinguishing several cases depending on the form of its associated enquiry. The first lines (1-5) introduce a new logic fact in the program with the state that corresponds to the answer A obtained for the enquiry E. The rest of the code distinguishes several cases depending on the form of the enquiry and its associated answer. For instance, if the enquiry is *(all V)* for some view V, and with an answer including either a wrong or a missing tuple, the function *slice* (line 12) is called. This function exploits the information contained in the parameter A (*missing(t)* or *wrong(t)*) for slicing the query Q in order to produce, if possible, new clauses which might allow the debugger to detect incorrect relations by asking simpler questions to the user. The implementation of *slice* can be found in Code 3. The function receives the view V, a subquery Q, and an answer A as parameters. Initially, Q is the query defining V, and A the user answer, but this situation can change in the recursive calls. The function distinguishes several particular cases.

If the query Q combines the results of Q_1 and Q_2 by means of either the operator union or union all, and A is *wrong(t)* (query Q produces too many copies of t), then, if any Q_i produces as many copies of t as Q, we can blame Q_i as the source of the excessive number of t's in the answer for V (lines 4 and 5). The case of subqueries combined by the operator intersect [all], with $A \equiv$ *missing(t)* is analogous, but now detecting that a subquery is the cause of the scanty number of copies of t in $\mathcal{SQL}(V)$. If Q is defined as a basic query without group by section (line 10), then either function *missingBasic* or *wrongBasic* is called depending on the form of A.

Code 3 slice(V,Q,A)

Input: V: view name, Q: query, A: answer
Output: A set of new clauses

```
1: P := ∅;    S= SQL(Q);
2: if (A ≡ wrong(t) and Q ≡ Q1 union [all] Q2) or
      (A ≡ missing(t) and Q ≡ Q1 intersect [all] Q2)
3:    S1= SQL(Q1);    S2= SQL(Q2);
4:    if |S1|t = |S|t   P:= P ∪ slice(V, Q1, A)
5:    if |S2|t = |S|t   P:= P ∪ slice(V, Q2, A)
6: else if A ≡ missing(t) and Q ≡ Q1 except [all] Q2
7:    S1= SQL(Q1);    S2= SQL(Q2);
8:    if |S1|t = |S|t   P:= P ∪ slice(V, Q1, A)
9:    if Q ≡ Q1 except Q2 and t ∈⊥ S2   P :=P∪ slice(V,Q2,wrong(t))
10: else if basic(Q) and groupBy(Q)=[]
11:    if A ≡ missing(t)   P := P ∪ missingBasic(V, Q, t)
12:    else if A ≡ wrong(t)   P := P ∪ wrongBasic(V, Q, t)
13: end if
14: return P
```

Both *missingBasic* and *wrongBasic* can add new clauses that allow the system to infer buggy relations by posing questions which are easier to answer. Function *missingBasic*, defined in Code 4, is called (line 11 of Code 3) when A is *missing(t)*. The input parameters are the view V, a query Q, and the missing tuple t. Notice that Q is in general a component of the query defining V. For each relation R with alias S occurring in the from section of Q, the function checks if R contains some tuple that might produce the attributes of the form $S.A$ occurring in the tuple t. This is done by constructing a tuple s undefined in all its components (line 4) except in those corresponding to the select attributes of the form $S.A$, which are defined in t (lines 5 - 7). If R does not contain a tuple matching s in all its defined attributes (line 8), then it is not possible to obtain the tuple t in V from R. In this case, a buggy clause is added to the program P (line 9) meaning that if the answer to the question *"Does the intended answer for R include a tuple s?"* is *no*, then V is an incorrect relation.

The implementation of *wrongBasic* can be found in Code 5. In line 1, this function creates an empty set of clauses. In line 2, variable F stands for the set containing all the relations in the from section of the query Q. Next, for each relation $R_i \in F$ (lines 4 - 7), a new view V_i is created in the database schema after calling the function *relevantTuples* (line 6). This auxiliary view contains only those tuples in relation R_i that contribute to produce the wrong tuple t in V. Finally, a new buggy clause for the view V is added to the program P (line 8) explaining that the relation V is buggy if the answer to the question associated to each enquiry of the form $V_i \subseteq \mathrm{R}_i$ is *yes* for $i \in \{1 \ldots n\}$.

We have introduced the debugging algorithm, explaining the intuitive ideas supporting the technique. Now we establish formally the correctness and the completeness of our proposal.

Code 4 missingBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A new list of Horn clauses

```
1: P := ∅;    S := SQL(SELECT getSelect(Q) FROM getFrom(Q) )
2: if t ∉⊥ S
3:    for (R AS S) in (getFrom(Q)) do
4:       s = generateUndefined(R)
5:       for i=1 to length(getSelect(Q)) do
6:          if t_i ≠⊥ and member(getSelect(Q),i) = S.A, A attrib.,  s(R.A) = t_i
7:       end for
8:       if s ∉⊥ SQL(R)
9:          P := P ∪ { (buggy(V) ← state((s ∈ R), nonvalid).) }
10:      end if
11:   end for
12: end if
13: return  P
```

Code 5 wrongBasic(V,Q,t)

Input: V: view name, Q: query, t: tuple
Output: A set of clauses

```
1: P := ∅
2: F := getFrom(Q)
3: N := length(F)
4: for i=1 to N do
5:    R_i as S_i := member(F,i)
6:    relevantTuples(R_i,S_i,V_i, Q, t)
7: end for
8: P := P ∪ { (buggy(V) ← state((V_1 ⊆ R_1), valid), ..., state((V_n ⊆ R_n), valid).) }
9: return  P
```

Teorema C.2.6. *Let* R *be a relation. Then:*

- *Completeness.Let A be the answer obtained after the call to* askOracle(all R) *in line 1 of Code 1. If A is of the form* nonvalid, wrong(t) *or* missing(t)*, then the call* debug(R) *(defined in Code 1) returns a list L containing at least one relation.*
- *Soundness. Let* L *be the list returned by the the call* debug(R)*. Then all relation names contained in* L *are erroneous relations.*

The algorithm presented in this Section has been implemented in the Datalog Educational System (DES [101, 100]). The current implementation of our proposal, including instructions about how to use it, can be downloaded from:

`https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/DesSQL`

This technique minimizes the main problem of declarative debugging when applied directly to SQL views, namely the huge number of tuples that the user must consider in

order to determine the validity of the result produced by a relation. Using a technique similar to dynamic slicing [3], we concentrate only in those tuples produced by the intermediate relations that are relevant for the error. The proposed algorithm looks for particular but common error sources, like tuples missed in the from section or in and conditions (that is, intersect components in our representation). If such shortcuts are not available, or if the user only answers *yes* and *no*, then the tools works as a pure declarative debugger. A more general contribution of the proposed technique is the idea of representing a declarative debugging computation tree by means of a set of logic clauses. In fact, the algorithm in Code 1 can be considered a general debugging schema, because it is independent of the underlying programming paradigm.

