UNIVERSITÀ DEGLI STUDI DI MODENA
E REGGIO EMILIA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Algoritmi di ottimizzazione di interrogazioni ricorsive nelle Basi di Dati ad Oggetti

Relatore:

Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di:

Ilario Benetti

Correlatori:

Ing. Domenico Beneventano
Ing. Maurizio Vincini

Anno Accademico 1997 - 98

Parole chiave:

Basi di dati ad oggetti
Ottimizzazione semantica di interrogazioni
Interrogazioni ricorsive
Logiche descrittive
Scomposizione in fattori

RINGRAZIAMENTI

Al termine di questo lavoro intendo ringraziare la Prof. Sonia Bergamaschi per la preziosa collaborazione prestata allo sviluppo del progetto.
Un dovuto ringraziamento va inoltre all'ing. Domenico Beneventano per il fondamentale contributo alla formalizzazione teorica dei risultati e all'ing. Maurizio Vincini per la puntuale assistenza nella reingegnerizzazione del sistema.
Infine intendo esprimere la piú sentita riconoscenza nei confronti dell'ing. Valeria Cannillo, dell'ing Alberto Corni e dell'ing. Roberto Montanari per l'aiuto prestato in questi mesi di lavoro.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Le Basi di Dati Orientate ad Oggetti, OODB (*Object Oriented Database*), sono da anni oggetto di intensi sforzi di ricerca e di sviluppo poiché il paradigma orientato ad oggetti offre una gamma di strutture dati e di facilità di manipolazione tali da renderlo adatto a supportare tanto le tradizionali funzionalitá quanto le nuove applicazioni.

L'aspetto fondamentale dei modelli di dati orientato ad oggetti, OODM (*Object Oriented Data Model*), proposti per OODB in [A+89] e [KL86], è che sono basati sulla definizione di oggetti, classi, attributi, inferenza e metodi, cosicché le classi e gli attributi possono essere usati per descrivere gli aspetti strutturali (la conoscenza in un dominio di applicazione) mentre i metodi possono essere usati per rappresentare gli aspetti comportamentali (come ad esempio l'incapsulazione).

Il problema di rappresentare la conoscenza è stato trattato anche dalla comunità della Intelligenza Artificiale. In questo settore la ricerca ha prodotto soprattutto formalismi di rappresentazione della conoscenza basati su una logica ristretta in modo da ottenere efficienti tecniche di inferenza.

In particolare verrá adottata la definizione formale di un modello di dati orientato ad oggetti presentato in [BN94a] e il suo accoppiamento con tecniche di inferenza sviluppate nei modelli di rappresentazione della conoscenza nell'area dell'Intelligenza Artificiale. Le tecniche di inferenza sono basate sulla relazione di sussunzione tra due classi di oggetti, cioè sulla relazione esistente tra due classi quando l'appartenenza di un oggetto ad una classe implica necessariamente l'appartenenza dell'oggetto all'altra classe.

Il prototipo ODB-Tools [BBSV97], in corso di sviluppo presso il dipartimento di scienze dell'ingegneria dell'universitá degli studi di Modena e Reggio Emilia, mostra l'applicabilitá delle tecniche di inferenza sopra citate in attivitá centrali per le basi di dati ad oggetti quali l'acquisizione di schemi

consistenti e minimali e l'ottimizzazione delle interrogazioni.

Obiettivo di questa tesi è l'integrazione del software ODB-Tools con le funzionalitá di gestione dei cicli; ci si propone, in altri termini, di garantire la possibilitá di acquisire schemi ciclici nonché di realizzare l'ottimizzazione semantica di interrogazioni ricorsive.

1.1 I modelli di dati orientati ad oggetti

Nel seguito sono brevemente riportate le principali caratteristiche strutturali di un OODM.

Ogni singolo componente della realtà da modellare viene rappresentato tramite un unico concetto di base: l'*oggetto*. Ogni oggetto è univocamente identificato da un identificatore di oggetto, *oid* (object identifier) [KC86], ed ha associato uno stato che è costituito dal valore delle sue *proprietà* o *attributi*. Nei classici linguaggi orientati ad oggetti, come ad esempio Smalltalk [GR83], il valore associato ad un oggetto è sempre atomico oppure una tupla di altri oggetti. Questa caratteristica è stata in parte ereditata da alcuni sistemi OODB, dove questo valore è una tupla oppure un insieme di altri oggetti, cioè è sempre un valore piatto che può solo contenere identificatori di altri oggetti e non direttamente altri valori complessi.

Per superare questa limitazione sono stati sviluppati diversi modelli ad oggetti con valori complessi [AK89, Atz93, LR89, Bee90]. In questi modelli si trattano in modo uniforme sia oggetti con identità sia valori complessi senza identità. Un *valore complesso* o *valore strutturato* è un valore definito a partire sia da valori atomici che da identificatori di oggetti mediante l'uso ricorsivo di costruttori, quali ad esempio il costruttore di *tupla*, di *insieme*, e di *sequenza*. Uno *schema* contiene le informazioni sulla struttura dei dati. Nei citati lavori sono presenti entrambe le nozioni di *classe* e di *tipo*. I tipi denotano una struttura e una *estensione*, intesa come insieme di valori. Anche le classi denotano un'estensione, intesa come insieme di oggetti. Comunque, mentre l'estensione denotata da un tipo è definita dalla sua struttura, l'estensione associata ad una classe è definita dall'utente. Ad ogni classe è normalmente associato un tipo che descrive la struttura degli oggetti che possono essere *istanziati* nella classe. Quindi le istanze della classe sono oggetti il cui valore associato è, a sua volta, istanza del tipo che descrive la classe.

L'*ereditarietà*, stabilita tramite la relazione *isa*, è un'importante caratteristica degli OODB. Uno dei principali vantaggi dell'ereditarietà è che essa costituisce un potente mezzo di modellazione, essendo in grado di dare una

precisa e concisa descrizione del dominio di applicazione [A+89]. Da un punto di vista intensionale, la dichiarazione di ereditarietà tra due classi A e B, cioè A *isa* B, consente la definizione della *sottoclasse* A come specializzazione della *superclasse* B. Una sottoclasse eredita le proprietà della superclasse, può avere proprietà aggiuntive e può ridefinire alcune proprietà della superclasse. Da un punto di vista estensionale, la dichiarazione A *isa* B, stabilisce che ogni oggetto di A sia anche un oggetto di B. Nel caso in cui una classe può avere più di una superclasse si parla di *ereditarietà multipla*.

Il modello introdotto nella presente tesi è un modello per basi di dati orientate ad oggetti con ereditarietà multipla che permette la modellazione di valori complessi tramite la nozione di tipo e di classe.

1.2 Sussunzione nei modelli orientati ad oggetti

Nei classici modelli orientati ad oggetti la descrizione di una classe è intesa solo a rappresentare una struttura dati e la classe deve essere esplicitamente riempita con oggetti, cioè l'*estensione* della classe è soggetta solo a condizioni necessarie. Nel nostro modello questa semantica viene rappresentata tramite i cosiddetti concetti *primitivi*. In aggiunta un concetto può essere *definito*, nel qual caso la struttura esprime condizioni sia necessarie che sufficienti.

Quindi, la fondamentale differenza del modello proposto nella tesi rispetto ai precedenti OODM, è la nozione, accanto a quella originaria di classe qui denominata come classe base, di *definizione di classe* tramite la cosiddetta *classe virtuale*. Si è preferito adottare il termine *virtuale*, anziché *definita*, perchè nella terminologia delle basi di dati tradizionalmente i tipi la cui estensione è determinata sulla base della loro espressione vengono denominati in questo modo. Inoltre, la nozione di *definizione di classe* è simile alla nozione di *vista* delle basi di dati, recentemente detta anche *classe virtuale* [AB91] nell'ambito dei sistemi ad oggetti.

Altra differenza è quella che nel nostro modello viene utilizzata la relazione di *sussunzione*, o generalizzazione, tra due classi, ovvero la relazione esistente tra due classi quando ogni elemento istanziato in una classe deve essere necessariamente, in virtú delle descrizioni, istanziato anche nell'altra classe. L'idea di sussunzione è quindi molto intuitiva:

$$\textit{classe}_1 \text{ sussume } \textit{classe}_2 \text{ se e solo se}$$
$$\text{ogni oggetto che è } \textit{classe}_2 \text{ è anche } \textit{classe}_1.$$

La presenza di classi definite (virtuali) comporta che le relazioni di specializzazioni stabilite nella tassonomia non sono solo quelle dichiarate esplicitamente dall'utente ma ve ne sono delle ulteriori *implicite* nelle descrizioni delle classi. Il progettista della base di conoscenza descrive una classe in termini di ereditarietà da altre classi e di proprietà *locali*, e il sistema *classifica* automaticamente la classe, cioè determina il suo posto "giusto" nella tassonomia esistente, tra le sue più specifiche generalizzazioni e le sue più generali specializzazioni. La classificazione è effettuata tramite il cosiddetto *ragionatore tassonomico* che trova tutte le relazioni di sussunzione tra la classe in questione e le classi nella tassonomia già esistente. Il ragionatore tassonomico è quindi un servizio *inferenziale*, o *deduttivo*, fornito all'utente dal sistema di rappresentazione.

Altro concetto utilizzato nella tesi è quello di *incoerenza*: intuitivamente un tipo si dice incoerente sse non è possibile trovare una istanza che soddisfi le condizioni di appartenenza alla descrizione del tipo.

L'intera base di dati si dirá incoerente se esiste almeno un tipo incoerente.

1.3 L'ottimizzazione semantica delle interrogazioni

L'applicazione del ragionamento tassonomico ai tradizionali modelli semantici dei dati porta a promettenti risultati per il progetto di schemi per basi di dati [FS86, DD89, BS91] e per altri rilevanti aspetti quali l'elaborazione di interrogazioni e il riconoscimento dei dati [BGN89, BBMR89].

L'obiettivo dell'ottimizzazione delle interrogazioni è quella di trasformare un'interrogazione in una *equivalente*, la quale restituisce lo stesso risultato di quella originale, con un minor costo di esecuzione. A differenza dell'ottimizzazione convenzionale [JK84], basata sulla conoscenza sintattica dell'interrogazione e sull'organizzazione fisica della base di dati, con la tecnica dell'ottimizzazione semantica le trasformazioni avvengono utilizzando la conoscenza semantica relativa alla base di dati [Kin81b, HZ80]. L'idea di base è che i *vincoli di integrità*, espressi per forzare la consistenza di una base di dati, possono essere utilizzati anche per ottimizzare le interrogazioni fatte dall'utente. Quindi il modello ad oggetti viene arricchito con vincoli di integrità espressi come *regole* [D.B94]. Intuitivamente il processo di espansione semantica è semplice: a partire da uno schema che definisce l'ODBMS di partenza supponiamo di aggiungere, a run-time, una interrogazione. L'ottimizzazione consiste nel controllare dapprima che l'interrogazione sia coerente (cioè che non esprima una classe di oggetti vuota) e poi cercare di incorporare ogni possibile restrizione che non sia presente nell'interrogazione di partenza ma che è *logicamente implicata* dal tipo della query e dallo schema (in partico-

lare dai vincoli di integrità). L'espansione semantica è basata sull'iterazione di questa semplice trasformazione: se l'interrogazione implica l'antecedente di un vincolo di integrità allora posso aggiungere il conseguente di quel particolare vincolo di integrità. Si noti che una query esprime la semantica di una classe virtuale, cioè la sua descrizione esprime un insieme di condizioni necessarie e sufficienti affinchè un oggetto del dominio applicativo appartenga all'insieme degli oggetti che soddisfano la query.

La relazione di implicazione logica viene ottenuta attraverso l'uso della relazione di sussunzione: ciò è motivato soprattutto dalla considerazione che la definizione formale di sussunzione come relazione semantica permette di dimostrare la correttezza delle trasformazioni, cioè di dimostrare che le interrogazioni trasformate siano equivalenti a quella originale [BM92]. Un'altra considerazione è che le trasformazioni riguardano esclusivamente quella parte dell'interrogazione che è esprimibile come una descrizione del modello, quindi un'espressione per la quale, in generale, il calcolo della sussunzione è efficiente.

Il risultato finale del metodo è quello di riuscire a riclassificare l'interrogazione dopo l'espansione semantica secondo la tassonomia delle classi dello schema rispetto alle nuove relazione di ereditarietà (*isa*) trovate e, quindi, di ottenere uno spostamento verso il basso della query nella tassonomia, restringendo l'insieme degli oggetti candidati a soddisfare la query.

In particolare, in questo lavoro, si intende estendere il risultato anche alle situazioni cicliche: in questi casi, infatti, la riclassificazione verso il basso di una interrogazione nella tassonomia delle classi diviene ricorsiva e puó determinare, in linea di principio, problemi di terminazione.

1.4 Esempi

Allo scopo di illustrare il metodo, andiamo a considerare il seguente esempio che riguarda la struttura di un società che si occupa della gestione di un magazzino (vedi figura 1.1, dove le classi sono rappresentate da ellissi, le relazioni di specializzazione esplicite da frecce in neretto, le relazioni di sussunzione calcolate da frecce tratteggiate e gli attributi da archi orientati.

I materiali (`material`) sono descritti da un nome (`name`) dato da una stringa di caratteri, un rischio (`risk`) dato da un intero e da una caratteristica (`feature`) data da un insieme di stringhe. I magazzini (`storage`) sono identificati da una categoria (`category`) data da una stringa, sono guidati (`managed-by`) da un dirigente (`manager`) e contengono (`stock`) un insieme di articoli (`item`) che sono dei materiali (`material`) per ciascuno dei quali è indicata con un intero la quantità presente (`qty`). I dirigenti (`manager`)

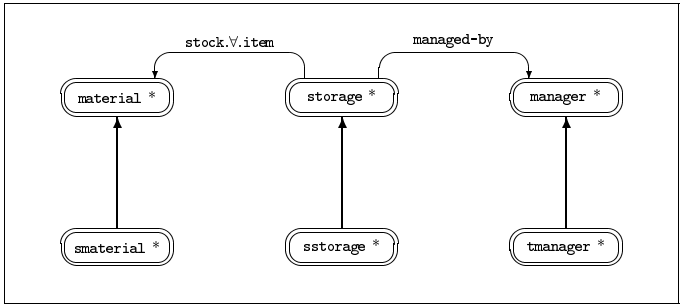


Figura 1.1: Il dominio Magazzino

hanno un nome (`name`) dato da una stringa, un salario (`salary`) superiore ai 40K dollari e un livello (`level`) compreso tra 1 e 15. I massimi dirigenti (`tmanager`) sono quei dirigenti (`manager`) che hanno un livello compreso tra 8 e 12. Infine abbiamo dei magazzini marcati come "speciali" (`sstorage`) che sono magazzini (`storage`) e materiali "speciali" (`smaterial`) che sono materiali (`material`). Le classi che abbiamo descritto sono tutte di tipo base, avendo fissato solo condizioni necessarie per l'appartenenza, quindi, per ora, le relazioni di sussunzione sono soltanto quelle esplicite mostrate dalle frecce in neretto in figura 1.1.
A queste classi vogliamo ora aggiungere i vincoli di integrità, i quali rappresentano condizioni necessarie e sufficienti per la legalità delle istanze dello schema. Questi vincoli possono essere descritti in linguaggio naturale nel seguente modo:

" **per tutti** i dirigenti (`manager`),
se il livello (`level`) è compreso tra 5 e 10,
allora il salario (`salary`) deve essere compreso tra 40K e 60K. "

" **per tutti** i materiali (`material`),
se il rischio (`risk`) è maggiore di 10,
allora devono essere dei materiali speciali (`smaterial`). "

" **per tutti** i magazzini (`storage`),
se la categoria (`category`) è "B4",
allora devono essere guidati da un massimo dirigente (`tmanager`). "

" **per tutti** i magazzini (`storage`),
se ciascun articolo (`item`) immagazzinato (`stock`)
è di un materiale speciale (`smaterial`),
allora devono essere dei magazzini speciali (`sstorage`). "

" **per tutti** i magazzini (`storage`),
se ciascuna quantità (`qty`) immagazzinata (`stock`)
è compresa tra 10 e 50,
allora la categoria (`category`) deve essere "A2". "

Usando un linguaggio di tipo OODB-like, lo schema del Magazzino comprensivo dei vincoli di integrità è riportato nella tabella 1.1.

class material = [name: string, risk: integer, feature : {string}]
class smaterial = **isa** material
class storage = [managed-by : manager, category : string, stock : {item : material, qty : $10 \div 300$]}]
class sstorage = **isa** storage
class manager = [name: string, salary: $40K \div \infty$, level : $1 \div 15$]
class tmanager = **isa** manager **and** [level : $8 \div 12$]

Vincoli di integrità:

if manager **and** (level : $5 \div 10$)	**then**	(salary : $40K \div 60K$)
if material **and** (risk > 10)	**then**	smaterial
if storage **and** (category $= "B4"$)	**then**	(managed-by : tmanager)
if storage **and** (stock.$\forall$.item = smaterial)	**then**	sstorage
if storage **and** (stock.$\forall$.qty : $10 \div 50$)	**then**	(category $= "B4"$)

Tabella 1.1: Lo schema del Magazzino in una sintassi OODB–like

A partire dallo schema con regole vogliamo ora mostrare come opera il metodo di ottimizzazione semantica delle interrogazioni in [BBS94]; il primo esempio riguarda la rilevazione di una interrogazione incoerente, la quale permette di fornire una risposta senza accedere al database.
Vediamo quindi la seguente interrogazione:

Q_1 : "Seleziona i dirigenti che hanno un livello superiore a 20".

La definizione di una interrogazione è esplicitamente una classe virtuale, poichè sono richieste condizioni necessarie e sufficienti di appartenenza, quindi, nella sintassi OODB-like introdotta, la Q_1 può essere espressa nel seguente modo:

virtual class Q_1 = **isa** manager **and** [level : $20 \div \infty$]

Poichè l'intervallo per il livello definito nell'interrogazione (level : $20 \div \infty$) è disgiunto da quello definito nella classe manager (level : $1 \div 15$) risulta che Q_1 è incoerente quindi nessun dominio di oggetti può popolare la classe Q_1.
Se ora applichiamo il nostro metodo di ottimizzazione semantica alla Q_1 viene riconosciuta l'incoerenza dell'interrogazione e quindi possiamo subito affermare che la ricerca non fornirá alcun risultato senza dover accedere fisicamente al database.

Il successivo esempio illustra l'effettiva ottimizzazione calcolata tramite la tecnica dell'espansione semantica che riclassifica l'interrogazione introducendo la conoscenza contenuta nei vincoli di integrità:

Q_2 : "Seleziona i magazzini che hanno categoria "B4", sono guidati da un funzionario con livello inferiore a 10 e contengono materiali ciascuno con rischio superiore a 10".

La corrispondente rappresentazione in sintassi OODB-like diventa:

virtual class Q_2 = **isa** storage
and [managed-by : **isa** manager **and** (level < 10),
stock.$\forall$.item : **isa** material **and** (risk > 10),
category $=$ "$B4$"]

Per l'interrogazione Q_2 possiamo immediatamente calcolare la classificazione nella gerarchia delle classi che è rappresentata in figura 1.2.a.
Vediamo il comportamento dell'ottimizzatore semantico: poichè l'interrogazione è coerente cerca di applicare i vincoli di integrità: a Q_2 possiamo applicare il secondo e il terzo vincolo di integrità (essendo verificate le condizioni) dando origine ad una nuova interrogazione trasformata:

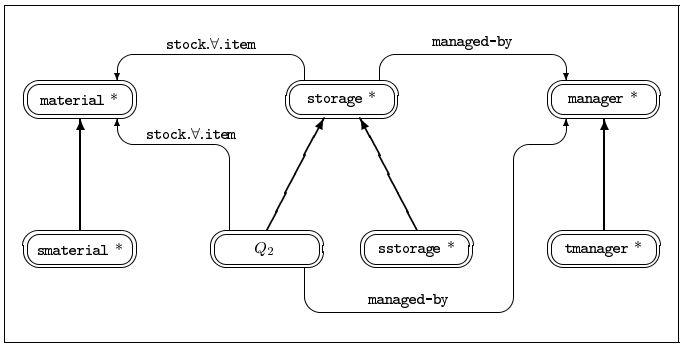


Figura 1.2.a: La query Q_2 prima dell'espansione semantica

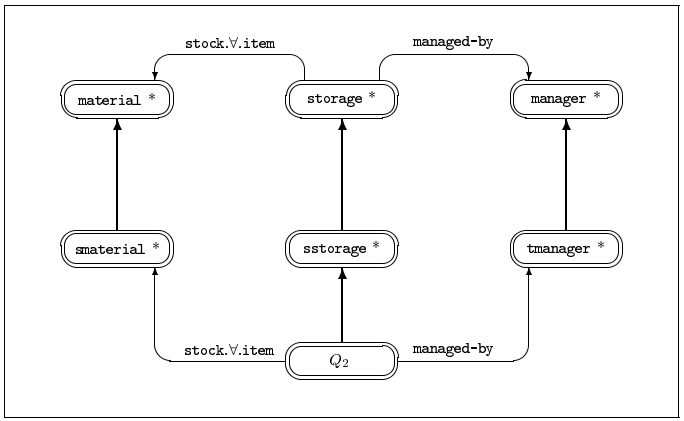


Figura 1.2.b: La query Q_2 dopo l'espansione semantica

Figura 1.2: Classificazione dell'interrogazione nella tassonomia delle classi

$$\underline{\textbf{virtual class}} \quad Q_2' \quad = \quad \underline{\textbf{isa}}\ \texttt{storage}\ \underline{\textbf{and}}\ [\texttt{managed-by}: \underline{\textbf{isa}}\ \texttt{tmanager}\ \underline{\textbf{and}}\ (\texttt{level}\ < 10),\ \texttt{stock.}\forall\texttt{.item}: \underline{\textbf{isa}}\ \texttt{smaterial}\ \underline{\textbf{and}}\ (\texttt{risk} > 10),\ \texttt{category} = \text{“}B4''\ ]$$

Iterando il procedimento di controllo sull'applicabilità dei vincoli di integrità alla Q_2' l'ottimizzatore trova che possiamo applicare il primo e il quarto vincolo dando origine alla:

$$\underline{\textbf{virtual class}} \quad Q_2'' \quad = \quad \underline{\textbf{isa}}\ \texttt{sstorage}\ \underline{\textbf{and}}\ [\texttt{managed-by}: \underline{\textbf{isa}}\ \texttt{tmanager}\ \underline{\textbf{and}}\ (\texttt{level}\ < 10),\ \underline{\textbf{and}}\ (\texttt{salary}\ : 40K \div 60K),\ \texttt{stock.}\forall\texttt{.item}: \underline{\textbf{isa}}\ \texttt{smaterial}\ \underline{\textbf{and}}\ (\texttt{risk} > 10),\ \texttt{category} = \text{“}B4''\ ]$$

Alla classe Q_2'' non è possibile applicare ulteriori vincoli di integrità; ció significa che abbiamo trovato l'espansione semantica dell'interrogazione che rappresenta la classe più specializzata tra quelle equivalenti all'interrogazione di partenza.

La trasformazione ottenuta per la Q_2 porta alla nuova rappresentazione dell'interrogazione nella gerarchia delle classi mostrata in figura 1.2.b. Si può notare come l'interrogazione risulti ottimizzata in quanto essa è ora una specializzazione di `sstorage` (invece della sola classe `storage`) ed inoltre contenga l'attributo `managed-by` il cui dominio ha valori in `tmanager` (anzichè `manager`) e l'attributo `stock` con valori in `smaterial` (anzichè `material`).

Il problema, come é giá stato accennato, diviene piú complesso se intendiamo determinare incoerenze e calcolare relazioni di sussunzione in presenza di interrogazioni e viste ricorsive. Per la trattazione di queste situazioni é opportuno fare un breve riferimento al concetto di chiusura transitiva.

La chiusura transitiva di un attributo, che peraltro non si puó esprimere in OQL, rappresenta una notazione per esprimere la ricorsione ed é stata introdotta in [BNPS92] per linguaggi di interrogazione di OODB e in [AV97] per le cosiddette *path queries*. Consideriamo dunque, a titolo di esempio, seguente lo schema di tabella 1.2.

```
interface Department () { attribute string dname;
                          attribute  range {1, 7}  category;
                          attribute  set<Employee> employs;
                          attribute  Employee administrator; };

interface Employee () {   attribute string name;
                          attribute range {1, 10}  qualification;
                          attribute range {10, 100}  salary;
                          attribute Employee  head;
                          attribute Department worksin; };

interface Manager: Employee () { attribute range {8, 10}  qualification;
                                 attribute Manager  head;
                                 attribute Department directs; };

interface CB_Department: Department () { attribute  range {5, 7}  category;
                                         attribute  Manager administrator; };

view Mdl_Employee: Employee () {   attribute range {5, 10}  qualification; };

rule r_1 forall X in Manager:  X.qualification >= 9
   then X.directs in CB_Department ;

rule r_2 forall X in Department:  forall Y in X.employs : Y.qualification >= 5
   then X in CB_Department  ;

rule r_3 forall X in Employee:  X.salary >= 60  and  X.head.qualification>=8
   then X in Manager ;

rule r_4 forall X in Manager: X.directs.category >= 5 and X.salary>=50
   then X.qualification = 10  ;
```

Tabella 1.2: Schema di esempio con la sintassi ODL–ODMG93 estesa

Prendiamo quindi in considerazione un'interrogazione ciclica:

Q_3 : "seleziona tutti gli impiegati che percepiscono un salario maggiore o uguale a 80 e che hanno un capo, a qualunque livello, con una qualifica maggiore o uguale a 6".

Supponiamo ora che il linguaggio utilizzato per interrogare il nostro OODB, l'OQL, sia in grado di esprimere la chiusura riflessiva e transitiva di un attributo `a`, indicandola con la notazione `(a)*`; in questo caso, potremmo esprimere la query Q_3 come:

```
select  *
from    Employee
where   salary >=80
and     (head)*.qualification >= 6
```

Applicando `r_3` il fattore ricorsivo `(head)*.qualification >= 6` puó essere eliminato dalla query ottenendo, in definitiva, una interrogazione non ricorsiva:

```
select  *
from    Manager
where   salary >=80
```

L'operatore `()*` può essere applicato anche ad un path di attributi; inoltre, l'espressione risultante dall'applicazione dell'operatore `()*`, può essere utilizzata ovunque si possa utilizzare un attributo; ad esempio, possiamo utilizzarlo in una espressione quantificata, come nella seguente query:

Q_4 : "seleziona gli impiegati con qualifica maggiore o uguale a 6 che lavorano in un dipartimento in cui vi sono esclusivamente impiegati che hanno *ricorsivamente* le stesse proprietá":

```
select  *
from    Employee
where   forall X in (worksin.employs)* : X.qualification >= 6
```

Applicando la regola `r_2`, si riesce ad ottenere un'ottimizzazione sulle classi nella gerarchia di aggregazione della query, sostituendo la classe implicita Department con la sottoclasse CB_Department:

```
select  *
from    Employee
where   forall X in (worksin[CB_Department].employs)* :
        X.qualification >= 6
```

I due esempi presentati hanno messo in luce da un lato le caratteristiche di una query ciclica, dall'altro come la chiusura transitiva costituisca uno strumento formale molto potente nella trattazione di questo tipo di interrogazioni.

Un ulteriore obiettivo di questo lavoro sará dunque quello di realizzare un componente che, interpretando gli schemi inseriti dall'utente integri, ove necessario, il consueto formalismo con la chiusura transitiva.

1.5 Contenuto della tesi

Il secondo capitolo presenta il modello di dati ad oggetti complessi OLCD, con particolare attenzione alla definizione di oggetti complessi, di schema di base di dati, dell'operatore di intersezione, tramite il quale viene descritta l'ereditarietà, di istanze (possibili e legali) di uno schema e di regole di integritá. Successivamente viene definita la relazione di sussunzione, e una conseguente relazione di incoerenza. Infine viene presentata la definizione di espansione semantica di un tipo.

Il terzo capitolo é interamente dedicato all'analisi del software ODB-Tools prima della reingegnerizzazione operata. Vengono dati dapprima i concetti fondamentali sui quali si basa il funzionamento del programma, vi é quindi una rapida rassegna dei due moduli fondamentali del programma. Per concludere viene proposta una normale sessione di lavoro di ODB-Tools allo scopo di sintetizzare i concetti introdotti e di familiarizzare con le strutture utilizzate.

Il quarto capitolo tratta della espansione semantica di un tipo da un punto di vista prettamente teorico. Partendo dalla formulazione originaria dell'algoritmo di espansione verranno sottolineati i problemi che ha evidenziato. Saranno quindi presentate le nuove versioni dell'algoritmo confortate da una breve trattazione sulla teoria degli schemi ciclici. Chiudono il capitolo alcuni esempi notevoli di espansione semantica ricorsiva.

Il quinto capitolo riguarda la fattorizzazione di schemi di basi di dati. Dapprima vengono presentati i fondamenti teorici del problema, quindi si

passa alla trattazione dettagliata della scomposizione in fattori. Una parte fondamentale del capitolo riguarda l'individuazione e la trattazione dei fattori ciclici; in questo contesto si colloca l'introduzione del concetto di chiusura transitiva che viene trattato tanto da un punto di vista teorico quanto da un punto di vista implementativo.

Il sesto ed ultimo capitolo intende sintetizzare i concetti esposti nelle sezioni precedenti attraverso una descrizione dettagliata di una sessione di lavoro della versione modificata di ODB-Tools.

Verranno quindi trattate tutte le fasi dell'ottimizzazione semantica a partire dall'acquisizione dello schema passando per la generazione dello schema canonico e per la fattorizzazione per giungere all'espansione semantica delle interrogazioni.

Capitolo 2

OLCD - Modello di Dati ad Oggetti Complessi

OLCD é un'estensione del linguaggio ODL (*Object Description Language*) [1], introdotto in [BN94b] e rientra nella tradizione dei modelli di dati ad oggetti complessi [AK89, LR89]. OLCD, al pari del suo predecessore ODL, fornisce un *sistema di tipi base*: string, boolean, integer, real; inoltre i costruttori di tipi *tuple*, *set* e *class* consentono la costruzione di tipi valore complessi e di tipi classe. I tipi classe (detti brevemente *classi*) denotano insiemi di *oggetti con un'identitá ed un valore*, mentre i *tipi valore* denotano insiemi di *valori complessi con annidamento finito senza identitá di oggetti*. Puó essere utilizzato anche un operatore intersezione per creare intersezioni di tipi precedentemente introdotti consentendo l'ereditarietá semplice e multipla. Infine i tipi possono essere dei nomi. Il tipo nome si divide, a sua volta, in due tipi: un tipo *primitivo*, caso in cui l'utente deve specificare l'appartenenza di un *elemento* all'interpretazione del nome, oppure un tipo *virtuale*, nel qual caso la sua interpretazione viene calcolata.

Le estensioni di ODL introdotte in OLCD sono: *tipi cammino quantificati* e *regole di integritá*. La prima estensione é stata introdotta per poter utilizzare piú facilmente e in modo piú efficace le strutture annidate. I path, che sono essenzialmente sequenze di attributi, rappresentano la caratteristica fondamentale dei linguaggi di interrogazione O–O per navigare attraverso le gerarchie delle classi e dei tipi di uno schema [KKS92, BNPS92]. In particolare, secondo [BNPS92], i path *quantificati* permetto di navigare attraverso i tipi insieme. Le quantificazioni consentite sono la quantificazione esistenziale e la quantificazione universale che possono comparire piú di una volta all'interno di uno stesso path. Un tipo cammino é un tipo che associa un

[1] da non confondersi con ODL–ODMG93

cammino ad un altro tipo previsto dal formalismo. Perció, mediante il tipo cammino, come introdotto in [CW91], possiamo esprimere una classe di vincoli di integritá.

La seconda estensione permette di esprimere i vincoli di integritá come *regole if then* universalmente quantificati sugli elementi del dominio con un antecedente ed un conseguente che sono tipi del formalismo. Queste regole permettono di rappresentare gran parte della conoscenza in modo dichiarativo.

2.1 Sistema dei tipi atomici

Sia $\mathbf{D}$ l'insieme infinito numerabile dei valori atomici (che saranno indicati con $d_1, d_2, \dots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani. Non distingueremo fra i valori atomici e la loro codifica.

