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

Progetto e realizzazione di un'interfaccia OQL per l'Ottimizzatore di interrogazioni su Basi di Dati ad Oggetti ODBQ-Optimizer

Tesi di Laurea di
Apparuti Paolo

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Correlatore
Dott. Ing. Domenico Beneventano

Anno Accademico 1995 - 96

Parole chiave:
Basi di dati ad oggetti
ODMG-93
Schemi di basi di dati

Ai miei genitori

RINGRAZIAMENTI

Ringrazio la Prof.ssa Sonia Bergamaschi, il Prof. Claudio Sartori, il Dott. Ing. Domenico Beneventano, il P.H.D. Ing. Maurizio Vincini, la Dott.ssa Alessandra Garuti e il P.H.D. Ing. Stefano Lodi per il prezioso aiuto fornito nella realizzazione della presente tesi.

Indice

1 Introduzione **1**

1.1 I modelli di dati orientati ad oggetti 2

1.2 I linguaggi di interrogazione 4

1.2.1 Obiettivi 5

1.2.2 Controllo di correttezza di un'interrogazione 7

1.2.3 Contenuto della tesi 8

2 Uno standard per basi di dati ad oggetti: ODMG-93 **9**

2.1 Aspetti fondamentali dello standard 10

2.1.1 Obiettivi 10

2.1.2 Architettura 10

2.2 Il modello ad oggetti 12

2.2.1 Il sistema dei tipi 13

2.3 Esempio: il dominio Magazzino 16

3 Il linguaggio OQL **19**

3.1 Introduzione 19

3.2 Aspetti principali del linguaggio 19

3.2.1 Input e Output di una query OQL 20

3.2.2 Identità degli oggetti 22

3.2.3 Invocazione di metodi 25

3.2.4 Polimorfismo 25

3.2.5 Composizione degli operatori 27

3.3 Definizione del linguaggio 28

3.3.1 Tipi compatibili 28

4 ODB-Tools **31**

4.1 Aspetti generali 32

4.2 Esempio: regole di integrità sul dominio Magazzino 34
4.3 OCDL: Un formalismo per Oggetti Complessi e Vincoli di Integrità 38
4.3.1 Schema e Istanza del Database 38
4.3.2 Sussunzione ed Espansione Semantica di un tipo 42
4.4 ODB-Tools: gli strumenti software 44

5 OQL e OCDL 47
5.1 La traduzione OQL-OCDL 47
5.1.1 Esempi di interrogazioni OQL traducibili in OCDL . . 48
5.1.2 Individuazione dei costrutti traducibili 59
5.1.3 Espressioni non tradotte 65
5.1.4 Problemi aperti 66
5.2 La traduzione da OCDL a OQL 72
5.2.1 Traduzione dei costrutti OCDL 79

6 Il software 83
6.1 Concetti Base 84
6.2 Generatori di parser 85
6.3 Progettazione 87
6.3.1 Informazioni semantiche 87
6.3.2 Type Checking 88
6.3.3 Symbol Table 90
6.3.4 Errori 93
6.4 L'interfaccia OQL-OCDL 94
6.4.1 Il modello DFD 95
6.4.2 Architettura 96
6.4.3 Azioni semantiche 100
6.4.4 Eliminazione dei fattori 101
6.4.5 Strutture Dati 101

Elenco delle figure

2.1 Gerarchia dei tipi del Modello ad Oggetti di ODMG-93 15

4.1 Relazione di sussunzione dovuta alle regole 43
4.2 Architettura funzionale di OCDL-Designer 44
4.3 Architettura funzionale di ODB-QOptimizer 45

6.1 Symbol Table. 93
6.2 Architettura. 95
6.3 Simboli del modello DFD. 96
6.4 Schema DFD dell'architettura del sistema 97
6.5 Schema DFD della funzione "interfaccia". 98
6.6 Funzione "traduzione OCDL→OQL". 100
6.7 Esempio di codifica. 106

Elenco delle tabelle

2.1 Lo schema del Magazzino in sintassi ODMG–93 17

4.1 Schema con Regole di Integrità in linguaggio OCDL 40

6.1 Record semantico utilizzato dal traduttore. 102

Capitolo 1

Introduzione

L'introduzione del paradigma ad oggetti nel campo delle basi di dati ha costituito, negli ultimi anni, uno dei campi di ricerca più importanti per la comunità scientifica. Il cambiamento introdotto non costituisce infatti una semplice evoluzione rispetto alle precedenti basi di dati relazionali, ma si configura come un'innovazione rivoluzionaria. Nonostante ciò, i sistemi di gestione di basi di dati orientate ad oggetti (OODBMS) non hanno ancora avuto una larghissima diffusione in campo commerciale, dove subiscono la concorrenza dei vecchi sistemi relazionali che per un ventennio hanno dominato il mercato. Uno dei punti di forza di questi sistemi è costituito dal linguaggio di interrogazione.

Lo scopo della presente tesi è quello di realizzare un componente software in grado di interpretare e di generare la traduzione di interrogazioni espresse in due differenti linguaggi per basi di dati ad oggetti.

Il lavoro di tesi rientra in un progetto più ampio, denominato ODB-Tools, volto alla realizzazione di un sistema per la progettazione assistita di basi di dati ad oggetti e l'ottimizzazione semantica di interrogazioni. Base di questo progetto è il modello di dati ad oggetti complessi OCDL (*Object Constraint Description Logics*). Esso consente di descrivere schemi di OODB unitamente ad un insieme rilevante di *vincoli d'integrità*, offrendo un controllo di coerenza e minimalità dello schema introdotto. OCDL, inoltre, rende possibile l'ottimizzazione semantica delle interrogazioni, utilizzando i vincoli di integrità per trasformare un'interrogazione in una equivalente con un *minor costo di esecuzione*.

Un'importante esigenza del progetto è quella di fornire un'interfaccia alle funzionalità del sistema realizzato, in modo da renderlo utilizzabile anche

dall'utente che non conosce il modello interno. Per non legare il sistema ad un particolare ODBMS commerciale è stato scelto di rendere tale interfaccia conforme alla proposta di standard ODMG-93, che si appresta a diventare uno standard di fatto del mercato. Il software oggetto del lavoro di tesi permette di formulare interrogazioni sul sistema ODBSQO utilizzando il linguaggio OQL (*Object Query Language*) definito da ODMG-93.

1.1 I modelli di dati orientati ad oggetti

Intuitivamente un modello dei dati è uno strumento concettuale che consente al progettista di attribuire un certo significato (o interpretazione) ai dati e di manipolare i dati stessi. Si può affermare che la vera e propria teoria delle basi di dati come disciplina informatica è nata con la nozione di modello dei dati.

I modelli di dati orientati ad oggetti nascono come risposta ad una serie di esigenze che i modelli precedenti non riuscivano a soddisfare: maggiore uniformità di trattazione di dati ed operazioni, maggiore separazione tra livello implementativo e livello di interfaccia, costituzione di librerie di oggetti che rendono semplice ed efficiente il riutilizzo del software, semplificare la realizzazioni di applicazioni complesse che trattano grandi quantità di dati. Questi modelli hanno trovato nei sistemi di gestione di basi di dati un importante campo applicativo.

Di seguito vengono presentate le principali caratteristiche di un OODBMS:

- **Oggetti e identità**: ogni entità del mondo reale è modellata come un oggetto, il quale presenta un insieme di attributi e operazioni. Ogni oggetto possiede un identificatore, detto *oid* (*Object Identifier*), che non dipende dal valore rappresentato. Due oggetti possono quindi essere *identici* (cioè sono lo stesso oggetto) oppure possono essere uguali (rappresentano lo stesso valore ma hanno un diverso *oid*).

- **Incapsulazione**: ogni oggetto ha un'interfaccia e un'implementazione. L'interfaccia, che costituisce la parte visibile dell'oggetto, definisce l'insieme delle operazioni (i *metodi*) che possono essere invocati sull'oggetto. L'implementazione è costituita da una struttura dati (gli *attributi*), che rappresenta lo *stato* dell'oggetto, e da un insieme di procedure che ne definiscono il *comportamento*. Le operazioni consentono

di variare lo stato di un oggetto. Gli oggetti con una struttura e un comportamento comune sono compresi in una *classe*.

- **Ereditarietà**: ogni classe (o tipo) può essere definita come specializzazione di una o più classi (o tipi) esistenti, ereditandone metodi e attributi.

- **Estendibilità**: un OODBMS presenta un insieme di tipi e classi predefiniti che puó essere sempre arricchito, senza che ci sia nessuna distinzione tra tipi o classi predefiniti e nuove classi o tipi creati dall'utente.

Nel lavoro presentato in questa tesi si è interessati esclusivamente ai concetti di tipo, ereditarietà e identità di oggetto, che costituiscono elementi fondamentali nel controllo di correttezza delle interrogazioni. Nel seguito vengono quindi esposti con maggior dettaglio questi concetti, trascurando, anche a causa di alcune limitazioni del modello OCDL, gli aspetti legati all'incapsulamento, ai metodi e alla modularizzazione.

Oggetti e identità

Ogni elemento componente la realtà da modellare viene rappresentato tramite un unico concetto di base: l'*oggetto*. Ogni oggetto è univocamente individuato da un identificatore di oggetto, *oid*, ed ha associato uno stato che è costituito dal valore delle sue *proprietà* o *attributi*. Nei classici linguaggi orientati agli oggetti, come ad esempio Smalltalk [GR83], il valore associato ad un oggetto è atomico oppure è una ennupla di altri oggetti. Questa caratteristica è stata in parte ereditata da alcuni ODBMS, dove il valore di un oggetto 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], indicati con CODMs (*Complex Object Data Models*). 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 lavori citati 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. Tuttavia, mentre l'estensione denotata da un tipo è definita dalla sua struttura (cioè tutti gli elementi di un dominio la cui struttura coincide con quella di un dato tipo sono istanze di quel tipo), l'estensione associata ad una classe è definita dall'utente (cioè gli elementi del dominio devono essere esplicitamente inseriti fra le istanze di una classe). Ad ogni classe è normalmente associato un tipo il quale 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.

Ereditarietà

L'*ereditarietà*, stabilita tramite la relazione *isa*, è un'importante caratteristica degli OODB. Essa costituisce un potente mezzo di modellazione, essendo in grado di dare una precisa e concisa descrizione del dominio di applicazione [A$^+$89]. Dal punto di vista intensionale, cioè delle strutture di concetti e delle relazioni tra concetti, 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. Dal punto di vista estensionale, la dichiarazione A *isa* B, stabilisce che ogni oggetto di A sia anche un oggetto di B. Si parla di *ereditarietà multipla* nel caso in cui sia ammesso che una classe possa avere più di una superclasse.

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

1.2 I linguaggi di interrogazione

Gli OODMBS, poichè utilizzano un modello ad oggetti, trattano in maniera uniforme i dati e le operazioni che agiscono sui dati stessi. Per questo motivo ogni OODMBS viene fornito di un potente linguaggio interno di programmazione che consente di scrivere intere applicazioni associate ai dati. Questo fatto, pur costituendo un evidente vantaggio rispetto ai sistemi relazionali, non ha risolto il problema del linguaggio di interrogazione, in quanto un

linguaggio completo di programmazione è uno strumento troppo complesso per poter consentire di formulare interrogazioni a sé stanti, al di fuori delle applicazioni, come ad esempio le query interattive. Un linguaggio di interrogazione dovrebbe presentare quindi delle caratteristiche risultanti da un compromesso tra semplicità e potenza, in particolare:

- *Alto livello.* Le interrogazioni devono poter essere formulate in maniera semplice e concisa. Un linguaggio non procedurale, che permette un accesso dichiarativo agli oggetti, favorisce questo aspetto.
- *Orientamento al recupero di oggetti.* Lo scopo del linguaggio di interrogazione è quello di estrarre oggetti da una base di dati. Dovrebbe quindi permettere di formulare un ragionevole insieme di query in maniera semplice, trovando un giusto compromesso tra semplicità e potenza. La completezza del linguaggio secondo quanto definito per i sistemi relazionali è sufficiente per soddisfare questo requisito.
- *Facilità di ottimizzazione.* Il sistema deve poter trovare una buona strategia per ottimizzare le query dell'utente.
- *Indipendenza dalle applicazioni.* Un linguaggio di interrogazione non deve essere legato ad una specifica classe di applicazioni.

