

Integración de Bases de Conocimiento

Clase 3 – Lenguajes Ontológicos.

Profesores: Maria Vanina Martinez y Ricardo Rodriguez

Esquema

En esta clase veremos:

- Conocimiento Ontológico
 - Representación: lenguajes ontológicos
- Lógicas de Descripción
- Ontology-based Data Access (OBDA)
- Complejidad de circuitos
- Datalog+/-



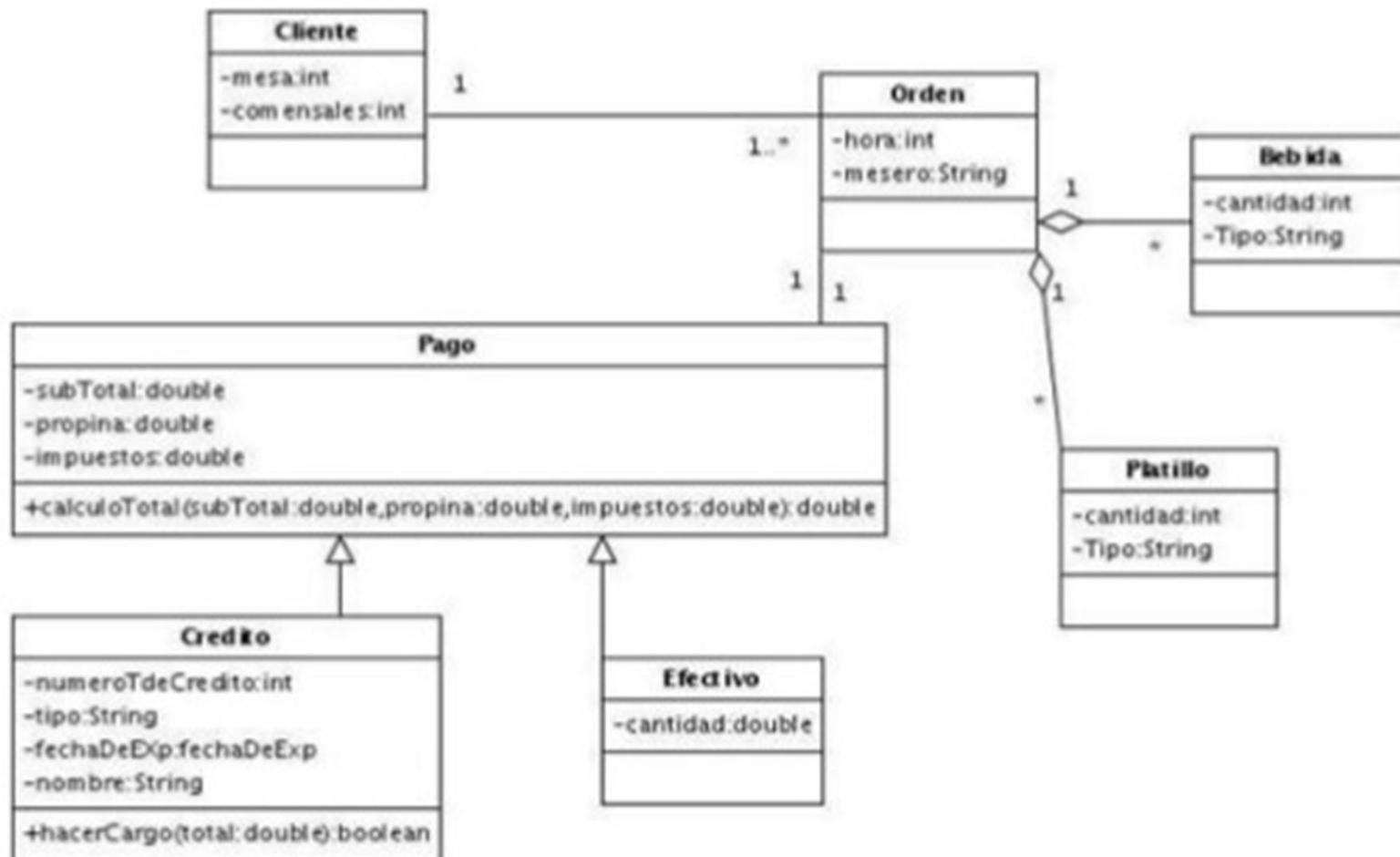
Conocimiento ontológico

- *Ontología*: es un esquema de representación que describe una conceptualización formal de un dominio de interés.
- Una *conceptualización* es una estructura semántica intensional que codifica conocimiento implícito que restringe la estructura de una porción del dominio.
- Una ontología es una especificación (parcial) de esta estructura:
 - Usualmente es una teoría *lógica* que expresa la conceptualización explícitamente en un lenguaje (declarativo).
- Facilita el *reúso* e *intercambio* de conocimiento:
 - Define el vocabulario con el cual las consultas y aserciones se intercambian entre agentes.

Conocimiento ontológico

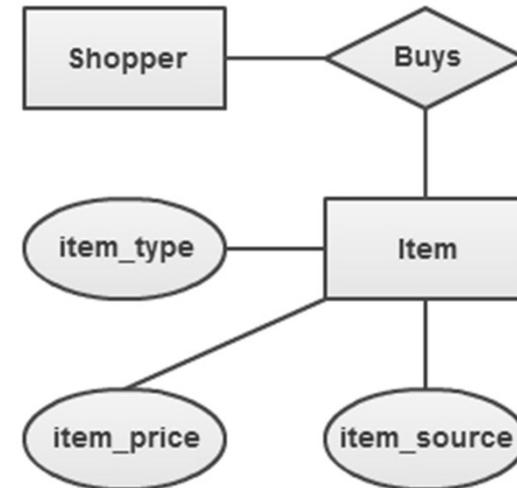
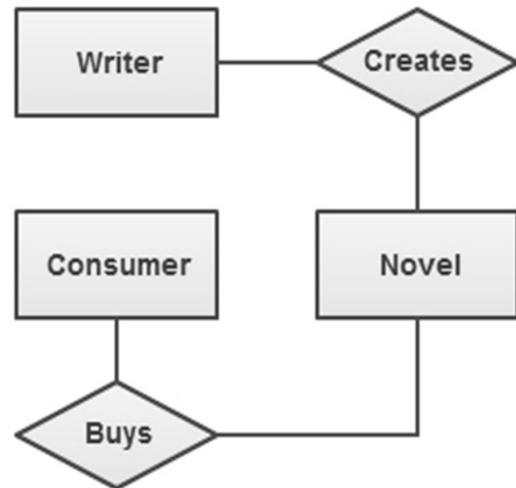
- Una conceptualización es una *vista abstracta y simplificada* del mundo que queremos representar.
- Cada *base de conocimiento* o sistema basado en conocimiento está asociado a una conceptualización (explícita o implícitamente).
 - Para estos sistemas, lo que *existe* es lo que se *representa*.
- Cuando el conocimiento de un dominio se representa en un formalismo declarativo, el conjunto de objetos que puede representarse se llama el *universo de discurso*.
 - Este conjunto de objetos y las relaciones describibles entre ellos se reflejan en el *vocabulario* de representación con el cual el programa basado en conocimiento representa el conocimiento.

Ejemplo Conceptualización



Fuente Imagen: <https://es.slideshare.net/nedowwhaw/diagrama-de-clases-16208245>

Ejemplo Conceptualización



Fuente Imagen Izquierda: <https://creately.com/blog/diagrams/er-diagrams-tutorial/>

Fuente Imagen Derecha: <https://sites.google.com/site/bsiscapstone/capstone-manuscript/chapter4/ultimate-guide-to-er-diagrams>

Conocimiento ontológico



Conocimiento ontológico

- En el contexto de *Inteligencia Artificial* (IA), o *Representación de Conocimiento y Razonamiento* (KR&R), podemos describir la ontología de un programa definiendo un conjunto de *términos representacionales*.
 - Las definiciones de nombres de *entidades* en el universo de discurso (clases, relaciones, funciones, y otros objetos) se asocian con descripciones de lo que los nombres significan y axiomas formales que restringen la interpretación y uso *bien formado* de esos términos.
 - Formalmente, se puede decir que una ontología *describe* una *teoría lógica*.

Conocimiento ontológico

- La especificación de una ontología comprende varios niveles:
 - *Meta-nivel*: especifica un conjunto de *categorías* a modelar.
 - *Nivel Intensional*: especifica un conjunto de elementos conceptuales (instancias de categorías) y de *reglas* para describir la estructura conceptual del dominio.
 - *Nivel Extensional*: especifica un conjunto de instancias de los elementos conceptuales descriptos por el nivel intensional.

Representación de conocimiento

- KR&R se enfoca en métodos para proveer descripciones de alto nivel del mundo, que pueden usarse para construir aplicaciones inteligentes.
 - Aquí, “*inteligencia*” es la habilidad de un sistema de encontrar consecuencias *implícitas* de conocimiento explícito.
- Los enfoques de KR se dividen en dos categorías:
 - *Formalismos basados en lógica*: evolucionaron de la intuición de que el cálculo de predicados se puede usar para capturar hechos sobre el mundo de manera no ambigua.
 - *Representaciones no basadas en lógica*: inspiradas en nociones más bien cognitivas; por ejemplo, estructuras de redes, y representaciones basadas en reglas derivadas de experimentos sobre actividades humanas.

Representación de conocimiento

- En el enfoque basado en *lógica*, el lenguaje de representación es en general una variante del cálculo de predicados de primer orden y la tarea de razonar implica la verificación de *consecuencias lógicas*.
- Los enfoques *no lógicos*, se basan en interfaces gráficas y el conocimiento es representado en estructuras de datos *ad hoc*.
 - Redes semánticas [Quillian1967]
 - *Frames* [Minsky1981]
 - Caracterizan el conocimiento y el razonamiento por medio de estructuras cognitivas en forma de redes.

Representación de conocimiento

- Los sistemas basados en redes son mas atractivos desde el punto de vista *práctico*.
- Sin embargo, su falta de caracterización *semántica* es un problema importante.
- Por esto, cada sistema se comportaba de manera diferente a los otros, aun cuando los componentes se veían muy parecidos o con nombres de relaciones idénticas.
- Sin embargo, puede dársele una semántica en *FOL* (al menos a un conjunto central de sus características).
- Las lógicas de descripción surgen de la *combinación* de ambos enfoques.

Lógicas de descripción

- Comenzaron a desarrollarse bajo el nombre de “*Sistemas Terminológicos*”, estableciendo la terminología básica adoptada en el modelamiento de un dominio.
- Luego, el énfasis fue en el conjunto de constructores formadores de *conceptos* admitidos en el lenguaje.
- Más recientemente la atención del I+D en el área se orientó hacia las propiedades de los sistemas lógicos subyacentes y se acuñó el término “Lógicas de Descripción” (*Description Logics* o DLs).

Lógicas de descripción

- Una *familia* de formalismos de representación de conocimiento basados en lógica:
 - Describen el *dominio* de interés en términos de *conceptos* (clases), *roles* (relaciones) e *individuos*.
 - Semántica formal (basada en *teoría de modelos*):
 - Corresponden a fragmentos decidibles de FOL.
 - Relacionadas con Lógicas Proposicionales Modales y Lógicas Dinámicas.

Lógicas de descripción

- Una *familia* de formalismos de representación de conocimiento basados en lógica:
 - Proveen servicios de *inferencia*:
 - Existen procedimientos sanos y completos para problemas específicos de *razonamiento*: inferencia lógica (sentencias atómicas o conjuntivas), respuesta a consultas, inferencia tolerante a la inconsistencia, etc.
 - Sistemas *implementados* con un alto grado de optimización que permite resolver problemas de la Web Semántica.

Lógicas de descripción

- Se asumen dos *alfabetos* de símbolos disjuntos:
 - Uno denota los conceptos atómicos – predicados *unarios*.
 - El otro para expresar relaciones entre conceptos (predicados *binarios*).
- El dominio de interpretación es arbitrario y puede ser infinito:
 - La posibilidad de dominios infinitos y la suposición de *mundo abierto* distinguen a las DLs de los lenguajes de bases de datos clásicos.
- Las características de cada DL están definidas por los *constructores* que establecen relaciones entre objetos.

DLS: Lenguaje de descripción

- Un lenguaje de descripción se caracteriza por un conjunto de *constructores* para crear conceptos complejos y roles a partir de los básicos:
 - Los conceptos corresponden a *clases*: interpretados como un conjunto de objetos.
 - Roles corresponden a *relaciones*: interpretados como relaciones binarias entre objetos.
- La semántica formal está dada en términos de *interpretaciones* (FOL).

DLS: Semántica (FOL)

- Una interpretación $I = (\Delta^I, .^I)$ consiste de:
 - Un conjunto no vacío Δ^I , con dominio I
 - una función de interpretación $.^I$, que mapea
 - cada individuo a a un elemento a^I de Δ^I
 - cada concepto atómico A a un subconjunto A^I de Δ^I
 - cada role atómico P a un subconjunto P^I de $\Delta^I \times \Delta^I$
- La función de interpretación se extiende a conceptos complejos y roles de acuerdo con la estructura sintáctica.

DLS: Constructores

Construct	Syntax	Example	Semantics
atomic concept	A	Doctor	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
atomic role	P	hasChild	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
atomic negation	$\neg A$	\neg Doctor	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$
conjunction	$C \sqcap D$	Hum \sqcap Male	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
(unqual.) exist. res.	$\exists R$	\exists hasChild	$\{ a \mid \exists b. (a, b) \in R^{\mathcal{I}} \}$
value restriction	$\forall R.C$	\forall hasChild.Male	$\{ a \mid \forall b. (a, b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}} \}$
bottom	\perp		\emptyset

C, D denotan conceptos arbitrarios y R un rol arbitrario.

- Estos constructores forman el lenguaje básico AL de la familia de lenguajes AL.

DLS: Constructores

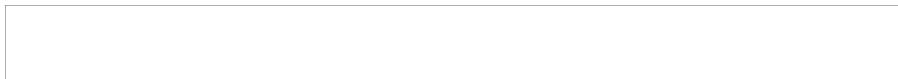
Construct	\mathcal{AL}	Syntax	Semantics
disjunction	\mathcal{U}	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
top		\top	$\Delta^{\mathcal{I}}$
qual. exist. res.	\mathcal{E}	$\exists R.C$	$\{ a \mid \exists b. (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \}$
(full) negation	\mathcal{C}	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
number restrictions	\mathcal{N}	$(\geq k R)$	$\{ a \mid \#\{b \mid (a, b) \in R^{\mathcal{I}}\} \geq k \}$
		$(\leq k R)$	$\{ a \mid \#\{b \mid (a, b) \in R^{\mathcal{I}}\} \leq k \}$
qual. number restrictions	\mathcal{Q}	$(\geq k R.C)$	$\{ a \mid \#\{b \mid (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \geq k \}$
		$(\leq k R.C)$	$\{ a \mid \#\{b \mid (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\} \leq k \}$
inverse role	\mathcal{I}	R^-	$\{ (a, b) \mid (b, a) \in R^{\mathcal{I}} \}$
role closure	reg	\mathcal{R}^*	$(R^{\mathcal{I}})^*$

- Se han investigado muchos constructores y sus combinaciones, dando lugar a *distintas familias* de DLS con diferente expresividad y complejidad computacional.

DLs: Razonamiento / Consultas

- El problema de razonamiento clásico en DLs es *inclusión de conceptos* (*concept subsumption*), denotado $C \sqsubseteq D$:
 - Se chequea si un concepto D se considera *más general* que el concepto $C \Rightarrow$ si C siempre denota un subconjunto de lo que denota D .
- Otro tipo de razonamiento típico en DLs es *satisfacción de conceptos*, es el problema de chequear si un concepto puede denotar un conjunto *no vacío* (concepto vacío).
 - Satisfabilidad es un caso especial de inclusión de conceptos, asumiendo que D es el conjunto vacío (concepto insatisfacible) $\Rightarrow C \sqsubseteq \emptyset ?$

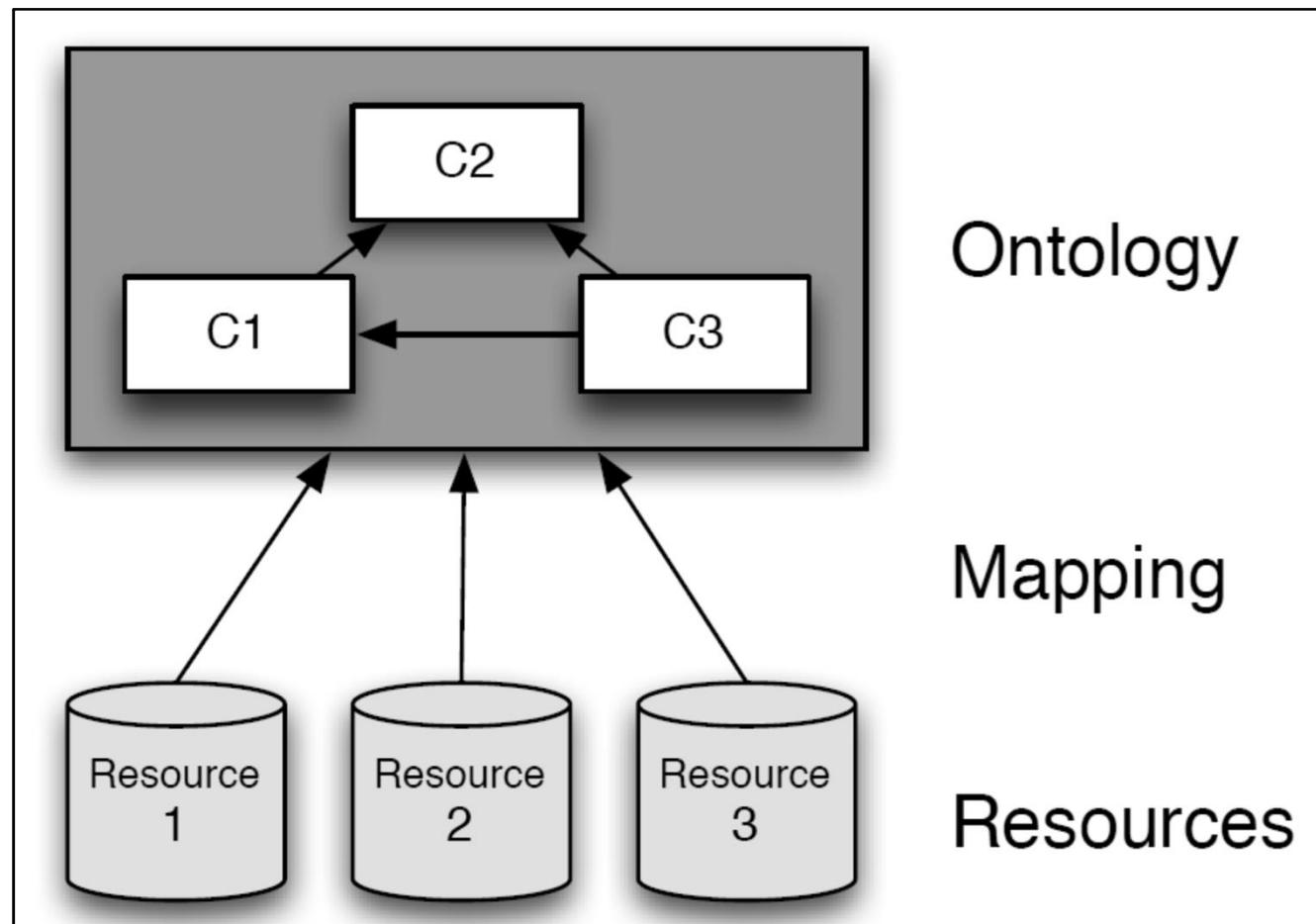
Hasta acá llegamos hoy...