A more detailed explanations about these proposals for debugging SQL views, including examples and the proofs of the theoretical results can be found in [36, 37, 38].

C.3. Semistructured databases

In the last years the eXtensible Markup Language XML [117] has become the *de facto* standard for exchanging structured data in plain text files. This was the key for its success as data structures are revealed and therefore they are readily available for its processing (even with usual text editors if one wishes to manually edit them). Structured data means that new, more involved access methods must be devised. XQuery [46, 121] has been defined as a query language for finding and extracting information from XML documents. It extends XPath [50], a domain-specific language that has become part of general-purpose languages. Due to its acknowledged importance, XML and its query languages have been embodied in many applications as in database management systems, which include native support for XML data and documents both in data representations and query languages (e.g., Oracle and SQL Server).

In this Section we present a programming framework for incorporating XPath and XQuery queries into the functional-logic language $\mathcal{TOY}$. The proposal exploits the language characteristics, including non-determinism, logic variables, and higher-order functions and patterns. We use these properties for tracing and debugging queries. From the point of view of functional-logic programming, the language is now able to deal with XML documents in a very simple way. From the point of view of XML, our approach presents several nice properties as the generation of XML test cases for XPath and XQuery queries, which can be useful for finding bugs in erroneous queries.

In the case of XPath, we represented queries by higher-order functions connected by higher-order combinators. Using this approach, an XPath query becomes at the same time implementation (code) and representation (data term). This approach allows to generate XML test cases for XPath queries, as well as to debug and trace erroneous queries.

In the case of XQuery, it is possible to consider XQuery expressions as higher order patterns, in order to manipulate XQuery programs by means of $\mathcal{TOY}$. However, we show that XQuery constructors can be encoded in $\mathcal{TOY}$ by means of (first-order) functions. This is a completely declarative proposal for integrating part of XQuery in $\mathcal{TOY}$, which restricts itself to the completely declarative features of the language. The

advantage of restricting to the purely declarative view is that proofs of correctness and completeness are provided. In [9] we take a different point of view, trying to define a more general XQuery framework although using non-purely declarative features as the (meta-)primitive `collect`.

In next Section we introduce the language $\mathcal{TOY}$ and its semantics. Additionally we present a representation of XML documents in $\mathcal{TOY}$.

C.3.1. The Functional-Logic Language $\mathcal{TOY}$

A $\mathcal{TOY}$ [84] program is composed of data type declarations, type alias, infix operators, function type declarations and defining rules for functions symbols. The syntax of (total) *expressions* in $\mathcal{TOY}$ $e \in Exp$ is $e ::= X \mid h \mid (e\ e')$ where X is a variable and h either a function symbol or a data constructor. Expressions of the form $(e\ e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument). Similarly, the syntax of (total) *patterns* $t \in Pat \subset Exp$ can be defined as $t ::= X \mid c\ t_1\ \ldots\ t_m \mid f\ t_1\ \ldots\ t_m$ where X represents a variable, c a data constructor of arity greater or equal to m, and f a function symbol of arity greater than m, while the t_i are partial patterns for all $1 \leq i \leq m$. Each rule for a function f in $\mathcal{TOY}$ has the form:

$$\underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \rightarrow \underbrace{r}_{\text{right-hand side}} \Leftarrow \underbrace{e_1, \ldots, e_k}_{\text{condition}} \text{ where } \underbrace{s_1 = u_1, \ldots, s_m = u_m}_{\text{local definitions}}$$

where e_i, u_i and r are expressions (that can contain new extra variables) and t_i, s_i are patterns.

The constructor-based ReWriting Logic (CRWL) [88] has been proposed as a suitable declarative semantics for functional-logic programming with lazy non-deterministic functions. The calculus is defined by five inference rules (see Figure C.5): (BT) that indicates that any expression can be approximated by bottom, (RR) that establishes the reflexivity over variables, the decomposition rule (DC), the (JN) (join) rule that indicates how to prove strict equalities, and the function application rule (FA).

In every inference rule, $e, e_i \in Exp_\perp$ are partial expressions and $t_i, t, s \in Pat_\perp$ are partial patterns. The notation $[P]_\perp$ in the inference rule *FA* represents the set $\{(l \rightarrow r \Leftarrow C)\theta \mid (l \rightarrow r \Leftarrow C) \in P,\ \theta \in Subst_\perp\}$ of partial instances of the rules in the program P. The most complex inference rule is *FA* (Function Application), which formalizes the steps for computing a *partial pattern* t as approximation of a function call $f\ \bar{e}_n$:

1. Obtain partial patterns t_i as suitable approximations of the arguments e_i.
2. Apply a program rule $(f\ \bar{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp$, verify the condition C, and check that t approximates the right-hand side r.

In this semantic notation, local declarations $a = b$ introduced in $\mathcal{TOY}$ syntax by the reserved word `where` are represented as part of the condition C as *approximation*

BT $\quad e \to \bot$

RR $\quad X \to X \qquad$ with $X \in Var$

DC $\quad \dfrac{e_1 \to t_1 \ \ldots \ e_m \to t_m}{h\ \overline{e}_m \to h\ \overline{t}_m} \qquad h\ \overline{t}_m \in Pat_\bot$

JN $\quad \dfrac{e \to t \quad e' \to t}{e \ == \ e'} \qquad t \in Pat$ (total pattern)

FA $\quad \dfrac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad C \quad r\ \overline{a}_k \to t}{f\ \overline{e}_n\ \overline{a}_k \ \to t}$
if $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\bot, t \neq \bot$

Figure C.5: CRWL Semantic Calculus.

statements of the form $b \to a$. Infix operators are allowed as particular case of program functions. Consider for instance the definitions:

```
infixr 30 /\                infixr 30 \/          infixr 45 ?
false /\ X = false          true \/ X = true      X ? _Y = X
true /\ X = X               false \/ X = X        _X ? Y = Y
```

The /\ and \/ operators represent the standard conjunction and disjunction, respectively, while `?` represents the non-deterministic choice. For instance the infix declaration `infixr 45 ?` indicates that `?` is an infix operator that associates to the right (the r in `infixr`) and that its priority is 35. The priority is used to assume precedences in the case of expressions involving different operators. Computations in $\mathcal{TOY}$ start when the user inputs some goal as

```
Toy> 1 ? 2 ? 3 ? 4 == R
```

This goal asks $\mathcal{TOY}$ for values of the logical variable R that make true the (strict) equality `1 ? 2 ? 3 ? 4 == R`. This goal yields four different answers `{R` $\mapsto$ `1 }`, `{R` $\mapsto$ `2 }`, `{R` $\mapsto$ `3 }`, and `{R` $\mapsto$ `4 }`. The next function extends the choice operator to lists:

```
member [X|Xs] = X ? member Xs
```

```
data xmlNode    = xmlText    string
                | xmlComment string
                | xmlTag     string [xmlAttribute] [xmlNode]
data xmlAttribute = xmlAtt     string string
type xml        = xmlNode
```

Figure C.6: XML documents in $\mathcal{TOY}$.