Il secondo punto, in particolare, mette in rilievo una distinzione tra linguaggi di programmazione e di interrogazione, in relazione alla completezza computazionale. Una linea di sviluppo nell'area delle basi di dati ha cercato di estendere i linguaggi di interrogazione per ottenere strumenti completi dal punto di vista computazionale. Lo svantaggio di un tale approccio risiede però nell'eccessiva complessità che presenta un linguaggio completo, come può essere inteso un linguaggio di programmazione, quando viene utilizzato in modo interattivo oppure per formulare query specifiche. Un secondo approccio vede il linguaggio di interrogazione come uno strumento semplice ma potente, adatto all'uso interattivo, con funzionalità diverse dal linguaggio applicativo, lasciando a quest'ultimo i compiti più complessi.

1.2.1 Obiettivi

Per i linguaggi di interrogazione che non prevedono le estensioni tipiche dei linguaggi applicativi, si possono individuare quattro modi fondamentali di

impiego. I primi tre sono gli stessi usati attualmente per SQL, il quarto è invece tipico dei linguaggi object-oriented:

Formulazione di query specifiche

La formulazione di query specifiche o in maniera interattiva è il principale sistema di utilizzo di un linguaggio di interrogazione. Gli utenti del sistema devono poter disporre di uno strumento potente, e allo stesso tempo facile da usare (almeno per query non troppo complesse), per estrarre i dati dal database. Queste considerazioni valgono anche nel caso degli ODBMS, in quanto in questi sistemi, specialmente per i compiti più semplici, l'utente può utilizzare il linguaggio di interrogazione al posto di quello di programmazione.

Accesso alla base di dati

Nei sistemi relazionali l'unico modo di accedere ai dati è attraverso il linguaggio di interrogazione. SQL è stato quindi progettato anche per essere utilizzato in maniera *embedded*, ovvero dall'interno di programmi applicativi scritti in C, Cobol ecc. Questo utilizzo presenta una serie di svantaggi in relazione alla diversità di tipi dato esistenti in SQL e nel linguaggio ospite, oltre ad uno scarso grado di integrazione. Normalmente, nel campo dei database orientati ad oggetti, questa funzionalità non è richiesta, in quanto il linguaggio di programmazione fornito dal sistema e il DML sono integrati.

Realizzazione di semplici programmi

La semplicità di utilizzo del linguaggio di interrogazione, unitamente alla sua potenza, rende a volte vantaggioso il suo utilizzo nella scrittura delle applicazioni meno complesse, in luogo di un linguaggio applicativo vero e proprio.

Manipolazione di dati dall'interno del linguaggio applicativo

A volte il linguaggio di interrogazione può essere utile per esprimere query molto complesse in maniera più semplice di quanto si possa fare attraverso il linguaggio di programmazione. Questo punto riguarda espressamente i sistemi orientati ad oggetti.

1.2.2 Controllo di correttezza di un'interrogazione

Un'interrogazione, prima di essere effettivamente eseguita, subisce un processo, suddiviso in più fasi, che porta alla formulazione di un piano di accesso al database. La prima di queste fasi è costituita dal controllo di correttezza. Tale controllo, in generale, riguarda la sintassi dell'interrogazione, ma può essere esteso, entro certi limiti, anche alla semantica. Riuscire a stabilire fin dal momento della compilazione, con un elevato grado di precisione, se un'interrogazione è corretta, evita, in caso di errore, di compiere inutilmente le costose fasi successive (traduzione in linguaggio a basso livello, scelta degli indici ecc.). Occorre notare però che mentre per il controllo sintattico di grammatiche non eccessivamente complesse sono stati individuati opportuni strumenti che ne consentono una trattazione efficace, per quanto riguarda il controllo semantico al momento della compilazione può essere effettuata solamente un'analisi statica.

Un'informazione fondamentale per l'analisi semantica statica di un'interrogazione è quella riguardante il tipo degli oggetti coinvolti. Tale informazione non sarebbe strettamente necessaria a tempo di compilazione, dove è sufficiente conoscere la *struttura* dei dati, ma risulta importante sia dal punto di vista del controllo di correttezza che da quello del miglioramento delle prestazioni. Nel primo caso rende possibile controllare che gli operandi di ogni operatore siano del tipo corretto. Nel secondo caso si osserva che, conoscendo il tipo esatto di un oggetto, è possibile compilare le operazioni invocate direttamente in codice a basso livello, ottenendo così una forma di ottimizzazione.

L'informazione riguardante il tipo può essere espressa direttamente dall'utente nel testo della query, oppure può essere ricavata dallo schema della base di dati. Tra le due possibilità la seconda appare del tutto preferibile, in quanto semplifica notevolmente la formulazione di una query, evitando all'utente di specificare il tipo di ogni oggetto nominato.

In generale il controllo di tipo potrebbe essere eseguito a tempo di compilazione per le interrogazioni *embedded* e a tempo di esecuzione per le interrogazioni interattive. Poichè in questo lavoro di tesi si è realizzato un modulo di un sistema in cui le interrogazioni non sono effettivamente eseguite, il *type-checking* è stato implementato nella fase di traduzione.

1.2.3 Contenuto della tesi

.... Quando avrò scritto il resto.

Capitolo 2

Uno standard per basi di dati ad oggetti: ODMG-93

ODMG–93 è una proposta di standard per sistemi di gestione di basi di dati orientate ad oggetti (ODBMS) realizzato dal gruppo di lavoro ODMG (Object Database Management Group), in cui sono rappresentate alcune tra le maggiori aziende di produzione di software e di ODBMS. ODMG–93 cerca in questo modo di colmare una lacuna, la mancanza di uno standard, che si è rivelata uno dei principali ostacoli ad una più larga diffusione degli ODBMS. Esaminando ciò che è avvenuto nel settore dei DBMS relazionali non si può infatti ignorare l'importanza che ha rivestito SQL come linguaggio standard di interrogazione: esso ha favorito l'adozione o l'aggiornamento dei DBMS relazionali da parte delle aziende, poichè semplifica l'apprendimento del nuovo sistema da parte del personale e garantisce un notevole grado di portabilità delle applicazioni preesistenti.

La proposta attuale di ODMG, pur costituendo un sforzo creativo notevole che cambierà in maniera significativa l'industria dei database ad oggetti, non deve tuttavia essere considerata un risultato conclusivo, bensì verrà sottoposta nel corso degli anni a continue revisioni. Inizialmente si è infatti scelto di supportare soltanto quelle funzionalità che sono già implementate o implementabili nel breve periodo dai produttori di ODBMS, allo scopo di far comparire in poco tempo sul mercato prodotti conformi allo standard. Nuove funzionalità saranno via via aggiunte nelle successive versioni.

2.1 Aspetti fondamentali dello standard

2.1.1 Obiettivi

L'obiettivo fondamentale di ODMG è quello di definire un insieme di standard che consentano al cliente di un ODBMS di produrre applicazioni in grado di essere eseguite anche su piattaforme diverse da quella per cui sono state inizialmente sviluppate. La portabilità deve essere garantita quindi sotto diversi aspetti, in particolare:

- *schema della base di dati.*
- *linguaggi di interrogazione e manipolazione dei dati.*
- *collegamenti con i linguaggi di programmazione.*

Appare chiaro che il grado di standardizzazione che si cerca di raggiungere va ben oltre quello ottenuto dall'SQL in campo relazionale: non si vuole portare da un ODBMS all'altro soltanto quella parte di applicazione che fa uso di costrutti *embedded*, bensì l'obiettivo è quello di poter addattare, con uno sforzo relativamente piccolo, l'intero codice sorgente dell'applicazione. La presenza in ODMG di persone appartenenti ad aziende leader nel campo dei database ad oggetti, unitamente al fatto che la proposta del gruppo si basa in larga misura sui prodotti maggiormente diffusi, fa prevedere che ODMG-93 si imporrà come standard "di fatto" del mercato.

2.1.2 Architettura

I DBMS orientati ad oggetti presentano una architettura sostanzialmente diversa dagli altri DBMS, in particolare possiedono tutti un linguaggio di programmazione completo che consente all'utente di scrivere intere applicazioni, a differenza di quanto avviene con i linguaggi di interrogazione per sistemi relazionali. Questa caratteristica fa sì che ci sia una forte integrazione tra le funzionalità del database e il linguaggio applicativo, ottenendo vantaggi sia dal punto di vista della rappresentazione dei dati [1] che da quello

[1] Basti pensare, ad esempio, ai problemi che insorgono quando si cerca di mappare i tipi di dato disponibili in un DBMS relazionale nei corrispondenti forniti dal linguaggio C. Tale difficoltà si incontra ogni volta che si realizza una applicazione che fa uso di istruzioni SQL *embedded*, e si ripresenta ogni volta che si cerca di ricompilare il codice sorgente su una piattaforma diversa.

della performance nella gestione dei dati stessi, rendendo trasparenti il trasferimento e la copia degli oggetti tra la base di dati e l'applicazione. Si può quindi pensare che il DBMS a oggetti estenda il linguaggio di programmazione con un insieme più vasto di funzionalità, tra le quali si sottolineano: gestione di oggetti persistenti, memoria secondaria, concorrenza, sicurezza e integrità dei dati, possibilità di formulare interrogazioni in modo interattivo ecc. Il gruppo ODMG ha quindi definito le modalità con cui un ODBMS deve integrarsi con alcuni linguaggi di programmazione *object-oriented* esistenti, prevedendo inoltre un certo livello di compatibilità con SQL per quanto riguarda il linguaggio di interrogazione, in modo da renderne più immediata la comprensione. La proposta di standard riguarda i seguenti componenti fondamentali:

- *Modello ad oggetti.* Il modello dei dati utilizzato dagli ODBMS conformi allo standard è basato su di un nucleo, definito in precedenza dall'Object Management Group (OMG), che costituisce un denominatore comune per sistemi di basi di dati ad oggetti, linguaggi di programmazione ad oggetti ed altre applicazioni. A tale nucleo sono stati aggiunti i componenti per implementare le funzionalità proprie di ODMG-93.

- *Linguaggio di definizione degli oggetti.* Anche per questo componente, chiamato Object Definition Language (ODL), è stata utilizzata come base una precedente specifica di OMG, denominata Interface Definition Language (IDL).

- *Linguaggio di interrogazione.* Il linguaggio di interrogazione, denominato Object Query Language (OQL), utilizza come base l'SQL relazionale estendendolo e potenziandolo con una serie di funzionalità mirate alla trattazione degli oggetti. Per alcuni operatori è accettata sia la sintassi originale SQL che una sintassi propria dei linguaggi di interrogazione object-oriented.

- Collegamento con il linguaggio C++. Il linguaggio maggiormente utilizzato nella programmazione di applicazioni per ODBMS è il linguaggio C++. Lo standard ODMG-93 definisce per tale linguaggio l'interfacciamento con l'ODBMS, un meccanismo per invocare OQL, le specifiche per scrivere applicazioni portabili.

- *Collegamento con il linguaggio Smalltalk.* L'interfacciamento con il linguaggio Smaltalk è stato definito per rendere standard un insieme di

applicazioni specifiche, per le quali tale linguaggio risulta essere il più appropriato.