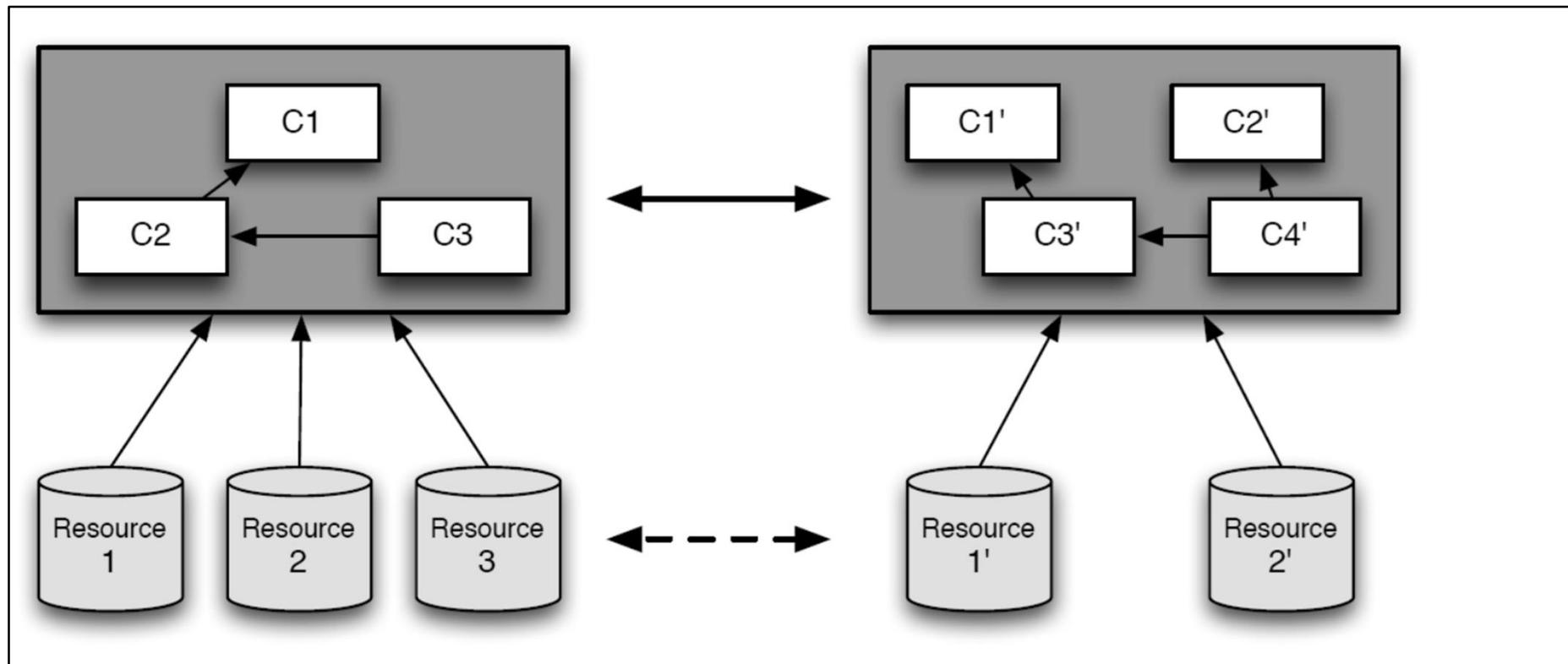
Ontology-Based Data Access (OBDA)

- Meta: alcanzar *transparencia lógica* en el acceso a los datos:
 - Esconder dónde y cómo están *almacenados* los datos.
 - Presentar al usuario una *vista conceptual* de los datos.
 - Usar un formalismo *semánticamente rico* para la vista conceptual.
- Objetivo similar al de *integración de datos*, pero con una descripción conceptual rica de la vista global.

Ontology-Based Data Access (OBDA)



Ontology-Based Data Access (OBDA)



Ontology-Based Data Access (OBDA)

- Meta: alcanzar transparencia lógica en el acceso a los datos.
 - Esconder dónde y cómo están almacenados los datos.
 - Presentar al usuario una vista conceptual de los datos.
 - Usar un formalismo semánticamente rico para la vista conceptual.
- Formalización de:
 - Lenguajes
 - Metodologías
 - Herramientas

Para especificar, construir, y administrar ontologías que se usan en sistemas de información.

Ontology-Based Data Access (OBDA)

- Una Lógica de Descripción se caracteriza por:
 - Un lenguaje de descripción: cómo formar conceptos y roles.

$Human \sqcap Male \sqcap \exists hasChild \sqcap \forall hasChild.(Doctor \sqcup Lawyer)$

- Un mecanismo para especificar conocimiento acerca de los conceptos y roles (una *TBox* o *axiomas terminológicos*).

$T = \{Father \equiv Human \sqcap Male \sqcap \exists hasChild,$

$HappyFather \sqsubseteq Father \sqcap \forall hasChild.(Doctor \sqcup Lawyer)\}$

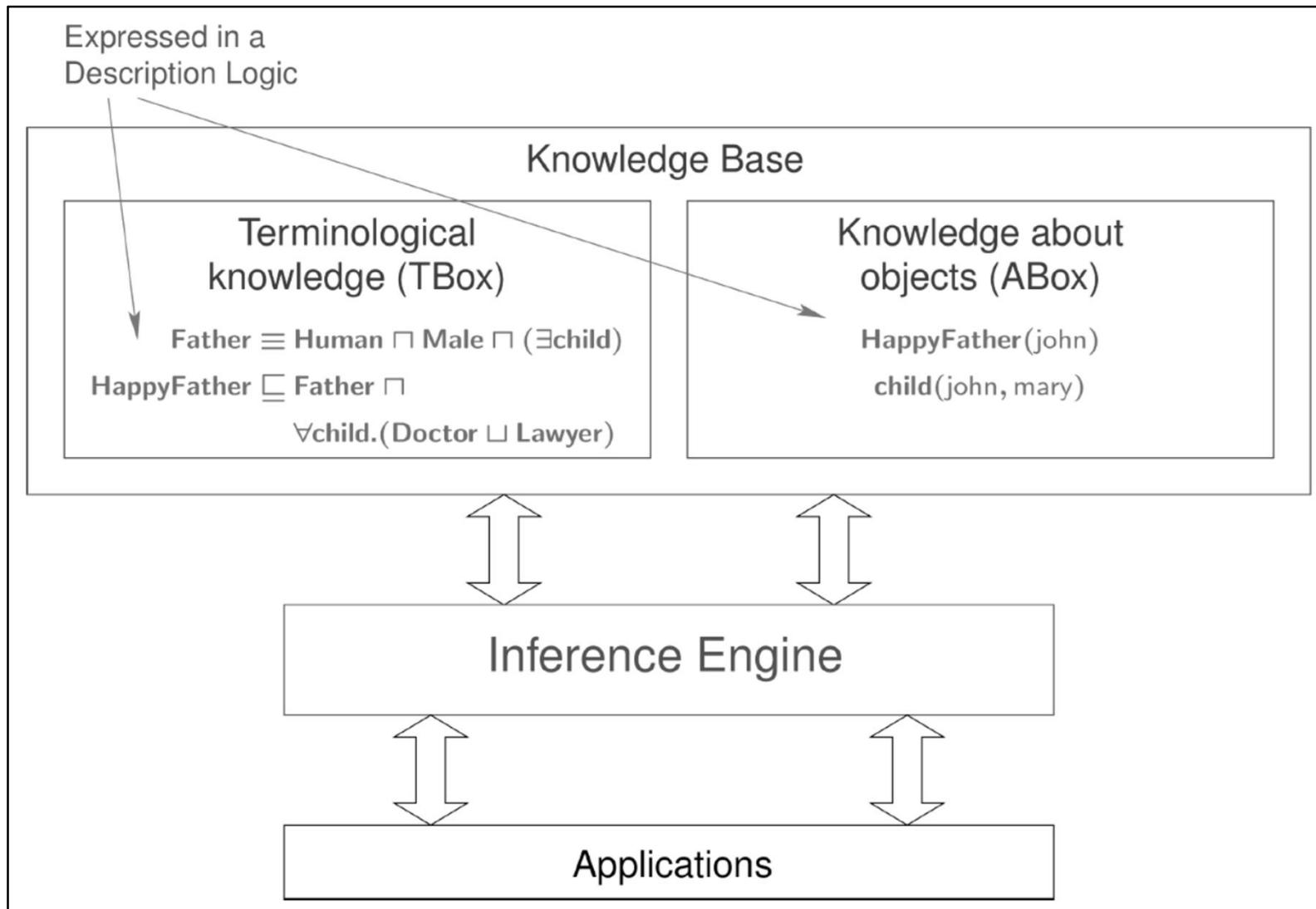
Ontology-Based Data Access (OBDA)

- Una Lógica de Descripción se caracteriza por:
 - Un mecanismo para especificar propiedades acerca de los objetos (*ABox* o *axiomas de aserciones*).
 - Un conjunto de servicios de inferencia: cómo razonar acerca del conocimiento contenido en una *KB*.

$$T \models HappyFather \sqsubseteq \exists hasChild.(Doctor \sqcup Lawyer)$$

$$T \cup A \models (Doctor \sqcup Lawyer)(mary)$$

Arquitectura de un sistema OBDA (con DLs)



Datalog

- Diseñado para bases de datos *deductivas*:
 - Bases de datos que permiten obtener información que está contenida *implícitamente*.
 - Dos partes: la parte *extensional* y la parte *intensional*; la extensional es un conjunto de *hechos* (proposiciones), la intensional un conjunto de *reglas* que permiten obtener nueva información a partir de la parte extensional.
- Datalog es un lenguaje de programación en lógica (sintácticamente es un subconjunto de *Prolog*).
- Reglas de la forma: $\forall X \forall Y \phi(X, Y) \rightarrow R(X)$

Datalog: Poder expresivo

- *No puede expresar algunos axiomas ontológicos importantes:*
 - Inclusión de conceptos que involucran *restricciones existenciales* en roles en la cabeza de las reglas:
$$\text{cientifico} \sqsubseteq \exists \text{esAutor} \text{de}$$
 - Conceptos *disjuntos*:
$$\text{artRevista} \sqsubseteq \neg \text{artConferencia}$$
 - *Funciones*: (*funct tienePrimerAutor*)
- Buena noticia: ¡Podemos extender Datalog para representar conocimiento ontológico rico!

Razonamiento ontológico y Datalog

DL Assertion	Datalog Rule
Concept Inclusion $emp \sqsubseteq person$	$emp(X) \rightarrow person(X)$
Concept Product $sen\text{-}emp \times emp \sqsubseteq moreThan$	$sen\text{-}emp(X), emp(Y) \rightarrow moreThan(X, Y)$
(Inverse) Role Inclusion $reports^- \sqsubseteq mgr$	$reports(X, Y) \rightarrow mgr(Y, X)$
Role Transitivity $trans(mgr)$	$mgr(X, Y), mgr(Y, Z) \rightarrow mgr(X, Z)$
Participation $emp \sqsubseteq \exists report$	$emp(X) \rightarrow \exists Y report(X, Y)$
Disjointness $emp \sqcap customer \sqsubseteq \perp$	$emp(X), customer(X) \rightarrow \perp$
Functionality $funct(reports)$	$reports(X, Y), reports(X, Z) \rightarrow Y = Z$

Desvío temporal: Homomorfismos



Sean A y B dos *estructuras relacionales finitas* con signatura σ (consistiendo de funciones y relaciones).

- Un *homomorfismo* de A en B es una función $h: \text{dom}(A) \rightarrow \text{dom}(B)$ que preserva la estructura:
 - para cada función n -aria f en σ y elementos $a_1, \dots, a_n \in \text{dom}(A)$, vale que $h(f^A(a_1, \dots, a_n)) = f^B(h(a_1), \dots, h(a_n))$; y
 - para cada relación n -aria R en σ y elementos $a_1, \dots, a_n \in \text{dom}(A)$, vale que si $(a_1, \dots, a_m) \in R^A$, entonces $(h(a_1), \dots, h(a_m)) \in R^B$
- Es una relajación del concepto de *isomorfismo* (todo isomorfismo es homomorfismo, pero no al revés).
- La *composición* de homomorfismos es un homomorfismo.

Desvío temporal: Homomorfismos



Sean I y J dos *instancias de BD* sobre un esquema S .

- Para BDs, un *homomorfismo* $h: adom(I) \rightarrow adom(J)$ es una función tal que para cada símbolo relacional $P \in S$ y cada tupla (a_1, \dots, a_m) tenemos que:
si $(a_1, \dots, a_m) \in P^I$, entonces $(h(a_1), \dots, h(a_m)) \in P^J$
- En BDs *no hay funciones*; la primera condición en la definición anterior no es necesaria (se cumple trivialmente).
- Dos instancias de BD I y J son *homomórficamente equivalentes* si existe un homomorfismo $I \rightarrow J$ y otro $J \rightarrow I$.

Desvío temporal: Homomorfismos



- Lema: Sea Q una BCQ y J una instancia de base de datos.
La siguientes afirmaciones son equivalentes:
 - $J \models Q$
 - Existe un homomorfismo $h: I^Q \rightarrow J$.
- Intuitivamente, h corresponde a la *asignación de variables* en Q que hace que ésta se satisfaga en J .

I^Q denota la “*instancia de BD canónica*” de Q , la cual es simplemente un conjunto de hechos construidos a partir de los predicados y variables de Q .

Desvío temporal: Homomorfismos



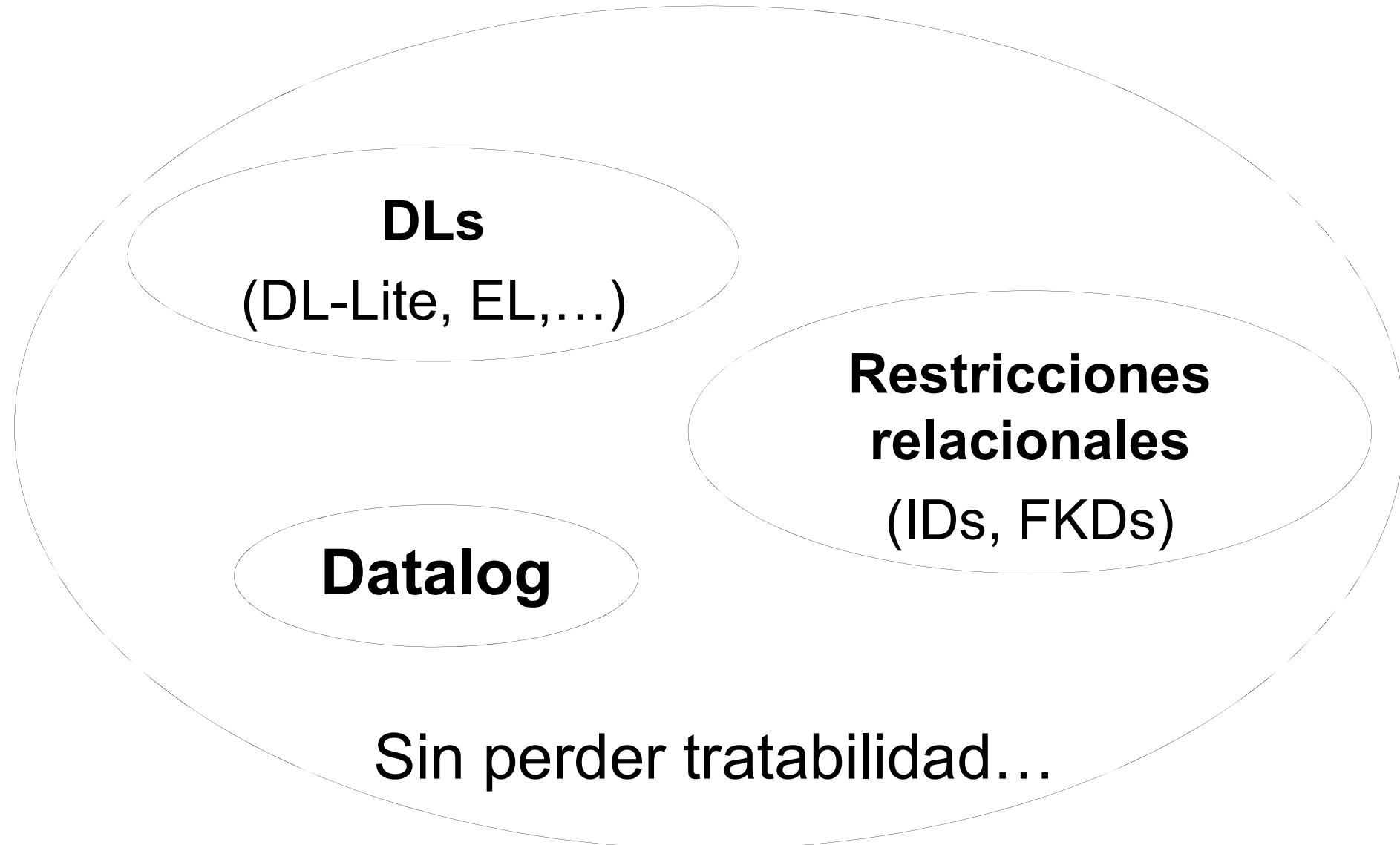
El “*Problema del homomorfismo*” pregunta simplemente si existe un homomorfismo entre dos estructuras finitas (en este caso, instancias de BD).

- Es *NP-completo*.
- Es un *problema fundamental* para la investigación algorítmica:
 - Todo problema de satisfacción de restricciones es un caso particular; por ejemplo, SAT.
 - Muchos problemas de IA son casos particulares; por ejemplo, planning.

Datalog+/-



Formalismos basados en Datalog



Extendiendo Datalog

- Extensión de Datalog permitiendo *existenciales* en la cabeza de las reglas: $\forall X \forall Y \Phi(X, Y) \rightarrow \exists Z \Psi(X, Z)$ (TGDs)
- Responder consultas (conjuntivas) en Datalog^Ξ (extensión con TGDs) es *indecidable* (se puede simular una MT).
- Datalog+/- extiende Datalog con *dependencias* y restricciones de *integridad*... pero con *limitaciones sintácticas* sobre las reglas.
- Datalog+/- es una *familia* de lenguajes ontológicos
 - las distintas restricciones dan lugar a *diferentes lenguajes* con distinto poder expresivo y complejidad computacional (para tareas como *query answering*).

Datalog⁺/–

- Asumimos:
 - Un universo infinito de constantes Δ
 - Un conjunto infinito de valores nulos (etiquetados) Δ_N
 - Un conjunto infinito de variables \mathcal{V}
 - Un esquema relacional \mathcal{R} , un conjunto finito de nombres de relaciones (o símbolos predicativos).
- Diferentes constantes representan diferentes valores; diferentes nulls pueden representar el mismo valor.
- Usamos X para denotar la secuencia X_1, \dots, X_n , $n \geq 0$.
- Una (instancia de) base de datos D sobre \mathcal{R} es un conjunto de átomos con predicados en \mathcal{R} y argumentos en Δ .