For instance, the goal `member [1,2,3,4] == R` has the same four answers that were obtained by trying `1 ? 2 ? 3 ? 4 == R`. Types do not need to be annotated explicitly by the user, they are inferred by the system, which rejects ill-typed expressions. As usual in functional programming languages, $\mathcal{TOY}$ allows partial applications in expressions and higher order parameters like `apply F X = F X`.

A particularity of $\mathcal{TOY}$ is that partial applications with pattern parameters are also valid patterns. They are called *higher-order patterns*. For instance, a program rule like `foo (apply member) = true` is valid, although `foo (apply member []) = true` is not because `apply member []` is a reducible expression and not a valid pattern. Higher-order variables and patterns play an important role in our setting. Functional-logic programming share with logic programming the possibility of using logic variables as parameters. For instance, the goal `member L == 3` asks for lists containing the value `3`. The first solution is `L ->[ 3 | _A ]`, which indicates that `L` can be a list starting by `3` and followed by any list (represented by the anonymous variable `_A`). The second answer is `L ->[ _A, 3 | _B ]`, indicating that `3` can be the second element of the list as well. In this way a (potentially) infinite number of answers can be obtained. The possibility of generating values for the parameters is employed for generating test cases.

Data type declarations and type alias are useful for representing XML documents in $\mathcal{TOY}$ (see Figure C.6). The data type `node` represents nodes in a simple XML document. It distinguishes three types of nodes: texts, tags (element nodes), and comments, each one represented by a suitable data constructor and with arguments representing the information about the node. For instance, the constructor `tag` includes the tag name (an argument of type `string`) followed by a list of attributes, and finally a list of child nodes. The data type `attribute` contains the name of the attribute and its value (both of type `string`). The last type alias, `xml`, renames the data type `node`. Figure C.7 shows an XML document and its representation in $\mathcal{TOY}$.

$\mathcal{TOY}$ includes two primitives for loading and saving XML documents, called `load_xml_file` and `write_xml_file` respectively. For convenience all the documents are started with a dummy node `root`. If the file contains only one node `N` at the outer level, `root` can be removed defining the following simple function:

```
<?xml version='1.0'?>
<food>
 <item type="fruit">
   <name>watermelon</name>
   <price>32</price>
 </item>
 <item type="fruit">
   <name>oranges</name>
   <variety>navel</variety>
   <price>74</price>
 </item>
 <item type="vegetable">
  <name>onions</name>
  <price>55</price>
 </item>
 <item type="fruit">
  <name>strawberries</name>
  <variety>alpine</variety>
  <price>210</price>
 </item>
</food>
```

```
xmlTag "root" [xmlAtt  "version"  "1.0"] [
xmlTag "food"  [][
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "watermelon"],
    xmlTag "price" [][xmlText "32" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"][
    xmlTag "name" [][xmlText "oranges" ],
    xmlTag "variety" [][xmlText "navel"],
    xmlTag "price" [][xmlText "74" ]
  ],
  xmlTag "item" [xmlAtt "type" "vegetable"][
    xmlTag "name" [][xmlText "onions" ],
    xmlTag "price" [][xmlText "55" ]
  ],
  xmlTag "item" [xmlAtt "type" "fruit"] [
    xmlTag "name" [][xmlText "strawberries"],
    xmlTag "variety" [][xmlText "alpine"],
    xmlTag "price" [][xmlText "210" ]
  ]
 ]]
```

Figure C.7: XML example (left) and its representation in $\mathcal{TOY}$ (right).

```
load_doc F = N <== load_xml_file F == xmlTag "root" [xmlAtt "version"
"1.0"] [N]
```

where `F` is the name of the file containing the document. If the strict equality `==` in the condition succeeds, `N` is returned.

C.3.2. XPath in $\mathcal{TOY}$

Typically, XPath expressions return several fragments of the XML document. Thus, the expected type in a functional language for `xPath` could be `type xPath = xml ->[xml]` meaning that a list or sequence of results is obtained. However, in our case we take advantage of the non-deterministic nature of our language, returning each result individually. We define an XPath expression as a function taking a (fragment of) XML as input and returning a (fragment of) XML as its result: `type xPath = xml ->xml`. In order to apply an XPath expression to a particular document, we use the following infix operator definition:

```
infix  20 <--
```

```
self :: xPath
self X = X

child :: xPath
child (xmlTag _Name _Attr L) = member L

descendant :: xPath
descendant X = Y ? descendant Y <== Y == child X

descendant_or_self :: xPath
descendant_or_self = self ? descendant
```

Figure C.8: XPath axes in $\mathcal{TOY}$.

```
(<--)::string  -> xPath ->  xml
Doc <-- Q = Q (load_xml_file Doc)
```

The input arguments of this operator are a string `Doc` representing the file name and an XPath query `Q`. The function applies `Q` to the XML document contained in file `Doc`. The XPath combinators `/` and `::` which correspond to the connection between steps and between axis and tests, respectively, are defined in $\mathcal{TOY}$ as function composition:

```
infixr 55 .::.
(.::.) :: xPath -> xPath -> xPath
(F .::. G) X  = G (F X)

infixr 40 ./.
(./.) :: xPath -> xPath -> xPath
(F ./. G) X = G (F X)
```

The function operator names `.::.` and `./.` are employed because the standard XPath separators `::` and `/` are already defined in $\mathcal{TOY}$ with a different meaning. Notice that the two definitions are the same since they stand for the application of an XPath expression to another XPath expression and return also an XPath expression, although they are intended to be applied to different fragments of XPath: `./.` for steps and `.::.` for combining axes and tests producing steps. The variable `X` represents the input XML fragment (the context node). The rules specify how the combinator applies the first XPath expression (`F`) followed by the second one (`G`). Figures C.8 and C.9 show the $\mathcal{TOY}$ definition of XPath main axes and tests.

```
nodeT :: xPath
nodeT X = X

nameT :: string ->xPath
nameT S (xmlTag S Att L) = xmlTag S Att L

textT :: string ->xPath
textT S (xmlText S) = xmlText S

commentT :: string ->xPath
commentT S (xmlComment S) = xmlComment S

elem :: xPath
elem = nameT _
```

Figure C.9: XPath tests in $\mathcal{TOY}$.

