Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Datalog +/-, una interfaz tolerante a la inconsistencia

Tesis de Licenciatura en Ciencias de la Computación

Pablo Víctor Fromer

Director: María Vanina Martinez

Codirector: Ricardo Oscar Rodriguez

Buenos Aires, 2019

Índice general

1. INTRODUCCIÓN

1.1. Motivación

Durante mucho tiempo y en distintas disciplinas dentro de las ciencias de la computación se ha estudiado el problema de cómo manejar información conflictiva o inconsistente. En los últimos años, ha habido un creciente interés en este tema con el advenimiento de la *Web Semántica*, que ha hecho que este asunto sea aún más relevante; teniendo en cuenta que en ambientes abiertos, con fuentes de información proveniente de diversos orígenes, es habitual que surjan contradicciones entre los datos.

Por otro lado, es bien sabido que la inconsistencia también genera problemas en la lógica clásica. En particular, las teorías lógicas de primer orden inconsistentes no tienen modelos, y por lo tanto implican todas las fórmulas; con lo cual, responder consultas en una base de conocimientos inconsistente carece de sentido. Esto llevó a que en el campo de la inteligencia artificial y de la teoría de base de datos, se haya mantenido tradicionalmente la postura de que las bases de conocimiento deberían ser totalmente libres de información inconsistente, y que esta debería ser erradicada inmediatamente. Sin embargo, en las últimas décadas, se ha reconocido que para muchas aplicaciones interesantes aquella postura es obsoleta.

Existen dos enfoques para lidiar con la inconsistencia: el primero consiste en que dada una base de conocimiento inconsistente, se debe arreglarla de una manera optimal; lo cual puede implicar cambios en la teoría lógica subyacente tanto como agregar o quitar sentencias de la base. Este enfoque se caracteriza por tener una “nocion coherentista” en la que nos quedamos con una nueva base de datos que reemplaza a la original. El segundo enfoque se basa en considerar todas las maneras posibles de arreglar esa base de conocimientos *on the fly*, es decir al momento de realizar las consultas. De alguna manera se “convive” con la inconsistencia. En este trabajo nos concentraremos en el segundo enfoque.

Distintas aproximaciones se pueden considerar también al momento de erradicar la consistencia; existen para esto diferentes semánticas, algunas más restrictivas que otras, y con diferente grado también de complejidad computacional. Queremos presentar aquí los diferentes enfoques que se han propuesto en este campo, y presentar también un algoritmo que ayude a resolver el problema de la inconsistencia de manera eficiente. Teniendo en cuenta que la semántica *AR*, es la semántica más aceptada para *query answering* en ontologías potencialmente inconsistentes, nuestra intención es proponer una estrategia algorítmica eficiente que permita responder consultas bajo esta semántica. Por otro lado, nuestra motivación se basa también en presentar una herramienta que le permita al usuario poder explorar distintas semánticas para respuesta a consultas y comparar resultados.

Enmarcaremos este problema en el contexto de *Datalog +/-*, una familia de extensiones de *Datalog*, que se ha propuesto como un nuevo paradigma para responder consultas sobre ontologías. Esta familia de extensiones permite cuantificar variables

existencialmente en la implicaciones lógicas, siendo esto una forma de *generar* nuevo conocimiento, como ejemplificaremos en las siguientes páginas. Además, con algunas restricciones sintácticas que rigen sobre los diferentes *fragmentos* de esta familia, se ha demostrado que se puede lograr tratabilidad y eficiencia computacional, lo cual hace que *Datalog +/-* sea lo suficientemente atractivo para el campo de la inteligencia artificial y las bases de datos. Nos entusiasma por lo tanto presentar una herramienta que no solo permita responder consultas sobre ontologías inconsistentes sino que además acerque *Datalog +/-* al público general de manera amigable.

1.2. Datalog +/-

En esta sección vamos a recordar los elementos principales del lenguaje Datalog tal como se definen en el artículo *A General Datalog-Based Framework for Tractable Query Answering over Ontologies*[8]. Con respecto a los ingredientes elementales, se asumen constantes, nulos y variables de la siguiente manera; estos son los argumentos de las fórmulas atómicas en las bases de datos, consultas y dependencias. Se asume (i) un universo infinito de constantes Δ (las cuáles forman el dominio de la base de datos), (ii) un conjunto infinito de nulos etiquetados Δ_N (representando valores desconocidos), y (iii) un conjunto infinito de variables X (que se utilizan en las consultas y en las dependencias). Cada constante representa un valor diferente, mientras que nulos distintos podrían representar el mismo valor. Se asume un orden lexicográfico en $\Delta \cup \Delta_N$, donde los símbolos de Δ_N siguen a los de Δ. Se denota por X a una sequencia de variables X_1, ..., X_K con $k > 0$.

Se definen las fórmulas atómicas, que ocurren en bases de datos, consultas, y dependencias, y que se construyen en base a los nombres de las relaciones y los términos. Se asume un *esquema relacional* R, el cual constituye un conjunto finito de *nombres de relación*, o *símbolos de predicado*. La posición *P[i]* identifica al *i*-esimo argumento de un predicado *P*. Un *término t* es una constante, un nulo o variable. Una *fórmula atómica* (o *átomo*) *a* tiene la forma $P(t_1,...,t_n)$, donde *P* es un predicado n-ario, y $t_1,...,t_n$ son términos. Se denotan por *pred*(*a*) y *dom(a)* su predicado y al conjunto de sus argumentos, respectivamente. Esta notación se extiende de manera natural para conjuntos y conjunciones de átomos.

Podemos definir ahora la noción de base de datos relativa a un esquema relacional, junto con la sintaxis y la semántica de las consultas conjuntivas y las consultas conjuntivas booleanas a una base de datos. Una *instancia de base de datos* D para un esquema relacional R es un conjunto de átomos (posiblemente infinito) con predicados de R y argumentos de Δ. Una consulta conjuntiva sobre R tiene la forma $Q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, donde $\Phi(\mathbf{X}, \mathbf{Y})$ es una conjunción de átomos con las variables $\mathbf{X}$ e $\mathbf{Y}$, y eventualmente constantes, pero sin nulos. Una consulta conjuntiva *Booleana* sobre R es una consulta conjuntiva de la forma $Q() = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, donde todas las variables están cuantificadas existencialmente. El conjunto de todas las respuestas a una consulta conjuntiva $Q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$ sobre una base de datos es el conjunto de todas las tuplas $\mathbf{t}$ sobre Δ para las cuales existe un homomorfismo $\mu : \mathbf{X} \cup \mathbf{Y} \rightarrow \Delta \cup \Delta_N$ tales que $\mu(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$ y $\mu(\mathbf{X}) = \mathbf{t}$

Ejemplo 1. *Consideremos una base de datos de películas, que guarda información acerca de actores, directores y películas. El esquema relacional R consiste en los predicado unarios pelicula, actor, director, y en los predicados binarios actuaEn, dirigidaPor y trabajaronJuntos. Una base de datos D para el esquema R puede estar dada por:*

D = {pelicula("Volver al Futuro"), pelicula("Esperando la Carroza"), actuaEn("Esperando la Carroza", "Antonio Gasalla"), actuaEn("Esperando la Carroza", "China Zorrilla"), actor("Antonio Gasalla"), dirigidaPor("Esperando la Carroza", "Alejandro Doria")}

(1.1)

Un ejemplo de consulta conjuntiva para esta base de datos es:

$$Q(X) = pelicula(X) \wedge \exists Y actuaEn(X, Y)$$

que pregunta por todas las películas en las que alguien haya actuado, cuya respuesta es el conjunto de una sola tupla {("Esperando la Carroza")}; mientras que un ejemplo de consulta conjuntiva booleana es

Q() = actor(X) ∧∃X actuaEn("Volver al Futuro", X)

la cual pregunta si hay algún actor que haya actuado en Volver al Futuro, cuya respuesta es No.

1.2.1. Dependencias generadoras de tuplas (TGDs)

Las dependencias generadoras de tuplas (TGDs) son restricciones sobre una base de datos en la forma general de las reglas de Datalog+/-. Dado un esquema relacional R, una TGD σ es una fórmula de primer orden de la forma $\forall X \forall Y \Phi(X, Y) \rightarrow \exists Z \Psi(X, Z)$, donde $\Psi(X, Z)$ y $\Phi(X, Y)$ son conjunciones de átomos sobre R, llamadas el *cuerpo* y la *cabeza* de σ, denotadas $cuerpo(\sigma)$ y $cabeza(\sigma)$ respectivamente. Tal σ es satisfecha en una base de datos D para R ssi, toda vez que exista un homomorfismo h que mapea los atomos de $\Phi(X, Y)$ a atomos de D, existe una extensión $h\prime$ de h que mapea los atomos de $\Psi(X, Z)$ a átomos de D. Todos los conjuntos de $TGDs$ son finitos. Usualmente se omiten los cuantificadores universales en las TGDs.

Ejemplo 2. *Tomando en cuenta el ejemplo anterior, un posible conjunto de TGDs parar R puede estar dado por*

- *Toda película fue dirigida por un director:*

$$pelicula(X) \rightarrow \exists Z dirigidaPor(X, Z)$$

- *Si alguien dirigió una película entonces es un director:*

$$dirigidaPor(X, Y) \rightarrow director(Y)$$

- *Si alguien actuó en una película entonces es un actor:*

$$actuaEn(X, Y) \rightarrow actor(Y)$$

- *Si un actor actuó en una película que fue dirigida por un director, eso implica que el actor y el director trabajaron juntos:*

$$actuaEn(X, Y) \wedge dirigidaPor(X, W) \rightarrow trabajaronJuntos(Y, W)$$

Es simple ver que ninguna de las TGDs anteriores es satisfecha en D. Consideremos entonces la base de conocimiento $D\prime$ que extiende a D de la siguiente manera:

$D\prime$ = D $\cup$ {dirigidaPor("Volver al Futuro", "Robert Zemeckis"), director("Robert Zemeckis"), actor("China Zorilla"), director("Alejandro Doria"), trabajaronJuntos("Antonio Gasalla", "Alejandro Doria"), trabajaronJuntos("China Zorilla", "Alejandro Doria")}

(1.2)

Ahora todas las TGDs están satisfechas en $D\prime$.

1.2.2. Respondiendo consultas bajo TGDs

La evaluación de una consulta conjuntiva o una consulta conjuntiva booleana sobre una base de datos bajo un conjunto de TGDs se define de la siguiente manera. Dada una base de datos D para R, y un conjunto de TGDs Σ sobre R, el conjunto de modelos de D y Σ, el cual denotamos por $mods(D, \Sigma)$, es el conjunto (posiblemente infinito) de todas las bases de datos B tales que (i) $D \subseteq B$ y (ii) toda $\sigma \in \Sigma$ es satisfecha en B. El conjunto de todas las respuestas para una consulta conjuntiva Q, denotado por $ans(Q, D, \Sigma)$, es el conjunto de todas las tuplas *a* tales que $a \in Q(B)$ para todo $B \in mods(D, \Sigma)$. La respuesta para una consulta conjuntiva booleana Q a D es *Sí*, denotada por $D \cup \Sigma \models Q$, si solo si, $ans(Q, D, \Sigma) \neq \emptyset$. Notar que responser consultas bajo TGDs para el caso general es indecidible [1], aún cuando el esquema y las TGDs son fijas [2].

Ejemplo 3. *Si consideramos la base D del ejemplo 1 y las TGDs del ejemplo 2 podemos ver que $D \notin mods(D, \Sigma)$ ya que como dijimos antes estas TGDs no se satisfacen en D. Por el contrario $D\prime \in mods(D, \Sigma)$, es decir $D\prime$ es modelo de D y Σ. En particular las siguientes bases de datos son dos de los modelos de D y Σ:*

- *D_1 = D $\cup$ {dirigidaPor("Volver al Futuro", "Steven Spielberg"), director("Steven Spielberg"), actor("China Zorilla"), director("Alejandro Doria"), trabajaronJuntos("Antonio Gasalla", "Alejandro Doria"), trabajaronJuntos("China Zorilla", "Alejandro Doria")}*

- *$D_2 = D \cup$ {dirigidaPor("Volver al Futuro", "Robert Zemeckis"), director("Robert Zemeckis"), actuaEn("Volver al Futuro", "Michael J. Fox"), actor("Michael J. Fox"), trabajaronJuntos("Michael J. Fox", "Robert Zemeckis"), actor("China Zorilla"), director("Alejandro Doria"), trabajaronJuntos("Antonio Gasalla", "Alejandro Doria"), trabajaronJuntos("China Zorilla", "Alejandro Doria")}*

Notemos que el átomo película("Volver al Futuro") está en todos los modelos de D y Σ[1], por lo tanto la consulta conjuntiva boleana Q() = película("Volver al Futuro") evalúa a Sí en D y Σ, y lo mismo ocurre con Q() = trabajaronJuntos("China Zorrilla", "Alegandro Doria"). Por el contrario, la consulta Q() = dirigidaPor("Volver al Futuro", "Robert Zemeckis") no es verdadera en todos los modelos de D y Σ[2], por lo tanto esta consulta evalúa a No.

Recordamos que el problema de evaluar una consulta conjuntiva bajo TGDs es LOGSPACE-equivalente al problema de evaluar una consulta conjuntiva booleana [[3], [4], [5], [6]]. Por eso nos enfocamos aquí solo en el problema de responder consultas conjuntivas booleanas. Recordamos también que responder consultas bajo TGDs es equivalente a responder bajo TGDs que tienen un solo átomo en sus cabezas. Por lo tanto, asumimos sin pérdida de generalidad, que toda TGD tiene un solo átomo en su cabeza.

1.2.3. El chase

El *chase* es un procedimiento para reparar una base de datos con respecto a un conjunto de dependencias, de manera tal que el resultado del *chase* satisfaga esas dependencias. Por *chase* nos referimos tanto al procedimiento como a su resultado. El *TGD chase* trabaja sobre una base de datos a través de aplicar las TGDs como se explica a continuación. Considere una base de datos D para un esquema relacional R, y una TGD σ sobre R de la forma $\Phi(X,Y) \rightarrow \exists Z \Psi(X,Z)$. Entonces, σ es *aplicable* a D si existe un homomorfismo h que mapee los átomos de $\Phi(X,Y)$ a átomos de D. Sea σ aplicable a D, y sea h_1 un homomorfismo que extiende h de la siguiente manera: para cada $X_i \in \mathbf{X}, h_1(X_i) = h(X_i)$; para cada $Z_j \in \mathbf{Z}, h_1(Z_j) = z_j$, siendo z_j un *nulo fresco*, es decir, $z_j \in \Delta_N, z_j$ no ocurre en D, y z_j sigue lexicográficamente a todos los otros nulos introducidos previamente. Al aplicar σ a D, se agrega a D el átomo $h_1(\Psi(X,Z))$, si no se encuentra en D previamente. El algoritmo del *chase* para una base de datos D y un conjunto de TGDs Σ consiste en una aplicación exhaustiva de la regla anterior en anchura, lo cual trae como resultado un *chase* (posiblemente infinito) para D y Σ. Formalmente, el *chase* de nivel 0 para D y Σ, denotado por $chase^0(D,\Sigma)$, se define como D, asignándole a cada átomo de D el nivel de derivación 0. Para cada $k \geq 1$, el chase de nivel k de D y Σ, denotado $chase^k(D,\Sigma)$, se construye de la siguiente manera: sean $I_1, ..., I_n$ todas las posibles imágenes de los cuerpos de las TGDs en Σ relativas a algún homomorfismo tal que (i) $I_1, ..., I_n \subseteq chase^{k-1}(D,\Sigma)$ y (ii) el nivel más alto de todo átomo en cada I_i es $k-1$; entonces, se aplica cada posible TGD sobre $chase^{k-1}(D,\Sigma)$, escogiendo las TGDs y los homomorfismos en un

[1] El átomo pertenece a D, entonces estará trivialmente en cualquier supraconjunto del mismo.
[2] En particular no es verdadera en D_2.

orden lineal y lexicográfico respectivamente, y asignándole a cada átomo nuevo el nivel de derivación k. El chase de D relativo a Σ, denotado $chase(D, \Sigma)$, se define entonces como el limite de $chase^k(D, \Sigma)$ para $K \to \infty$.

El *chase* (posiblemente infinito) es un modelo universal[8] de D y Σ, es decir que existe un homomorfismo del $chase(D, \Sigma)$ a cada $B \in mods(D, \Sigma)$. Este resultado implica que las consultas conjuntivas boolenas sobre D y Σ pueden ser evaluadas en el chase para D y Σ, es decir que $D \cup \Sigma \models Q$ es equivalente a $chase(D, \Sigma) \models Q$.

Ejemplo 4. *Consideremos otra vez la base de datos D del ejemplo 1 y las TGDs del ejemplo 2, agregando ahora la TGD $actor(X) \wedge actuaEn(X, Y) \to \exists Z interpretoA(X, Y, Z)$; significando que el actor X, en la película Y interpretó al actor Z.*

Entonces, en la construcción del chase (D, Σ), iteramos de la siguiente forma.

- *En la primera iteración obtenemos los siguientes átomos a través de las TGDs que se indican en cada paso.*
  - $pelicula(X) \to \exists Z dirigidaPor(X, Z)$:
    - *dirigidaPor("Volver al Futuro", z_1)*
  - $dirigidaPor(X, Y) \to director(Y)$:
    - *director("Alejandro Doria")*
  - $actuaEn(X, Y) \to actor(Y)$:
    - *actor("China Zorrilla")*
  - $actuaEn(X, Y) \wedge dirigidaPor(X, W) \to trabajaronJuntos(Y, W)$:
    - *trabajaronJuntos("Antonio Gasalla", "Alegandro Doria")*
    - *trabajaronJuntos("China Zorrilla", "Alegandro Doria")*
  - $actor(X) \wedge actuaEn(X, Y) \to \exists Z interpretoA(X, Y, Z)$:
    - *interpretoA("Antonio Gasalla", "Esperando la Carroza", z_2)*
- *En la segunda iteración agregamos primero todos los átomos que obtuvimos en la iteración anterior y aplicamos sobre los átomos iniciales más los nuevos las TGDs. En este caso solo dos TGDs arrojan átomos nuevos:*
  - $dirigidaPor(X, Y) \to director(Y)$:
    - *director(z_1)*
  - $actor(X) \wedge actuaEn(X, Y) \to \exists Z interpretoA(X, Y, Z)$:
    - *interpretoA("China Zorilla", "Esperando la Carroza", z_3)*

Para este ejemplo particular el resultado del chase, es decir un modelo universal de (D, Σ) es la siguiente instancia de base de datos:

chase(D, Σ) = $D \cup$ {dirigidaPor("Volver al Futuro", z_1), director("Alejandro Doria"), actor("China Zorilla"), trabajaronJuntos("Antonio Gasalla", "Alejandro Doria"), trabajaronJuntos("China Zorilla", "Alejandro Doria"), interpretoA("Esperando la Carroza",

"Anotnio Gasalla", z_2), director(z_1), interpretoA("Esperando la Carroza", "China Zorrilla", z_3)}

(1.3)

1.2.4. Datalog+/-, fragmento Guarded

Esta fragmento consiste en una clase especial de TGDs que es tratable computacionalmente, siendo a la vez lo suficientemente expresiva como para modelar ontologías. Las consultas conjuntivas booleanas relativas a estas TGDs pueden ser evaluadas en una parte finita del *chase*, que es de tamaño constante cuando las consulta y las TGDs están fijas. En base a este resultado, la complejidad de datos de evaluar una consulta conjuntiva booleana relativa a TGDs dentro del fragmento *guarded*, resulta polinomial en el caso general y lineal para consultas atómicas. Una TGD σ está en el fragmento *guarded* si solo si contiene un átomo en su cuerpo que contiene todas las variables cuantificadas universalmente en σ. El primer átomo más a la izquierda con estas característica se denomina la *guarda* de σ. Al resto de los átomos de σ los llamamos *átomos laterales* de σ.