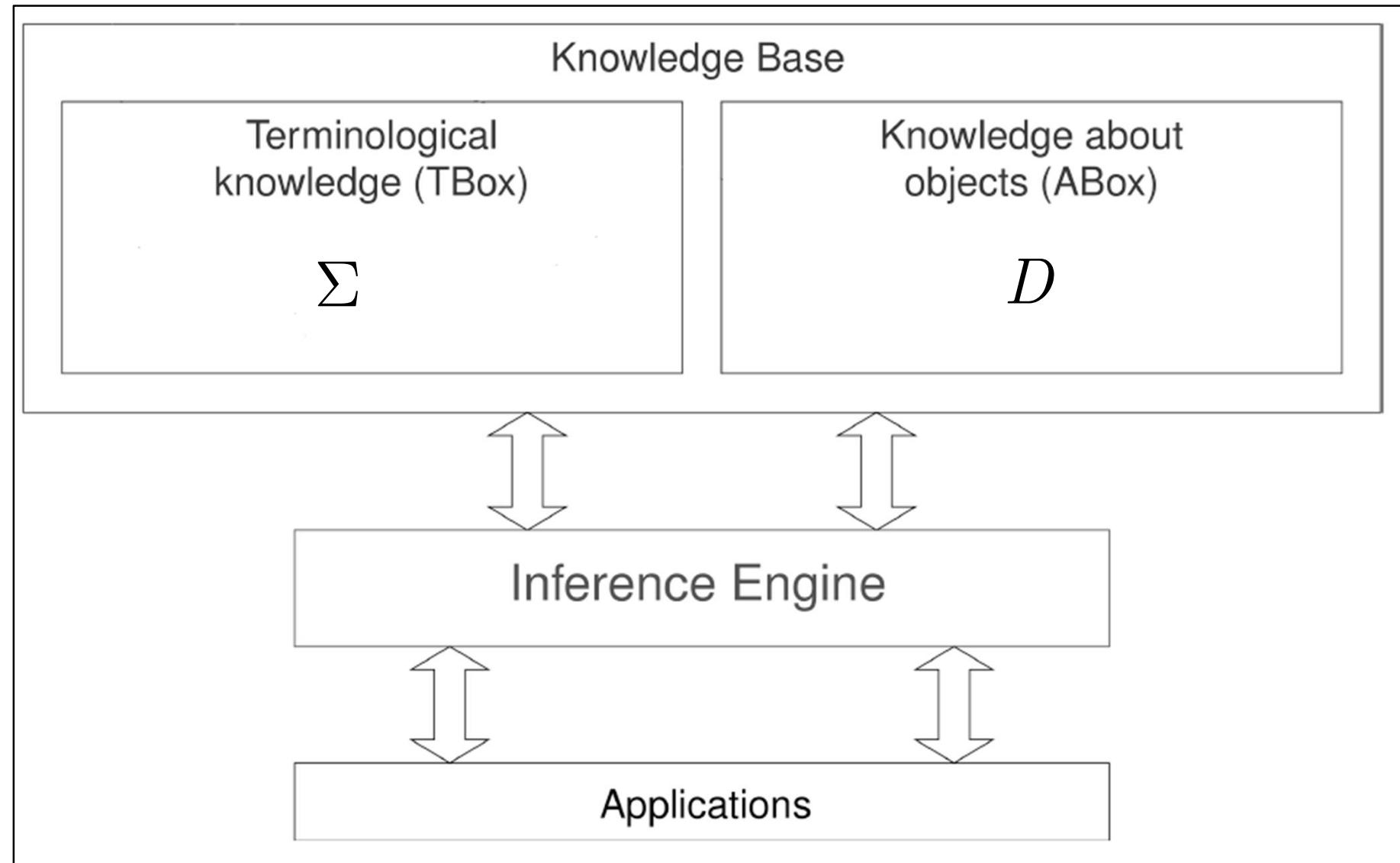
Datalog+/-

- Una consulta conjuntiva (CQ) sobre \mathcal{R} tiene la forma $Q(\mathbf{X}) = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, Φ es una conjunción de átomos.
- Una consulta conjuntiva Booleana (BCQ) sobre \mathcal{R} tiene la forma $Q() = \exists \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$, Φ es una conjunción de átomos.
- Las *respuestas* a una consulta se definen vía homomorfismos, mapeos $\mu: \Delta \cup \Delta_N \cup \mathcal{V} \rightarrow \Delta \cup \Delta_N \cup \mathcal{V}$:
 - si $c \in \Delta$ entonces $\mu(c) = c$
 - si $c \in \Delta_N$ entonces $\mu(c) \in \Delta \cup \Delta_N$
 - μ se extiende a (conjuntos de) átomos y conjunciones.
- Conjunto de *respuestas* $Q(D)$: conjunto de tuplas t sobre Δ t.q. $\exists \mu: \mathbf{X} \cup \mathbf{Y} \rightarrow \Delta \cup \Delta_N$ t.q. $\mu(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$, y $\mu(\mathbf{X}) = t$.

Datalog+/-

- Tuple-generating Dependencies (TGDs) son restricciones de la forma $\sigma: \forall \mathbf{X} \forall \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} \Psi(\mathbf{X}, \mathbf{Z})$ donde Φ y Ψ son conjunciones atómicas sobre \mathcal{R} :
 - $\forall \mathbf{X} \forall \mathbf{Y} \Phi(\mathbf{X}, \mathbf{Y})$ se denomina el cuerpo de σ ($body(\sigma)$)
 - $\exists \mathbf{Z} \Psi(\mathbf{X}, \mathbf{Z})$ se denomina la cabeza de σ ($head(\sigma)$)
- Dada una BD D y un conjunto Σ de TGDs, el conjunto de modelos $mods(D, \Sigma)$ es el conjunto de todos los B tal que:
 - $D \subseteq B$
 - cada $\sigma \in \Sigma$ es satisfecho en B (clásicamente).
- El conjunto de respuestas para una CQ Q en D y Σ , $ans(Q, D, \Sigma)$, es el conjunto de todas las tuplas a tal que $a \in Q(B)$ para todo $B \in mods(D, \Sigma)$.

Arquitectura de un sistema OBDA



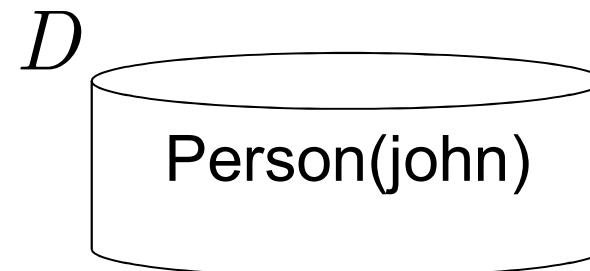
Chase

- El Chase es un procedimiento para reparar una BD en relación a un conjunto de dependencias (TGDs).
- (*Informalmente*) Regla de aplicación de TGD:
 - una TGD σ es aplicable a una BD D si $body(\sigma)$ mapea a átomos en D
 - la aplicación de σ sobre D agrega (si ya no existe) un átomo con nulos “frescos” correspondientes a cada una de las variables existenciales cuantificadas en $head(\sigma)$.

Chase

Input: Base de datos D , conjunto de TGDs Σ

Output: Un modelo de $D \cup \Sigma$



Σ

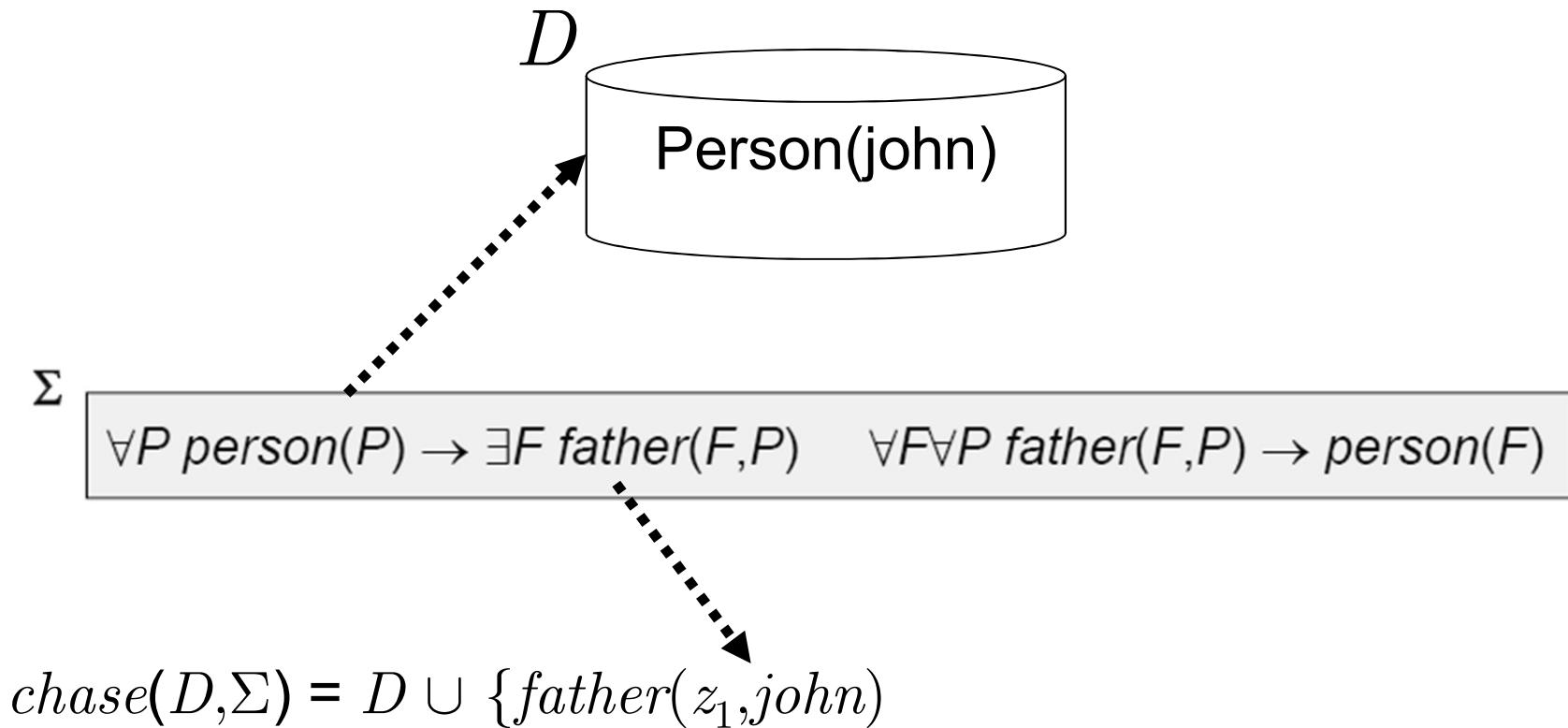
$$\forall P \text{ person}(P) \rightarrow \exists F \text{ father}(F,P) \quad \forall F \forall P \text{ father}(F,P) \rightarrow \text{person}(F)$$

$$\text{chase}(D, \Sigma) = D \cup ?$$

Chase

Input: Base de datos D , conjunto de TGDs Σ

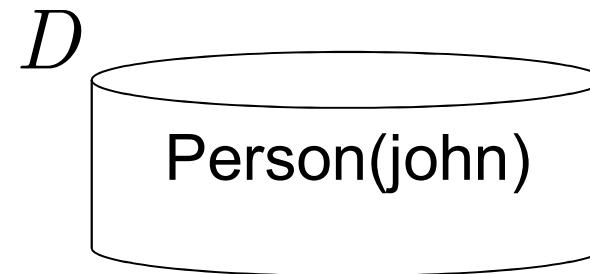
Output: Un modelo de $D \cup \Sigma$



Chase

Input: Base de datos D , conjunto de TGDs Σ

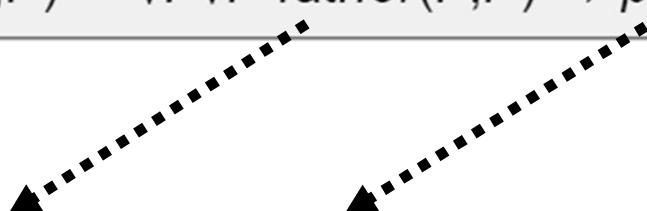
Output: Un modelo de $D \cup \Sigma$



Σ

$$\forall P \text{ person}(P) \rightarrow \exists F \text{ father}(F,P) \quad \forall F \forall P \text{ father}(F,P) \rightarrow \text{person}(F)$$

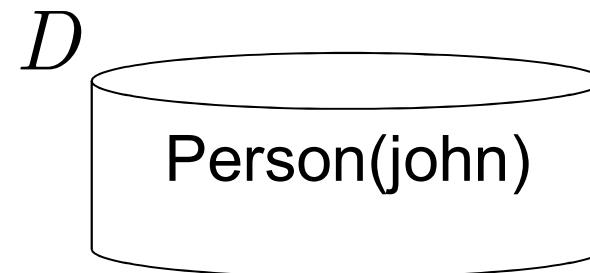
$$\text{chase}(D, \Sigma) = D \cup \{\text{father}(z_1, \text{john}), \text{person}(z_1)\}$$



Chase

Input: Base de datos D , conjunto de TGDs Σ

Output: Un modelo de $D \cup \Sigma$



Σ

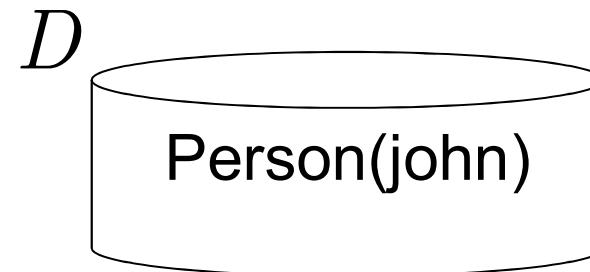
$$\forall P \text{person}(P) \rightarrow \exists F \text{father}(F,P) \quad \forall F \forall P \text{father}(F,P) \rightarrow \text{person}(F)$$

$$\text{chase}(D, \Sigma) = D \cup \{\text{father}(z_1, \text{john}), \text{person}(z_1), \text{father}(z_2, z_1)\}$$

Chase

Input: Base de datos D , conjunto de TGDs Σ

Output: Un modelo de $D \cup \Sigma$



Σ

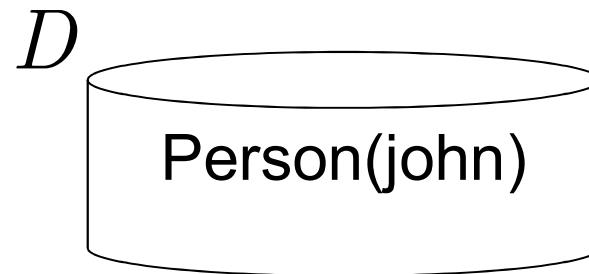
$$\forall P \text{ person}(P) \rightarrow \exists F \text{ father}(F,P) \quad \forall F \forall P \text{ father}(F,P) \rightarrow \text{person}(F)$$

$$\text{chase}(D, \Sigma) = D \cup \{\text{father}(z_1, \text{john}), \text{person}(z_1), \text{father}(z_2, z_1), \dots\}$$

Chase

Input: Base de datos D , conjunto de TGDs Σ

Output: Un modelo de $D \cup \Sigma$



Σ

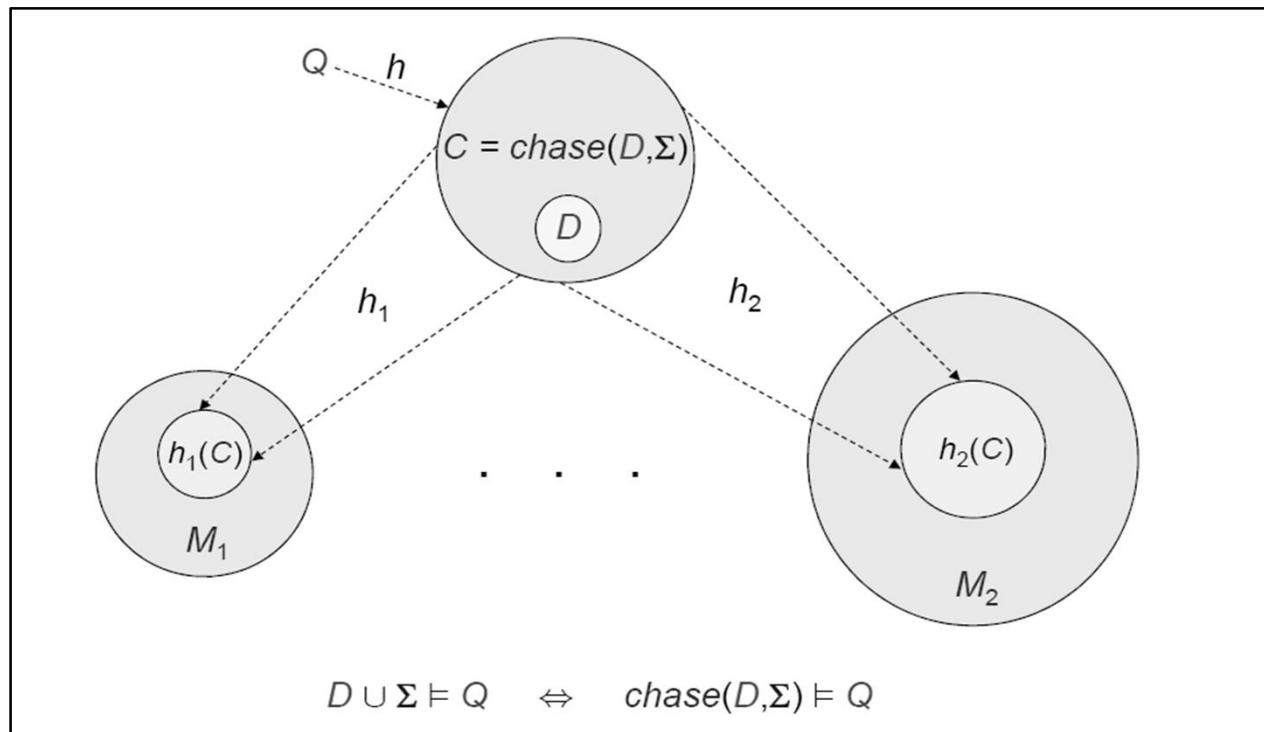
$$\forall P \text{ person}(P) \rightarrow \exists F \text{ father}(F,P) \quad \forall F \forall P \text{ father}(F,P) \rightarrow \text{person}(F)$$

$$chase(D, \Sigma) = D \cup \{\text{father}(z_1, john), \text{person}(z_1), \text{father}(z_2, z_1), \dots\}$$

INSTANCIA INFINITA

Query Answering vía el chase

- El chase (posiblemente infinito) es un *modelo universal*: existe un homomorfismo de $\text{chase}(D, \Sigma)$ en cada $B \in \text{mods}(D, \Sigma)$.
- Por lo tanto, tenemos que $D \cup \Sigma \models Q$ ssi $\text{chase}(D, \Sigma) \models Q$.



Negative Constraints y EGDs

- *Negative constraints* (NCs) son fórmulas de la forma
 $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow \perp$, donde $\Phi(\mathbf{X})$ es una conjunción of átomos.
- Las NCs son fáciles de verificar: podemos verificar que la CQ $\Phi(\mathbf{X})$ tiene un conjunto vacío de respuestas en D y Σ .
- *Equality Generating Dependencies* (EGDs) son de la forma
 $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow X_i = X_j$, donde Φ es una conjunción of átomos y X_i, X_j son variables que aparecen en \mathbf{X} .
- Se asume un conjunto de EGDs separables; intuitivamente significa que las EGDs y TGDs son independientes entre sí.