2.2 Il modello ad oggetti

In questa sezione viene fornita una breve descrizione del modello ad oggetti definito da ODMG-93, evidenziando gli aspetti che riguardano il concetto di tipo e di classe, in quanto essi rivestono un ruolo fondamentale nell'effettuazione del controllo di correttezza di una interrogazione, che verrà trattato in seguito. Il modello ad oggetti implementato da ODBMS che rispettano le specifiche ODMG-93 presenta le seguenti caratteristiche fondamentali:

- La primitiva di base del modello è l'*oggetto*.

- Gli oggetti sono suddivisi in categorie dette *tipi*. Tutti gli oggetti di un dato tipo presentano lo stesso comportamento e possono assumere stati appartenenti ad uno stesso insieme.

- Il *comportamento* di un oggetto è definito dall'insieme delle *operazioni* che possono essere eseguite sull'oggetto stesso.

- Lo *stato* di un oggetto, ad un dato istante, è definito dal valore delle sue *proprietà*. Le proprietà di un oggetto si distinguono in *attributi* dell'oggetto stesso e in *associazioni* tra l'oggetto dato ed uno o più altri oggetti.

Tipo

Un *tipo* rappresenta la struttura comune di un insieme di oggetti con le stesse caratteristiche e corrisponde alla nozione di tipo di dato astratto. Ogni tipo ha quindi una *interfaccia* ed una o più *implementazioni*. L'interfaccia definisce la parte visibile delle istanze di un dato tipo, ovvero le proprietà e le operazioni che si possono invocare, mentre l'implementazione definisce le *strutture dati*, che costituiscono la rappresentazione fisica delle istanze di un dato tipo, e i *metodi*, che descrivono, in un qualunque linguaggio di programmazione, l'implementazione delle operazioni definite nell'interfaccia. Nel modello definito da ODMG anche i tipi sono considerati oggetti e di conseguenza presentano delle proprietà, tra cui è importante sottolineare:

- *Supertipo.* Un tipo oggetto può essere definito come specializzazione di uno o più tipi esistenti (*supertipi*) dai quali eredita l'intero insieme di attributi, associazioni e operazioni. Il *sottotipo* così definito può operare una specializzazione delle proprietà ereditate, restringendo il range degli stati e delle operazioni ammesse per le sue istanze, oppure può aggiungere nuove proprietà a quanto ereditato. Le relazioni che intercorrono tra i tipi oggetto possono quindi essere rappresentate tramite un grafo *sottotipo/supertipo*.

- *Estensione.* L'insieme di tutte le istanze di un dato tipo è chiamata *estensione*.

- *Chiave.* Le proprietà o l'insieme di proprietà che individuano univocamente ogni istanza di un tipo costituiscono una *chiave*. Un tipo può avere nessuna, una o più chiavi; in ogni caso gli oggetti istanze di quel tipo *non* sono identificati nel sistema attraverso il valore della chiave.

Classe

Una *classe* è la combinazione dell'interfaccia di un tipo con una delle implementazioni definite per quello stesso tipo. La possibilità di definire più implementazioni per la stessa interfaccia consente di ottenere vantaggi sia sotto il profilo della realizzazione dell'ODBMS che dal punto di vista della programmazione delle applicazioni. Nel primo caso è possibile supportare più facilmente sistemi distribuiti su macchine con architetture diverse e ambienti di sviluppo diversi. Dal punto di vista del programmatore è invece possibile specificare, al momento della creazione di ogni singolo oggetto, quale implementazione utilizzare, in modo da privilegiare alcune caratteristiche (performance, occupazione di memoria ecc.) a seconda dell'uso che si intende fare dell'oggetto che viene creato. In assenza di questa possibilità il programmatore sarebbe costretto a definire tanti sottotipi quante sono le diverse implementazioni che deve utilizzare per un dato tipo.

2.2.1 Il sistema dei tipi

La gerarchia dei tipi definita da ODMG-93 è rappresentata in Figura 2.1. La relazione tra supertipi e sottotipi viene evidenziata, nella figura, ricorrendo a successivi livelli di indentazione, ognuno dei quali rappresenta un ulteriore grado di specializzazione rispetto ai precedenti. Di seguito vengono descritte

brevemente le caratteristiche dei tipi che rivestono maggior interesse in relazione alla formulazione di interrogazioni mediante il linguaggio OQL, uno dei principali componenti dello standard. Il sistema dei tipi prevede una prima distinzione fondamentale tra *tipi oggetto*, i quali possiedono una identità intrinseca, e *tipi caratteristica*, le cui istanze possono essere identificate soltanto facendo riferimento all'oggetto a cui appartengono. I tipi oggetto possono essere suddivisi ortogonalmente in due modi:

mutable/immutable Si definiscono *mutable* i tipi le cui istanze possono variare, mantenendo la propria identità. Appartengono a questa categoria, ad esempio, gli oggetti complessi definiti come collezioni di altri oggetti: aggiungendo o togliendo un elemento dall'insieme l'identità dell'oggetto collezione non varia. Si definiscono invece *immutable* i tipi le cui istanze, una volta create, non possono essere aggiornate. Appartengono a questa categoria gli oggetti di tipo stringa, intero, ecc.

L'identità degli oggetti di tipo *mutable* è rappresentata attraverso uno speciale identificatore detto *OID*, mentre per gli oggetti di tipo *immutable* l'identificazione avviene attraverso la *codifica* del valore che rappresentano. In Figura 2.1 (e nel seguito) si fa riferimento ai tipi *mutable* e *immutable* utilizzando i termini, rispettivamente, Object e Literal.

atomic/structured I tipi atomici sono i tipi base supportati dal modello (*Atomic_Literal*), le variabili, le eccezioni e il tipo *Type*, che fornisce l'accesso ai *metadati*. In particolare tutti i tipi rappresentati in Figura 2.1 sono istanze del tipo *Type*. I tipi strutturati si suddividono in collezioni, ordinate e non ordinate, e strutture.

Tipi strutturati

I tipi strutturati possono appartenere alle categorie Structured_Object e Structured_Literal. Tutte e due le categorie presentano collezioni e strutture come sottotipi, differenziandosi unicamente per quanto detto in precedenza a proposito dei tipi *mutable/immutable*. Il tipo collezione viene specializzato nei tipi Set, Bag, List e Array, che si differenziano nell'ordinamento, nella possibilità di contenere oggetti duplicati e nella possibilità di accedere agli elementi tramite indice. Gli oggetti che appartengono ad una collezione devono essere dello stesso tipo oppure di tipo compatibile, e possono essere in numero

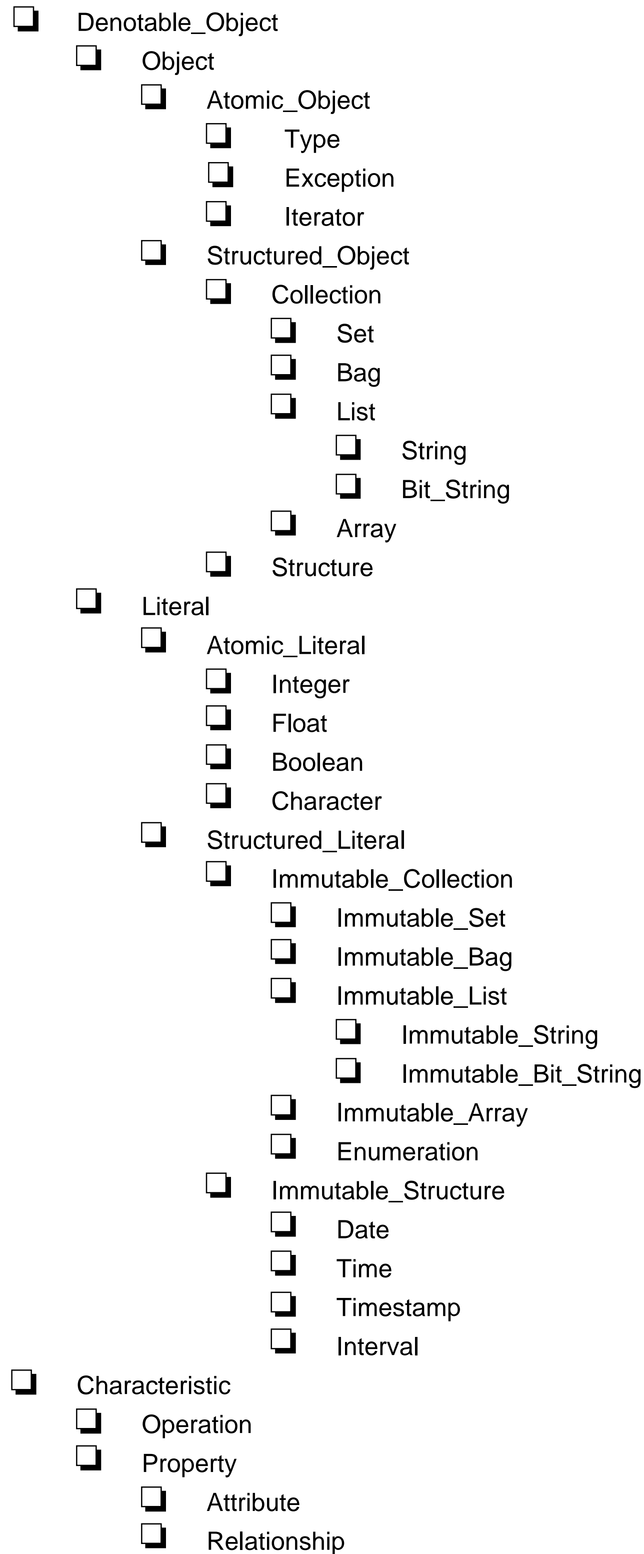

Figura 2.1: Gerarchia dei tipi del Modello ad Oggetti di ODMG-93

qualunque. Il tipo struttura ha invece un numero definito di campi, ognuno di nome diverso e, solitamente, anche di tipo diverso.

Variabili

Ogni tipo collezione possiede un'operazione in grado di definire una variabile (il cui supertipo è Iterator) che può assumere l'identità di un qualunque elemento dell'insieme. Il tipo Iterator possiede poi altre operazioni per spostarsi all'interno della collezione su cui è definito, rendendo possibili procedimenti iterativi sull'insieme. Se la collezione non è ordinata l'ordine di scansione è arbitrario.

Il modello ad oggetti definito da ODMG-93 è fortemente tipizzato, ovvero ogni oggetto ha un tipo ed ogni operazione invocabile richiede operandi di un dato tipo. La compatibilità tra tipi viene definita nei due paragrafi seguenti.

Compatibilità tra tipi *Object*

Tipi con nomi diversi che possiedono lo stesso insieme di proprietà e lo stesso insieme di operazioni sono diversi, di conseguenza due oggetti sono dello stesso tipo soltanto se il tipo di cui sono istanze è il medesimo. Due tipi sono compatibili se rispetto alla gerarchia di ereditarietà hanno un supertipo in comune, oppure se uno è una specializzazione dell'altro.

Compatibilità tra tipi *Literal*

I tipi atomici appartenenti a questa categoria sono dello stesso tipo se appartengono allo stesso insieme (interi, decimali ecc.). I tipi strutturati sono dello stesso tipo se, ricorsivamente, ad ogni livello presentano la stessa struttura e se ogni parte atomica è dello stesso tipo. Tipi *Literal* e tipi *Object* non sono compatibili.