Ejemplo 5. *La TGD $actor(X) \wedge actuaEn(X,Y) \rightarrow \exists Z interpretoA(X,Y,Z)$ está en el fragmento guarded, siendo actuaEn(X, Y) la guarda y actor(X) el átomo lateral. Mientras que la TGD $actuaEn(X,Y) \wedge dirigidaPor(X,W) \rightarrow trabajaronJuntos(Y,W)$ no está en el fragmento, dado que para ningún átomo del cuerpo sucede que este contenga a todas las variables que existen en el cuerpo.*

Notar que esta clase de TGDs asegura decibilidad. Como se muestra en [2], agregar una sola TGD fuera del fragmento a un programa Datalog+/- puede quitar esta propiedad. Sea entonces R un esquema relacional, D una base de datos para R, y Σ un conjunto de TGDs en el fragmento guarded. El *chase graph* para D y Σ es un grafo dirigido que consiste en los átomos de $chase(D,\Sigma)$ como el conjunto de nodos en los cuales habrá una flecha de **a** hacia **b** si solo si **b** es obtenido de **a** y posiblemente otros átomos por medio de una sola aplicación de alguna TGD $\sigma \in \Sigma$. Aquí marcamos a **a** como la guarda si solo si **a** es la guarda de σ. Consideramos ahora al *guarded chase forest* como al subgrafo del *chase graph* de D y Σ que consta de (i) todos los átomos del *chase graph* como nodos y (ii) una flecha desde **a** hacia **b** si solo si **b** fue obtenido a partir de **a** y posiblemente otros átomos por medio de una sola aplicación de una TGD $\sigma \in \Sigma$ con **a** como guarda.

Definimos la profundidad *guarded* de un átomo **a** en el *guarded chase forest* para D y Σ, denotada *profundidad(**a**)*, como la longitud del camino desde D hasta **a** en el *forest*. El *guarded chase* de nivel menor o igual que k, denotado $g - chase^k(D,\Sigma)$, es el conjunto de todos los átomos en el *guarded chase forest* de profundidad a lo sumo k.

Ejemplo 6. *Sea R un esquema relacional y sea Σ un conjunto de TGDs para R dado por:*

- $\sigma_1 = r_1(X,Y) \wedge r_2(X) \rightarrow \exists Z r_3(Z,X,Y)$
- $\sigma_2 = r_3(X,Y,W) \rightarrow r_2(Y)$

- $\sigma_3 = r_2(Y) \wedge r_4(X, Y) \rightarrow r_1(Y, X)$

Sea D una base de datos para R con

$$D = \{r_4(a, b), r_1(a, b), r_2(a)\}$$

La siguiente figura muestra el chase graph para D y Σ. Para cada nodo en el grafo se muestra su nivel de derivación en el chase, es decir el número de iteración en el algoritmo del chase en el que cada nodo fue agregándose al grafo.

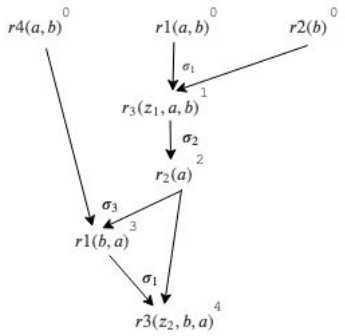

Fig. 1.1: Chase Graph

A modo de comparación, mostramos en la figura 1.2 el guarded chase forest, para mismo D y Σ. Notar que el mismo es un sub-grafo del anterior y que las profundidades de los nodos son ahora siempre menores o iguales que sus respectivos niveles de derivación en el chase graph.

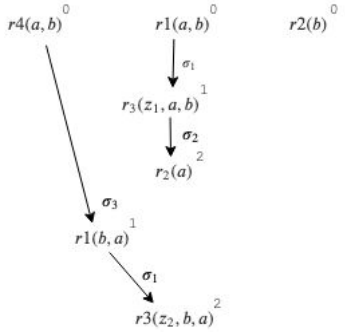

Fig. 1.2: Guarded Forest

Bounded guard-depth property para el fragmento Guarded

Sea R un esquema relacional y Σ un conjunto de TGDs para R. Decimos que Σ tiene la propiedad *Bounded guard-depth property* solo si, para toda base de datos D y para toda consulta conjuntiva booleana Q, para toda vez que exista un homomorfismo μ que mapea Q en el $chase(D, \Sigma)$, entonces existe un homomorfiso $\delta(Q)$ tal que todos los ancestros de $\delta(Q)$ en el $chasegraph$ para D y Σ están contenidos en el $g{-}chase^{\gamma_k}(D, \Sigma)$, donde γ_k solo depende de Q y R.

Tal como se muestra en [8], las TGDs del fragmento *Guarded* poseen la propiedad anterior. En resumen, todos los átomos que se encuentran en la respuesta a una consulta conjuntiva se encuentran dentro del $g - chase^{(n+1)\cdot\delta}$, con $\delta = (2 \cdot \omega)^{\omega} \cdot 2^{(2\omega)^{\omega \cdot |R|}}$, siendo ω la máxima aridad de algún predicado en R y n la cantidad de átomos en la consulta.

Esta profundidad en el *guarded chase forest*, es independiente del tamaño de D, gracias a lo cual se puede afirmar que responder consultas en el fragmento *Guarded* es $P - completo$ en la complejidad de datos.

1.2.5. Datalog+/-, fragmento Lineal

En el fragmento *Lineal* de Datalog+/- las TGDs tienen un solo átomo en su cuerpo, es decir que tienen la forma $\forall X \forall Y \Phi(X, Y) \rightarrow \exists Z \Psi(X, Z)$, siendo $\Phi(X, Y)$ un sólo átomo. Este fragmento esta estrictamente dentro del fragmento guarded, y aún perdiendo cierto poder expresivo con respecto al anterior, es lo suficientemente expresivo como para representar ontologías encodeadas en las *Description Logics* de la familia *Dl-Lite*. Además, tiene la ventaja de ser *FO-rewritable* en la complejidad de datos.

Bounded derivation-depth property

Definimos ahora la *bounded derivation-depth property* para un conjunto de TGDs, la cual es estrictamente más fuerte que la *bounded guard-depth property*, dado que la primera implica la segunda, pero no al revés. Informalmente, la *bounded derivation-depth property* dice que para responder a una consulta conjuntiva no es necesario explorar más allá de una profundidad k en el *chase graph* (y no en el *guarded chase forest* como en el fragmento anterior), cuyo tamaño solo depende de Q y R.

First-order rewritability

Una clase de TGDs Σ_t es reescribible en primer orden o *first-order rewritable* si solo si para todo conjunto de TGDs en Σ_t y para toda consulta conjuntiva booleana Q, existe una consulta de primer orden Q_{Σ_t} tal que, para toda base de datos D, sucede que $D \cup \Sigma_t \models Q$ si solo si $D \models Q_{\Sigma_t}$. Dado que responder consultas de primer orden es un problema en AC_0 en la complejidad de datos [7], también se encuentra en esta clase de problemas responder consultas bajo TGDs que sean reescribibles en primer orden. Teniendo en cuenta además que toda consulta de primer orden es reescribible a una consulta Q^* de *SQL*, esta propiedad nos indica que para el fragmento *lineal* podemos hacer uso práctico de los motores de base de datos *SQL* con todas las optimizaciones ya provistas por los mismos.

Ejemplo 7. *Consideremos el siguiente conjunto Σ_T de TGDs en el fragmento lineal sobre un esquema R:*

- $\sigma_1 = bonaerense(X) \rightarrow argentino(X)$
- $\sigma_2 = cordobes(X) \rightarrow argentino(X)$
- $\sigma_3 = londinense(X) \rightarrow inglés(X)$

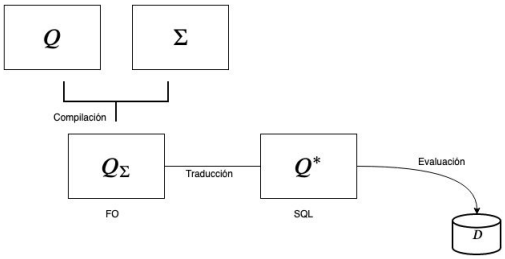

Fig. 1.3: First Order Rewrite

- $\sigma_4 = casado(X) \rightarrow \exists Z casadoCon(X, Z)$

Si tenemos $Q(X) = argentino(X) \wedge casadoCon(X, Z)$ *podemos reescribir esta consulta como una unión de consultas en las que para cualquier base de datos* D *sobre* R*, y sin tener en cuenta* Σ_T*, la unión de los resultados de tales consultas será el mismo que el resultado de* $Q(X)$ *sobre* D *y* Σ_T*. Es fácil ver que en este caso, la unión de las siguientes consultas es una reescritura válida* $Q(X)$ Σ_T*:*

- $Q_1(X) = argentino(X) \wedge casadoCon(X, Z)$
- $Q_2(X) = bonaerense(X) \wedge casadoCon(X, Z)$
- $Q_3(X) = cordobes(X) \wedge casadoCon(X, Z)$
- $Q_4(X) = argentino(X) \wedge casado(X)$
- $Q_5(X) = bonaerense(X) \wedge casado(X)$
- $Q_6(X) = cordobes(X) \wedge casado(X)$

Se han propuesto diversos algoritmos de reescritura [[16], [17]]. El objetivo es siempre conseguir una reescritura lo más chica posible, dado que esto permitirá un uso práctico real de los sistemas de base de datos *SQL*. Cuando el resultado de la reescritura es demasiado grande, el uso práctico se diluye; dado que los motores *SQL* van perdiendo su buena performance a medida que el tamaño de la consulta aumenta.

1.2.6. Negative Constraints

Una *Negative Constraint* es una fórmula de primer orden de la forma $\forall X \Phi(X) \rightarrow \perp$, donde $\Phi(X)$ es una conjunción de átomos. Estas *constraints* nos ayudan a expresar restricciones sobre las ontologías.

Ejemplo 8. *Podemos decir que los predicados esNegativo y esPositivo representan dos clases que no tienen instancias en común de la siguiente forma:*

$$esPositivo(X) \wedge esNegativo(X) \rightarrow \perp$$

De manera similar podemos decir que ningún miembro de una clase pertenece a una relación:

$$soltero(X) \wedge casadoCon(X, Y) \rightarrow \perp$$

Podemos inclusive expresar que dos relaciones son disjuntas:

$$mayor(X, Y) \wedge menor(X, Y) \rightarrow \bot$$

Responder consultas en una base de datos bajo un conjunto de TGDs Σ_T y un conjunto de restricciones Σ_C puede realizarse sin agregar complejidad de la siguiente manera: para cada *negative constraint* $\sigma = \Phi(X, Y) \rightarrow \bot \in \Sigma_C$ verificamos que se satisface en D y Σ_T, lo cual se puede hacer chequeando que la consulta conjuntiva booleana $Q_\sigma = \Phi(X, Y)$ evalúe a Falso en D y Σ_T. Escribimos $D \cup \Sigma_T \models \Sigma_C$ si solo si toda $\sigma \in \Sigma_C$ es falsa en D y Σ_T. Esto nos lleva inmediatamente al siguiente resultado: decimos que una consulta conjuntiva booleana es verdadera en D, Σ_T y Σ_C, y lo anotamos $D \cup \Sigma_T \cup \Sigma_C \models Q$ si solo si (i) $D \cup \Sigma_T \models Q$ o (ii) $D \cup \Sigma_T \not\models \Sigma_C$.

1.2.7. Equality-Generating Dependencies (EGDs) y Keys

Las dependencias generadoras de igualdad o *Equality-Generating Dependencies* también son importantes a la hora de representar ontologías. Estas son fórmulas de primer orden que generalizan dependencias funcionales y en particular claves.

Sin embargo, si bien es cierto que agregar *negative constraints* es sencillo desde el punto de vista computacional, no sucede lo mismo con las EGDs: la interacción entre las TGDs y las EGDs puede tornar indecidible el problema de responder consultas[6]. Se puede comprobar que un conjunto fijo de EGDs y guarded TGDs pueden simular una maquina de Turing universal, con lo cual responder consultas es indecidible para tales dependencias. Por tal motivo, consideramos ahora una clase más restringida de EGDs, a las cuales llamamos *claves no conflictivas*. Tal clase de EGDs muestra una interacción controlada con las TGDs (y NCs), de manera tal que no incrementan la complejidad de responder consultas, agregando al mismo tiempo suficiente poder expresivo para poder modelar ontologías.

Una *dependencia generadora de igualdad* (o EGD) es una fórmula σ de primer orden de la forma $\forall X \Phi(X) \rightarrow X_i = X_j$, donde $\Phi(X)$, el cual llamamos el cuerpo de σ, es una conjunción (no necesariamente guarded) de átomos, y X_i y X_j son variables en X. Llamamos a $X_i = X_j$ a la cabeza de σ. Tal σ se satisface en una base de datos D para R si solo si, siempre que exista un homomorfismo h tal que $h(\Phi(X) \subseteq D$, sucede que $h(X_i) = h(X_J)$. Se suelen omitir los cuantificadores universales en estas fórmulas, y consideramos solo conjuntos finitos de EGDs.

Ejemplo 9. *La siguiente fórmula σ es una EGD:*

$$r(X, Y_1) \wedge r(X, Y_2) \rightarrow Y_1 = Y_2.$$

La base de datos $D = \{r(a, b), r(c, d)\}$ satisface σ, dado que todo homomorfismo h que mapee el cuerpo de σ a átomos de D es tal que $h(Y_1) = h(Y_2)$. Por el contrario, la base de datos $D = \{r(a, b), r(a, d)\}$ no satisface σ.

Una EGD σ sobre R de la forma $\Phi(X) \rightarrow X_i = X_j$ es *aplicable* a una base de datos D si solo si existe un homorfismo $\eta : \Phi(X) \rightarrow D$ tal que $\eta(X_i)$ y $\eta(X_j)$ son distintos y no son ambos constantes. Si $\eta(X_i)$ y $\eta(X_j)$ son constantes distintas en Δ, entonces

hay una violación de σ, y el chase falla. Caso contrario, el resultado de aplicar σ a D es la base de datos $h(D)$ que se obtiene a partir de D al reemplazar toda ocurrencia de algún elemento no constante $e \in \{\eta(X_i), \eta(X_j)\}$ en D para el otro elemento $e\prime$ (si e y $e\prime$ son los dos nulos, entonces e precede a $e\prime$ en orden lexicográfico.)

El chase para una base de datos D y un conjunto Σ_T de TGDs más un conjunto Σ_E de $EGDs$, el cual llamamos *full chase* y denotamos $chase(D, \Sigma_T \cup \Sigma_E)$, se computa de manera iterativa por medio de aplicar todas las EGDs que sean aplicables entre cada aplicación de cada TGD como se describió previamente.

Ejemplo 10. *Consideremos el siguiente conjunto de TGDs y EGDs* $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$*:*

$\sigma_1 : \ p(X) \rightarrow \exists Z s(X, Z)$

$\sigma_2 : \ s(X, Y), r(X) \rightarrow X = Y$

$\sigma_3 : \ s(X, Y), s(X, Z) \rightarrow Y = Z$

Sea D *la base de datos* $\{p(a), r(a), s(a, b)\}$*. En el cómputo de* $chase(D, \Sigma)$*, primero aplicamos* σ_1 *y agregamos el átomo* $s(a, z_1)$*, donde* z_1 *es un nulo. Entonces, al aplicar* σ_2 *sobre* $r(a)$ *y* $s(a, z_1)$ *transformamos* $s(a, z_1)$ *en* $s(a, a)$*. Ahora* σ_3 *resulta aplicable sobre* $s(a, a)$ *y* $s(a, b)$*, pero al intentar igualar* $a = b$ *el chase falla, pues hay una violación, dado que* a *y* b *son constantes distintas en* Δ*.*

Separabilidad

Sea R un esquema relacional, y sean Σ_T y Σ_E dos conjuntos de *TGDs* y *EGDs* sobre R, respectivamente. Entonces, Σ_E es separable de Σ_T si solo si para toda base de datos D sobre R, se cumplen las siguientes condiciones:

- Si hay una violación de una *EGD* $\sigma \in \Sigma_E$ en $chase(D, \Sigma_T \cup \Sigma_E)$, entonces hay también una violación de σ en D.
- Si no hay violación de ninguna *EGD* $\sigma \in \Sigma_E$ entonces para toda consulta Q, sucede que $chase(D, \Sigma_T \cup \Sigma_E) \models Q$ si solo si $chase(D, \Sigma_T) \models Q$.

Esto implica que agregar *EGDs* separables a un conjunto de *TGDs* no incrementa la complejidad de contestar consultas, tanto en el caso *lineal* como el el *guarded*. Lo anterior se puede afirmar gracias a que es suficiente evaluar si cada EGD se satisface en D, y si esto ocurre es posible construir el *chase* sin necesidad de aplicar el algoritmo del *full chase*, pues ambos arrojarán el mismo resultado.

Sea q un predicado sobre R y p1,p2,..pn posiciones en p, una clave K, denotada como K =q[p1], q[p2], ...,q[n],....

Claves

Dado un predicado relacional p sobre un esquema relacional R una clave K sobre p es un conjunto de posiciones para el cual sucede que para toda base de datos D sobre R no pueden existir dos *hechos* en D para p tales que los dos hechos tengan los mismos valores en todas las posiciones de K pero distintos valores en posiciones que

no estén incluidas en K. Sean entonces $p_1, p_2, ...p_n$ el conjunto de posiciones y sea q el predicado sobre el cual se quiere definir una clave k. Denotamos esta clave por medio de $k = \{q[p1], q[p2], ..., q[p_n]\}$.

Ejemplo 11. *Sea p el predicado binario padreDe sobre un esquema relacional R, donde la primer posición indica el DNI de una persona y la segunda posición el DNI de su padre. Sea $k\{p[1]\}$ una clave definida sobre padreDe. Esto indica que no pueden existir dos átomos en D con igual DNI en la primera posición y distinto DNI en la segunda posición, es decir que ninguna persona puede tener dos padres diferentes.*

Podemos definir una clave a partir de un conjunto de EGDs.

Ejemplo 12. *Sea p un predicado ternario sobre un esquema relacional R. Sea $k[1]$ una clave para p. Podemos expresar esta clave por medio de dos EGDs:*

- $p(X, Y_1, Z_1) \wedge p(X, Y_2, Z_2) \rightarrow Y_1 = Y_2$.
- $p(X, Y_1, Z_1) \wedge p(X, Y_2, Z_2) \rightarrow Z_1 = Z_2$.

Claves no conflictivas

Sea K una clave, y sea σ una TGD de la forma $\Phi(X, Y) \rightarrow \exists Z r(X, Z)$. Decimos que K es no conflictiva (NC) con σ si se cumple alguna de las siguientes condiciones:

- El predicado relacional sobre el cual K está definido es diferente de r.
- Las posiciones de K en r no son un subconjunto propio de las posiciones de X en r en la cabeza de σ, y toda variable en Z aparece una sola vez en la cabeza de σ.

Decimos que K es no conflictiva con un conjunto de TGDs Σ_T si solo si k es no conflictiva con toda $\sigma \in \Sigma_T$. Un conjunto de claves Σ_K es no conflictivo con Σ_T si solo si todas clave $k \in \Sigma_K$ es no conflictiva con Σ_T.

Ejemplo 13. *Consideremos las siguientes claves K_1, K_2, K_3, K_4 definidas por los siguientes conjuntos de posiciones para el predicado r: $K_1 = \{r[3]\}, K_2 = \{r[2], r[3]\}, K_3 = \{r[1]\}, K_4 = \{r[1], r[2]\}$, y la TGD $\sigma = p(X, Y) \rightarrow \exists Z r(Z, X, Y)$. Teniendo en cuenta entonces que la cabeza de σ es r, y el conjunto de posiciones en r con variables cuantificada universalmente es $H = \{r[2], r[3]\}$, resulta que K_1 es la única clave definida por un conjunto de posiciones que es subconjunto propio de H. Por lo tanto salvo K_1, todas las claves son no conflictivas con σ.*

Tal como se muestra en [8], la propiedad de no conflictivdad entre un conjunto de claves y un conjunto de TGDs, implica su separabilidad. La principal idea detrás de la prueba se describe a continuación. La condición de no conflictividad entre una clave K y una TGD σ asegura que o bien (a) la aplicación de σ en el *chase* genera un átomo con un nuevo nulo en una posición de K, y entonces este nuevo átomo no viola K; o bien (b) las posiciones cuantificadas universalmente en la cabeza de σ coinciden con las posiciones de la clave K, con lo cual cualquier átomo nuevo generado a partir de

σ debe tener nulos frescos en todas las posiciones salvo las de la clave, no produciéndose una violación en este caso tampoco. Como todos los nuevos nulos son distintos entre sí, el *chase* es homomórficamente equivalente al *TGD full chase*.

Concluímos entonces esta sección afirmando que en el caso NC, las claves no incrementan la complejidad-datos de contestar consultas bajo TGDs y constraints, tanto para el fragmento *guarded* como para el *lineal*.

1.3. Manejo de inconsistencia

En lo que sigue describiremos distintas semánticas para *query answering* tolerante a la inconsistencia. Todas ellas se basan en la noción de *data base repair*. El objetivo es, a través de las consultas, obtener siempre información consistente; intentando a la vez alcanzar un buen equilibrio entre el poder expresivo de las semánticas y la complejidad computacional que estas requieren.