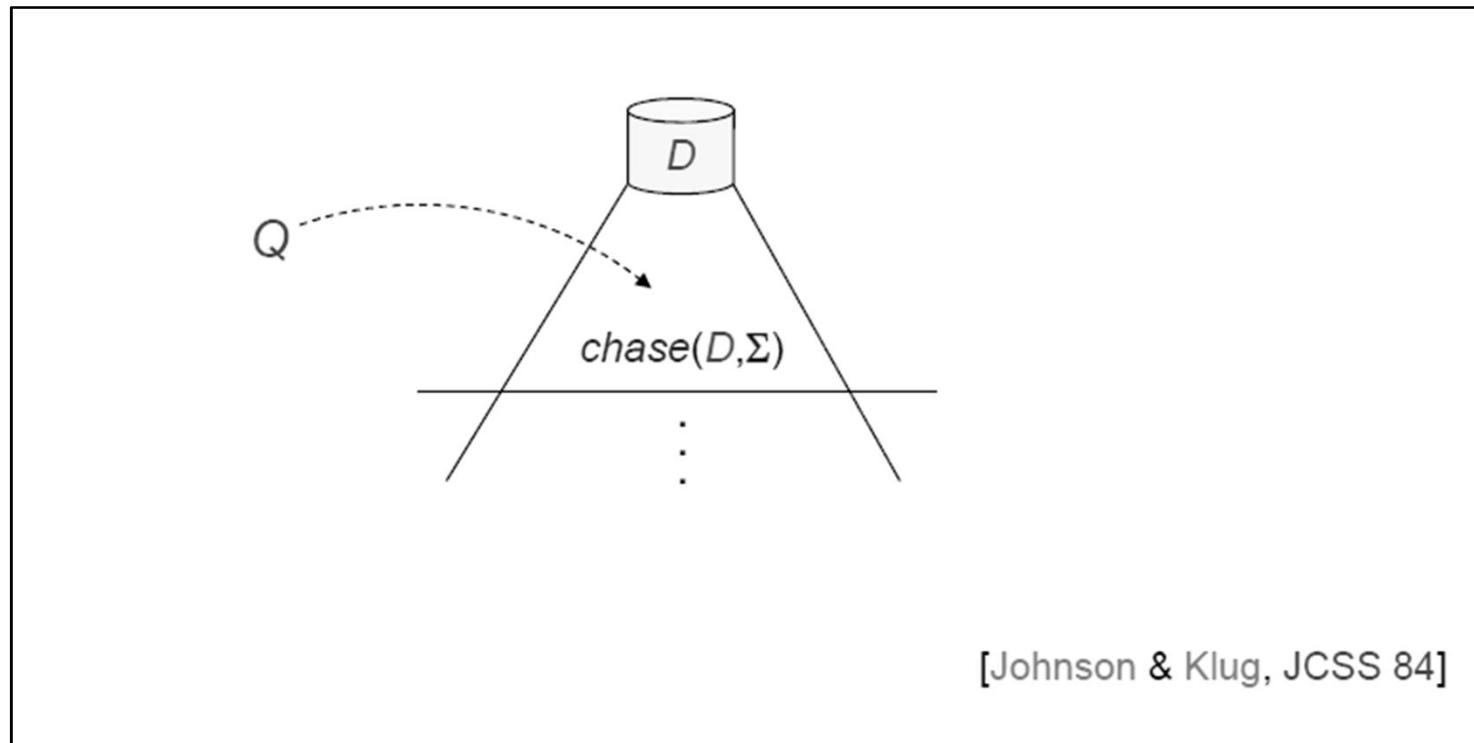
Datalog+/-: Ejemplo

$$D = \{ \text{directs}(john, sales), \text{directs}(anna, sales), \\ \text{directs}(john, finance), \text{supervises}(anna, john), \\ \text{works_in}(john, sales), \text{works_in}(anna, sales) \}$$
$$\Sigma_T = \{ \text{works_in}(X, D) \rightarrow \text{emp}(X), \\ \text{manager}(X) \rightarrow \exists Y \text{ supervises}(X, Y), \\ \text{supervises}(X, Y) \wedge \text{directs}(X, D) \rightarrow \text{works_in}(Y, D) \}$$
$$\Sigma_{NC} = \{ \text{supervises}(X, Y) \wedge \text{manager}(Y) \rightarrow \perp, \\ \text{supervises}(X, Y) \wedge \text{works_in}(X, D) \wedge \text{directs}(Y, D) \rightarrow \perp, \\ \text{directs}(X, D) \wedge \text{directs}(X, D') \rightarrow D = D' \}$$


Resultados positivos

Query Answering con IDs es decidible

- PSPACE-completo en complejidad *combinada*
- NP-completo complejidad *ba-combinada*



Guarded Datalog+/-

- Una TGD se dice guarded si existe un átomo en su cuerpo que contiene todas las variables que aparecen en el cuerpo.

$$\forall X \forall Y \forall Z \ R(X, Y, Z), S(Y), P(X, Z) \rightarrow \exists W \ Q(X, W)$$

guard 

- El *chase* tiene *treewidth finito* \Rightarrow query answering decidible
- Query answering es PTIME-completo en complejidad *data*.
- Extiende la Lógica de descripción ELH (misma complejidad *data*).

Guarded Datalog+/-

- ELH lógica de descripción muy popular para representar datasets biológicos con complejidad data PTIME.

EL TBox	Datalog $^{\pm}$ Representation
$A \sqsubseteq B$	$\forall X A(X) \rightarrow B(X)$
$A \sqcap B \sqsubseteq C$	$\forall X A(X), B(X) \rightarrow C(X)$
$\exists R. A \sqsubseteq B$	$\forall X R(X, Y), A(Y) \rightarrow B(X)$
$A \sqsubseteq \exists R. B$	$\forall X A(X) \rightarrow \exists Y R(X, Y), B(Y)$
$R \sqsubseteq P$	$\forall X \forall Y R(X, Y) \rightarrow P(X, Y)$

Linear Datalog $+/-$

- Una TGD se dice *linear* (lineal) si tiene sólo un átomo en su cuerpo.

$$\forall X \forall Y \ R(X, Y) \rightarrow \exists Z \ Q(X, Z)$$

guard 

- Las linear TGDs son (trivialmente) *guarded*.
- Query answering está en AC_0 en complejidad *data* (*reescritura de primer orden – FO rewritability*).
- Extiende la (familia de) lógicas de descripción *DL-Lite* (misma complejidad data).

Linear Datalog $^{+/-}$

- DL-Lite familia de lógicas de descripción con data complejidad AC_0 (OWL 2 QL).

DL-Lite TBox	Datalog $^{\pm}$ Representation
$A \sqsubseteq B$	$\forall X A(X) \rightarrow B(X)$
$A \sqsubseteq \exists R$	$\forall X A(X) \rightarrow \exists Y R(X, Y)$
$\exists R \sqsubseteq A$	$\forall X \forall Y R(X, Y) \rightarrow A(X)$
$R \sqsubseteq P$	$\forall X \forall Y R(X, Y) \rightarrow P(X, Y)$

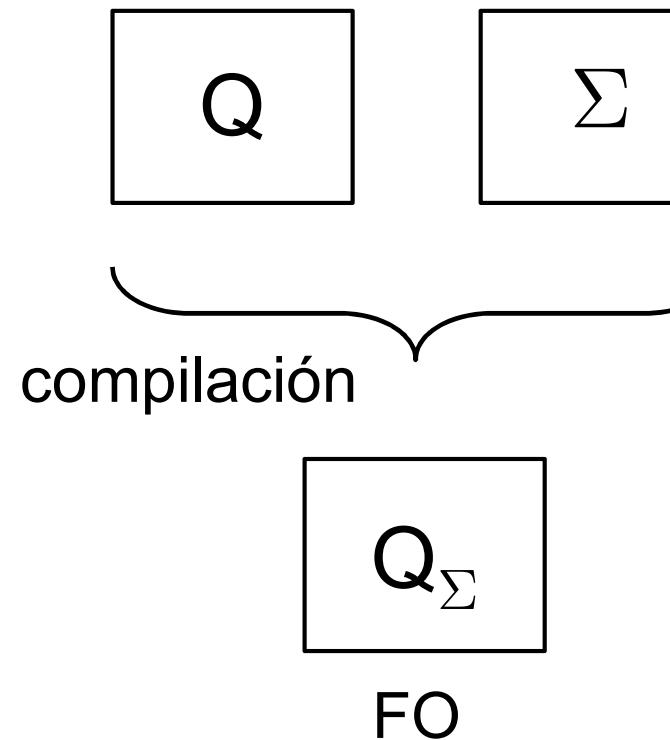
TGDs FO Re-escribibles

Q

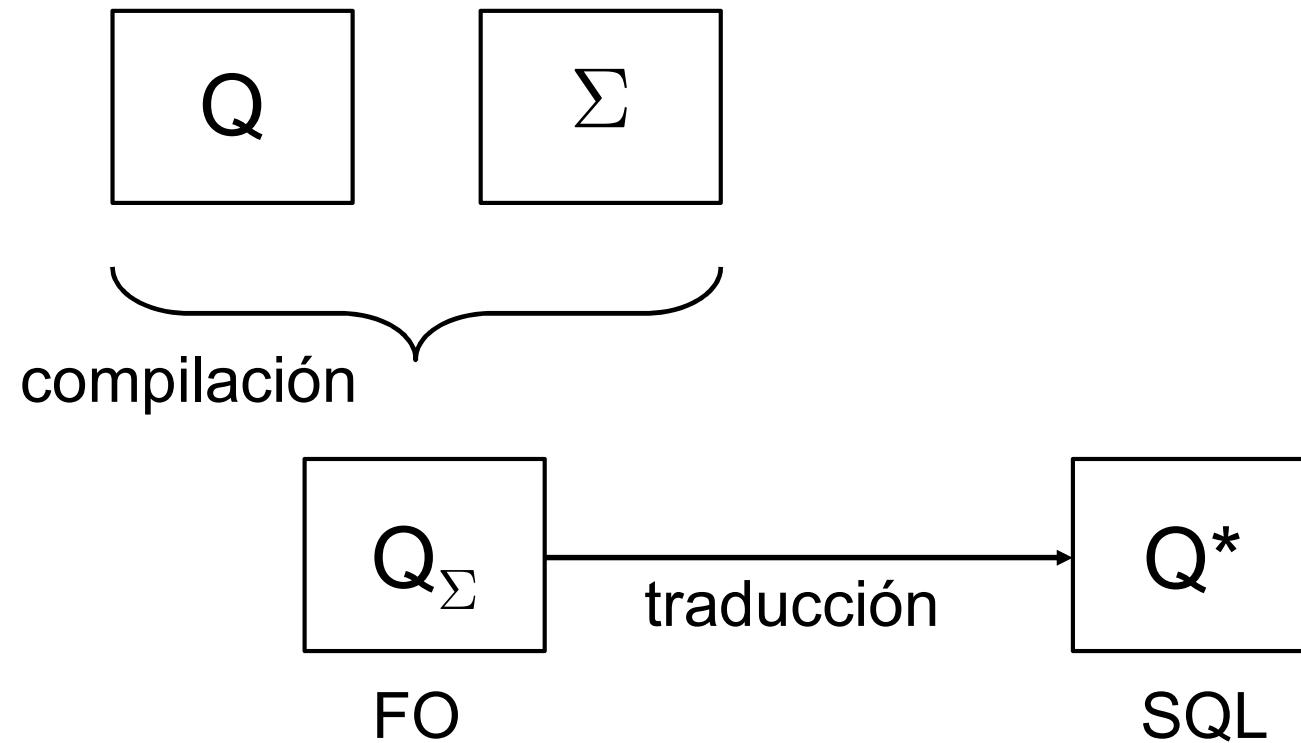
Σ



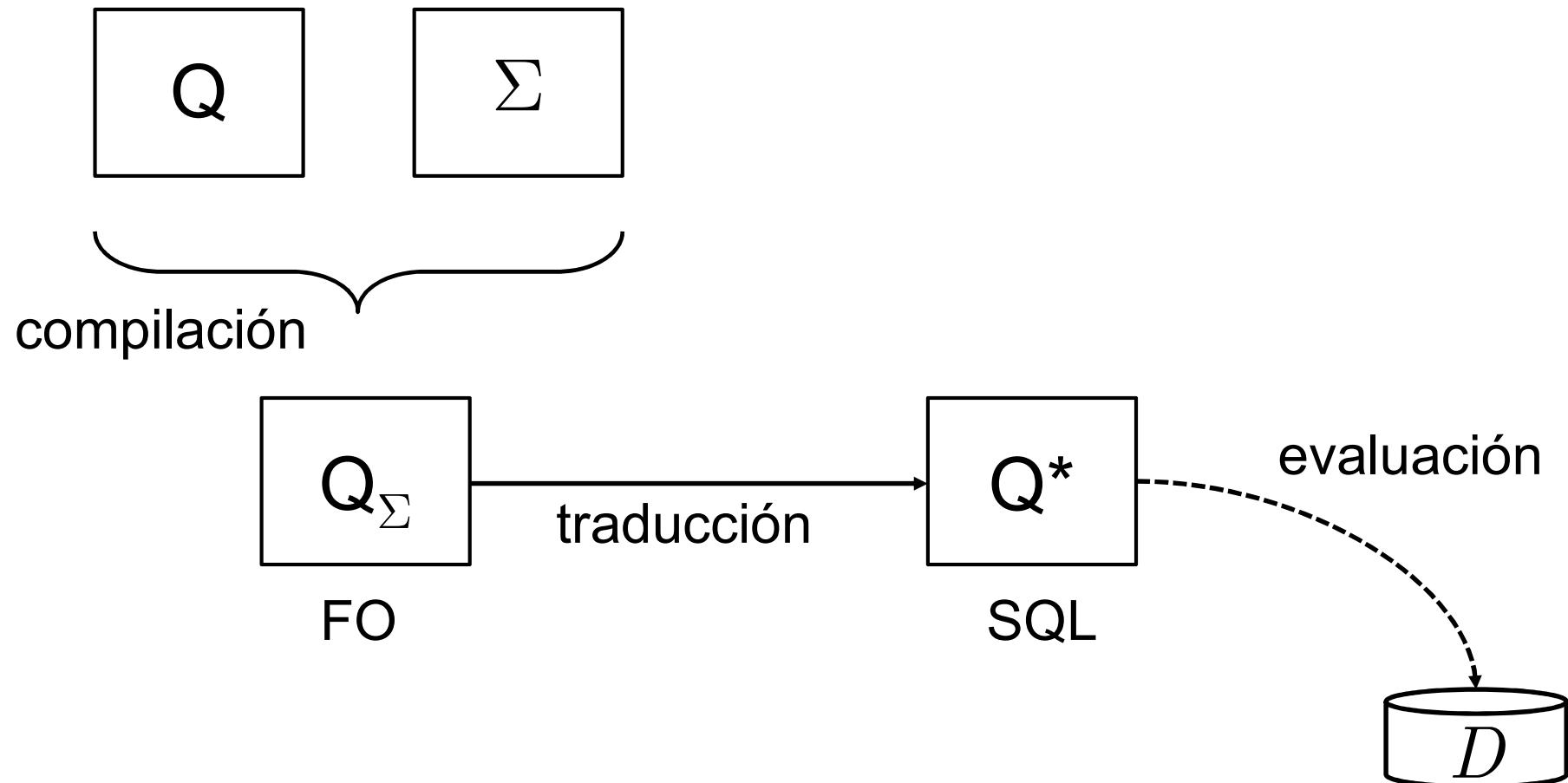
TGDs FO Re-escribibles



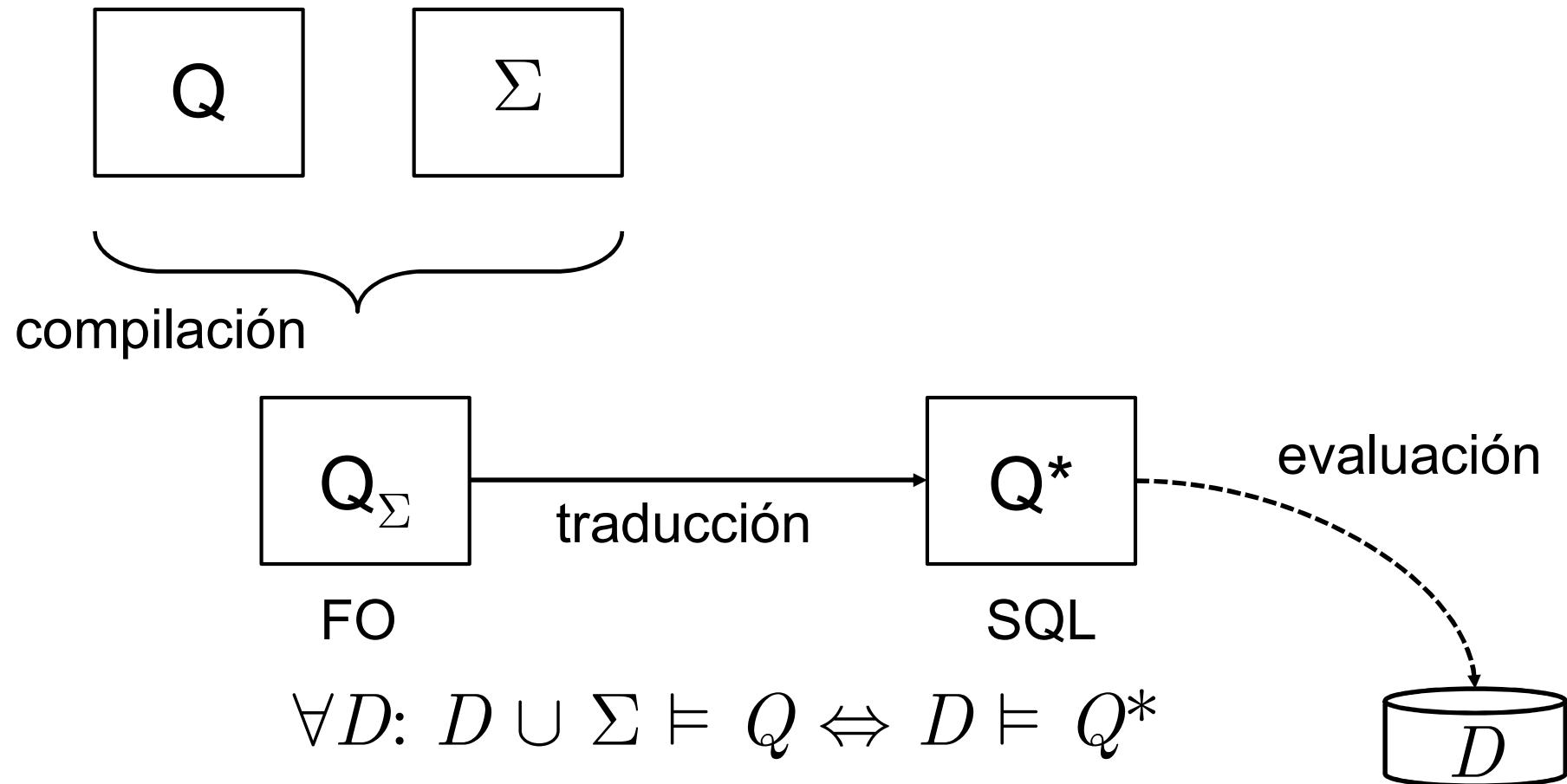
TGDs FO Re-escribibles



TGDs FO Re-escribibles

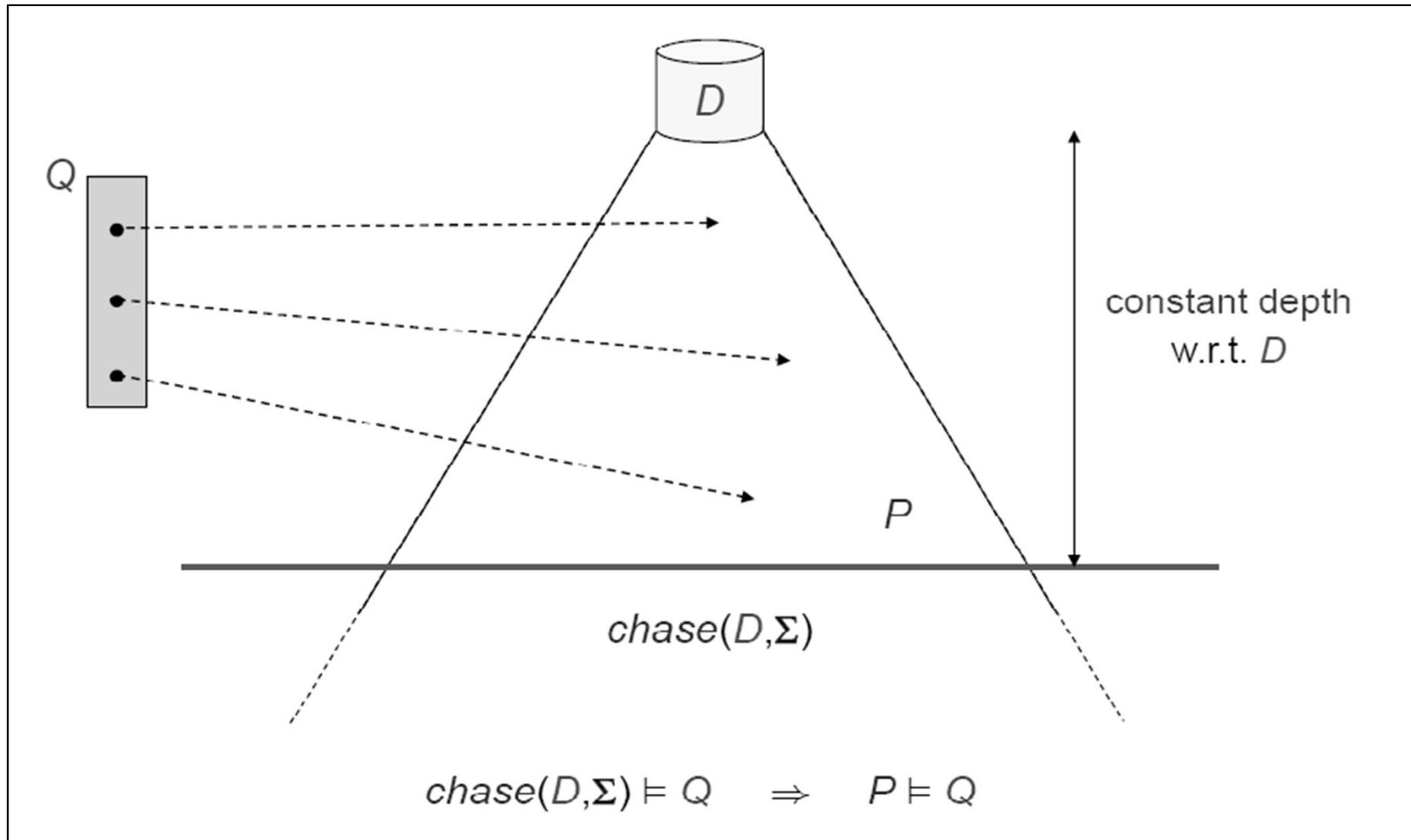


TGDs FO Re-escribibles



Query answering está en AC_0
(complejidad data)

Bounded Derivation-Depth Property (BDDP)



Bounded Derivation-Depth Property (BDDP)

- Query answering en programas Datalog \exists que satisfacen BDDP está en AC₀ (complejidad data).
- *Bounded derivation-depth* es una propiedad semántica.
- Queremos identificar un *fragmento sintáctico* de Datalog \exists que satisfaga la propiedad.
- Algunos *resultados*:
 - Guarded Datalog \exists no satisface BDDP (*¿cómo lo probaría?*)
 - Linear Datalog \exists satisface BDDP.
 - BDDP \Rightarrow FO rewritability

Pero...