2.3 Esempio: il dominio Magazzino

Per terminare questo capitolo presentiamo, in Tabella 2.1 un esempio di schema in sintassi ODMG–93, che verrà utilizzato nei capitoli successivi per la formulazione di interrogazioni. L'esempio considerato è relativo alla struttura organizzativa di una società: i materiali (`Material`) sono descritti da un nome, un rischio e da un insieme di caratteristiche; gli `SMaterial` sono

Classi dello Schema

```
interface Material
{
   attribute string  name;
   attribute integer risk;
   attribute string  code;
   attribute set <string> feature;
};
interface DMaterial: Material
{
   attribute range  {15, 100} risk;
};
interface SMaterial: Material{ };
interface Storage
{
   attribute string category;
   attribute Manager managed_by;
   attribute set< struct {
         attribute Material item;
         attribute range {10, 300} qty; } > stock;
   attribute integer maxrisk;
};
interface Manager
{
   attribute string  name;
   attribute range  {40K, 100K} salary;
   attribute range  {1, 15} level;
};
interface TManager: Manager
{
   attribute range  {8, 12} level;
};
interface SStorage: Storage{ };
interface DStorage: Storage
{
   attribute set < struct {
         attribute DMaterial item; } > stock;
};
```

Tabella 2.1: Lo schema del Magazzino in sintassi ODMG–93

un sottoinsieme dei `Material`. I `DMaterial` (materiali "pericolosi") sono un sottoinsieme dei `Material`, caratterizzati da un rischio compreso tra 15 e 100. I manager hanno un nome, un salario compreso tra 40K e 100K dollari e un livello compreso tra 1 e 15. I `TManager` sono manager che hanno un livello compreso tra 8 e 12. I magazzini (`Storage`) sono descritti da una categoria, sono diretti (`managed_by`) da un manager e contengono (`stock`) un insieme di articoli (`item`) che sono dei materiali, per ciascuno dei quali è indicata la quantità presente; è inoltre riportato il rischio massimo (`maxrisk`) ammissibile per i materiali conservati. Gli `SStorage` sono un sottoinsieme dei magazzini. Infine vi sono i magazzini "pericolosi" (`DStorage`) che sono un sottoinsieme degli `Storage` caratterizzati dallo stoccaggio di soli materiali "pericolosi" (`DMaterial`). Si noti che le classi che abbiamo descritto sono tutte di tipo "base", esprimono cioè solo condizioni necessarie per l'appartenenza di un oggetto alla classe.

Capitolo 3

Il linguaggio OQL

3.1 Introduzione

In questo capitolo viene descritto il linguaggio Object Query Language (OQL), il componente dello standard ODMG-93 utilizzato per l'interrogazione di basi di dati orientate ad oggetti. La versione di OQL che viene descritta è diversa da quella presentata in [Cat94] poiché comprende una serie di aggiornamenti, pubblicati separatamente da ODMG, che compariranno nella prossima versione dello standard.

Di seguito vengono illustrate le caratteristiche principali di OQL, in particolare: input e output di un'interrogazione, identità degli oggetti, navigazione attraverso le associazioni, invocazione di metodi, polimorfismo, composizione degli operatori. Gli esempi utilizzati si riferiscono allo schema di tabella 2.1.

3.2 Aspetti principali del linguaggio

Il linguaggio OQL si basa sui seguenti principi ed assunzioni:

- OQL è basato sul modello ad oggetti definito da ODMG.

- OQL utilizza una sintassi simile a quella definita per SQL 92. Rispetto a questa presenta delle estensioni finalizzate alla gestione degli aspetti object-oriented, in particolare gli oggetti complessi, l'identità degli oggetti, le espressioni di percorso, il polimorfismo, l'invocazione delle operazioni e il late binding.

- OQL fornisce delle primitive ad alto livello per manipolare insiemi di oggetti, ma non è strettamente legato a questo costrutto. Fornisce infatti anche le primitive per gestire, con la stessa efficienza, altri tipi strutturati come array, liste e strutture.

- OQL è un linguaggio funzionale in cui gli operatori possono essere liberamente composti, a patto di rispettare la compatibilità tra tipi, secondo quanto previsto dal sistema dei tipi di ODMG-93.

- OQL non è completo dal punto di vista computazionale. Come visto in precedenza ciò non rientra negli obiettivi del linguaggio.

- OQL può essere invocato dall'interno di applicazioni scritte nei linguaggi per cui lo standard ODMG-93 prevede l'interfacciamento. Inoltre eventuali operazioni codificate in questi linguaggi possono essere invocate dall'interno di interrogazioni.

- OQL non fornisce in modo esplicito operatori per l'aggiornamento della base di dati, lasciando questo compito a operazioni opportune che fanno parte delle caratteristiche di ogni oggetto che popola il database. In questo modo si rispetta la semantica propria degli ODBMS che, per definizione, vengono gestiti attraverso i metodi definiti sugli oggetti.

- OQL permette di accedere agli oggetti in maniera dichiarativa. Questa caratteristica rende più immediata la formulazione dell'interrogazione da parte dell'utente e più semplice ottimizzare le interrogazioni.

3.2.1 Input e Output di una query OQL

Utilizzato come un linguaggio indipendente, OQL consente di estrarre gli oggetti da una base di dati attraverso il loro *nome*, che agisce come entry-point. Un nome denota un qualunque tipo di oggetto, sia esso atomico, strutturato, letterale, collezione ecc. Questo è il principale modo di impiego del linguaggio.

Utilizzato invece in maniera *embedded*, OQL permette di utilizzare le query, all'interno del linguaggio di programmazione, come *funzioni* che restituiscono oggetti. Il tipo degli oggetti estratti, che può essere dedotto dall'operatore OQL che denota l'interrogazione, deve poter essere gestito dal linguaggio ospite.

Fatte le precedenti precisazioni possiamo osservare che, in generale, un'interrogazione OQL estrae oggetti da una collezione che corrisponde all'estensione di una classe. In particolare, nel linguaggio di definizione dello schema in sintassi ODMG-93, occorre dichiarare il nome dell'estensione di ogni tipo.

Si noti però che lo schema ODMG-93 presentato in Tabella 2.1 rappresenta solo la struttura delle classi, senza estensioni, poiché nello schema corrispondente in linguaggio OCDL ad un nome di classe viene assegnata semplicemente una descrizione.

Facendo riferimento a tale schema, quindi, la seguente interrogazione in linguaggio OQL

```
select *  from Material
```

non sarebbe intesa sui dati (come avviene generalmente nei sistemi relazionali) ma sui metadati: infatti restituirebbe in uscita la struttura della classe Material [1].

Per poter introdurre degli oggetti nelle classi definite nello schema di Tabella 2.1, e quindi effettuare in seguito delle interrogazioni su tali oggetti, occorre aggiungere a tale schema le dichiarazioni dei nomi delle *estensioni* delle classi:

```
interface Material( extent Materials )
{
   attribute string  name;
   attribute integer risk;
   attribute string  code;
   attribute set <string> feature;
};
```

Supponiamo ad esempio, di istanziare[2] un oggetto di tipo `Material`:

```
Material( name: "M1", risk: 20, code: "C 60")
```

Volendo estrarre dal database l'oggetto istanziato si potrebbe ricorrere all'interrogazione seguente:

[1] Il manuale di OQL cui facciamo riferimento non tratta la struttura del risultato di questo tipo di interrogazione. Di conseguenza omettiamo la relativa rappresentazione.

[2] La costruzione di oggetti verrà trattata nella sezione seguente.

```
select * from Materials where name = "M1"
```

il cui risultato sarebbe appunto l'oggetto **M1**. Si noti che tale interrogazione ha per oggetto l'estensione della classe `Material`.

Nel seguito, allo scopo di semplificare la notazione utilizzata, un'interrogazione OQL espressa su un nome di classe si intende riferita direttamente all'estensione della classe; non verranno pertanto prese in considerazione le dichiarazioni dei nomi delle estensioni delle classi. Questa semplificazione è possibile in quanto l'ottimizzazione riguarda esclusivamente interrogazioni sui dati e inoltre nel formalismo interno (nel quale verranno successivamente tradotte le interrogazioni) le classi sono descritte attraverso un unico nome al quale viene associata sia una descrizione (attraverso il concetto di *schema*) sia un'estensione (tramite la nozione di *interpretazione*). Ad esempio, l'interrogazione OQL:

```
select * from Material
```

si intende riferita agli oggetti istanze della classe Material.

Nel seguito vengono presentate le caratteristiche più importanti di OQL.

3.2.2 Identità degli oggetti

Questo linguaggio di interrogazione è in grado di gestire sia oggetti, identificati da una proprietà detta *Object Identifier* (OID), sia letterali, oggetti identificati dal valore che rappresentano, a seconda di come questi vengono selezionati o costruiti dall'interrogazione.

Creazione di oggetti

La creazione di oggetti che possiedono un OID è ottenuta mediante un costruttore che ha lo stesso nome del tipo dell'oggetto che deve essere creato. Tra parentesi possono essere specificati i valori di alcuni attributi che devono essere inizializzati all'atto della creazione. Ad esempio, per creare un'istanza del tipo `Storage` si può utilizzare l'interrogazione

```
Material( name: "steel", risk: 20 )
```

Gli attributi non specificati assumono un valore di default.

I letterali vengono invece creati utilizzando i costruttori forniti direttamente dal linguaggio, ad esempio `struct`, `bag`, ecc.

```
struct( number: 10, word: "Pat" )
```

La creazione di oggetti senza *oid* può avvenire anche in maniera implicita, come evidenziato in una delle successive sezioni a proposito dell'operatore `select`.

Selezione di oggetti esistenti

Le espressioni che estraggono oggetti dal database possono restituire:

- una collezione di oggetti con OID. Ad esempio l'interrogazione `select M from Material M where M.name = "steel"` restituisce una collezione di materiali di nome "steel".
- un oggetto con identità. Ad esempio la query seguente `element(select M from Material M where M.code = "C 60")` restituisce l'unico materiale il cui codice è "C 60".
- una collezione di letterali. Ad esempio l'interrogazione `select M.code from Material M where M.name = "steel"` restituisce una collezione di stringhe di caratteri corrispondenti al codice di ogni materiale di nome "steel".
- un letterale. Ad esempio la query seguente `element(select M.risk from Material M where M.code = "C 60")` restituisce il numero intero corrispondente al rischio dell'unico materiale il cui codice è "C 60".

Una query può quindi restituire un oggetto (o un insieme di oggetti) con o senza identità; alcuni di questi oggetti possono essere generati dall'interprete del linguaggio, se l'interrogazione fa uso di determinati costruttori, altri possono essere estratti direttamente dalla base di dati.

Espressioni di percorso

Come detto in sezione 3.2.1 l'accesso alla base di dati può avvenire tramite il nome degli oggetti. In generale emerge però la necessità di *navigare* attraverso la gerarchia delle classi nello schema, per raggiungere altri oggetti che costituiscono l'obiettivo dell'interrogazione. OQL prevede quindi la notazione "." (o indifferentemente "$\rightarrow$") per accedere alle proprietà di oggetti

complessi e per attraversare semplici associazioni. Ad esempio, la query seguente parte dalla variabile `S` di tipo `Storage`, accede, tramite l'attributo `managed_by`, all'oggetto `Manager`, di cui ottiene il nome:
Esempio

```
S.managed_by.name
```

Questo esempio presenta una associazione, `Storage.managed_by`, di tipo 1-1. Occore sottolineare che in OQL un'espressione di percorso può essere usata solo per associazioni di molteplicità 1-1 oppure m-1, poichè per le associazioni di tipo 1-m oppure m-n la notazione fornita sarebbe ambigua. Se, ad esempio, l'interrogazione `S.stock.qty` fosse ammessa, il risultato della query sarebbe un insieme di interi, dato che `S.stock` rappresenta una collezione. Possono però presentarsi dei casi meno intuitivi, in cui non è chiaro a quali elementi del percorso è dovuta la molteplicità del risultato e per i quali è necessaria una notazione non ambigua. Per navigare attraverso associazioni multiple di questo tipo OQL estende, rispetto all'SQL relazionale, la sintassi della clausola `from` nel costrutto `select-from-where`.
Esempio

```
select  X.qty
from    S.stock X
```

Per navigare attraverso percorsi che fanno uso di più attributi multivalore OQL consente di specificare nella parte `from` dell'operatore `select-from--where` più collezioni, ognuna delle quali è rappresentata da un cammino che deriva da quelli che lo precedono nella lista dichiarativa.
Esempio

```
select  X.item.risk
from    Storage S,
        S.stock X
```