Inconsistencia en Datalog +/-

Asumiremos en este trabajo que las *TGDs, NCs y Keys* son correctas; es decir, capturan correctamente la semántica del dominio. Esta suposición implica que el conjunto Σ dado por $\Sigma_T \cup \Sigma_N \cup \Sigma_K$ es siempre satisfacible; es decir que la aplicación de las TGDs no generan inconsistencias. En este enfoque los conflictos se generan a partir de los datos: la instancia de base de datos es la parte que debe ser *reparada* o *modificada* para restaurar la consistencia pudiendo así cumplir con las restricciones que impone el conjunto dado por $\Sigma_N \cup \Sigma_K$. Cabe mencionar que esta no es la única opción. Muchos trabajos actuales se enfocan en otras posibilidades ([9], [10], [11], [12]).

Database repairs

Mencionamos anteriormente que nuestro objetivo es obtener información consistente, aún conviviendo en ambientes con datos inconsistentes. ¿Pero como podemos obtener información consistente a partir de una base de datos que no lo es?. La principal herramienta que utilizaremos para alcanzar este objetivo es la noción de *data base repair*. Dada una base de datos D que contradice un conjunto de restricciones de integridad y de claves obtendremos un subconjunto de D que no contradice estas restricciones pero que difiere de D *minimalmente*. Formalmente decimos que un *repair* para (D, Σ) es una base de datos $D\prime$ tal que:

- $D\prime \subseteq D$
- $mods(D\prime, \Sigma) \neq \varnothing$
- $\nexists D\prime\prime$ tal que $D\prime\prime \subset D\prime$ y $mods(D\prime\prime, \Sigma) \neq \varnothing$

Ejemplo 14. *Sea R un esquema relacional y sea Σ_T un conjunto de TGDs para R dado por:*

- $\sigma_1 = amigos(X, Y) \rightarrow tieneAmigos(X)$

- $\sigma_2 = amigos(X, Y) \rightarrow tieneAmigos(Y)$
- $\sigma_3 = amigos(X, Y) \wedge tienePiojos(X) \rightarrow tienePiojos(Y)$

Sea Σ_n el conjunto de restricciones dado por una única restricción

- $\sigma_n = pelado(X) \wedge tienePiojos(X) \rightarrow \bot$

Sea D una base de datos para R dada por los siguiente hechos:

- *tienePiojos('Federico')*
- *amigos('Federico', 'Miguel')*
- *amigos('Miguel', 'Pablo')*
- *esPelado('Pablo')*

Al aplicar el algoritmo del chase sobre D y Σ_T obtenemos que:

$$chase(D, \Sigma_T) = D \cup \{tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), tienePiojos('Miguel'), tienePiojos('Pablo')\}$$

Vemos entonces que en $chase(D, \Sigma_T)$ hay una violación de $sigma_n$, pues resulta que este contiene simultáneamente los átomos $esPelado('Pablo')$ y $tienePiojos('Pablo')$. El conjunto de repairs para (D, Σ) está dado entonces por:

- $r_1 = \{tienePiojos('Federico'), amigos('Federico','Miguel'), amigos('Miguel','Pablo')\}$
- $r_2 = \{tienePiojos('Federico'), amigos('Federico','Miguel'), esPelado('Pablo')\}$
- $r_3 = \{tienePiojos('Federico'), amigos('Miguel','Pablo'), esPelado('Pablo')\}$
- $r_4 = \{amigos('Federico','Miguel'), amigos('Miguel','Pablo'), esPelado('Pablo')\}$

Notar que en este caso cada uno de los repairs está dado por incluír en el repair cualquier subconjunto de tres átomos de entre los cuatro átomos de D. Al decir que Federico es amigo de Miguel, y que Miguel es amigo de Pablo, y al decir que los amigos se contagian los piojos, tiene que ser que Federico le contagio los piojos a Miguel y este se los contagió a Pablo. Pero esto no puede ser porque Pablo es pelado. Observar entonces que quitando cualquier átomo de D, obtenemos un subconjunto maximal consistente.

Semántica AR

La semántica AR, es la más aceptada para *query answering* en ontologías potencialmente inconsistentes. Dada una base de conocimiento (D, Σ) y una consulta conjuntiva Q, decimos que $(D, \Sigma) \models_{AR} Q$ si solo si $(R, \Sigma) \models Q$ para todo R *repair* de (D, Σ).

En el ejemplo anterior, si tomamos $Q() = tieneAmigos('Miguel')$, resulta que $(D, \Sigma) \models_{AR} Q$. En cambio, si consideramos la consulta $Q() = tieneAmigos('Pablo')$ resulta que $(D, \Sigma) \not\models_{AR} Q$.

La semántica AR definida arriba coincide con la semántica tolerante a la inconsistencia para Lógicas de Descripción presentada en [13]. Aunque esta semántica puede ser considerada en cierto sentido la elección natural para el objetivo semántico que estamos buscando, tiene la desventaja de ser dependiente de la forma sintáctica de la base de conocimiento. Supongamos que la base de conocimiento $KB\prime = (D\prime, \Sigma)$ difiere de la base de conocimiento $KB = (D, \Sigma)$ simplemente porque $D\prime$ incluye átomos que se pueden inferir a partir de Σ y un subconjunto consistente de D. En este caso KB y $KB\prime$ son lógicamente equivalentes, y cabe esperar por lo tanto que sus *repairs* coincidan y que las respuestas a toda consulta sobre KB y $KB\prime$ coincidan bajo toda semántica. Sin embargo esto no sucede con la semántica *AR*.

Ejemplo 15. *Considerando D y Σ del ejemplo anterior, definamos ahora la base de datos $D\prime$ dada por $D \cup \{tieneAmigos('Pablo')\}$. Es claro que el nuevo átomo introducido se puede inferir lógicamente de D y Σ, notar de hecho que el mismo se encuentra en el chase de aquel ejemplo. Si consideramos ahora los repairs para $D\prime$ y Σ obtenemos*

- $r\prime_1 = r_1 \cup \{tieneAmigos('Pablo')\}$
- $r\prime_2 = r_2 \cup \{tieneAmigos('Pablo')\}$
- $r\prime_3 = r_3 \cup \{tieneAmigos('Pablo')\}$
- $r\prime_4 = r_4 \cup \{tieneAmigos('Pablo')\}$

Con lo cual resulta que $(D\prime, \Sigma) \models_{AR} tieneAmigos('Pablo')$, resultado opuesto para la KB del ejemplo anterior.

Semántica CAR

Dependiendo del escenario particular en el que se quiera resolver este tipo de inconsistencias, el comportamiento anterior podría ser considerado incorrecto. Esto motivó la definición de una nueva semántica que no presenta esta característica. De acuerdo con esta semántica, llamada *Closed ABox Repair*, los *repairs* toman en cuenta no solo los hechos explícitamente incluídos en D, sino también aquellos hechos que se pueden inferir junto con Σ y al menos un subconjunto de D que sea consistente. Para formalizar la idea anterior, es necesario dar primero las siguientes definiciones. Dada una base de conocimiento $K = (D, \Sigma)$, denotamos por medio de $HB(K)$ a la *Herbrand Base* de K, es decir el conjunto de hechos que se pueden construir sobre el alfabeto de de Γ_K. Definimos después las consecuencias lógicas de K como el conjunto $clc(K) = \{\alpha | \alpha \in HB(K) \wedge \exists S \subseteq D$ tal que $Mods(S, \Sigma) \neq \varnothing \wedge (S, \Sigma) \models \alpha\}$. Podemos ahora dar la siguiente definición de *Closed ABox Repair*.

Sea $K = (D, \Sigma)$ una base de conocimiento en Datalog+/-. Un *Car Repair* de K es un conjunto $D\prime$ de hechos tales que:

1. $D\prime \subseteq clc(K)$
2. $mods(D\prime, \Sigma) \neq \varnothing$
3. $\nexists D\prime\prime$ tal que $mods(D\prime\prime, \Sigma) \neq \varnothing$ y :

- $D'' \cap D \supset D' \cap D$ o,
- $D'' \cap D = D' \cap D \wedge D'' \supset D'$

Denotamos por medio de $CAR - Rep(D, \Sigma)$ al conjunto de todos los *Car Repairs* para K. Intuitavemnte, un *CAR-repair* es un subconjunto de *clc(K)* consistente con Σ que preserva máximamente al conjunto de hechos D. En particular, la condición 3 dice que preferimos a D' por encima de cualquier otro $D_R \subseteq clc(K)$ consistente con Σ tal que $D_R \cap D \subset D' \cap D$ (es decir que D_R mantiene un subconjunto más pequeño de D con respecto a D'). Entonces, entre todos aquellos subconjuntos D_R que tengan la misma intersección con D, preferimos a aquellos que contengan la mayor cantidad de hechos en *clc(K)* posibles. Dada $KB = (D, \Sigma)$ y una consulta conjuntiva Q, decimos que $KB \models_{CAR} Q$ si solo si $(R, \Sigma) \models Q$ para cada $R \in CAR - Rep(D, \Sigma)$.

Ejemplo 16. *Consideremos las bases de conocimiento presentadas en 14 y en el 15 y veamos que en este caso el conjunto de CAR-repairs es el mismo para las dos bases. Para eso construímos primero el conjunto clc(K). Es fácil ver que para ambos ejemplos este conjunto esta dado por clc(K) = {amigos('Federico', 'Miguel'), amigos('Miguel', 'Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), tienePiojos('Federico'), tienePiojos('Miguel'), tienePiojos('Pablo'), esPelado('Pablo')} y que en ambos casos el conjunto de CAR-repairs está dado por:*

- *{amigos('Federico', 'Miguel'), amigos('Miguel', 'Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), tienePiojos('Federico'), tienePiojos('Miguel'), tienePiojos('Pablo')}*
- *{amigos('Federico', 'Miguel'), amigos('Miguel', 'Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), esPelado('Pablo')}*
- *{amigos('Federico', 'Miguel'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), tienePiojos('Federico'), tienePiojos('Miguel'), esPelado('Pablo')}*
- *{amigos('Miguel', 'Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo'), tienePiojos('Federico'), esPelado('Pablo')}*

Notar que dado que el conjunto de CAR-repairs es el mismo para ambas bases de conocimiento, las respuestas serán siempre las mismas para toda consulta Q bajo esta semántica. En particular teniendo en cuenta D y Σ del ejemplo 14 tenemos que

$$(D, \Sigma) \models_{CAR} tieneAmigos('Pablo'),$$

y lo mismo ocurre para D' y Σ del ejemplo 15.

Semánticas de intersección

En general, responder consultas es intratable tanto para la semántica *CAR* como *AR*. Dado que esto supone un obstáculo en su uso práctico, se propusieron aproximaciones de estas semánticas, para las cuales se ha demostrado que responder consultas conjuntivas es polinomial en las lógicas de descripción *DL-Lite*. En ambos casos, la

aproximación consiste en tomar como único *repair* la intersección de los *AR-repairs* y de los *CAR-repairs* respectivamente. Esta idea se corresponde con el enfoque *WIDTIO (When you are in doubt throw it out)*, provenitente del área de revisión de creencias [[14],[15]].

Semántica IAR

Sea $KB = (D, \Sigma)$ una base de conocimiento, y sea Q una consulta conjuntiva. Decimos que $KB \models_{IAR}$ si solo si $(\cap_{R \in AR-Rep(D,\Sigma)}, \Sigma) \models Q$

Ejemplo 17. *Consideremos (D, Σ) y el conjunto de AR-repairs del ejemplo 14. La intersección de aquellos repairs es el conjunto vacío. Por lo tanto, para toda consulta Q resulta que $(D, \Sigma) \not\models_{IAR} Q$. Si consideramos en cambio $(D\prime, \Sigma)$ del ejemplo 15 y consideramos la consulta $Q() = tieneAmigos('Pablo')$, resulta que $(D, \Sigma) \models_{IAR} Q$. Notar entonces que la semántica IAR sufre la misma desventaja que la semántica AR, pues también es dependiente de la forma sintáctica de la base de conocimiento.*

Semántica ICAR

De manera análoga a la definición de la semántica *IAR* definimos la semántiuca *ICAR*. Sea $KB = (D, \Sigma)$ una base de conocimiento, y sea Q una consulta conjuntiva. Decimos que $KB \models_{ICAR}$ si solo si $(\cap_{R \in CAR-Rep(D,\Sigma)}, \Sigma) \models Q$

Ejemplo 18. *Al igual que la semántica CAR, la semántica ICAR tiene la ventaja de no ser dependiente de la forma sintáctica de la KB. Teniendo en cuenta lo comentado en el ejemplo 16, es fácil ver que tanto para la base de conocimiento presentada en el ejemplo 14 como para la presentada en el ejemplo 15, la intersección de los CAR-repairs coincide. Esta intersección está dada por el conjunto {tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo')}. Podemos ejemplificar entonces diciendo que:*

- $(D, \Sigma) \models_{ICAR} tieneAmigos('Pablo')$.
- $(D, \Sigma) \not\models_{ICAR} sonAmigos('Miguel', 'Pablo')$.

Semántica ICR

Sea $Cn(D, \Sigma)$ la clausura lógica de D y Σ. Sea KB una base de conocimiento (D, Σ), y sea Q una consulta conjuntiva. Decimos que $KB \models_{ICR} Q$ si solo si $(\cap_{R \in Rep(KB)}, Cn(R, \Sigma)) \models Q$. Veremos a continuación que esta semántica es una *aproximación sana* de *AR* e *ICAR*, y que además es una aproximación *completa* de *IAR*.

Ejemplo 19. *Consideremos D y Σ del ejemplo 14. Tenemos que calcular las clausuras lógicas de los cuatro repairs de ese mismo ejemplo.*

- *$Cn(r_1)$ = { tienePiojos('Federico'), amigos('Federico', 'Miguel'), amigos('Miguel', 'Pablo'), tienePiojos('Miguel'), tienePiojos('Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo')}*

- $Cn(r_2)$ = { *tienePiojos('Federico'), amigos('Federico', 'Miguel'), esPelado('Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tienePiojos('Miguel')*}
- $Cn(r_3)$ = { *tienePiojos('Federico'), amigos('Miguel', 'Pablo'), esPelado('Pablo'), tieneAmigos('Miguel'), tieneAmigos('Pablo')*}
- $Cn(r_4)$ = { *amigos('Federico', 'Miguel'), amigos('Miguel', 'Pablo'), esPelado('Pablo'), tieneAmigos('Federico'), tieneAmigos('Miguel'), tieneAmigos('Pablo')* }

La intersección de estos conjuntos esta dada por { tieneAmigos('Federico'), tieneAmigos('Miguel') }. Podemos decir entonces que $(D, \Sigma) \models_{ICR} tieneAmigos('Federico') \wedge tieneAmigos('Miguel')$.

Notemos que tal como se comentó en el ejemplo 17, para mismo D y Σ, la intersección de los *repairs* fue el conjunto vacío. Es decir que al menos en este caso la semántica *IAR* fue más restrictiva que la semántica *ICR*. Veremos en la próxima sección que en verdad esto se cumple para todo D y todo Σ.

1.3.1. Aproximaciones sanas y completas

Dadas dos semánticas *A* y *B*, decimos que la semántica *A* es *sana* con respecto a la semántica *B* si solo si para toda consulta Q resulta que si Q es verdadera bajo la semántica *A* también lo es bajo la semántica *B*. Por el contrario, *A* es *completa* con respecto a *B* si solo si para toda consulta Q resulta que si Q es verdadera bajo *B* también lo es bajo *A*. De esto se desprende que si *A* es *sana* respecto de *B* entonces *B* es *completa* respecto de *A*.

Veamos ahora como se aproximan entre sí las semánticas que hemos mencionado. En la siguiente figura hay una flecha que parte desde la semántica *A* hacia la semántica *B* si solo si la semántica *A* es una aproximación sana de la semántica *B*.

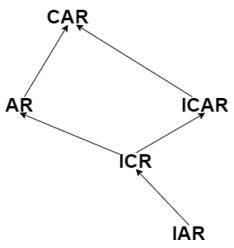

Fig. 1.4: Aproximaciones entre las semánticas

Tal como se muestra en el gráfico, *IAR* es la semántica más restrictiva, siendo esta una aproximación *sana* de *ICR*, y siendo esta última una aproximación *sana* de *AR* e *ICAR*, aunque estas dos no se aproximan entre sí. Por otro lado *AR* e *ICAR* son aproximaciones *sanas* de *CAR*, la menos restrictiva de todas las semánticas.

2. RESULTADOS FORMALES

2.1. Algoritmo RepairsFinder

Presentamos ahora un algoritmo que dada una base de conocimientos (D, Σ), con $\Sigma = \Sigma_T \cup \Sigma_N$, retorna todos sus *repairs*, es decir todos los subconjuntos de D maximales consistentes con respecto a Σ. Nuestro objetivo es recortar el espacio de búsqueda y evitar a la vez la mayor cantidad de consultas a la base de conocimiento. Dada una base de datos D con n *hechos*, la cantidad de posibles subconjuntos para esa base de datos es 2^n. Podríamos de manera *naive* considerar la totalidad de los subconjuntos y para cada uno de ellos evaluar consistencia y maximalidad. Sin embargo, sabemos que es necesario organizar la búsqueda de los *repairs* de manera eficiente, dada la gran cantidad de subconjuntos que necesitaríamos explorar de otro modo. Tengamos en cuenta además, que para evaluar si un subconjunto s es consistente, necesitamos evaluar cada una de las *Negative Constraints* en Σ_N contra s. Tal como se detalló en la sección 1.2.6, para evaluar una $NC : \phi(x, y) \rightarrow \bot$, podemos evalular la consulta $Q(x, y) = \phi(x, y)$ sobre (s, Σ_T) y si la respuesta es distinta al conjunto vacío entonces decimos que hay una violación de tal *NC* y que por lo tanto s no es consistente. Pero como ya mencionamos, queremos minimizar estas consultas lo más posible.

El algoritmo *RepairsFinder* intenta minimizar el espacio de búsqueda con una estrategia *BottomUp* y *TopDown* en simultáneo, descartando todos los subconjuntos de D que sean subconjunto de algún *repair* ya encontrado o superconjunto de algún subconjunto minimal inconsistente, lo que llamaremos de ahora en más *culprit*.

Teniendo en cuenta que si etiquetamos numéricamente todos los *hechos* en D y mantenemos los subconjuntos ordenados a lo largo de todo el algoritmo, verificar si un subconjunto es subconjunto de otro, tiene un costo lineal en el tamaño del subconjunto más chico. Por tal motivo, verificar inclusión de conjuntos no es significativamente menos eficiente que consultar la *KB* desde el punto de vista computacional.

2.1.1. Entendiendo RepairsFinder

En el algoritmo *RepairsFinder* se define el siguiente conjunto de variables:

- *top*: se inicializa como un conjunto cuyo único elemento es la base de datos D.
- *bottom*: se inicializa como un conjunto cuyo único elemento es el conjunto vacío.
- *repairsChicos y repairsGrandes*: estas dos variables se inicializan como el conjunto vacío.