Sia $\mathbf{B}$ un insieme numerabile di designatori di tipi atomici (denotati da $B, B', \dots$) che contiene $\mathbf{D}$ (i.e., tutti i tipi mono–valore), e sia $\mathcal{I}_\mathbf{B}$ la funzione di interpretazione standard (fissata) da $\mathbf{B}$ a $2^\mathbf{D}$ tale che per ogni $d \in \mathbf{D}$: $\mathcal{I}_\mathbf{B}[d] = \{d\}$. Sia "$\sqcap$" un'operazione di congiunzione su $\mathbf{B}$ definita da:

$$B' \sqcap B'' = B \text{ sse } \mathcal{I}_\mathbf{B}[B'] \cap \mathcal{I}_\mathbf{B}[B''] = \mathcal{I}_\mathbf{B}[B].$$

Diciamo che $\mathbf{B}$ è un *sistema di tipi atomici* sse $\mathbf{B}$ è completo rispetto a $\sqcap$. Il tipo speciale che ha un'interpretazione vuota è detto *tipo vuoto* ed è indicato con $\bot$.[2] Un sistema di tipi atomici $\mathbf{B}$ è detto *PTIME* sse $B' \sqcap B'' = B$ può essere deciso in tempo polinomiale. In seguito assumiamo che un sistema di tipi atomici abbia questa proprietà.

A volte parleremo anche di sistemi di tipi atomici con una particolare semplice struttura, ovvero sistemi tali che per ogni sottoinsieme $\mathbf{X} \subseteq \mathbf{B}$ con $\bigsqcap \mathbf{X} = B$, ci sono due elementi $B', B'' \in \mathbf{X}$ tali che $B' \sqcap B'' = B$. Tale sistema di tipi atomici è detto *compatto binario*.

Consideriamo il seguente insieme di designatori di tipi atomici, che useremo in tutti gli esempi:

$$\mathbf{B} = \{\texttt{integer}, \texttt{string}, \texttt{bool}, \texttt{real}, i_1\text{–}j_1, i_2\text{–}j_2, \dots, d_1, d_2, \dots\},$$

dove gli i_k–j_k indicano tutti i possibili intervalli di interi e i d_k indicano tutti gli elementi di $\texttt{integer} \cup \texttt{string} \cup \texttt{bool}$ (i_k può essere $-\infty$ per denotare il minimo elemento di `integer` e j_k può essere $+\infty$ per denotare il massimo

[2] Questo tipo deve appartenere a $\mathbf{B}$ perchè la congiunzione di differenti tipi mono–valore è vuota.

elemento di `integer`). Assumendo l'interpretazione standard dei designatori di tipi atomici, $\mathbf{B}$ è ovviamente un sistema di tipi atomici compatto binario.

Sia $\mathcal{O}$ un insieme numerabile di *identificatori di oggetti*, (detti anche brevemente *oid*, e denotati da $o, o', \ldots$) disgiunto da $\mathbf{D}$.

Definizione 1 (Valori) *Dati gli insiemi $\mathcal{O}$ e $\mathbf{D}$, si definisce l'insieme $\mathcal{V}(\mathcal{O})$ dei* valori su $\mathcal{O}$ *(denotati da v, v') come segue (assumendo $p \geq 0$ e $a_i \neq a_j$ per $i \neq j$):*

v	$\rightarrow$	d	*valore atomico*
		o	*identificatore di oggetto*
		$\{v_1, \ldots, v_p\}$	*valore insieme*
		$[a_1 \colon v_1, \ldots, a_p \colon v_p]$	*valore tupla*

Definizione 2 (Dominio) *Dato un insieme di identificatori di oggetti $\mathcal{O}$, un* dominio δ *su $\mathcal{O}$ è una funzione totale da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$.*

Un dominio δ associa agli identificatori di oggetti un valore. In genere si dice che il valore $\delta(o)$ è lo *stato* dell'oggetto identificato dall'oid o. Un dominio verrà detto *finito* se l'insieme $\mathcal{O}$ è finito.

2.2 Oggetti complessi, tipi e classi

Sia $\mathbf{A}$ un insieme numerabile di *attributi* (denotati da $a_1, a_2, \ldots$) e $\mathbf{N}$ un insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che $\mathbf{A}$, $\mathbf{B}$, e $\mathbf{N}$ siano a due a due disgiunti.

Definizione 3 (Tipi) *Dati gli insiemi $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{N}$, il* sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ *denota l'insieme di tutte le* descrizioni dei tipi *(S, S', ...), detti anche brevemente* tipi*, su $\mathbf{A}, \mathbf{B}, \mathbf{N}$, che sono costruiti rispettando la seguente regola sintattica astratta (assumendo $a_i \neq a_j$ per $i \neq j$):*

S	$\rightarrow$	$\top$	
		B	*tipo atomico*
		N	*nome di tipo*
		$\forall\{S\}$	*tipo insieme*
		$\exists\{S\}$	*tipo esistenziale*
		$[a_1 \colon S_1, \ldots, a_k \colon S_k]$	*tipo tupla*
		$S \sqcap S'$	*intersezione*
		$\triangle S$	*tipo oggetto*
		$(p \colon S)$	*tipo cammino*

Un *cammino* p su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ é una sequenza di elementi $p = e_1 \,.\, e_2 \,.\, \ldots \,.\, e_n$, con $e_i \in \mathbf{A} \cup \{\forall, \exists, \triangle\}$.
Si denota con ϵ il *cammino vuoto*, e con $(p)*$ la *chiusura riflessiva e transitiva* di p.

Il tipo–cammino è (una notazione per) un tipo e quindi la sua estensione su un certo dominio è individuata dalla funzione interpretazione $\mathcal{I}$. Ad esempio, il tipo–cammino ($\triangle$.`name`: *"Silvano"*) individua tutti gli oggetti (il primo elemento del cammino è $\triangle$) che hanno un attributo `name` con un valore *"Silvano"*; in particolare questi oggetti possono appartenere ad una generica classe che ha l'attributo `name` definito come una stringa. Per considerare una determinata classe, ad esempio `employee`, il tipo–cammino deve essere congiunto con il nome della relativa classe: `employee` $\sqcap$ ($\triangle$.`name`: *"Silvano"*). Nello stesso modo, il tipo–cammino $S = (\triangle$.`managed-by`.$\triangle$.`salary`: $40 \div 60)$ non impone restrizioni sul dominio dell'attributo `managed-by`; se si considera la sua congiunzione con la classe `storage`, cioè `storage` $\sqcap\, S$, allora σ(`storage`) impone implicitamente che gli oggetti del dominio di `managed-by` appartengano alla classe `manager`. Inoltre, è possibile imporre esplicitamente una classe come dominio di un attributo nel seguente modo: $S' =$ ($\triangle$.`managed-by`: `tmanager` $\sqcap$ ($\triangle$.`salary`: $40 \div 60$)).

2.3 Schemi e regole di integrità

Definizione 4 (Schema di base di dati) *Dato un sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, uno schema di base di dati su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è una coppia $(\sigma, \mathbf{R})$, dove:*

- σ é una funzione totale da $\mathbf{N}$ a $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, che associa ai nome di tipi-classe e di tipi-valore la loro descrizione. σ é partizionata in due funzioni: $\sigma_{\mathtt{P}}$, che introduce la descrizione di *nomi di tipi primitivi* ($\mathbf{P}$) la cui estensione deve essere necessariamente fornita dall'utente; $\sigma_{\mathtt{V}}$, che introduce la descrizione di *nomi di tipi virtuali* ($\mathbf{V}$) la cui estensione puó essere invece calcolata ricorsivamente a partire dalle estensioni dei tipi che occorrono nella loro descrizione.
- $\mathbf{R}$ é un insieme di regole di integritá su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$

Definiamo formalmente la nozione di *regola di integrità*.

Definizione 5 (Regola di Integrità) *Dato un sistema di tipi $\mathbf{S}$, una* regola di integrità*, o più semplicemente* regola*, R su $\mathbf{S}$ è un elemento (S^a, S^c) del prodotto cartesiano $\mathbf{S} \times \mathbf{S}$.*

$$\sigma_P \begin{cases} \sigma_P(\texttt{department}) & = & \triangle[\ \texttt{dname: string}, \texttt{category: } 1 \div 7, \\ & & \quad \texttt{enrolls: } \forall\{\texttt{employee}\}, \texttt{administrator: employee}] \\ \sigma_P(\texttt{employee}) & = & \triangle[\ \texttt{name: string}, \texttt{qualification: } 1 \div 10, \texttt{salary: } 10 \div 100, \\ & & \quad \texttt{head: employee}, \texttt{worksin: department}] \\ \sigma_P(\texttt{CB\_Department}) & = & \texttt{department} \sqcap \triangle[\texttt{category: } 5 \div 7, \texttt{administrator: manager}] \\ \sigma_P(\texttt{manager}) & = & \texttt{employee} \sqcap \triangle[\ \texttt{qualification: } 8 \div 10, \\ & & \quad \texttt{head: manager}, \texttt{directs: department}] \end{cases}$$

$$\sigma_V \begin{cases} \sigma_V(\texttt{Mdl\_Employee}) & = & \texttt{employee} \sqcap \triangle[\texttt{qualification: } 5 \div 10] \\ \sigma_V(\texttt{Clerk}) & = & \texttt{employee} \sqcap \triangle[\texttt{qualification: } 7 \div 10, \texttt{worksin: Office}] \\ \sigma_V(\texttt{Office}) & = & \texttt{department} \sqcap \triangle[\texttt{enrolls: } \forall\{\texttt{Clerk}\}] \end{cases}$$

$$\mathbf{R} \begin{cases} \texttt{manager} \sqcap (\triangle.\texttt{qualification: } 9 \div \infty) \rightarrow \triangle[\texttt{directs: CB\_Department}] & \% \ \texttt{r\_1} \\ \texttt{department} \sqcap (\triangle.\texttt{enrolls}.\forall.\triangle.\texttt{qualification: } 9 \div \infty) \rightarrow \texttt{CB\_Department} & \% \ \texttt{r\_2} \\ \texttt{employee} \sqcap (\triangle.\texttt{salary: } 60 \div \infty) & \% \ \texttt{r\_3} \\ \qquad \sqcap (\triangle.\texttt{head}.\triangle.\texttt{qualification: } 8 \div \infty) \rightarrow \texttt{manager} \\ \texttt{manager} \sqcap (\triangle.\texttt{salary: } 50 \div \infty) & \% \ \texttt{r\_4} \\ \qquad \sqcap (\triangle.\texttt{directs}.\triangle.\texttt{category: } 5 \div \infty) \rightarrow (\triangle.\texttt{qualification: } 10 \div 10) \end{cases}$$

Tabella 2.1: Schema del dominio Compagnia nella sintassi OLCD

Informalmente, una regola di integrità $R = (S^a, S^c)$ ha lo scopo di vincolare ulteriormente l'istanza legale di uno schema, stabilendo una *relazione inclusione* tra il tipo S^a e il tipo S^c: per ogni valore v, se v è di tipo S^a ($v \in \mathcal{I}[S^a]$) allora v deve essere di tipo S^c ($v \in \mathcal{I}[S^c]$). Una regola di integrità R è quindi universalmente quantificata su tutti i valori $\mathcal{V}(\mathcal{O})$.

Nella regola di integrità $R = (S^a, S^c)$ i tipi S^a e S^c vengono chiamati rispettivamente *antecedente* e *conseguente* della regola e la regola verrà scritta anche nella usuale forma $R = S^a \rightarrow S^c$.

Ad esempio, lo schema con regole mostrato in tabella 2.1 descrive, in sintassi OLCD, lo schema di pagina 11

Sono consentiti nomi di tipi ciclici: la possibilitá di utilizzare un nome di tipo nella descrizione di un altro tipo puó far sorgere nello schema *dipendenze circolari*, cioé descrizioni di nome che fanno riferimento direttamente o indirettamente, tramite altri nomi, allo schema stesso. Parimenti sono consentite *regole cicliche* dal momento che un nome di tipo puó apparire tanto nell'antecedente quanto nel conseguente di una regola.
Formalmente, i nomi di tipi ciclici vengono definiti attraverso il concetto di *dipendenza*: N_1 *dipende da* N_2, scritto $N_1 \hookrightarrow N_2$, ove $N_1, N_2 \in \mathbf{N}$, se

- N_2 é contenuto nell'espressione che definisce N_1, $\sigma(N_1)$, oppure

- esiste una regola $R = (S^a, S^c)$ tale che N_1 é contenuta in S^a e N_2 é contenuta in S^c.

La chiusura transitiva $\hookrightarrow$ viene indicata con $\overset{+}{\hookrightarrow}$. Dunque $N \in \mathbf{N}$ é *ciclica*, se e solo se $N \overset{+}{\hookrightarrow} N$.

2.4 Istanza legale di uno schema

Nel seguito, scriveremo $\mathbf{S}$ in luogo di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ quando i componenti sono deducibili immediatamente dal contesto.

Sia $\mathcal{I}_\mathbf{B}$ la funzione interpretazione standard (fissata) da $\mathbf{B}$ a $2^\mathcal{D}$. Per un dato δ, ciascun tipo S é mappato in un insieme di valori (la sua interpretazione). Una *funzione interpretazione* é una funzione $\mathcal{I}$ da $\mathbf{S}$ a $2^\mathcal{V}$ che soddisfa le seguenti equazioni (dove $(p)^n$, $n \geq 0$, é definito da: $(p)^0 = \epsilon$, $(p)^1 = p$, $(p)^n = p.(p)^{n-1}$):

$$\begin{aligned}
\mathcal{I}[\top] &= \mathcal{V} \\
\mathcal{I}[\bot] &= \emptyset \\
\mathcal{I}[B] &= \mathcal{I}_\mathbf{B}[B] \\
\mathcal{I}[\forall\{S\}] &= \{M \mid M \subseteq \mathcal{I}[S]\} \\
\mathcal{I}[\exists\{S\}] &= \{M \mid M \cap \mathcal{I}[S] \neq \emptyset\} \\
\mathcal{I}\Big[[a_1 : S_1, \ldots, a_p : S_p]\Big] &= \Big\{[a_1 \colon v_1, \ldots, a_q \colon v_q] \Big| \begin{array}{r} p \leq q, v_i \in \mathcal{I}[S_i], 0 \leq i \leq p, \\ v_j \in \mathcal{V}(\mathcal{O}), p+1 \leq j \leq q \end{array} \Big\} \\
\mathcal{I}[S_1 \sqcap S_2] &= \mathcal{I}[S_1] \cap \mathcal{I}[S_2] \\
\mathcal{I}[\triangle S] &= \Big\{o \in \mathcal{O} \Big| \delta(o) \in \mathcal{I}[S]\Big\} \\
\mathcal{I}[(\epsilon \colon S)] &= \mathcal{I}[S] \\
\mathcal{I}[(a \colon S)] &= \mathcal{I}[[a \colon S]] \\
\mathcal{I}[(\triangle \colon S)] &= \mathcal{I}[\triangle S] \\
\mathcal{I}[(\forall \colon S)] &= \mathcal{I}[\forall\{S\}] \\
\mathcal{I}[(\exists \colon S)] &= \mathcal{I}[\exists\{S\}] \\
\mathcal{I}[(p.p' \colon S)] &= \mathcal{I}[(p \colon (p' \colon S))] \\
\mathcal{I}[((p)^* \colon S)] &= \bigcup_{n \geq 0} \mathcal{I}[((p)^n \colon S)]
\end{aligned}$$

L'interpretazione non soddisfa la relazione d'inclusione tra interpretazioni di tipi stabilita mediante regole e non é sufficiente ad assicurare che valori ed

oggetti siano legali *legali*, cioé che i tipi nome soddifino la loro descrizione. Per questo motivo é necessario fornite una definzione piú stringente.

Definizione 6 (Istanza possibile) *Dato uno schema di base di dati* $\Sigma = (\sigma, \mathbf{R})$ *definito su* $\mathbf{S}$*, un insieme di oid* $\mathcal{O}$*, e un dominio* δ *du* $\mathcal{O}$*, un'interpretazione* $\mathcal{I}$ *di* $\mathbf{S}$ *su* δ *é una* istanza possibile *di* Σ *se e solo se* $\mathcal{O}$ *é* finito *e*

1. $\mathcal{I}[N] \subseteq \mathcal{I}[\sigma_P(N)]$*, if* $N \in$ *dom* σ_P*.*
2. $\mathcal{I}[N] = \mathcal{I}[\sigma_V(N)]$*, if* $N \in dom\sigma_V$*.*
3. $\mathcal{I}[S^a] \subseteq \mathcal{I}[S^c]$*, if* $R = (S^a, S^c) \in \mathbf{R}$*;*

Dalla definizione data si evince che l'interpretazione di un nome di tipo primitivo *é inclusa* nell'interpretazione della sua descrizione, mentre l'interpretazione di un tipo virtuale *é* l'interpretazione della sua descrizione. In altre parole, l'interpretazione di un nome di tipo primitivo deve essere fornita dall'utente in base alla descrizione data, mentre l'interpretazione di un nome di tipo virtuale viene ricavata a partire dalla sua descrizione e dall'interpretazione dei nomi di tipi primitivi come accade per le viste all'interno dei database.

Di conseguenza, la regola $N \longrightarrow S$ é equivalente a $\sigma_P(N) = S$; le regole $N \longrightarrow S$ e $S \longrightarrow N$ sono equivalenti a $\sigma_V(N) = S$.

Sembrerebbe dunque, per quanto affermato, di poter definire uno schema di base di dati $\Sigma = (\sigma, \mathbf{R})$ come un insieme di regole. Tuttavia le funzioni σ_P e σ_V vengono comunque introdotte per due importanti ordini di motivi:

1. per ragioni di comoditá - σ_P fornisce una visione sintetica di una classe mentre σ_V dá una vista sintetica di una classe virtuale e, quindi, anche di una query;
2. per ragioni semantiche - é necessario definire univocamente l'estensione di una classe virtuale ciclica (quindi anche di una query ricorsiva).

Soffermiamoci sulla seconda motivazione. É possibile assegnare differenti interpretazioni ad un nome virtuale ciclico N, a seconda che questo venga inserito nello schema attraverso la definizione $\sigma_V(N) = S$ oppure attraverso le regole $N \longrightarrow S$ e $S \longrightarrow N$.

Si possono osservare le differenze che riguardano l'introduzione di un'istanza possibile di uno schema. Si considerino gli oggetti di tabella 2.2 e le seguenti istanze delle classi `employee` e `department`: $\mathcal{I}$[`employee`] = { o_1, o_3, o_5, o_7, o_9 } $\mathcal{I}$[`department`] = { o_2, o_4, o_6, o_8 } L'istanza della classe virtuale aciclica `Mdl_Employee` puó essere calcolate esclusivamente come $\mathcal{I}$[`Mdl_Employee`] = $\{o_3, o_5, o_7, o_9\}$. D'altro canto é possibile avere svariate istanze legali per la classe virtuale ciclica `Clerk`: $\mathcal{I}$[`Clerk`] = $\{o_5, o_7, o_9\}$, o $\mathcal{I}$[`Clerk`] = $\{o_5, o_9\}$, o $\mathcal{I}$[`Clerk`] = $\{o_9\}$ sia introducendo `Mdl_Employee` e `Clerk`, sia mediante due regole sia con una definizione σ_V.

$\mathcal{O}$	=	$\{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9\}$
$\delta(o_1)$	=	[`name:` *"Mark"*, `qualification:` 3, `salary:` 32, `head:` o_3, `worksin:` o_2]
$\delta(o_3)$	=	[`name:` *"Robert"*, `qualification:` 6, `salary:` 43, `head:` o_3, `worksin:` o_2]
$\delta(o_5)$	=	[`name:` *"Andy"*, `qualification:` 7, `salary:` 67, `head:` o_7, `worksin:` o_4]
$\delta(o_7)$	=	[`name:` *"Peter"*, `qualification:` 8, `salary:` 76, `head:` o_9, `worksin:` o_6]
$\delta(o_9)$	=	[`name:` *"Franz"*, `qualification:` 9, `salary:` 80, `head:` o_9, `worksin:` o_8]
$\delta(o_2)$	=	[`dname:` *"Administration"*, `category` : 5, `enrolls:` $\{o_1, o_3\}$, `administrator:` o_3]
$\delta(o_4)$	=	[`dname:` *"Development"*, `category:` 6, `enrolls:` $\{o_5\}$, `administrator:` o_5]
$\delta(o_6)$	=	[`dname:` *"Research"*, `category:` 7, `enrolls:` $\{o_7\}$, `administrator:` o_9]
$\delta(o_8)$	=	[`dname:` *"Project"*, `category:` 7, `enrolls` : {}, `administrator:` o_9]

Tabella 2.2: Istanza possibile del Dominio Compagnia

Tuttavia, al fine di definire univocamente l'estensione di un nome virtuale ciclico N, occorre adottare una semantica *fixed point* (semantica del punto fisso): nella fattispecie occorrerebbe una semantica lfp o gfp per *essere in grado di esprimere le definizioni* attraverso σ_V. Infatti una semantica fixed point applica soltanto a punti fissi espressioni come $\sigma_V(N) = S$ dove S é una "funzione" di N, cioé N compare in S. Al contrario una semantica descrittiva interpreta le dichiarazioni solo come restrizioni dell'insieme dei modelli possibili, senza significato di definizione , d'altro canto puó essere facilmente estesa alle regole $N \longrightarrow S$ e $S \longrightarrow N$. Nell'esempio precedente,quindi, se la classe `Clerk` é definita con le regole si avrebbe: $\mathcal{I}$[`Clerk`] = $\{o_5, o_7, o_9\}$, or $\mathcal{I}$[`Clerk`] = $\{o_5, o_9\}$, or $\mathcal{I}$[`Clerk`] = $\{o_9\}$.

Una semantica gfp rappresenta una buona scelta per modellare la chiusura transitiva di un attributo, di conseguenza opteremo per quest'ultima. Con riferimento all'esempio precedente avremo, con la semantica gfp, $\mathcal{I}$[`Clerk`] = $\{o_5, o_7, o_9\}$.

$\mathcal{I}[(\delta_\sigma \triangle.\texttt{directs}.\triangle.\texttt{employs}.\forall.\triangle.\texttt{qualification}\colon 7 \div 10)] = \{o_5, o_7, o_9\}$

Introduciamo ora la nozione di *istanza legale* di uno schema la quale permetta di interpretare i nomi virtuali ciclici definiti mediante la funzione σ_V utilizzando una semantica gfp e nomi virtuali ciclici definiti in $\mathbf{R}$ come una semantica descrittiva.

Dato $\Sigma = (\sigma, \mathbf{R})$ definito su $\mathbf{S}$, sia $\Psi_{\mathcal{I}}$ l'*insieme delle istanza legali con uguale* $\mathcal{O}$ *e* δ tele che per ogni $\mathcal{I}, \mathcal{I}' \in \Psi_{\mathcal{I}}$: $\mathcal{I}[N] = \mathcal{I}'[N]$ if $N \in \mathbf{N}/dom\sigma_V$. Inoltre, sia "$\overset{\Psi}{\sqsubseteq}$" una relazione su $\Psi_{\mathcal{I}}$ tale che per ogni $\mathcal{I}, \mathcal{I}' \in \Psi_{\mathcal{I}}$:

$$\mathcal{I} \overset{\Psi}{\sqsubseteq} \mathcal{I}' \quad \text{sse} \quad \mathcal{I}[N] \subseteq \mathcal{I}'[N] \text{ per ogni } N \in dom\sigma_V.$$

Allora $(\Psi_{\mathcal{I}}, \overset{\Psi}{\sqsubseteq})$ costituisce un ordine parziale. Diremo che $\mathcal{I}$ é un'*istanza legale* di uno schema di base di dati Σ se e solo se é l'unica *greatest* istanza dell'insieme $\Psi_{\mathcal{I}}$ w.r.t. $\overset{\Psi}{\sqsubseteq}$.

Teorema 1 *Se* $\mathcal{I}$ *é un'istanza legale, allora esiste un'istanza legale* $\mathcal{I}'$ *tale che* $\mathcal{I} \overset{\Psi}{\sqsubseteq} \mathcal{I}'$.

2.5 Sussunzione e coerenza

In questa sezione definiamo una relazione di inclusione semantica, detta *relazione di sussunzione*, tra i tipi in uno schema, indicata con il simbolo $\sqsubseteq$.

Definizione 7 (Sussunzione) *Dato uno schema* σ *su* $\mathbf{S}$ *e due tipi* $S, S' \in \mathbf{S}$*, si definisce la relazione di sussunzione* $\sqsubseteq_\sigma$ *come segue:*

$$S \sqsubseteq_\sigma S' \quad \textit{sse} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \quad \textit{per ogni istanza } \mathcal{I} \textit{ di } \sigma;$$

La relazione $\sqsubseteq_\sigma$ è un preordine (cioè transitivo e riflessivo ma antisimmetrico) che induce una relazione di *equivalenza* $\simeq$ sui tipi: $S \simeq S'$ sse $S \sqsubseteq_\sigma S'$ e $S' \sqsubseteq_\sigma S$. La relazione di equivalenza $\simeq$ permette di definire i tipi S che hanno, nello schema σ, un'interpretazione sempre vuota.

Definizione 8 (Incoerenza) *Dato uno schema* σ *su* $\mathbf{S}$*, un tipo* $S \in \mathbf{S}$ *è detto* incoerente *nello schema* σ *sse* $S \simeq \bot$.

Uno schema σ è detto *coerente* sse per ogni $N \in \mathbf{N}$, $N \not\simeq \bot$. Si noti che in uno schema coerente vi possono essere dei tipi incoerenti usati nei tipi set e sequence: infatti i tipi $\{\bot\}$ e $\langle\bot\rangle$ sono coerenti e denotano rispettivamente l'insieme vuoto e la sequenza vuota.

Mediante la relazione di sussunzione si individuano, per ogni nome N, tutte le sue *generalizzazioni* rispetto all'intera tassonomia, dalle quali è possibile selezionarne le *più specifiche*. Formalmente, dato uno schema σ, per un nome $N \in \mathbf{N}$ si definisce l'insieme $GS(N)$ (*Generalizations Set*):

$$GS(N) = \{N' \in \mathbf{N} \mid N' \neq N \wedge N \sqsubseteq_\sigma N'\}$$

e l'insieme $MSGS(N)$ (*Most Specialized Generalizations Set*)

$$MSGS(N) = \{N' \in GS(N) \mid \nexists N'' \in GS(N)\colon N'' \sqsubseteq_\sigma N'\}$$

Infine, in uno schema si può introdurre la *classe universale*, denotata con $\top_C$ $(top-class)$, che rappresenta la classe *più generale* dello schema e che quindi sussume tutte le altre classi, come classe virtuale con descrizione $\sigma_{\mathrm{V}}(\top_C) = \triangle\top$. Infatti è immediato verificare che, in qualsiasi istanza possibile $\mathcal{I}$, si ha sempre $\mathcal{I}[\top_C] = \mathcal{O}$.

2.6 Espansione semantica di un tipo

L'espansione semantica di un tipo permette di incorporare ogni possibile restrizione che non è presente nel tipo originale ma che è *logicamente implicata* dal tipo e dallo schema. Formalmente questo viene espresso tramite la seguente definizione di espansione semantica:

Definizione 9 (Espansione Semantica) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, e un tipo* $S \in \mathbf{S}$*; l'*espansione semantica *di* S *rispetto a* $(\sigma, \mathbf{R})$*,* $EXP(S)$*, è un tipo di* $\mathbf{S}$ *tale che:*

1. $EXP(S) \simeq_{\mathrm{R}} S$*;*
2. *per ogni* $S' \in \mathbf{S}$ *tale che* $S' \simeq_{\mathrm{R}} S$ *si ha che* $EXP(S) \sqsubseteq_\sigma S'$.

$EXP(S)$ è il tipo più specializzato tra tutti i tipi $\simeq_{\mathrm{R}}$–equivalenti al tipo S in quanto include tutte le possibili restrizioni implicate dalle regole $\mathbf{R}$. In questo modo è il tipo più piccolo rispetto alla relazione $\sqsubseteq_\sigma$ tra tutti i tipi $\simeq_{\mathrm{R}}$–equivalenti a S. Si noti che $EXP(S)$ individua una classe di tipi $\simeq$–equivalenti, nella quale ogni elemento è un tipo $\simeq_{\mathrm{R}}$–equivalente al tipo S.

Il metodo proposto per determinare l'espansione semantica è caratterizzato dai seguenti punti:

- iterazione della trasformazione "se un tipo implica l'antecedente di una regola allora il conseguente di tale regola può essere ad esso congiunto";
- valutazione delle implicazioni logiche tramite il calcolo della sussunzione tra tipi.

Allo scopo di individuare tutte le trasformazioni che uno schema con regole $(\sigma, \mathbf{R})$ induce su un tipo, si introduce la funzione totale $, \ : \ \mathbf{S} \longrightarrow \mathbf{S}$ tale che

$$, (S) \ = \ \begin{cases} S \sqcap \bigsqcap_k (p_k : S_k^c) & \forall R_k, p_k : \ S \sqsubseteq_\sigma (p_k : S_k^a), \ S \not\sqsubseteq_\sigma (p_k : S_k^c) \\ S & \text{altrimenti} \end{cases}$$

e si definisce $\tilde{,} = , ^{\bar{i}}$, dove $\bar{i}$ è il più piccolo intero tale che $, ^{\bar{i}} = , ^{\bar{i}+1}$.

Proposizione 1 *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, per ogni* $S, S' \in \mathbf{S}$*, si ha che* $S \sqsubseteq_\Sigma S'$ *se e solo se* $\tilde{,}(S) \sqsubseteq_\sigma S'$*.*

Pertanto, il calcolo della sussunzione in uno schema con regole $(\sigma, \mathbf{R})$ può essere effettuato prima determinando l'espansione semantica di un tipo e quindi calcolando la sussunzione in σ.

Capitolo 3

Analisi di ODB-Tools

Una gran parte del presente lavoro é stata indubbiamente assorbita dall'analisi dell'intero programma ODB-Tools e dalla realizzazione degli interventi indispensabili per consentire al software la trattazione delle interrogazioni e degli schemi ciclici.

Come complemento alle note teoriche proposte nei capitoli precedenti, ci sembra opportuno riportare i diversi moduli che, nella versione originaria, implementavano gli algoritmi introdotti.

Per la parte di acquisizione dello schema si é fatto riferimento al modello di [Gar95], mentre, per la parte di ottimizzazione vera e propria sono stati considerati gli schemi in [Vin94] e in [App96].

Questo capitolo é dunque interamente dedicato alla presentazione della struttura dei moduli di ODB-Tools così come si presentavano prima delle modifiche introdotte. A conclusione di questa ampia sezione introduttiva verrá proposto un esempio del funzionamento del software presentato.

3.1 Aspetti generali

Il formalismo ODL è stato proposto in [BN94a] ed estende l'espressività di linguaggi di logica descrittiva [WS92] al fine di rappresentare la semantica dei modelli di dati ad oggetti complessi (*CODMs*) recentemente proposti in ambito basi di dati deduttive [AK89] e basi di dati orientate agli oggetti [LR89, Atz93].

In [BBG+96] ODL è stato esteso per permettere la formulazione dichiarativa di un insieme rilevante di vincoli d'integrità sulla base di dati. L'estensione di ODL con vincoli è stata denominata OCDL (*Object Constraint Description Logics*). Gli schemi di basi di dati reali sono generalmente forniti in termini di classi base mentre l'ulteriore conoscenza è espressa attraverso

vincoli d'integrità che garantiscono la consistenza dei dati. In particolare, le estensioni riguardano i *tipi path quantificati* e *regole d'integrità*. I path, che sono essenzialmente sequenze di attributi, rappresentano l'elemento centrale dei linguaggi d'interrogazione OODB per navigare attraverso le gerarchie di aggregazione di classi e tipi di uno schema. È possibile esprimere path *quantificati* per navigare attraverso gli attributi multivalore: le quantificazioni ammesse sono quella esistenziale e quella universale e possono comparire più di una volta nello stesso path. Le regole di integrità permettono la formulazione dichiarativa di un insieme rilevante di vincoli d'integrità sotto forma di regole *if then* i cui antecedenti e conseguenti sono espressioni di tipo ODL. È possibile, in tal modo, esprimere correlazioni fra proprietà strutturali della stessa classe o condizioni sufficienti per il popolamento di sottoclassi di una classe data.