The first one is `self`, which returns the context node. In our setting, it corresponds simply to the identity function. A more interesting axis is `child` which returns, using the non-deterministic function `member`, all the children of the context node. Observe that in XML only *element nodes* have children, and that in the $\mathcal{TOY}$ representation these nodes correspond to terms rooted by constructor `tag`. Once `child` has been defined, `descendant` and `descendant-or-self` are just generalizations.

With respect to test nodes, the first test defined in Figure C.9 is `nodeT`, which corresponds to `node()` in the usual XPath syntax. This test is simply the identity. For instance, here is the XPath expression that returns all the nodes in an XML document, together with its $\mathcal{TOY}$ equivalent:

```
XPath → doc("food.xml")/descendant-or-self::node()
TOY → ("food.xml" <-- descendant_or_self.::.nodeT) == R
```

The only difference is that the $\mathcal{TOY}$ expression returns one result at a time in the variable `R`, asking the user if more results are needed.

XPath abbreviated syntax allows the programmer to omit the axis `child::` from a location step when it is followed by a name. Thus, a query of the form `child::food/child::price/child::item` becomes in XPath simply as `food/price/item`. In $\mathcal{TOY}$ we cannot do that directly because we are in a typed language and the combinator `./.` expects XPath expressions and not strings. However, we can introduce a similar abbreviation by defining new unitary operators `name` (and similarly `text`), which transform strings into XPath expressions:

```
name :: string -> xPath
name S = child.::.(nameT S)
```

So, we can write in $\mathcal{TOY}$ a query of the form `name "food"./.name "item"./.name "price"`. Other tests as `nameT` and `textT` *select* fragments of the XML input, which can be returned in a logical variable.

Optionally, XPath tests can include a predicate or filter. Filters in XPath are enclosed between square brackets. In $\mathcal{TOY}$, they are enclosed between round brackets and connected to its associated XPath expression by the operator `.#`:

```
infixr  60 .#
(.#) :: xPath -> xPath -> xPath
(Q .# F)  X =  if F Y == _ then Y  where Y = Q X
```

This definition can be understood as follows: first the query `Q` is applied to the context node `X`, returning a new context node `Y`. Then the `if` condition checks whether `Y` satisfies the filter `F`, simply by checking that `F Y` does not fail, which means that it returns some value represented by the anonymous variable in `F Y == _`.

A description of how to download and install the $\mathcal{TOY}$ system including the source code of the XPath library, and a description of some extensions like the `ancestor` axis, position filters, and more, can be found at [34]. This representation of XPath allows to generate test cases for XPath queries in a simple way.

C.3.2.1. Generating Test Cases for XPath Expressions

Sometimes it is useful to have a test case, i.e., an XML file which contains some answer for the query. Comparing the test case and the original XML document can help to find the error. In our setting, such test cases are obtained for free. For instance, the following XPath query:

```
Toy> "food.xml" <-- (name "food" ./. name "item" ./. name "type" ./.
                     child.::.textT "navel")  == R
```

falls and returns no answer. For instance, we can submit the goal:

```
Toy> (name "food" ./. name "item" ./. name "type" ./.
       child.::.textT "navel") X  == _
```

asking for an XML document `X` such that the query succeeds. The anonymous variable at the right-hand side of the strict equality indicates that we are not interested in the output. However, the answer is difficult to read and understand:

```
{ X -> (xmlTag _A _B
         [ (xmlTag  "food" _C
             [ (xmlTag  "item" _D
                 [ (xmlTag  "type" _E
                   [ (xmlText  "navel") | _F ]) | _G
                 ]) | _H
             ]) | _I
        ])
```

The logic variables indicate that replacing them by any valid XML fragment produces a valid XML test case for the query. In particular, in the case of lists, they indicate that other elements can be added, and the smaller test case corresponds to substituting these variables by the empty list. In order to enhance the readability of the result we define a function:

```
generateTC:: xPath -> string -> bool
generateTC Q S = if (Q X == _) then  write_xml_file X S
```

This function receives the XPath expression and the file name `S` as input parameters, looks for an XML test case `X`, and writes it to the file using the primitive `write_xml_file`. The goal:

```
Toy> generateTC (name "food" ./. name "item" ./. name "type"
             ./. child.::.textT "navel") "tc.xml" == R
```

produces the following XML file "tc.xml":

```
<food>
    <item>
       <type>navel</type>
   </item>
</food>
```

It is worth noticing that the primitive has replaced the logic variables by empty elements. Comparing this file and our example `"food.xml"`, we see that `"type"` is an attribute, but not a child node.

XPath queries are represented in this setting by non-deterministic higher-order expressions, thus becoming first-class citizens of the language that can be readily extended and adapted by the programmer. The *Higher-order patterns* of $\mathcal{TOY}$ allows us to consider $\mathcal{TOY}$ expressions that are not yet reducible as patterns. This means in our case that XPath expressions can be considered at the same time executable code (when applied to an input XML document), or data structures when considered as higher-order patterns. We use this powerful characteristic for tracing and debugging queries. We distinguish two types of possible errors in an XPath Query depending on the erroneous result produced: *wrong* queries when the query returns an unexpected result, and *missing* when the query does not produce some expected result. We present a different proposal depending on the error.

C.3.2.2. Wrong XPath Queries

Consider for instance the goal:

```
Toy> "bib.xml" <-- ( name "bib" ./. name "book" ./.
                     name "author" ./. name "last" ) == R
```

and suppose that it produces the unexpected answer:

```
R -> (tag "last" [] [ (txt "Abiteboul") ])
```

See Figure C.10 to check the structure of this document [118]. If the error is just some misspelling of the author's last name it is easy to look for the wrong information in the document in order to correct the error. However in some situations, and in particular when dealing with complicated, large documents, the error can be in the XPath query, that has selected an erroneous path in the document. In these cases it is also useful to find the answer in the document and then trace back the XPath query until the error is found. Observe that the erroneous answer can be just one of the produced answers (in the example the query can produce many other, expected, answers), and that we are interested only in those portions of the document that produce this unexpected result.