Predicati

La parte `where` dell'operatore `select-from-where`, definendo una serie di predicati booleani, permette di effettuare una restrizione degli oggetti appartenenti alle collezioni interrogate. Vengono quindi selezionati solo i dati che soddisfano l'espressione specificata. L'interrogazione seguente, ad esempio, seleziona tra tutti i materiali soltanto quelli con rischio inferiore a 20 e ne riporta il nome:

```
select  M.name from Material M where M.risk < 20
```

Join

In OQL è possibile formulare interrogazioni che riguardano più classi non direttamente correlate da un cammino, effettuando una operazione di *join* in maniera simile a quanto avviene in SQL. Questo esempio seleziona i manager che hanno lo stesso nome di un materiale.
Esempio

```
select  X
from    Manager  X,
        Material M
where   X.name = M.name
```

3.2.3 Invocazione di metodi

I metodi possono essere invocati utilizzando la stessa notazione usata per navigare le associazioni o per accedere agli attributi di una classe. L'invocazione avviene specificando il nome del metodo ed, eventualmente, un elenco di parametri fra parentesi. Nel caso questi non siano previsti nella *signature* del metodo, è possibile trattare le proprietà di un oggetto in maniera uniforme: l'utente, potendole trattare con una identica sintassi, non è tenuto a sapere se le proprietà di un oggetto a cui accede sono calcolate mediante operazioni oppure sono memorizzate in attributi. Un metodo può restituire un oggetto complesso e può quindi essere impiegato in qualunque punto di una complicata espressione di percorso. Supponiamo, ad esempio, che la classe `Storage` abbia un metodo `max_stock` che restituisce il materiale stoccato in maggior quantità. Si può allora formulare un'interrogazione che seleziona il nome di questo materiale per ogni magazzino:

```
select  S.max_stock.name
from    Storage S
```

3.2.4 Polimorfismo

Uno dei maggiori contributi portati dal paradigma ad oggetti è costituito dalla possibilità di gestire collezioni polimorfiche e di invocare le operazioni

generiche degli elementi che ne fanno parte grazie al meccanismo di *late-binding*. Una interrogazione formulata sull'estensione di una classe data si applica quindi anche agli oggetti delle sottoclassi che specializzano la classe di partenza. Gli oggetti di una sottoclasse possono perciò essere manipolati sia interrogando direttamente la sottoclasse stessa sia formulando una query su di una superclasse, applicando un'opportuna restrizione sui dati ottenuti. Questa seconda possibilità si esprime in OQL tramite un apposito operatore detto *indicatore di classe*. Occorre inoltre sottolineare che l'estensione di una classe, quando viene specificata in una query OQL, identifica *staticamente* il tipo degli oggetti ad essa appartenenti. Questo significa che se gli oggetti fanno parte anche di una sottoclasse della classe di partenza, non è possibile accedere agli attributi e ai metodi propri delle istanze del tipo più specializzato, poichè a tempo di compilazione fallirebbe il meccanismo di type-checking. Esistono soltanto due eccezioni a quanto detto, trattate in seguito: late-binding di metodi e indicazione esplicita della classe.

Late Binding

Quando viene invocato un metodo l'azione eseguita non dipende soltanto dal nome del metodo ma anche dal tipo dell'oggetto da cui è invocato, per cui il codice da eseguire viene determinato a tempo di esecuzione (*late-binding*) e non al momento della compilazione della query. L'azione eseguita è sempre quella definita nella classe di cui l'oggetto è istanza, anche se nella query OQL viene inizialmente specificata una classe diversa, meno specializzata.

Indicazione esplicita della classe

OQL fornisce un operatore specifico per dichiarare esplicitamente la classe di appartenenza di un oggetto. Vi sono infatti dei casi in cui il tipo di un oggetto non può essere dedotto staticamente, cioè a tempo di compilazione, ma deve essere indicato nel testo della query. Questa eventualità si presenta nei casi in cui l'utente sa che gli oggetti cui si fa riferimento nella query non appartengono alla classe dichiarata, bensì ad una sua specializzazione, perché, ad esempio, vengono applicate determinate restrizioni sui dati. Ad esempio, nell'interrogazione seguente è stato specificato esplicitamente che i magazzini con rischio massimo superiore a 20 sono anche istanze del tipo `SStorage`.

```
select  ((SSstorage)S).maxrisk
```

```
from    Storage S
where   S.makrisk > 20
```

3.2.5 Composizione degli operatori

OQL è un linguaggio puramente funzionale, in cui ogni operatore viene visto come una interrogazione che restituisce un risultato di un determinato tipo. Per questo motivo tutti gli operatori possono essere composti in qualunque modo purché venga sempre rispettato il sistema dei tipi. OQL è quindi, a differenza dell'SQL relazionale, un linguaggio completamente ortogonale. Questa caratteristica permette di non limitare la potenza delle espressioni e rende il linguaggio più semplice da apprendere, poiché mantiene la stessa sintassi di SQL per le interrogazioni più semplici. Le espressioni SQL che non rientrano in una categoria puramente funzionale, e che quindi non rientrano nella filosofia di OQL, sono viste come variazioni sintattiche dei costrutti OQL equivalenti. Tra queste espressioni poniamo in evidenza, a titolo di esempio e poiché comparirà spesso nei capitoli successivi, la `select-list` del filtro `select-from-where`. La forma generale dell'espressione è

select *proiezione* { , proiezione} ...

dove per *proiezione* si intende uno dei costrutti seguenti

1. espressione[3] **as** identificatore
2. identificatore: espressione
3. espressione

Questo modo di specificare la proiezione, attraverso un elenco di espressioni, consente di accettare la sintassi SQL. Nel linguaggio OQL però, per ottenere lo stesso risultato, bisognerebbe utilizzare il costruttore di struttura (record):

struct(identificatore: espressione { , identificatore: espressione})

In OQL il problema viene risolto ammettendo entrambe le sintassi. Semplicemente l'operatore `select` costruisce *implicitamente* la struttura quando viene utilizzato con la sintassi SQL. Se però viene specificata soltanto una

[3] il termine espressione è utilizzato come sinonimo di interrogazione OQL.

proiezione utilizzando la sintassi (3) la struttura non viene creata, in quanto non è prevista nemmeno in OQL. In questo modo la sintassi SQL viene vista come una “abbreviazione” dell'espressione funzionale corrispondente in linguaggio OQL.

3.3 Definizione del linguaggio

Una interrogazione OQL è un'espressione ottenuta componendo ricorsivamente gli operatori del linguaggio ed in cui ogni operando possiede un tipo ben preciso. In questa sezione si utilizzerà quindi il termine *espressione* come sinonimo di query. Ogni espressione restituisce come risultato un oggetto (atomico o collezione, con o senza identità) di cui è noto il tipo. Questo tipo può essere rilevato staticamente, tranne nei casi trattati in sezione 3.2.4, osservando i seguenti elementi:

- *struttura dell'espressione.* Il tipo dell'espressione cambia a seconda degli operatori utilizzati.
- *dichiarazione dei tipi nello schema.* La correttezza dell'interrogazione rispetto al sistema dei tipi viene verificata senza che l'utente debba dichiarare esplicitamente il tipo degli operandi. Tale informazione viene ottenuta dallo schema della base di dati.
- *tipo degli oggetti coinvolti.* Ogni espressione può utilizzare oggetti con o senza identità.

Per determinare se una espressione OQL è corretta occorre conoscere l'insieme di "regole" semantiche che definiscono ciascun operatore. Per brevità non riportiamo tali regole, diciamo soltanto che sono state implementate nel software realizzato in questo lavoro di tesi, descritto nel capitolo 6. Riteniamo invece più importante definire in maniera più rigorosa il concetto di compatibilità tra i tipi presenti nel sistema.

3.3.1 Tipi compatibili

La compatibilità tra tipi viene definita in maniera ricorsiva nel modo seguente:

1. `t` è compatibile con `t`

2. se `t` è compatibile con $\mathtt{t}'$ allora

   `set(t)` è compatibile con $\mathtt{set}(\mathtt{t}')$

   `list(t)` è compatibile con $\mathtt{list}(\mathtt{t}')$

   `bag(t)` è compatibile con $\mathtt{bag}(\mathtt{t}')$

   `array(t)` è compatibile con $\mathtt{array}(\mathtt{t}')$

3. se esiste `t` tale che `t` è un supertipo di $\mathtt{t}_1$ e $\mathtt{t}_2$ allora $\mathtt{t}_1$ e $\mathtt{t}_2$ sono compatibili.

Questa definizione implica che:

- tipi letterali non sono compatibili con tipi oggetto (oggetti con identità).
- tipi letterali atomici sono tra loro compatibli solo se sono dello stesso tipo.
- tipi letterali strutturati sono tra loro compatibili soltanto se hanno un supertipo comune.
- tipi letterali collezione sono tra loro compatibili se il tipo di collezione è il medesimo e se risulta compatibile il tipo dei loro membri.
- tipi oggetto atomici sono compatibili solo se hanno un supertipo comune.
- tipi oggetto collezione sono compatibili solo se il tipo della collezione è lo stesso e se il tipo dei loro membri è compatibile.

Si noti che se $\mathtt{t}_1$, $\mathtt{t}_2$, $\cdots$, $\mathtt{t}_\mathtt{n}$ sono compatibili, allora esiste un unico `t` tale che:

1. $\mathtt{t} > \mathtt{t}_\mathtt{i}$ per ogni i
2. per ogni $\mathtt{t}'$ tale che $\mathtt{t}' \neq \mathtt{t}$ e $\mathtt{t}' > \mathtt{t}_\mathtt{i}$ per ogni i, $\mathtt{t}' > \mathtt{t}$

In altre parole, il tipo `t` è il *least upper bound* dell'insieme $\mathtt{t}_1$, $\mathtt{t}_2$, $\cdots$, $\mathtt{t}_\mathtt{n}$ rispetto all'ordinamento parziale $>$.
La funzione che restituisce il tipo `t` è indicata con $\mathtt{lub}(\mathtt{t}_1, \mathtt{t}_2, \cdots, \mathtt{t}_\mathtt{n})$.

Capitolo 4

ODB-Tools

In questo capitolo viene presentato ODB-Tools, un sistema per l'acquisizione e la verifica di consistenza di schemi e per l'ottimizzazione semantica di interrogazioni nelle Basi di Dati Orientate agli Oggetti (OODB) sviluppato presso il Dipartimento di Scienze dell'Ingegneria (DSI) dell'Università di Modena. Il contenuto di questo capitolo riassume il lavoro presentato in [BN94, JBBV95, Ben94, Vin94, Gar95] e viene riportato per motivi di completezza.

L'ambiente teorico su cui è basato ODB-Tools include due elementi fondamentali. Il primo è *ODL* (*Object Description Logics*), proposto come formalismo comune per esprimere descrizioni di classi, vincoli d'integrità ed interrogazioni e dotato di tecniche di inferenza basate sul calcolo della *sussunzione* introdotte per le *Logiche Descrittive* nell'ambito dell'Intelligenza Artificiale. Il secondo elemento è costituito dal concetto di *espansione semantica* di un tipo e dalla sua computazione attraverso la sussunzione.

Il capitolo è organizzato come segue. In sezione 4.2 viene riportato un esempio che mostra sia l'attivita' di OCDL-Designer che quella di ODB-QOptimizer. In particolare, sono anche riportate alcune idee preliminari relative all'eliminazione dall'interrogazione di fattori ridondanti; questa attività risulta utile sia per l'ottimizzazione, in quanto l'eliminazione di un fattore può ridurre il numero di classi da visitare per risolvere l'interrogazione, sia per filtrare dall'interrogazione semanticamente estesa i fattori inutili. In sezione 4.3 viene brevemente riportato il formalismo ODL esteso con i path ed i vincoli di integrità e vengono formalizzate la sussunzione e l'espansione semantica dei tipi. In sezione 4.4 vengono descritti i componenti del sistema ODB-Tools.