El algoritmo consta de un *bucle* principal entre las líneas 8 y 11, sobre el cual se irá iterando hasta que alguna de las condiciones que figuran en la línea 8 no se cumplan. Sea n la cantidad de hechos en D y sea K algún número de iteración en la ejecución del algoritmo. Entonces, en algún momento de esa iteración k se cumplirán las siguientes condiciones:

Algorithm 1: RepairsFinder

Input: (D, Σ)
Output: Conjunto de repairs para (D, Σ)

$nTop \leftarrow \sharp(D)$;
$nBottom \leftarrow 0$;
$top \leftarrow \{D\}$;
$bottom \leftarrow \{\emptyset\}$;
$culprits \leftarrow \emptyset$;
$repairsChicos \leftarrow \emptyset$;
$repairsGrandes \leftarrow \emptyset$;
while *top* $\neq \emptyset$ **and** *bottom* $\neq \emptyset$ **and** *nTop* $>$ *nBottom* **do**
    $IterarTop()$;
    $IterarBottom()$;
end
if *nBottom* == *nTop* **and** $top \neq \emptyset$ **then**
    $AgregarConsistentesEnTopARepairs()$;
end
return $repairsChicos \cup repairsGrandes$

Algorithm 2: IterarTop

$AgregarConsistentesEnTopARepairs()$;
$top \leftarrow subConjuntosDeKElementos(D, nTop - 1).filtrarPor(t \rightarrow repairsGrandes.ningunoCumple(rg \rightarrow t \subset rg))$;
$nTop - -$;

Algorithm 3: AgregarConsistentesEnTopARepairs

for $t \in top$ **do**
    if $culprits.algunoCumple(c \rightarrow c \subset t)$ **then** $t.consistente \leftarrow false$;
    else
        $t.consistente \leftarrow chequearConsistencia(t, \Sigma)$;
        if *t.consistente* **then** repairsGrandes.agregar(t);
    end
end

Algorithm 4: IterarBottom

```
1  proximoBottom ← subConjuntosDeKElementos(D, nBottom +
     1).filtrarPor(pb → culprits.ningunoCumple(c → c ⊂ pb));
2  for pb ∈ proximoBottom do
3      if repairsGrandes.algunoCumple(rg → pb ⊂ rg) then pb.consistente ← true;
4      else
5          pb.consistente ← chequearConsistencia(pb, Σ);
6          if ¬pb.consistente then culprits.agregar(pb);
7      end
8  end
9  for b ∈ bottom do
10     if proximoBottom.filtrarPor(pb → b ⊂ pb).todosCumplen(pb →
         ¬pb.consistente) then
11         repairsChicos.agregar(b)
12     end
13 end
14 bottom ← proximoBottom.filtrarPor(b → b.consistente);
15 nBottom + +;
```

- *Top* es el conjunto formado por todos los subconjuntos t de D que no son subconjunto de ningún *repair* y tales que $|t| = n+1-k$ para todo t. Esto es consecuencia de los lemas 4 y 6.
- *Bottom* es el conjunto formado por todos los subconjuntos consistentes b de D tales que $|b| = k - 1$ (lema 14).
- Todos los *repairs* r tales que $|r| = k - 1$ serán agregados al conjunto *repairsChicos* (lema 12).
- El conjunto *repairsGrandes* es el conjunto formado por todos los *repairs* r tales que $|r| > n - k$ (lema 11).

En la demostración formal que presentamos en la sección 2.1.2, probaremos que el *output* de *RepairsFinder* es correcto. Teniendo en cuenta que el algoritmo retorna la unión de los conjuntos *repairsGrandes* y *repairsChicos*, probar lo anterior es similar a afirmar que la unión de los conjuntos *repairsGrandes* y *repairsChicos* es igual al conjunto de *repairs* para la base de conocimiento sobre la cual se haya evaluado *RepairsFinder*.

En la figura 2.1 representamos el espacio de búsqueda del algoritmo *RepairsFinder*. A medida que se ejecuten las iteraciones del algoritmo, las variables *top* y *bottom* se moverán en ese espacio, mostrándose en la figura los valores a donde *apuntan* esas variables antes de empezar la primera iteración. En cada iteración, *top "bajará"* un nivel en ese espacio y *bottom "subirá"* un nivel. Mostramos en la figura 2.2 los nuevos valores que tomarán estos conjuntos antes de comenzar la segunda iteración, de manera de ilustrar como van cambiando estas variables a medida que se ejecuta el

algoritmo. Cabe mencionar aquí que los *repairs* que se agreguen a *repairsGrandes* serán aquellos elementos en *top* que cumplan las condiciones necesarias para garantizar que esos conjuntos son verdaderamente *repairs*. Análogamente, los *repairs* que se agreguen a *repairsChicos* serán aquellos elementos en *bottom* que cumplan con aquellas mismas condiciones.

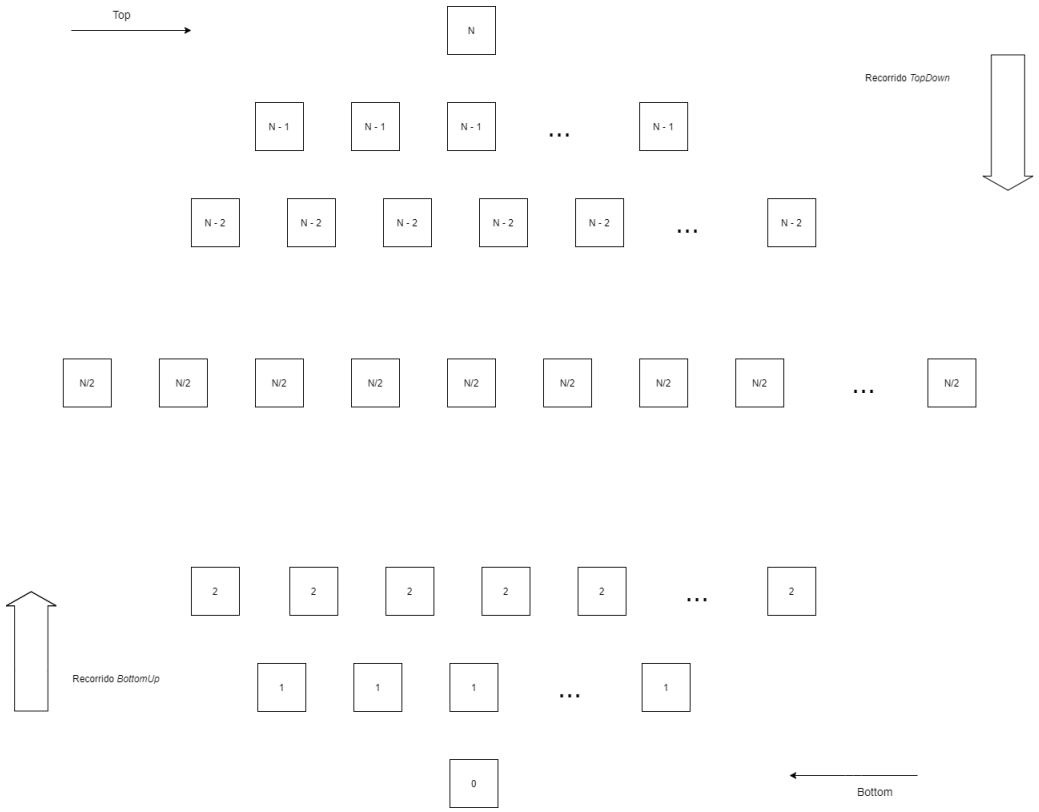

Fig. 2.1: Espacio de búsqueda RepairsFinder - iteración 1.

Describiremos ahora de manera informal como funciona el algoritmo. Notemos que en cada iteración se llama a las funciones *IterarTop* e *IterarBottom*. La función *IterarTop* procede de la siguiente manera: para cada elemento t en *top* miramos si es superconjuto de algún *culprit*, y en caso afirmativo lo marcamos como inconsistente. En caso negativo lo validamos de la manera usual, es decir consultando la *KB*. Finalmente si t resulta consistente, será agregado al conjunto de *repairsGrandes*.

A su vez, la función *IterarBottom()* procede como se describe a continuación. Primero se define el conjunto *proximoBottom* como el conjunto de todos los subconjuntos posibles de D de cardinalidad k que no sean superconjunto de ningún *culprit*. Para cada uno de ellos verificamos si son subconjunto de algún elemento en *repairsGrandes*, y en caso afirmativo los marcamos como consistentes. En caso contrario verificamos consistencia consultando la *KB*. Para todos aquellos que resulten inconsistentes los agregamos al conjunto de *culprits*. Por otro lado, todos los elementos de *bottom* que no sean subconjunto de algún elemento consistente en *proximoBottom* podemos afirmar que son un *repair*, y los agregamos al conjunto de *repairsChicos*. Finalmente, en la próxima iteración, *bottom* estará definido por todos los elementos en *proximoBot-*

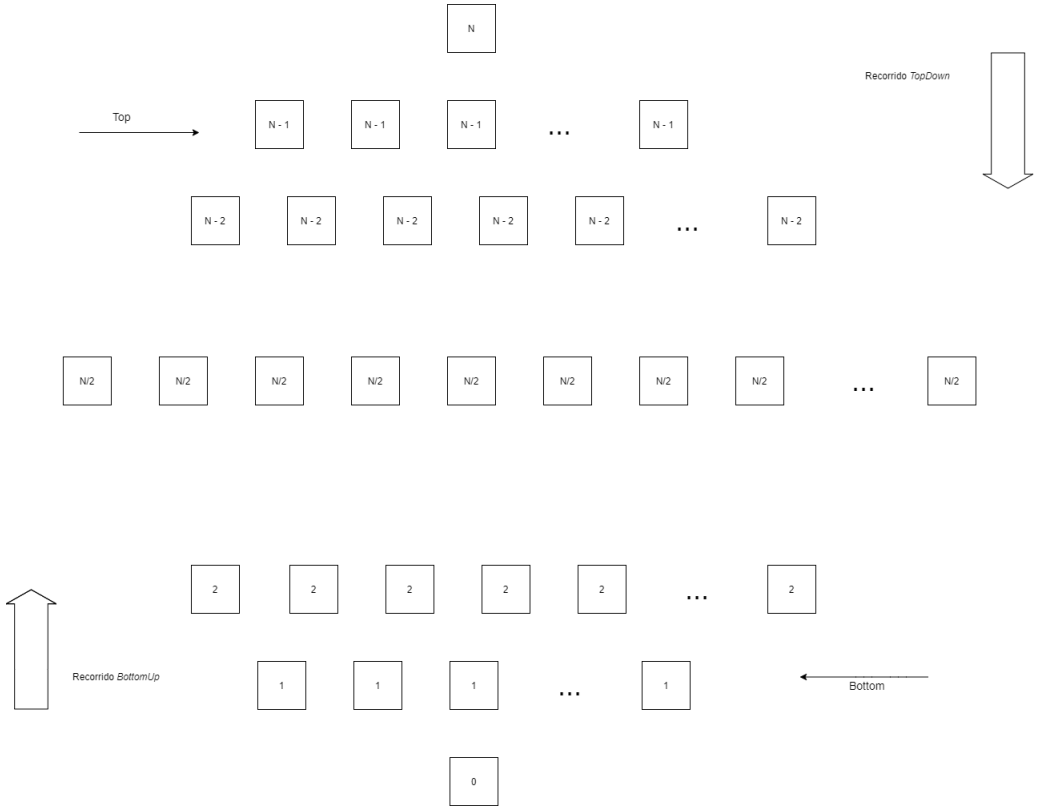

Fig. 2.2: Espacio de búsqueda RepairsFinder - iteración 2.

tom que sean consistentes.

2.1.2. Demostración Formal de correctitud del algoritmo RepairsFinder

Teorema 1. *Dada una KB =* (D, Σ)*, el algoritmo RepairsFinder computa exactamente el conjunto de repairs para esa base de conocimiento.*

Demostración. Sea $repairs(D, \Sigma)$ el conjunto de *repairs* para una base de conocimiento (D, Σ). Para probar este teorema necesitamos demostrar que:

(A) Dado un *r* cualquiera en el *output* de $RepairsFinder(D, \Sigma)$, r pertenece al conjunto $repairs(D, \Sigma)$.

(B) Dado un *r* cualquiera perteneciente al conjunto $repairs(D, \Sigma)$, r pertenece al *output* de $RepairsFinder(D, \Sigma)$.

Es decir, necesitamos ver que el primer conjunto está incluido en el segundo y el segundo en el primero, lo cual demostrará que ambos son el mismo conjunto.

Demostremos primero que (A) es verdadero. Veamos entonces que dado un *r* cualquiera en el *output* de $RepairsFinder(D, \Sigma)$, r pertenece al conjunto $repairs(D, \Sigma)$.

Para probar esto, veamos que r cumple las tres cualidades necesarias para ser un *repair*:

(1) r es consistente con respecto a Σ.

(2) r es un subconjunto de D.

(3) r es maximal, es decir que no existe $r\prime \supset r$ tal que (1) y (2) valen para $r\prime$.

Dado que el *output* de *RepairsFinder* está compuesto por la unión entre *repairsChicos* y *repairsGrandes*, podemos contemplar ahora dos casos, o bien r pertenece a *repairsChicos* o bien r pertenece a *repairsGrandes*.

Caso 1: $r \in repairsGrandes$.

Asumamos entonces que $r \in repairsGrandes$ y veamos que se cumplen las tres condiciones enumeradas anteriormente.

(1) r es consistente: Notemos que *RepairsFinder* solo agrega elementos a *repairsGrandes* en la línea 5 de la función *AgregarConsistentesEnTopARepairs* y que en esa misma línea, en la cláusula del *if* se verifica consistencia. Por lo tanto, si r pertenece a *repairsGrandes*, entonces r es consistente.

(2) r es un subconjunto de D: Si r fue agregado a *repairsGrandes*, entonces r es un elemento que perteneció al conjunto *top* para alguna iteración k del algoritmo. Para verificar esto, notar que tal como se dijo en el párrafo anterior, solo se agregan elementos a *repairsGrandes* en la línea 5 de la función *AgregarConsistentesEnTopARepairs*, y que en esa línea se agrega el elemento t al conjunto, donde t es un elemento perteneciente a *top*. Notemos entonces que *top* se define por primera vez en la tercera línea del algoritmo *RepairsFinder* como el conjunto formado por todos los subconjuntos de D de n elementos[1]. Notemos también, que en cada iteración del algoritmo, el conjunto *top* se redefine en la función *IterarTop* como un conjunto formado por k elementos de D, donde $k = nTop - 1$.

(3) r es maximal: Vimos que si r se agregó a *repairsGrandes* este fue agregado en la línea 5 de la función *AgregarConsistentesEnTopARepairsGrandes*. Notemos ahora que hay dos casos posibles, o bien la llamada a esta función se ejecutó desde la función *IterarTop* en alguna iteración k del *while* principal del algoritmo *RepairsFinder* (caso I) o bien la llamada a *AgregarConsistentesEnTopARepairsGrandes* se ejecutó fuera del ciclo, en la linea 14 del algoritmo *RepairsFinder* (caso II).

Caso I: Sea k el número de iteración en el que r fue agregado a *repairsGrandes*. Teniendo en cuenta que r es un elemento perteneciente a *top* en la iteración k, y considerando el lema 4, resulta entonces que $|r| = n+1-k$. Considerando el mismo lema, r no es subconjunto de ningún elemento en *repairsGrandes*. Por el lema 6, todos los *repairs* de cardinalidad mayor estricta que $n+1-k$ están en el conjunto *repairsGrandes*. Por lo tanto, r no es subconjunto propio de ningún *repair*. Por absurdo asumamos que r no es maximal, es decir asumamos que existe un subconjunto $s \subseteq D$ tal que s es consistente y $r \subset s$. Existen dos opciones, o bien s es un *repair* o bien s está incluido en un *repair*. En ambos casos resulta que r es un subconjunto propio de algún *repair*, lo cual es absurdo, pues dijimos que por los lemas 4 y 6 r no es subconjunto propio de ningún *repair*.

[1] Trivialmente hay un solo subconjunto de D de n elementos.

Caso II: Sea k la última iteración que se ejecutó del ciclo *RepairsFinder* antes de la llamada a *AgregarConsistentesEnTopARepairsGrandes* en la línea 14 del algoritmo. Teniendo en cueneta que r es un elemento de *top*, y considerando el lema 10, resulta entonces que $|r| = n - k$. Considerando el mismo lema, r no es subconjunto de ningún elemento en *repairsGrandes*. Por el lema 11, todos los *repairs* de cardinalidad mayor estricta que $n - k$ están en el conjunto *repairsGrandes*. Por lo tanto, r no es subconjunto propio de ningún *repair*. Al igual que en el caso I asumamos por absurdo que r no es maximal. Siguiendo el mismo razonamiento concluimos que r es un subconjunto propio de algún *repair*, lo cual es absurdo, pues dijimos que r no es subconjunto propio de ningún *repair*.

Caso 2: $r \in repairsChicos$.

Asumamos entonces que $r \in repairsChicos$ y veamos que también se cumplen las tres condiciones.

(1) r es consistente: Notemos que solo se agregan elementos a *repairsChicos* en la línea 11 de la funcion *IterarBottom*. Notemos además que esa línea dice "*repairsChicos.agregar(b)*", donde b es un elemento en *bottom*. Debemos probar entonces que los elementos en *bottom* al momento de ejecutarse tal línea del algoritmo, son subconjuntos consistentes. Podemos ahora separar nuevamente en dos casos: o bien r fue agregado en la primera iteración del algoritmo *RepairsFinder*, o bien r fue agregado en alguna iteración posterior. En el primer caso, *bottom* consta de un solo elemento, el conjunto vacío, el cual es trivialmente consistente. En el segundo caso, el conjunto *bottom* fue redefinido en la iteración anterior del ciclo en la línea 14 de la función *IterarBottom*, otra vez como un conjunto de subconjuntos consistentes. Por lo tanto, si r pertenece a *repairsChicos*, entonces r es consistente.

(2) r es un subconjunto de D: Si r fue agregado a *repairsChicos*, entonces r estaba en el conjunto *bottom*, que está definido en toda iteración del *while* principal como un conjunto de subconjuntos de k elementos de D. Para verficar esto, notemos que como dijimos anteriormente *bottom* se inicializa como un conjunto cuyo único elemento es el conjunto vacío, el cual es trivialmente un subconjunto de D. Veamos además que en cada iteración, *bottom* se redefine en la línea 14 de la función *IterarBottom* como todos los elementos consistentes de *proximoBottom*, donde *proximoBottom* está definido como un conjunto formado por subconjuntos de *nBottom + 1* elementos de D, donde *nBottom* siempre vale igual al número anterior al número de iteración que se está ejecutando.

(3) r es maximal: Sea r un elemento en *repairsChicos*. Asumamos por absurdo que r no es maximal; es decir que existe un elemento $\alpha \in D$ tal que $\alpha \notin r$ y $r \cup \{\alpha\}$ es consistente. Llamamos c al conjunto $r \cup \{\alpha\}$. Sea k el número de iteración en el que r fue agregado a *repairsChicos*. Considerando la línea 11 de la función *IterarBottom* es fácil ver que si r fue agregado a *repairsChicos* entonces r perteneció al conjunto *bottom* en esa iteración k. Por el lema 2 sabemos además que entonces $|r| = k - 1$. Veamos además que en esa iteración k, el conjunto c perteneció a *proximoBottom*. Para afirmar esto tenemos en cuenta el lema 3, el cual dice que en una iteración k, si un subconjunto s de D tiene cardinalidad k y no es superconjunto de ningun *culprit*,

entonces al ejecutarse la línea 9 de la función *IterarBottom*, s está en *proximoBottom*. Trivialmente, $|c| = k$, y además por el lema 7, c no puede ser superconjunto de ningún elemento en *culprits*, pues c es consistente. Tengamos en cuenta ahora que en la linea 10 de *IterarBottom* la cláusula del *if* determina que solo se agregará r a *repairsChicos* cuando todos los conjuntos en *proximoBottom* que sean superconjunto de r no sean consistentes. Por lo tanto, en esa iteración no se pudo haber agregado r a *repairsChicos*, pues c esta en *proximoBottom* y c es superconjunto de r y c es consistente. Pero esto es absurdo, pues dijimos que r fue agregado a *repairsChicos* en esa iteracion k.

Habiendo probado (1), (2) y (3) para el caso $r \in repairsChicos$ y para el caso $r \in repairsGrandes$ hemos probado que (A) es verdadero.

Probemos ahora que (B) es verdadero. Es decir veamos que dado un *repair* r_0 perteneciente al conjunto de $repairs(D, \Sigma)$, r_0 se encuentra en el *output* de *repairsFinder(D, Σ)*.

Sea $n = |D|$ y separemos en los siguientes cuatro casos:

(1) $|r_0| > \lfloor n/2 \rfloor$

(2) n impar y $|r_0| \leq \lfloor n/2 \rfloor$

(3) n par y $|r_0| < \lfloor n/2 \rfloor$

(4) n par y $|r_0| = \lfloor n/2 \rfloor + 1$

Caso (1): $|r_0| > \lfloor n/2 \rfloor$

Notemos que por el lema 6, en caso de que se haya ejecutado la iteración $k = n - |r_0| + 1$, todos los *repairs* r tales que $|r| \geq n - k + 1$ habrán sido agregados al conjunto *repairsGrandes*. Pero $k = n - |r_0| + 1$, por lo tanto $n - k + 1 = |r_0|$. Es decir que si se ejecutó la iteración k todos los *repairs* r tales que $r \geq |r_0|$ habrán sido agregados a *repairsGrandes*, por lo tanto, en particular r_0 habrá sido agregado a *repairsGrandes*. Queremos ver entonces que esta iteración k se ejecutó.

Asumamos por absurdo que esta iteración no se ejecutó. Es decir asumamos que la última iteración ejecutada fue la iteración j con $j < k$. Notemos que por la condición del *while* en la línea 8 del algoritmo *RepairsFinder*, la última iteración que se ejecutará será la que haga valer alguna de las tres condiciones siguientes:

(a) $bottom = \emptyset$

(b) $top = \emptyset$

(c) $nBottom \geq nTop$

Asumamos por absurdo que una vez ejecutada la iteración j vale alguna de esas tres condiciones.