In [BN94a] è stato presentato il sistema OCDL-Designer, per l'acquisizione e la validazione di schemi OODB che preserva la consistenza della tassonomia ed effettua inferenze tassonomiche. In particolare, il sistema prevede un algoritmo di *sussunzione*, che determina tutte le relazioni di *specializzazioni* tra tipi, e un algoritmo per rilevare tipi *inconsistenti*, cioè tipi necessariamente vuoti. In [BBG+96] l'ambiente teorico sviluppato in [BN94a] è stato esteso per effettuare l'ottimizzazione semantica delle interrogazioni, dando vita al sistema ODB-QOptimizer.

La nozione di ottimizzazione semantica di query è stata introdotta per le basi di dati relazionali da King [Kin81a, Kin81b] e da Hammer e Zdonik [HZ80]. L'idea di base di queste proposte è che i vincoli di integrità, espressi per forzare la consistenza di una base di dati, possono essere utilizzati anche per ottimizzare le interrogazioni fatte dall'utente, trasformando un'interrogazione in una *equivalente*, ovvero con la stessa risposta, ma che può essere elaborata in modo più efficiente.

Un insieme rilevante di query per OODB [Kim89] può essere espresso come un tipo *virtuale*, vista la ricchezza del formalismo ODL. Tuttavia, poiché i linguaggi di interrogazione sono più espressivi del nostro formalismo, introduciamo, seguendo l'approccio proposto in [BJNS94], una separazione ideale di un'interrogazione in una parte *clean*, che può essere rappresentata come tipo in ODL, e una parte *dirty*, che va oltre l'espressività del sistema di tipi; l'ottimizzazione semantica sarà effettuata solo sulla parte *clean*.

Sia il processo di controllo di consistenza e classificazione delle classi dello schema che quello di ottimizzazione semantica delle interrogazioni sono basati sulla nozione di *espansione semantica* di un tipo. L'espansione semantica permette di incorporare ogni possibile restrizione che non è presente nel tipo originale ma che è *logicamente implicata* dallo schema globale (classi + tipi + regole d'integrità). L'espansione dei tipi è basata sull'iterazione di questa tra-

sformazione: se un tipo implica l'antecedente di una regola d'integrità allora il conseguente di quella regola può essere aggiunto alla descrizione del tipo stesso. Le implicazioni logiche fra i tipi (il tipo da espandere e l'antecedente di una regola) sono determinate attraverso la relazione di sussunzione. La relazione di sussunzione è simile alla relazione di *raffinamento di tipi* (*subtyping relation* definita in [Car84] ed adottata negli OODBs [LR89]).

Il calcolo dell'espansione semantica di una classe permette di rilevare nuove relazioni *isa*, cioè relazioni che non sono esplicitate dal progettista ma che sono logicamente implicate dalla descrizione della classe e dello schema globale. In questo modo una classe può essere classificata automaticamente all'interno di una gerarchia di ereditarietà. La presenza delle regole di integrità rende questa classificazione significativa anche rispetto alle sole classi base, cioè con le regole si possono determinare nuove relazioni *isa* anche tra due classi base.

Per quanto riguarda l'ottimizzazione, seguendo l'approccio di [SO89], viene eseguita a run time l'espansione semantica di una interrogazione in modo da ottenere l'*interrogazione più specializzata* fra tutte quelle semanticamente equivalenti a quella iniziale. In questo modo l'interrogazione viene spostata verso il basso nella gerarchia delle classi e le classi presenti nell'interrogazione vengono sostituite con classi più specializzate; questo costituisce un'effettiva ottimizzazione dell'interrogazione, indipendente da ogni modello di costo, in quanto riduce l'insieme di oggetti da controllare per individuarne il risultato.

3.2 OCDL-Designer: architettura e funzionalitá

OCDL-Designer implementa l'acquisizione e la modifica dello schema con regole.
In particolare, consente di:

- verificare che lo schema sia:
  - *consistente*: esiste almeno uno stato del DB tale che ogni tipo, classe e regola abbia un'estensione non vuota
- ottenere la *minimalitá* dello schema rispetto alla relazione isa, cioé per ogni tipo (classe) viene calcolata la giusta posizione nella tassonomia dei tipi (classi)
  - il tipo (classe) viene inserito sotto tutti i tipi (classi) che specializza;
  - il tipo (classe) viene inserito sopra tutti i tipi (classi) che lo specializzano.

OCDL-Designer è stato realizzato in ambiente di programmazione C, versione standard ANSI C su piattaforma hardware SUN SPARCstation 20, sistema operativo SOLARIS 2.3.

In figura 3.1 riportiamo l'architettura funzionale di OCDL-Designer. Come si può vedere il programma è diviso in due sottocomponenti funzionali principali che corrispondono a due fasi distinte : il primo, denotato con `OCDL-COHE`, è quello che permette il controllo della coerenza dello schema generando la forma canonica dello schema.

Il secondo componente, denotato con `OCDL-SUBS`, è quello che, partendo dallo schema OCDL canonico, calcola le relazioni di sussunzione e le relazioni isa minimali che intercorrono fra i tipi (classi).

3.3 OCDL-Designer: Programma principale

Il programma acquisisce schemi di basi di dati ad oggetti complessi espressi nel linguaggio `OCDL`, opera la trasformazione in forma canonica al fine di controllare la consistenza dello schema e calcola le relazioni isa eventualmente implicite nelle descrizioni.

Il programma prende in ingresso un file di testo `nomefile.sc` contenente lo schema iniziale e, durante le fasi successive, comunica a video eventuali messaggi di errori e incoerenze rilevate; se l'esecuzione ha termine correttamente i risultati dell'elaborazione vengono scritti in due file: `nomefile.fc` e `nomefile.sb`. Il primo contiene i risultati della trasformazione canonica, il secondo le relazioni di sussunzione e le relazioni `isa` minimali calcolate.

La fase di acquisizione consiste nella lettura del file contenente lo schema e la creazione delle relative strutture dinamiche rappresentanti le definizioni dei tipi (classi). Durante questa fase non viene realizzato un controllo sintattico e semantico sistematico, ma solo alcuni controlli di comodo per rilevare eventuali errori nella scrittura del file di input. Si assume infatti che lo schema preso in input sia corretto dal punto di vista sintattico e semantico.

Una volta acquisito lo schema ha inizio la fase di generazione dello schema canonico mediante l'applicazione delle funzioni ricorsive ι e μ, definite nella teoria. La generazione di tale schema permette inoltre di determinare quali sono i tipi (classi) incoerenti, quelli, cioè, la cui descrizione è inconsistente e che, di conseguenza, avranno sempre estensione vuota.

Determinata la forma canonica si passa all'esecuzione dell'algoritmo di sussunzione che permette di ricalcolare tutte le relazioni `isa` riuscendo a

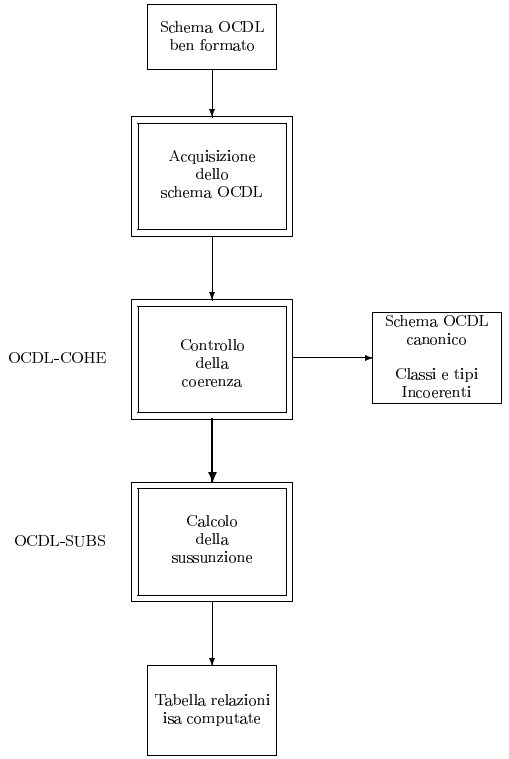


Figura 3.1: Architettura funzionale di OCDL-Designer

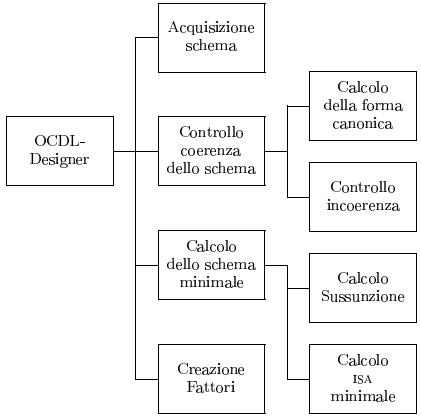


Figura 3.2: Struttura del programma OCDL-Designer

rilevare anche le relazioni implicite nella descrizione originale di tipi e classi e determinando i tipi e le classi equivalenti. Inoltre vengono calcolate le relazioni di sussunzione esistenti tra antecedenti e conseguenti di regole.

Il programma si compone di quattro moduli richiamati dal programma principale:

Acquisizione schema
Controllo coerenza dello schema
Calcolo dello schema minimale
Fattorizzazione dello schema canonico

Il modulo indicato in figura 3.2 col termine "*Creazione fattori*" realizza la fattorizzazione delle classi definite dall'utente e portate dal programma in forma canonica.
ODB-QOptimizer sviluppa un procedimento analogo (vedi figura 3.6) per ogni classe virtuale ottenuta dai vari passi dell'espansione semantica della query inserita dall'utente.

E' bene osservare che, in buona sostanza, l'algoritmo di fattorizzazione non introduce nessuna informazione ulteriore, da un punto di vista semantico, rispetto a quanto riportato nella forma canonica dello schema con regole ovvero della query ottimizzata. Cionondimeno i fattori di una classe rappresentano un formalismo piú intuitivo ed immediato per esprimerne la descrizione in forma canonica.

Anche da un punto di vista strettamente operativo, il ricorso alla fattorizzazione dei risultati finisce per costituire un punto di partenza privilegiato per la visualizzazione dei risultati dell'ottimizzazione. Avendo a disposizione la forma fattorizzata della query ottimizzata infatti, un apposito modulo puó facilmente determinare attraverso un confronto con i fattori OCDL dello schema, quali riportare in output e quali invece ignorare. Dunque, con riferimento allo schema di figura 3.3, si puó osservare che la query finale, indicata con Q', é costituita dalla traduzione da OCDL a OQL dei fattori da visualizzare.

3.4 ODB-QOptimizer: architettura e funzionalitá

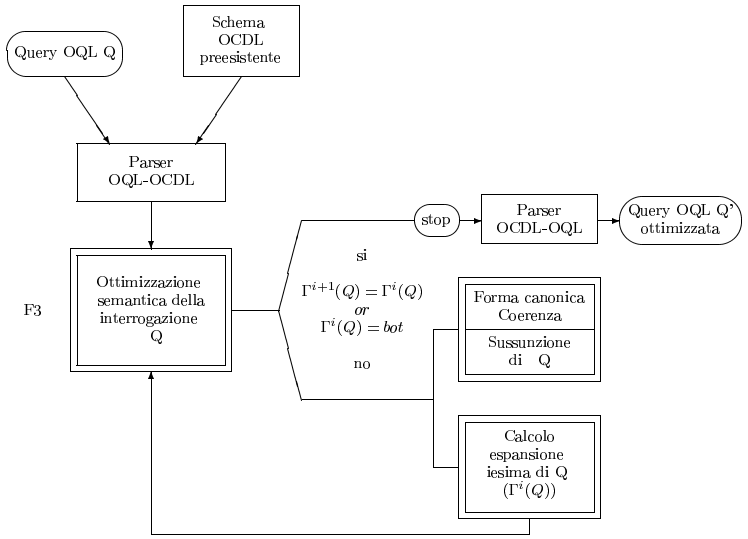


Figura 3.3: Architettura funzionale di ODB-QOptimizer

ODB-QOptimizer é un software complesso che, partendo da una generica interrogazione espressa in linguaggio ODL, ne calcola l'espansione semantica

in forma canonica. Tale descrizione, come specificato nel capitolo precedente, é la piú specifica tra quelle semanticamente equivalenti, contenendo le restrizioni fornite dai vincoli di integritá specificati nello schema ODL.

Naturalmente ha senso parlare di interrogazioni di una base di dati solamente quando esiste lo schema che descrive la realtá da rappresentare e l'istanza della base di dati. D'altra parte, poiché l'ottimizzazione viene effettuata a livello intensionale, un requisito indispensabile é l'esistenza dello schema con regole.

In figura 3.3 é illustrata l'architettura funzionale di ODB-QOptimizer. La funzione F3 calcola l'ottimizzazione semantica dell'interrogazione attraverso l'iterazione delle operazioni necessarie fino al raggiungimento della condizione di terminazione: essa coincide con il punto fisso della funzione di espansione semantica in cui il risultato dell'ottimizzazione è quello definitivo, oppure con il rilevamento di incoerenza dell'interrogazione via via trasformata.

Il primo blocco di operazioni specificate dalla funzione F3 di figura 3.3 (cioè il calcolo dello schema canonico, di incoerenza e sussunzione di una query Q) viene eseguito dal componente OCDL-D mentre la parte riguardante l'espansione semantica viene realizzata dal componente GES.

La funzione F3 si arresta quando si raggiunge il punto fisso di , (Q) (quando cioè non si possono più applicare regole), oppure quando si rileva l'incoerenza dell'interrogazione.

Il controllo sulle condizioni di terminazione della funzione F1 è ottenuto controllando la costante di ritorno di GES:

- `REGOLA_APPLICATA`: significa che non abbiamo ancora terminato e che è nuovamente necessario iterare la funzione F3, calcolando il nuovo schema canonico, la classificazione e chiamando nuovamente GES.
- `NESSUNA_REGOLA_APPLICATA`: significa che abbiamo terminato l'ottimizzazione correttamente e l'interrogazione ottimizzata è presente in forma canonica.
- `QUERY_INCOERENTE`: significa che abbiamo terminato l'ottimizzazione avendo riscontrato l'incoerenza dell'interrogazione.

Poichè sono da escludere errori sintattici, al termine dell'ottimizzazione abbiamo quindi disponibile in memoria l'espansione semantica dell'interrogazione in forma canonica, oppure sappiamo che l'interrogazione è risultata incoerente.

Il prototipo ODB-QOptimizer è stato sviluppato in linguaggio C, versione standard ANSI C. La piattaforma hardware utilizzata è una SUN SPARCstation 20, sistema operativo SOLARIS 2.3.

3.5 Architettura e funzionalitá di GES

GES é un componente software che realizza l'espansione semantica di un tipo sviluppato in linguaggio C, versione standard ANSI C.

Il programma si compone essenzialmente delle due funzioni denotate in figura 3.4 come F1 ed F2.

F1 legge la descrizione della query e dei tipi che la compongono ad ogni livello di innestamento, gli insiemi GS relativi a tali tipi (classi) ed i tipi che definiscono i conseguenti di ogni regola (ad ogni livello di innestamento); controlla la correttezza sintattica di ciascuna descrizione.

Se l'interrogazione non è incoerente, si prosegue eseguendo F2, che ne calcola l'espansione semantica in base all'algoritmo di Espansione Semantica di un Tipo: per ciascun tipo S che compare nella descrizione dell'interrogazione ad ogni livello di innestamento viene controllata la condizione di applicabilità delle regole. Ora tutte le regole applicabili, vengono applicate attraverso la congiunzione con il rispettivo conseguente.

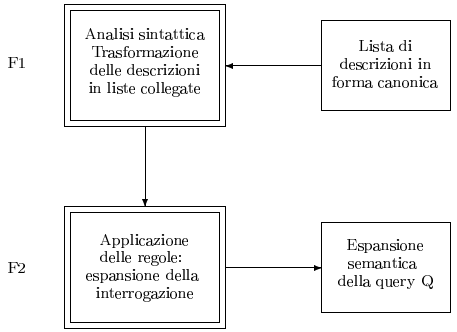


Figura 3.4: Architettura funzionale di GES

3.6 ODB-QOptimizer: Programma principale

Il programma prende in ingresso un file di testo contenente lo schema ODL iniziale, lo schema in forma canonica e le relazioni di sussunzione tra i tipi e

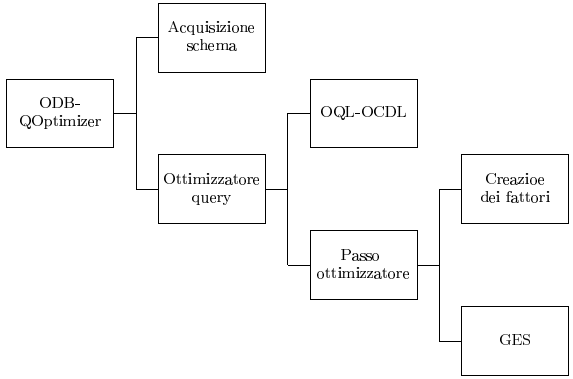


Figura 3.5: Struttura del programma ODB-QOptimizer

classi dello schema.

Una volta acquisito lo schema preesistente, il programma prende in ingresso il file contenente la query `nomequery` scritta in formato OQL.

Durante le fasi successive comunica a video eventuali messaggi di errore e incoerenze rilevate; se l'esecuzione ha termine correttamente i risultati dell'elaborazione vengono scritti in due file: `nomequery.oql.fc` e `nomequery.oql.sb`. Il primo contiene i risultati dell'ottimizzazione, ossia la stampa della query e della sua forma canonica nei vari passi di ottimizzazione. Il secondo le relazioni di sussunzione e le relazioni `isa` minimali.

La fase di acquisizione consiste nella lettura del file contenente lo schema preesistente, e la creazione delle relative strutture dinamiche rappresentanti lo schema. Una volta acquisito lo schema ha inizio la fase di ottimizzazione della query: prima di tutto viene chiamata l'interfaccia OQL per l'acquisizione della query e la relativa trasformazione in classe virtuale. Dopo di che parte la fase vera e propria dell'ottimizzazione, essa consiste in ciclo che termina quando viene raggiunto il punto fisso dell'espansione semantica della query.

Ad ogni passo vengono chiamati in successione i moduli che calcolano la forma canonica della query e la sussunzione, e il componente GES che realizza l'espansione semantica di un tipo.

Se durante l'esecuzione vengono rilevate delle incoerenze o degli errori nel procedimento il programma termina dopo aver visualizzato un messaggio.

3.7 Sessione di lavoro

Allo scopo di sintetizzare i concetti esposti e per rendere piú immediata la comprensione delle modifiche ai programmi (che saranno presentate nei prossimi capitoli) viene proposto un semplice esempio del funzionamento del software.

Nella fattispecie si introdurrá dapprima uno schema con regole in OCDL e saranno analizzati gli output generati nella fase di designer, quindi verranno proposti un'interrogazione OQL e, di seguito, i risultati dell'ottimizzazione.

```
prim Storage =  ^ [ name : string , category : string ,
                  managed_by : Manager ] ;
prim TManager =  Manager & ^ [ level : range 8 12 ] ;
prim Manager =  ^ [ name : string , salary : range 40000 100000 ,
                    level : range 1 15 ] ;
antev rule1a = Storage & ^ [ category : vstring "B4" ] ;
consv rule1c = Storage & ^ [ managed_by : TManager ] ;
```

Figura 3.6: Schema con regole OCDL

Lanciando il designer per lo schema di figura 3.6 otteniamo in output tutti i file descritti in precedenza. In particolare si vuole analizzare la forma fattorizzata in rapporto alla forma canonica dello schema.

In figura 3.7 sono riportate, in forma canonica, le classi dello schema inserito dall'utente.

Per comoditá sono state omesse le righe che riguardano la descrizione dei nuovi tipi inseriti dal programma per ottenere tutte le descrizioni dello schema canonico.

Come giá anticipato, la creazione dei fattori parte dal risultato di figura 3.7 trasformandolo per arrivare al formalismo riportato in figura 3.8.

Si puó facilmente osservare che la forma fattorizzata reca, per ciascuna classe dello schema iniziale, un fissato numero di fattori (uno per riga). Nel loro insieme, detti fattori, costituiscono la descrizione della classe mutuata dalla schema canonico.

Il generico fattore sará costituito, in generale, dai seguenti componenti:

- Path
- Tipo di dominio del fattore
- Tipo di fattore
- Nome del dominio


```
prim Storage 0 =  ^ [ category : string , managed_by : Manager ,
                    name : string ] ;
--------> -Storage  & ^ NewType1 ;
gs Storage = Storage : prim 0  1 ;
prim TManager 0 =  Manager  & ^ [ level : range 8 12 ] ;
--------> -TManager  & -Manager  & ^ NewType2 ;
gs TManager = TManager : prim 0  1 , Manager : prim 1  0 ;
prim Manager 0 =  ^ [ level : range 1 15 , name : string ,
                     salary : range 40000 100000 ] ;
--------> -Manager  & ^ NewType3 ;
gs Manager = Manager : prim 0  1 ;
antev rule1a 0 =  Storage  & ^ [ category : vstring "B4" ] ;
--------> -Storage  & ^ NewType4 ;
gs rule1a = rule1a : antev 0  1 , Storage : prim 1  0 ;
consv rule1c 0 =  Storage  & ^ [ managed_by : TManager ] ;
--------> -Storage  & ^ NewType5 ;
gs rule1c = rule1c : consv 0  1 , Storage : prim 1  0 ;
```

Figura 3.7: Schema con regole in forma canonica (file .fc)

- Molteplicitá del fattore

Il *path* rappresenta il cammino, tra gli attributi della classe (in dot notation), per giungere all'attributo di cui si intende riportare la descrizione: puó trattarsi dell'attributo stesso ovvero essere nullo: in questo caso viene indicato con la parola chiave *void*.

Il *tipo di dominio* é un numero intero che rappresenta uno dei tipi previsti secondo le corrispondenze di tabella 3.1.

Il *tipo di fattore* é un numero intero che costituisce una *label* utilizzata per la classificazione dei fattori in fase di visualizzazione.

Il *nome del dominio* corrisponde al dominio in cui mappa l'attributo espresso nel campo *path*.

La *molteplicitá del fattore* vale 1 se il fattore é multiplo, 0 altrimenti.

A questo punto il programma é in grado di procedere con l'espansione semantica delle interrogazioni.

Inserendo una query come input, il modulo ODB-QOptimizer acquisisce gli schemi sopra descritti e procede con l'ottimizzazione. Se, ad esempio, la

```
<CLASS_TYPE>
Storage 3 0
category 52 20001 string 0
managed_by 3 20001 Manager 0
name 52 20001 string 0
;
TManager 3 0
void 11 20001 Manager 0
level 100031 20001 8 0
level 100032 20001 12 0
name 52 20001 string 0
salary 100031 20001 40000 0
salary 100032 20001 100000 0
;
Manager 3 0
level 100031 20001 1 0
level 100032 20001 15 0
name 52 20001 string 0
salary 100031 20001 40000 0
salary 100032 20001 100000 0
;
rule1a 14 0
void 11 20001 Storage 0
category 55 20001 "B4" 0
managed_by 3 20001 Manager 0
name 52 20001 string 0
;
rule1c 15 0
void 11 20001 Storage 0
category 52 20001 string 0
managed_by 3 20001 TManager 0
name 52 20001 string 0
</CLASS_TYPE>
```

Figura 3.8: Forma fattorizzata dello schema canonico (file .ft)


```
#define BOTTOM   0     tipo vuoto
#define B        1     tipo base
#define T        2     nome tipo valore
#define C        3     nome classe primitiva
#define D        4     nome classe virtuale
#define ESET     5     tipo insieme con quantificatore esistenziale
#define SET      6     tipo insieme con quantificatore universale
#define SEQ      7     tipo sequenza
#define EN       8     tipo ennupla
#define ITEM     9     tipo attributo ennupla
#define DELTA   10     tipo oggetto
#define CB      11     nome classe fittizia
#define NB      12     nome tipo base
#define NTC     13     nome di tipo o di classe
#define RA_V    14     antecedente di una regola di tipo classe virtuale
#define RC_V    15     conseguente di una regola di tipo classe virtuale
#define RA_T    16     antecedente di una regola di tipo valore
#define RC_T    17     conseguente di una regola di tipo valore
#define ESEQ    18     tipo sequenza con quantificatore esistenziale
#define TOP     99     top class
#define MAXTYPE 18     massimo tipo definito
```

Tabella 3.1: Corrispdenza fra tipi ed interi nel software

query iniziale é quella di figura 3.9, si possono enucleare i risultati parziali, nonché il risultato finale dell'ottimizzazione.

```
select *
from Storage
where category = ‘‘B4’’
```

Figura 3.9: Query iniziale inserita dall'utente (file .oql)

Il funzionamento dell'ottimizzatore, come detto in 3.3, prevede di generare iterativamente l'espansione semantica della query partendo da quella iniziale. Ad ogni passo vengono inoltre applicate tutte la possibili regole applicabili tramte la congiunzione del conseguente di ciascuna di esse.

L'espansione termina allorché non si riscontri l'esistenza di ulteriori regole applicabili.

Per ognuna delle query intermedie, che risultano dalle varie applicazioni dell'espansione, viene generata la rispettiva forma canonica che costituisce l'input per la successiva iterazione dell'algoritmo.

In corrispondenza della query finale ottimizzata (virt query2 nell'esempio) ottenimo lo schema di figura 3.10. Al solito per ragioni di praticitá sono state omesse le descrizioni dei risultati parziali che il programma memorizza all'interno dello stesso file .oql.fc.

```
type NewType9 = [ category : vstring "B4" , managed_by : NewName2 ,
                  name : string ] ;
--------> [ category : vstring "B4" , managed_by : NewName2 ,
            name : string ] ;
gs NewType9 = NewType9 : type 0  1 , NewType1 : type 0  576840 ,
              NewType4 : type 0  576872 , NewType5 : type 0  576968 ,
              NewType7 : type 0  577064
virt query2 = -Storage  & ^ [ category : vstring "B4" ,
              managed_by : -Manager  & -TManager
              & ^ [ level : range 8 9 , name : string ,
               salary : range 40000 100000 ] , name : string ] ;
--------> -Storage  & ^ NewType9 ;
gs query2 = query2 : virt 0  1 , Storage : prim 0  0 ,
            rule1a : antev 0  0 , rule1c : consv 1  0 ,
            query : virt 1  0 , query1 : virt 0  1
```

Figura 3.10: Query finale ottimizzata in forma canonica (file .oql.fc)

Lo schema di figura 3.10, che é l'ultimo ad essere generato, viene portato in forma di fattori (figura 3.11)

```
void 11 20001 Storage 0
category 55 20001 "B4" 0
managed_by 3 20001 TManager 0
name 52 20001 string 0
```

Figura 3.11: Fattori della query finale ottimizzata

Alla luce del risultato ottenuto fattorizzando lo schema canonico dalla query finale, si evince come il programma possa giungere all'output da visualizzare. Confrontando infatti i fattori in figura 3.11 con quelli dello schema iniziale (figura 3.8), é possibile marcare quelli della query che non comparivano all'inizio nello schema.

Ebbene un parser puó agevolmente tradurre in OQL i fattori marcati e visualizzare infine la query indicata in figura 3.3 con Q'.

L'output é riportato di seguito.

```
select *
from Storage as S
where S.category = "B4" and
      S.managed_by in TManager
```

Figura 3.12: Fattori della query finale ottimizzata

Capitolo 4

Ottimizzazione semantica di interrogazioni

Il calcolo dell'espansione semantica di un tipo si articola in due passi: la iterazione della trasformazione secondo cui " se un tipo implica il precedente di una regola allora il conseguente della regola stessa puó essere aggiunto" e la valutazione delle implicazioni logiche attraverso il *calcolo della sussunzione* [BN94b, BB97] tra tipi.

Gli algoritmi per il calcolo dell'incoerenza e della sussunzione tra i tipi in uno schema ODL, adottando una semantica gfp, sono introdotti in [BN94b]. In [BB97], questo discorso é esteso fornendo algoritmi per la rilevazione delle incoerenze e per il calcolo della sussunzione anche con semantiche descrittive e lfp.

Si osservi che questi problemi non sono teoricamente risolvibili dal punto di vista della complessitá computazionale, tuttavia, se si introducono gli stessi algoritmi su una *descrizione semplificata dello schema* (chiamata *schema canonico*), si possono risolvere entrambi i problemi in un tempo polinomiale nella dimensione dello schema canonico.

In questo capitolo viene analizzata la problematica introdotta dai cicli nell'espansione semantica e viene presentato il nuovo algoritmo che effettua l'ottimizzazione semantica negli schemi ciclici. L'algoritmo é stato progettato,realizzato e verificato per essere integrato nel sistema ODB-Tools.

4.1 Forma canonica

La *forma canonica* di uno schema é una forma in cui, essenzialmente, sono state risolte le congiunzioni tra le strutture complesse. Formalmente, dato un sistema di tipi $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{P}\cup\mathbf{V})$ un *tipo canonico* é un tipo ottenuto applicando

la seguente regola sintattica:

$$S \rightarrow N \mid P_1 \sqcap \ldots \sqcap P_n \sqcap N \mid [a_1 : N_1, \ldots, a_k : N_k] \mid \forall\{N\} \mid \exists\{N_1\} \sqcap \ldots \sqcap \exists\{N_n\} \sqcap \forall\{N\} \mid \triangle N^*$$

Uno schema canonico é allora uno schema definito su un sistema di tipi canonici.

La trasformazione deriva dall'applicazione di un insieme di regole che riscrivono i tipi mantenendo l'interpretazione della descrizione di tipo. Per questo motivo, le incoerenze rilevate e la sussunzioni calcolate rispetto ad uno schema canonico sono *le stesse* che osserveremmo sullo schema originale.

L'algoritmo per trasformare uno schema σ in uno schema canonico ν é definito in [BB93].

I passaggi fondamentali della trasformazione sono:

1. Ogni classe base C dello schema σ é trasformata in una classe virtuale D dello schema ν, che la rappresenta attraverso alla congiunzione tra un *atomo fittizio* $\hat{C}$, che corrisponde ad una classe base priva di descrizione, quindi $\nu(\hat{C}) = \triangle\top$, e la sua descrizione originaria ($\sigma(C)$), ottenendo così $\nu(D) = \hat{C} \sqcap \sigma(C)$. Si osservi che l'interpretazione di $\hat{C}$ é un insieme arbitrario di oggetti del dominio. Ad esempio, la classe base `employee` dello schema viene trasformata nel modo seguente:

$$\begin{aligned} \nu(\texttt{employee}) &= \widehat{\texttt{employee}} \sqcap \sigma(\texttt{employee}) \\ &= \widehat{\texttt{employee}} \sqcap \triangle[\texttt{name: string}, \texttt{level: level}, \\ &\qquad \texttt{worksin: department}] \end{aligned}$$

2. La descrizione di ciascuna classe viene trasformata sostituendo i nomi delle sole classi dalle quali eredita con la rispettiva descrizione, espandendo i nomi di tipo–valore e applicando le regole di congiunzione possibili.

   Ad esempio, se vale $\sigma(\texttt{manager}) = \texttt{employee} \sqcap \triangle[\texttt{level: mdmLevel}]$, avremo:

$$\begin{aligned} \nu(\texttt{manager}) &= \widehat{\texttt{employee}} \sqcap \triangle[\texttt{name: string}, \texttt{level: mdmLevel}, \\ &\qquad \texttt{worksin: department}] \end{aligned}$$

   É opportuno notare che in $\nu(\texttt{manager})$ si perde la relazione di ereditarietá con `employee`.

3. Viene assegnato un nuovo nome a ciascuno dei nuovi tipi introdotti dalla trasformazione. La trasformazione di `employee` sará dunque:

$$\begin{aligned} \nu(\texttt{employee}) &= \widehat{\texttt{employee}} \sqcap \triangle T_0 \\ \nu(T_0) &= [\texttt{name: string}, \texttt{worksin: department}, \texttt{level: level}] \end{aligned}$$

Il punto chiave della trasformazione é il passo due, dal momento che permette di *risolvere*, attraverso la semantica delle congiunzioni, i conflitti in caso di ereditarietá: se un attributo locale é ereditato anche da una (o piú) classi, il tipo di dominio risultante dell'attributo é ottenuto come congiunzione dei tipi dati (vedi l'attributo `level` di `manager`).