- ¿Qué sucede con los *joins* en los cuerpos de las reglas?

$$\forall A \forall D \forall P \text{ } runs(D,P), \text{ } area(P,A) \rightarrow \exists E \text{ } employee(E,D,P,A)$$



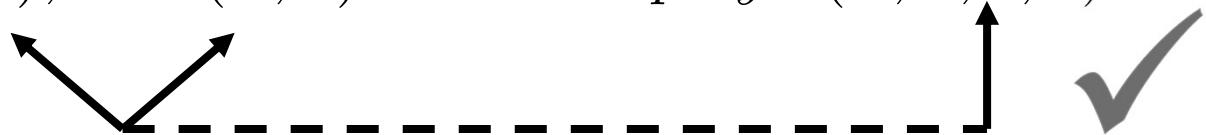
- ¿Y los axiomas de *productos* de conceptos (cartesiano)?

$$\forall E \forall M \text{ } elephant(E), \text{ } mouse(M) \rightarrow biggerThan(E,M)$$

- No se pueden garantizar modelos con *forma de árbol*

$$\begin{array}{l} \forall X \forall Y \text{ } R(X,Y) \rightarrow \exists Z \text{ } R(Y,Z) \\ \forall X \forall Y \text{ } R(X,Y) \rightarrow S(X) \\ \forall X \forall Y \text{ } S(X), \text{ } S(Y) \rightarrow P(X,Y) \end{array} \left. \begin{array}{l} \text{Infinita cantidad de} \\ \text{símbolos en S} \\ P \text{ forma un clique infinito} \end{array} \right\}$$

Stickiness

$$\forall A \forall D \forall P \ runs(D,P), \ area(P,A) \rightarrow \exists E \ employee(E,D,P,A)$$

$$\forall E \forall M \ elephant(E), mouse(M) \rightarrow biggerThan(E,M)$$


Stickiness

$$\forall X \forall Y \forall Z \ R(X, Y), \ P(Y, Z) \rightarrow \exists W \ T(X, Y, W)$$

$$\forall X \forall Y \forall Z \ T(X, Y, Z) \rightarrow \exists W \ S(Y, W)$$



Stickiness

$$\forall X \forall Y \forall Z \ R(X, Y), \ P(Y, Z) \rightarrow \exists W \ T(X, Y, W)$$
$$\forall X \forall Y \forall Z \ T(X, Y, Z) \rightarrow \exists W \ S(X, W)$$

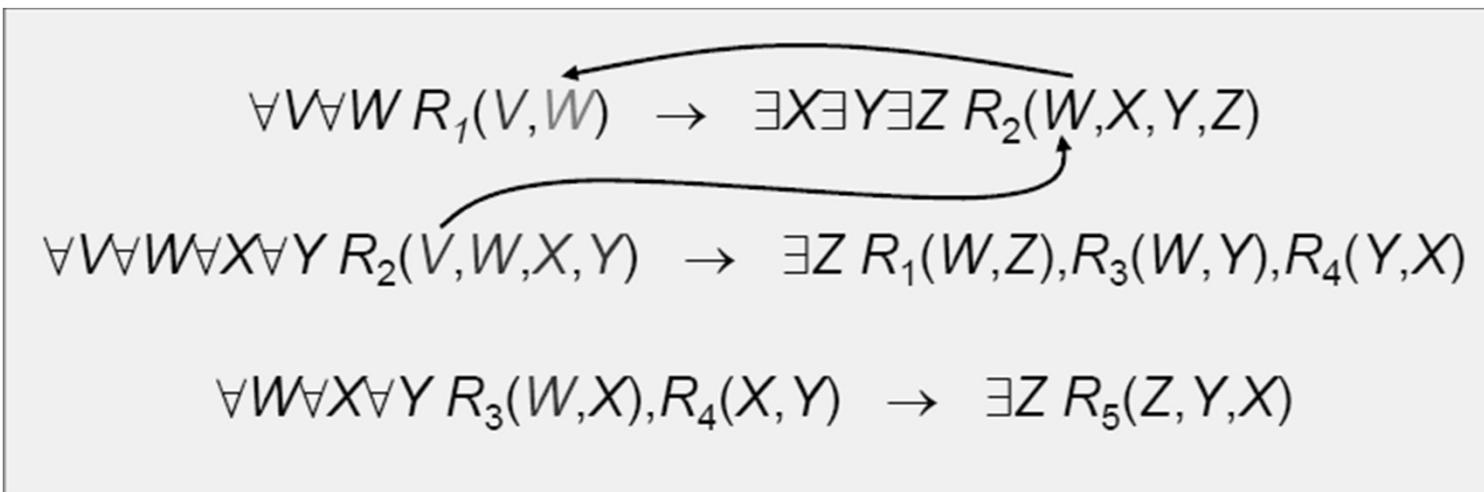

Stickiness: procedimiento de marcado

- Marcado Inicial: marcar *todas* las ocurrencias de las *variables del cuerpo* de la regla que no aparecen en *todos* los átomos de la cabeza.

$$\forall W \forall W R_1(V, W) \rightarrow \exists X \exists Y \exists Z R_2(W, X, Y, Z)$$
$$\forall W \forall W \forall X \forall Y R_2(V, W, X, Y) \rightarrow \exists Z R_1(W, Z), R_3(W, Y), R_4(Y, X)$$
$$\forall W \forall X \forall Y R_3(W, X), R_4(X, Y) \rightarrow \exists Z R_5(Z, Y, X)$$

Stickiness: procedimiento de marcado

- Marcado Inicial: marcar *todas* las ocurrencias de las *variables del cuerpo* de la regla que no aparecen en *todos* los átomos de la cabeza.
- *Propagación*: propagar el marcado de las variables del cuerpo vía los átomos de la cabeza.



Stickiness: procedimiento de marcado

- Las variables marcadas ocurren sólo una vez en el cuerpo de cada TGD.

$$\forall V \forall W \ R_1(V,W) \rightarrow \exists X \exists Y \exists Z \ R_2(W,X,Y,Z)$$

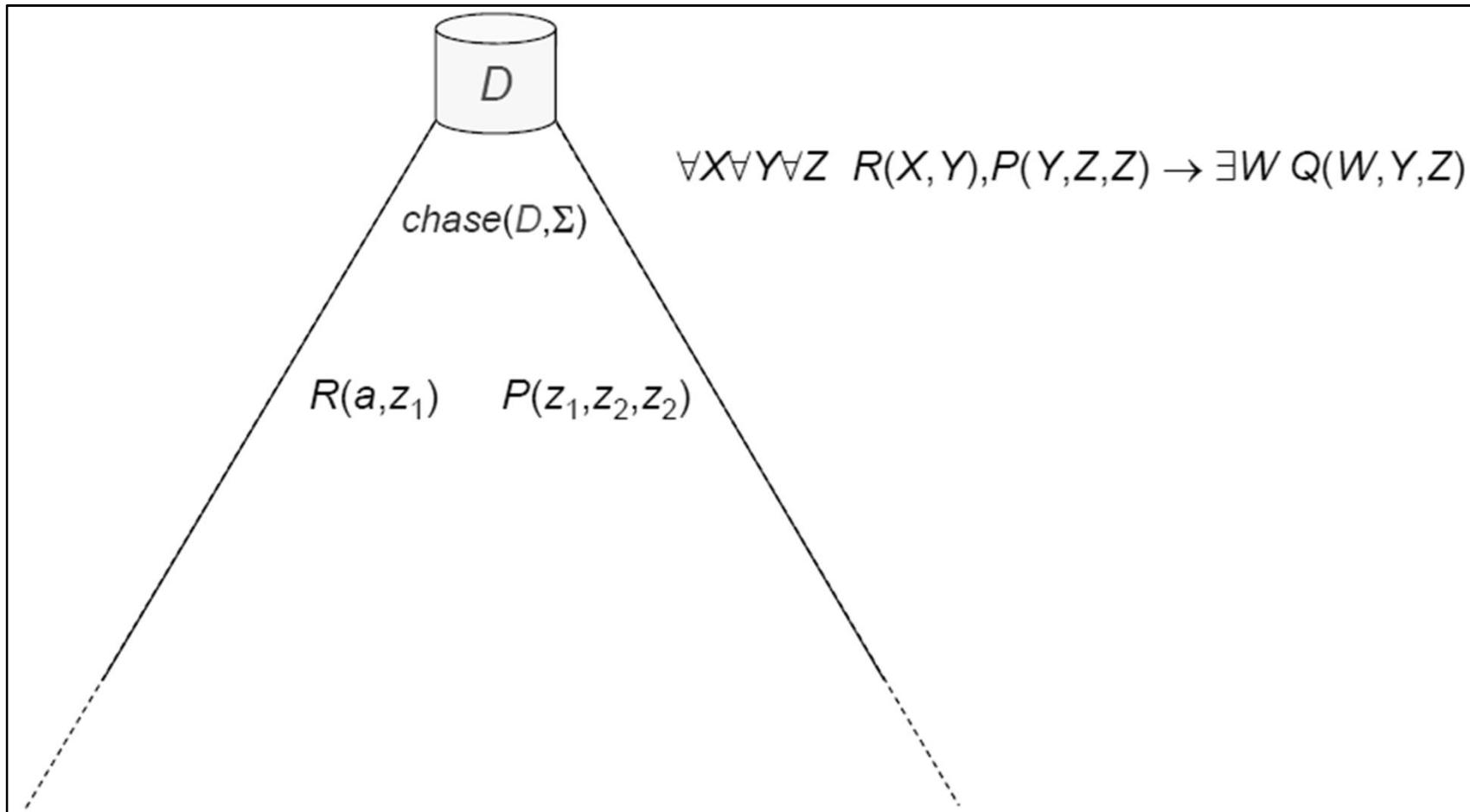
$$\forall V \forall W \forall X \forall Y \ R_2(V,W,X,Y) \rightarrow \exists Z \ R_1(W,Z), R_3(W,Y), R_4(Y,X)$$

$$\forall W \forall X \forall Y \ R_3(W,X), R_4(X,Y) \rightarrow \exists Z \ R_5(Z,Y,X)$$

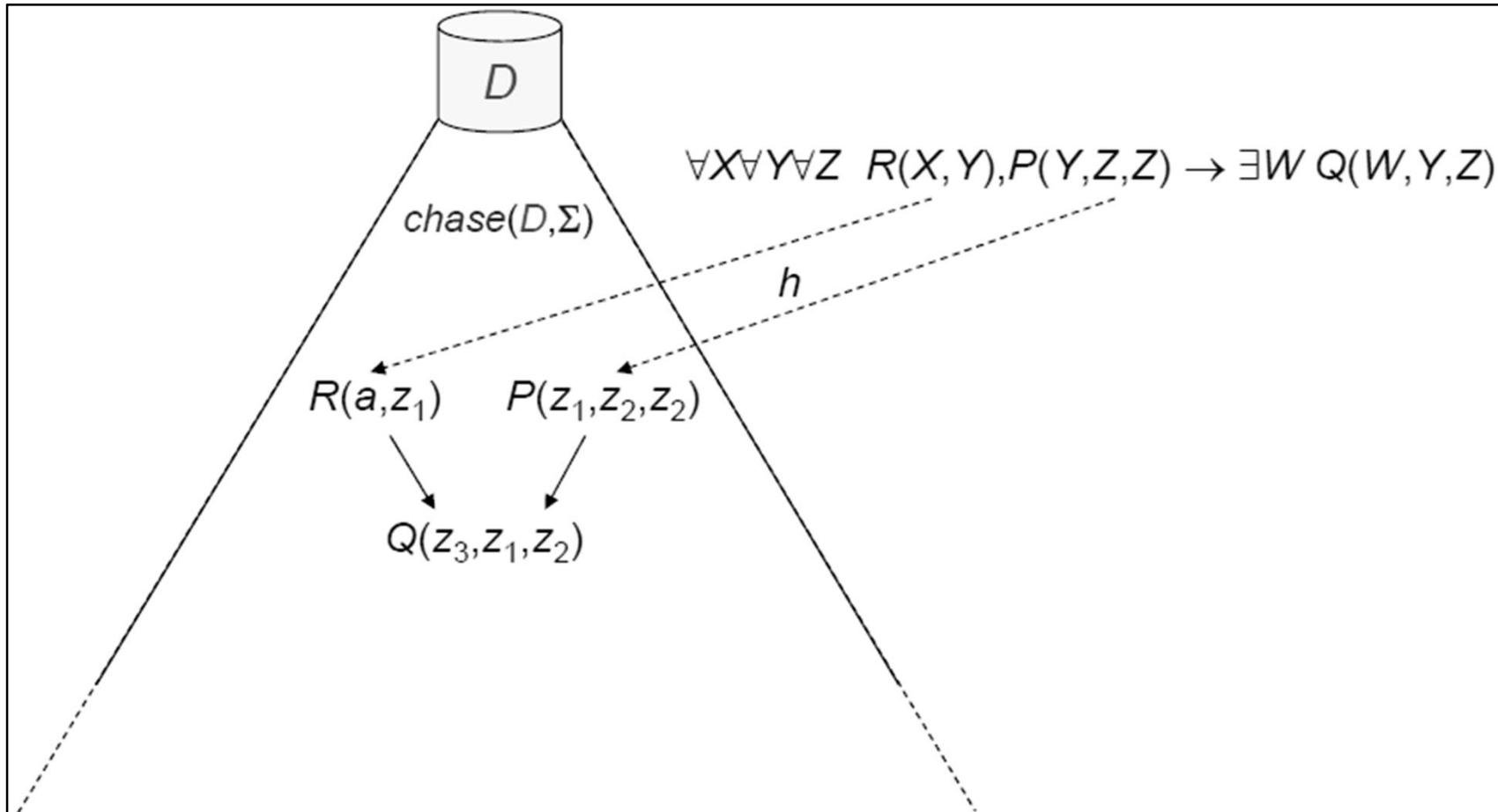
- El *chase* tiene la propiedad *sticky* (backward-resolution termina)



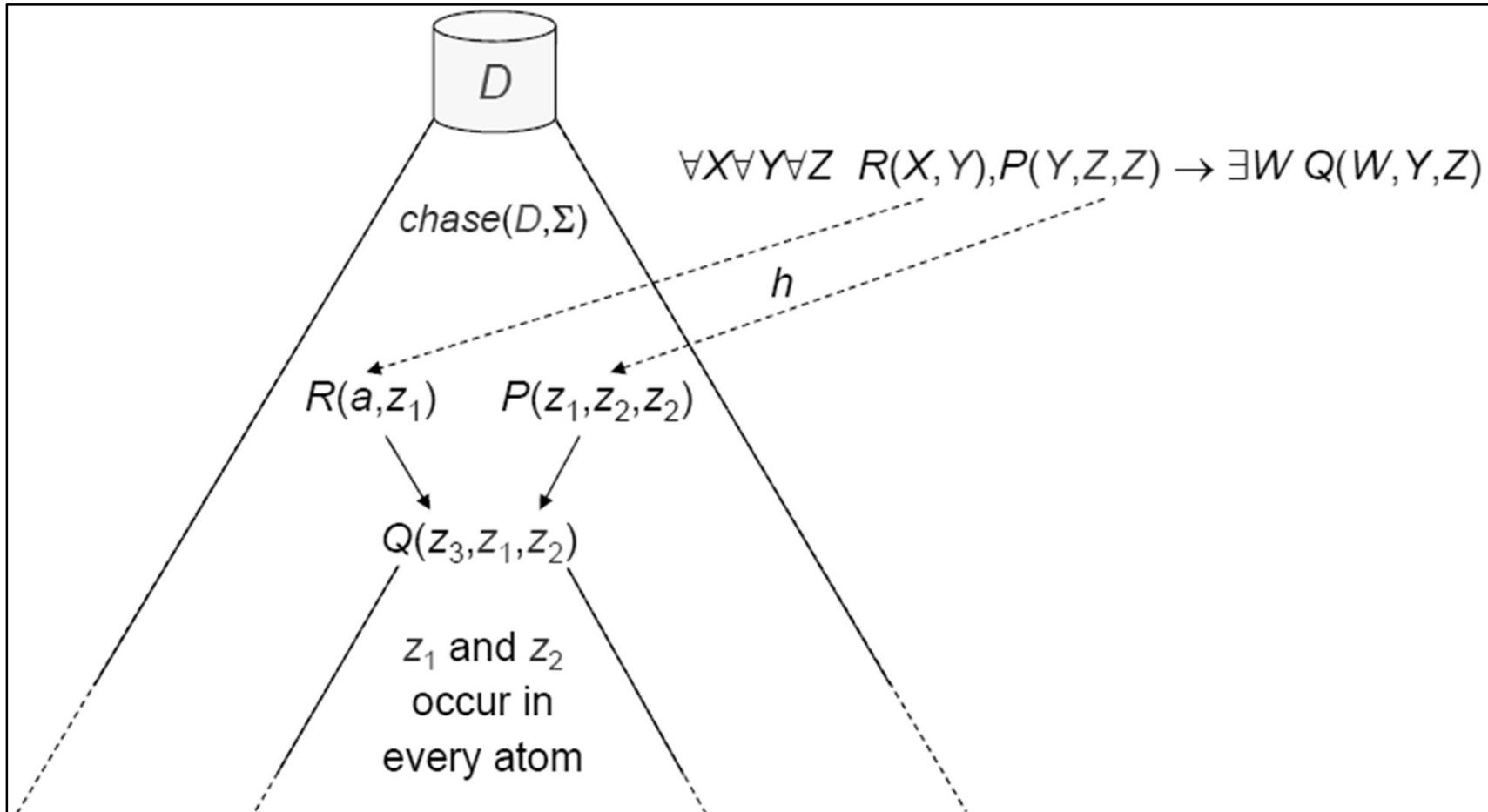
Stickiness



Stickiness



Stickiness



Stickiness: procedimiento de marcado

- Las variables marcadas ocurren sólo una vez en el cuerpo de cada TGD.