Condición (a): $bottom = \emptyset$: En este caso, considerando que la variable *bottom* se definió por última vez en la línea 14 de la función *iterarBottom* durante la iteración j, teniendo en cuenta además que *proximoBottom* es en este caso el conjunto de

todos los subconjuntos $s \subset D$, con $|s| = nBottom + 1$ tales que no son superconjunto de ningún elemento en *culprits*, y considerando también que por el lema 7 todos los *culprits* son inconsistentes podemos decir que, gracias al lema 15, si la variable *bottom* es el conjunto vacío entonces todos los subconjuntos $s \subset D$ con $|s| = j$ son inconsistentes[2].

Por el lema lema 16 $k \leq \lfloor n/2 \rfloor + 1$, pero teniendo en cuenta que $j < k$ resulta que $j \leq \lfloor n/2 \rfloor$ y por lo tanto $|r_0| > j$. Teniendo en cuenta que todos los subconjuntos s de D tal que $|s| = j$ son inconsistentes tomo algun subconjunto s tal que $s \subset r_0$. Resulta entonces que r_0 es superconjunto de un conjunto inconsistente, lo cual es absurdo.

Condición (b): top = $\emptyset$: En este caso, la línea 2 de la función *iterarTop* retornó el conjunto vacío durante la ejecución de la iteración j, lo cual significa que todos los subconjuntos de D con cardinalidad $nTop - 1$ son subconjuntos de algún elemento en *repairsGrandes*. Teniendo en cuenta el lema 6 todos los *repairs* r tales que $r > n+1-j$ ya están en *repairsGrandes*. Además, por el el lema 5, durante la ejecución de la llamada a la función *AgregarConsistentesEnTopARepairs*, todos los *repairs* r tales que $|r| = n - j + 1$, y solamente ellos, serán agregados al conjunto *repairsGrandes*. Considerando entonces estos dos lemas, al momento de ejecutarse la línea 2 de esta función, el conjunto *repairsGrandes* es el conjunto de todos los *repairs* r tales que $|r| \geq n + 1 - j$. Considerando ahora que $nTop - 1 = n - j$, si la línea 2 de esta función retornó vacío durante la ejecución de la iteración j, esto significa que todos los subconjuntos $s \subset D$ tales que $|s| = n - j$ son subconjuntos propios de algún *repair*.

Dijimos que $j < k$, y por lo tanto $n - j > n - k$, con lo cual resulta que $n - j \geq n - k + 1 = |r_0|$. Si $n - j = |r_0|$ entonces por lo dicho en el párrafo anterior r_0 es subconjunto propio de algún *repair*, lo cual es absurdo. Si $n - j > |r_0|$ entonces tomo algún subconjunto s tal que $r_0 \subset s$ con $|s| = n - j$. Dado que s es subconjunto propio de algún *repair* r resulta que $r_0 \subset r$, lo cual es absurdo.

Condición (c): *nBottom* $\geq$ *nTop*: En el caso n par consideremos que dado que *nBottom* se inicializa en 0, y dado que *nTop* se inicializa en n, y teniendo en cuenta que el primero incrementa de a un valor por iteración, y que el segundo decrementa también de a un valor por iteración, este caso solo puede ocurrir cuando ya se hayan ejecutado $\lfloor n/2 \rfloor$ iteraciones, pues una vez ejecutadas esas iteraciones valdrá la condición *nBottom* = *nTop*. Consideremos ahora que para n par $n - \lfloor n/2 \rfloor = \lfloor n/2 \rfloor$. Dado entonces que $|r_0| > \lfloor n/2 \rfloor$, resulta que $n - |r_0| < \lfloor n/2 \rfloor$, y por lo tanto $n - |r_0| + 1 \leq \lfloor n/2 \rfloor$. Teniendo en cuenta entonces que la iteración $\lfloor n/2 \rfloor$ se ejecutó, sí o sí tuvo que ejecutarse la iteración $k = n - |r_0| + 1$, y por lo tanto es absurdo que la última iteración que se ejecutó haya sido la iteración j.

En el caso n impar considerando otra vez que dado que *nBottom* se inicializa en 0, y dado que *nTop* se inicializa en n, y teniendo en cuenta también que el primero incrementa de a un valor por iteración, y que el segundo decrementa también de a un valor por iteración, este caso solo puede ocurrir cuando ya se hayan ejecutado $\lfloor n/2 \rfloor + 1$ iteraciones, pues una vez ejecutadas esas iteraciones valdrá la condición *nBottom* > *nTop*. Consideremos ahora que para n impar $n - \lfloor n/2 \rfloor = \lfloor n/2 \rfloor + 1$. Dado entonces que $|r_0| > \lfloor n/2 \rfloor$, resulta que $n - |r_0| < \lfloor n/2 \rfloor + 1$, y por lo tanto $n - |r_0| + 1 \leq$

[2] Notemos que al ejecutarse la línea 1 de la función *iterarBottom* en la iteración j la variable *nBottom* vale j - 1, pues *nBottom* vale 0 en la primera iteración y se incrementa en un valor en cada iteración.

$\lfloor n/2 \rfloor + 1$. Teniendo en cuenta entonces que la iteración $\lfloor n/2 \rfloor + 1$ se ejecutó, sí o sí tuvo que ejecutarse la iteración $k = n - |r_0| + 1$, y por lo tanto es absurdo que la última iteración que se ejecutó haya sido la iteración j.

Concluimos entonces que sería absurdo que alguna iteración $j < k$ con $k = n - |r_0| + 1$ haga valer alguna de las condiciones (a), (b) o (c). Por lo tanto, para el caso (1): n par y $|r_0| > \lfloor n/2 \rfloor$, resulta que la iteración k tuvo que haberse ejecutado, y entonces por el lema 6, r_0 habrá sido agregado a *repairsGrandes*.

Caso (2): n impar y $|r_0| \leq \lfloor n/2 \rfloor$

Queremos ver que se ejecutó la iteración $k = |r_0| + 1$, pues si se ejecutó tal iteración, por el lema 12 todos los *repairs* r tales que $|r| = k - 1$ estarán en *repairsChicos*, por lo tanto en particular también lo estará r_0.

Asumamos por absurdo que no se ejecutó tal iteración. Sea j la última iteración que se ejecutó, con $j < k$. Demostremos nuevamente por absurdo, que la iteración j no puede hacer valer ninguna de las condiciones (a), (b) o (c).

Condición (a): *top* = $\emptyset$: En este caso, evaluando la línea 2 de la función *iterarTop*, todos los subconjuntos s de D tal que $|s| = n - j$ son subconjuntos propios de algún *repair*. Tenemos que $j < |r_0| + 1 \leq \lfloor n/2 \rfloor + 1$, con lo cual $j < \lfloor n/2 \rfloor + 1$, es decir que $j \leq \lfloor n/2 \rfloor$ y por lo tanto $n - \lfloor n/2 \rfloor \leq n - j$. Teniendo en cuenta que n es impar resulta que $\lfloor n/2 \rfloor < n - \lfloor n/2 \rfloor$ y por lo tanto $\lfloor n/2 \rfloor < n - j$. Considerando ahora que $|r_0| \leq \lfloor n/2 \rfloor$ resulta que $|r_0| < n - j$. Tomo entonces algún subconjunto s de D tal que $|s| = n - j$ y $|r_0| \subset s$. Resulta que s es subconjunto propio de algún *repair*, y por lo tanto r_0 también lo es, lo cual es absurdo.

Condición (b): *bottom* = $\emptyset$: En este caso, evaluando la línea 14 de la función *iterarBottom*, todos los subconjuntos s tales que $|s| = nBottom + 1 = j$ son inconsistentes. Dijimos que $j < k$, esto implica que $j < |r_0| + 1$, y por lo tanto $j \leq |r_0|$. En el caso $j = |r_0|$, tendríamos r_0 inconsistente lo cual es absurdo. En el caso $j < |r_0|$ tomo algún subconjunto s de D tal que $s \subset r_0$. Resulta entonces que s es inconsistente y es subconjunto de un *repair*, lo cual es absurdo.

Condición (c): *nBottom* $\geq$ *nTop*: Esto solo puede pasar cuando ya se ejecutaron $\lfloor n/2 \rfloor + 1$ iteraciones. Si la última iteración ejecutada fue la iteración j entonces no se ejecutó la iteración $|r_0| + 1$, pero $|r_0| \leq \lfloor n/2 \rfloor$, con lo cual $|r_0| + 1 \leq \lfloor n/2 \rfloor + 1$ por lo tanto es absurdo que no se haya ejecutado esa iteración.

Concluimos entonces que para el caso (2), sería absurdo, al igual que en el caso (1), que una iteración $j < k$ con $k = |r_0| + 1$ haga valer alguna de las condiciones (a), (b) o (c). Por lo tanto, para el caso (2) resulta que la iteración k tuvo que haberse ejecutado, y entonces por el lema 12, r_0 habrá sido agregado a *repairsChicos* y se encontrará en el *output* de *RepairsFinder*.

Caso (3): n par y $|r_0| < \lfloor n/2 \rfloor$

Por el mismo motivo que en el caso (2) queremos ver que se ejecutó la iteración $k = |r_0| + 1$.

Asumamos por absurdo que no se ejecutó tal iteración. Sea j la última iteración que se ejecutó, con $j < k$. Demostremos otra vez por absurdo, que la iteración j no puede hacer valer ninguna de las condiciones (a), (b) o (c).

Condición (a): *top* = $\emptyset$: En este caso, evaluando la línea 2 de la función *iterarTop*, todos los subconjuntos s de D tal que $|s| = n - j$ son subconjuntos propios de algún *repair*. Tenemos que $j < |r_0| + 1 < \lfloor n/2 \rfloor + 1$, con lo cual $j < \lfloor n/2 \rfloor + 1$, es decir que $j \leq \lfloor n/2 \rfloor$ y por lo tanto $n - \lfloor n/2 \rfloor \leq n - j$. Teniendo en cuenta que n es par resulta que $\lfloor n/2 \rfloor - 1 < n - \lfloor n/2 \rfloor$ y por lo tanto $\lfloor n/2 \rfloor - 1 < n - j$. Considerando ahora que $|r_0| < \lfloor n/2 \rfloor$ resulta que $|r_0| \leq \lfloor n/2 \rfloor - 1$, con lo cual $|r_0| < n - j$. Tomo entonces algún subconjunto s de D tal que $|s| = n - j$ y $r_0 \subset s$. Resulta entonces que r_0 es subconjunto propio de algún *repair*, lo cual es absurdo.

Condición (b): *bottom* = $\emptyset$: En este caso, evaluando la línea 14 de la función *iterarBottom*, todos los subconjuntos s tales que $|s| = nBottom + 1 = j$ son inconsistentes. Dijimos que $j < k$, esto implica que $j < |r_0| + 1$, y por lo tanto $j \leq |r_0|$. En el caso $j = |r_0|$, tendríamos r_0 inconsistente lo cual es absurdo. En el caso $j < |r_0|$ tomo algún subconjunto s de D tal que $s \subset r_0$. Resulta entonces que s es inconsistente y es subconjunto de un *repair*, lo cual es absurdo.

Condición (c): *nBottom* $\geq$ *nTop*: Esto solo puede pasar cuando ya se ejecutaron $\lfloor n/2 \rfloor$ iteraciones. Asumamos que la última iteración ejecutada fue la iteración $j < k = |r_0| + 1$. Considerando que $|r_0| < \lfloor n/2 \rfloor$, resulta entonces que $|r_0| + 1 < \lfloor n/2 \rfloor + 1$, y por lo tanto $|r_0| + 1 \leq \lfloor n/2 \rfloor$, lo cual lleva a que necesariamente se tuvo que ejecutar la iteración $|r_0| + 1$ lo cual es absurdo.

Concluimos entonces que para el caso (3), tuvo que haberse ejecutado necesariamente la iteración $k = |r_0| + 1$, pues demostramos por absurdo que una iteración $j < k$ no puede hacer valer alguna de las condiciones (a), (b) o (c). Por lo tanto, por el lema 12, r_0 será agregado al conjunto *repairsChicos* y se encontrará en el *output* de *RepairsFinder*.

Caso (4): n par y $|r_0| = \lfloor n/2 \rfloor$

Veamos primero que en este caso se tuvieron que ejecutar necesariamente $\lfloor n/2 \rfloor$ iteraciones. Para esto veamos que la condición del *while* en la línea 8 del algoritmo solo será falsa cuando *nTop* = *nBottom*, es decir veamos que las condiciones *top* $\neq \emptyset$ y que la condición *bottom* $\neq \emptyset$ serán siempre verdaderas durante las $\lfloor n/2 \rfloor$ iteraciones, que por el lema 16 es el máximo de iteraciones que se pueden ejecutar cuando n es par.

Asumamos por absurdo que después de alguna iteración j resulta *bottom* $= \emptyset$. Esto significa que todos los subconjuntos $s \subset D$ con $|s| = j$ son inconsistentes. Dijimos que por el lema 16 $j \leq \lfloor n/2 \rfloor$. Tomo algún conjunto s particular tal que $|s| = j$ y $s \subset r_0$. Pero esto es absurdo, pues r_0 es un *repair* y es superconjunto de un conjunto inconsistente.

Asumamos ahora por absurdo que después de alguna iteración j resulta *top* $= \emptyset$. Esto significa que todos los subconjuntos $s \subset D$ con $|s| = n - j$ son subconjuntos propios de algún *repair*. Pero dado que $j \leq \lfloor n/2 \rfloor$ resulta que $n - j \geq \lfloor n/2 \rfloor$, pues n es impar. En el caso $n - j = \lfloor n/2 \rfloor$, teniendo en cuenta que estamos en el caso $|r_0| = \lfloor n/2 \rfloor$ resulta que r_0 es subconjunto propio de algún *repair*, lo cual es absurdo. En el caso $n - j > \lfloor n/2 \rfloor$, tomo algún s tal que $|s| = n - j$ y $r_0 \subset s$. En este caso r_0 es subconjunto propio de un conjunto que es subconjunto propio de algún *repair*, lo cual es absurdo.

Teniendo en cuenta los dos absurdos, probamos que se han ejecutado las $\lfloor n/2 \rfloor$ iteraciones y por lo tanto, al ser n par resulta que *nBottom* = *nTop* = $\lfloor n/2 \rfloor + 1$. Por

lo tanto la condición del *if* en la línea 12 del algoritmo es verdadera, pues se cumple *nBottom* $=$ *nTop* y se cumple *top* $\neq \emptyset$. Esto significa que se va a ejecutar otra vez la llamada a la función *AgregarConsistentesEnTopARepairs*, y por lo tanto, teniendo en cuenta que la última asignación de la variable *top* fue en la iteración $\lfloor n/2 \rfloor$, y teniendo en cuenta el lema 11 resulta que *top* es el conjunto de todos los subconjuntos $s \subseteq D$ tales que $|s| = \lfloor n/2 \rfloor$ y s no es subconjunto propio de ningún *repair*. Por lo tanto r_0 tiene que estar en *top*. Teniendo en cuenta ahora el lema 9 resulta que r_0 será agregado a *repairsGrandes*.

Hemos probado entonces el caso (4), es decir que en este caso r_0 también estará en el *output* de *RepairsFinder*.

Habiendo entonces probado los casos (1), (2), (3) y (4), hemos probado que (B) es verdadero. □

Lema 2. *Sea k un número de iteración en la ejecución del algoritmo RepairsFinder. Entonces, al momento de ejecutarse la linea 9 de la función IterarBottom, todos los elementos en bottom son subconjuntos de D de $k-1$ elementos.*

Demostración. Para el caso $k = 1$, vemos que el conjunto *bottom* se inicializó en la línea 4 del algoritmo *RepairsFinder* como un conjunto cuyo único elemento es el conjunto vacío, el cual tiene trivialmente $k-1$ elementos. Para el caso $k > 1$ notemos que la última asignación a la variable *bottom* fue en la iteración $k-1$ en la línea 14 de la función *iterarBottom*. En esa línea se redefine a *bottom* como todos los elementos de *proximoBottom* consistentes. Necesitamos ver entonces que en esa iteración $k-1$ los elementos en *proximoBottom* tienen cardinalidad $k-1$, lo cual queda probado por el lema 3. □

Lema 3. *Sea k una iteración en el algoritmo RepairsFinders. Entonces, al momento de ejecutarse la línea 9 de la función IterarBottom, el conjunto proximoBottom consta de todos los subconjuntos de k elementos que no son superconjunto de ningún culprit.*

Demostración. Trivialmente, al comienzo de cada iteración k, la variable *nBottom* guarda el valor $k-1$, pues esta se asigna en 0 antes de la primera iteración y se incrementa en una unidad al final de cada iteración. Teniendo en cuenta entonces que la variable *proximoBottom* se asigna únicamente en la línea 1 de la función *IterarBottom* como todos los subconjuntos de D de $nBottom + 1$ elementos que no sean superconjunto de ningún *culprit*, el lema queda probado. □

Lema 4. *Sea k un número de iteración en la ejecución del algoritmo RepairsFinder. Entonces, al momento de llamarse la función AgregarConsistentesEnTopARepairs para esa iteración k, el conjunto top es exactamente el conjunto de todos los subconjuntos t de D tales que $|t| = n + 1 - k$ con $t \not\subset r$ para todo r en repairsGrandes.*

Demostración. Notemos que para el caso $n = 1$ esto es trivial, pues en esta iteración el único subconjunto de D con $n + 1 - k = n$ elementos es el mismo conjunto D. Y notemos que justamente en esa iteración, top consta justamente de un solo elemento, el conjunto D. Veamos además que en esta iteración al momento de llamarse la función *AgregarConsistentesEnTopARepairs*, el conjunto *RepairsGrandes* es el conjunto

vacío, por lo tanto, de manera trivial, ningún elemento en *top* es subconjunto de algun r en *repairsGrandes*.

Para el caso $k \geq 2$ el conjunto *top* fue definido en la la línea 2 de la función *IterarTop* en la ejecución de la iteración $k-1$, por lo tanto resta ver que en la iteración $k-1$ top se definió como el conjunto formado por todos los conjuntos de $n+1-k$ elementos que no son subconjuntos de ningún r en *repairsGrandes*. Tomando el cambio de variable $k-1=j$, quiero ver que en la iteración j, *top* se definió como el conjunto de todos los conjuntos de $n+1-(j+1)$ elementos. Teniendo en cuenta el primer parámetro que recibe la función subconjuntosDeKElementos[3] en la línea 2 de la función *IterarTop*, quiero ver que en la iteración j, $nTop-1=n+1-(j+1)$, es decir $nTop-1=n-j$. Para el caso $j=1$ esto es trivial. Asumamos que para un j cualquiera vale que $nTop-1=n-j$ y veamos que esto implica que vale para la iteración $j+1$. Teniendo en cuenta la línea 3 de esta función, sabemos que en la iteración $j+1$ la variable *nTop* valdrá un valor menos que en la iteración j, por lo tanto queremos ver que $(nTop-1)-1=n-(j+1)$. Notemos entonces que $(nTop-1)-1=n-(j+1) \iff nTop-1-1=n-j-1 \iff nTop-1=n-j$, siendo esta igualdad válida por hipótesis inductiva. Por último, en la línea 2 de aquella función, se excluye explícitamente de *top* a los subconjuntos de *repairsGrandes*. □

Lema 5. *Sea k un número de iteración en la ejecución de $repairsFinder(D,\Sigma)$. Entonces, durante la ejecución de la llamada a la función AgregarConsistentesEnTopARepairs, todos los repairs r tales que $|r|=n-k+1$, y solamente ellos, serán agregados al conjunto repairsGrandes.*

Demostración. Probamos este lema por inducción en el número de iteración k. Notemos que para probar que el lema vale para $k=1$ necesitamos probar que el conjunto D será agregado a *repairsGrandes* si solo si D es consistente. Notemos que al momento de llamarse la función *AgregarConsistentesEnTopARepairs* en esta primera iteración, la variable *top* consta de un conjunto que contiene a un único conjunto: el conjunto D. Notemos ahora que por el lema 9 D será agregado a *repairsGrandes* si solo si D es consistente.

Para probar el paso inductivo asumimos que la condición vale para todas las iteraciones entre 1 y k y queremos ver que esto implica que vale para la iteración $k+1$. Es decir, queremos ver que si vale para todas las iteraciones menores a $k+1$, entonces durante la ejecución de la iteración $k+1$, durante la ejecución de la funcion *AgregarConsistentesEnTopARepairs*, se agregaran todos, y solamente aquellos *repairs* r tales que $|r|=(n+1-(k+1))=n-k$.