4.1 Aspetti generali

Il formalismo ODL è stato proposto in [BN94] 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].

Le principali caratteristiche di ODL sono brevemente riportate nel seguito. In ODL viene assunta una ricca struttura per il *sistema di tipi atomici*; oltre ai tipi atomici *integer*, *boolean*, *string*, *real* e *tipi mono-valore*, viene considerata anche la possibilità di utilizzare dei sottoinsiemi di tipi atomici, come ad esempio intervalli di interi. Partendo da questi tipi atomici si possono creare *tipi valori*, attraverso gli usuali costruttori di tipo definiti nei CODMs, quali *tuple* ed *insiemi*, e *tipi classe*, che denotano insiemi di oggetti con una identità ed un valore associato. Ai tipi può essere assegnato un nome, con la distinzione fra nomi per tipi–valore e nomi per tipi–classe (chiamati semplicemente *classi*). Ciò equivale a dire che i nomi di tipi sono partizionati fra quelli che rappresentano insiemi di oggetti e quelli che rappresentano insiemi di valori. L'ereditarietà, sia semplice che multipla, è espressa direttamente nella descrizione di una classe tramite l'operatore di *intersezione*.

ODL introduce inoltre la distinzione tra nomi di tipi *virtuali*, che descrivono condizioni necessarie e sufficienti di appartenenza di un oggetto del dominio ad un tipo, e nomi di tipi *base*, che descrivono condizioni necessarie di appartenenza. In altri termini, l'appartenenza di un elemento all'interpretazione di un nome di tipo base deve essere stabilita esplicitamente, mentre l'interpretazione dei nomi di tipo virtuale è calcolata.

In [JBBV95] 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 [BN94] 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 [JBBV95] l'ambiente teorico sviluppato in [BN94] è stato esteso per effettuare l'ottimizzazione semantica delle interrogazioni, dando vita al sistema ODB-QOptimizer.

Nel presente lavoro di tesi il componente ODB-QOptimizer viene integrato da un'interfaccia che permette la formulazione delle interrogazioni in un linguaggio, denominato OQL, proposto dal gruppo di standardizzazione ODMG–93 [Cat94]. Il software realizzato fa parte di un sistema, denominato ODB-Tools, che comprende i componenti ODB-QOptimizer e OCDL-Designer e li fornisce di interfacce conformi ad ODMG–93. In particolare è in fase di realizzazione il modulo che rende compatibile con lo standard il linguaggio di definizione dello schema.

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 trasformazione: 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.

4.2 Esempio: regole di integrità sul dominio Magazzino

Per illustrare il nostro metodo, consideriamo l'esempio del dominio Magazzino presentato nel capitolo 2 a cui aggiungiamo alcuni vincoli d'integrità per garantire la consistenza della base di dati. Le regole di integrità, espresse in una sintassi simile a ODMG–93 sono le seguenti:

rule R1 **for all** `X` **in** `Material` (`X.risk` ≥ 10)

```
          then    X in SMaterial
rule R2   for all X in Storage (
                  for all X1 in X.stock (X1.item in SMaterial))
          then    X in SStorage
rule R3   for all X in Storage (X.managed_by.level ≥ 6 and
                                X.managed_by.level ≤ 12)
          then    X.category = "A2"
rule R4   for all X in Storage (X.category = "A2" and
                  exists X1 in X.stock (X1.item.risk ≥ 10))
          then    X.managed_by in SMaterial)
```

La regola R1 stabilisce che, per un `Material`, avere un rischio superiore a 10 è condizione sufficiente per essere considerato anche un `SMaterial`. Nello stesso modo, la regola R2 forza gli `Storage` in cui *tutti* gli articoli immagazzinati sono `SMaterial` ad essere anche `SStorage`. R3 dice che i magazzini guidati da un manager con livello compreso tra 6 e 12 debbono appartenere alla categoria “A2”. Infine, R4 forza i magazzini con categoria di tipo “A2” e contenenti *almeno* un materiale con rischio superiore a 10 ad essere guidati da un `TManager`.

L'attività di inferenza iniziale da parte del sistema, effettuata da OCDL-Designer, consiste nel determinare l'espansione semantica per ogni classe dello schema. Questo permette di rilevare relazioni *isa* non esplicitate nella definizione delle classi stesse. Ad esempio, nel calcolo dell'espansione semantica della classe `DMaterial` si rileva che tale classe è sussunta (implica) l'antecedente della regola R1 quindi congiungendone il conseguente si determina che `DMaterial` è una specializzazione di `SMaterial`. Nel caso in cui vengono applicate più regole durante l'espansione semantica di una classe, le relazioni *isa* ricavate sono meno ovvie di quella appena descritta. Ad esempio, nel calcolo dell'espansione semantica della classe `DStorage` si applica prima la regola R1 e poi la regola R2, concludendo che `DStorage` è una specializzazione di `SStorage`. Si noti che queste due nuove relazioni di specializzazioni riguardano classi base.

L'altra componente del sistema, ODB-QOptimizer, si occupa dell'ottimizzazione semantica delle interrogazioni a run time; anche l'ottimizzazione dell'interrogazione è basata sull'espansione semantica, in modo da specializzare le classi interessate dall'interrogazione. Consideriamo ad esempio la seguente interrogazione:

Q_1 : “Seleziona i magazzini che contengono materiali che hanno tutti rischio maggiore o uguale a 15”.

Nel nostro sistema la query puo’ essere scritta in linguaggio OQL nel seguente modo :

```
select   * from  Storage S
where    for all X in S.stock : (  X.item in Material and
                                   X.item.risk ≥ 15 )
```

L’espansione semantica di quest’interrogazione, che si ottiene applicando prima la regola R1 e poi R2, porta alla seguente interrogazione equivalente:

```
select   * from  SStorage S
where    for all X in S.stock : (  X.item in SMaterial and
                                   X.item.risk ≥ 15 )
```

Questo esempio mostra un’effettiva ottimizzazione dell’interrogazione, indipendente da ogni specifico modello di costo, dovuta alla sostituzione delle classi presenti nell’interrogazione con loro specializzazioni. Infatti, sostituendo `Storage` con `SStorage` e `Material` con `SMaterial` si riduce l’insieme di oggetti da controllare per individuare il risultato dell’interrogazione.

Per illustrare la separazione di un’interrogazione nella parte *clean* e *dirty*, modifichiamo la precedente interrogazione:

Q_1' : “Seleziona i magazzini che contengono materiali che hanno tutti rischio maggiore o uguale a 15 e nei quali è presente almeno un materiale con rischio superiore al massimo ammissibile per il magazzino”.

```
select   * from  Storage S
where    for all X in S.stock :(  X.item in Material and
                                  X.risk ≥ 15 ) and
         exists Y in S.stock :    Y.item.risk > S.maxrisk
```

Il nuovo fattore aggiunto rappresenta la parte *dirty* dell’interrogazione che non può essere tradotta nel formalismo OCDL (essenzialmente perchè esprime un confronto tra due attributi); comunque, è importante notare che la parte *clean*, corrispondente a Q_1, può essere ottimizzata come illustrato in precedenza.

La possibilità di individuare, in una interrogazione espressa in linguaggio OQL, una parte *clean* esprimibile in OCDL e sulla quale è quindi effettuabile l’ottimizzazione semantica, costituisce un punto di partenza fondamentale

della presente tesi, il cui obiettivo principale è appunto quello di tradurre una interrogazione OQL in OCDL.

Un'altra forma di ottimizzazione altrettanto generale può essere ottenuta riducendo il numero di classi coinvolte nell'interrogazione. In letteratura [SO89, SSS92] questo problema è generalmente noto come *rimozione di vincoli*, cioè eliminazione di fattori ridondanti all'interno di una interrogazione.

In OCDL, i fattori eliminabili di un'interrogazione, cioè i fattori che sono logicamente implicati dagli altri fattori dell'interrogazione, possono essere individuati tramite la sussunzione. Ad esempio, consideriamo la seguente interrogazione:

Q_2 : "Seleziona i magazzini di categoria A2 diretti da un `TManager`".

```
select  * from Storage S
where   S.category = "A2"            and
        S.managed_by in TManager
```

In base alla regola R4 e alla descrizione di `TManager` si ottiene che il fattore (`category` = "A2") è implicato da (sussume) il resto dell'interrogazione e quindi può essere eliminato da Q_2.

Nei citati lavori sulla rimozione di vincoli, la ridondanza di un fattore è controllata rispetto all'interrogazione originale; questo controllo costituisce però soltanto una condizione sufficiente per l'eliminazione di un fattore. Allo scopo di ottenere condizioni necessarie e sufficienti noi proponiamo di controllare la ridondanza di un fattore rispetto all'espansione semantica dell'interrogazione. Questo è mostrato tramite il seguente esempio:

Q_3 : "Seleziona i magazzini diretti da un `TManager` e che contengono almeno un materiale con rischio maggiore o uguale a 10".

```
select  * from Storage S
where   S.managed_by in TManager                  and
        exists X in S.stock : X.item.risk ≥ 10
```

Analizzando l'eliminabilità del fattore (`managed_by in TManager`) si rileva che esso non sussume il resto dell'interrogazione e quindi non è ridondante rispetto a Q_3. Comunque, considerando l'espansione semantica dell'interrogazione (ottenuta applicando le regole R1, R3 e R4):

```
select   * from Storage S
where    category = "A2"             and
         S.managed_by in TManager    and
         exists X in S.stock :(      X.item in SMaterial and
                                     X.item.risk ≥ 10 )
```

il fattore (managed_by in TManager) risulta ridondante, in quanto sussume il resto dell'interrogazione; tale fattore può essere quindi eliminato, evitando l'accesso alla classe TManager.

Lo sviluppo futuro del sistema prevede di determinare i fattori ridondanti di una interrogazione direttamente dall'analisi della sua espansione semantica interrogazione e dalle regole applicate per ottenerle. Intuitivamente, in entrambi i due esempi di eliminazione, considerati il fattore eliminabile corrisponde ad un conseguente di una delle regole applicate durante il calcolo dell'espansione semantica dell'interrogazione.

4.3 OCDL: Un formalismo per Oggetti Complessi e Vincoli di Integrità

In questa sezione viene brevemente riportato OCDL (*Object Constraint Description Logics*), presentato in [JBBV95], che deriva da ODL [BN94] con in aggiunta i vincoli di integrità. In particolare, vengono formalmente definite la sussunzione e l'espansione semantica.

4.3.1 Schema e Istanza del Database

Sia **D** l'insieme infinito numerabile dei valori atomici (che saranno indicati con $d_1, d_2, \ldots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani. Sia **B** l'insieme di designatori di tipi atomici, con $\mathbf{B} = \{$`integer`, `string`, `boolean`, `real`, i_1–j_1, i_2–j_2, ..., $d_1, d_2, \ldots\}$, dove i d_k indicano tutti gli elementi di `integer` $\cup$ `string` $\cup$ `boolean` e dove gli i_k–j_k indicano tutti i possibili intervalli di interi (i_k può essere $-\infty$ per denotare il minimo elemento di `integer` e j_k può essere $+\infty$ per denotare il massimo elemento di `integer`).