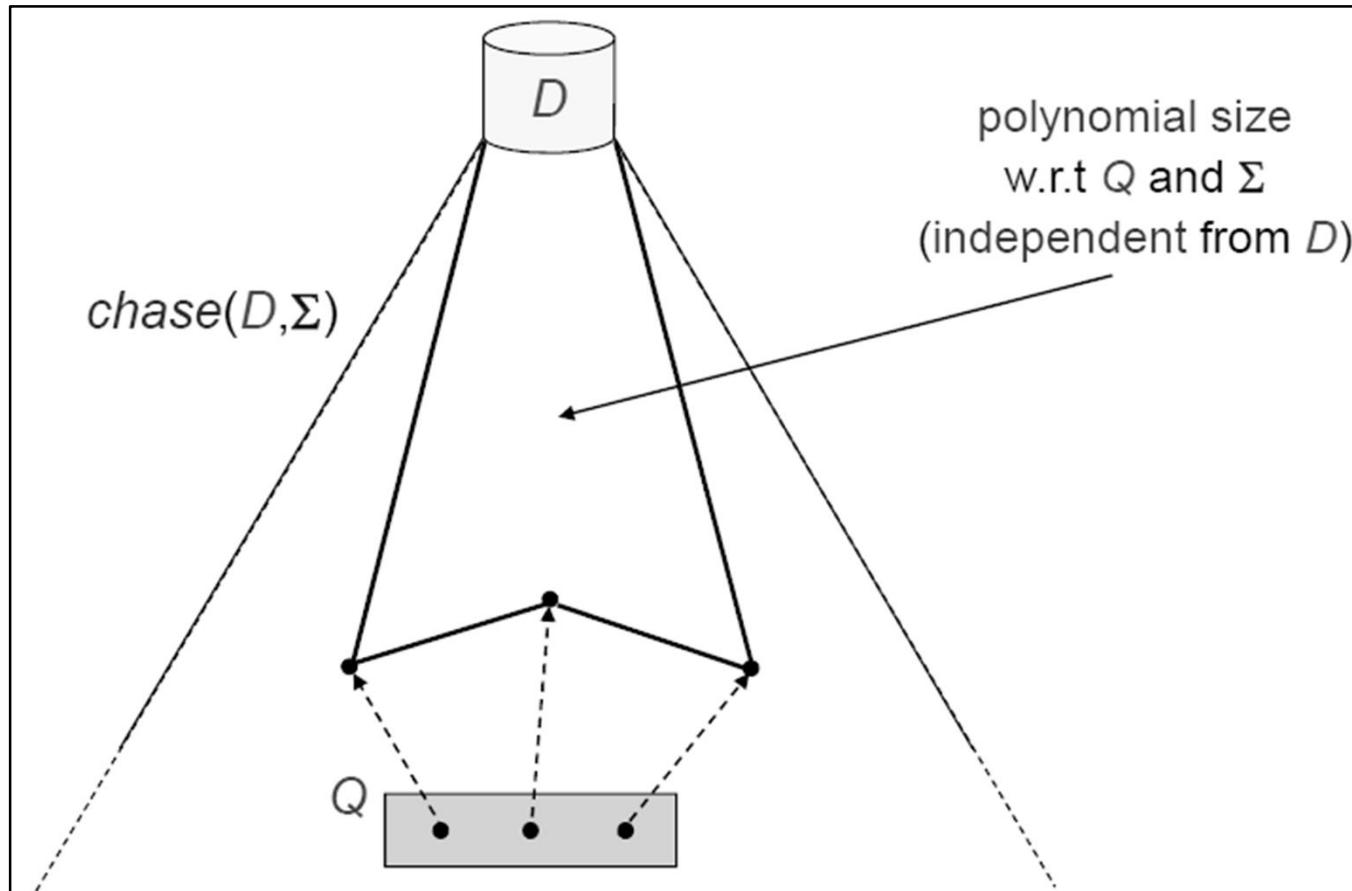
$$\forall V \forall W R_1(V, W) \rightarrow \exists X \exists Y \exists Z R_2(W, X, Y, Z)$$

$$\forall V \forall W \forall X \forall Y R_2(V, W, X, Y) \rightarrow \exists Z R_1(W, Z), R_3(W, Y), R_4(Y, X)$$

$$\forall W \forall X \forall Y R_3(W, X), R_4(X, Y) \rightarrow \exists Z R_5(Z, Y, X)$$

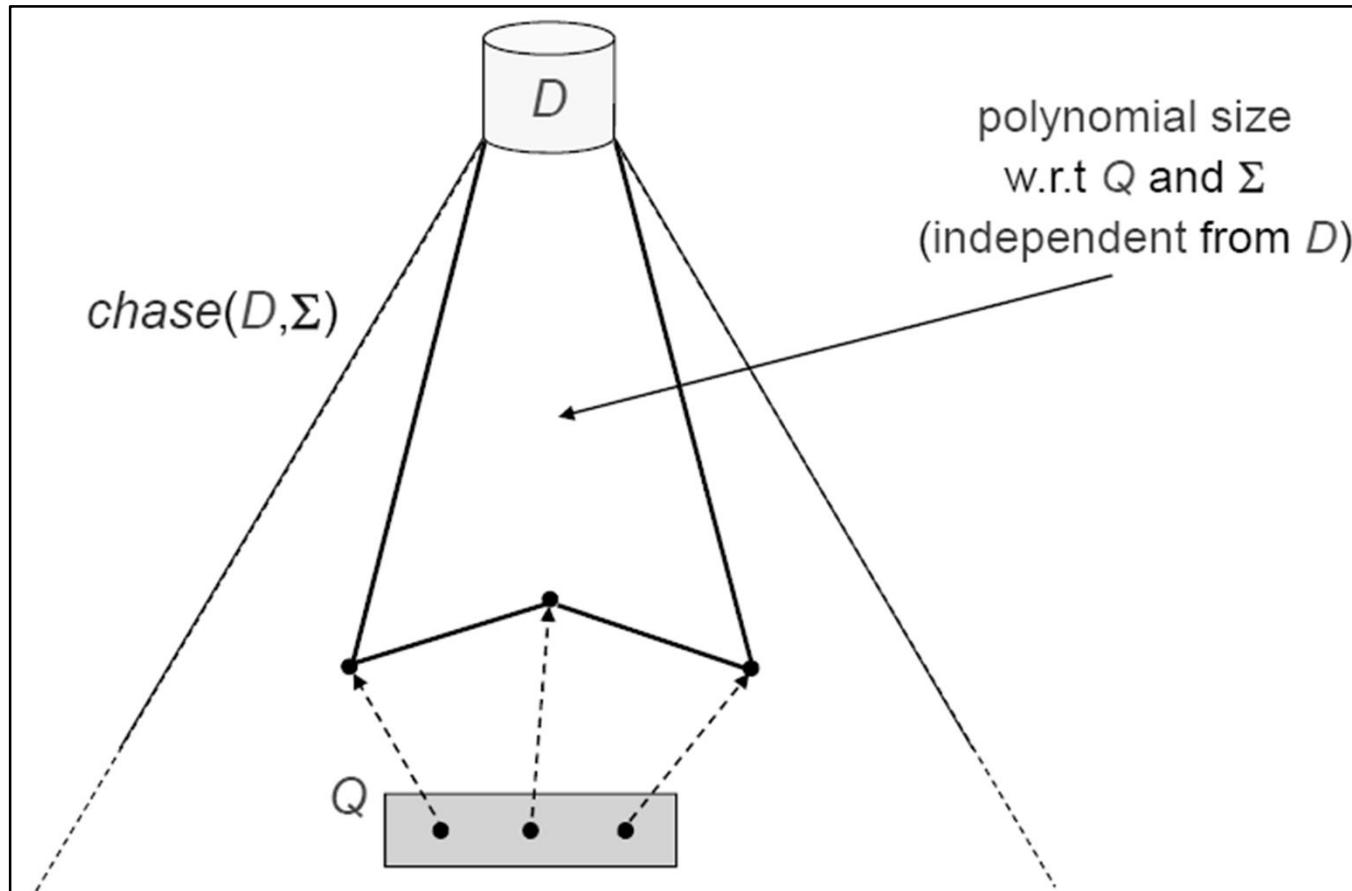
- Query answering en Sticky Datalog+/- está en AC₀ (data complejidad – FO rewritability)
- Extiende a la familia DL-Lite (misma complejidad data)

Polynomial Witness Property (PWP)



- PWP \Rightarrow re-escritura en *Datalog* (no recursivo) de tamaño polinomial.

Polynomial Witness Property (PWP)



- Linear y Sticky TGDs tienen la propiedad PWP.

Finite Controllability

$$D \cup \Sigma \models Q \Leftrightarrow D \cup \Sigma \models_{fin} Q$$

?

- La propiedad vale para:
 - *Dependencias de inclusión*
 - Guarded TGDs
 - Sticky TGDs



Otras propiedades

- **EGDs:** $\forall X \forall Y \forall Z \text{ } \textit{reports}(X, Y), \text{ } \textit{reports}(Y, Z) \rightarrow Y = Z$
 - *Non-Conflicting EGDs:* no *interactúan* con el conjunto de TGDs.
 - Chequeo de *satisfabilidad* no agrega complejidad (misma complejidad que *query answering* para el fragmento al que pertenece $D \cup \Sigma$).
- **Negative constraints:** $\forall X \text{ } \textit{emp}(X), \text{ } \textit{customer}(X) \rightarrow \perp$
 - Se puede *verificar* si $D \cup \Sigma$ satisface el conjunto de NCs sin agregar complejidad.

EGDs: Finite Controllability

- Finite controllability no vale en general en la presencia de EGDs *arbitrarias*.

$$D = \{R(a,b)\}$$

$$\Sigma = \left\{ \begin{array}{l} \forall X \forall Y R(X,Y) \rightarrow \exists Z R(Y,Z) \\ \forall X \forall Y \forall Z R(Y,X), R(Z,X) \rightarrow Y = Z \end{array} \right\}$$

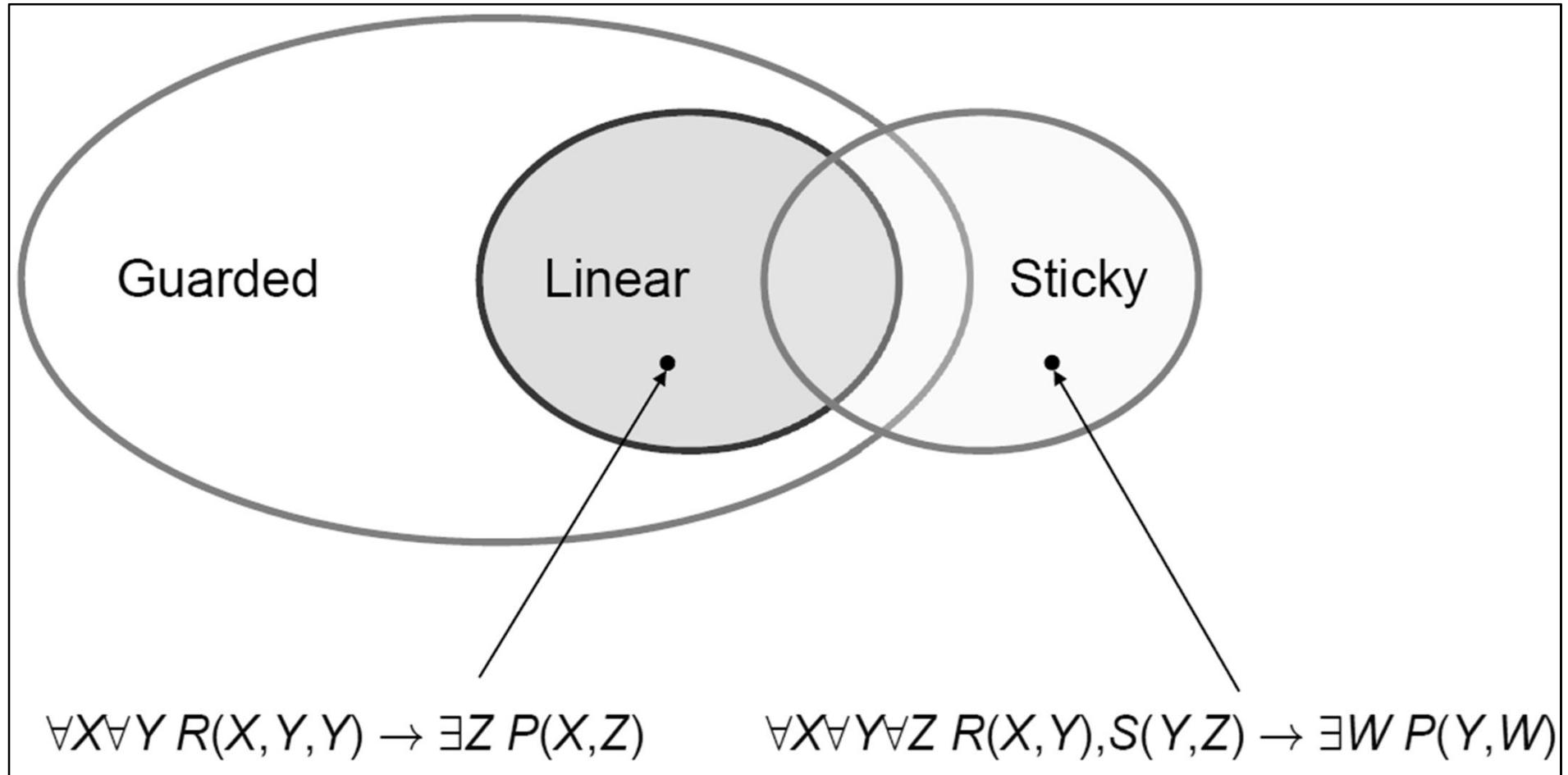
$$Q \leftarrow R(A,a)$$

$$D \cup \Sigma \not\models Q$$

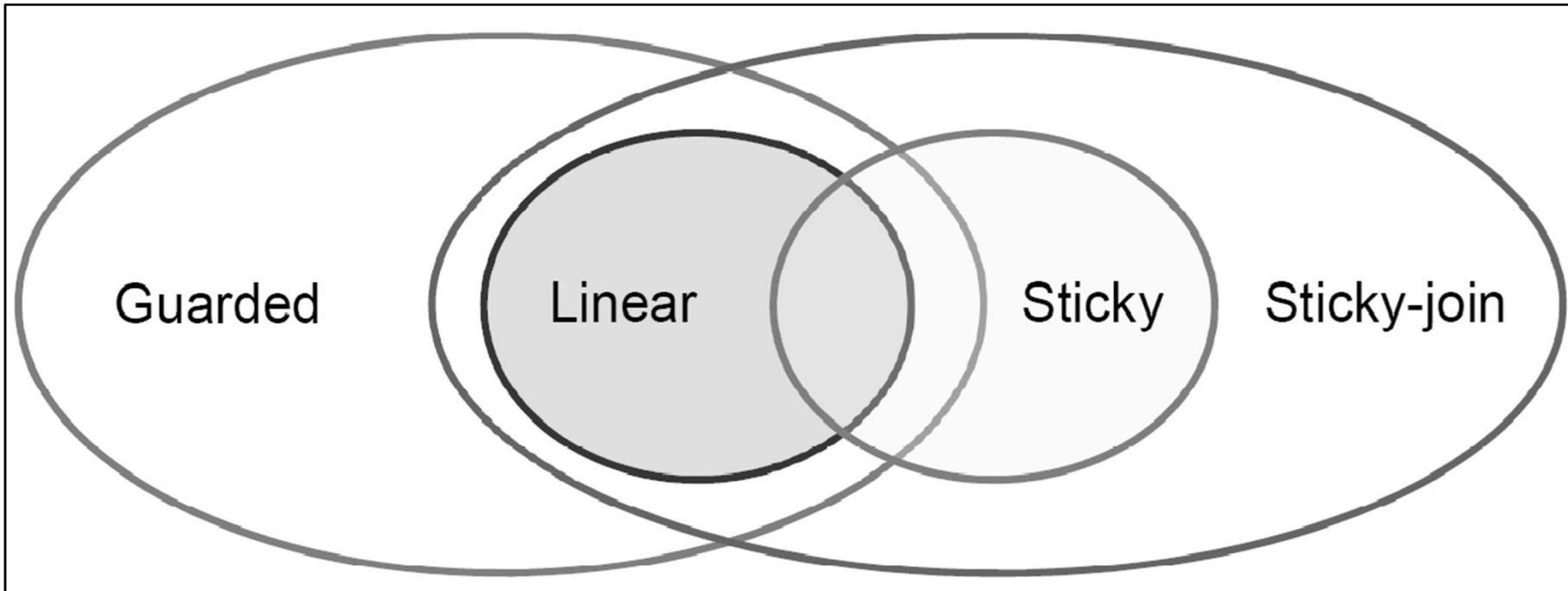
but

$$D \cup \Sigma \models_{\text{fin}} Q$$

Datalog^{+/−}: Resumen

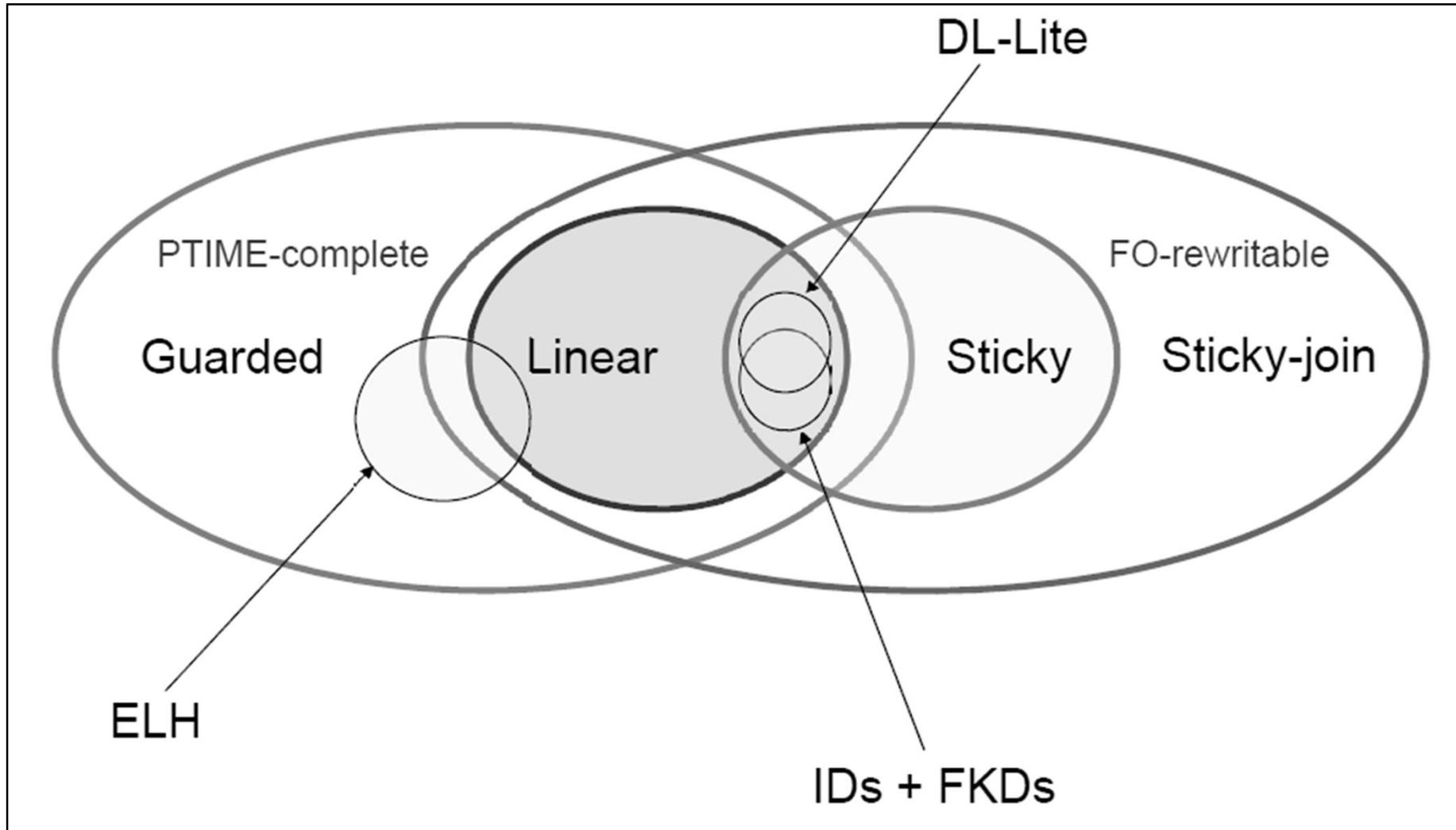


Datalog⁺/−: Resumen



- Sticky-join: sin perder FO rewritability y Polynomial Witness property (PWP) ... pero *más difíciles de identificar*: PSPACE-completo

Datalog⁺/−: Resumen



Datalog^{+/−}: Resumen

	Data	Fixed Σ	Combined
Guarded	PTIME-complete	NP-complete	2EXPTIME-complete
Linear	in AC ₀	NP-complete	PSPACE-complete
Sticky	in AC ₀	NP-complete	EXPTIME-complete
Sticky-join	in AC ₀	NP-complete	EXPTIME-complete

- Misma complejidad con NCs y EGDs *no conflictivas*.
- Misma complejidad bajo *modelos finitos*.