Asumamos entonces que se está ejecutando la llamada a la función *AgregarConsistentesEnTopARepairs* para la iteración $k+1$. Por el lema 4, al momento de llamarse a esta función, top consta de exactamente todos los subconjuntos de D con cardinalidad $n-(k+1)+1=n-k$ que no son subconjunto de ningún elemento en *repairsGrandes*. Notemos que si se cumple la hipótesis inductiva, el conjunto *repairsGrandes* es exactamente el conjunto formado por todos los *repairs* r tales que $|r|>n-k$. Juntando

[3] Asumimos en este punto que la función subconjuntosDeKElementos(k, D) devuelve exactamente el conjunto de todos los subconjuntos de k elementos de D.

esto con lo anterior resulta entonces que top consta de exactamente todos los subconjuntos de D con cardinalidad $n-k+1$ que no son subconjunto propio de ningún *repair*.

Veamos ahora que el conjunto de *repairs* de cardinalidad $n-k$ es exactamente igual al subconjunto de top formado por los elementos en top consistentes. Llamamos a este conjunto $topConsistentes$. Veamos entonces las dos inclusiones. Sea r un *repair* de cardinalidad $n-k$. Queremos ver que $r \in topConsistentes$. Dado que r es un *repair*, no es subconjunto propio de ningún otro *repair*. Además, por ser un *repair*, es consistente. Por lo tanto, r pertenece a $topConsistentes$. Sea t un elemento en *topConsistentes*, veamos que es un *repair* de cardinalidad $n-k$. Sabemos que su cardinalidad es $n-k$ por el lema 4. Sabemos además que es consistente y que no es subconjunto propio de ningún *repair*. Entonces por el lema 8, t es un *repair*.

Por el lema 9 solo los elementos en *topConsistentes* serán agregados a *repairsGrandes*, por lo tanto solo los *repairs* de $n-k$ elementos son agregados a *repairsGrandes*. □

Lema 6. *Sea k un número de iteración en la ejecución de $repairsFinder(D, \Sigma)$. Entonces, al momento de ejecutarse la llamada a la función IterarTop en esa iteración k, el conjunto repairsGrandes es el conjunto formado por todos los repairs r tal que $|r| > (n+1-k)$.*

Demostración. Por el lema 5 para toda iteración j anterior a k sucedió que solo los *repairs* r tales que $|r| = n-j+1$ fueron agregados al conjunto *repairsGrandes*. Esto quiere decir que en la primera iteración se agregaron solo los *repairs* de cardinalidad n, en la segunda los de $n-1$, y así sucesivamente, en la iteración $k-1$ se agregaron todos los *repairs* de cardinalidad $n-k+1$. Por lo tanto al momento de ejecutarse la iteración k el conjunto *repairsGrandes* es el conjunto formado por todos los *repairs* r tal que $|r| > (n+1-k)$. □

Lema 7. *Sea c un elemento en el conjunto culprits. Entonces c es inconsistente.*

Demostración. Trivialmente, el algoritmo *RepairsFinder* solo agrega elementos a la variable *culprits* si son conjuntos inconsistentes, basta con ver la línea 3 de la función *InicializarBottom* y la línea 6 de la función *IterarBottom* y notar que en ninguna otra línea del algoritmo se agregan elementos a este conjunto. □

Lema 8. *Sea KB una base de conocimientos (D, Σ) y sea s un subconjunto consistente de D. Si s no es subconjunto estricto de ningún repair, entonces s es un repair.*

Demostración. Por definición de *repair*, el conjunto s es un *repair* de (D, Σ) si solo si s es subconjunto de D, s es consistente y s es maximal. Ya sabemos por enunciado del lema que s es subconjunto de D y que es consistente. Solo falta ver que es maximal. Por absurdo asumamos que no es maximal. Es decir que existe un conjunto c que es superconjunto estricto de s, tal que c es consitente y subconjunto de D. Hay dos opciones, o bien c es un *repair* o bien c es un subconjunto de un *repair*. En ambos casos resulta que s es subconjunto estricto de algún *repair*. Absurdo, el cual provino de suponer que s no es maximal. □

Lema 9. *La funcion AgregarConsistentesEnTopARepairs agrega solo los elementos consistentes en top a repairsGrandes.*

Demostración. Notemos que en la función *AgregarConsistentesEnTopARepairs* se itera sobre todos los elementos t en *top* y que para cada uno de ellos, t será agregado a *repairsGrandes* si solo si t es consistente. Notemos que para cada t en *top* se verifica que no sea superconjunto de ningún elemento en *culprits*. Por el lema 7 todos los elementos en *culprits* son inconsistentes, por lo tanto si t es superconjunto de algún $c \in culprits$ entonces t es inconsistente. En este caso t no es agregado a *repairsGrandes*. Caso contrario, t podría ser consistente o podría no serlo. Se verifica consistencia de t de la manera usual, y t es agregado a *repairsGrandes* si solo si t es consistente. □

Lema 10. *Sea k un numero de iteración en la ejecución del algoritmo RepairsFinder. Entonces, al momento de finalizar tal iteración, el conjunto top es exactamente el conjunto de todos los subconjuntos t de D tales que $|t| = n - k$ con $t \not\subset r$ para todo r en repairsGrandes.*

Demostración. Por el lema 4, al momento de arrancar la iteración k, el conjunto *top* es exactamente el conjunto de todos los subconjuntos t de D tales que $|t| = n + 1 - k$ con $t \not\subset r$ para todo r en *repairsGrandes*. Notemos que esto quiere decir que al final de la iteracion $k - 1$ *top* era exactamente ese mismo conjunto, pues el momento en el que finaliza la iteracion $k - 1$ es el mismo momento que el que comienza la iteración k. Haciendo un cambio de variable, tomando $j = k - 1$ podemos decir entonces que al momento de terminar la iteracion j el conjunto *top* es exactamente el conjunto de todos los subconjuntos t de D tales que $|t| = n - j = n - (1 + k)$, tales que $t \not\subset r$ para todo r en *repairsGrandes*. □

Lema 11. *Sea k un número de iteración en la ejecución de RepairsFinder. Entonces, al momento de finalizar tal iteracion k, el conjunto repairsGrandes es el conjunto formado por todos los repairs r tal que $|r| > n - k$.*

Demostración. Por el lema 6, al momento de llamarse a la función *AgregarConsistentesEnTopARepairs* en la iteración k, el conjunto *repairsGrandes* consta de exactamente todos los *repairs* r tales que $r < |n - k + 1|$. Por el lema 5, durante la ejecución de la función *AgregarConsistentesEnTopARepairs* se agregarán a *repairsGrandes* exactamente todos los *repairs* r tales que $|r| = n - k + 1$. Por lo tanto, una vez finalizada la iteracion k, el conjunto *repairsGrandes* es exactamente el conjunto de todos los *repairs* r tales que $|r| > n - k$. □

Lema 12. *Sea k una iteración en la ejecución del algoritmo repairsFinder. Entonces, al finalizar tal iteración, todos los repairs r tal que $|r| = k - 1$ han sido agregados al conjunto repairsChicos.*

Demostración. Sea entonces el momento en el que se va a ejecutar el *for* en la línea 9 de la función *IterarBottom* en la iteración k. Teniendo en cuenta el lema 14, este *for* va a iterar sobre el conjunto formado por todos los subconjuntos b consistentes de D tal que $|b| = k - 1$. En el cuerpo de cada iteración de este *for*, se evaluará si para

cada uno de estos subconjuntos b, sucede que para todo conjunto en *proximoBottom* pb que sea superconjunto de b sucede que pb es inconsistente. En caso de ser esto verdadero se agrega b a *repairsChicos*. Veamos entonces que si r es un *repair* r tal que $|r| = k - 1$, entonces r será agregado a *repairsChicos* en alguna iteracion de este *for*. Por absurdo asumamos que r no se agrega. Hay solo dos razones posibles: o bien r no está en *bottom*, o bien r está en *bottom* pero no resulta verdadera la condición del *if* en la línea 10. Caso 1: r no esta en *bottom*. Absurdo, pues r es un conjunto consistente y $|r| = k - 1$. Caso 2: r está en *bottom* pero la cláusula del *if* en la linea 10 es falsa. Esto quiere decir que hay algún elemento en *proximoBottom* pb que es superconjunto de r y pb está marcado como consistente. Pero por lema 13 pb es un subconjunto consistente de D con $|pb| = k$. Por lo tanto, si la condición dio falsa, pb es un superconjunto estricto de r, y pb es consistente. Absurdo, pues r es un *repair*.

Teniendo en cuenta los dos absurdos, si r es un *repair* tal que $|r| = k - 1$, r será agregado a *repairsChicos* en la iteración k. □

Lema 13. *Sea k una iteración en la ejecución del algoritmo repairsFinder. Entonces, al finalizar la línea 8 de la función IterarBottom, si para un elemento pb en proximoBottom resulta que $pb.consistente$ es verdadero, entonces pb es un subconjunto consistente de D y $|pb| = k$.*

Demostración. pb es consistente: vemos que en el *for* que se ejecuta en la línea 2 de la función *IterarBottom*, pb fue marcado como consistente en solo dos casos: o bien pb es un subconjunto de algún elemento en *repairsGrandes*, o bien la llamada a la función *chequearConsistencia(pb, Σ)* retornó *true*. En el primer caso pb es consistente, pues todos los elementos en *repairsGrandes* son conjuntos consistentes, en el segundo caso, dado que la función *chequearConsistencia* es correcta, pb también es consistente.

$|pb| = k$: *proximoBottom* se define en la línea 1 de la función *IterarBottom* como el conjunto formado por todos los elementos de $nBottom + 1$ elementos, y sabemos que $nBottom = k - 1$, pues $nBottom$ vale 0 en la iteración k y $nBottom$ se incrementa en uno en cada iteración.

$pb \subset D$: Trivialmente, *proximoBottom* fue definido como un conjunto de subconjuntos de D. □

Lema 14. *Sea k una iteración en la ejecución del algoritmo repairsFinder. Entonces al finalizar la línea 8 de la funcion IterarBottom en esa iteración, el conjunto bottom consiste exactamente en todos los subconjuntos b tal que $b \subset D$, b consistente y $|b| = k - 1$.*

Demostración. Ya vimos que todos los elementos en *bottom* son subconjuntos de D. Veamos además que en cada iteración k se cumple que al finalizar la línea 8 de la función *IterarBottom* se cumple que b es consistente y que $|b| = k - 1$. Separamos en dos casos, $k = 1$ y $k > 1$. Caso $k = 1$: *bottom* se definió en la línea 4 del algoritmo como un conjunto cuyo único elemento es el conjunto vacío. Trivialmente, el conjunto vacío tiene $k - 1$ elementos y además es consistente.

Caso $k > 1$: Notar que en este caso *bottom* se definió en la iteración $k - 1$ en la línea 14 de la función *IterarBottom* como el conjunto de todos los elementos en *proxi-*

moBottom que estén marcados como consistentes. Veamos entonces que al momento de ejecutarse tal línea de la función, los elementos en *proximoBottom* que están marcados como consistentes son exactamente todos los subconjuntos pb de D tales que pb es consistente y $|pb| = k - 1$. Veamos entonces las dos inclusiones, llamo subKCosistenes al conjunto definido por todos los subconjuntos consistes s de D tales que $|s| = k - 1$ y llamo *proximoBottomConsistenes* al conjunto de elementos que están en *proximoBottom* marcados como consistentes al momento de ejecutarse la línea 14 de la función *IterarBottom* en la iteración $k - 1$.

$subKCosistenes \subseteq proximoBottomConsistentes$: Notemos que *proximoBottom* fue definido en la línea 1 de la misma función como el conjunto formado por todos los subconjuntos $pb \subset D$ tales que $|pb| = nBottom + 1$ y pb no es superconjunto de ningún *culprit*. Teniendo en cuenta que al momento de ejecutarse tal línea de la función la variable *nBottom* vale $k - 1$ para toda iteración k, resulta entonces trivial que en la iteración $k - 1$, $nBottom + 1$ vale $k - 1$ y por lo tanto, después de ejecutada la línea 1, el conjunto *proximoBottom* es exactamente el conjunto de todos los conjuntos pb tales que $|pb| = k - 1$ y pb no es superconjunto de ningún *culprit*. Dado que por el lema 7 todos los elementos en *culprits* son inconsistentes, resulta que s no puede ser superconjunto de algún *culprit*, pues s es consistente, por lo tanto s está en *proximoBottom* una vez ejecutada la línea 1 en la iteración $k - 1$. Notemos que por el lema 15 una vez iterados todos los elementos pb en el *for* de al línea 2 de la función, todos aquellos elementos pb serán marcados como consistentes si solo si son consistentes. Por lo tanto s será marcado como consistente una vez finalizado tal *for* y por lo tanto $s \in proximoBottomConsistentes$.

$proximoBottomConsistentes \subseteq subKCosistenes$: Sea entonces pb un elemento en *proximoBottom* marcado como consistente en la línea 14 de la función *IterarBottom* en la iteración $k - 1$. Ya vimos que $|pb| = k - 1$, pues $nBottom + 1 = k - 1$. Además por el lema 15 dado que pb fue marcado como consistente resulta que pb es consistente. Por lo tanto $pn \in subKCosistenes$.

□

Lema 15. *Una vez finalizado el for de la línea 2 de la función IterarBottom, todos los elementos en proximoBottom son marcados como consistentes si solo si son consistentes.*

Demostración. Notemos que el *for* de la línea 2 de la función *IterarBottom* itera sobre todos los elementos en *proximoBottom*. Sea entonces un elemento en pb en *proximoBottom* consistente. En caso de que sea subconjunto de algún elemento rg en *repairsGrandes* será marcado como consistente, lo cual es correcto pues todos los elementos en *repairsGrandes* son consistentes. En caso contrario se evaluará su consistencia por medio de la función *chequearConsistencia*, que al ser correcta retornará *verdadero*. En caso de que pb sea inconsistente, pb no puede ser subconjutno de ningun elemento en *repairsGrandes*, y por lo tanto se evaluará consistencia por medio de la función *chequearConsistencia*, que al ser correcta en este caso retornará *falso*. □

Lema 16. *La máxima cantidad de iteraciones que se pueden ejecutar en el while de repairsFinder es* $\lfloor n/2 \rfloor$ *en el caso* n *par y* $\lfloor n/2 \rfloor + 1$ *en el caso* n *impar.*

Demostración. Caso n impar: Teniendo en cuenta que *nBottom* se inicializa en 0, y que *nTop* se inicializa en n, y notando que la primera se incrementa en cada iteración en una unidad, y que la segunda va decrementando también de a una unidad, trivialmente la condición $nTop > nBottom$ será falsa solo cuando ocurra $nTop < nBottom$, lo cual sucederá una ves que se ejecuten $\lfloor n/2 \rfloor + 1$ iteraciones.

Caso n par: Con el mismo razonamiento, la condición $nTop > nBottom$ será falsa solo cuando las dos variables valgan lo mismo, y esto sucederá justo después de que se ejecuten $\lfloor n/2 \rfloor$ iteraciones, momento en el que las dos valdrán $\lfloor n/2 \rfloor$.

□

3. IMPLEMENTACIÓN

Tal como mencionamos en la introducción, parte de la motivación del presente trabajo es presentar una herramienta que acerque de manera *amigable* el lenguaje *Datalog +/-* al público general. De esta manera, por medio de una aplicación *web* queremos proveer la funcionalidad de crear ontologías, así como también la posibilidad de ejecutar consultas sobre las mismas, siendo estas consultas tolerantes a las inconsistencias que pudieran existir en los datos almacenados sobre esas ontologías.

En las páginas que siguen describiremos en detalle la implementación de esta herramienta. Cabe primero destacar que para construir nuestro sistema, hemos utilizado una herramienta ya existente, llamada *IRIS*, que brinda la posibilidad de crear ontologías y ejecutar consultas en el lenguaje. Nos enfocaremos entonces en describir cuál es el alcance de *IRIS*, detallando el límite entre las facilidades provistas por éste y las nuevas funcionalidades implementadas por nosotros. El código completo de este trabajo se puede encontrar en https://github.com/pfromer/tesis/tree/new_refactor.

3.1. IRIS

IRIS es una herramienta *open source* hecha en *JAVA* que permite construir ontologías en *Datalog +/-* y ejecutar consultas sobre las mismas. El código de *IRIS* se encuentra publicado en *Github* y se puede descargar desde *https://github.com/NICTA/iris-reasoner*. De los elementos que provee el lenguaje, *IRIS* brinda la posibilidad de definir *TGDs*, *hechos*, y *consultas*. Existen otras herramientas que permiten definir ontologías en *Datalog +/-* como *vLog4J*, que también es *open source* y *VADALOG*, siendo este último propietario.

3.1.1. Usabilidad

Para utilizar *IRIS* es necesario descargar el código de *github*, instalar *JAVA 8*, compilar el código en un archivo *.jar*, y luego ejecutarlo. Al momento de ejecutar el archivo *.jar* es necesario pasarle una serie de parámetros, entre ellos el *path* a un archivo de texto que contiene el *programa* a ejecutar, es decir que en este último se deben encontrar las *TGDs*, los *hechos* y las *consultas*. Las respuestas a las *consultas* son retornadas por *IRIS* en *standard output*. A grandes rasgos, podemos describir entonces a *IRIS* como una aplicación de consola que puede procesar un *programa* y devolver un resultado.

3.1.2. Sintaxis

La figura 3.1 ejemplifica la sintaxis que deberá tener el input para poder ser procesado por *IRIS*. En este input *IRIS* puede identificar una base de conocimiento (Σ_t, D)

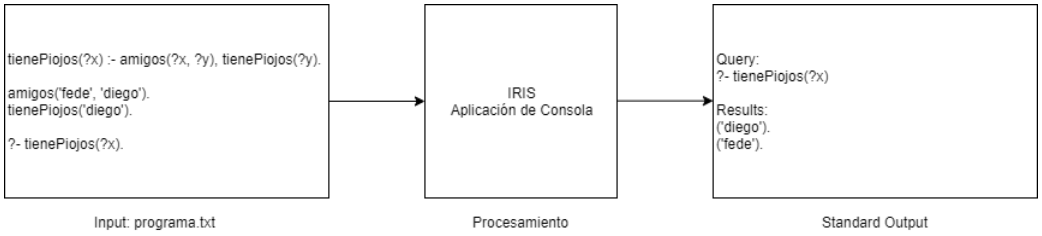

Fig. 3.1: IRIS: aplicacion de consola

donde Σ_t es un conjunto de *TGDs* y D es un conjunto de *hechos*. En ese mismo archivo se podrán incluir un conjunto de consultas, cuyas respuestas serán retornadas por *IRIS* en *standard output*. Veamos ahora en detalle cuál es la sintaxis que debe respetar cada uno de los elementos del lenguaje.

Constantes y Variables

Las constantes se deben notar entre comillas simples y las variables deben ser antecedidas por el símbolo de interrogación *?*.

Átomos

Los átomos constan de un nombre de predicado alfanumérico seguidos por un conjunto de argumentos encerrado entre paréntesis donde los argumentos están separados por coma. Cada argumento puede ser una constante o una variable.

TGDs

Para cada *TGD* se debe separar la cabeza y el cuerpo intermediando la cadena *":-"*, ubicando la cabeza a la izquierda y el cuerpo a la derecha. La cabeza constará de un solo átomo mientras que el cuerpo contendrá un conjunto de átomos separados por coma. Se debe ubicar un punto final a la derecha de cada *TGD*. Las variables que se encuentren en la cabeza y no se encuentren en ningún átomo del cuerpo serán consideradas por *IRIS* como *nulos*. El siguiente ejemplo muestra una *TGD* que podrá ser correctamente procesada por *IRIS*.

q(?x, ?y, ?z) :- p1(?x, ?y), p2('a').

Hechos

Los hechos constan de un solo átomo en donde todos los argumentos son constantes. Cada hecho debe finalizar con un punto a la derecha.

Consultas

Las consultas deben ser precedidas por la cadena *"?-"* y seguidas por un conjunto de átomos separados por coma, en donde los argumentos de cada átomo pueden ser tanto variables como constantes. Ejemplificamos ahora con la siguiente consulta:

?- p1(?x, 'a'), p2(?z).

Es importante mencionar que *IRIS* no permite cuantificar existencialmente las variables en las consultas. De esta manera, para que la respuesta no sea vacía, todo el cuerpo de la consulta deberá mapear homomórficamente con un conjunto de átomos en el *chase* que computará *IRIS*, donde todos esos átomos del *chase* deberán exclusivamente contener constantes en sus argumentos.