Sia **A** un insieme numerabile di *attributi* (denotati da $a_1, a_2, \ldots$) e $\mathcal{O}$ un insieme numerabile di *identificatori di oggetti* (denotati da $o, o', \ldots$) disgiunti

da **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 $i \neq j$):

$$v \rightarrow d \mid o \mid \{v_1, \ldots, v_p\} \mid [a_1 : v_1, \ldots, a_p : v_p]$$

Gli identificatori di oggetti sono associati a valori tramite una *funzione totale* δ da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$; in genere si dice che il valore $\delta(o)$ è lo *stato* dell'oggetto identificato dall'oid o. Sia **N** l'insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che **A**, **B**, e **N** siano a due a due disgiunti. **N** è partizionato in tre insiemi **C**, **V** e **T**, dove **C** consiste di nomi per *tipi-classe base* $(C, C' \ldots)$, **V** consiste di nomi per *tipi-classe virtuali* $(V, V' \ldots)$, e **T** consiste di nomi per *tipi-valori* $(t, t', \ldots)$. Un *path* p è una sequenza di elementi $p = e_1 . e_2 . \ldots . e_n$, con $e_i \in \mathbf{A} \cup \{\triangle, \forall, \exists\}$. Con ϵ si indica il path vuoto. $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$[1] indica l'insieme di tutte le *descrizioni di tipo finite* $(S, S', \ldots)$, dette brevemente *tipi*, su di un dato $\mathbf{A}, \mathbf{B}, \mathbf{N}$, ottenuto in accordo con la seguente regola sintattica:

$$S \rightarrow \top \mid B \mid N \mid [a_1 : S_1, \ldots, a_k : S_k] \mid \forall\{S\} \mid \exists\{S\} \mid \triangle S \mid S \sqcap S' \mid (p : S)$$

$\top$ denota il *tipo universale* e rappresenta tutti i valori; [ ] denota il costruttore di tupla. $\forall\{S\}$ corrisponde al comune costruttore di insieme e rappresenta un insieme i cui elementi sono *tutti* dello stesso tipo S. Invece, il costruttore $\exists\{S\}$ denota un insieme in cui *almeno* un elemento è di tipo S. Il costrutto $\sqcap$ indica la *congiunzione*, mentre $\triangle$ è il costruttore di oggetto. Il tipo $(p : S)$ è detto *tipo path* e rappresenta una notazione abbreviata per i tipi ottenuti con gli altri costruttori.

Dato un dato sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, uno *schema* σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è una funzione totale da **N** a $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, che associa ai nomi di tipi la loro descrizione. Diremo che un nome di tipo N *eredita* da un altro nome di tipo N', denotato con $N \prec\!\mid_\sigma N'$, se $\sigma(N) = N' \sqcap S$. Si richiede che la relazione di ereditarietà sia priva di cicli, i.e., la chiusura transitiva di $\prec\!\mid_\sigma$, denotata $\prec$, sia un ordine parziale stretto.

Dato un dato sistema di tipi **S**, una *regole di integrità* R è espressa nella forma $R = S^a \rightarrow S^c$, dove S^a e S^c rappresentano rispettivamente l'antecedente e il conseguente della regola R, con $S^a, S^c \in \mathbf{S}$. Una regola R esprime il seguente vincolo: per tutti gli oggetti v, se v è di tipo S^a allora v deve essere di tipo S^c. Con **R** si denota un insieme finito di regole.

Uno *schema con regole* è una coppia $(\sigma, \mathbf{R})$, dove σ è uno schema e **R** un insieme di regole. Ad esempio, il dominio Magazzino rappresentato in

[1] In seguito, scriveremo **S** in luogo di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ quando i componenti sono ovvi dal contesto.

Tabella 2.1 ed esteso in sezione 4.2 con l'aggiunta di alcune regole di integrità ed alcune interrogazioni, è descritto in OCDL tramite lo schema con regole mostrato in Tabella 4.1. In particolare, si noti la rappresentazione delle interrogazione espresse in linguaggio OQL come classi virtuali OCDL: tale traduzione sarà discussa dettagliatamente nel prossimo capitolo.

$$
\begin{array}{l}
\mathbf{C} = \{\mathtt{Material}, \mathtt{SMaterial}, \mathtt{DMaterial}, \mathtt{Storage}, \mathtt{SStorage}, \mathtt{DStorage}, \mathtt{Manager}, \\
\quad \mathtt{TManager}\}, \\
\mathbf{V} = \{Q_1, Q_2, Q_3\}, \\
\mathbf{T} = \emptyset
\end{array}
$$

$$
\sigma \left\{
\begin{array}{lcl}
\sigma(\mathtt{Material}) & = & \triangle[\mathtt{name}\colon \mathtt{string}, \mathtt{risk}\colon \mathtt{integer}, \mathtt{code}\colon \mathtt{string}, \mathtt{feature}\colon \{\mathtt{string}\}] \\
\sigma(\mathtt{SMaterial}) & = & \mathtt{Material} \\
\sigma(\mathtt{DMaterial}) & = & \mathtt{Material} \sqcap \triangle[\mathtt{risk}\colon 15 \div 100] \\
\sigma(\mathtt{Storage}) & = & \triangle[\,\mathtt{managed_by}\colon \mathtt{Manager}, \mathtt{category}\colon \mathtt{string}, \mathtt{maxrisk}\colon \mathtt{integer}, \\
 & & \quad \mathtt{stock}\colon \{[\mathtt{item}\colon \mathtt{Material}, \mathtt{qty}\colon 10 \div 300]\}] \\
\sigma(\mathtt{SStorage}) & = & \mathtt{Storage} \\
\sigma(\mathtt{DStorage}) & = & \mathtt{Storage} \sqcap \triangle[\mathtt{stock}\colon \{[\mathtt{item}\colon \mathtt{DMaterial}]\}] \\
\sigma(\mathtt{Manager}) & = & \triangle[\mathtt{name}\colon \mathtt{string}, \mathtt{salary}\colon 40K \div 100K, \mathtt{level}\colon 1 \div 15] \\
\sigma(\mathtt{TManager}) & = & \mathtt{Manager} \sqcap \triangle[\mathtt{level}\colon 8 \div 12] \\
 & & \\
\sigma(Q_1) & = & \mathtt{Storage} \sqcap \triangle[\mathtt{stock}\colon \{[\mathtt{item}\colon \triangle[\mathtt{risk}\colon 15 \div \infty]]\}] \\
\sigma(Q_2) & = & \mathtt{Storage} \sqcap \triangle[\mathtt{category}\colon \text{``}\mathit{A2}\text{''}] \sqcap \triangle[\mathtt{managed_by}\colon \mathtt{TManager}] \\
\sigma(Q_3) & = & \mathtt{Storage} \sqcap \triangle[\mathtt{managed_by}\colon \mathtt{TManager}] \sqcap \\
 & & \qquad \triangle[\mathtt{stock}\colon \exists\{[\mathtt{item}\colon \triangle[\mathtt{risk}\colon 10 \div \infty]]\}]
\end{array}
\right.
$$

$$
\mathbf{R} \left\{
\begin{array}{ll}
R_1 : & \mathtt{Material} \sqcap (\triangle.\mathtt{risk}\colon 10 \div \infty) \rightarrow \mathtt{SMaterial} \\
R_2 : & \mathtt{Storage} \sqcap (\triangle.\mathtt{stock}.\forall.\mathtt{item}\colon \mathtt{SMaterial}) \rightarrow \mathtt{SStorage} \\
R_3 : & \mathtt{Storage} \sqcap (\triangle.\mathtt{managed_by}.\triangle\,\mathtt{level}\colon 6 \div 12) \rightarrow (\triangle.\mathtt{category}\colon \text{``}\mathit{A2}\text{''}) \\
R_4 : & \mathtt{Storage} \sqcap (\triangle.\mathtt{category}\colon \text{``}\mathit{A2}\text{''}) \sqcap (\triangle.\mathtt{stock}.\exists.\mathtt{item}.\triangle.\mathtt{risk}\colon 10 \div \infty) \\
 & \qquad\qquad \rightarrow \triangle[\mathtt{managed_by}\colon \mathtt{TManager}]
\end{array}
\right.
$$

Tabella 4.1: Schema con Regole di Integrità in linguaggio OCDL

La *funzione interpretazione* $\mathcal{I}$ è una funzione da $\mathbf{S}$ a $2^{\mathcal{V}(\mathcal{O})}$ tale che: $\mathcal{I}[\top] = \mathcal{V}(\mathcal{O})$, $\mathcal{I}[B] = \mathcal{I}_{\mathbf{B}}[B]$[2] , $\mathcal{I}[C] \subseteq \mathcal{O}$, $\mathcal{I}[V] \subseteq \mathcal{O}$, $\mathcal{I}[t] \subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O}$.

[2]Assumendo $\mathcal{I}_{\mathbf{B}}$ funzione di *interpretazione standard* da $\mathbf{B}$ a $2^{\mathbf{B}}$ tale che per ogni $d \in \mathbf{D} : \mathcal{I}_{\mathbf{B}}[d] = \{d\}$.

L'interpretazione è estesa agli altri tipi come segue:

$$\begin{aligned}
\mathcal{I}[[a_1\colon S_1,\ldots,a_p\colon S_p]] &= \Big\{[a_1\colon v_1,\ldots,a_q\colon v_q]\Big| p\le q, v_i\in\mathcal{I}[S_i], 1\le i\le p,\\
&\qquad v_j\in\mathcal{V}(\mathcal{O}), p{+}1\le j\le q\Big\}\\
\mathcal{I}[\forall\{S\}] &= \Big\{\{v_1,\ldots,v_p\}\Big| v_i\in\mathcal{I}[S], 1\le i\le p\Big\}\\
\mathcal{I}[\exists\{S\}] &= \Big\{\{v_1,\ldots,v_p\}\Big| \exists i, 1\le i\le p, v_i\in\mathcal{I}[S]\Big\}\\
\mathcal{I}[\triangle S] &= \Big\{o\in\mathcal{O}\Big|\delta(o)\in\mathcal{I}[S]\Big\}\\
\mathcal{I}[S\sqcap S'] &= \mathcal{I}[S]\cap\mathcal{I}[S']
\end{aligned}$$

Per i tipi cammino abbiamo $\mathcal{I}[(p\colon S)] = \mathcal{I}[(e\colon (p'\colon S))]$ se $p = e.p'$ dove

$$\mathcal{I}[(\epsilon\colon S)]=\mathcal{I}[S],\ \mathcal{I}[(a\colon S)]=\mathcal{I}[[a\colon S]],\ \mathcal{I}[(\triangle\colon S)]=\mathcal{I}[\triangle S],$$

$$\mathcal{I}[(\forall\colon S)]=\mathcal{I}[\forall\{S\}],\ \mathcal{I}[(\exists\colon S)]=\mathcal{I}[\exists\{S\}]$$

Quindi il tipo path è un'abbreviazione per un altro tipo: ad esempio, il tipo path ($\triangle$.`stock`.$\forall$.`item`: `SMaterial`) è equivalente a $\triangle$[`stock`: $\forall$\{[`item`: `SMaterial`]\}].

Si introduce ora la nozione di istanza legale di uno schema con regole come una interpretazione nella quale un valore istanziato in un nome di tipo ha una descrizione corrispondente a quella del nome di tipo stesso e dove sono valide le relazioni di inclusioni stabilite tramite le regole.

Definizione 1 (Istanza Legale) *Una funzione di interpretazione $\mathcal{I}$ è una* istanza legale *di uno schema con regole $(\sigma, \mathbf{R})$ sse l'insieme $\mathcal{O}$ è* finito *e per ogni $C \in \mathbf{C}, V \in \mathbf{V}, t \in \mathbf{T}, R \in \mathbf{R}$: $\mathcal{I}[C] \subseteq \mathcal{I}[\sigma(C)]$, $\mathcal{I}[t] = \mathcal{I}[\sigma(t)]$, $\mathcal{I}[V] = \mathcal{I}[\sigma(V)]$, $\mathcal{I}[S^a] \subseteq \mathcal{I}[S^c]$.*

Si noti come, in una istanza legale $\mathcal{I}$, l'interpretazione di un nome di classe base *è contenuta* nell'interpretazione della sua descrizione, mentre per un nome di classe virtuale, come per un nome di tipo-valore, l'interpretazione *coincide* con l'interpretazione della sua descrizione. In altri termini, mentre l'interpretazione di una classe base è fornita dall'utente, l'interpretazione di una classe virtuale è calcolata sulla base della sua descrizione. Pertanto, in particolare, le classi virtuali possono essere utilizzate per rappresentare interrogazioni effettuate sulla base di dati. In presenza di classi virtuali cicliche questa computazione non è univoca: fissata un'interpretazione per le classi

base, una classe virtuale ciclica può avere più di una interpretazione possibile. Tra tutte queste interpretazioni, noi selezioniamo come istanza legale quella *più grande*, cioè adottiamo una semantica di *massimo punto fisso*. Le definizioni formali di tale semantica e le motivazioni della scelta sono discusse in [BN94]; in questa sede si vuole soltanto evidenziare il fatto che il modello ammette delle classi virtuali cicliche e quindi il metodo di ottimizzazione proposto è applicabile anche alle query cicliche o ricorsive [BB96] .