Allo scopo di garantire la terminazione dell'algoritmo in presenza di classi cicliche, prima di introdurre un nuovo nome, viene sempre eseguito un controllo. Per esempio,

$$\begin{aligned}\sigma_P(C_1) &= (\triangle.a\colon 20 \div \infty) \sqcap \triangle[c\colon C_1]\\ \sigma_P(C_2) &= (\triangle.b\colon 30 \div \infty) \sqcap \triangle[c\colon C_2]\\ \sigma_V(N) &= C_1 \sqcap C_2 \sqcap (\triangle.a\colon 60 \div \infty)\end{aligned}$$

$$\begin{aligned}\nu(N) &= \widehat{C_1} \sqcap \widehat{C_2} \sqcap (\triangle.a\colon 60 \div \infty) \sqcap (\triangle.b\colon 30 \div \infty) \sqcap \triangle[c\colon N_1]\\ &\quad \text{dove } \ \sigma(N_1) = C_1 \sqcap C_2\\ \nu(N_1) &= \widehat{C_1} \sqcap \widehat{C_2} \sqcap (\triangle.a\colon 20 \div \infty) \sqcap (\triangle.b\colon 30 \div \infty) \sqcap \triangle[c\colon N_2]\\ &\quad \text{dove } \ \sigma(N_2) = C_1 \sqcap C_2 \Rightarrow N_1 = N_2\\ \nu(N_1) &= \widehat{C_1} \sqcap \widehat{C_2} \sqcap (\triangle.a\colon 20 \div \infty) \sqcap (\triangle.b\colon 30 \div \infty) \sqcap \triangle[c\colon N_1]\end{aligned}$$

4.2 Algoritmo di Espansione Semantica di un tipo

Il punto di partenza per definire un algoritmo di espansione semantica è uno schema con regole $(\sigma, \mathbf{R})$, nel quale sia stata controllata la coerenza delle classi e (eventualmente) le contraddizioni e le ridondanze dell'insieme delle regole di integrità [BBLS94].

L'algoritmo è basato sul calcolo della funzione di espansione semantica $\tilde{,}(Q)$, la quale viene trasformata in una nuova funzione che permette un algoritmo di calcolo più efficiente. Gli strumenti necessari per l'algoritmo sono lo schema canonico (vedi algoritmo di tabella A.2), l'incoerenza (vedi algoritmo di tabella A.5) e la sussunzione (vedi algoritmo di tabella A.6).

In particolare volendo calcolare l'espansione semantica del tipo S occorre sempre calcolarne la coerenza poichè nel caso in cui S sia effettivamente incoerente significa che è il tipo più specifico che si possa ottenere ($S \simeq_{\mathrm{R}} \bot$) e quindi non ha più significato il calcolo dell'espansione semantica.

Consideriamo, a titolo di esempio, la seguente interrogazione definita sullo schema di tabella 2.1:

[Q_1] "Seleziona gli `storage` in cui tutti i materiali conservati hanno un rischio più grande o uguale a 15":

$$\sigma_{\mathrm{V}}(Q_1) \;=\; \texttt{storage} \sqcap (\triangle.\texttt{stock}.\forall.\texttt{item}.\triangle.\texttt{risk}\colon 15 \div \infty)$$

Prima di descrivere l'algoritmo, riprendiamo il concetto di *dipendenza* di un tipo che lega il tipo stesso ai tipi contenuti nella propria descrizione: esso permette di definire l'insieme $\mathbf{CT}^+(S)$ dei tipi (compreso S) che compaiono nella descrizione di S a qualsiasi livello di innestamento.

A partite dallo schema canonico $\overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$ diciamo che:

- $\mathbf{CT}^0(S) = \{S\} \cup \{S_1 \in \overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}}) \setminus (\mathbf{B} \cup \overline{\mathbf{C}}) \mid S \textit{ dipende da } S_1\}$
- $\mathbf{CT}^{i+1}(S) = \mathbf{CT}^i(S) \cup \{S_1 \in \overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}}) \setminus \mathbf{B} \cup \overline{\mathbf{C}} \mid S_2 \in \mathbf{CT}^i(S) \wedge S_2 \textit{ dipende da } S_1\}$
- $\mathbf{CT}^+(S) = \mathbf{CT}^{\bar{i}}(S)$

dove $\bar{i}$ è il piú piccolo intero tale che $\mathbf{CT}^{\bar{i}}(S) = \mathbf{CT}^{\bar{i}+1}$.

Quindi $\mathbf{CT}^+(S)$ contiene tutti i tipi che compaiono nella descrizione di S a qualsiasi livello di innestamento ad esclusione degli *atomi fittizi* ed i *tipi − valore base*; il motivo di tale esclusione apparirá piú chiaro in seguito, per ora basti ricordare che le regole non sono definite sui *tipi − valore base* e che gli *atomi fittizi* hanno descrizione $\triangle\top$.

Consideriamo ora come esempio l'interrogazione Q_1 ed esprimiamola nella sua forma canonica:

$$
\begin{array}{lcl}
\nu(Q_1) & = & \overline{\texttt{storage}} \sqcap \triangle t_1 \\
 & & \\
\nu(t_1) & = & [\texttt{managed-by}\colon \texttt{manager}, \texttt{category}\colon \texttt{string}, \\
 & & \ \texttt{stock}\colon t_2] \\
\nu(t_2) & = & \{t_3\} \\
\nu(t_3) & = & [\texttt{item}\colon Q_1', \texttt{qty}\colon b_1] \\
\nu(Q_1') & = & \overline{\texttt{material}} \sqcap \triangle t_4 \\
\nu(t_4) & = & [\texttt{name}\colon \texttt{string}, \texttt{risk}\colon b_2, \texttt{feature}\colon t5] \\
\nu(t_5) & = & \{\texttt{string}\} \\
\nu(b_1) & = & 10 \div 300 \\
\nu(b_2) & = & 15 \div \infty \\
\nu(\texttt{manager}) & = & \overline{\texttt{manager}} \sqcap \triangle t_6 \\
\nu(t_6) & = & [\texttt{name}\colon \texttt{string}, \texttt{salary}\colon b_3, \texttt{level}\colon b_4] \\
\nu(b_3) & = & 40 \div \infty \\
\nu(b_4) & = & 1 \div 15
\end{array}
$$

Avendo a disposizione la forma canonica di Q_1, l'insieme $\mathbf{CT}^+(Q_1)$ si calcola molto facilmente ottenendo:

$$\mathbf{CT}^+(Q_1) = \{Q_1, t_1, \texttt{manager}, t_6, t_2, t_3, Q_1', t_4, t_5\}$$

L'insieme $\mathbf{CT}^+(S)$ permette di introdurre la seguente proposizione di trasformazione della funzione di espansione semantica:

Proposizione 2 *Dato lo schema canonico* $\overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$*, il tipo* S *e l'insieme* $\mathbf{CT}^+(S) \subseteq \overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$*,*
la funzione di Espansione Semantica $, (S)$*:*

$$
, (S) = \begin{cases} S \sqcap \bigsqcap_k (p_k : S_k^c) & \forall R_k, p_k : \ S \sqsubseteq_\sigma (p_k\colon S_k^a), \\ & \qquad S \not\sqsubseteq_\sigma (p_k\colon S_k^c) \\ & \\ S & \textit{altrimenti} \end{cases} \tag{4.1}
$$

è equivalente alla seguente funzione:

$\forall\ S_1 \in \mathbf{CT}^+(S)$

$$
, (S_1) = \begin{cases} S_1 \sqcap \bigsqcap_k (\epsilon\colon S_k^c) & \forall R_k : \ S_1 \sqsubseteq_\sigma (\epsilon\colon S_k^a), \\ & \qquad S_1 \not\sqsubseteq_\sigma (\epsilon\colon S_k^c) \\ & \\ S_1 & \textit{altrimenti} \end{cases} \tag{4.2}
$$

che può anche essere scritta come:

$\forall\ S_1 \in \mathbf{CT}^+(S)$

$$
, (S_1) = \begin{cases} S_1 \sqcap \bigsqcap_k S_k^c & \forall R_k : \ S_k^a \in GS(S_1), \\ & \qquad S_k^c \notin GS(S_1) \\ & \\ S_1 & \textit{altrimenti} \end{cases} \tag{4.3}
$$

Illustriamo la proposizione applicandola all'esempio dell'interrogazione Q_1. Nella funzione 4.1 i cammini significativi per il controllo della sussunzione sono soltanto quelli in cui il generico cammino p_k identifica un *tipo cammino* $(p_k : S_k^c)$ confrontabile con S (ad esempio sono due classi oppure due tipi-valore). Inoltre, poichè le regole non sono definite sui $tipi - valore\ base$, per i cammini che mappano su di un $tipo - valore\ base$ possiamo evitare di controllarne la sussunzione in quanto a priori si puó affermare che le regole non sono applicabili con tali cammini.

Quindi i cammini significativi per Q_1, seguendo una visita depth first, sono i seguenti:
ϵ
$\triangle$
$\triangle$.managed-by
$\triangle$.managed-by.$\triangle$
$\triangle$.stock
$\triangle$.stock.$\forall$
$\triangle$.stock.$\forall$.item
$\triangle$.stock.$\forall$.item.$\triangle$
$\triangle$.stock.$\forall$.item.$\triangle$.feature

Mostriamo ora quali sono i cammini p_k che vengono implicitamente considerati dalla funzione 4.2 al variare di $S_1 \in \mathbf{CT}^+(Q_1)$ attraverso il controllo della condizione $S_1 \sqsubseteq_\sigma (\epsilon\colon S_k^a)$:

$S_1 = Q_1$	cammino	ϵ
$= t_1$		$\triangle$
$= \texttt{manager}$		$\triangle$.managed-by
$= t_6$		$\triangle$.managed-by.$\triangle$
$= t_2$		$\triangle$.stock
$= t_3$		$\triangle$.stock.$\forall$

$= Q_1'$	$\triangle$.stock.$\forall$item
$= t_4$	$\triangle$.stock.$\forall$item.$\triangle$
$= t_5$	$\triangle$.stock.$\forall$item.$\triangle$.feature

Questi cammini coincidono esattamente con quelli considerati dalla funzione 4.1, quindi abbiamo verificato intuitivamente l'equivalenza tra 4.1 e 4.2 nell'esempio presentato.

Ricordando poi la definizione di $GS(S)$:

$$GS(S) = \{S_1 \in \mathbf{S} \mid S \sqsubseteq_\sigma S_1\}$$

risulta ovvia l'equivalenza tra $S \sqsubseteq_\sigma (\epsilon\colon S_k^a)$ e $S_k^a \in GS(S)$ da cui segue immediatamente che la 4.3 può essere applicata in luogo della 4.2 (e della 4.1).

Questa considerazione è molto importante poichè permette di ridurre il numero di $tipi - cammino$ da verificare nel calcolo dell'espansione semantica e porta alla definizione di un algoritmo efficiente di ottimizzazione semantica basato appunto sulla funzione 4.3.

L'algoritmo è riportato in tabella 4.1.

Il processo termina quando non abbiamo piú regole applicabili al tipo S (ed ai tipi che compaiono nella sua descrizione ad ogni livello di innestamento). Come risultato l'algoritmo fornisce il tipo S in forma canonica descritto dalla propria espansione semantica; per semplicità, rappresentiamo questa trasformazione con un nuovo schema canonico $\tilde{\nu}$, cioè $\tilde{\nu}(S) = \tilde{,}(S)$.

Vediamo l'espansione semantica che si ottiene per la Q_1 applicando l'algoritmo. Come abbiamo detto, un'interrogazione Q viene considerata una classe virtuale dello schema σ; si considera quindi lo schema canonico ν equivalente a σ e, in particolare, $\nu(Q_1)$. L'espressione $\nu(Q_1)$ permette tra l'altro il controllo di coerenza dell'interrogazione rispetto allo schema σ. Viene quindi determinata l'espansione semantica di Q_1 tramite $\tilde{,}$, cioè il tipo $\tilde{,}(Q_1)$, e viene considerata la sua forma canonica $\tilde{\nu}(Q_1)$ che è la seguente:

Algoritmo (Espansione semantica di un tipo S)

- **Inizializzazione**:
  Acquisizione del tipo S
- **Iterazione**:
  $\forall\ S_1 \in \mathbf{CT}^+(S)$

$$\begin{aligned} \textit{Forma canonica} &: \nu(S_1) \\ \textit{Incoerenza} &: S_1 \in \widetilde{\Phi}_\nu \\ \textit{Sussunzione} &: GS(S_1) \end{aligned}$$

$$\textit{se } S_1 \notin \widetilde{\Phi}_\nu : \quad ,^i(S_1) = \begin{cases} S_1 \sqcap \bigsqcap_k S_k^c & \forall R_k : \ S_k^a \in GS(S_1), \ S_k^c \notin GS(S_1) \\ S_1 & \text{altrimenti} \end{cases}$$

- **Stop**:
  $\forall\ S_1 \in \mathbf{CT}^+(S), ,^{i+1}(S_1) = ,^i(S_1)$,
  or
  $\exists\ S_1 \in \mathbf{CT}^+(S), ,^i(S_1) = \bot$.

  L'espansione semantica è $\tilde{,}(S) = ,^i(S)$

Tabella 4.1: Algoritmo di espansione semantica

$$
\begin{array}{lcl}
\tilde{\nu}(Q_1) & = & \overline{\texttt{storage}} \sqcap \overline{\texttt{sstorage}} \sqcap \triangle t_1 \\
\tilde{\nu}(t_1) & = & [\texttt{managed-by}\colon \texttt{manager}, \texttt{category}\colon \texttt{string}, \\
 & & \texttt{stock}\colon t_2] \\
\tilde{\nu}(t_2) & = & \{t_3\} \\
\tilde{\nu}(t_3) & = & [\texttt{item}\colon Q_1', \texttt{qty}\colon b_1] \\
\tilde{\nu}(b_1) & = & 10 \div 300 \\
\tilde{\nu}(Q_1') & = & \overline{\texttt{material}} \sqcap \overline{\texttt{smaterial}} \sqcap \triangle t_4 \\
\tilde{\nu}(t_4) & = & [\texttt{name}\colon \texttt{string}, \texttt{risk}\colon b_2, \texttt{feature}\colon t_5\{\texttt{string}\}] \\
\tilde{\nu}(t_5) & = & \{\texttt{string}\} \\
\tilde{\nu}(b_2) & = & 15 \div \infty \\
\tilde{\nu}(\texttt{manager}) & = & \overline{\texttt{manager}} \sqcap \triangle t_6 \\
\tilde{\nu}(t_6) & = & [\texttt{name}\colon \texttt{string}, \texttt{salary}\colon b_3, \texttt{level}\colon b_4] \\
\tilde{\nu}(b_3) & = & 40 \div \infty \\
\tilde{\nu}(b_4) & = & 1 \div 15
\end{array}
$$

L'espressione $\tilde{\nu}(Q)$ consente di ottenere in modo immediato la classificazione dell'interrogazione rispetto alla tassonomia delle classi base. Più precisamente, è possibile ottenere sia le classi base più specifiche che generalizzano l'interrogazione sia le classi base più specifiche che generalizzano le sotto–interrogazioni. Riportiamo le definizioni:

$$GS(Q) = \{C \in \mathbf{C} \mid Q \sqsubseteq_\Sigma C\}$$

Da $GS(Q)$ è ricavabile in maniera immediata

$$MSGS(Q) = \{C \in GS(Q) \mid \nexists C' \in GS(Q)\colon C' \sqsubseteq_\Sigma C\}$$

$MSGS(Q)$ individua l'insieme delle classi base più specifiche che generalizzano l'interrogazione Q.

L'aspetto fondamentale è che l'insieme $GS(Q)$, e quindi $MSGS(Q)$, è ottenibile direttamente dalla forma canonica dell'espansione semantica e non si devono fare ulteriori calcoli di sussunzione.

4.3 Limiti dell'algoritmo di espansione semantica

Un assunto fondamentale dell'algoritmo presentato in tabella 4.1 riguarda la condizione di terminazione.

In particolare, la condizione di terminazione sarebbe garantita dal fatto che il numero di regole è sempre finito e che una regola non può essere

applicata piú di una volta al tipo $S_1 \in \mathbf{CT}^+(S)$ per via del controllo $S_k^c \notin GS(S_1)$.

Se, da un lato, in assenza di cicli questa condizione é comunque verificata, dall'altro, la possibilitá di introdurre nello schema classi e regole cicliche puó ingenerare dei cicli infiniti. In altri termini, si puó verificare che una regola venga iterativamente applicata un numero infinito di volte. Alcuni esempi potranno chiarire meglio questo concetto.

$$(\triangle.a\colon 10 \div \infty) \;\rightarrow\; (\triangle.c.\triangle.a\colon 20 \div \infty)$$

L'esempio sopra introdotto riporta una regola che agisce su una classe ciclica. É stato sottolineato nella sezioni precedenti come l'algoritmo che realizza la forma canonica dello schema preveda la possibilitá di avere classi o regole cicliche. L'acquisizione dello schema dunque non comporta problemi di terminazione e viene calcolata correttamente la forma canonica.

A questo punto, tuttavia, una qualsivoglia interrogazione che faccia scattare la regola comporta la non terminazione dell'algoritmo di espansione semantica. Infatti, pur essendoci una sola regola (quindi un numero finito), questa stessa viene applicata un numero infinito di volte per via della ciclicitá della classe su cui agisce. Una query con questa caratteristiche é proposta di seguito.

$$\sigma_V^0(N) \;=\; (\triangle.a\colon 45 \div \infty)$$

Il punto critico, per un algoritmo così concepito, risiede nel fatto che questo non confronta semanticamente i tipi per i quali si appresta a calcolare l'espansione con quelli giá considerati nel corso delle precedenti iterazioni. Puó accadere dunque che una stessa regola (o un insieme finito di regole) continui ad essere applicata all'infinito ad uno stesso tipo poiché ogni volta quest'ultimo non viene riconosciuto come giá espanso. É proprio in questo caso che la condizione di terminazione dell'algoritmo 4.1 diventa inefficace.

Una situazione del tutto analoga ricorre introducendo il seguente schema:

$$
\begin{array}{l}
(\triangle.a\colon 20 \div \infty) \sqcap (\triangle.b\colon 30 \div \infty) \;\rightarrow\; (\triangle.c.\triangle.b\colon 60 \div \infty) \\
(\triangle.a\colon 41 \div \infty) \sqcap (\triangle.b\colon 33 \div \infty) \;\rightarrow\; (\triangle.c.\triangle.a\colon 75 \div \infty)
\end{array}
$$

e la query:

$$\sigma_V^0(N) \;=\; (\triangle.a\colon 81 \div \infty) \sqcap (\triangle.b\colon 99 \div \infty)$$

Al fine di sottolineare ulteriormente i limiti della versione descritta in tabella 4.1 dell'algoritmo, possiamo calcolare l'espansione semantica delle

regole e della query proposte. Con riferimento alle regole il risultato sarebbe:

$$\begin{aligned}
\nu^0(N_1) &= \triangle N_2 \\
\nu^0(N_2) &= (a\colon 20 \div \infty) \sqcap (b\colon 30 \div \infty) \\
\nu^0(N_3) &= \triangle N_4 \\
\nu^0(N_4) &= [c\colon N_5] \\
\nu^0(N_5) &= \triangle N_6 \\
\nu^0(N_6) &= (b\colon 60 \div \infty) \\
\nu^0(N_{11}) &= \triangle N_{12} \\
\nu^0(N_{12}) &= (a\colon 41 \div \infty) \sqcap (b\colon 33 \div \infty) \\
\nu^0(N_{13}) &= \triangle N_{14} \\
\nu^0(N_{14}) &= [c\colon N_{15}] \\
\nu^0(N_{15}) &= \triangle N_{16} \\
\nu^0(N_{16}) &= (a\colon 75 \div \infty)
\end{aligned}$$

$$\begin{aligned}
N_1 &\rightarrow N_3 \\
N_{11} &\rightarrow N_{13}
\end{aligned}$$

Sviluppando il calcolo dell'espansione per la query:

$$\begin{aligned}
\nu^0(N) &= \triangle N_{20} \\
\nu^0(N_{20}) &= (a\colon 81 \div \infty) \sqcap (b\colon 99 \div \infty) \\
\sigma^1_V(N) &= \sigma^0_V(N) \sqcap N_3 \Rightarrow \nu^1(N) = \triangle N_{21} \\
\nu^1(N_{21}) &= (a\colon 81 \div \infty) \sqcap (b\colon 99 \div \infty) \sqcap [c\colon N_5] \\
\sigma^2_V(N) &= \sigma^1_V(N) \sqcap N_{13} \Rightarrow \nu^2(N) = \triangle N_{22} \\
\nu^2(N_{22}) &= (a\colon 81 \div \infty) \sqcap (b\colon 99 \div \infty) \sqcap [c\colon N_{23}] \\
&\quad \text{dove } \sigma^2_V(N_{23}) = N_5 \sqcap N_{15} \Rightarrow \nu^2(N_{23}) = \triangle N_{24} \\
\nu^2(N_{24}) &= (a\colon 75 \div \infty) \sqcap (b\colon 60 \div \infty) \\
\sigma^3_V(N_{23}) &= \sigma^2_V(N_{23}) \sqcap N_3 \Rightarrow \nu^3(N_{23}) = \triangle N_{25} \\
\nu^3(N_{25}) &= (a\colon 75 \div \infty) \sqcap (b\colon 60 \div \infty) \sqcap [c\colon N_5] \\
\sigma^4_V(N_{23}) &= \sigma^3_V(N_{23}) \sqcap N_{13} \Rightarrow \nu^4(N_{23}) = \triangle N_{26} \\
\nu^4(N_{26}) &= (a\colon 75 \div \infty) \sqcap (b\colon 60 \div \infty) \sqcap [c\colon N_{27}] \\
&\quad \text{dove } \sigma^4_V(N_{27}) = N_5 \sqcap N_{15} \Rightarrow N_{27} = N_{23} \\
\nu^4(N_{26}) &= (a\colon 75 \div \infty) \sqcap (b\colon 60 \div \infty) \sqcap [c\colon N_{23}]
\end{aligned}$$

Procedendo in questo modo, si é pervenuti ad una espansione corretta in un numero finito di passi. Questo perché, tenendo traccia dei tipi espansi viene, rilevato il ciclo allorché si cerca di introdurre un nuovo nome che corrisponde, in realtá, ad uno giá espanso ($N_{27} = N_{23}$).

Si impone allora la necessitá di rivistare l'algoritmo di espansione, allo scopo di consentire la trattazione dei cicli eventualmente presenti nelle definizioni dello schema.

4.4 Algoritmo di Espansione con soglia

L'algoritmo di espansione con soglia rappresenta la prima soluzione proposta in ordine alla trattazione di interrogazioni ricorsive. L'assunzione fondamentale del metodo parte dal presupposto che, per un dato schema con regole e per una qualsivoglia interrogazione, l'espansione semantica di quella interrogazione rispetto allo schema deve necessariamente concludersi in un numero finito di iterazioni e che tale numero é determinabile a priori. Di qui il nome di algoritmo con soglia. La soglia rappresenta, dunque, il massimo numero di iterazioni previsto per realizzare l'espansione semantica.

Qualora l'espansione semantica dovesse richiedere un numero di passi superiore rispetto alla soglia calcolata, si tratterebbe sicuramente di una situazione ciclica.

Da quanto detto si puó facilmente dedurre che, al fine della determinazione dei cicli e, quindi, dell'affidabilitá stessa del metodo, risulta di fondamentale importanza il criterio in base al quale la soglia viene determinata.

Le considerazioni che portano alla determinazione della soglia possono essere riassunte come segue. In primo luogo, va osservato che la soglia non puó essere inferiore al numero di regole dello schema: puó infatti verificarsi che, durante l'espansione di una query, ciascuna regola venga applicata in corrispondenza di una diversa iterazione. In altri termini accade che, in uno schema con k regole, l'applicazione della $i-esima$ regola, alla $n-esima$ iterazione, comporti l'applicazione della $j-esima$ regola alla $i+1-esima$ iterazione, e così via per tutte le k regole dello schema. Se la soglia, in questo caso, fosse minore di k, l'espansione semantica terminerebbe prematuramente.

Una seconda considerazione fondamentale riguarda l'interrogazione da ottimizzare. Puó infatti accadere che una determinata *path query* richieda, per essere correttamente ottimizzata, un numero di iterazioni del metodo superiore al numero di regole dello schema. Per rendersene conto é sufficiente considerare uno schema senza regole di integritá: ebbene l'espansione semantica di qualunque interrogazione, anche in assenza di regole, richiede

una numero di passi maggiore di zero. Ricordiamo infatti che una query puó essere riclassificata anche in base alla tassonomia della classi dello schema.

Partendo dalle considerazione svolte possiamo allora pensare di fissare la soglia sulla base del massimo tra due valori: il numero di regole dello schema e la lunghezza del massimo cammino presente nella query ove, per massimo cammino, si intende il numero massimo di attributi, in dot notation, che compaiono nell'interrogazione considerata.

Una soglia calcolata in questo modo consente, come abbiamo visto, di giungere comunque alla corretta espansione di una query non ricorsiva. In ordine alla trattazione dei cicli, invece, é facile rendersi conto che, da un lato, quando la soglia coincide col numero di regole presenti nello schema, se si eccede tale valore, significa che almeno una delle regole continuerebbe ad essere applicata ciclicamente. D'altro canto, se la soglia coincide con il massimo path, allorché il numero di iterazioni oltrepassa la soglia, significa che uno stesso tipo continuerebbe ad essere espanso cilicamente.

Per quanto riguarda la terminazione del metodo, va osservato che la condizione di terminazione dell'algoritmo 4.1 permane e continua a valere nei casi non ricorsivi. Quando invece, in presenza di cicli, il metodo tenderebbe a superare la soglia, interviene una terminazione forzata che interrompe l'espansione.

L'algoritmo di espansione con soglia é presentato in tabella 4.2

Una caratteristica determinante dell'algoritmo 4.2 sta nel fatto che la soglia viene calcolata una volta per tutte in fase di inizializzazione del calcolo. Benché questo approccio, come detto, assicuri la terminazione del metodo, allo stesso tempo puó ridurne drasticamente l'efficienza.

Consideriamo, infatti, uno schema con classi cicliche e con un certo numero di regole di integritá. Supponiamo che la soglia coincida con tale numero. Se si intende ottimizzare una query ricorsiva che determini un loop infinito sin dalle prime iterazioni, la terminazione forzata occorre solo dopo un numero di iterazioni pari al valore della soglia. Se il numero delle regole é elevato, quindi, verranno eseguite inutilmente tutte la iterazioni successive al verificarsi delle condizioni di ciclo fino al raggiungimento della soglia.

Si consideri lo schema di tabella 4.3

Osserviamo cinque regole di integritá. Supponiamo ora di dover ottimizzare la seguente interrogazione:

select $*$
from test
where a $>= 6$

Il massimo cammino dell'interrogazione considerata vale uno. Con riferi-

Algoritmo (Espansione semantica con soglia di un tipo S)

- **Inizializzazione**:
  Acquisizione del tipo S
  $threshold = max(max\_path, rule\_number)$
  $j = 0$

- **Iterazione**:
  $if\ j < threshold$
  $\forall\ S_1 \in \mathbf{CT}^+(S)$

$$
\begin{array}{rcl}
Forma\ canonica & : & \nu(S_1) \\
Incoerenza & : & S_1 \in \widetilde{\Phi}_\nu \\
Sussunzione & : & GS(S_1)
\end{array}
$$

$$
if\ S_1 \notin \widetilde{\Phi}_\nu :
$$

$$
,^i(S_1) = \begin{cases} S_1 \sqcap \bigsqcap_k S_k^c & \forall R_k : \quad S_k^a \in GS(S_1), \\ & \qquad\quad S_k^c \notin GS(S_1) \\ S_1 & \text{otherwise} \end{cases}
$$

$$
j = j+1
$$

  else Stop

- **Stop**:
  $\forall\ S_1 \in \mathbf{CT}^+(S),\ ,^{i+1}(S_1) = ,^i(S_1),$
  or
  $\exists\ S_1 \in \mathbf{CT}^+(S), ,^i(S_1) = \bot.$

  L'espansione semantica è $\tilde{,}(S) = ,^i(S)$

Tabella 4.2: Algoritmo di espansione con soglia

```
prim test = ^ [ a : range 1 +inf , b : range 1 +inf ,
              c : classe , d : range 1 +inf ] ;
antev rule1a = test & ^ [ a : range 5 +inf ] ;
consv rule1c = test & ^ [ c : ^ [ a : range 10 +inf ] ] ;
antev rule2a = test & ^ [ b : range 6 +inf ] ;
consv rule2c = test & ^ [ c : ^ [ d : range 20 +inf ] ] ;
antev rule3a = test & ^ [ d : range 15 +inf ] ;
consv rule3c = test & ^ [ c : ^ [ d : range 6 +inf ] ] ;
antev rule4a = test & ^ [ d : range 15 +inf ] ;
consv rule4c = test & ^ [ c : ^ [ b : range 6 +inf ] ] ;
antev rule5a = test & ^ [ b : range 15 +inf ] ;
consv rule5c = test & ^ [ c : ^ [ d : range 2 +inf ] ] .
```

Tabella 4.3: Schema con regole di prova

mento al formalismo di tabella 4.2 avremo:

$$threshold \;=\; max(1,5) \;=\; 5$$

vale a dire che l'espansione semantica per la query considerata deve prevedere al piú cinque iterazioni. Tuttavia osserviamo dallo schema che, sin dal secondo passo di espansione si verifica una condizione di ciclo.

Come é stato detto, l'algoritmo in questione continua a iterare il procedimento fino a raggiungere la soglia; nella fattispecie cinque volte. Il risultato che si ottiene é il seguente:

```
5 rule(s) detected
query longest path = 1 (d)
Max 5 iteration(s)

select *
from classe as S
where S.a > 5 and
      S.c.a > 9 and
      S.c.c.a > 9 and
      S.c.c.c.a > 9 and
      S.c.c.c.c.a > 9
```

In definitiva questo metodo non é in grado di adattarsi alle condizioni nelle quali si calcola l'espansione semantica; questa staticitá é appunto la

causa del degrado delle prestazioni nei casi sopra descritti. Si impone dunque la necessitá di realizzare una nuova versione dell'algoritmo di espansione semantica che rechi le medesime garanzie di terminazione nei casi ricorsivi, ma che sia contemporaneamente in grado di adattarsi alla query da espandere al fine di presentare, in tutti i casi, la stessa efficienza computazionale. L'algoritmo in questione viene proposto nella sezione successiva.

4.5 Algoritmo di Espansione Semantica Generale

La caratteristica fondamentale del nuovo algoritmo di espansione semantica é costituita dalla proprietá di memorizzare tutte le informazioni possibili sulle iterazioni giá effettuate. Questo approccio consente di mantenere invariato, rispetto alla versione precedente, il controllo sulla terminazione poiché, utilizzando le informazioni relative alle iterazioni precedenti, é possibile trattare il caso ciclico.

Come é stato detto nella sezione precedente, la funzione di espansione semantica , di un tipo S agisce effettuandone la congiunzione con i conseguenti delle regole che non compaiono in $\mathrm{GS}(S_1)\ \forall\ S_1 \in \mathbf{CT}^+(S)$. Tipicamente, una situazione di ciclo comporta che, ad ogni iterazione dell'algoritmo 4.1, venga riconosciuta sempre la stessa regola S_k^a come applicabile ad un determinato tipo S_1 con un diverso cammino e, di conseguenza, venga effettuata la congiunzione $S_1 \sqcap \bigsqcap_k S_k^c$ senza che si pervenga mai alla terminazione.

Si considerino ora i cammini p_k che vengono implicitamente considerati dalla funzione 4.2 al variare di $S_1 \in \mathbf{CT}^+(Q_1)$ attraverso il controllo della condizione $S_1 \sqsubseteq_\sigma (\epsilon\colon S_k^a)$. Ebbene, all'interno di un ciclo, é immediato verificare che, a partire dall'istante di inizio del ciclo in poi, i path che si considerano sono caratterizzati dal ripetersi, appunto ciclico, dello stesso percorso.