In $\mathcal{TOY}$ we can obtain each intermediate step with its associated answer by defining a suitable function `wrong` that receives three arguments: the query, its input (initially the whole document) and the unexpected output (initially the unexpected answer). The implementation is straightforward:

```
<?xml version='1.0'?>
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
        <author><last>Stevens</last><first>W.</first></author>
        <publisher>Addison-Wesley</publisher>
        <price>65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
    </book>

    <book year="1999">
        <title>The Economics of Technology and Content for Digital TV</title>
        <editor>
               <last>Gerbarg</last><first>Darcy</first>
                <affiliation>CITI</affiliation>
        </editor>
            <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>

</bib>
```

Figure C.10: XML document: bib.xml.

```
wrong (A.::.B) I O = [((A.::.B), I, O)] <== (A.::.B) I == O
wrong (A./.B) I O  = [(A, I, O1 ) | wrong B O1 O]
                       <== A I == O1, B O1 == O
```

In the case of a single step `(A.::.B)` the first rule checks that indeed the step applied to the input produces the output returning the three elements. In the case of two or more steps, the second rule looks for the value `O1` produced by the single step `A` such that the rest of the query `B` applied to `O1` produces the erroneous result `O`. The variable `O1` is a new logic variable, and that the code uses the *generate and test* feature typical of functional languages. The function `wrong` produces a list where

each step is associated with its input and its output. However, using `wrong` directly produces a verbose, difficult to understand output due to the representation of XML elements as a data terms in $\mathcal{TOY}$. This can be improved by building a new XML document containing all the information and saving it to a file using the primitive `write_xml_file`:

```
traceStep (Step,I,O) = xmlTag "step" [] [ xmlTag "query" []
                       [xmlText (show Step)],
                        xmlTag "input" [] [I],
                        xmlTag "output" [] [O] ]

generateTrace L = xmlTag "root" [] (map traceStep (rev L))

writeTrace XPath InputFile WrongOutput OutputFile =
      write_xml_file (
           generateTrace (
              wrong XPath (
                 load_xml_file InputFile)
              WrongOutput))
           OutputFile

show (A.::.B)    = (show A)++".::."++(show B)
show nodeT       = "nodeT"
...
show child      = "child"
...
```

The first function `traceStep` generates an XML element `step` containing the information associated with an XPath step: the step combinator, its input, and its output. This function uses the auxiliary function `show` to obtain the string representation of the step (only part of the code of this function is displayed). Then function `generateTrace` applies `traceStep` to a list of steps. It uses the functions `map` and `rev` whose definition is the same as in functional programming. In particular, `rev` is employed to ensure that the last step of the query is the first in the document, which is convenient for tracing back the result. Finally `writeTrace` combines the previous functions. It receives four parameters: the query, the name of the initial document, the unexpected XML fragment we intend to trace, and the name of the output file where the result is saved. Now we can try the goal:

```
Toy> writeTrace (name "bib" ./. name "book" ./. name "author" ./.
                 name "last" ) "bib.xml"
                 (tag  "last" [] [txt  "Abiteboul"])
                 "trace.xml"
```

In this case the symbol `==` is not used which indicates to $\mathcal{TOY}$ that the result of this expression must be true. After the goal is solved, the document `"trace.xml"` contains all the information that allows the user to trace the query.

C.3.2.3. Missing XPath Queries

Sometimes an XPath query produces no answer, although some result was expected. For instance the goal:

```
   "food.xml" <-- name "food"./. name "item" ./. name "type" ./.
child.::.textT "navel" == R
```

simply fails in $\mathcal{TOY}$. The reason is that the user wrote `type` where it should be `variety`. This error is very common, and the source of the error can be difficult to detect. For these situations we propose trying the XPath query without the last step, then if it fails without the last two steps and so on. The idea is to find the first step that produces an empty result, because it is usually the source of the error.

```
missing  (A.::.B) R = (A.::.B)
missing (X ./. Y) R = if (collect (X R) == []) then X
                      else missing Y (X R)
```

The previous definition of function `missing` relies on the primitive `collect` that accumulates all the results of a function call in a list. Therefore `collect (X R) == []` means that `X` applied to the input XML fragment `R` fails. Now we can try the goal:

```
Toy> missing (name "food" ./. name "item" ./. name "type" ./.
              child.::.textT "navel") (load_xml_file "food.xml") == R
    { R -> child .::. (nameT  "type") }
```

The answer indicates that the step `child .::. (nameT "type")` is the possible source of the error. Thus this simply function is useful, but we can do better. Instead of simply returning the erroneous step we can try to *guess* how the error can be corrected. In the case of *name* tests as the example the error is usually the same erroneous string has been used. Replacing the string by a logic variable such that the query now succeeds can help to find the error. Therefore, we implement a second version of `missing`:

```
missing (A.::.B) R =  guess (A.::.B) self R

missing (Step ./. Y) R = if (collect (Step R) == [])
                         then guess Step Y R
                         else missing Y (Step R)

guess Step Y R = if Step==(A.::.nameT B)
                 then if (StepBis ./. Y) R == _
                      then ( Step, "Substitute "++B++" by "++C )
                      else (Step, "No suggestion")
                 else (Step, "No suggestion")
    where StepBis = (A.::.nameT C)
```

In this case `missing` returns a pair. The first element of the pair is the same as in the first version, and the second element is a suggestion produced by function `guess`. This function first checks if the `Step` is of the form `(A.::.nameT B)`. If this is the case, it replaces the name `B` by a new variable `C` and uses the condition in the second `if` statement (`(StepBis ./. Y) R == _`) to check if `C` can take any value such that the query does not fail. If such value for `C` is found the returned string proposes replacing `B` by `C`. Otherwise `"No suggestion"` is returned. Since function `guess` requires an argument `Y` with the rest of the XPath query, the basis case of `missing` (first rule) uses `self` to represent the identity query. Now we can try:

```
Toy>  missing (name "food" ./. name "item" ./. name "type" ./.
               child.::.textT "navel") (load_xml_file "food.xml") == R
{ R -> (child .::. (nameT  "type"),  "Substitute type by variety") }
```

Thus, the debugger finds the erroneous step and proposes the correct solution.

The use of *logic variables*, specially when used in *generate and test* expressions are very suitable for obtaining the values of intermediate computations, and in our case also for guessing values in the debugger of missing answers.

C.3.3. XQuery in $\mathcal{TOY}$

XQuery allows the user to query several documents, applying join conditions, generating new XML fragments, and many other features [46, 121]. The syntax and semantics of the language are quite complex, and XQuery expressions can not be represented easily by means of higher-order expressions. In this Section we describe a completely declarative proposal for integrating part of XQuery in $\mathcal{TOY}$, which restricts itself to the completely declarative features of the language. The advantage of restricting to

$$
\begin{array}{rcl}
query & ::= & query\ query \mid tag \\
 & \mid & var \mid var/axis :: test \\
 & \mid & \textbf{for}\ var\ \textbf{in}\ query\ \textbf{return}\ query \\
 & \mid & \textbf{if}\ cond\ \textbf{then}\ query \\
 & & \\
cond & ::= & var = var \mid query \\
tag & ::= & \langle a\rangle\ \langle/a\rangle \mid \langle a\rangle var \ldots var \langle/a\rangle \mid \langle a\rangle tag \langle/a\rangle \\
 & & \\
axis & ::= & \textbf{self} \mid \textbf{child} \mid \textbf{descendant} \mid \textbf{dos} \mid \ldots
\end{array}
$$

Figure C.11: Syntax of SXQ, a simplified version of XQ.

the purely declarative view is that proofs of correctness and completeness are provided. Our setting deals with a simple fragment of XQuery, including *for* statements for traversing XML sequences, *if/where* conditions, and the possibility of returning XML elements as results. Some basic XQuery constructions such as *let* statements are not considered, but we think that the proposal is powerful enough for representing many interesting queries.