Complejidad de circuitos



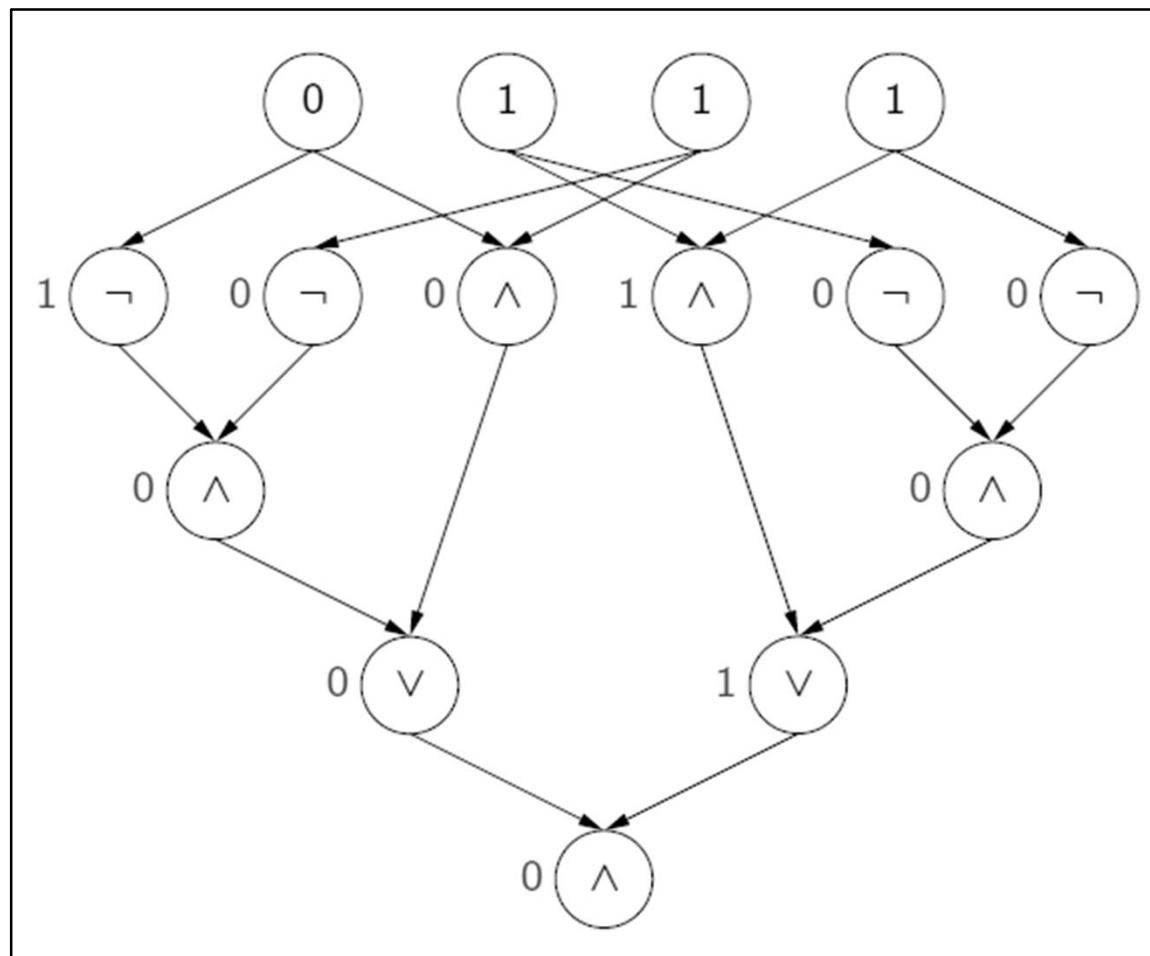
Circuitos Booleanos

- Un *circuito* Booleano con n entradas y m salidas es un grafo acíclico dirigido tal que:
 - Cada nodo *sin arcos* de entrada tiene etiqueta 0 o 1 (*true* o *false*)
 - Cada nodo *con arcos* de entrada tiene etiqueta \neg , \wedge , o \vee (compuertas lógicas)
 - Existe un *único* nodo sin arcos de salida
 - Tamaño = *número* de compuertas
 - Profundidad = largo del *camino más largo* entre un nodo de entrada y el nodo de salida

Circuitos Booleanos

- Cada circuito Booleano puede ser interpretado como una *función*:
 - Toma como entrada los valores en los nodos de entrada
 - Asocia a cada nodo interior u el resultado de *evaluar la etiqueta de u* sobre los valores asociados a los nodos con arcos dirigidos a u
 - Tiene como resultado el valor asociado al nodo de *salida*

Circuitos Booleanos: Ejemplo



Circuitos Booleanos como funciones: Forma normal

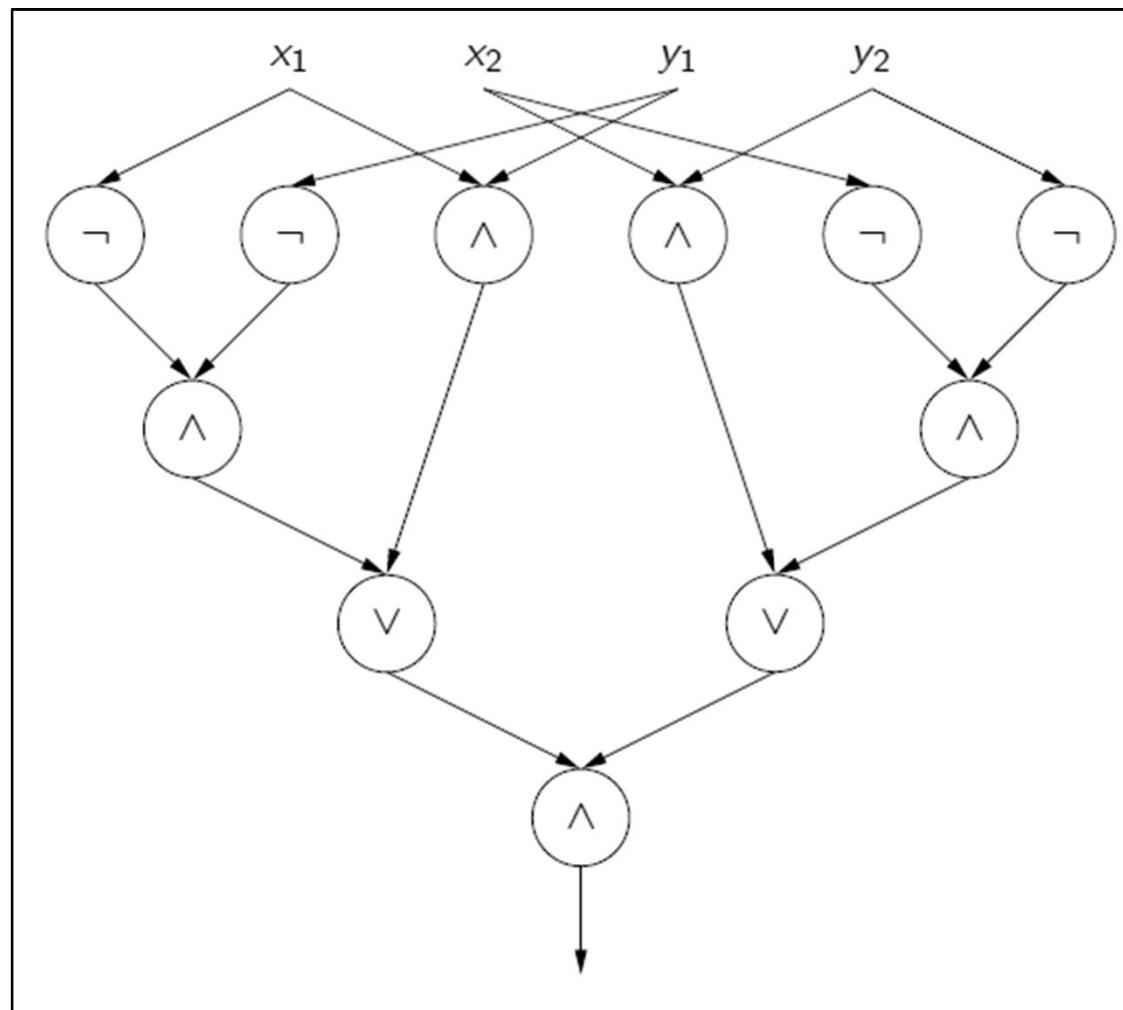
El circuito en el ejemplo anterior representa la función:

$$f(x_1, x_2, y_1, y_2) = \begin{cases} 1 & x_1 = y_1 \text{ y } x_2 = y_2 \\ 0 & \text{en otro caso} \end{cases}$$

Para indicar la función que representa un circuito:

- Reemplazamos las etiquetas 0 y 1 por *variables*
- Usamos un arco sin nodo de destino para indicar la *salida*

Circuitos Booleanos: Ejemplo



Circuitos Booleanos: Forma normal

- Usando $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$ y $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$ podemos demostrar que cada circuito es equivalente a otro que sólo tiene negaciones en el primer nivel.



Circuitos Booleanos: Forma normal

- El circuito generado tiene la misma profundidad que el circuito original y a lo sumo el *doble* de nodos.
- De ahora en adelante: suponemos que los circuitos sólo tienen negaciones en el *primer nivel*.
- Definimos el tamaño de un circuito sólo considerando el número de nodos con *etiquetas* \wedge y \vee .

Lenguajes aceptados por circuitos

- Dado: Circuito $C(X)$ con n entradas.

C define el lenguaje: $\{w \in \{0, 1\}^n \mid C(w) = 1\}$

- Para el caso anterior:

$$\{a_1 a_2 a_3 a_4 \in \{0, 1\}^4 \mid a_1 = a_3 \text{ y } a_2 = a_4\}$$

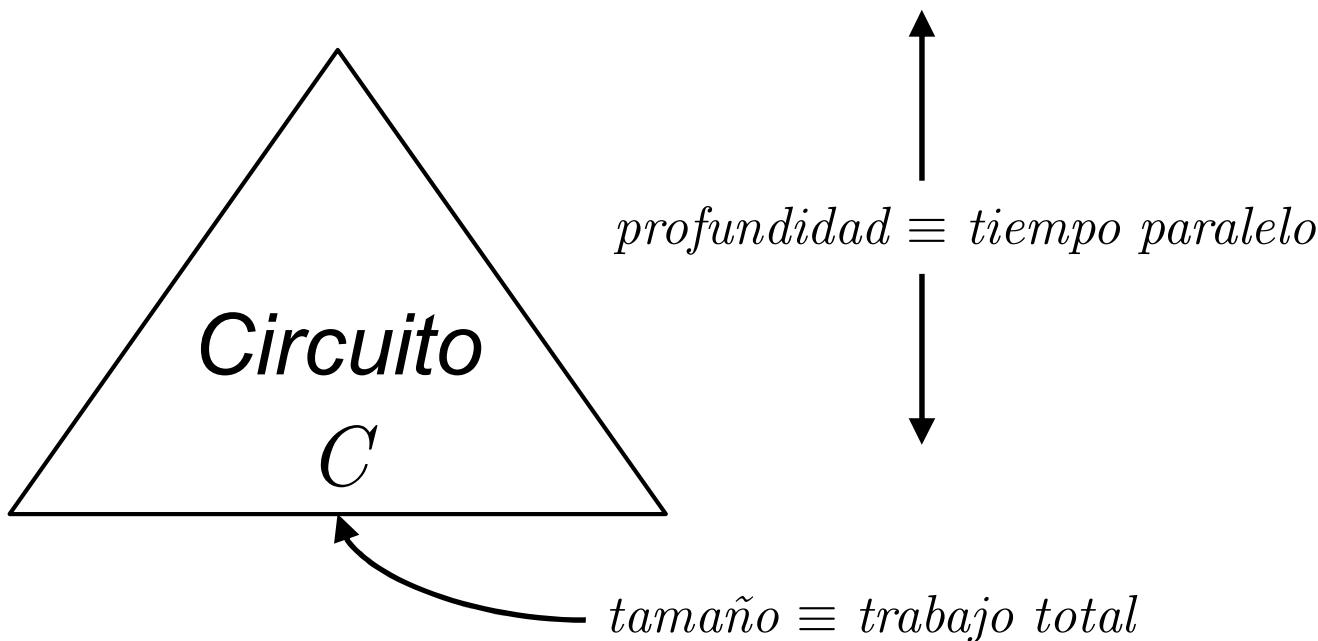
- El lenguaje definido por un circuito siempre es finito:
 - Tenemos que usar *familias* de circuitos para poder generar lenguajes *infinitos*
- Una familia de circuitos $\{C_n\}$, $n \geq 0$ acepta $L \subseteq \{0, 1\}^*$ si:
 - C_n tiene n entradas (C_0 es 0 o 1)
 - Para cada cadena w de largo n : $w \in L$ si y sólo si $C_n(w) = 1$

Lenguajes aceptados por circuitos

- La noción de aceptación que acabamos de definir es muy general.
- Proposición: cada $L \subseteq \{0, 1\}^*$ es aceptado por una *familia* de circuitos $\{C_n\}$.
- Si queremos utilizar una familia de circuitos como un algoritmo tenemos que introducir una noción de *uniformidad*.
 - Debe existir un *algoritmo* que genere los circuitos.
- Una MT determinista M genera una familia de circuitos $\{C_n\}$ si esta máquina con entrada 0^k genera C^k :
 - M genera C^k codificado como una cadena en $\{0, 1\}^*$

Circuitos Booleanos

- Circuitos Booleanos *Uniformes* permiten definir clases de complejidad que refinan el tiempo polinomial:



Clases de complejidad y circuitos

Clase de complejidad AC_i :

- Para $i \geq 0$: Un lenguaje $L \subseteq \{0, 1\}^*$ está en AC_i si existe una familia de circuitos $\{C_n\}$, una MT determinista M y una constante k tal que:
 - L es aceptado por $\{C_n\}$
 - M genera $\{C_n\}$ y funciona en espacio logarítmico
 - $\text{profundidad}(C_n) \leq k * (\log n)^i$

Clases de complejidad y circuitos

Clase de complejidad NC_i :

- Para $i \geq 0$: Un lenguaje $L \subseteq \{0, 1\}^*$ está en NC_i si existe una familia de circuitos con fan-in 2 $\{C_n\}$, una MT determinista M y una constante k tal que:
 - L es aceptado por $\{C_n\}$
 - M genera $\{C_n\}$ y funciona en espacio logarítmico
 - $\text{profundidad}(C_n) \leq k * (\log n)^i$

Clases de complejidad y circuitos

Clases de complejidad AC_i y NC_i :

$$NC_0 \subseteq AC_0 \subseteq NC_1 \subseteq AC_1 \subseteq \dots \subseteq AC_k \subseteq NC_{k+1}$$

$$AC_0 \subseteq NC_1 \subseteq \text{LOGSPACE} \subseteq NC_2$$

AC_0 : Circuitos tienen profundidad constante.

AC_1 : Circuitos tienen profundidad logarítmica.

Esto nos permite resolver muchos problemas.

- Si L es un lenguaje regular, entonces $L \in NC_1$.
- Sea $\Sigma = \{0,1\}$:
 - 1. $\Sigma^* . 1 \in NC_0$
 - $\{0^k 1^k\} \in AC_0$
 - PARES := $\{w \in \Sigma^* \mid w \text{ tiene un número par de 1's}\} \in NC_1$

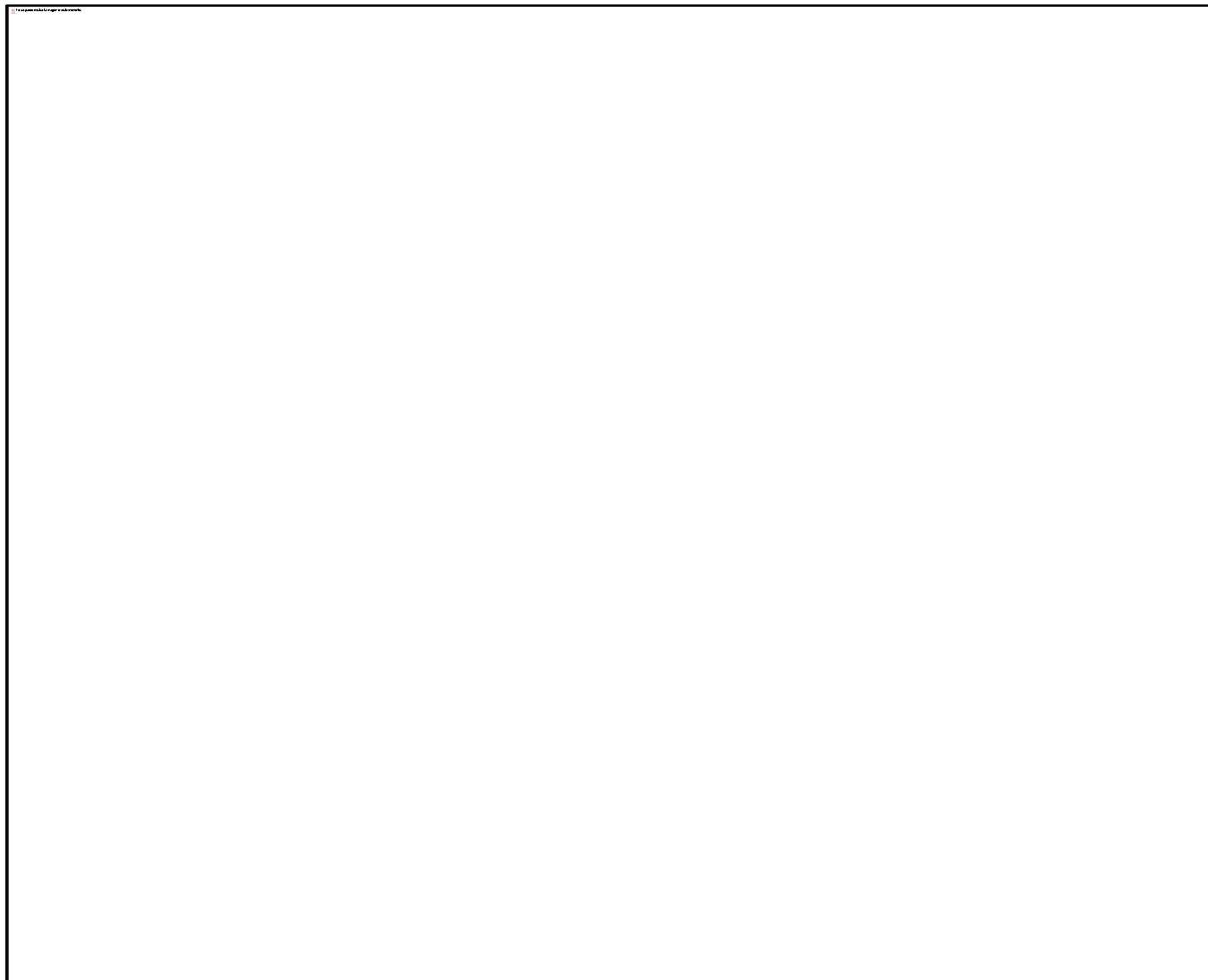
Consultas FO: Complejidad data

- Teorema: Query answering para consultas FO es *completo* para logtime-uniform AC_0 en complejidad data.
- Prueba:
 - Membresía en AC_0 : Dada una base de datos, se *construye* un circuito. La consulta está fija.
 - Hardness: *Transformar* los circuitos a consultas FO (transformación logtime).