Prendiamo dunque in considerazione la terna (S_1, S_k^a, p_k) ove:

S_1 rappresenta il nome del tipo da espandere
S_k^a é l'antecedente di una regola applicabile al tipo in questione
p_k rappresenta il cammino di applicazione della regola

Se, dopo ogni applicazione della funzione , di espansione semantica, viene memorizzata una terna con queste caratteristiche é possibile stabilire, in corrispondenza di eventuali iterazioni successive, se la stessa regola debba ancora essere applicata allo stesso tipo o meno. In altri termini queste informazioni sono sufficienti ad individuare, nelle successive iterazioni, una condizione di ciclo.

Piú in particolare, durante il processo di ottimizzazione, viene mantenuta in memoria una tabella ordinata su R_j^a con un numero di enrty (costuite ciacsuna da una terna (S^j, R_j^a, p_j)) pari al numero di precedenti applicazioni della funzione , . Quando si individua una regola applicabile si confronta il suo antecedente con tutti gli antecedenti memorizzati fino a quel momento. Se occorrono una o piú corrispondenze vengono raffrontati il nome del tipo cui la regola candidata andrebbe congiunta con il nome del tipo presente nella entry della tabella. Solo se anche in questo esiste un'uguaglianza si prendono i considerazione i cammini di applicazione della regola nei casi attuale e memorizzato. Una opportuna funzione (che nell'algoritmo é stata indicata p_j *cicla su* p_k) consente di valutare se il path memorizzato in tabella p_j ricorre ciclicamente all'interno del path attuale p_k: se ció accadesse si troveremmo all'interno di un ciclo e la regola in questione non verrebbe applicata.

Riprendiamo l'esempio di pagina 52.
Alla prima iterazione dell'algoritmo l'unica regola presente viene riconosciuta come applicabile. Viene quindi applicata la funzione , in corrispondenza di un cammino:
$\triangle.c$
Alla seconda iterazione ci troviamo giá all'interno del ciclo: la regola e il tipo sono gli stessi del passo precedente mentre il cammino si é aggiornato nel modo seguente:
$\triangle.c.\triangle.c$
A questo punto l'algoritmo 4.1 continuerebbe all'infinito esaminando i cammini:
$\triangle.c.\triangle.c.\triangle.c$
$\triangle.c.\triangle.c.\triangle.c.\triangle.c$
$\triangle.c.\triangle.c.\triangle.c.\triangle.c.\triangle.c$
e così via.

In realtá, dopo la prima applicazione della regola, con il nuovo algoritmo si avrebbe una prima entry in tabella con i seguenti valori:

$$\begin{aligned} S^1 &= \texttt{N} \\ S_1^a &= R^a \\ p_1 &= \triangle.c \end{aligned}$$

Alla seconda iterazione, come abbiamo detto, risulta nuovamente applicabile la stessa regola. Esplorando la tabella ci accorgeremmo che la regola R verrebbe ancora una volta applicata al tipo `N`. Quanto ai cammini di applicazione alla seconda passata avremo $\triangle.c.\triangle.c$. La funzione *cicla* allora individua che il cammino memorizzato $\triangle.c$ ricorre ciclicamente all'interno $\triangle.c.\triangle.c$. Dunque la regola R non viene piú applicata essendo stata individuata una condizione di ciclo e, dal momento che non vi sono ulteriori regole applicabili, l'algoritmo finisce sulla base della condizione di terminazione originaria.

La nuova versione dell'algoritmo é presentata in tabella 4.4.

Il metodo proposto per realizzare l'espansione semantica di un tipo prevede di applicare in parallelo tutte le regole applicabili ad un dato passo dell'esecuzione. Se le regole da applicare sono piú di una, queste non vengono mai raffrontate tra loro. In questo modo si riescono a trattare efficientemente quelle situazioni non cicliche che, tuttavia, presentano ugualmente cammini ciclici.

4.6 Esempi di espansione semantica di interrogazioni

Un'interrogazione della base di dati corrisponde alla semantica di una classe virtuale poiché contiene le condizioni necessarie e sufficienti per definire gli oggetti da selezionare. Il risultato dell'interrogazione é l'istanza della classe virtuale, cioé l'insieme di oggetti *esistenti* nella base di dati che soddisfano la descrizione della classe virtuale.

Dopo aver dato le definizioni formali, in questa sezione vengono proposti alcuni esempi notevoli di espansione semantica ottenuta mediante la nuova versione dell'algoritmo presentato a pagina 61.

Non siamo tanto interessati, in questa sede, a illustrare in dettaglio le caratteristiche del software che é stato ottimizzato, quanto ad analizzare il funzionamento del nuovo algoritmo con riferimento ad esempi pratici e, quindi, di piú immediata comprensione. Sará dapprima introdotto uno schema di base di dati con classi e regole cicliche, quindi verranno prese in considerazione alcune query notevoli non necessariamente cicliche.

Consideriamo innanzi tutto lo schema con regole di tabella 4.5; si possono osservare due classi (`employee` e `manager`) ed una regola (R_3) cicliche. Vogliamo esprimere la seguente interrogazione

[Q_1]: " seleziona tutti gli impiegati con livello superiore a 15"

che puó essere riscritta attraverso il consueto formalismo:

$$\sigma_V^0(Q_1) \;=\; \texttt{employee} \sqcap (\triangle.\texttt{level}: 15 \div \infty)$$

Per come é stato concepito lo schema, alla query Q_1 puó essere applicata

Algoritmo (Espansione semantica di un tipo S - nuova versione)

- **Inizializzazione**:
  Acquisizione del tipo S
- **Iterazione**:
  $\forall\ S_1 \in \mathbf{CT}^+(S)$

$$
\begin{array}{rcl}
\mathit{Forma\ canonica} & : & \nu(S_1) \\
\mathit{Incoerenza} & : & S_1 \in \widetilde{\Phi}_\nu \\
\mathit{Sussunzione} & : & GS(S_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{if}\ S_1 \notin \widetilde{\Phi}_\nu : \\
\mathit{while}\ (j \neq \mathit{NULL}) \\
\qquad \mathit{se}\ (S_k^a \neq S_j^a), \\
\qquad \mathit{se}\ (S_1 \neq S_j), \\
\qquad \mathit{se}\ (p_k\ \mathit{cicla\ su}\ p_j)\ =\ \mathit{FALSE}, \\
\qquad \{ \\
\qquad ,^i(S_1)\ =\ \begin{cases} S_1 \sqcap \bigsqcap_k S_k^c & \forall R_k : \ S_k^a \in GS(S_1),\ S_k^c \notin GS(S_1) \\ S_1 & \text{otherwise} \end{cases} \\
\qquad \mathit{Store}(S_1, S_k^a, p_k) \\
\qquad \}
\end{array}
$$

- **Stop**:
  $\forall\ S_1 \in \mathbf{CT}^+(S),\ ,^{i+1}(S_1) = ,^i(S_1)$,
  or
  $\exists\ S_1 \in \mathbf{CT}^+(S), ,^i(S_1) = \bot$.

L'espansione semantica è $\tilde{,}(S) = ,^i(S)$

Tabella 4.4: Algoritmo di espansione semantica generale

$\sigma_{\mathcal{P}}$(employee)	=	$\triangle$[salary: $30000 \div \infty$, level: $8 \div 10$, worksin: Office, head: manager]
$\sigma_{\mathcal{P}}$(manager)	=	employee $\sqcap$ $\triangle$.[level: $15 \div \infty$]
$\sigma_{\mathcal{P}}$(Office)	=	$\triangle$.[category: $1 \div 6$, dname: string]
R_1 :		employee $\sqcap$ ($\triangle$.level: $15 \div \infty$) $\rightarrow$ manager
R_2 :		manager $\sqcap$ ($\triangle$.level: $16 \div \infty$) $\rightarrow$ salary: $60000 \div 90000$
R_3 :		manager $\sqcap$ ($\triangle$.salary: $50000 \div \infty$) $\rightarrow$ head.salary: $60000 \div \infty$
R_4 :		employee $\sqcap$ ($\triangle$.level: $8 \div 13$) $\rightarrow$ worksin.category: $1 \div 3$

Tabella 4.5: Schema con Regole di Esempio

solo la regola R_1; attraverso la congiunzione con il conseguente della regola arriviamo a:

$$\sigma_V^1(Q_1) \ = \ \texttt{manager} \sqcap (\triangle.\texttt{level}: 15 \div \infty)$$

e si conclude la prima iterazione dell'algoritmo. A questo punto diventa tuttavia lecito applicare anche R_1 per giungere a:

$$\sigma_V^2(Q_1) = \ \texttt{manager} \sqcap \ (\triangle.\texttt{level}: 15 \div \infty \ \triangle.\texttt{salary}: 50000 \div \infty)$$

A questo punto, ovvero in corrispondenza della terza passata dell'algoritmo, diventa applicabile la regola ciclica R_3;

$$\sigma_V^3(Q_1) = \ \texttt{manager} \sqcap \ (\triangle.\texttt{level}: 15 \div \infty \ \triangle.\texttt{salary}: 50000 \div \infty \ \triangle.\texttt{head.salary}: 50000 \div \infty)$$

Il punto critico é proprio questo: la regola R_3 continuerebbe ad essere applicata indefintamente. La nuova versione dell'algoritmo, invece, riesce a determinare la situazione di stallo che sta per verificarsi e impedisce che R_3 venga nuovamente applicata nelle successive iterazioni.

In queste condizioni, non essendoci ulteriori regole applicabili, la condizione di terminazione della versione di pagina 50 risulta vera e dunque il procedimento si conclude a questo punto.

Vediamo ora un esempio aciclico espresso dall'interrogazione $[Q_2]$.

$[Q_2]$: "Seleziona tutti gli impiegati per i quali il capo del proprio diretto superiore lavora in un ufficio di categoria superiore a 4"

La query risulterá piú chiara se espressa formalmente come segue:

$$\sigma^0_V(Q_2) \quad = \quad \texttt{employee} \sqcap (\triangle.\texttt{head.head.worksin.category}: 4 \div \infty)$$

Pur presentando un cammino ciclico, l'interrogazione $[Q_2]$ non risulta ciclica nel senso argomentato alla sezione precedente: non comporterá, cioé, una situazione di stallo dell'algoritmo 4.1 a causa di cicli infiniti. É stato giá osservato, in precedenza, come la nuova versione dell'algoritmo sia stata progettata tenendo conto di queste particolari interrogazioni.

Analizzando il funzionamento dell'algoritmo ci accorgiamo che per la classe $\sigma^0_V(Q_2)$, viene generato un insieme $\mathbf{CT}^+$ che contiene tutti i path seguenti:
ϵ
$\triangle$.head
$\triangle$.worksin
$\triangle$.head.head
$\triangle$.head.head.worksin
$\triangle$.head.head.head

É immediato osservare che l'unica regola applicabile, in fase di espansione, é la R_1 dal momento che ogni head é, per definizione, un manager. Questa regola viene applicata tre volte in corrispondenza degli altrettanti path che terminano con l'attributo head. Le congiunzioni vengono tutte risolte alla prima passata. In definitiva si otterrá:

$$\begin{aligned}\sigma^1_V(Q_2) = \quad \texttt{employee} \sqcap \quad & (\triangle.\texttt{head.head.worksin.category}: 4 \div \infty) \sqcap \\ & (\triangle.\texttt{head.level}: 14 \div \infty) \sqcap \\ & (\triangle.\texttt{head.head.level}: 14 \div \infty) \sqcap \\ & (\triangle.\texttt{level}: 7 \div \infty)\end{aligned}$$

Capitolo 5

Fattorizzazione

In questo capitolo viene ripreso il concetto di fattorizzazione giá introdotto nel secondo capitolo. Verrá in primo luogo illustrata l'importanza della fattorizzazione di un'interrogazione (o di una vista) nell'analisi dei suoi fattori eliminabili. In particolare si mostrerá come la fattorizzazione di interrogazioni ricorsive o di viste cicliche richieda l'uso della *chiusura transitiva*. Infine si discuterá l'implementazione di questi concetti nel sistema ODB-Tools.

5.1 Eliminazione di fattori

L'eliminazione dei fattori permette di rimuovere da una query i fattori che sono logicamente implicati dallo schema o dalle regole e che, perció, non influenzano il risultato dell'ottimizzazione semantica. Un'eliminazione completa dei fattori ridondanti genera una *forma minima* dell'interrogazione che é semanticamente equivalente a quella originaria.

In primo luogo é necessario esprimere una query come congiunzione di fattori booleani.

Definizione 10 (Fattore) *Dato* $\Sigma = (\sigma, \mathbf{R})$ *su* $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$*, un* fattore *é un tipo cammino* $(p\colon S)$ *ove* $S \in \mathbf{B} \cup \mathbf{N} \cup \{\top, \bot\}$*. Un fattore viene detto* esistenziale *se il su path include il simbolo* $\exists$*. Un fattore é detto* virtuale *se* S_i *é un nome di tipo virtuale, detto* primitivo *altrimenti.*

Uno *schema fattorizzato* (*primitivo*) $\Sigma = (\sigma, \mathbf{R})$ é uno schema in cui, per ciascun nome di tipo N e per ogni regola $R = (S^a, S^c)$, si ha che $\sigma(N)$, S^a e S^c sono una congiunzione di fattori (primitivi). Dal momento che in uno schema fattorizzato primitivo un nome di tipo virtuale non puó apparire nella descrizione di nomi o nelle regole, si ha che in tali schemi i nomi di tipi virtuali sono necessariamente aciclici.

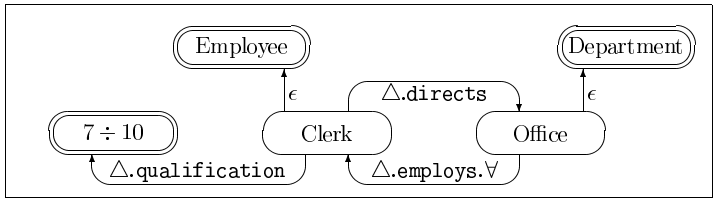
L'equivalenza tra un DL con definizioni cicliche interpretato con una semantica gfp ed un DL con la chiusura transitiva delle regole viene presentata e dimostrata in [Baa91]. In [Baa91] viene utilizzata la teoria degli automi e dei linguaggi regolari.

Di seguito viene mostrato come, utilizzando una semantica gfp, sia possibile trasformare uno schema di base di dati ciclico in uno schema fattorizzato primitivo. Piú formalmente:

Proposizione 3 *Qualsiasi schema* Σ_1 *su* $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N}_1)$ *puó essere effettivamente trasformato in uno schema fattorizzato primitivo* Σ_2 *su* $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N}_2)$*, che rappresenta un'estensione transitiva di* Σ_1*.*

Proveremo questa proposizione utilizzando alcuni esempi; Intuitivamente, come proposto in [Baa91], si puó associare un automa a stati finiti allo schema, quindi determinare lo schema fattorizzato equivalente sulla base dei linguaggi accettati dell'automa.

A titolo di esempio, possiamo considerare l'automa associato alle viste cicliche `Office` e `Clerk` [1]:



La forma canonica fattorizzata per la classe virtuale ciclica `Clerk` si ricava considerando i linguaggi accettati dall'automa che hanno `Clerk` come stato iniziale e, come stati finali, i nomi di tipi primitivi `employee` e `department` e il tipo base $7 \div 10$:

$$\begin{aligned}\sigma(\texttt{Clerk}) \;=\; & (\triangle.\texttt{directs}.\triangle.\texttt{employs}.\forall)^*\colon \texttt{employee}) \sqcap \\ & ((\triangle.\texttt{directs}.\triangle.\texttt{employs}.\forall.\triangle.\texttt{qualification})^*\colon 7 \div 10) \sqcap \\ & (\triangle.\texttt{directs}.(\triangle.\texttt{employs}.\forall.\triangle.\texttt{directs})^*\colon \texttt{department})\end{aligned}$$

In generale, la forma canonica fattorizzata di un nome non é unica e puó includere fattori con lo stesso path. Allo scopo di ottenere un processo di fattorizzazione univoco, useremo una particolare forma fattorizzata definita come segue.

[1] In [Baa91] un automa di questo tipo viene definito "generalizzato" poiché le transizioni sono etichettate con parole di alfabeto e non solo con simboli dell'alfabeto stesso.

Definizione 11 (Forma Canonica Fattorizzata) *La* Forma Canonica Fattorizzata (CFF) *di un tipo S é una Forma Fattorizzata $F(S) = \bigsqcap_{i=1}^{n} f_i$ di S tale che, per ogni k, $1 \leq k \leq n$, valgono le seguenti proprietá:*

1. $\bigsqcap_{i \neq k} f_i \not\sqsubseteq_\sigma f_k$
2. *se $f_k = (p_k\colon S_k)$ é un fattore* non-esistenziale *e S_k é un tipo base, allora non esiste $j \neq k$ tale che $f_j = (p_j\colon S_j)$ é un fattore* non-esistenziale *con $p_j = p_k$ e $S_j \neq S_k$.*

Data una query `query`*, con* Φ(`query`) *denotiamo il suo insieme di fattori canonici.*

In altre parole, Φ(`query`) non contiene fattore ridondanti rispetto allo schema considerato senza regole, e i fattori con lo stesso path, caratterizzato da tipi base diversi, vengono congiunti. La congiunzione non viene effettuata se ci sono dei nomi di tipo (a meno che uno non sussuma l'altro), poiché sembra piú utile, per l'eliminazione, considerarli separatamente. Quando il path é esistenziale, la congiunzione non é consentita.

L'aspetto da mettere in evidenza è il fatto che la CFF è la forma fattorizzata che si ottiene partendo dallo schema canonico, il cui automa corrispondente é "quasi" deterministico, in quanto, ove possibile, le congiunzioni sono state risolte.

Consideriamo, ad esempio, la seguente interrogazione:

```
select *                                          (1)
from   CB_Department
where  category >= 6
and    administrator.qualification >= 6
```

Il fattore con path `administrator.qualification` non é compreso in $\Phi(1)$, essendo incluso nella definizione di `CBDepartment`. Se non consideriamo regole e classi virtuali, la Φ(`query`) rappresenta la riscrittura *minimale* equivalente, i.e. ha il minimo numero di fattori. Se vengono considerate anche le classi virtuali, la riscrittura non é minimale, dal momento che una classe virtuale puó sostituire un numero di fattori che sono implicati dalla sua definizione. Questo tipo di riscrittura é una forma di *answering queries using views*[BLR97].

Quando invece consideriamo le regole, l'introduzione di un nuovo fattore ridondante potrebbe far scattare alcune regole e determinare l'eliminazione di altri fattori.

Il controllo della ridondanza di un fattore rispetto ad uno schema con regole senza viste e rispetto alla espressione della query specificata inizialmente

é quello che avviene nella maggior parte dei lavori presenti in letteratura; in questo caso parleremo di *eliminabilità diretta*.

Definizione 12 (Eliminabilità Diretta) *Data una query* `query`*, diremo che un set di fattori $A \subset \Phi$(`query`) é* direttamente eliminabile *dalla query* `query` *se*

$$\bigsqcap_{f_i \in \Phi(\texttt{query}) \setminus A} f_i \simeq_\Sigma \texttt{query}$$

Un set di fattori è direttamente eliminabile da un tipo se è implicato dai rimanenti fattori come segue.

Proposizione 4 *Data una query* `query`*, A è direttamente eliminabile da* `query` sse

$$\bigsqcap_{f_i \in \Phi(\texttt{query}) \setminus A} f_i \sqsubseteq_\Sigma \bigsqcap_{f_j \in A} f_j$$

Sketch Proof 1 *L'implicazione se è immediata. Per l'implicazione solo se, sia Φ(`query`) $= \{f_1, \ldots, f_k, f_{k+1}, \ldots, f_n\}$ e sia $A = \{f_{k+1}, \ldots, f_n\}$. Se $f_1 \sqcap \ldots \sqcap f_k \not\sqsubseteq_\Sigma f_{k+1} \sqcap \ldots \sqcap f_n$ allora, $\exists$ un'istanza legale $\mathcal{I}$ di Σ ed un valore v tali che $v \in \mathcal{I}[f_1 \sqcap \ldots \sqcap f_k]$ e $v \notin \mathcal{I}[f_{k+1} \sqcap \ldots \sqcap f_n]$. Quindi, $f_1 \sqcap \ldots \sqcap f_k \sqcap f_{k+1} \sqcap \ldots \sqcap f_n (=$ `query`$) \not\simeq_\Sigma f_1 \sqcap \ldots \sqcap f_k$.*

Per esempio, consideriamo la seguente interrogazione:

```
select *
from   Manager
where  qualification = 10
and directs.category >= 4
```

(2)

Il corrispondente insieme di fattori canonici é:

$$\Phi(\texttt{query}) = \Big\{(\epsilon\colon \texttt{manager}), (\triangle.\texttt{qualification}\colon 10 \div 10), (\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 4 \div 7)$$

Considerando lo schema con regole completo Σ, vale la seguente sussunzione:

$$(\epsilon\colon \texttt{manager}) \sqcap (\triangle.\texttt{level}\colon 10 \div 10) \sqsubseteq_\Sigma (\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 4 \div 7)$$

quindi ($\triangle$.`directs`.$\triangle$.`category`$\colon 4 \div 7$) é direttamente eliminabile da `query`.

Tuttavia la eliminabilitá diretta costituisce solo una condizione sufficiente per affermare che un fattore é ininfluente per il risultato finale della query. Una definizione piú generale considera l'equivalenza tra tipi in Σ nel modo seguente.

Definizione 13 (Eliminabilitá) *Data una query* `query`*, sia* $\Phi(\texttt{query})$ *il suo set di fattori. Diremo che un set di fattori* $A \subset \Phi(\texttt{query})$ *puó essere* eliminato *dalla query* `query` *se esiste un nome* N*, tale che* `query` $\simeq_\Sigma N$ *e* $A \cap \Phi(N) = \emptyset$*.*

In altre parole, talvolta un fattore non é direttamente eliminabile, ma la sua eliminazione puó essere ottenuta attraverso l'introduzione di nuovi ridondanti fattori. In particolare, ció accade allorché lo schema include le *regole cicliche*.

In questo caso, possiamo considerare l'*espansione semantica* della query originale `query`, che genera il tipo equivalente a `query` contenente il massimo numero di fattori.

Proposizione 5 *Un set di fattori* $A \subset \Phi(\texttt{query})$ *puó essere eliminato dalla query* `query` sse *esiste un set di fattori* $B \subset \Phi(EXP(\texttt{query})) \setminus A$ *tali che*

$$\bigsqcap_{f_i \in B} f_i \sqsubseteq_\Sigma \bigsqcap_{f_j \in A} f_j$$

Consideriamo, a titolo di esempio la interrogazione `query`:

```
select *
from   Manager
where  salary =100
and    directs.category >= 6
```

il suo insieme di fattori canonici é:

$$\Phi(\texttt{query}) = \Big\{(\epsilon\colon \texttt{manager}), (\triangle.\texttt{salary}\colon 100 \div 100), (\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 6 \div 7)\Big\}$$

Ora, $(\epsilon\colon \texttt{manager}) \sqcap (\triangle.\texttt{salary}\colon 100 \div 100) \not\sqsubseteq_\Sigma (\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 6 \div 7)$
ma,

$$\begin{aligned}\Phi(EXP(\texttt{query})) \;=\; \Big\{&(\epsilon\colon \texttt{manager}), (\triangle.\texttt{salary}\colon 100 \div 100),\\ &(\triangle.\texttt{qualification}\colon 10 \div 10),\\ &(\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 6 \div 7)\Big\}\end{aligned}$$

e

$$\begin{aligned}(\epsilon\colon \texttt{manager}) \sqcap (\triangle.\texttt{salary}\colon 100 \div 100) \sqcap (\triangle.\texttt{qualification}\colon 10 \div 10) \;\sqsubseteq_\Sigma\\ (\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 6 \div 7)\end{aligned}$$

dunque $(\triangle.\texttt{directs}.\triangle.\texttt{category}\colon 5 \div 7)$ é eliminabile da `query`.

Si puó facilmente dimostrare che l'eliminabilitá diretta implica l'eliminabilitá, ma non viceversa.

5.2 Fattorizzazione in ODB-Tools

Come abbiamo avuto modo di anticipare, il software ODB-Tools fa propri i concetti sopra esposti ricorrendo alla fattorizzazione tanto degli schemi quanto delle interrogazioni.

Una parte cospicua di questo lavoro é stata dedicata alla trattazione dei moduli software che operano la fattorizzazione ed alla realizzazione delle modifiche che consentono ora, a questi stessi moduli, di acquisire e fattorizzare query e schemi ricorsivi.

Il procedimento che porta alla realizzazione della CFF di uno schema inserito dall'utente puó essere schematizzato come segue. Il primo passo consiste nella costruzione della forma canonica dello schema, che rappresenta dunque l'input vero e proprio del processo di fattorizzazione. Ogni classe dello schema in forma canonica viene poi considerata singolarmente allo scopo di realizzare la forma fattorizzata di ciascuna descrizione.

Fissata che sia una classe, il programma analizza i tipi che compaiono nella sua descrizione stessa processandoli in modo diverso a seconda che si tratti di tipi-base, classi dello schema originario, antecedenti e conseguenti di regole ecc., in base alla classificazione proposta in tabella 3.1.

In particolare, in presenza di nomi di tipi base o di tipi base, la procedura é immediata: si provvede ad aggiornare in modo opportuno il path corrente, quindi si aggiunge alla classe un fattore caratterizzato dal path aggiornato. La complessitá del metodo aumenta quando nella descrizione della classe compaiono classi base, classi virtuali, atomi fittizi, antecedenti o conseguenti di regole o nomi di tipo valore. Tutti questi casi vengono trattati in modo analogo.

In questi casi é opportuno stabilire, anzitutto, se il tipo che di volta in volta analizziamo sia stato inserito dall'utente o se invece si tratti di un nuovo tipo (o di un nuovo nome) che é stato introdotto dal programma per portare lo schema in forma canonica. In questo secondo caso, infatti, bisogna evitare di far comparire nella descrizione fattorizzata un nome che non compariva nello schema inserito dall'utente, inoltre é obbligatorio applicare ricorsivamente, all'interno delle descrizioni dei tipi incontrati, il medesimo procedimento.

Questo fatto mette in luce i problemi che possono intervenire allorché si tenti di fattorizzare una classe ciclica. É infatti sufficiente che un qualsivoglia nuovo tipo (o nuovo nome) che compare nella descrizione della classe abbia a sua volta una descrizione ciclica per determinare una condizione di stallo: il programma continua ricorsivamente ad esplorare la classe senza possibilitá di terminazione.

Si veda, ad esempio lo schema di tabella 5.1

```
virt Clerk =  MdlEmployee &  ^ [ worksin : Office ] ;
virt Office =  Department &  ^ [ enrolls : { Clerk } ] ;
virt MdlEmployee =  Employee &  ^ [ level : range 5 10 ] ;
prim CBDept =  Department &  ^ [ category : range 5 7 ,
                                 administrator : Manager ] ;
prim Manager =  Employee &  ^ [ level : range 8 10 ,
                                directs : Department ] ;
prim Employee =  ^ [ name : string , level : range 1 10 ,
                     worksin : Department ] ;
prim Department =  ^ [ dname : string , category : range 1 7 ,
                       enrolls : { Employee } ,
                       administrator : Employee ] ;
antev r_1a = Department & ^ [ category : range 3 +inf ] ;
consv r_1c = Department & ^ [ administrator : Manager ] .
```

Tabella 5.1: Esempio di schema con classi ricorsive

Le classi `Clerk` e `Office` sono coinvolte in una descrizione ciclica. Il modulo OCDL-Designer realizza la trasformazione dello schema in forma canonica (tabella 5.2) che, a sua volta, viene utilizzata per creare la CFF.

In tabella 5.2 osserviamo l'introduzione dei nuovi tipi e dei nuovi nomi che sono necessari per ottenere la descrizione in forma canonica. Inoltre, navigando la descrizione della classe primitiva `Clerk`, possiamo osservare il ciclo che riguarda il nuovo tipo `NewType1` (si vedano le descrizioni canoniche di `Clerk` e `NewName2`).

Un discorso analogo si potrebbe ripetere per la classe `Office` con il tipo `NewType3`. Si impone dunque la necessitá di individuare possibili fattori ciclici all'interno della descrizione fattorizzata di una classe allo scopo di prevenire situazioni di stallo. In corrispondenza di schemi come quello di tabella 5.2, infatti, se non vengono individuate le descrizioni ricorsive si tenterebbe di associare alle classi `Clerk` e `Office` un numero infinito di fattori.

Un secondo problema, subordinato tuttavia all'individuazione di eventuali ricorsioni, é costituito dalla rappresentazione formale delle ricorsioni stesse. Nel seguito verranno illustrate le soluzioni sviluppate in questo lavoro di tesi per ciascuno dei due problemi.

In ordine all'individuazione dei cicli nello schema canonico, é opportuno premettere che non é sufficiente un semplice controllo sintattico sui noi che vengono processati. In altre parole una classe puó contenere piú di un fattore con lo stesso nome di dominio. Ancora, con riferimento alla tabella 5.2, il fatto che il nome `NewType1` ricorra piú di una volta non é indice, di per sé, di


```
--------> -Employee  & ^ NewType2 ;
gs VQ4 = VQ4 : virt 0  1 , Employee : prim 1  0 ;
virt VQ3 0 =  Employee  & ^ [ head : VQ3 , level : range 6 10 ] ;
--------> -Employee  & ^ NewType1 ;
gs VQ3 = VQ3 : virt 0  1 , Employee : prim 1  0 , NewName1 : virt 0  0 ;
prim Manager 0 =  Employee  & ^ [ head : Manager ,
                  qualification : range 8 10 ] ;
--------> -Manager  & -Employee  & ^ NewType3 ;
gs Manager = Manager : prim 0  1 , Employee : prim 1  0 ,
             NewName2 : virt 0  0 ;
prim Employee 0 =  ^ [ head : Employee , name : string ,
                       qualification : range 1 10 ,
                       salary : range 1 100 ] ;
--------> -Employee  & ^ NewType4 ;
gs Employee = Employee : prim 0  1 ;
virt NewName1 1 = Employee  & VQ3 ;
--------> -Employee  & ^ NewType1 ;
gs NewName1 = NewName1 : virt 0  1 , VQ3 : virt 0  1 , E
              mployee : prim 1  0 ;
type NewType1 1 = [ head : NewName1 , level : range 6 10 ,
                    name : string , qualification : range 1 10 ,
                    salary : range 1 100 ] ;
--------> [ head : NewName1 , level : range 6 10 , name : string ,
            qualification : range 1 10 , salary : range 1 100 ] ;
gs NewType1 = NewType1 : type 0  1 , NewType4 : type 0  0 ;
type NewType2 1 = [ head : NewName1 , name : string ,
                    qualification : range 1 10 , salary : range 80 100 ] ;
--------> [ head : NewName1 , name : string ,
            qualification : range 1 10 , salary : range 80 100 ] ;
gs NewType2 = NewType2 : type 0  1 , NewType4 : type 0  0 ;
virt NewName2 1 = Employee  & Manager ;
--------> -Employee  & -Manager  & ^ NewType3 ;
gs NewName2 = NewName2 : virt 0  1 , Manager : prim 0  1 ,
              Employee : prim 1  0 ;
type NewType3 1 = [ head : NewName2 , name : string ,
                    qualification : range 8 10 , salary : range 1 100 ] ;
--------> [ head : NewName2 , name : string ,
            qualification : range 8 10 , salary : range 1 100 ] ;
gs NewType3 = NewType3 : type 0  1 , NewType4 : type 0  0 ;
type NewType4 1 = [ head : Employee , name : string ,
                    qualification : range 1 10 , salary : range 1 100 ] ;
--------> [ head : Employee , name : string ,
            qualification : range 1 10 , salary : range 1 100 ] ;
gs NewType4 = NewType4 : type 0  1 .
tipi_nuovi: 4
classi_nuove: 2
```

Tabella 5.2: Schema ciclico in forma canonica

un ciclo. Di conseguenza il nuovo programma di fattorizzazione é chiamato ad un controllo ben piú complesso.

5.2.1 Criterio di rilevazione dei cicli

Al solito, il metodo che consente di individuare una ricorsione nello schema canonico, si basa su procedure di *tracing*. In altre parole, ogni qualvolta viene calcolato ed aggiunto allo schema un fattore, vengono mantenute tutte le informazioni disponibili per evitare che, durante una delle successive applicazioni del metodo, lo stesso fattore venga nuovamente inserito.