In [22] a declarative subset of XQuery, called XQ, is presented. This subset is a core language for XQuery expressions consisting of *for*, *let* and *where/if* statements. We consider a simplified version of XQ, which we call SXQ and whose syntax can be found in Figure C.11. In this grammar, a denotes a label and ν refers to a *label test* which is a label.

Consider an XML file "bib.xml" containing data about books, and another file "reviews.xml" containing reviews for some of these books. See Figures C.10 and C.12 to check the structure of these documents. Then, we can list the reviews corresponding to books in "bib.xml" as follows:

```
for $b in doc("bib.xml")/bib/book,
    $r in doc("reviews.xml")/reviews/entry
where $b/title = $r/title
for $booktitle in $r/title,
    $revtext in $r/review
return <rev> $booktitle  $revtext </rev>
```

The variable $\$b$ takes the value of each different book, and $\$r$ represents the different reviews. The last two variables are only employed to obtain the book title and the text of the review, the two values that are returned as output of the query by the *return* statement. It can be argued that the code of this example does not follow the syntax of Figure C.11. While this is true, it is very easy to define an algorithm that converts a query formed by *for*, *where* and *return* statements into a SXQ query (as long as it only includes variables inside tags, as stated above). The idea is simply to replace the references to XML documents by new indexed variables $\$x_1, \$x_2, \ldots,$

```
<?xml version='1.0'?>
<reviews>
    <entry>
        <title>Data on the Web</title>
        <price>34.95</price>
        <review>
               A very good discussion of semi-structured database
               systems and XML.
        </review>
    </entry>
    <entry>
        <title>Advanced Programming in the Unix environment</title>
        <price>65.95</price>
        <review>
               A clear and detailed discussion of UNIX programming.
        </review>
    </entry>
    <entry>
        <title>TCP/IP Illustrated</title>
        <price>65.95</price>
        <review>
               One of the best books on TCP/IP.
        </review>
    </entry>
</reviews>
```

Figure C.12: XML document : reviews.xml.

and convert the *where* into *if*s, following each *for* by a *return*, and decomposing XPath expressions including several steps into several *for* expressions by introducing a new auxiliary variables, each one consisting of a single step:

```
for $x3 in $x1/child::bib return
for $x4 in $x3/child::book  return
for $x5 in $x2/child::reviews return
for $x6 in $x5/child::entry  return
for $x7 in $x4/child::title return
for $x8 in $x6/child::title return
if ($x7 = $x8) then
   for $x9 in $x6/child::title return
     for $x10 in $x6/child::review return <rev> $x9  $x10 </rev>
```

Notice that the expressions doc("bib.xml") and doc("reviews.xml") have been substituted by the variables `$x1` and `$x2` respectively. Both variables are free in the query and the value of each one is the XML document contained in the corresponding XML file. We assume that XML documents must be accessed initially via indexed variables $\$x_1, \$x_2, \ldots$ belonging to the set $free(Q)$. The concept of set of *free* variables of a SXQ query is given by the following inductive definition:

Definición C.3.1. *Let Q be a SXQ query. The set of free variables of Q, denoted by free(Q), is defined as follows:*

- *If* $Q \equiv Q_1\ Q_2$, *then* $free(Q) := free(Q_1) \cup free(Q_2)$
- *If* $Q \equiv$ for $\$x$ in Q_1 return Q_2, *then* $free(Q) := (free(Q_1) \cup free(Q_2)) \setminus \{\$x\}$
- *If* $Q \equiv$ if $\$x_i = \$x_j$ then Q_2, *then* $free(Q) := \{\$x_i, \$x_j\} \cup free(Q_2)$
- *If* $Q \equiv$ if Q_1 then Q_2, *then* $free(Q) := free(Q_1) \cup free(Q_2)$
- *If* $Q \equiv \$x$, *then* $free(Q) := \{\$x\}$
- *If* $Q \equiv \$x/axis :: \nu$, *then* $free(Q) := \{\$x\}$
- *If* $Q \equiv \langle a\rangle\langle/a\rangle$, *then* $free(Q) := \emptyset$
- *If* $Q \equiv \langle a\rangle tag\langle/a\rangle$, *then* $free(Q) := free(tag)$
- *If* $Q \equiv \langle a\rangle \$x_i \ldots \$x_j \langle/a\rangle$, *then* $free(Q) := \{\$x_i, \ldots, \$x_j\}$

The semantics of XQ can be found in [22]. We will use XML documents represented as *data trees*. A *data forest* is a sequence of data trees and an *indexed forest* is a pair consisting of a data forest and a sequence of nodes in it. Figure C.13 introduces the operational semantics of an SXQ expression α with at most k free variables using a function $[\![\alpha]\!]_k$ that takes a data forest $\mathcal{F}$ and a $k-$tuple of nodes from the forest as input and returns an indexed forest.

$$
\begin{array}{rcl}
[\![\alpha\ \beta]\!]_k(\mathcal{F}, \overline{e}) & := & [\![\alpha]\!]_k(\mathcal{F}, \overline{e}) \uplus [\![\beta]\!]_k(\mathcal{F}, \overline{e}) \\
[\![\text{for } \$x_{k+1} \text{ in } \alpha \text{ return } \beta]\!]_k(\mathcal{F}, \overline{e}) & := & \text{let } (\mathcal{F}', \bar{l}) = [\![\alpha]\!]_k(\mathcal{F}, \overline{e}) \text{ in} \\
 & & \quad \biguplus_{1 \le i \le |\bar{l}|} [\![\beta]\!]_{k+1}(\mathcal{F}', \overline{e} \cdot \bar{l}_i) \\
[\![\$x_i]\!]_k(\mathcal{F}, t_1, \ldots, t_k) & := & (\mathcal{F}, [t_i]) \\
[\![\$x_i/\chi :: \nu]\!]_k(\mathcal{F}, t_1, \ldots, t_k) & := & (\mathcal{F}, \text{list of nodes } v \text{ such that } \chi^{\mathcal{F}}(t_i, v) \text{ and} \\
 & & \quad \text{node } v \text{ has label } \nu \text{ in order } <_{doc}^{tree(t_i)}) \\
[\![\text{if } \phi \text{ then } \alpha]\!]_k(\mathcal{F}, \overline{e}) & := & \text{if } \pi_2([\![\phi]\!]_k(\mathcal{F}, \overline{e})) \neq [\,] \text{ then } [\![\alpha]\!]_k(\mathcal{F}, \overline{e}) \\
 & & \text{else } (\mathcal{F}, [\,]) \\
[\![\text{if } \$x_i = \$x_j \text{ then } \alpha]\!]_k(\mathcal{F}, t_1, \ldots, t_k) & := & \text{if } t_i = t_j \text{ then } [\![\alpha]\!]_k(\mathcal{F}, t_1, \ldots, t_k) \\
 & & \text{else } (\mathcal{F}, [\,]) \\
[\![\langle a\rangle\ \langle/a\rangle]\!]_k(\mathcal{F}, \overline{e}) & := & construct(a, (\mathcal{F}, [\,])) \\
[\![\langle a\rangle\ tag\ \langle/a\rangle]\!]_k(\mathcal{F}, \overline{e}) & := & construct(a, [\![tag]\!]_k(\mathcal{F}, \overline{e})) \\
[\![\langle a\rangle\ \$x_i\ \ldots\ \$x_j\ \langle/a\rangle]\!]_k(\mathcal{F}, t_1, \ldots, t_k) & := & construct(a, (\mathcal{F}, [t_i, \ldots, t_j]))
\end{array}
$$

Figure C.13: Semantics of SXQ.