3.2. Extendiendo IRIS

Con el objetivo final de alcanzar *Datalog +/-* al usuario general, y queriendo ofrecer a la vez la posibilidad de responder consultas bajo ambientes inconsistentes, hemos extendido a *IRIS* con tres nuevas funcionalidades:

- Nueva *api* de manera que *IRIS* pueda ser consultada a través de la *web*.
- Nuevo módulo de manejo de inconsistencia de manera que *IRIS* pueda responder consultas bajo las semánticas *AR* e *IAR*.
- Posibilidad de cuantificar existencialmente las variables en las conusltas.

Detallaremos ahora nuestra implementación para cada una de estas funcionalidades.

3.2.1. Api Web

Para que *IRIS* pueda ser consultada a través de la *web*, necesitamos exponer una *api* pública, con una interfaz clara, que le permita al usuario definir un *programa*, donde se puedan especificar *restricciones de integridad*, *tgds*, *hechos*, *consultas*; y donde se pueda además especificar bajo cual *semántica* se quiere ejecutar a esas *consultas*.

Para poder responder las consultas, la implementación de esta *api* accederá a los otros módulos implementados en el mismo servidor, entre los cuales se encuentra *IRIS* y otro módulos de implementación propia.

La figura 3.2 muestra la arquitectura general del nuevo sistema *web*. La nueva *api* está expuesta en un servidor que recibe peticiones de clientes. El módulo *Api Web* solo es encargado de parsear el input recibido en formato *Json*, enviarle la consulta al *Módulo de Semánticas*, esperar su respuesta y devolverla al cliente otra vez en el formato *Json*. Cabe aclarar que todas las comunicaciones entre los módulos *Api Web*, *Módulo de Semanticas*, *Módulo de Repairs* y *IRIS* ocurren en un mismo proceso dentro del servidor a través de llamadas a funciones implementadas en *JAVA* dentro de esos módulos. De esta forma podemos ver al módulo *Api Web* como un mero intermediario entre las llamadas *web* de los clientes y los demás módulos del servidor que se encargan en su conjunto de procesar las consultas.

Veamos ahora cuáles son los *endpoints* que expone nuestra *api*, y cómo deben ser los *requests* junto con sus respuestas. La implementación de estos *endpoints* se puede encontrar en https://github.com/pfromer/tesis/tree/new_refactor/eclipse-workspace/iris-web/src/api.

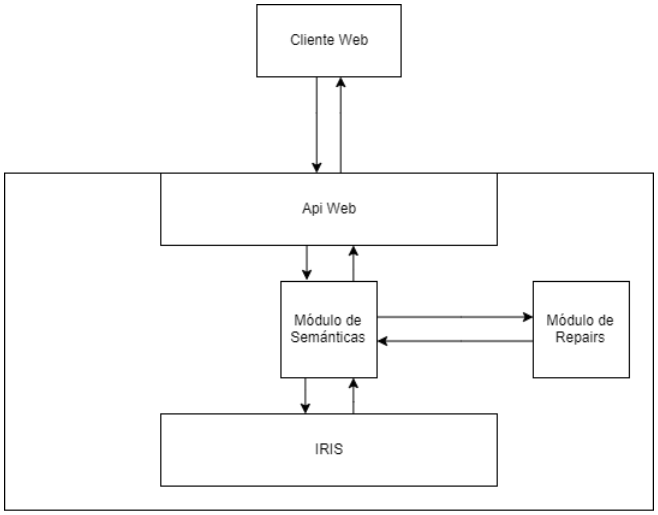

Fig. 3.2: Cliente - Servidor

Query Endpoint

Este *endpoint* permite realizar *consultas* para un conjunto de *Restricciones de Integridad*, *TGDs* y *hechos*. En la figura 3.3 se muestra en un ejemplo el formato esperado para el *body* que deben tener los *request* de este *endpoint*.

Endpoint Url: http://[url_base]/iris/query[1].

Http Method: Post.

Tal como se ejemplifica en la figura 3.3, el campo *ncs* es un arreglo en el que cada elemento debe ser un *Json* con el que campo *body* que consta de una serie de átomos separados por coma donde cada átomo debe seguir la sintaxis tal como se especificó en la sección 3.1.2. Cada uno de esos elementos será interpretado como el cuerpo de una *Restricción de Integridad*, donde las comas que separan cada átomo representan conjunción.

De manera similar, el campo *tgds* es también un arreglo de objetos *Json* donde cada elemento debe contener los valores *head* y *body*. En el campo *head* se debe incluir un solo átomo y en el campo *body* una secuencia de átomos separados por coma. En cuanto al campo *facts*, cada elemento en el arreglo será un objeto con un único campo *value* que deberá contener un solo átomo con parámetros constantes.

Con respecto al campo *queries*, cada elemento en el arreglo deberá ser un *Json* de dos campos, donde el campo *showInOuput* es un arreglo con las variables cuyos mapeos en el *chase* deberán mostrarse en el *output* de la consulta y donde el campo *body* es el cuerpo de la consulta en sí. Notar que las variables que aparezcan en el cuerpo de la consulta y no se incluyan en el arreglo del campo *showInOutput* son justamente las variables que queremos cuantificar existencialmente. En el caso en el que este arreglo sea un arreglo vacío, estaremos en el caso en el que todas las variables se deberán cuantificar existencialmente, siendo la consulta una *consulta conjuntiva booleana* tal como se la detalló en la sección 1.2.

[1] Se hará un *deploy* de esta *api* en los servidores de la facultad paro no se han definido aún las rutas.

```
{
    "program": {
        "ncs": [{
            "body": "r1(?x, ?y)"
        }],
        "tgds": [{
            "head": "r1(?z, ?x)",
            "body": "r1(?x, ?y), r2(?y)"
        }],
        "facts": [{
            "value": "r1('a', 'b')"
        }],
        "queries": [{
            "showInOutput": ["x"],
            "body": "r1(?x, ?y), r2(?y)"
        }],
        "semantics": "standard",
        "max_depth": 100
    }
}
```

Fig. 3.3: Formato esperado para una petición al Query Endpoint

Con respecto al campo *semantics*, este podrá contener alguno de los siguientes tres valores: *standard*, *AR* o *IAR*. En el caso en que se especifique *standard*, las consultas se ejecutarán bajo la semántica detallada en la sección 1.2.2, es decir sin manejo de inconsistencia, lo cual podrá ocurrir solo cuando todas las *restricciones de integridad* se cumplan para ese conjunto de *TGDs* y de *hechos*. En caso contrario se le avisará al cliente que las consultas no se pueden responder bajo esa semántica, dado que no se cumplen todas las restricciones, indicándose a la vez cuáles son aquellas que no se cumplen.

El campo max_depth indicará hasta cuantas iteraciones en el algoritmo del *chase* el *cliente* está dispuesto a esperar. Si el valor de este campo es *null* entonces no habrá un máximo de iteraciones definido y se seguirá ejecutando hasta que finalice la construcción del *chase* o bien hasta que ocurra un *timeout*. El valor del *timeout* será un valor fijo definido en la configuración del servidor.

En la figura 3.4 se puede ver un ejemplo de respuesta de este *endpoint* para el caso en el que en la petición del cliente el valor del campo *semantics* fue *standard* pero no se cumplieron algunas de las *restricciones de integridad* pasadas en el campo *ncs*. Vemos que en la respuesta, el campo *unsatisfied* es un arreglo donde se indican los índices de las *restricciones de integridad* que no fueron satisfechas en el arreglo de *ncs* que el cliente envió en la petición.

En al figura 3.5 se muestra un ejemplo de respuesta para una petición de un cliente

```
{
    "unsatisfied": [0, 2]
}
```


Fig. 3.4: Ejemplo de respuesta del Query Endpoint

que se encontró en alguno de estos dos casos:

- El valor del campo *semantics* es *AR* o *IAR*
- El valor del campo *semantics* es *standard* y se cumplen todas las *restricciones de integridad*.

En este caso la *api* devolverá un arreglo de respuestas que estará ordenado en el mismo orden en el que se encuentren sus respectivas consultas en la petición del cliente.


```
{
    "answers": [
        [
            ["a", "b"],
            ["c", "d"]
        ],
        true,
        []
    ]
}
```


Fig. 3.5: Ejemplo de respuesta del Query Endpoint

En el caso en el que el servidor no logre procesar la consulta antes del tiempo definido en la configuración del *tiemout*, la *api* responderá como se muestra en la figura 3.6.


```
{
    "error": "timeout"
}
```


Fig. 3.6: Respuesta con error de *timeout*

Repairs Endpoint

Hemos agregado además un endpoint que dado un conjunto de *restricciones, TGDs* y *hechos*, retorna el conjunto de *repairs*. La figura 3.7 muestra con un ejem-

plo el formato esperado que deben tener las peticiones a este *Repairs Endpoint*.

Endpoint Url: http://[url_base]/iris/repairs_finder.

Http Method: Get.

```
{
   "ncs": [{
       "body": "r1(?x, ?y)"
   }],
   "tgds": [{
       "head": "r1(?z, ?x)",
       "body": "r1(?x, ?y), r2(?y)"
   }],
   "facts": [{
       "value": "r1('a', 'b')"
   }],
   "max_depth": null
}
```

Fig. 3.7: Formato esperado para una petición al Repairs Endpoint

En la figura 3.8 se muestra un ejemplo de respuesta para el *Repairs Endpoint*.

```
{
    "repairs": [
        ["r1('a', 'b')."],
        ["r2('a', 'd').", "r1('c','b')"]
    ]
}
```

Fig. 3.8: Ejemplo de respuesta del Repairs Endpoint

3.2.2. Módulo de Semánticas

Veamos ahora en detalle cuál es la responsabilidad del *Módulo de Semánticas* y cómo está implementado. Veamos primero que la interfaz que expone este módulo consta de un sólo método al que llamamos *Execute*. Este método recibe un objeto *executeParams*, el cual es una instancia de una clase *JAVA* que contiene los mismos campos que se especificaron en el *Json* de la figura 3.3, es decir las *restricciones de integridad*, las *TGDs*, los *hechos*, las *consultas* y la *semántica* bajo la cual se desea ejecutar las consultas.

Dependiendo del valor en el campo *semantics*, el método *Execute* hará una llamada a alguno de estos tres métodos privados: *ExecuteStandard*, *ExecuteAR*, o *ExeciteIAR*. Veamos entonces cómo están implementados cada uno de estos tres métodos.

Método ExecuteStandard

Este método debe primero evaluar si las *restricciones de integridad* son satisfechas por las *TGDs* y los *hechos*. Para poder determinar lo anterior, consultará al módulo de *IRIS*, generando un nuevo *programa*, con la sintaxis especificada en la sección 3.1.2, en las que los *hechos* y las *TGDs* son las que recibió por parámetro, mientras que las *consultas* serán consultas conjuntivas *booleanas* en las que el cuerpo de cada consulta será el cuerpo de cada *restricción de integridad*.

Si el módulo de *IRIS* devolviera *true* para alguna de esas consultas, entonces determinamos que la *restricción* relacionada a esa consulta es violada por las *TGDs* y los *hechos*. El método indicará en su respuesta cuales son aquellas restricciones que no fueron satisfechas, entendiéndose así que es imposible ejecutar las *consultas* que se han requerido bajo esta semántica.

En el caso en el que ninguna de las restricciones fueran violadas, entonces este método enviará un nuevo *programa* a *IRIS*, con las mismas *TGDs* y *hechos* que en la llamada anterior pero ahora sí con las consultas que el cliente desea ejecutar. Para cada consulta se instanciará una respuesta, donde la respuesta puede ser o bien un valor *booleano* (en caso de que la consulta haya sido *booleana*) o bien una matriz, representando las filas y las columnas de la respuesta, donde cada columna se corresponderá con cada una de las variables en el arreglo *showInOutput* que se especificó para esa consulta.

La implementación de este módulo se puede encontrar en https://github.com/pfromer/tesis/tree/new_refactor/IRIS%2B-/src/org/deri/iris/semantic_executor.

Método ExecuteAR

Este método ejecutará las consultas bajo la semántica *AR*. Para ello deberá conocer primero el conjunto de *repairs* correspondientes a esas *restricciones*, *TGDs* y *hechos*, el cual lo obtendrá por medio del *Módulo de Repairs*, sobre el cual detallaremos su implementación más adelante.

Una vez obtenido el conjunto de *repairs*, el método *ExecuteAR* procederá de la siguiente manera: para cada *repair* armará un nuevo *programa* en el que los *hechos* serán los mismos *hechos* que se encuentren en ese *repair*. Para cada *programa*, las *consultas* y las *TGDs* serán siempre las mismas, es decir las que el método recibió por parámetro. Una vez enviados a ejecutar todos los programas al módulo de *IRIS*, se computará la intersección de las respuestas para cada consulta, en donde esta intersección consistirá en una intersección de matrices para el caso de las consultas no *booleanas* y en una conjunción lógica en el caso contrario.

Método ExecuteIAR

Ejecutaremos en este caso a las consultas bajo la semántica *IAR*. Al igual que en el caso anterior, el método *ExecuteIAR* obtendrá el conjunto de *repairs* por medio del *Módulo de Repairs*. Luego computará la intersección de los *repairs* y armará un solo *programa* en el que los *hechos* serán la intersección obtenida, mientras que las *TGDs* y las *consultas* serán las mismas que recibió por parámetro. La respuesta se

armará de igual manera que en el caso en el que ninguna *restricción* fuera violada en el método *ExecuteStandard*.

3.2.3. Módulo de Repairs

Este módulo es una implementación hecha en *JAVA* del algoritmo *RepairsFinder* presentado en la sección 2.1. El código completo de esta implementación se puede ver en https://github.com/pfromer/tesis/blob/new_refactor/IRIS%2B-/src/org/deri/iris/repairs_finder/RepairsFinder.java.

Si bien nuestro objetivo en esta ocasión es encontrar conjuntos maximales *consistentes*, cabe mencionar aquí que esta implementación puede ser fácilmente reutilizable para encontrar conjuntos maximales con respecto a otras características más allá de consistencia. Esto se debe a que los únicos parámetros que recibe esta implementación son el conjunto de *hechos* más una función que debe poder recibir un subconjunto de esos hechos y devolverá un booleano indicando si aquella propiedad para la cual se desea encontrar conjuntos maximales se cumple o no para ese subconjunto. En este caso la función que le pasamos por parámetro al *Modulo de Repairs* es una función que dado un conjunto de *hechos* llamará al módulo de *IRIS* para un conjunto de *restricciones* y *TGDs* fijas, con el objetivo de saber si aquellas *restricciones* se sastisfacen para esas *TGDs* y ese subconjunto de *hechos*. De esta manera el *Módulo de Repairs* desconoce totalmente a *IRIS*, ya que solo ejecuta la función recibida por parámetro sin conocer la implementación de dicha función[2].

Teniendo en cuenta que el algoritmo necesita computar reiteradas veces inclusión entre dos conjuntos, vale destacar que implementamos esta funcionalidad numerando a todos los *hechos* y manteniéndolos ordenado en todo momento. De esta manera, dado que todo subconjunto de *hechos* será una lista ordenada de esos hechos, computar inclusión entre estos dos conjuntos se hará a lo sumo en n pasos, donde n es el tamaño del conjunto más pequeño. La implementación de inclusión entre conjuntos se puede ver en el método *isSubSetOf* en la clase *AboxSubSet* en el siguiente link: https://github.com/pfromer/tesis/blob/new_refactor/IRIS%2B-/src/org/deri/iris/repairs_finder/AboxSubSet.java.

3.2.4. Consultas con cuantificadores existenciales

Como mencionamos en la sección 3.1.2, *IRIS* no posee la capacidad de responder consultas en las que se cuantifique existencialmente a las variables de la forma $Q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, ni tampoco consultas conjuntivas *booleanas* de la forma $Q() = \exists \mathbf{Y} \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$. En efecto *IRIS* solo puede responder consultas del tipo $Q(\mathbf{X}, \mathbf{Y}) = \Phi(\mathbf{X}, \mathbf{Y})$.

Veamos entonces cuales fueron las modificaciones que hemos añadido de manera que *IRIS* pueda responder todos los tipos de consultas ejemplificadas. Tengamos en cuenta que las variables que se cuantifiquen existencialmente podrán mapear tanto a constantes como a *nulos frescos* en el *chase*. De esta manera las modificaciones que hicimos consistieron primero en que al momento de evaluar la consulta, buscando áto-

[2] Teniendo en cuenta esta aclaración cabe mencionar que los métodos *ExecuteAr* y *ExecuteIAR* deben ser capaces de instanciar esta función y pasarla por parámetro al *Módulo de Repairs*.

mos en el *chase* que mapearan *homomórficamente* con el cuerpo de la misma, *IRIS* dejara de descartar necesariamente los *átomos* que tuvieran *nulos*. De esta manera, el método original *evaluateQuery* en la clase *StratifiedBottomUpEvaluationStrategy* fue modificado para que al evaluar una consulta pueda devolver tuplas en las que no necesariamente todas las posiciones contengan constantes sino también *nulos frescos*.

Una vez hecho este cambio agregamos una capa intermedia[3] que toma la lista de tuplas retornadas por la nueva versión del método *evaluateQuery* y solo se queda con aquellas columnas que figuran en el arreglo *showInOutput* que describimos en la sección 3.2.1. Una vez filtradas las columnas descartamos ahora sí a todas aquellas filas que contengan tuplas con elementos *nulos*, pues solo puede haber *nulos* en las columnas correspondientes a aquellas variables que se quieren cuantificar existencialmente, y justamente esas son las columnas que descartamos en el paso anterior, pues son las que no figuran en el arreglo *showInOutput*. Notar que de esta manera la respuesta final solo contiene tuplas con valores constantes.

Cabe mencionar que para el caso de las consultas conjuntivas *booleanas* la nueva capa intermedia no retornará ahora una lista de tuplas, sino que devolverá un valor booleano, siendo este *verdadero* solo cuando el resultado devuelto por el método *evaluateQuery* sea una lista de tuplas no vacía.

3.2.5. Profundidad máxima en el chase

Es importante mencionar que la propiedad *Bounded guard-depth property* para el fragmento *Guarded* tiene el valor de determinar una cota en la cantidad de iteraciones que se deberán ejecutar para computar las consultas pero que lamentablemente a fines prácticos termina siendo demasiado alta. Lo interesante de esta propiedad es que fija la cantidad de iteraciones para un determinado esquema R, lo cual hace que esta cota sea independiente de la cantidad de datos. Pero cuando intentamos implementar esta propiedad, que vale aclarar no está implementada en *IRIS*, notamos que los valores de estas cotas resultaban extremadamente altos, aún para programas pequeños.

A modo de ejemplo, el valor de δ, tal como se describe en la sección 1.2.4 es de $8{,}589934592E9$ para el siguiente *programa*:

```
p2(?y) :- p1(?x).
p1(?y) :- p2(?x).

q(?x) :- p2(?x).

p1('a').
q('b').
r1('b').
r2('c').
```

Por tal motivo el servidor implementa un *timeout* como así también la posibilidad de que el cliente le pueda indicar la máxima cantidad de iteraciones que está

[3] ver código fuente de esta capa en la clase *QueryResult*.

dispuesto a esperar a través del campo max_depth como se mostró en la figura 3.3.

3.3. Cliente Web

Nos enfocaremos ahora en detallar la funcionalidad de nuestro *cliente*, el cual está conformado por una sola página *web* que brindará al usuario la posibilidad de ejecutar consultas al servidor de manera sencilla. En la figura 3.9 se puede ver una captura de pantalla de tal página. Durante el resto de esta sección describiremos el comportamiento de cada uno de los componentes que se muestran en la captura, así como su implementación.

Dicha implementación se puede encontrar en `https://github.com/pfromer/tesis/tree/new_refactor/NodeProject/src`.

3.3.1. Parser en el cliente

Tal como se muestra en la figura 3.9, en la pantalla del *cliente* se muestran dos editores de texto. El contenido de estos dos editores puede ser o bien ingresado por el usuario o bien cargado desde un archivo de texto almacenado en al computadora del cliente, solo después de que el usuario haya hecho *click* en el botón *Load Program*. En el editor izquierdo se deberán ingresar las *restricciones*, *claves*, *tgds* y *hechos*, y en el derecho las *consultas*. Si el texto hubiera sido ingresado a través de la carga de un archivo, el *cliente* se encargará de mostrar las *consultas* que hubiere en ese archivo en el editor derecho, separándolas de los demás elementos que se mostrarán en el editor *izquierdo*.