Vediamo ora quali sono le informazioni che é necessario mantenere per individuare univocamente una situazione di ciclo. É giá stato detto che il nome del dominio del fattore da solo non é sufficiente ad indicare una ricorsione. Se lo stesso nome, cioé, compare due o piú volte durante il procedimento, esso puó essere caratterizzato da un path diverso.

Si capisce allora che, anche in questo caso, risulta fondamentale mantenere due informazioni distinte per fattore: il path e il nome del dominio. Dal momento che, come é stato detto, la condizione di ciclo viene innescata solo durante l'analisi ricorsiva di un nuovo tipo/nome, é utile anche mantenere l'informazione sulle eventuali equivalenze presenti nello schema. Accade infatti, in alcuni casi, che taluno di questi tipi/nomi sia equivalente ad uno dei tipi/nomi inseriti dall'utente.

Possiamo dunque riassumere il criterio di rilevazione dei cicli nel seguente modo. Stante che i tipi base vengono aggiunti allo schema fattorizzato in modo diretto, possiamo concentrarci sulle classi base, atomi fittizi, regole, classi virtuali nel seguente modo. In primo luogo va stabilito se il nome/tipo che analizziamo era presente o meno nello schema dell'utente. Nel primo caso, il nome viene memorizzato in una struttura opportuna che definiremo `olds`. Se il nome/tipo é invece nuovo, vengono considerati, se esistono, i suoi equivalenti nello schema di partenza. Ciascuno di questi equivalenti viene raffrontato con tutte le entry di `olds`: se vi é una corrispondenza il procedimento termina poiché é stato incontrato un nuovo nome/tipo equivalente ad uno inserito dall'utente e giá esplorato.

Ovviamente si puó verificare il caso in cui nessun nuovo nome/tipo abbia equivalenti nello schema originale. Ció non toglie, tuttavia, che lo stesso nome/tipo possa dare luogo a situazioni di stallo del procedimento. Vanno dunque memorizzati anche i nuovi nomi/tipi in un'apposita matrice che chiameremo `news`.

Insieme al nome nuovo memorizziamo in `news` anche il cammino con cui siamo giunti ad analizzare il nome stesso.

A questo punto, se durante un'iterazione successiva ricorre un nome/tipo

nuovo, questo verrá raffrontato con tutte le entry di `news`. Se il nome/tipo era giá stato considerato con una path analogo o quantomento ciclicamente ripetuto, il nome/tipo viene marcato come ciclico ed il procedimento si arresta.

In definitiva, se lo schema che intendiamo fattorizzare presenta descrizioni ricorsive, il procedimento continua iterativamente fin tanto che vi siano nuovi nomi/tipi da esplorare che non siano risultati equivalenti a nomi dello schema originale e che non siano giá stati incontrati con un path che si ripete. Qualora una di queste tre condizioni venga meno, il procedimento esce dalle ricorsioni e termina in un numero finito di passi.

Lo schema che si ottiene a partire dalla forma canonica di tabella 5.2 viene riportato in tabella 5.3

```
<CLASS_TYPE>
VQ4 4 0
void 11 20001 Employee 0
head 4 20001 VQ3 0
name 52 20001 string 0
qualification 100031 20001 1 0
qualification 100032 20001 10 0
salary 100031 20001 80 0
salary 100032 20001 100 0
;
VQ3 4 0
void 11 20001 Employee 0
head 4 20001 VQ3 0
level 100031 20001 6 0
level 100032 20001 10 0
name 52 20001 string 0
qualification 100031 20001 1 0
qualification 100032 20001 10 0
salary 100031 20001 1 0
salary 100032 20001 100 0
;
Manager 3 0
void 11 20001 Employee 0
head 3 20001 Manager 0
name 52 20001 string 0
qualification 100031 20001 8 0
qualification 100032 20001 10 0
salary 100031 20001 1 0
salary 100032 20001 100 0
;
Employee 3 0
head 3 20001 Employee 0
name 52 20001 string 0
qualification 100031 20001 1 0
qualification 100032 20001 10 0
salary 100031 20001 1 0
salary 100032 20001 100 0
</CLASS_TYPE>
```

Tabella 5.3: Forma canonica fattorizzata (CFF)

Analizzando la classe `Clerk` fattorizzata vediamo che il ciclo viene risolto interrompendo il procedimento allorché si incontra il nome `Office`. La classe `Office`, a sua volta termina con un domino `Clerk`. Analizzando a fondo il procedimento si osserverebbe infatti l'equivalenza di `NewName1` con la classe primitiva `Office`. Analogamente si vede dalla forma canonica che `Clerk` é equivalente a `NewName2`. Coerentemente con quanto detto in precedenza, quindi, il metodo si arresta in corrispondenza delle equivalenze.

Per quanto riguarda la fattorizzazione dei risultati intermedi e finale dell'ottimizzazione semantica il metodo é del tutto analogo. L'unica differenza sta nel fatto che le query intermedie e la query finale ottimizzata vengono rappresentate ciascuna attraverso una classe virtuale. In questa fase sará quindi sufficiente applicare il metodo descritto alla sola classe presente.

Per aumentare la leggibilitá dei risultati l'ultima versione del software prevede di scrivere, al termine dell'ottimizzazione, un file contenente i fattori della query finale ottimizzata scritti con il formalismo consueto (file `.oql.ft`).

5.3 Chiusura transitiva

Nelle sezioni precedenti é stato piú volte sottolineato come la chiusura transitiva di un cammino permetta di descrivere situazioni cicliche.

In questo lavoro é stato affiancato al procedimento di fattorizzazione delle classi cicliche sopra descritto un metodo di riscrittura della CFF dotato della notazione di chiusura transitiva.

Il calcolo della chiusura transitiva dello schema utilizza, nel programma realizzato, lo schema in forma fattorizzata. In particolare, dall'analisi del campo *path* di ciascun fattore di una classe, viene stabilito se quest'ultimo é ricorsivo. In caso affermativo il path viene rielaborato e riscritto con l'operatore $()^*$ che denota la chiusura transitiva del path stesso.

Da un punto di vista piú strettamente operativo va sottolineato il fatto che il software crea ex novo un file con questa particolare notazione. Ciascuna classe dello schema originale é caratterizzata da un descrizione fattorizzata con la seguente sintassi:

$$path \quad : \quad nome\ del\ dominio$$

ove, sia il campo *path*, sia il campo *nome del dominio*, mantengono il consueto significato.

Con la parola chiave *void* si intende il cammino vuoto.

A titolo di esempio viene presentata, in tabella 5.4 il nuovo formalismo ottenuto a partire dai fattori di tabella 5.3

```
<CLASS>
;
VQ4
epsilon : Employee
head : VQ3
name : string
qualification : range 1 10
salary : range 80 100
;
VQ3
epsilon : Employee
(head)* : VQ3
(head)*.level : range 6 10
(head)*.name : string
(head)*.qualification : range 1 10
(head)*.salary : range 1 100
;
Manager
epsilon : Employee
(head)* : Manager
(head)*.name : string
(head)*.qualification : range 8 10
(head)*.salary : range 1 100
;
Employee
epsilon : Employee
(head)* : Employee
(head)*.name : string
(head)*.qualification : range 1 10
(head)*.salary : range 1 100
</CLASS_TYPE>
```

Tabella 5.4: Forma canonica fattorizzata (CFF) - nuovo formalismo

Si puó osservare immediatamente che, nello schema proposto, non viene introdotta la chiusura transitiva per nessun attributo: nessuna classe presentava infatti cammini di tipo ciclico.

quello ottenuto finora non é ancora, in effetti, la CFF in quanto i fattori non rispettano la prima condizione della definizione 11.

Ad esempio, in tabella 5.4, la classe `manager` ha sia il fattore

$$(\epsilon\text{: } \texttt{employee})$$

sia tutti i fattori che derivano direttamente dalla classe `employee` come

$$(\texttt{name}\text{: } \texttt{string})$$

pertanto possiamo scrivere:

$$(\epsilon\text{: } \texttt{employee}) \sqsubseteq (\texttt{name}\text{: } \texttt{string})$$

Si é deciso di riportare comunque questo risultato in output per compatibilitá con il modulo di visualizzazione delle interrogazioni.

D'altra parte il passaggio dal risultato ottenuto alla CFF é immediato: consiste infatti in un controllo sintattico tra i fattori delle diverse classi.

Capitolo 6

Sessione di lavoro con ODB-Tools

Questa sezione é interamente dedicata all'analisi di una sessione di lavoro del software ODB-Tools.

Saranno presentate nel dettaglio quelle funzionalitá che, nel corso di questo lavoro, sono state oggetto delle piú rilevanti innovazioni.

Innanzi tutto premettiamo che,durante le fasi di realizzazione e di testing del programma, é stato utilizzato un elaboratore SUN SPARCSTATION-20 con un sistema operativo SOLARIS release 2.5.

La sessione di lavoro si articola in alcuni passaggi notevoli che analizzeremo e che possono essere sintetizzati come segue:

- *introduzione dello schema con regole*
- *trasformazione dello schema in forma canonica*
- *realizzazione della forma fattorizzata (CFF)*
- *trasposizione della CFF nel nuovo formalismo con chiusura transitiva*

Questi primi quattro punti sono realizzati dal componente software OLCD-Designer che consente dunque acquisire lo schema e trasformarlo opportunamente per la successiva ottimizzazione delle interrogazioni.

Il programma ODB-QOptimizer, a questo punto, acquisisce la query e realizza le seguenti funzioni:

- *verifica della coerenza dell'interrogazione*
- *calcolo dell'espansione semantica*
- *fattorizzazione dei risultati*

- *riscrittura del risultato dell'ottimizzazione secondo il nuovo formalismo*

6.1 Primo esempio di ottimizzazione

Come primo esempio intendiamo considerare uno schema con regole non ricorsivo. In queste condizioni, é bene puntualizzarlo, ODB-Tools funzionava correttamente anche prima delle modifiche apportate in questa sede (vedi capitolo 3). L'unica differenza risiede nella riscrittura di schema canonico e query fattorizzati secondo il nuovo formalismo. Cionondimeno questo esempio é stato di fondamentale importanza nelle fasi di progetto e sviluppo della nuova versione essendo stato assunto come benchmark delle prestazioni del software. In altre parole, ogni variazione apportata é stata testata utilizzando questo esempio come input.

6.1.1 Acquisizione dello schema

Lo schema con regole che consideriamo é riportato, nella sintassi OCDL, di seguito. Si possono osservare, per ognuna delle cinque regole di integritá presenti, un antecedente e un conseguente. Questo schema é contenuto all'interno di un file con estensione `.sc` ed é l'unico parametro fornito al modulo OCDL-Designer.

```
type t_stock = [  item : Material , qty : range 10 300 ] ;
type t_stock = t_stock ;
prim SStorage =  Storage ;
prim Storage =  ^ [ name : string , category : string ,
                    managed_by : Manager , s
tock : { t_stock } ] ;
prim TManager =  Manager &  ^ [ level : range 8 12 ] ;
prim Manager =  ^ [ name : string , salary : range 40000 100000 ,
                    level : range
1 15 ] ;
prim SMaterial =  Material ;
prim Material =  ^ [ name : string , risk : integer ,
                     feature : { string } ] ;
antev rule5a = Storage & ^ [ stock : { [ qty : range 10 50 ] } ] ;
consv rule5c = Storage & ^ [ category : vstring "A2" ] ;
antev rule4a = Storage & ^ [ stock : { [ item : SMaterial ] } ] ;
consv rule4c = Storage & SStorage ;
antev rule3a = Storage & ^ [ category : vstring "B4" ] ;
consv rule3c = Storage & ^ [ managed_by : TManager ] ;
antev rule2a = Material & ^ [ risk : range 10 +inf ] ;
consv rule2c = Material & SMaterial ;
antev rule1a = Manager & ^ [ level : range 5 10 ] ;
consv rule1c = Manager & ^ [ salary : range 40000 60000 ] .
```

Il primo passo, come abbiamo giá avuto modo di rilevare, é costituito dalla verifica della coerenza dello schema e dal calcolo della forma canonica. Se lo schema contenesse incoerenze il programma terminerebbe anticipatamente generando un opportuno messaggio di errore. Se, come in questo caso, lo schema non presenta incoerenze si passa alla generazione della forma canonica dello schema. Per risolvere le congiunzioni, il programma introduce alcuni nuovi nomi e nuovi tipi verificando, ad ogni iterazione, se ad una determinata descrizione non corrisponda giá un nome dello schema ovvero un nome nuovo precedentemente inserito.

La forma canonica dello schema viene mantenuta in memoria all'interno di un'opportuna struttura dati e, contemporaneamente, viene scritta sul file `.fc`. Con riferimento allo schema precedente abbiamo il seguente schema canonico:

```
type t_stock 0 =  [ item : Material , qty : range 10 300 ] ;
--------> [ item : Material , qty : range 10 300 ] ;
gs t_stock = t_stock : type 0  1 , t_stock : type 0  1 ;
type t_stock 0 =  t_stock ;
--------> [ item : Material , qty : range 10 300 ] ;
gs t_stock = t_stock : type 0  1 , t_stock : type 0  1 ;
prim SStorage 0 =  Storage ;
--------> -SStorage  & -Storage  & ^ NewType2 ;
gs SStorage = SStorage : prim 0  1 , Storage : prim 1  0 ,
            rule4c : consv 0  1 ;
prim Storage 0 =  ^ [ category : string , managed_by : Manager ,
                  name : string , stock : { t_stock } ] ;
--------> -Storage  & ^ NewType2 ;
gs Storage = Storage : prim 0  1 ;
prim TManager 0 =  Manager  & ^ [ level : range 8 12 ] ;
--------> -TManager  & -Manager  & ^ NewType3 ;
gs TManager = TManager : prim 0  1 , Manager : prim 1  0 ;
prim Manager 0 =  ^ [ level : range 1 15 , name : string ,
                  salary : range 40000 100000 ] ;
--------> -Manager  & ^ NewType4 ;
gs Manager = Manager : prim 0  1 ;
prim SMaterial 0 =  Material ;
--------> -SMaterial  & -Material  & ^ NewType6 ;
gs SMaterial = SMaterial : prim 0  1 , Material : prim 1  0 ,
prim Material 0 =  ^ [ feature : { string } , name : string ,
                   risk : integer ] ;
--------> -Material  & ^ NewType6 ;
gs Material = Material : prim 0  1 ;
antev rule5a 0 =  Storage  & ^ [ stock : { [ qty : range 10 50 ] } ] ;
--------> -Storage  & ^ NewType9 ;
gs rule5a = rule5a : antev 0  1 , Storage : prim 1  0 ;
consv rule5c 0 =  Storage  & ^ [ category : vstring "A2" ] ;
```



```
--------> -Storage  & ^ NewType10 ;
gs rule5c = rule5c : consv 0  1 , Storage : prim 1  0 ;
antev rule4a 0 =  Storage  & ^ [ stock : { [ item : SMaterial ] } ] ;
--------> -Storage  & ^ NewType13 ;
gs rule4a = rule4a : antev 0  1 , Storage : prim 1  0 ;
consv rule4c 0 =  Storage  & SStorage ;
--------> -Storage  & -SStorage  & ^ NewType2 ;
gs rule4c = rule4c : consv 0  1 , SStorage : prim 0  1 ,
           Storage : prim 1  0 ;
antev rule3a 0 =  Storage  & ^ [ category : vstring "B4" ] ;
--------> -Storage  & ^ NewType14 ;
gs rule3a = rule3a : antev 0  1 , Storage : prim 1  0 ;
consv rule3c 0 =  Storage  & ^ [ managed_by : TManager ] ;
--------> -Storage  & ^ NewType15 ;
gs rule3c = rule3c : consv 0  1 , Storage : prim 1  0 ;
antev rule2a 0 =  Material  & ^ [ risk : range 10 +inf ] ;
--------> -Material  & ^ NewType16 ;
gs rule2a = rule2a : antev 0  1 , Material : prim 1  0 ;
consv rule2c 0 =  Material  & SMaterial ;
--------> -Material  & -SMaterial  & ^ NewType6 ;
gs rule2c = rule2c : consv 0  1 , SMaterial : prim 0  1 ,
           Material : prim 1  0 ;
antev rule1a 0 =  Manager  & ^ [ level : range 5 10 ] ;
--------> -Manager  & ^ NewType17 ;
gs rule1a = rule1a : antev 0  1 , Manager : prim 1  0 ;
consv rule1c 0 =  Manager  & ^ [ salary : range 40000 60000 ] ;
--------> -Manager  & ^ NewType18 ;
gs rule1c = rule1c : consv 0  1 , Manager : prim 1  0 ;
type NewType1 1 = { t_stock } ;
--------> { t_stock } ;
gs NewType1 = NewType1 : type 0  1 ;
type NewType2 1 = [ category : string , managed_by : Manager ,
                  name : string , stock : NewType1 ] ;
--------> [ category : string , managed_by : Manager , name :
           string , stock : NewType1 ] ;
gs NewType2 = NewType2 : type 0  1 ;
type NewType3 1 = [ level : range 8 12 , name : string ,
                  salary : range 40000 100000 ] ;
--------> [ level : range 8 12 , name : string ,
           salary : range 40000 100000 ] ;
gs NewType3 = NewType3 : type 0  1 , NewType4 : type 0  0 ;
type NewType4 1 = [ level : range 1 15 , name : string ,
                  salary : range 40000 100000 ] ;
--------> [ level : range 1 15 , name : string ,
           salary : range 40000 100000 ] ;
gs NewType4 = NewType4 : type 0  1 ;
type NewType5 1 = { string } ;
--------> { string } ;
gs NewType5 = NewType5 : type 0  1 ;
```

rule

```
type NewType6 1 = [ feature : NewType5 , name : string ,
                  risk : integer ] ;
--------> [ feature : NewType5 , name : string , risk : integer ] ;
gs NewType6 = NewType6 : type 0  1 ;
type NewType7 1 = [ qty : range 10 50 ]  & t_stock ;
--------> [ item : Material , qty : range 10 50 ] ;
gs NewType7 = NewType7 : type 0  1 , t_stock : type 1  0 ,
              t_stock : type 1  0 ;
type NewType8 1 = { NewType7 } ;
--------> { NewType7 } ;
gs NewType8 = NewType8 : type 0  1 , NewType1 : type 0  0 ;
type NewType9 1 = [ category : string , managed_by : Manager ,
                    name : string , stock : NewType8 ] ;
--------> [ category : string , managed_by : Manager ,
           name : string , stock : NewType8 ] ;
gs NewType9 = NewType9 : type 0  1 , NewType2 : type 0  0 ;
type NewType10 1 = [ category : vstring "A2" , managed_by : Manager ,
                     name : string , stock : NewType1 ] ;
--------> [ category : vstring "A2" , managed_by : Manager ,
           name : string , stock : NewType1 ] ;
gs NewType10 = NewType10 : type 0  1 , NewType2 : type 0  0 ;
type NewType11 1 = [ item : SMaterial ]  & t_stock ;
--------> [ item : SMaterial , qty : range 10 300 ] ;
gs NewType11 = NewType11 : type 0  1 , t_stock : type 1  0 ,
               t_stock : type 1  0 ;
type NewType12 1 = { NewType11 } ;
--------> { NewType11 } ;
gs NewType12 = NewType12 : type 0  1 , NewType1 : type 0  0 ;
type NewType13 1 = [ category : string , managed_by : Manager ,
                     name : string , stock : NewType12 ] ;
--------> [ category : string , managed_by : Manager , name : string ,
           stock : NewType12 ] ;
gs NewType13 = NewType13 : type 0  1 , NewType2 : type 0  0 ;
type NewType14 1 = [ category : vstring "B4" , managed_by : Manager ,
                     name : string , stock : NewType1 ] ;
--------> [ category : vstring "B4" , managed_by : Manager ,
           name : string , stock : NewType1 ] ;
gs NewType14 = NewType14 : type 0  1 , NewType2 : type 0  0 ;
type NewType15 1 = [ category : string , managed_by : TManager ,
                     name : string , stock : NewType1 ] ;
--------> [ category : string , managed_by : TManager ,
           name : string , stock : NewType1 ] ;
gs NewType15 = NewType15 : type 0  1 , NewType2 : type 0  0 ;
type NewType16 1 = [ feature : NewType5 , name : string ,
                     risk : range 10 +inf ] ;
--------> [ feature : NewType5 , name : string , risk : range 10 +inf ] ;
gs NewType16 = NewType16 : type 0  1 , NewType6 : type 0  0 ;
type NewType17 1 = [ level : range 5 10 , name : string ,
                     salary : range 40000 100000 ] ;
```



```
--------> [ level : range 5 10 , name : string ,
           salary : range 40000 100000 ] ;
gs NewType17 = NewType17 : type 0  1 , NewType4 : type 0  0 ;
type NewType18 1 = [ level : range 1 15 , name : string ,
                     salary : range 40000 60000 ] ;
--------> [ level : range 1 15 , name : string ,
           salary : range 40000 60000 ] ;
gs NewType18 = NewType18 : type 0  1 , NewType4 : type 0  0 .
tipi_nuovi: 18
classi_nuove: 0
```

Per ciascuno dei tipi che compare nello schema viene riportata la descrizione, la descrizione in forma canonica e l'insieme GS specificando, per ogni elemento che vi compare, se si tratta di un nuovo nome (o tipo) e se questo é equivalente o meno al tipo in questione.

Partendo dalla struttura dati che, a run time, mantiene in memoria la descrizione di tabella viene realizzata la forma fattorizzata. Da un punto di vista teorico la forma canonica fattorizzata é giá stata approfondita nel capitolo precedente. Ci limitiamo a sottolineare che, in assenza di ricorsioni, i controlli previsti dalla nuova versione non sortiscono alcun effetto indesiderato. Anche per la forma fattorizzata il programma mantiene in memoria una struttura dati opportuna e, contemporaneamente, ne copia il contenuto sul file `.ft` che, per comoditá, viene riportato di seguito:

```
<CLASS_TYPE>
t_stock 2 0
item 3 20001 Material 0
qty 100031 20001 10 0
qty 100032 20001 300 0
;
t_stock 2 0
item 3 20001 Material 0
qty 100031 20001 10 0
qty 100032 20001 300 0
;
SStorage 3 0
void 11 20001 Storage 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
;
Storage 3 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
```

```
;
TManager 3 0
void 11 20001 Manager 0
/\.level 100031 20001 8 0
/\.level 100032 20001 12 0
/\.name 52 20001 string 0
/\.salary 100031 20001 40000 0
/\.salary 100032 20001 100000 0
;
Manager 3 0
/\.level 100031 20001 1 0
/\.level 100032 20001 15 0
/\.name 52 20001 string 0
/\.salary 100031 20001 40000 0
/\.salary 100032 20001 100000 0
;
SMaterial 3 0
void 11 20001 Material 0
/\.feature.all 52 20001 string 6
/\.name 52 20001 string 0
/\.risk 51 20001 integer 0
;
Material 3 0
/\.feature.all 52 20001 string 6
/\.name 52 20001 string 0
/\.risk 51 20001 integer 0
;
rule5a 14 0
void 11 20001 Storage 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all.item 3 20001 Material 6
/\.stock.all.qty 100031 20001 10 6
/\.stock.all.qty 100032 20001 50 6
;
rule5c 15 0
void 11 20001 Storage 0
/\.category 55 20001 "A2" 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
;
rule4a 14 0
void 11 20001 Storage 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all.item 3 20001 SMaterial 6
```



```
/\.stock.all.qty 100031 20001 10 6
/\.stock.all.qty 100032 20001 300 6
;
rule4c 15 0
void 11 20001 Storage 0
void 11 20001 SStorage 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
;
rule3a 14 0
void 11 20001 Storage 0
/\.category 55 20001 "B4" 0
/\.managed_by 3 20001 Manager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
;
rule3c 15 0
void 11 20001 Storage 0
/\.category 52 20001 string 0
/\.managed_by 3 20001 TManager 0
/\.name 52 20001 string 0
/\.stock.all 2 20001 t_stock 6
;
rule2a 14 0
void 11 20001 Material 0
/\.feature.all 52 20001 string 6
/\.name 52 20001 string 0
/\.risk 100031 20001 10 0
/\.risk 100032 20001 2147483647 0
;
rule2c 15 0
void 11 20001 Material 0
void 11 20001 SMaterial 0
/\.feature.all 52 20001 string 6
/\.name 52 20001 string 0
/\.risk 51 20001 integer 0
;
rule1a 14 0
void 11 20001 Manager 0
/\.level 100031 20001 5 0
/\.level 100032 20001 10 0
/\.name 52 20001 string 0
/\.salary 100031 20001 40000 0
/\.salary 100032 20001 100000 0
;
rule1c 15 0
void 11 20001 Manager 0
```

```
/\.level 100031 20001 1 0
/\.level 100032 20001 15 0
/\.name 52 20001 string 0
/\.salary 100031 20001 40000 0
/\.salary 100032 20001 60000 0
</CLASS_TYPE>
```

La forma fattorizzata viene usata dal software per riscrivere la query finale ottimizzata da visualizzare. Si capisce perché, allora, non vi compaia la descrizione dei nuovi tipi introdotti per generare lo schema canonico: questi infatti non avranno mai modo di comparire nell'output visualizzato. Come abbiamo giá anticipato, le funzionalitá ed i risultati che abbiamo fin qui esposto, pur essendo stati ottenuti con la nuova versione di ODB-Tools erano ottenibili anche in precedenza. La prima differenza sta nel prossimi risultato che esporremo. Si tratta delle forma fattorizzata espressa tramite il nuovo formalismo.

```
<CLASS>
;
t_stock
item : Material
qty : range 10 300
;
t_stock
item : Material
qty : range 10 300
;
SStorage
epsilon : Storage
category : string
managed_by : Manager
name : string
stock.all : t_stock
;
Storage
epsilon : Storage
category : string
managed_by : Manager
name : string
stock.all : t_stock
;
TManager
epsilon : Manager
level : range 8 12
name : string
salary : range 40000 100000
;
```



```
Manager
epsilon : Manager
level : range 1 15
name : string
salary : range 40000 100000
;
SMaterial
epsilon : Material
feature.all : string
name : string
risk : integer
;
Material
epsilon : Material
feature.all : string
name : string
risk : integer
;
rule5a
epsilon : Storage
category : string
managed_by : Manager
name : string
stock.all.item : Material
stock.all.qty : range 10 50
;
rule5c
epsilon : Storage
category : "A2"
managed_by : Manager
name : string
stock.all : t_stock
;
rule4a
epsilon : Storage
category : string
managed_by : Manager
name : string
stock.all.item : SMaterial
stock.all.qty : range 10 300
;
rule4c
epsilon : Storage
epsilon : SStorage
category : string
managed_by : Manager
name : string
stock.all : t_stock
;
```

```
rule3a
epsilon : Storage
category : "B4"
managed_by : Manager
name : string
stock.all : t_stock
;
rule3c
epsilon : Storage
category : string
managed_by : TManager
name : string
stock.all : t_stock
;
rule2a
epsilon : Material
feature.all : string
name : string
risk : range 10 2147483647
;
rule2c
epsilon : Material
epsilon : SMaterial
feature.all : string
name : string
risk : integer
;
rule1a
epsilon : Manager
level : range 5 10
name : string
salary : range 40000 100000
;
rule1c
epsilon : Manager
level : range 1 15
name : string
salary : range 40000 60000
</CLASS_TYPE>
```

Poiché non si tratta di uno schema ciclico, non abbiamo ovviamente modo di osservare la chiusura transitiva di nessun attributo.

6.1.2 Ottimizzazione semantica di una query

Con il calcolo dei fattori e la loro riscrittura esposto termina la fase di *Designer* di ODB-Tools. A questo punto, per realizzare l'ottimizzazione semantica di un'interrogazione sullo schema che abbiamo introdotto, occorre creare un

file `.oql` che contenga la query scritta appunto in linguaggio OQL. Il modulo ODB-QOptimizer acquisisce la query stessa e carica, a partire dai file che abbiamo descritto, tutte le informazioni elaborate dal Designer memorizzandole nelle rispettive strutture dati.

Apprestiamoci dunque ad analizzare il funzionamento concreto dell'ottimizzatore introducendo un'interrogazione di esempio:

Q_1 : "seleziona tutti i magazzini di categoria uguale a 'B4' che siano diretti da un manager di livello inferiore a dieci"
Formalizzando l'interrogazione con il linguaggio OQL si ottiene:

```
select *
from Storage
where category = "B4"
and   managed_by.level < 10
```

Alla luce dell'algoritmo generale di espansione semantica schematizziamo i passi che portano al risultato finale dell'ottimizzazione.

In primo luogo, sulla base degli algoritmi presentati nella prima appendice, viene valutata la coerenza dell'interrogazione e se ne calcola la forma canonica. La coerenza della query prescelta é facilmente verificabile mentre omettiamo, per ragioni di comoditá, lo schema canonico $\nu(Q_1)$ che viene generato.

In corrispondenza della prima iterazione osserviamo che la query inserita contiene l'antecedente della terza regola. Utilizzando il formalismo consueto avremo quindi:

$$\nu^1(Q_1) \;=\; \nu(Q_1) \sqcap \nu(rule3c)$$

Essendo la regola 3, a questo livello, l'unica regola applicabile la prima iterazione termina con la congiunzione che abbiamo mostrato sopra.

In corrispondenza della seconda iterazione viene nuovamente calcolata la forma canonica della query parziale. Si puó inoltre osservare che risulta ora applicabile la regola 1. Varrá dunque:

$$\nu^2(Q_1) \;=\; \nu^1(Q_1) \sqcap \nu(rule1c)$$

In corrispondenza della terza iterazione dell'algoritmo non esistono ulteriori regole applicabili e il procedimento termina con la forma ottimizzata finale $\nu^3(Q_1)$.

La query finale ottimizzata in forma canonica $\nu^3(Q_1)$ viene opportunamente processata per realizzarne la fattorizzazione attraverso i due formalismi descritti. Il primo viene memorizzato in un file di nuova introduzione `.oql.ft`:

```
void 11 20001 Storage 0
category 55 20001 "B4" 0
managed_by 11 20001 Manager 0
managed_by 11 20001 TManager 0
managed_by.level 100031 20001 8 0
managed_by.level 100032 20001 9 0
managed_by.name 52 20001 string 0
managed_by.salary 100031 20001 40000 0
managed_by.salary 100032 20001 60000 0
name 52 20001 string 0
stock.all 2 20001 t_stock 6
```

Il file che mantiene i fattori nel nuovo formalismo é anch'esso di nuova introduzione ed ha estensione `.oql.fst`:

```
epsilon : Storage
category : "B4"
managed_by : Manager
managed_by : TManager
managed_by.level : range 8 9
managed_by.name : string
managed_by.salary : range 40000 60000
name : string
stock.all : t_stock
```

Con la scrittura dei fattori termina la fase di optimizer vera e propria. Si tratta, a questo punto, di visualizzare i risultati. Esistono, a questo scopo, alcune procedure che non sono state oggetto di analisi nel corso di questo lavoro. Naturalmente queste procedure si basano sugli schemi acquisiti durante la fase di designer e dei risultati dell'espansione semantica.

In definitiva il risultato, in OQL, dell'interrogazione Q_1 sará il seguente:

```
select *
from Storage as S
where S.category = "B4" and
      S.managed_by in ( select T
                        from TManager as T
                        where T.level < 10 and
                              T.salary < 60001)
```


6.2 Secondo esempio di ottimizzazione

In questa sezione intendiamo descrivere l'ottimizzazione semantica di una interrogazione ricorsiva allo scopo di analizzare nel dettaglio tutte le nuove funzionalitá di ODB-Tools.

In particolare faremo riferimento al caso giá presentato a pagina 53. In quella sede l'esempio notevole considerato ci ha premesso di mettere in evidenza i limiti della precedente versione dell'ottimizzatore. In modo del tutto analogo vogliamo qui delineare le nuove proprietá acquisite da ODB-QOptimizer al termine di questo lavoro.