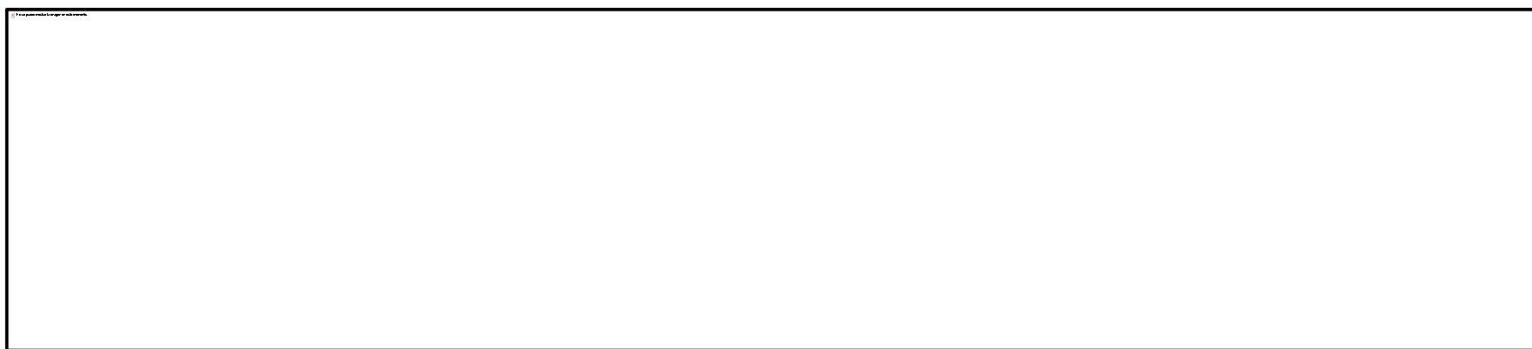
Consultas FO: Complejidad data

- El esquema y la consulta se asumen *fijos*.
- La base de datos y el tamaño del dominio activo *pueden variar*.
- Familias *uniformes*: el número total de compuertas de entrada se determina únicamente por el tamaño del *dominio activo*.
- Ejemplo:
 - Esquema: $R(A,B,C)$, $S(D)$, $T(E,F)$
 - Número de compuertas de entrada: $n^3 + n + n^2$ para algún n (es el tamaño de dominio activo).

Circuitos Booleanos: Ejemplo



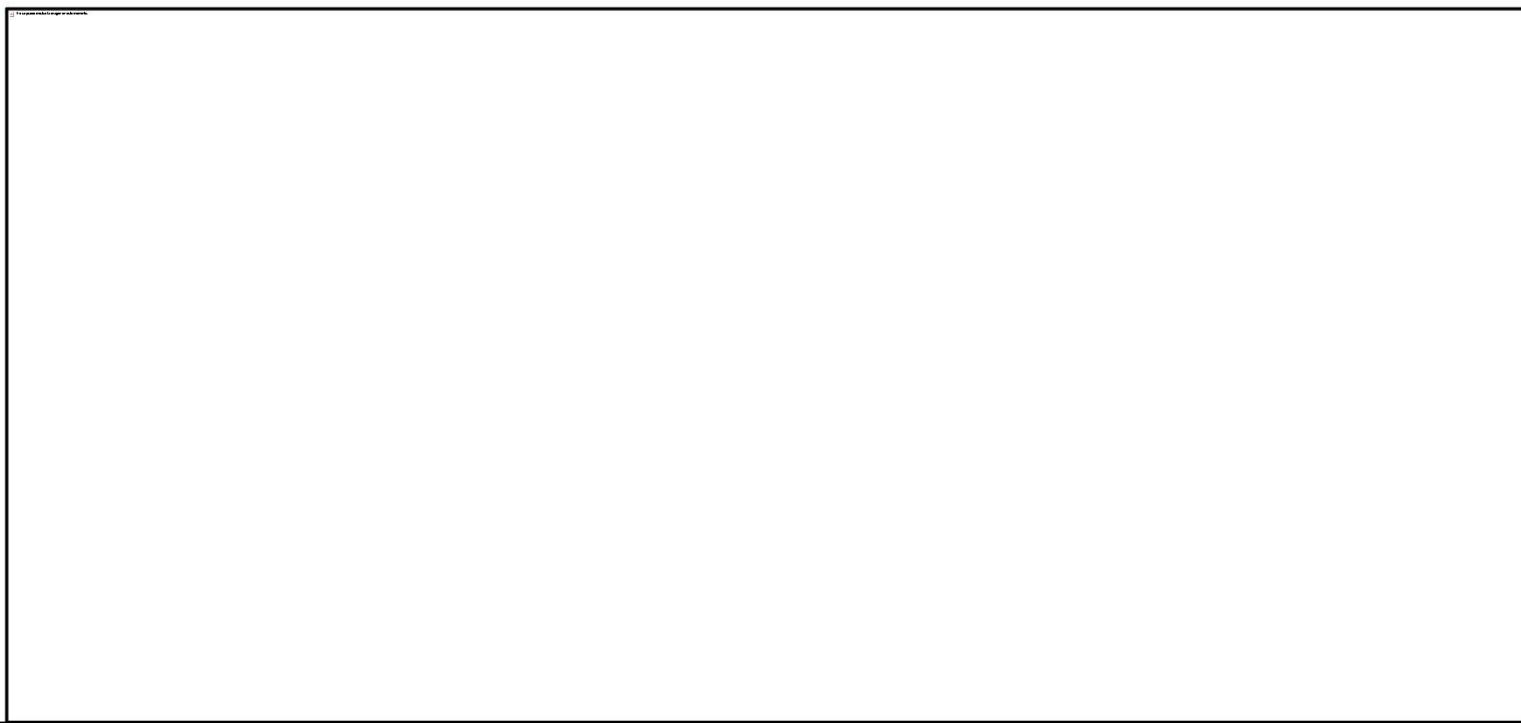
Circuitos Booleanos: Ejemplo



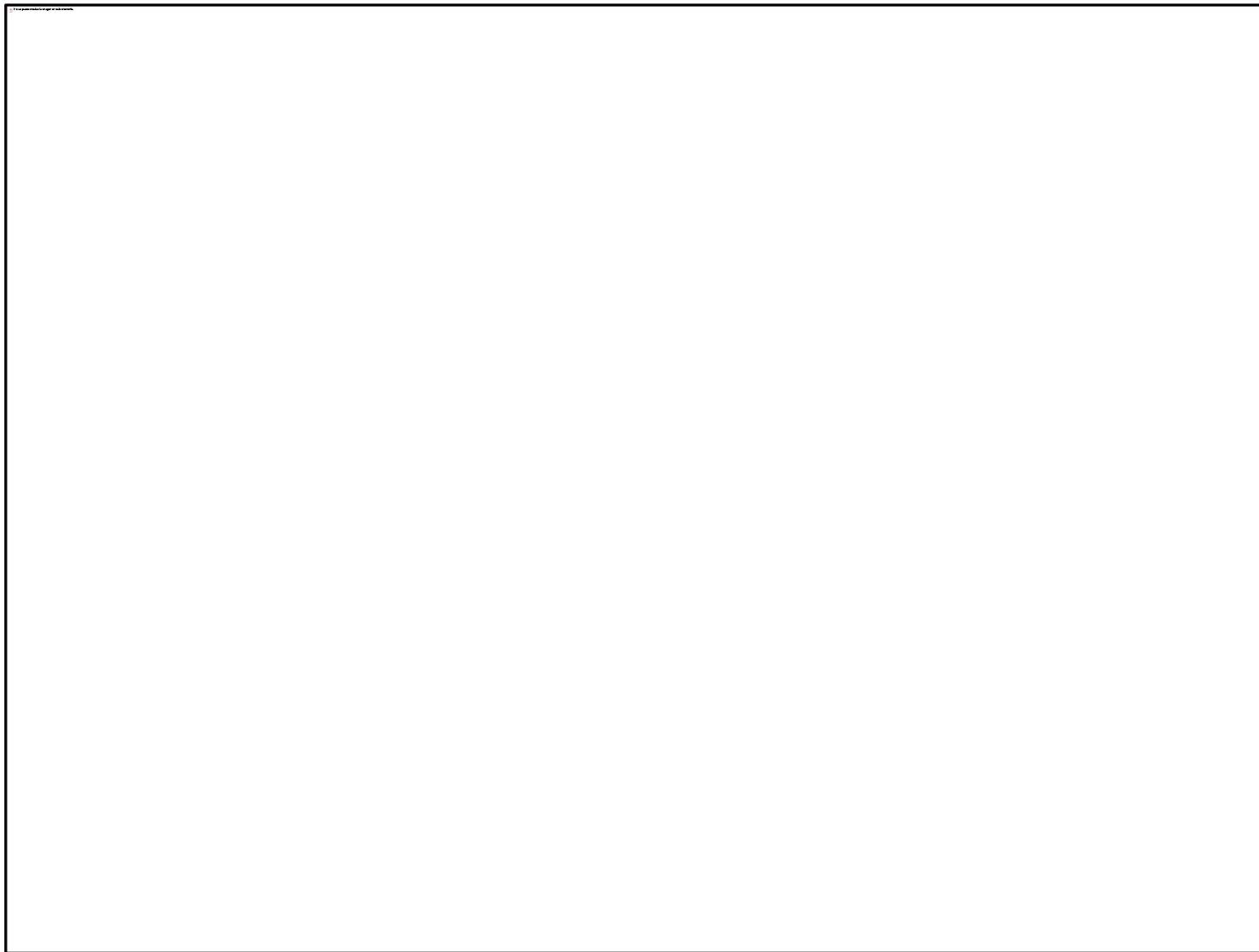
Circuitos Booleanos: Ejemplo



Circuitos Booleanos: Ejemplo



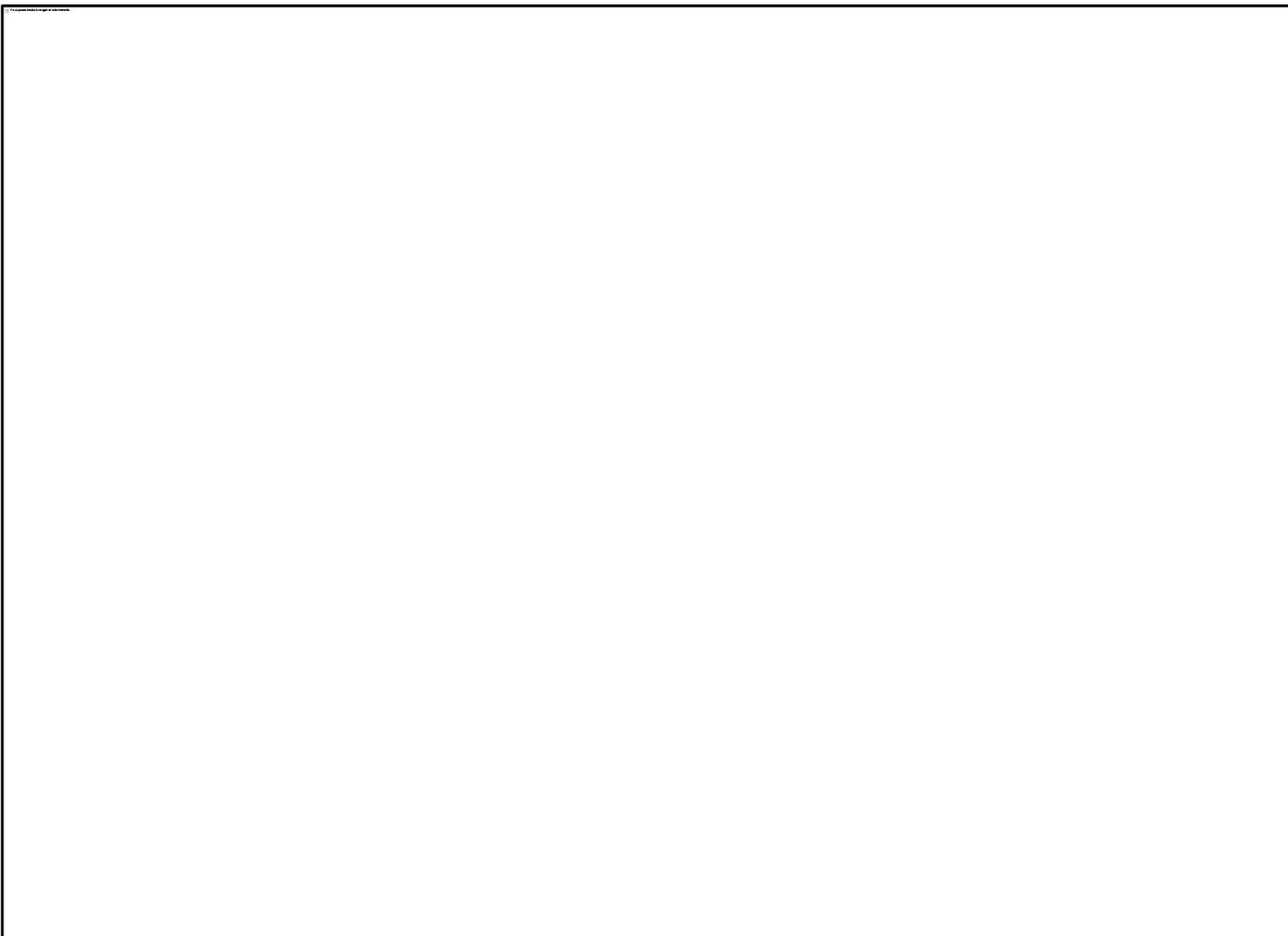
Circuitos Booleanos: Ejemplo



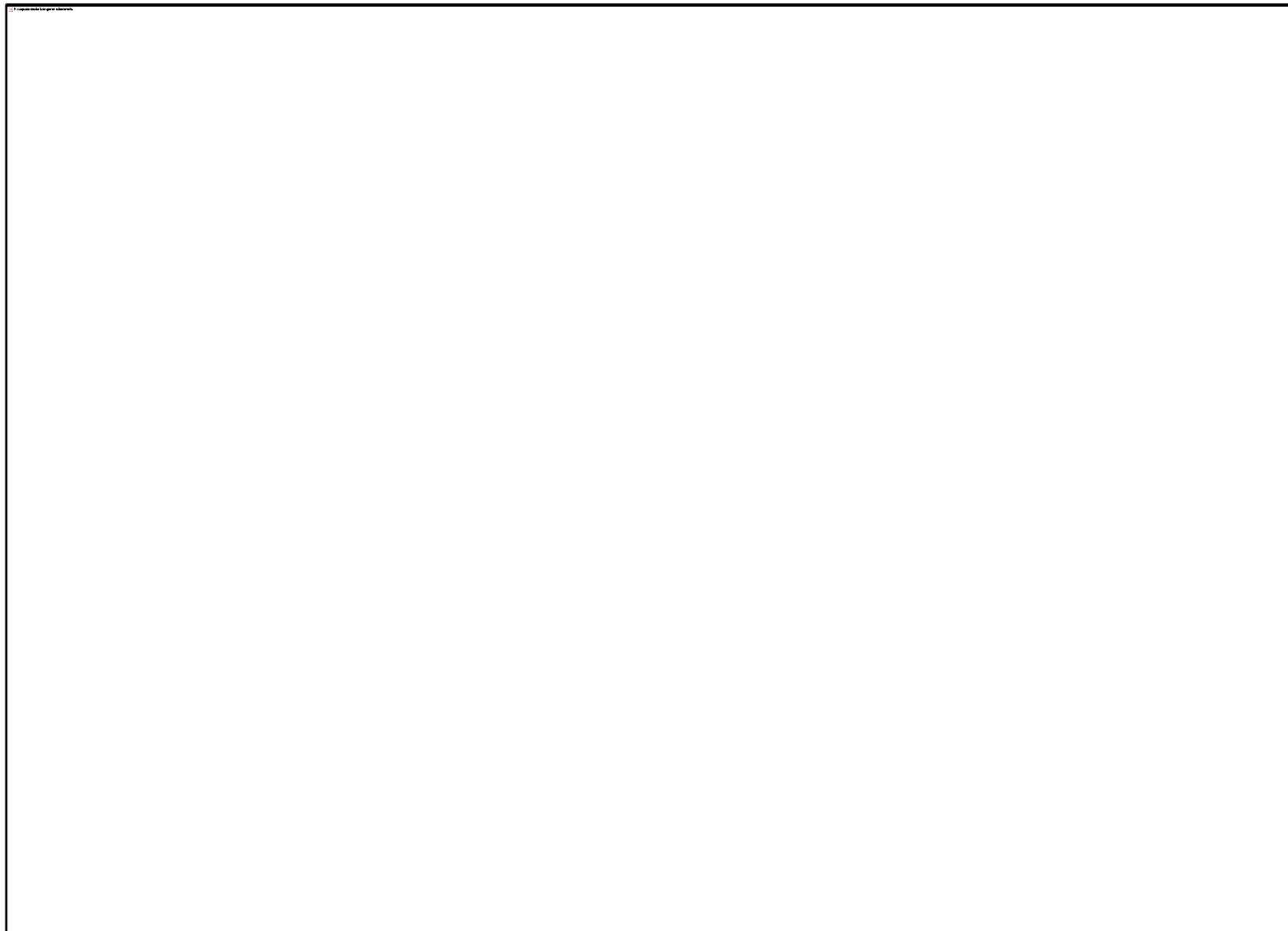
Circuitos Booleanos: Ejemplo



Circuitos Booleanos: Ejemplo



Circuitos Booleanos: Ejemplo



Circuitos Booleanos: Ejemplo



Referencias

[NB2012] Daniele Nardi and Ronald J. Brachman. 2003. “*An introduction to description logics*”. The Description Logic Handbook, Cambridge University Press, New York, NY, USA pp. 1–40.

[CL2007] Diego Calvanese Domenico Lembo. 2007. “*Ontology-based Data Access*”. Tutorial at the 6th International Semantic Web Conference (ISWC 2007).

[Johnson & Klug JCSS 84] D.S. Johnson and A. Klug. “*Testing containment of conjunctive queries under functional and inclusion dependencies*”. JCSS, 28:167189, 1984.

“*Theory of Data and Knowledge Bases*”, dictado originalmente en TU Wien por Georg Gottlob y luego en University of Oxford por Georg Gottlob y Thomas Lukasiewicz.

M. Arenas: “*Complejidad basada en circuitos*”. Complejidad Computacional – IIC3242, Pontificia Universidad Católica de Chile, 2014.

Parte del contenido de este curso está basado en trabajo de investigación realizado en colaboración con Thomas Lukasiewicz, Georg Gottlob, V.S. Subrahmanian, Avigdor Gal, Andreas Pieris, Giorgio Orsi, Livia Predoiu y Oana Tifrea-Marcuska.

DLs: otros constructores

- Disyunción: $\forall hasChild. (Doctor \sqcup Lawyer)$
- Restricciones de valores: $\forall tieneHijo. Femenino$
 - Equivalente a $\forall Y tieneHijo(X, Y) \rightarrow Femenino(Y)$ en FOL
- Restricciones existenciales: $\exists tieneHijo. Femenino$
 - Equivalente a $\exists Y tieneHijo(X, Y) \rightarrow Femenino(Y)$ en FOL.
- Restricciones de número: representan restricciones de cardinalidad en los individuos de los conceptos. Por ejemplo, $(\geq 3 tieneHijo) \sqcap (\leq 2 familiaresFemeninos)$
- Negación de conceptos: $\neg(Doctor \sqcup Lawyer)$
- Inverse role: $\forall hasChild^-. Doctor$
- Reflexive-transitive role closure: $\exists hasChild^*. Doctor$