This semantics makes use of some functions that construct indexed forest. The operator $construct(a, (\mathcal{F}, [w_1 ... w_n]))$, denotes the construction of a new tree, where a is a label, $\mathcal{F}$ is a data forest, and $[w_1 \ldots w_n]$ is a list of nodes in $\mathcal{F}$. When applied,

construct returns an indexed forest $(\mathcal{F} \cup T', [root(T')])$, where T' is a data tree with domain a new set of nodes, whose root is labeled with a, and with the subtree rooted at the i-th (in sibling order) child of root(T') being an isomorphic copy of the subtree rooted by w_i in $\mathcal{F}$. The symbol $\uplus$ used in the rules takes two indexed forests $(\mathcal{F}_1, \bar{l}_1), (\mathcal{F}_2, \bar{l}_2)$ where the $\mathcal{F}_i$ are a data forest and $\bar{l}_i$ are lists of nodes in $\mathcal{F}_i$, and returns an indexed forest $(\mathcal{F}_1 \cup \mathcal{F}_2, \bar{l})$, where $\bar{l}$ is the concatenation of $\bar{l}_1$ and $\bar{l}_2$.

For a data tree $\mathcal{F}$, we let the binary relation $<^{\mathcal{F}}_{doc}$ on nodes be the *document-order* on $\mathcal{F}$: the depth-first left-to-right traversal order through $\mathcal{F}$. In the semantics of $\$x_i/\chi :: \nu$ we use $tree(t_i)$ to denote the maximal tree within the input forest that contains the node t_i, hence $<^{tree(t_i)}_{doc}$ is the document-order on the tree containing t_i. $\chi^{\mathcal{F}}$ is the interpretation of the axis relation of the same name in the data forest.

These semantic rules constitute a term rewriting system (TRS in short, see [16]), with each rule defining a single reduction step. The symbol $:=^*$ represents the reflexive and transitive closure of $:=$ as usual. The TRS is terminating and confluent (the rules are not overlapping).

As explained in [22], this semantics does not model the document() function of XQuery. Instead, we assume that there exist one or more initial variables that are each bound to a node of the input forest.

Given a SXQ query Q with $free(Q) = \{\$x_1, \ldots, \$x_k\}$, the semantics evaluates a query Q starting with the expression $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k)$. The initial data forest $\mathcal{F}$ is a forest containing k input XML documents represented as data trees as explained in [22]. Each variable $\$x_i$ in $free(Q)$ represents an initial XML document and it is bound to a node of the input data forest $\mathcal{F}$. The sequence of nodes $t_1, \ldots, t_k$ from $\mathcal{F}$, corresponds to the nodes assigned to the variables $\{\$x_1, \ldots, \$x_k\}$. Along intermediate steps, expressions of the form $[\![Q']\!]_{k+n}(\mathcal{F}', t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+n})$ are obtained. The data forest $\mathcal{F}'$ is built from the input data forest $\mathcal{F}$ by adding (possible) new data trees, which are constructed by the operator *construct* representing new XML fragments.

The evaluation of a query returns as a result an indexed forest as a pair of the form $(\mathcal{F}', [e_1, \ldots, e_m])$ meaning that the query returns a sequence of $m-$nodes from $\mathcal{F}'$ representing XML fragments.

In order to represent SXQ queries in $\mathcal{TOY}$ we use some auxiliary datatypes:

```
data sxq  = xfor xml sxq sxq | xif cond sxq | xmlExp xml |
                    xp path | comp sxq sxq
data cond = xml := xml | cond sxq
data path = var xml | xml :/ xPath | doc string xPath
```

The structure of the datatype `sxq` allows representing any SXQ query. $\mathcal{TOY}$ includes a primitive `parse_xquery` that translates any SXQ expression into its corresponding representation as a term of this datatype. The primitive `parse_xquery` takes as input a SXQ expression Q and returns as ouput the query `Q` represented as a $\mathcal{TOY}$ dataterm. Without loss of generality, in order to simplify our implementation, the primitive `parse_xquery` also allows as input queries without free variables. This

```
sxq (xp E)              = sxqPath E
sxq (xmlExp X)          = X
sxq (comp Q1 Q2)        = sxq Q1
sxq (comp Q1 Q2)        = sxq Q2
sxq (xfor X Q1 Q2)      = sxq Q2 <== X== sxq Q1
sxq (xif (Q1:=Q2) Q3)   = sxq Q3 <== sxq Q1  == sxq Q2
sxq (xif (cond Q1) Q2) = sxq Q2 <== sxq Q1  == _

sxqPath (var X)  = X
sxqPath (X :/ S) = S X
sxqPath (doc F S) = S (load_xml_file F)
```

Figure C.14: $\mathcal{TOY}$ transformation rules for SXQ .

is possible by replacing all the free variables in the SXQ query Q by its corresponding XML files.

The rules of the $\mathcal{TOY}$ interpreter that processes SXQ queries can be found in Figure C.14. The main function is `sxq`, which distinguishes cases depending of the form of the query. If it is an XPath expression then the auxiliary function `sxqPath` is used. If the query is an XML expression, the expression is just returned (this is safe thanks to our constraint of allowing only variables inside XML expressions). If we have two queries (`comp` construct), the result of evaluating any of them is returned using non-determinism. The `for` statement (`xfor` construct) forces the evaluation of the query `Q1` and binds the variable `X` to the result. Then the result query `Q2` is evaluated. The case of the `if` statement is analogous.

Although only a small subset of XQuery consisting only of *for*, *where/if* and *return* statements has been considered, the users of $\mathcal{TOY}$ can now perform simple queries typical of database queries such as *join* operations. The embedding has respected the declarative nature of $\mathcal{TOY}$, and we have provided the soundness of the approach with respect to the operational semantics of XQuery. Next theorem establishes the correctness of the approach.