6.2.1 Acquisizione dello schema

In primo luogo occorre tradurre lo schema di pagina 53 secondo il formalismo OLCD. Avremo dunque lo schema seguente:

```
prim alfa = ^ [ a : integer , b : integer , c : alfa ] ;
antev rule2a = alfa & ^ [ a : range 41 +inf ] & ^ [ b : range 33 +inf ] ;
consv rule2c = alfa & ^ [ c : ^ [ a : range 75 +inf ] ] ;
antev rule1a = alfa & ^ [ a : range 20 +inf ] & ^ [ b : range 30 +inf ]  ;
consv rule1c = alfa & ^ [ c : ^ [ b : range 60 +inf ] ] .
```

Osserviamo un'unica classe primitiva e ciclica (alfa) e le due regole cicliche giá presentate. Ancora una volta é necessario realizzare la forma canonica dello schema. Avremo quindi:

```
prim alfa 0 =  ^ [ a : integer , b : integer , c : alfa ] ;
--------> -alfa  & ^ NewType1 ;
gs alfa = alfa : prim 0  1 ;
antev rule2a 0 =  alfa  & ^ [ a : range 41 +inf ]  & ^ [ b : range 33 +inf ] ;
--------> -alfa  & ^ NewType2 ;
gs rule2a = rule2a : antev 0  1 , alfa : prim 0  0 , rule1a : antev 1  0 ;
consv rule2c 0 =  alfa  & ^ [ c : ^ [ a : range 75 +inf ] ] ;
--------> -alfa  & ^ NewType4 ;
gs rule2c = rule2c : consv 0  1 , alfa : prim 1  0 ;
antev rule1a 0 =  alfa  & ^ [ a : range 20 +inf ]
                        & ^ [ b : range 30 +inf ] ;
--------> -alfa  & ^ NewType5 ;
gs rule1a = rule1a : antev 0  1 , alfa : prim 1  0 ;
consv rule1c 0 =  alfa  & ^ [ c : ^ [ b : range 60 +inf ] ] ;
--------> -alfa  & ^ NewType7 ;
gs rule1c = rule1c : consv 0  1 , alfa : prim 1  0 ;
type NewType1 1 = [ a : integer , b : integer , c : alfa ] ;
--------> [ a : integer , b : integer , c : alfa ] ;
gs NewType1 = NewType1 : type 0  1 ;
type NewType2 1 = [ a : range 41 +inf , b : range 33 +inf , c : alfa ] ;
```

```
--------> [ a : range 41 +inf , b : range 33 +inf , c : alfa ] ;
gs NewType2 = NewType2 : type 0  1 , NewType1 : type 0  0 ,
              NewType5 : type 0  0 ;
virt NewName1 1 = alfa  & ^ [ a : range 75 +inf ] ;
--------> -alfa  & ^ NewType3 ;
gs NewName1 = NewName1 : virt 0  1 , alfa : prim 1  0 ;
type NewType3 1 = [ a : range 75 +inf , b : integer , c : alfa ] ;
--------> [ a : range 75 +inf , b : integer , c : alfa ] ;
gs NewType3 = NewType3 : type 0  1 , NewType1 : type 0  0 ;
type NewType4 1 = [ a : integer , b : integer , c : NewName1 ] ;
--------> [ a : integer , b : integer , c : NewName1 ] ;
gs NewType4 = NewType4 : type 0  1 , NewType1 : type 0  0 ;
type NewType5 1 = [ a : range 20 +inf , b : range 30 +inf , c : alfa ] :
--------> [ a : range 20 +inf , b : range 30 +inf , c : alfa ] ;
gs NewType5 = NewType5 : type 0  1 , NewType1 : type 0  0 ;
virt NewName2 1 = alfa  & ^ [ b : range 60 +inf ] ;
--------> -alfa  & ^ NewType6 ;
gs NewName2 = NewName2 : virt 0  1 , alfa : prim 1  0 ;
type NewType6 1 = [ a : integer , b : range 60 +inf , c : alfa ] ;
--------> [ a : integer , b : range 60 +inf , c : alfa ] ;
gs NewType6 = NewType6 : type 0  1 , NewType1 : type 0  0 ;
type NewType7 1 = [ a : integer , b : integer , c : NewName2 ] ;
--------> [ a : integer , b : integer , c : NewName2 ] ;
gs NewType7 = NewType7 : type 0  1 , NewType1 : type 0  0 .
tipi_nuovi: 7
classi_nuove: 2
```

Il controllo sui nuovi nomi effettuato dall'algoritmo A.2 previene condizioni di stallo durante la realizzazione dello schema canonico.

La forma fattorizzata dello schema non presenta novitá di rilievo rispetto al caso precedente.

```
<CLASS_TYPE>
alfa 3 0
a 51 20001 integer 0
b 51 20001 integer 0
c 3 20001 alfa 0
;
rule2a 14 0
void 11 20001 alfa 0
a 100031 20001 41 0
a 100032 20001 2147483647 0
b 100031 20001 33 0
b 100032 20001 2147483647 0
c 3 20001 alfa 0
;
rule2c 15 0
void 11 20001 alfa 0
a 51 20001 integer 0
```



```
b 51 20001 integer 0
c 11 20001 alfa 0
c.a 100031 20001 75 0
c.a 100032 20001 2147483647 0
c.b 51 20001 integer 0
c.c 3 20001 alfa 0
;
rule1a 14 0
void 11 20001 alfa 0
a 100031 20001 20 0
a 100032 20001 2147483647 0
b 100031 20001 30 0
b 100032 20001 2147483647 0
c 3 20001 alfa 0
;
rule1c 15 0
void 11 20001 alfa 0
a 51 20001 integer 0
b 51 20001 integer 0
c 11 20001 alfa 0
c.a 51 20001 integer 0
c.b 100031 20001 60 0
c.b 100032 20001 2147483647 0
c.c 3 20001 alfa 0
</CLASS_TYPE>
```

Una caratteristica importante viene messa in luce dai fattori scritti col nuovo formalismo. Infatti la ciclicitá dell'attributo "`c`" della classe `alfa` si presta all'introduzione della chiusura transitiva sull'attributo stesso:

```
<CLASS>
;
alfa
epsilon : alfa
(c)*.a : integer
(c)*.b : integer
(c)* : alfa
;
rule2a
epsilon : alfa
a : range 41 2147483647
b : range 33 2147483647
c : alfa
;
rule2c
epsilon : alfa
a : integer
b : integer
c : alfa
```

```
c.a : range 75 2147483647
c.b : integer
c.c : alfa
;
rule1a
epsilon : alfa
a : range 20 2147483647
b : range 30 2147483647
c : alfa
;
rule1c
epsilon : alfa
a : integer
b : integer
c : alfa
c.a : integer
c.b : range 60 2147483647
c.c : alfa
</CLASS_TYPE>
```

6.2.2 Ottimizzazione semantica di una query

Come é stato anticipato a pagina 53, é sufficiente lanciare una query che faccia scattare le regole per entrare in una situazione ciclica. Per comprendere al meglio il funzionamento dell'algoritmo introduciamo la query seguente e, di seguito, il suo schema canonico:

```
select *
from alfa
where a >= 81
and b >= 99
```

```
virt query = alfa  & ^ [ a : range 81 +inf ]
                   & ^ [ b : range 99 +inf ] ;
--------> -alfa  & ^ NewType8 ;
type NewType8 = [ a : range 81 +inf , b : range 99 +inf , c : alfa ] ;
--------> [ a : range 81 +inf , b : range 99 +inf , c : alfa ] ;
```

Si nota immediatamente che a questa interrogazione sono applicabili entrambe la regole. Dunque, al termine della prima iterazione, verranno congiunti di conseguenti `rule1c` e `rule2c` alla classe virtuale `query`.

Per rifarci al formalismo introdotto dall'algoritmo 4.4 scriveremo le due entry della tabella come si presentano al termine della prima interrogazione:

$$
\begin{aligned}
S^1 &= \texttt{query} \\
S^a_1 &= \texttt{rule2a} \\
p_1 &= void
\end{aligned}
$$

$$
\begin{aligned}
S^2 &= \texttt{query} \\
S^a_2 &= \texttt{rule1a} \\
p_2 &= void
\end{aligned}
$$

Il primo risultato parziale dell'ottimizzazione, riportato in forma canonica, sará:

```
virt query1 = -alfa  & ^ [ a : range 81 +inf , b : range 99 +inf ,
                           c : alfa ]  & rule2c  & rule1c ;
--------> -alfa  & ^ NewType10 ;
virt NewName3 = alfa  & ^ [ a : range 75 +inf , b : range 60 +inf ] ;
--------> -alfa  & ^ NewType9 ;
type NewType9 = [ a : range 75 +inf , b : range 60 +inf , c : alfa ] ;
--------> [ a : range 75 +inf , b : range 60 +inf , c : alfa ] ;
ype NewType10 = [ a : range 81 +inf , b : range 99 +inf , c : NewName3 ] ;
--------> [ a : range 81 +inf , b : range 99 +inf , c : NewName3 ] ;
```

Questo costituisce, ovviamente, l'input per la nuova iterazione dell'algoritmo. Osservando attentamente questo schema canonico si nota che al tipo `NewName3` sono, a questo punto, applicabili nuovamente entrambe le regole con il medesimo path `c`. Tuttavia osserviamo anche che, rispetto terne che abbiamo sopra riportato, non ricorre alcuna corrispondenza dal momento che le stesse regole, giá applicate al tipo `query` vengono ora applicata ad un tipo diverso (`NewName3`). Possiamo dunque effettuare le dovute congiunzioni ed

aggiornare la tabella come segue:

$$\begin{aligned} S^1 &= \texttt{query} \\ S_1^a &= \texttt{rule2a} \\ p_1 &= void \end{aligned}$$

$$\begin{aligned} S^2 &= \texttt{query} \\ S_2^a &= \texttt{rule1a} \\ p_2 &= void \end{aligned}$$

$$\begin{aligned} S^3 &= \texttt{NewName3} \\ S_3^a &= \texttt{rule2a} \\ p_3 &= \texttt{c} \end{aligned}$$

$$\begin{aligned} S^4 &= \texttt{NewName3} \\ S_4^a &= \texttt{rule1a} \\ p_4 &= \texttt{c} \end{aligned}$$

Dopo l'applicazione delle regole e la trasformazione in forma canonica, la successiva iterazione si trova in ingresso la seguente struttura:

```
virt query2 = -alfa  & ^ [ a : range 81 +inf , b : range 99 +inf ,
                          c : -alfa  & ^ [ a : range 75 +inf ,
                          b : range 60 +inf , c : alfa ] & rule2c
                          & rule1c ] ;
--------> -alfa  & ^ NewType12 ;
virt NewName4 = -alfa  & rule2c  & rule1c  & ^ [ a : range 75 +inf ,
                                                 b : range 60 +inf ,
                                                 c : alfa ] ;
--------> -alfa  & ^ NewType11 ;
type NewType11 = [ a : range 75 +inf , b : range 60 +inf , c : NewName3 ] ;
--------> [ a : range 75 +inf , b : range 60 +inf , c : NewName3 ] ;
type NewType12 = [ a : range 81 +inf , b : range 99 +inf ,
                   c : NewName4 ] ;
--------> [ a : range 81 +inf , b : range 99 +inf , c : NewName4 ] ;
```

A questo punto siamo effettivamente all'interno di un ciclo: infatti ancora una volta le due regole sono applicabili al tipo `NewName3` con path `c.c`. L'analisi della tabella di terne che é stata costruita fino a questo momento permette di rilevare la condizione di ciclo poiché occorre una corrispondenza tra nomi di tipi e nomi di regole, inoltre i cammini di applicazione si ripetono ciclicamente.

In questo caso le regole vengono marcate opportunamente senza essere applicate ed il programma lancia i seguenti messaggi:

```
---->CICLO di rule2a su NewName3
---->La regola non verra' applicata


---->CICLO di rule1a su NewName3
---->La regola non verra' applicata
```

Quindi si ha la terminazione dell'algoritmo di espansione non essendoci ulteriori regole applicabili.

In queste condizioni assume particolare rilievo la procedura di fattorizzazione dei risultati dell'espansione semantica. Le regole che vengono marcate come cicliche vengono infatti richiamate dalla tabella di terne per modificare opportunamente il path, in sede di schema fattorizzato, attraverso la chiusura transitiva degli attributi che si ripetono ciclicamente (`c` nella fattispecie). Otterremo quindi:

```
void 11 20001 alfa 0
a 100031 20001 81 0
a 100032 20001 2147483647 0
b 100031 20001 99 0
b 100032 20001 2147483647 0
(c)* 11 20001 alfa 0
(c)*.a 100031 20001 75 0
(c)*.a 100032 20001 2147483647 0
(c)*.b 100031 20001 60 0
(c)*.b 100032 20001 2147483647 0
```

Analogamente avremo la nuova notazione che, in queste condizioni, risulta particolarmente espressiva:

```
epsilon : alfa
a : range 81 INFINITY
b : range 99 INFINITY
(c)* : alfa
(c)*.a : range 75 INFINITY
(c)*.b : range 60 INFINITY
```

6.2.3 Visualizzazione dei risultati

Per completezza ci sembra opportuno precisare che, in presenza di situazioni ricorsive, l'attuale modulo di visualizzazione dei risultati non é esente da problemi. Essendo stato progettato solo per situazioni acicliche, infatti, non

prevede una specifica notazione per i cicli e, naturalmente, non si avvale dei risultati ottenuti in questa sede.

Per ovviare a questo genere di inconvenienti si é pensato di visualizzare, a margine delle query ottimizzate, un messaggio che sottolinei l'eventuale ciclicitá delle interrogazioni.

Nel caso trattato si ha infatti, come output finale, la seguente interrogazione:

```
      Semantic Expansion compulsorily terminated: cyclical rule(s) detected

select *
from alfa as S
where S.a > 80 and
      S.b > 98 and
      S.c.a > 74 and
      S.c.b > 59 and
```

Conclusioni

L'obiettivo di questo lavoro, svolto presso il dipartimento di scienze dell'ingegneria dell'universitá di Modena e Reggio Emilia, era l'estensione del sistema ODB-Tools per l'ottimizzazione semantica delle interrogazioni ricorsive. In particolare ci si proponeva, come obiettivo fondamentale, la realizzazione di un algoritmo di espansione semantica di un interrogazione la cui robustezza fosse garantita anche in presenza di ricorsioni. In questo senso possiamo dire che i risultati raggiunti sono stati sicuramente rilevanti. Sono stati infatti proposti e discussi due distinti algoritmi: l'algoritmo di espansione *con soglia* e l'algoritmo di espansione semantica che abbiamo definito *generale*. Entrambi sono stati realizzati in linguaggio C. Un'estesa ed accurata fase di testing ha messo in evidenza la maggiore efficienza del secondo che é stato scelto, quindi, per essere integrato in ODB-Tools.

La necessitá di acquisire e processare interrogazioni e schemi ciclici ha inoltre comportato una reingegnerizzazione globale di tutti i moduli del programma ODB-Tools e, in particolare, di quelle funzionalitá che non erano state progettate per supportare modalitá ricorsive. In quest'ottica si collocano il progetto e lo sviluppo delle nuove procedure di fattorizzazione di schemi canonici che consentono ora di formalizzare anche schemi ricorsivi.

La rivisitazione di gran parte di ODB-Tools ha pure suggerito di estendere il formalismo della logica descrittiva ODL attraverso l'introduzione del nuovo modello OLCD. In particolare é stata introdotta la notazione di chiusura transitiva di un attributo che rappresenta un efficace strumento in ordine alla formalizzazione di situazioni cicliche.

Da un punto di vista strettamente informatico ci sembra il caso di sottolineare che i moduli che, nel loro complesso, realizzano l'espansione semantica di un tipo costituiscono una parte strutturalmente fondamentale per tutto il resto del programma. In altre parole vi é un elevato grado di accoppiamento tra l'espansione semantica e gli altri moduli funzionali, il cui corretto funzionamento é in gran parte subordinato alla stabilitá dello stesso procedimento

di espansione. Si capisce allora come un intervento strutturale sull'algoritmo di espansione semantica comporti a sua volta interventi piú o meno rilevanti sui moduli ad esso collegati.

L'implementazione del nuovo algoritmo di espansione semantica ha dunque comportato anche un risultato, per così dire, indiretto: l'analisi delle funzioni collegate ha infatti portato alla luce ridondanze ed errori che sono stati opportunamente corretti. In definitiva ODB-Tools, al termine del lavoro, non é solo in grado di trattare le ricorsioni, ma, come hanno messo in luce le ripetute sessioni di testing, risulta globalmente piú stabile ed efficiente.

Rimane tuttavia, come problema aperto, la visualizzazione del risultato finale dell'ottimizzazione di un interrogazione nel linguaggio OQL. Le attuali funzioni prevedono esclusivamente situazioni acicliche e, naturalmente, non si avvalgono dei risultati ottenuti al termine di questo lavoro. Accade dunque che l'output presentato a video dal programma non coincida sempre, nei casi ricorsivi, con il risultato dell'ottimizzatore.

La possibilitá di visualizzare risultati ciclici, l'integrazione con il nuovo formalismo nonché l'utilizzo della nuova versione dei fattori in fase di riscrittura della query, costituiscono indubbiamente un importante punto di partenza per i futuri sviluppi delle funzionalitá di ODB-Tools.

Appendice A

Schemi Canonici ed Algoritmi di Incoerenza e Sussunzione

In uno schema l'ereditarietà è espressa nella descrizione di un nome di tipo tramite l'operatore di intersezione. Tramite la nozione di *schema canonico* si vuole definire uno schema nel quale la descrizione di un nome di tipo è espressa senza congiunzioni e contiene tutte le proprietà del nome, sia quelle definite localmente sia quelle ereditate. Lo schema canonico deve preservare la semantica dello schema originale, nel senso che le relazioni semantiche valide nello schema originale devono essere conservate nello schema canonico.

Il principale obiettivo dello schema canonico è quello di rendere relativamente semplici i problemi del controllo di incoerenza e del calcolo di sussunzione, che invece sono complicati in uno schema generale.

Prima di dare le definizioni formali, facciamo alcune considerazioni intuitive. Lo schema canonico si ottiene sostituendo ricorsivamente, in una descrizione di un tipo, i nomi dei tipi dal quale esso eredita con la rispettiva descrizione (questo è possibile in quanto la relazione di ereditarietà è un ordine parziale stretto). Nel tipo risultante vengono quindi risolte le intersezioni. La sostituzione di un nome di tipo preserva la semantica dello schema originale ad eccezione che si tratti di un nome di classe base: infatti l'estensione di un nome di classe base è un sottoinsieme dell'estensione della rispettiva descrizione. Allora, prima di effettuare le sostituzioni, ad una classe base C nello schema σ si fa corrispondere nello schema canonico una classe virtuale espressa come l'intersezione della descrizione di C, $\sigma(C)$, e di un nuovo nome di classe base $\overline{C}$, detto *atomo fittizio*, la cui descrizione è $\triangle\top$.

A.1 Schema canonico

Si introduce prima di tutto la nozione di tipo in forma canonica.

$\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ denota l'insieme delle *descrizioni canoniche di tipo* ($\overline{S}$, $\overline{S}'$, ...) su $\mathbf{A},\mathbf{B},\overline{\mathbf{N}}$, con $\overline{\mathbf{N}} = \overline{\mathbf{C}} \cup \overline{\mathbf{V}} \cup \mathbf{T}$ e $\overline{\mathbf{V}} = \mathbf{C} \cup \mathbf{D}$, che sono costruite rispettando la seguente regola sintattica astratta: [1]

$$\overline{S} \;\rightarrow\; \overline{S} \mid \{\overline{S}\} \mid \langle\overline{S}\rangle \mid [a_1\colon \overline{S}_1, \ldots, a_k\colon \overline{S}_k] \mid \bigsqcap_i \overline{C}_i \sqcap \triangle\overline{S}.$$

con $k \geq 0$, $i \geq 0$ e $\overline{S} \in \{\top\} \cup \mathbf{B} \cup \overline{\mathbf{V}} \cup \overline{\mathbf{T}}$ e $\overline{C}_i \in \overline{\mathbf{C}}$[2].

Definizione 14 (Schema Canonico equivalente) *Uno* schema canonico ν *è uno schema su* $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ *tale che per ogni* $\overline{C}_i \in \overline{\mathbf{C}}$, $\nu(\overline{C}_i) = \triangle\top$. *Dato uno schema* σ *su* $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ *si dice che uno schema canonico* ν *su* $\mathbf{S}$ *con* $\mathbf{N} \subseteq \overline{\mathbf{N}}$ *è* equivalente *a* σ *sse per ogni istanza possibile* $\mathcal{I}$ *di* σ *esiste un'istanza possibile* $\mathcal{I}'$ *di* ν *e* viceversa *tale che* $\mathcal{I}[N] = \mathcal{I}'[N], \forall N \in \mathbf{N}$.

Da questa definizione segue immediatamente che le relazioni semantiche in due schemi equivalenti sono le stesse:

Proposizione 6 *Dato uno schema* σ *su* $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, *per ogni* $N, N' \in N$, $N \sqsubseteq_\sigma N'$ *sse* $N \sqsubseteq_\nu N'$, *con* ν *schema canonico su* $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ equivalente *a* σ.

Di conseguenza, il calcolo della sussunzione e il controllo di coerenza di uno schema arbitrario possono essere effettuati considerando lo schema canonico equivalente.

Per quanto riguarda la possibilità di trasformare effettivamente uno schema in una forma canonica equivalente, vale quanto segue:

Proposizione 7 *Ogni schema* σ *su* $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$ *aciclico può essere effettivamente trasformato in uno schema canonico* ν *su* $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ *che è* equivalente *a* σ.

[1] Si assume che per $i = 0$ la regola $\bigsqcap_i \overline{C}_i \sqcap \triangle\overline{S}$ produca il tipo $\triangle\overline{S}$.

[2] Si assume che nel dominio di un attributo non compaiono intervalli di valori ed enumerazioni ma i tipi che li descrivono.

$$
\begin{array}{rcl}
S \sqcap S & \simeq & S \\
S \sqcap S' & \simeq & S' \sqcap S \\
S \sqcap (S' \sqcap S'') & \simeq & (S \sqcap S') \sqcap S''
\end{array}
$$

$$
[a_1\colon S_1,\ldots,a_p\colon S_p] \;\sqcap\; [a'_1\colon S'_1,\ldots,a'_q\colon S'_q] \;\simeq\; \begin{cases} [a_1\colon S_1 \sqcap S'_1, a_2\colon S_2,\ldots,a_p\colon S_p,] \sqcap \\ [a'_2\colon S'_2,\ldots,a'_q\colon S'_q,] \\ \text{se } a'_1 = a_1 \\ \\ [a'_1\colon S'_1, a_1\colon S_1,\ldots,a_p\colon S_p,] \sqcap \\ [a'_2\colon S'_2,\ldots,a'_q\colon S'_q,] \\ \text{se } a'_1 \neq a_i \text{ per } 1 \leq i \leq p \end{cases}
$$

$$
\begin{array}{rcl}
\{S\} \sqcap \{S'\} & \simeq & \{S \sqcap S'\} \\
\langle S\rangle \sqcap \langle S'\rangle & \simeq & \langle S \sqcap S'\rangle \\
\triangle S \sqcap \triangle S' & \simeq & \triangle(S \sqcap S')
\end{array}
$$

$$
N \sqcap N' \;\simeq\; N \quad \text{se } N \prec_\sigma N'
$$

$$
S \sqcap S' \;\simeq\; \bot \quad \text{se } (S,S') \in \mathbf{I}
$$

$$
B \sqcap B' \;\simeq\; B'' \quad \text{dove } B'' = B \sqcap B'
$$

Tabella A.1: Equivalenze tra tipi

A.2 Generazione dello schema canonico

Un generico tipo di uno schema σ può essere trasformato in un tipo equivalente nella quale le intersezioni riguardano *esclusivamente* nomi di tipo, cioè siano esprimibili come $\bigsqcap_k N_k \sqcap S'$; una tale espressione di tipo è detta *ridotta all'intersezione*. Questa trasformazione è naturalmente indispensabile per ottenere uno schema canonico.

Proposizione 8 *Dato uno schema σ su $\mathbf{S}$, per ogni tipo $S \in \mathbf{S}$, esiste un'espressione di tipo di $\mathbf{S}$ ridotta all'intersezione, denotata con $\tilde{\mu}(S)$, equivalente a S: $\tilde{\mu}(S) \simeq S$.*

La dimostrazione della proposizione è data dal fatto che $\tilde{\mu}(S)$ è ottenibile tramite delle semplici manipolazioni algebriche basate sulle equivalenze tra tipi mostrate in tabella A.1. (Per ulteriori chiarimenti vedere [Bal92]).

Sulle espressioni di tipo $\tilde{\mu}(S)$ si considera una relazione di *uguaglianza sintattica*, indicata con $\approx$, definita a meno di permutazione sugli attributi dei

tipi tupla sui fattori nelle congiunzioni $\bigsqcap_k N_k \sqcap S'$. Ovviamente, se $S \approx S'$ allora $S \simeq S'$ mentre il viceversa non è valido.

In seguito sono descritte le varie fasi di trasformazione di uno schema σ in uno schema canonico ν. Prima di tutto, ad uno schema σ viene associato uno schema τ detto *schema equazionale* di σ: ogni classe base C di σ viene sostituita in τ con una classe virtuale espressa come la congiunzione di una *classe atomica fittizia* $\overline{C}$ e della sua descrizione $\sigma(C)$. Una classe atomica fittizia è una classe base di τ la cui descrizione è $\triangle\top$.

Formalmente, dato uno schema σ su $\mathbf{S}(\mathbf{A},\mathbf{B},\mathbf{N})$, sia $\overline{\mathbf{C}}$ una copia disgiunta di $\mathbf{C}$, π una biiezione, $\pi\colon \mathbf{C} \to \overline{\mathbf{C}}$, e $\overline{\mathbf{N}} = \overline{\mathbf{C}} \cup \overline{\mathbf{V}} \cup \mathbf{T}$ con $\overline{\mathbf{V}} = \mathbf{C} \cup \mathbf{D}$. Lo *schema equazionale* di σ è uno schema τ su $\mathbf{S}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ definito come segue:

$$
\tau(N) \;=\; \begin{cases} \triangle\top & \text{se } N \in \overline{\mathbf{C}} \\ \pi(N) \sqcap \sigma(N) & \text{se } N \in \mathbf{C} \\ \sigma(N) & \text{altrimenti} \end{cases}
$$

È immediato verificare che per ogni istanza possibile $\mathcal{I}$ di σ esiste un'istanza possibile $\mathcal{I}'$ di τ, e viceversa, tale che

$$
\mathcal{I}[N] = \mathcal{I}'[N], \qquad \text{per ogni } N \in \mathbf{N}
$$

La funzione che segue fornisce, per un nome di tipo $N \in \overline{\mathbf{V}} \cup \overline{\mathbf{T}}$, la descrizione di tipo che si ottiene sostituendo ogni nome di tipo N' da cui N eredita con la rispettiva descrizione:

$$
\iota(S) \;=\; \begin{cases} \tau(S) & \text{se } S \in \overline{\mathbf{V}} \cup \overline{\mathbf{T}} \\ \iota(S') \sqcap \iota(S'') & \text{se } S = S' \sqcap S'' \\ S & \text{altrimenti} \end{cases}
$$

Si definisce $\tilde{\iota} = \iota^n$, dove n è il più piccolo numero naturale tale che $\iota^n = \iota^{n+1}$; tale numero esiste sempre per schemi ben–formati sull'ereditarietà. Il tipo che si ottiene con $\tilde{\iota}$ è equivalente (per tutte e tre le semantiche) in τ a quello originale: $\tilde{\iota}(S) \simeq_\tau S$.

La composizione della funzione $\tilde{\mu}$ con la funzione $\tilde{\iota}$ verrà indicata con κ: per ogni tipo S

$$
\kappa(S) = \tilde{\mu}(\tilde{\iota}(S))
$$

Si noti che $\kappa(S)$ è un'espressione di tipo ridotta all'intersezione particolare: le eventuali intersezioni a livello più esterno riguardano esclusivamente nomi di classi atomiche fittizie. Per quanto detto sulle trasformazioni $\tilde{\iota}$ e $\tilde{\mu}$ segue immediatamente che $\kappa(S) \simeq_\tau S$.

A questo punto può essere data la definizione operazionale di schema canonico ν. Informalmente, partendo da τ, si costruiscono una sequenza di schemi f su $\overline{\mathbf{N}}^0, \overline{\mathbf{N}}^1, \overline{\mathbf{N}}^2 \ldots$; si passa da $\overline{\mathbf{N}}^i$ a $\overline{\mathbf{N}}^{i+1}$ sostituendo ogni tipo S di f che non è in $\{\top\} \cup \mathbf{B} \cup \overline{\mathbf{N}}^i$, con un nome N dove N è un nome di $\overline{\mathbf{N}}^i$ se $f(N)$ è uguale a S oppure N è un nuovo nome introdotto in $\overline{\mathbf{N}}^{i+1}$, con $f(N) = S$. Il processo termina quando tutti i nomi dello schema hanno una descrizione in forma canonica. L'algoritmo è riportato in tabella A.2.

L'introduzione di nomi nuovi, e quindi l'algoritmo, si conclude dopo un numero finito di passi, in quanto nella determinazione di $\tilde{\mu}$, compresa in $\kappa(S)$, si utilizza la seguente equivalenza tra nomi di tipo: $N \sqcap N' \simeq N$ se $N \prec_\sigma N'$.

Per lo schema di riferimento di tabella A.3 l'algoritmo produce lo schema canonico mostrato in tabella A.4.

Assegnamento disgiunto di oggetti alle classi

In genere nei sistemi di basi di dati orientate agli oggetti si impone la restrizione che un oggetto *appartenga* ad una singola classe con un valore *corrispondente* al tipo associato a tale classe. Per tener conto della gerarchia di ereditarietà, l'oggetto viene quindi considerato *membro* di tutte le superclassi della classe nella quale l'oggetto è istanziato.

Questa differenza non è effettuata nella nostra definizione di istanza di uno schema, dove un oggetto può essere istanziato in più classi nel rispetto, oltre che del tipo associato alla classe, della gerarchia di ereditarietà: un oggetto istanziato in una classe deve essere istanziato in tutte le superclassi di questa classe. Inoltre, dovuto alla "semantica di mondo aperto" adottata per i tipi tupla, la nostra definizione di istanza ammette, come in O_2 (nozione di *istanza eccezionale*), che un oggetto istanziato in una certa classe abbia un valore associato con attributi aggiuntivi rispetto a quelli specificati nel tipo associato alla classe. Infine, la definizione di istanza legale di uno schema ammette che vi siano oggetti non istanziati in nessuna classe, e quindi in particolare, in nessuna classe base.

Nel seguito verrà mostrato come imporre le precedenti restrizioni sulla nostra nozione di istanza possibile di uno schema e quindi, di conseguenza, su quella di istanza legale relativa ad una certa semantica. A tale scopo si prende come riferimento il modello presentato in [AK89], dove le condizioni sono state formalizzate, e si utilizza in particolare la nozione di schema canonico.

In [AK89], la definizione di istanza di uno schema, inizialmente indipendente dalla nozione di ereditarietà, è basata sul concetto di un *assegnamento di oid disgiunto* (π) alle classi e di una funzione interpretazione che ammette per i tipi tuple solo i valori che hanno *esattamente* (tutti e soli) gli attributi della tupla. Per considerare l'ereditarietà si definisce l'*assegnamento di oid ereditato* $\bar{\pi}$ relativo a π e si impone che il valore associato ad un oggetto

Algoritmo (**Schema Canonico**)

- **Inizializzazione**: $f = \tau$
- **Iterazione**: dato f su $\overline{\mathbf{N}}^i$ calcola f su $\overline{\mathbf{N}}^{i+1}$, con $\overline{\mathbf{N}}^i \subseteq \overline{\mathbf{N}}^{i+1}$, nel seguente modo:

  per ogni $N \in \overline{\mathbf{N}}^i$:

  $$f(N) = \mathbf{GC}(\kappa(N))$$

  dove

  $$\begin{aligned}
  \mathbf{GC}(S) &= S \text{ se } S \in \{\top\} \cup \mathbf{B} \cup \overline{\mathbf{V}} \cup \mathbf{T} \\
  \mathbf{GC}(\{S\}) &= \{\mathbf{NT}(S)\} \\
  \mathbf{GC}(\langle S \rangle) &= \langle \mathbf{NT}(S) \rangle \\
  \mathbf{GC}([a_1\colon S_1, \ldots, a_p\colon S_p]) &= [a_1\colon \mathbf{NT}(S_1), \ldots, a_p\colon \mathbf{NT}(S_p)] \\
  \mathbf{GC}(\bigsqcap_i \overline{C}_i \sqcap \triangle S) &= \bigsqcap_i \overline{C}_i \sqcap \triangle \mathbf{NT}(S)
  \end{aligned}$$

  e

  $$\mathbf{NT}(S) = \begin{cases} S & \text{se } S \in \{\top\} \cup \mathbf{B} \cup \overline{\mathbf{V}} \cup \mathbf{T} \\ N & \text{se } \exists N \in \overline{\mathbf{N}}^{i+1}\colon \kappa(N) \approx \kappa(S) \\ N' & \text{nuovo in } \overline{\mathbf{N}}^{i+1} \text{ con } f(N') = \kappa(S) \\ & \text{aggiungi } N' \text{ a } \overline{\mathbf{N}}^{i+1} \end{cases}$$

- **Stop**: $\overline{\mathbf{N}}^{i+1} = \overline{\mathbf{N}}^i$; definisci $\tilde{\nu} = f$.