Para poder hacer esta separación, el *cliente* cuenta con un parser propio, el cual, a través de una serie de *expresiones regulares*[4] estará encargado de *parsear* los textos ingresados en los editores, detectar posibles errores de sintaxis, y luego instanciar nuestros objetos en *Javascript* los cuales representarán cada uno de los elementos ingresados. Hablaremos sobre estos objetos en la sección 3.3.3, pero detallaremos primero cuál es la sintaxis esperada por nuestro *parser*.

3.3.2. Sintaxis

El hecho de que el cliente posea su propio *parser* brinda la posibilidad no solo de detectar errores sintácticos antes de que las consultas sean enviadas al servidor, sino de también definir una sintaxis propia, no necesariamente igual a la que se mostró en la sección 3.1.2, la cual deberemos respetar de todos modos al momento de generar el *input* para *IRIS*. Hemos mantenido de todas maneras en gran medida aquella sintaxis, pues las variables, constantes, átomos y *hechos* mantendrán el mismo formato. Nos enfocaremos entonces en los elementos que han cambiado su formato esperado, como las *tgds* y las *consultas*, así como también en aquellos nuevos elementos desconocidos por *IRIS*, como las *restricciones de integridad* y las *claves*.

[4] La implementación de estas expresiones regulares se encuentra en el archivo *regexService*.

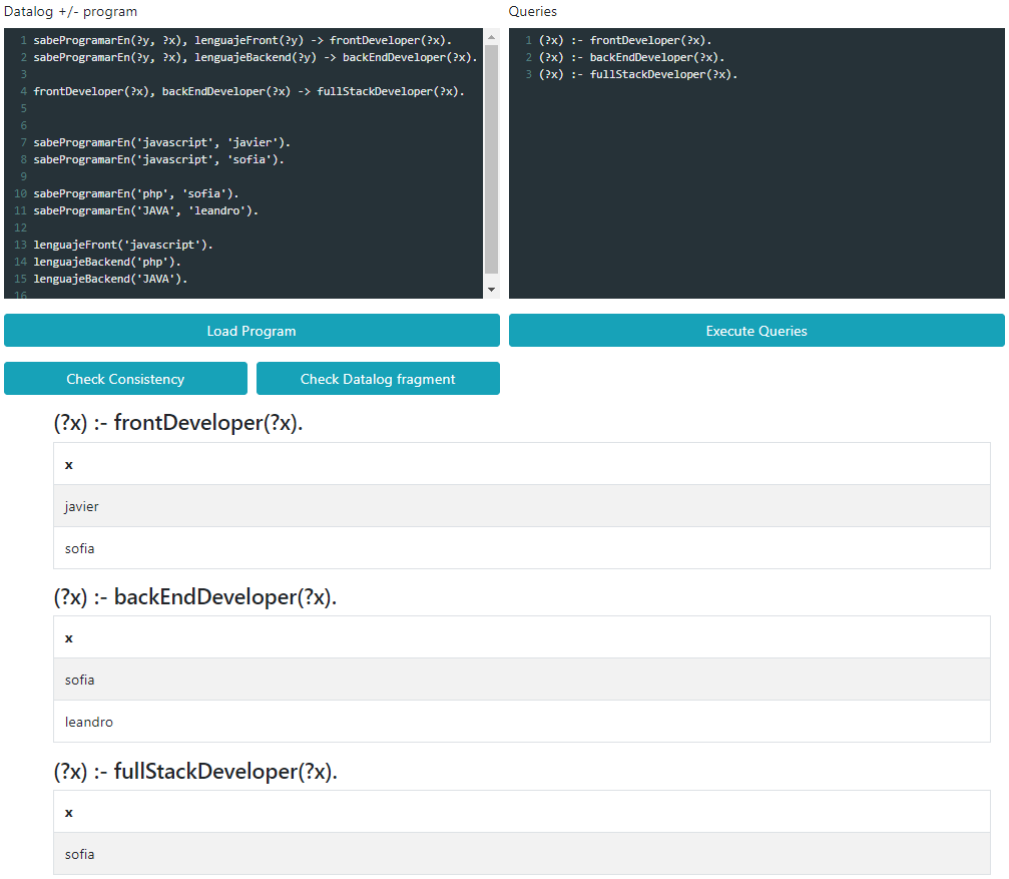

Fig. 3.9: Cliente Web

TGDs

Las *TGDs* deberán respetar el siguiente formato: la *cabeza* y el *cuerpo* no se encontrarán separadas por la cadena ":-", sino por la cadena "->"; además la *cabeza* no se encontrará a la izquierda y el *cuerpo* a la derecha, sino al revés. El siguiente es un ejemplo válido de una *TGD* para nuestro cliente.

sabeProgramarEn(?x, ?y), lenguajeFront(?x) ->frontDeveloper(?y).

Consultas

La sintaxis de las consultas deberá poder diferenciar aquellas variables que se quieran cuantificar existencialmente de aquellas que no. De esta manera se separará el cuerpo de la consulta de una lista de variables que no se quiere cuantificar existencialmente. Estas lista de variables se encontrarán encerradas por paréntesis, y cada una de ellas separadas por coma. El cuerpo de la consulta se encontrará a la derecha de la cadena ":-" tal como se muestra en el siguiente ejemplo:

(?x) :- sabeProgramarEn(?x, ?y), lenguajeFront(?x).

Restricciones de integridad

Las *restricciones* deberán respetar el siguiente formato: se deberá mostrar el cuerpo, seguido de la cadena "->", seguido de o bien el caracter ⊥, o bien la cadena *"bottom"*. Los siguientes son ejemplos correctos de *restricciones* en la sintaxis de nuestro cliente:

- *sabeProgramarEn(?x, 'pedro') ->⊥.*
- *casadoCon(?x, ?y), soltero(?y) ->bottom.*

Claves

Las *claves* se notarán siempre con la la letra *k* seguida del nombre del predicado sobre el cual se quiere definir la *clave* y un arreglo con los números de posiciones que definirán la clave. El nombre del predicado y el arreglo se encierran entre paréntesis. En el siguiente ejemplo se define que las posiciones de parámetro 1 y 3 son una clave sobre el predicado *p*.

k(p[1,3]).

3.3.3. Objetos del modelo

Tal como comentamos en la sección anterior, por medio de las expresiones regulares definidas en nuestro *cliente*, iremos identificando los elementos del *programa* introducidos en los editores y en base a esto instanciaremos objetos que representarán a cada uno de aquellos elementos. De esta manera hemos implementado un *builder* para cada tipo de elemento del lenguaje, es decir un *builder* para las *TGDs*, otro para las *consultas* y lo mismo para las *restricciones de integridad*, *claves* y *hechos*. De esta forma, en base a la expresión regular que haga *match* con cada línea del texto ingresado en el editor, nuestro parser sabrá a cual *builder* llamar, y le pasará por parámetro la línea de texto correspondiente. Así, cada *builder* debe implementar el método *build(line)* en donde *line* es una línea de texto ingresada en el editor que cumple con el formato esperado por ese *builder*. Cómo ejemplo, el *tgdBuilder* espera que *line* sea una línea de texto que cumpla con el formato especificado para las *TGDs* en la sección anterior.

Cada uno de los objetos devueltos por el método *build(line)* de nuestros *builders* implementará una serie de métodos concernientes al elemento del lenguaje que esté representando. Por ejemplo el objeto construido por el *tgdBuilder* implementa el método *isGuarded*, que retornará *true* solo cuando la *TGD* en cuestión pertenezca al fragmento *guarded*.

Para ver la implementación completa de estos *builders* ver carpeta *builders* en el apéndice.

3.3.4. Método toJson

En particular, cada uno de esos objetos deberá implementar el método *toJson()*, lo cual permitirá que podamos construir el *json* que enviaremos al servidor. Dicho *json*

deberá cumplir el formato mostrado en la figura 3.3. De esta manera, al momento en el que el usuario haga *click* en el botón *Execute Queries*, se construirá un objeto *json*, en base a ir llamando al método *toJson()* de cada uno de los objetos anteriores.

Veamos ahora algunos ejemplos del resultado esperado por el método toJson().

Ejemplo 20. *Sea tgdObj el resultado de ejecutar:*

tgdBuilder.build("sabeProgramarEn(?x, ?y), lenguajeFront(?x) ->frontDeveloper(?y).")

Entonces, al ejecutar el método tgdObj.toJson() obtendremos el siguiente json como resultado:

```
{
    "head": "frontDeveloper(?y)",
    "body": "sabeProgramarEn(?x, ?y), lenguajeFront(?x)"
}
```

el cual será agregado al campo TGDs del json presentado en la figura 3.3.

Ejemplo 21. *Sea queryObj el resultado de ejecutar:*

queryBuilder.build("(?x) :- frontDeveloper(?x)")

Entonces, al ejecutar el método queryObj.toJson() obtendremos el siguiente json como resultado:

```
{
    "showInOutput": ["x"],
    "body": "frontDeveloper(?x)"
}
```

Especial atención requiere la implementación del método *toJson()* en los objetos devueltos por el *keyBuilder*. Recordemos que las *claves* no son parte de la interfaz de nuestra *Api Web* y que *IRIS* no implementa *claves*, pero teniendo en cuenta que nuestra *api* implementa *restricciones de integridad* podemos entonces reescribir las *claves* como tales *restricciones*. Veremos en la próxima sección que nuestro *cliente* solo hará tal reescritura cuando las *claves* sean *no conflictivas*, tal como se las describió en la sección 1.2.7.

Ejemplo 22. *Sea p un predicado ternario y sea keyObj el resultado de ejecutar:*

queryBuilder.build("k(p[1])")

En este caso estamos diciendo que dadas dos instancias de p, a igual valor del primer parámetro en cada instancia, necesariamente no podrá ocurrir que cada una de las instancias tenga una constante diferente en el segundo parámetro, y lo mismo para el tercer parámetro. Asumiendo que esta clave es no conflictiva con todas las TGDs del programa, podemos decir que esta clave se cumplirá solo cuando ninguna de las dos siguientes restricciones fallen:

- $p(x, y_1, z_1), p(x, y_2, z_2), y_1 \neq y_2 \rightarrow \bot$.
- $p(x, y_1, z_1), p(x, y_2, z_2), z_1 \neq z_2 \rightarrow \bot$.

Por lo tanto, en este caso el resultado de ejecutar keyObj.toJson() será un arreglo representando ambas restricciones:

```
[{
        "body": "p(x, y1, z1), p(x, y2, z2), y1 != y2"
    },
    {
        "body": "p(x, y1, z1), p(x, y2, z2), z1 != z2"
    }
]
```

Veremos más adelante que el cliente mantendrá un *mapa* que indicará para toda *restricción* enviada al servidor a cual número de línea en el *programa* corresponde esa restricción, pudiendo ser esa línea en el programa una *restricción* o bien una *clave*. De esta manera el *cliente* podrá indicarle al usuario cuáles fueron aquellas *claves* o *restricciones* que no se satisfacen en el *programa* ingresado.

3.3.5. Validación de claves no conflictivas

Tal como mencionamos, el *cliente* hará una validación de las *claves* ingresadas en el programa, y en el caso en el que alguna clave ingresada no fuera *no conflictiva*, se le avisará al usuario que ese tipo de *claves* no son soportadas por el sistema. Para poder hacer este chequeo, todos los objetos instanciados en el cliente que representan las *claves* del programa implementan el método *isNonConflicting(tgd)*, cuya implementación cumple con la definición de no conflictividad presentada en la sección 1.2.7:

```
1 isNonConflicting : function(tgd){
2           return this.predicate != tgd.head.predicate.name ||
3             (!this.keyPositions.isProperSubsetOf(tgd.xPositionsInHead()) && tgd.
                  allNullsAppearOnlyOnceInTheHead());
4           }
```

En el mismo sentido, el objeto que representa el *programa* completo, definirá si una *key* es no conflictiva utilizando la implementación de no conflictividad anterior:

```
1 isNonConflicting: function(key){
2   return this.tgds.every(tgd => key.isNonConflicting(tgd));
3 }
```

Para entender la implementación de esta funcionalidad en su totalidad será necesario desde ya entender el comportamiento de todas las propiedades y métodos que se invocan desde la función *isNonConflicting*, cuyas implementaciones se encuentran en el apéndice en los archivos *tgdBuilder*, *keyBuilder* y *arrayPrototype*. Creemos de

todas maneras que los nombres de estos métodos y propiedades son lo suficientemente descriptivos como para que el lector pueda imaginar su comportamiento sin necesidad de leer el código completo.

3.3.6. Experiencia de usuario

Enumeraremos ahora todas las funcionalidades brindadas por nuestro cliente, detallando como será la *experiencia del usuario* en cada caso.

Errores de sintaxis

Los errores sintácticos se mostrarán en rojo en el editor. Cuando el usuario intente ejecutar una consulta sobre un *programa* que tenga errores sintácticos, se le mostrará una alerta avisándole que debe corregir dichos errores. Esta implementación se encuentra en nuestra clase *Editor* que captura todos los eventos que suceden en los editores. Para cada uno de estos eventos, la clase validará los cambios contra todas nuestras expresiones regulares; de esta manera toda línea en el editor que no haga *match* con ninguna expresión será marcada en rojo, tal como se muestra en la siguiente pantalla:

Fig. 3.10: Error de sintaxis

Validación de fragmento

El usuario podrá validar también el fragmento de *Datalog +/-* al que pertenece el programa ingresado. Al hacer *click* en el botón *Check Datalog Fragment*, aparecerá un cartel en pantalla indicando cual es el fragmento. Tal funcionalidad se pudo lograr gracias a la implementación del método *fragment()* en los objetos devueltos por la clase *tgdBuilder*. Esto quiere decir que cada objeto *tgd* instanciado en nuestro cliente puede *decir* a qué fragmento pertenece.

Para una futura versión de este *cliente* dicha funcionalidad podría ser utilizada para definir una estrategia particular de resolución de las consultas. Por ejemplo, si el *programa* perteneciera al fragmento *lineal* se podría llamar a un servicio de reescritura que reescriba las consultas tal como se describió en la figura 1.3 de la sección

First-order rewritability y una vez obtenida la reescritura enviar tales consultas a un motor de base de datos *SQL*.

Validación de consistencia

El cliente provee también la posibilidad de validar la consistencia de un *programa* sin necesidad de ejecutar consultas. En este caso, cuando el usuario haga *click* en el botón *Check Consistency*, el *cliente* aramará un *json* con la misma estructura que se muestra en la figura 3.3 donde el campo *queries* será un campo vacío. De esta manera, en caso de que el programa fuera inconsistente, la *Api Web* retornara una respuesta con el formato que se muestra en la figura 3.4, indicando los índices de las restricciones que no se cumplieron en el arreglo enviado por el cliente en el campo *ncs*. El *cliente* mantendrá una estructura de *diccionario* que indicará a cual número de línea de texto en el programa ingresado corresponde cada índice de *restricción* enviada en la petición. De esta manera se marcarán en color verde aquellas *restricciones* o *claves* que violen la consistencia, tal como se muestra en la figura 3.11.

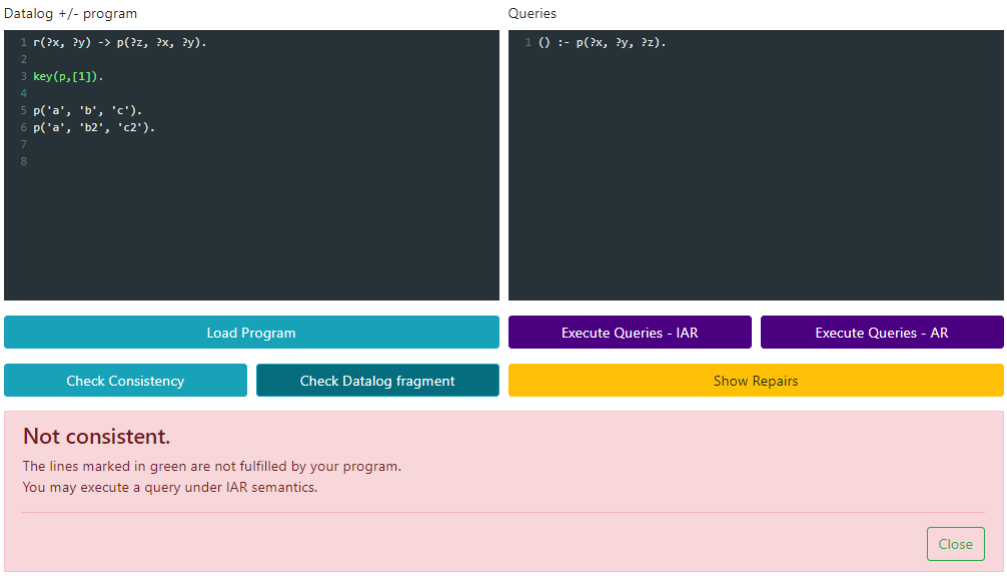

Fig. 3.11: Ejecuar bajo AR - IAR

Vemos que además en este caso aparecen dos nuevos botones que brindarán la posibilidad de ejecutar las consultas bajo la semántica *AR* o *IAR* más un nuevo botón que brindará la posibilidad de mostrar el conjunto de *repairs* para el *programa* ingresado.

Ejecutando consultas

Cuando el usuario presione en el botón *Execute Queries* tal como se muestra en la figura 3.9, se enviará el *json* a la *api web* tal como se muestra en la figura 3.3. Hay dos posibilidades, o bien el programa es consistente o bien no lo es. En el primer caso se mostrarán las respuestas para cada consulta tal como se muestra en la figura 3.9.

En el segundo caso, se ocultará el botón de *Execute Queries* y se mostrarán los dos botones que brindan la posibilidad de ejecutar consultas bajo semántica *AR* o *IAR*. Al presionar alguno de esos botones se mostrarán ahora sí los resultados de ejecutar la consultas bajo la semántica elegida.

Manejo de error de timeout

En el caso en el que el servidor responda con error de *timeout* o en el caso en el que el servidor tarde demasiado en responder[5], se le ofrecerá al usuario la posibilidad de definir la máxima cantidad de iteraciones a ejecutar en el algoritmo del *chase*. De esta manera, el *cliente* hará una segunda petición al servidor pero asignándole al campo *max_depth* el valor ingresado por el usuario.

[5] El *cliente* definirá por configuración cuanto tiempo está dispuesto a esperar la respuesta del servidor.

Bibliografía

[1] C. Beeri y M. Y. Vardi. *The implication problem for data dependencies*. En *Proc. ICALP-1981*, pp. 73–85, 1981.

[2] A. Calı, G. Gottlob, y M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. En *Proc. KR-2008, pp. 70–80, 2008.*

[3] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. En *Proc. KR-2008, pp. 70–80, 2008.*

[4] A. Deutsch, A. Nash, and J. B. Remmel. The chase revisited. En *Proc. PODS-2008, pp. 149–158, 2008.*

[5] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. En *Theor. Comput. Sci., 336(1):89–124, 2005.*.

[6] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. En *J. Comput. Syst. Sci., 28(1):167–189, 1984.*

[7] M. Y. Vardi On the complexity of bounded-variable queries. En *Proc. PODS-1995, pp. 266–176, 1995.*

[8] Andrea Calì, Georg Gottlobc, Thomas Lukasiewicz. A General Datalog-Based Framework for Tractable Query Answering over Ontologies En *QUE PONGO ACA.*

[9] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. En *Proc. of IJCAI 2005, pages 454–459, 2003.*

[10] Y. Ma and P. Hitzler. Paraconsistent reasoning for owl 2. En *Proc. of RR 2009, pages 197–211, 2009.*

[11] B. Parsia, E. Sirin, and A. Kalyanpur. Debugging OWL ontologies. En *Proc. of WWW 2005, pages 633–640, 2005.*

[12] P. Haasa, F. van Harmelen, Z. Huang, H. Stuckenschmidt, y Y. Sure. A framework for handling inconsistency in changing ontologies. En *Proc. of ISWC 2005, 2005.*

[13] D. Lembo and M. Ruzzi. Consistent query answering over description logic ontologies. En *Proc. of RR 2007, 2007.*

[14] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. En *Artificial Intelligence, 57:227–270, 1992.*

[15] M. Winslett. Updating Logical Databases. En *Cambridge University Press, 1990*

[16] Zhe Wang, Peng Xiao, and Kewen Wang. Practical Datalog Rewriting for Existential Rules En *Griffith University, Australia*

[17] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological Queries: Rewriting and Optimization En *Department of Computer Science, University of Oxford. Oxford-Man Institute of Quantitative Finance, University of Oxford, UK. Institute for the Future of Computing, University of Oxford, UK*