Teorema C.3.2. *Let P be the $\mathcal{TOY}$ program of Figure C.14, Q a SXQ query with free$(Q) = \{\$x_1, \ldots, \$x_m\}$. Let `Q` be the representation of Q as a $\mathcal{TOY}$ dataterm, `t` be a $\mathcal{TOY}$ pattern, and θ a substitution such that dom(θ) = free(Q) and $\mathcal{P} \vdash$ (`sxq` `Q`θ == `t`). Then, for all data forest $\mathcal{F}$ containing the nodes $t_1, \ldots, t_m$ with $t_1 = \theta(x_1), \ldots, t_m = \theta(x_m)$, there exists an indexed forest $(\mathcal{F}', L')$ such that:*

$$[\![Q]\!]_m(\mathcal{F}, t_1, \ldots, t_m) :=^* (\mathcal{F}', L')$$

verifying $t \in L'$.

Next theorem establishes the completeness of the approach, in the sense that the $\mathcal{TOY}$ program can produce every answer obtained by the SXQ operational semantics.

Teorema C.3.3. *Let P be the $\mathcal{TOY}$ program of Figure C.14. Let Q be a SXQ query with free$(Q) = \{\$x_1, \ldots, \$x_k\}$ and suppose that $[\![Q]\!]_k(\mathcal{F}, t_1, \ldots, t_k) :=^* (\mathcal{F}', L)$ for*

some $\mathcal{F}, \mathcal{F}', t_1, \ldots, t_k, L$. *Then, for every* $t \in L$, *there is a substitution* θ *such that* $\theta(\$x_i) = t_i$ *for all* $\$x_i \in free(Q)$ *and a CRWL-proof proving* $\mathcal{P} \vdash sxq\ Q\theta == \mathtt{t}$.

C.3.3.1. Generating Test-Cases for XQuery Expresions

We show how the embedding of SXQ in $\mathcal{TOY}$ can be used for obtaining test cases for the queries. Suppose that the user wants to include the publisher of the book among the data obtained in the example in previous Section.The following query tries to obtain this information:

```
Q = for $b in doc("bib.xml")/bib/book,
        $r in doc("reviews.xml")/reviews/entry,
        where $b/title = $r/title
        for $booktitle in $r/title,
            $revtext in $r/review,
            $publisher in $r/publisher
        return  <rev> $booktitle $publisher $revtext </rev>
```

However, there is an error in this query, because in the expression `$r/publisher` the variable `$r` should be `$b`, since the publisher is in the document "bib.xml", not in "reviews.xml". The user does not notice that there is an error, tries the query (in $\mathcal{TOY}$ or in any XQuery interpreter) and receives an empty answer.

In order to check whether a query is erroneous, or even to help finding the error, it is sometimes useful to have test cases, i.e., XML files which can produce some answer for the query. Then the test cases and the original XML documents can be compared, and this can help finding the error. In our setting, such test cases are obtained for free, thanks to the generate and test capabilities of logic programming. The general process can be described as follows:

1. Let `Q'` be the translation of the SXQ query Q as a $\mathcal{TOY}$ dataterm by means the primitive `parse_xquery`.

2. Let $F_1, \ldots, F_k$ be the names of the XML documents occurring in `Q'`.

3. Let `Q"` be the result of replacing each expression of the form `doc(`F_i`)` by a new variable D_i, for $i = 1 \ldots k$.

4. Let `"expected.xml"` be a document containing an expected answer for the query `Q`.

5. Try the following goal:

```
prepareTC (xp E)             = (xp E',L)
                               where (E',L) = prepareTCPath E
prepareTC (xmlExp X)         = (xmlExp X, [])
prepareTC (comp Q1 Q2)       = (comp Q1' Q2', L1++L2)
                                where (Q1',L1) = prepareTC Q1
                                      (Q2',L2) = prepareTC Q2
prepareTC (xfor X Q1 Q2)     = (xfor X Q1' Q2', L1++L2)
                               where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2
prepareTC (xif (Q1:=Q2) Q3) = (xif (Q1':=Q2') Q3',L1++(L2++L3))
                                where (Q1',L1) = prepareTC Q1
                                      (Q2',L2) = prepareTC Q2
                                      (Q3',L3) = prepareTC Q3
prepareTC (xif (cond Q1) Q2) = (xif (cond Q1) Q2, L1++L2)
                                where (Q1',L1) = prepareTC Q1
                                     (Q2',L2) = prepareTC Q2

prepareTCPath (var X)   = (var X, [])
prepareTCPath (X :/ S)  = (X :/ S, [])
prepareTCPath (doc F S) = (A :/ S, [write_xml_file A ("tc"++F)])
```

Figure C.15: $\mathcal{TOY}$ test case generation rules for SXQ.

```
Toy> sxq Q'' == load_doc "expected.xml",
                write_xml_file D1 F1',
                ... ,
                write_xml_file Dk Fk'
```

The idea is that the goal above looks for values of the logic variables D_i fulfilling the strict equality. The result is that after solving this goal, the D_i variables contain XML documents that can produce the expected answer for this query. Then each document is saved into a new file with name F_i'. For instance F_i' can consist of the original name F_i preceded by some suitable prefix tc. The process can be automatized, and the result is the code of Figure C.15.

The code uses the list concatenation operator `++` which is defined in $\mathcal{TOY}$ as usual in functional languages such as Haskell. It is worth observing that if there are no test case documents that can produce the expected result for the query, the call to `generateTC` will loop. The next example shows the generation of test cases for the wrong query `Q`. Suppose the user writes the following document "expected.xml":

```
<rev>
```

```
 <title>Some title</title>
 <review>The review</review>
 <publisher>Publisher</publisher>
</rev>
```

This is a possible expected answer for the query. Now we can try the goal:

```
Toy> Q == parse_xquery "for....", R == generateTC Q "expected.xml"
```

The first strict equality parses the query, and the second one generates the XML documents which constitute the test cases. In this example the test cases obtained are:

```
% bibtc.xml
<bib>
 <book>
  <title>Some title</title>
 </book>
</bib>
```

```
% revtc.xml
<reviews>
 <entry>
  <title>Some title</title>
  <review>The review </review>
  <publisher>Publisher</publisher>
 </entry>
</reviews>
```

By comparing the test case "revtc.xml" with the file "reviews.xml" we observe that the publisher is not part of the structure defined for reviews. Then, it is easy to check that in the query the publisher is obtained from the reviews instead of from the *bib* document, and that this constitutes the error.

A more detailed explanations about these proposals for testing XPath and XQuery queries, including examples, a description of some extensions like the `ancestor` axis, position filters, and the proofs of the theoretical results, can be found in [35, 6, 8, 7].

The effort for embedding XPath and XQuery in $\mathcal{TOY}$ is rewarded due to the simplicity for generating test cases automatically when possible, which is useful for testing the query, or even for helping to find the error in the query.