Tabella A.2: Generazione dello schema canonico

$$
\begin{array}{rcl}
\mathbf{T} & = & \{\} \\
\mathbf{C} & = & \{\texttt{manager}, \texttt{tmanager}, \texttt{material}, \\
 & & \texttt{smaterial}, \texttt{storage}, \texttt{sstorage}\} \\
\mathbf{D} & = & \{\}
\end{array}
$$

$$
\begin{array}{rcl}
\sigma_{\mathtt{P}}(\texttt{material}) & = & \triangle\Big[\texttt{name}\colon \texttt{string}, \texttt{risk} : \texttt{integer}, \texttt{feature} : \{\texttt{string}\}\Big] \\
\sigma_{\mathtt{P}}(\texttt{smaterial}) & = & \texttt{material} \\
\sigma_{\mathtt{P}}(\texttt{storage}) & = & \triangle\Big[\texttt{managed-by}\colon \texttt{manager}, \texttt{category}\colon \texttt{string} \\
 & & \quad \texttt{stock} : \{[\texttt{item} : \texttt{material}, \texttt{qty} : 10 \div 300]\Big] \\
\sigma_{\mathtt{P}}(\texttt{sstorage}) & = & \texttt{storage} \\
\sigma_{\mathtt{P}}(\texttt{manager}) & = & \triangle\Big[\texttt{name}\colon \texttt{string}, \texttt{salary} : > 40000, \texttt{level} : 1 \div 15\Big] \\
\sigma_{\mathtt{P}}(\texttt{tmanager}) & = & \texttt{manager} \sqcap \triangle\Big[\texttt{level}\colon 8 \div 12\Big]
\end{array}
$$

Tabella A.3: Schema del domino Magazzino

corrisponde al tipo della più piccola classe della gerarchia alla quale l'oggetto appartiene. Si noti che per ottenere il tipo di una classe come intersezione dei tipi associati alle superclassi e del tipo della classe stessa, si deve definire un'ulteriore funzione interpretazione che adotta per i tipi tuple una semantica di mondo aperto.

Vediamo come imporre queste condizioni sulla nostra nozione di istanza possibile $\mathcal{I}$. Data un'istanza possibile $\mathcal{I}$ relativa ad un certo dominio δ, per prima cosa si impone che ogni oid del dominio sia istanziato in almeno una classe base: $\mathcal{O} = \bigcup_{C \in \mathbf{C}} \mathcal{I}[C]$. Si controlla quindi se è possibile determinare un'assegnamento di oid disgiunto π ($\pi\colon \mathbf{C} \to 2^{\mathcal{O}}$ e $\pi[C] \cap \pi[C'] = \emptyset$, per $C \neq C'$) tale che il relativo assegnamento ereditato coincide con $\mathcal{I}$, cioè tale che per ogni C, $\mathcal{I}[C] = \bigcup_{C_i \preceq_\sigma C} \pi[C_i]$. È facile verificare che se tale assegnamento esiste esso puo' essere univocamente determinato come $\pi[C] = \mathcal{I}[C] \setminus \bigcup_{C_i \prec_\sigma C} \mathcal{I}[C_i]$.

Per imporre che il valore associato ad un oggetto corrisponda al tipo della più piccola classe della gerarchia alla quale l'oggetto appartiene si definisce una funzione interpretazione *chiusa* per i tipi tuple. Questa funzione interpretazione non può essere usata direttamente nello schema σ in quanto darebbe risultato necessariamente vuoto per l'intersezioni di due tipi tupla con attributi diversi. Si considera quindi lo schema canonico ν dove non vi

$$
\begin{array}{rcl}
\overline{\mathbf{T}} & = & \{b_1 \ldots b_4\} \cup \{t_1 \ldots t_7\} \\
\overline{\mathbf{C}} & = & \{\overline{\texttt{material}}, \overline{\texttt{smaterial}}, \overline{\texttt{storage}}, \overline{\texttt{sstorage}}, \\
 & & \overline{\texttt{manager}}, \overline{\texttt{tmanager}}\} \\
\overline{\mathbf{D}} & = & \{\texttt{material}, \texttt{smaterial}, \texttt{storage}, \texttt{sstorage}, \\
 & & \texttt{manager}, \texttt{tmanager}\}
\end{array}
$$

$$
\begin{array}{rcl}
\nu(\texttt{material}) & = & \overline{\texttt{material}} \sqcap \triangle t_1 \\
\nu(\texttt{smaterial}) & = & \overline{\texttt{smaterial}} \sqcap \triangle t_1 \\
\nu(\texttt{storage}) & = & \overline{\texttt{storage}} \sqcap \triangle t_3 \\
\nu(\texttt{sstorage}) & = & \overline{\texttt{storage}} \sqcap \overline{\texttt{sstorage}} \sqcap \triangle t_3 \\
\nu(\texttt{manager}) & = & \overline{\texttt{manager}} \sqcap \triangle t_6 \\
\nu(\texttt{tmanager}) & = & \overline{\texttt{manager}} \sqcap \overline{\texttt{tmanager}} \sqcap \triangle t_6 \\
\nu(t_1) & = & [\texttt{feature} : t_2, \texttt{name} : \texttt{string}, \texttt{risk} : \texttt{integer}] \\
\nu(t_2) & = & \{\texttt{string}\} \\
\nu(t_3) & = & [\texttt{category} : \texttt{string}, \texttt{stock}\colon t_4, \texttt{managed-by} : \texttt{manager}] \\
\nu(t_4) & = & \{t_5\} \\
\nu(t_5) & = & [\texttt{item} : \texttt{material}, \texttt{qty} : b_1] \\
\nu(t_6) & = & [\texttt{level} : b_2, \texttt{name} : \texttt{string}, \texttt{salary} : b_3] \\
\nu(t_7) & = & [\texttt{level} : b_4, \texttt{name} : \texttt{string}, \texttt{salary} : b_3] \\
\nu(b_1) & = & 10 \div 300 \\
\nu(b_2) & = & 1 \div 15 \\
\nu(b_3) & = & > 40000 \\
\nu(b_4) & = & 8 \div 12
\end{array}
$$

Tabella A.4: Schema canonico di un Magazzino

sono intersezioni tra tipi tupla.

In definitiva, dato uno schema σ su $\mathbf{S}$, un'istanza possibile $\mathcal{I}$ di σ relativa ad un dominio δ su $\mathcal{O}$, si definisce

$$\pi[C] = \mathcal{I}[C] \setminus \bigcup_{C_i \prec_\sigma C} \mathcal{I}[C_i]$$

e si effettuano i seguenti controlli:

1. $\mathcal{O} = \bigcup_{C_i \in \mathbf{C}} \mathcal{I}[C_i]$
2. $\pi[C] \cap \pi[C'] = \emptyset$, per $C' \neq C$;
3. $\mathcal{I}[C] = \bigcup_{C_i \preceq_\sigma C} \pi[C_i]$
4. $\forall C$, $\pi[C] \subseteq \{o \in \mathcal{O} | \delta(o) \in \hat{\mathcal{I}}[\tilde{S}]\}$, dove
   - $\tilde{S}$ è la descrizione canonica della classe C: nello schema canonico ν su $\overline{\mathbf{S}}$ equivalente a σ si ha che $\nu(C) = \bigsqcap_k \overline{C}_k \sqcap \triangle\tilde{S}$
   - $\hat{\mathcal{I}}$ è la *interpretazione chiusa* relativa a $\mathcal{I}$ definita come segue:

$$\begin{aligned}
\hat{\mathcal{I}}[S] &= \mathcal{I}[S] \text{ se } S \in \mathbf{C} \cup \mathbf{D} \text{ o } S = \triangle S' \\
\hat{\mathcal{I}}[t] &= \hat{\mathcal{I}}[\nu(t)] \\
\\
\hat{\mathcal{I}}[\top] &= \mathcal{V}(\mathcal{O}) \\
\hat{\mathcal{I}}[B] &= \mathcal{I}_{\mathbf{B}}[B] \\
\hat{\mathcal{I}}\Big[[a_1 : S_1, \ldots, a_p : S_p]\Big] &= \Big\{[a_1 : v_1, \ldots, a_q : v_q] \Big| \; p = q, v_i \in \hat{\mathcal{I}}[S_i], \\
&\qquad\qquad 1 \leq i \leq p\Big\} \\
\hat{\mathcal{I}}[\{S\}] &= \Big\{\{v_1, \ldots, v_p\} \Big| v_i \in \hat{\mathcal{I}}[S], \leq i \leq p\Big\} \\
\hat{\mathcal{I}}[\langle S \rangle] &= \Big\{\langle v_1, \ldots, v_p \rangle \Big| v_i \in \hat{\mathcal{I}}[S], 1 \leq i \leq p\Big\}
\end{aligned}$$

Consideriamo un'istanza possibile $\mathcal{I}$ che rispetti tali condizioni; al relativa $\hat{\mathcal{I}}$ dei tipi–oggetto $\triangle S$ e dei nomi delle classi resta invariata. In questo modo, mentre

$$[\texttt{name}: \textit{"Mark"}, \texttt{salary}: 8000, \texttt{level}: 3] \notin \hat{\mathcal{I}}\Big[[\texttt{name}: \texttt{string}]\Big]$$

considerando

$$\delta(o) = [\texttt{name}: \textit{"Mark"}, \texttt{salary}: 8000, \texttt{level}: 3]$$

si ha che $o \in \hat{\mathcal{I}}[\triangle[\texttt{name}: \texttt{string}]]$, cioè anche dopo aver imposto le suddette condizioni il tipo $\triangle[\texttt{name}: \texttt{string}]$ continua a rappresentare la generalizzazione di tutte le classi, basi e virtuali, che hanno almeno l'attributo `name`; tra queste classi deve esistere necessariamente un'unica classe base C dove $o \in \pi[C]$; tale classe avrà esattamente gli attributi `name`, `salary` e `level`.

A.3 Algoritmi di incoerenza e sussunzione

In questa sezione presentiamo gli algoritmi per il controllo di coerenza e il calcolo della sussunzione in uno schema canonico. Per ciascun algoritmo sarà dimostrata la sua correttezza e completezza e ne verrà discussa la sua complessità.

Siccome per ogni tipo S è stata fissata una precisa semantica estensionale, la relazione di sussunzione tra due tipi S e S', definita per inclusione semantica, è determinabile tramite un algoritmo che effettua il confronto strutturale delle espressioni S e S'. Il confronto strutturale è definito nello schema canonico, cioè in uno schema dove sono state risolte le intersezioni (le intersezioni riguardano solo le classi atomiche fittizie). Di conseguenza, i confronti per il calcolo della sussunzione sono simili a quelli che definiscono sintatticamente la relazione strutturale di *sottotipo* (*subtyping* o *refinement*) generalmente adottata negli OODB.

Nello stesso modo la coerenza di un tipo S è determinabile tramite un'analisi strutturale. Da un punto di vista teorico, l'incoerenza è definita sulla base della relazione di sussunzione e quindi la rilevazione dell'incoerenza è riconducibile al calcolo della sussunzione. In pratica si osserva che la rilevazione di incoerenza è più efficiente se implementata come algoritmo separato.

Nel seguito, per semplicità, un tipo in forma canonica verrà indicato con S.

A.3.1 Controllo di incoerenza

L' algoritmo di incoerenza è basato sul calcolo iterativo di un insieme Φ di tipi. Fissato l'insieme dei nomi incoerenti, i restanti tipi canonici incoerenti sono ricavabili in maniera immediata. A tale scopo, vengono imposte delle condizioni sui tipi presenti in Φ ad un generico passo dell'algoritmo.

Dato uno schema canonico ν su $\overline{\mathbf{S}}(\mathbf{A}, \mathbf{B}, \overline{\mathbf{N}})$, sia Φ un sottoinsieme di

$\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ tale che[3]:

$$\begin{array}{c} \bot \in \Phi \\ [a_1\colon S_1,\ldots,a_n\colon S_n] \in \Phi \quad \text{sse} \quad \exists k, 1 \leq k \leq n\colon S_k \in \Phi \\ \bigsqcap \overline{C}_i \sqcap \triangle S \in \Phi \quad \text{sse} \quad S \in \Phi \end{array}$$

Fissato una relazione di incoerenza per i nomi di tipo, le condizioni imposte su Φ permettono di estenderla agli altri tipi in forma canonica. Ora si deve appunto completare il calcolo dei tipi incoerenti considerando i nomi di tipo $N \in \overline{\mathbf{N}}$. Si definisce l'algoritmo di incoerenza nel quale, dato un insieme di partenza Φ^0, si calcolano iterativamente l'insieme dei nomi di tipo incoerenti sulla base della loro descrizione.

Algoritmo (**Incoerenza**)

- **Inizializzazione**: $N \notin \Phi^0$, per ogni $N \in \overline{\mathbf{N}}$;
- **Iterazione**: $N \in \Phi^{i+1}$ sse $\nu(N) \in \Phi^i$;
- **Stop**: $\Phi^{i+1} = \Phi^i$; definisci $\widetilde{\Phi}^g_\nu = \Phi^i$.

Tabella A.5: Algoritmo di Incoerenza

La funzione Φ^i incrementa in modo monotono all'aumentare di i; di conseguenza, poichè $\overline{\mathbf{N}}$ è finito, esiste un numero naturale n tale che $\Phi^n = \Phi^{n+1}$, cioè l'algoritmo termina. Inoltre il massimo n è $|\overline{\mathbf{N}}|$, cioè l'algoritmo è polinomiale in ν. L'algoritmo calcola l'insieme di nomi di tipi incoerenti; le condizioni imposte su Φ permettono di ricavare l'insieme totale dei tipi incoerenti.

L'insieme $\widetilde{\Phi}_\nu$ determinato dagli algoritmi ha le seguenti proprietà:

1. $N \simeq_\nu \bot$ **se** $N \in \widetilde{\Phi}_\nu$ (**correttezza** dell'algoritmo)
2. $N \simeq_\nu \bot$ **solo se** $N \in \widetilde{\Phi}_\nu$ (**completezza** dell'algoritmo)

Formalmente, ciò è espresso dal seguente teorema dimostrato in [D.B94].

Teorema 2 *Sia ν uno schema canonico su $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$. Allora*

$$N \simeq_\nu \bot \quad \textit{sse} \quad N \in \widetilde{\Phi}_\nu$$

[3] Si ricordi che $\{\bot\}$ e $\langle\bot\rangle$ sono tipi coerenti che indicano rispettivamente l'insieme vuoto e la sequenza vuota.

A.3.2 Calcolo della sussunzione

L'algoritmo di sussunzione è basato sul calcolo iterativo di una relazione $\leq$ tra i tipi in forma canonica. Fissata una relazione di sussunzione tra i nomi di tipo è facilmente estendibile a tutti i restanti tipi canonici. Pertanto, si definisce prima di tutto la relazione $\leq$ per una coppia di tipi diversa da una coppia di nomi.

Dato uno schema canonico ν su $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$, sia $\leq$ una relazione tra $S, S' \in \overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$ tale che:

- per ogni tipo canonico $S \in \overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$

$$S \leq \top$$

- per ogni coppia di tipi atomici $B, B' \in \mathbf{B}$

$$B \leq B' \text{ sse } B \sqcap B' = B$$

- per ogni $S \in \{\top\} \cup \mathbf{B}$, $N \in \overline{\mathbf{N}}$:

$$\begin{array}{c} S \leq N \quad \text{sse} \quad S \leq \nu(N) \\ N \leq S \quad \text{sse} \quad \nu(N) \leq S \end{array}$$

- per ogni $S, S' \in \{\top\} \cup \mathbf{B} \cup \overline{\mathbf{N}}$:

$$\begin{array}{rcl} \{S\} \leq \{S'\} & \text{sse} & S \leq S' \\ \langle S\rangle \leq \langle S'\rangle & \text{sse} & S \leq S' \\ [\ldots,a_i\colon S_i,\ldots] \leq [\ldots,a'_j\colon S'_j,\ldots] & \text{sse} & \forall j \exists i\colon\ (a_i = a'_j \wedge S_i \leq S'_j) \\ \bigsqcap_k \overline{C}_k \sqcap \triangle S \leq \bigsqcap_j \overline{C}'_j \sqcap \triangle S' & \text{sse} & S \leq S' \wedge \forall j \exists k\colon \overline{C}_k = \overline{C}'_j. \end{array}$$

Intuitivamente, fissato un ordinamento tra i nomi dei tipi, le condizioni imposte su $\leq$ permettono di estenderlo a tutti i tipi in forma canonica. Ora si deve appunto completare la definizione della relazione $\leq$ considerando i confronti tra due nomi di tipo $N \in \overline{\mathbf{N}}$.

Si definisce l'algoritmo di sussunzione che, fissata una relazione di partenza $\leq^0$, calcola iterativamente la relazione $\leq$ tra i nomi di tipo basandosi sulle loro descrizione e sull'insieme dei nomi di tipo incoerenti.

Nell'algoritmo la relazione $\leq^i$ incrementa in modo monotono all'aumentare di i. Di conseguenza, come nel caso dell'algoritmo di incoerenza, poichè $\overline{\mathbf{N}}$ è finito, esiste un numero naturale n tale che $\leq^n = \leq^{n+1}$. Il massimo n è $|\overline{\mathbf{N}}|^2$, cioè anche l'algoritmo di sussunzione è polinomiale in ν.

La relazione $\lesssim_\nu$ determinata dall'algoritmo ha le seguenti proprietà:

Algoritmo (**Sussunzione**)

- **Inizializzazione**: $N \leq^0 N'$ sse $(N,N') \in \overline{\mathbf{V}}^2 \cup \overline{\mathbf{T}}^2 \cup \left\{ (\overline{C},\overline{C}) \middle| \overline{C} \in \overline{\mathbf{C}} \right\}$
- **Iterazione**: $N \leq^{i+1} N'$ sse $\nu(N) \leq^i \nu(N') \ \vee \ (N \in \tilde{\Phi}^g_\nu)$
- **Stop**: $\leq^{i+1} = \leq^i$; definisci $\lesssim^g_\nu = \leq^i$.

Tabella A.6: Algoritmo di Sussunzione

1. $S \sqsubseteq_\sigma S'$ **se** $S \lesssim_\nu S'$ (**correttezza** dell'algoritmo)
2. $S \sqsubseteq_\sigma S'$ **solo se** $S \lesssim_\nu S'$ (**completezza** dell'algoritmo)

Formalmente, ciò è espresso dal seguente teorema dimostrato in [D.B94].

Teorema 3 *Sia ν uno schema canonico su $\overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$. Allora per ogni $S, S' \in \overline{\mathbf{S}}(\mathbf{A},\mathbf{B},\overline{\mathbf{N}})$:*

$$S \sqsubseteq^g_\nu S' \quad sse \quad S \lesssim^g_\nu S'$$
$$S \sqsubseteq^d_\nu S' \quad sse \quad S \lesssim^d_\nu S'$$
$$S \sqsubseteq^l_\nu S' \quad sse \quad S \lesssim^l_\nu S'$$

Appendice B

Sintassi OCDL

In questa sezione è riportata la sintassi dell'OCDL riconosciuta dal validatore.

$$
\begin{array}{rcl}
< \textit{linguaggio} >< \text{linguaggio} > & ::= & < \text{def-term} > \mid \\
 & & < \text{linguaggio} >< \text{def-term} > \\
< \text{def-term} > & ::= & < \text{def-tipovalore} > \mid \\
 & & < \text{def-classe} > \mid \\
 & & < \text{def-regola} > \\
< \text{def-tipovalore} > & ::= & < \text{def-tipobase} > \mid \\
 & & < \text{def-tipo} > \\
< \text{def-classe} > & ::= & < \text{def-classe-prim} > \mid \\
 & & < \text{def-classe-virt} > \\
< \text{def-regola} > & ::= & < \text{def-antconV} > \\
 & & < \textit{nome regola} > = < \text{classe} > \mid \\
 & & < \text{def-antconT} > \\
 & & < \textit{nome regola} > = < \text{def-tipoT} > \\
< \text{def-antconV} > & ::= & \textbf{antev} \mid \textbf{consv} \\
< \text{def-antconT} > & ::= & \textbf{antet} \mid \textbf{const} \\
< \text{def-tipoT} > & ::= & < \text{tipo} > \mid < \text{tipobase} > \\
< \text{def-tipo} > & ::= & \textbf{type} < \textit{nome tipovalore} > = < \text{tipo} >
\end{array}
$$

$$
\begin{array}{rcl}
< \text{def-classe-prim} > & ::= & \textbf{prim} < \textit{nome classe} > = < \text{classe} > \\
< \text{def-classe-virt} > & ::= & \textbf{virt} < \textit{nome classe} > = < \text{classe} > \\
< \text{def-antecedente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{def-conseguente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{tipo} > & ::= & \#\textbf{top}\# \mid \\
 & & < \text{insiemi-di-tipi} > \mid \\
 & & < \text{esiste-insiemi-di-tipi} > \mid \\
 & & < \text{sequenze-di-tipi} > \mid \\
 & & < \text{ennuple} > \mid \\
 & & < \text{nomi-di-tipi} > \mid \\
 & & < \text{tipo-cammino} > \\
< \text{classe} > & ::= & < \text{insiemi-di-classi} > \mid \\
 & & < \text{esiste-insiemi-di-classi} > \mid \\
 & & < \text{sequenze-di-classi} > \mid \\
 & & < \text{nomi-di-classi} > \mid \\
 & & \hat{} < \text{tipo} > \mid \\
 & & < \text{nomi-di-classi} > \ \& \ \hat{} < \text{tipo} > \\
< \text{insiemi-di-tipi} > & ::= & \{ < \text{tipo} > \} \mid \\
 & & \{ < \text{tipo} > \} \ \& \ < \text{insiemi-di-tipi} > \mid \\
 & & \{ < \text{tipobase} > \} \\
< \text{esiste-insiemi-di-tipi} > & ::= & !\{ < \text{tipo} > \} \mid \\
 & & !\{ < \text{tipo} > \} \ \& \ < \text{esiste-insiemi-di-tipi} > \mid \\
 & & !\{ < \text{tipobase} > \} \\
< \text{sequenze-di-tipi} > & ::= & \langle \ < \text{tipo} > \ \rangle \mid \\
 & & \langle \ < \text{tipo} > \ \rangle \ \& \ < \text{sequenze-di-tipi} > \mid \\
 & & \langle \ < \text{tipobase} > \ \rangle \\
< \text{ennuple} > & ::= & [< \text{attributi} >] \mid \\
 & & [< \text{attributi} >] \ \& \ < \text{ennuple} > \\
< \text{attributi} > & ::= & < \textit{nome attributo} > : < \text{desc-att} > \mid \\
 & & < \textit{nome attributo} > : < \text{desc-att} > , < \text{attributi} >
\end{array}
$$

$$
\begin{array}{rcl}
< \text{desc-att} > & ::= & < \text{tipobase} > \ | \\
 & & < \text{tipo} > \ | \\
 & & < \text{classe} > \\
< \text{nomi-di-tipi} > & ::= & < \textit{nome tipovalore} > \ | \\
 & & < \textit{nome tipovalore} > \ \& \ < \text{nomi-di-tipi} > \\
< \text{tipo-cammino} > & ::= & (< \textit{nome attributo} > : < \text{desc-att} >) \\
< \text{insiemi-di-classi} > & ::= & \{ < \text{classe} > \} \ | \\
 & & \{ < \text{classe} > \} \ \& \ < \text{insiemi-di-classi} > \\
< \text{esiste-insiemi-di-classi} > & ::= & !\{ < \text{classe} > \} \ | \\
 & & !\{ < \text{classe} > \} \ \& \ < \text{esiste-insiemi-di-classe} > \\
< \text{sequenze-di-classi} > & ::= & \langle < \text{classe} > \rangle \\
 & & \langle < \text{classe} > \rangle \ \& \ < \text{sequenze-di-classi} > \\
< \text{nomi-di-classi} > & ::= & < \textit{nome classe} > \ | \\
 & & < \textit{nome classe} > \ \& \ < \text{nomi-di-classi} > \\
< \text{def-tipobase} > & ::= & \textbf{btype} < \textit{nome tipobase} > = \ < \text{tipobase} > \\
< \text{tipobase} > & ::= & \textbf{real} \ | \\
 & & \textbf{integer} \ | \\
 & & \textbf{string} \ | \\
 & & \textbf{boolean} \ | \\
 & & < \text{range-intero} > \ | \\
 & & \textbf{vreal} < \textit{valore reale} > \ | \\
 & & \textbf{vinteger} < \textit{valore intero} > \ | \\
 & & \textbf{vstring} < \textit{valore string} > \ | \\
 & & \textbf{vboolean} < \text{valore-boolean} > \ | \\
 & & < \textit{nome tipobase} > \\
< \text{valore boolean} > & ::= & \textbf{true} \ | \ \textbf{false} \\
< \text{range-intero} > & ::= & \textbf{range} < \textit{valore intero} > \ +\textbf{inf} \ | \\
 & & -\textbf{inf} \ < \textit{valore intero} > \ | \\
 & & < \textit{valore intero} > \ < \textit{valore intero} >
\end{array}
$$

Bibliografia

[A+89] M. Atkinson et al. The object-oriented database system manifesto. In *1nd Int. Conf. on Deductive and Object-Oriented Databases*. Springer-Verlag, 1989.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD*, pages 238–247. ACM Press, 1991.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173. ACM Press, 1989.

[App96] P. Apparuti. Progetto e realizzazione di un'interfaccia oql per l'ottimizzazione di interrogazioni su basi di dati ad oggetti odbq-optimizer. Tesi di Laurea, Univeristá di Modena Facoltá di Ingegneria, corso di laurea in Informatica, 1996.

[Atz93] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg - Germany, 1993.

[AV97] S. Abiteboul and V. Vianu. Regular path queries with constraint. In *PODS - Principles of Database Systems*. ACM Press, 1997.

[Baa91] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *12th International Joint Conference on Artificial Intelligence.*, Sydney, Australia, 1991.

[Bal92] J.P. Ballerini. Odl-designer: un sistema per il progetto automatico di schemi di basi di dati ad oggetti complessi. Tesi di Laurea, Facoltà di Scienze MM. FF. NN., Corso di Laurea in Scienze dell'Informazione, Università di Bologna, 1992.

[BB93] J. P. Ballerini and S. Bergamaschi. Automatic building and validation of complex object database schemata. Technical Report 5/124, CNR - Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, July 1993.

[BB97] Domenico Beneventano and Sonia Bergamaschi. Incoherence and subsumption for recursive views and queries in object-oriented data models. *Data and Knowledge Engineering*, 21(3):217–252, February 1997.

[BBG+96] D. Beneventano, S. Bergamaschi, A. Garuti, C. Sartori, and M. Vincini. ODB-QOptimizer: un Ottimizzatore Semantico di interrogazioni per OODB. In *Terzo Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD95, Salerno*, 1996.

[BBLS94] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Reasoning with constraints in database models. Technical Report 103, CIOC - CNR, Viale Risorgimento, 2 Bologna, Italia, September 1994.

[BBMR89] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD*, pages 58–67, Portland, Oregon, 1989.

[BBS94] D. Beneventano, S. Bergamaschi, and C. Sartori. Using subsumption for semantic query optimization in OODB. In *Int. Workshop on Description Logics*, volume D-94-10 of *DFKI - Document*, pages 97–100, Bonn, Germany, June 1994.

[BBSV97] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in oodb. In *Int. Conference on Data Engineering - ICDE97*, 1997.

[Bee90] C. Beeri. Formal models for object-oriented databases. In W. Kim, J. M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 405–430, B.V. - North-Holland, 1990. Elsevier Science Publisher.

[BGN89] H.W. Beck, S.K. Gala, and S.B. Navathe. Classification as a query processing technique in the CANDIDE data model. In *5th Int. Conf. on Data Engineering*, pages 572–581, Los Angeles, CA, 1989.

[BJNS94] M. Buchheit, M. A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented database. In *EDBT*, pages 348–353, 1994.

[BLR97] Catriel Beeri, Alon Y. Levy, and Marie-Christine Rousset. Rewriting queries using views in description logics. In ACM, editor, *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, pages 99–108, New York, NY 10036, USA, 1997. ACM Press.

[BM92] E. Bertino and D. Musto. Query optimization by using knowledge about data semantics. *Data and Knowledge Engineering*, 9(2):121–155, December 1992.

[BN94a] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[BN94b] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[BNPS92] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Trans. Knowl. and Data Engineering*, 4(3):223–236, June 1992.

[BS91] S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. In *ACM Transaction on Database Systems*, volume 17, pages 23–27, September 1991.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[CW91] N. Coburn and G. E. Weddel. Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations and functional dependencies. In *2nd Int. Conf. on Deductive and Object-Oriented Databases*, pages 312–331, Heidelberg, Germany, December 1991. Springer-Verlag.

[D.B94] D.Beneventano. *Uno strumento di inferenza nelle basi di dati ad oggetti: la sussunzione*. PhD thesis, Dip. di Elettronica, Informatica e Sistemistica, Università di Bologna, Bologna, 1994.

[DD89] L. M. L. Delcambre and K. C. Davis. Automatic validation of object-oriented database structures. In *5th Int. Conf. on Data Engineering*, pages 2–9, Los Angeles, CA, 1989.

[FS86] T. Finin and D. Silverman. Interactive classification as a knowledge acquisition tool. In L. Kershberg, editor, *Expert Database Systems*, pages 79–90. The Benjamin/Cummings Publishing Company, 1986.

[Gar95] A. Garuti. Ocdl-designer: un componente software per il controllo di consistenza di schemi di basi di dati ad oggetti con vincoli di integritá. Tesi di Laurea, Univeristá di Bologna, Facoltá di scienze MM. FF. NN. corso di laurea in Informatica, 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk - 80: the language and its implementation*. Addison∼Wesley, 1983.

[HZ80] M. M. Hammer and S. B. Zdonik. Knowledge based query processing. In *6th Int. Conf. on Very Large Databases*, pages 137–147, 1980.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KC86] S. Khoshafian and G. Copeland. Object identity. In *1st Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Portland, 1986.

[Kim89] W. Kim. A model of queries for object-oriented database systems. In *Int. Conf. on Very Large Databases*, Amsterdam, Holland, August 1989.

[Kin81a] J. J. King. *Query optimization by semantic reasoning*. PhD thesis, Dept. of Computer Science, Stanford University, Palo Alto, 1981.

[Kin81b] J. J. King. QUIST: a system for semantic query optimization in relational databases. In *7th Int. Conf. on Very Large Databases*, pages 510–517, 1981.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD '92*, pages 393–402. ACM, June 1992.

[KL86] W. Kim and F. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading (Mass.), 1986.

[LR89] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.

[SO89] S. Shenoy and M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Trans. Knowl. and Data Engineering*, 1(3):344–361, September 1989.

[Vin94] M. Vincini. Odb-qoptimizer: un ottimizzatore semantico di interrogazioni. Tesi di Laurea, Univeristá di Modena, Facoltá di Ingegneria, corso di laurea in Ingegneria Informatica, 1994.

[WS92] W. A. Woods and J. G. Schmolze. The KL-ONE family. In F. W. Lehman, editor, *Semantic Networks in Artificial Intelligence*, pages 133–178. Pergamon Press, 1992. Pubblished as a Special issue of *Computers & Mathematics with Applications*, Volume 23, Number 2-9.