4.3.2 Sussunzione ed Espansione Semantica di un tipo

Introduciamo la relazione di sussunzione in uno schema con regole.

Definizione 2 (Sussunzione) *Dato uno schema con regole* $(\sigma, \mathbf{R})$*, la* relazione di sussunzione *rispetto a* $(\sigma, \mathbf{R})$*, scritta* $S \sqsubseteq_{\mathrm{R}} S'$ *per ogni coppia di tipi* $S, S' \in \mathbf{S}$*, è data da:* $S \sqsubseteq_{\mathrm{R}} S'$ *sse* $\mathcal{I}[S] \subseteq \mathcal{I}[S']$ *per tutte le istanze legali* $\mathcal{I}$ *di* $(\sigma, \mathbf{R})$.

Segue immediatamente che $\sqsubseteq_{\mathrm{R}}$ è un preordine (i.e., transitivo e riflessivo ma antisimmetrico) che induce una *relazione di equivalenza* $\simeq_{\mathrm{R}}$ sui tipi: $S \simeq_{\mathrm{R}} S'$ sse $S \sqsubseteq_{\mathrm{R}} S'$ e $S' \sqsubseteq_{\mathrm{R}} S$. Diciamo, inoltre, che un tipo S è *inconsistente* sse $S \simeq_{\mathrm{R}} \bot$, cioè per ciascun dominio l'interpretazione del tipo è sempre vuota.

È importante notare che la relazione di sussunzione rispetto al solo schema σ, cioè considerando $\mathbf{R} = \emptyset$, denotata con $\sqsubseteq_{\sigma}$, è simile alle relazioni di *subtyping* o *refinement* tra tipi definite nei *CODMs* [Atz93, LR89]. Questa relazione può essere calcolata attraverso una comparazione sintattica sui tipi; per il nostro modello tale l'algoritmo è stato presentato in [BN94].

È immediato verificare che, per ogni S, S', se $S \sqsubseteq_{\sigma} S'$ allora $S \sqsubseteq_{\mathrm{R}} S'$. Comunque, il viceversa, in generale, non vale, in quanto le regole stabiliscono delle relazioni di inclusione tra le interpretazioni dei tipi che fanno sorgere *nuove* relazioni di sussunzione. Intuitivamente, come viene mostrato nell'esempio di figura 4.1, se $S \sqsubseteq_{\sigma} S^a$ e $S^c \sqsubseteq_{\sigma} S'$ allora $S \sqsubseteq_{\mathrm{R}} S'$.

La relazione esistente tra $\sqsubseteq_{\mathrm{R}}$ e $\sqsubseteq_{\sigma}$ può essere espressa formalmente attraverso la nozione di *espansione semantica*.

Definizione 3 (Espansione Semantica) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *e un tipo* $S \in \mathbf{S}$*, l'*espansione semantica *di* S *rispetto a* $\mathbf{R}$*, 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* $EXP(S) \sqsubseteq_{\sigma} S'$.

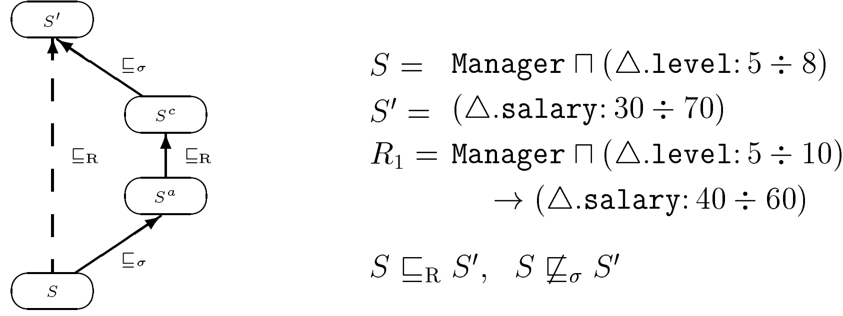

Figura 4.1: Relazione di sussunzione dovuta alle regole

In altri termini, EXP(S) è il tipo più specializzato (rispetto alla relazione $\sqsubseteq_\sigma$) tra tutti i tipi $\simeq_R$–equivalenti al tipo S.

L'espressione $EXP(S)$ permette di esprimere la relazione esistente tra $\sqsubseteq_R$ e $\sqsubseteq_\sigma$: per ogni $S, S' \in \mathbf{S}$ si ha $S \sqsubseteq_R S'$ se e solo se $EXP(S) \sqsubseteq_\sigma S'$. Questo significa che, dopo aver determinato l'espansione semantica, anche la relazione di sussunzione nello schema con regole può essere calcolata tramite l'algoritmo presentato in [BN94].

È facile verificare che, per ogni $S \in \mathbf{S}$ e per ogni $R \in \mathbf{R}$, se $S \sqsubseteq_\sigma (p : S^a)$ allora $S \sqcap (p\colon S^c) \simeq_R S$. Questa trasformazione di S in $S \sqcap (p\colon S^c)$ è la base del calcolo della $EXP(S)$: essa viene effettuata iterativamente, tenendo conto che l'applicazione di una regola può portare all'applicazione di altre regole. Per individuare tutte le possibili trasformazioni di un tipo implicate da uno schema con regole $(\sigma, \mathbf{R})$, si definisce la funzione totale $\Gamma : \mathbf{S} \longrightarrow \mathbf{S}$, come segue:

$$\Gamma(S) = \begin{cases} S \sqcap (p : S^c) & \text{se esistono } R \text{ e } p \text{ tali che } S \sqsubseteq_\sigma (p\colon S^a) \text{ e } S \not\sqsubseteq_\sigma (p\colon S^c) \\ S & \text{altrimenti} \end{cases}$$

e poniamo $\tilde{\Gamma} = \Gamma^i$, dove i è il più piccolo intero tale che $\Gamma^i = \Gamma^{i+1}$. L'esistenza di i è garantita dal fatto che il numero di regole è finito e una regola non può essere applicata più di una volta con lo stesso cammino ($S \not\sqsubseteq_\sigma (p\colon S^c)$). Si può dimostrare che, per ogni $S \in \mathbf{S}$, EXP(S) è effettivamente calcolabile tramite $\tilde{\Gamma}(S)$.

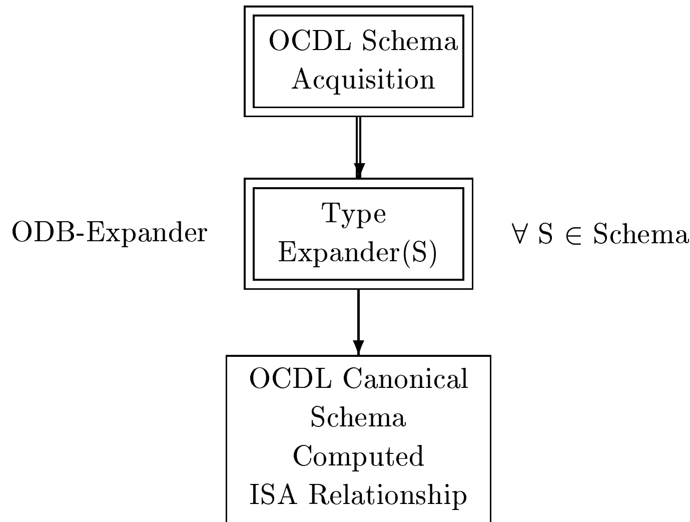

Figura 4.2: Architettura funzionale di OCDL-Designer

4.4 ODB-Tools: gli strumenti software

ODB-Tools è un sistema software per la validazione degli schemi di database e l'ottimizzazione delle interrogazioni che utilizza l'espansione semantica. ODB-Tools si compone di due prototipi software: OCDL-Designer ed ODB-QOptimizer sviluppati entrambi presso il Dipartimento di Scienze dell'Ingegneria (DSI) dell'Università di Modena in linguaggio C, gcc gnu compiler, su piattaforma hardware SUN SPARCstation 20, sistema operativo Solaris 2.x.

- OCDL-Designer, la cui architettura è mostrata in figura 4.2 realizza la fase di precompilazione dello schema descritta in sezione 4.1: lo schema delle classi e dei vincoli di integrità viene descritto dal progettista in forma OCDL. A questo punto può partire la fase di espansione realizzata da Type Expander che restituisce lo schema compilato contenente tutte le relazioni *isa* che sono implicitamente definite dai vincoli.

- A run time possiamo eseguire ODB-QOptimizer (si veda fig 4.3 per l'architettura) che realizza l'ottimizzazione di una interrogazione. La query viene portata in input al sistema nel linguaggio OQL insieme

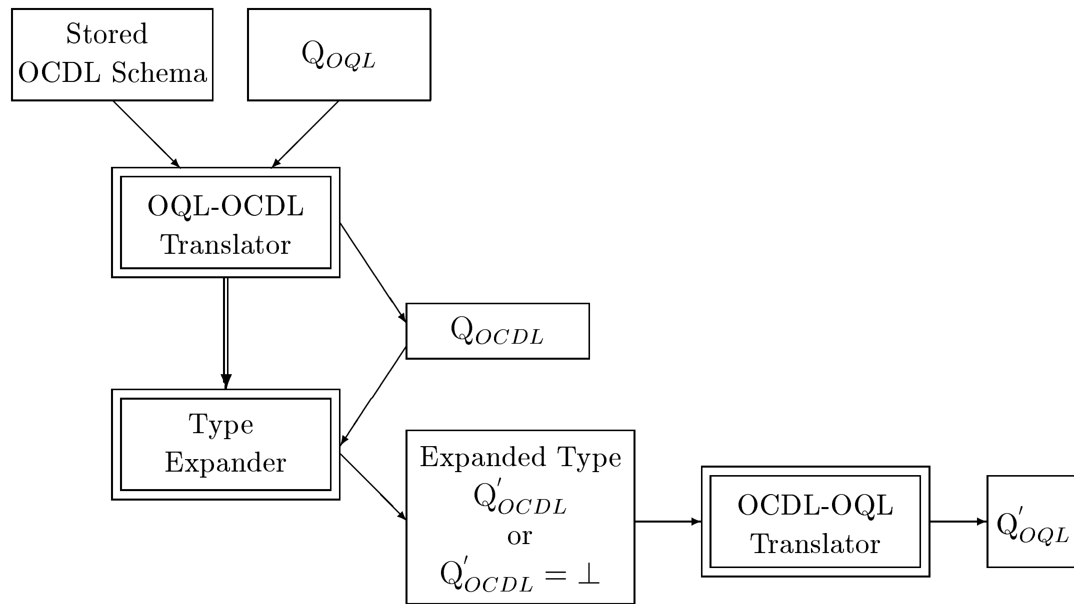

Figura 4.3: Architettura funzionale di ODB-QOptimizer

al nuovo schema ottenuto grazie ad OCDL-Designer. Un interprete converte l'interrogazione nella corrispondente rappresentazione in forma OCDL per poter invocare l'esecuzione dell'espansione semantica della stessa tramite Type Expander, il quale fornisce come risultato finale l'interrogazione ottimizzata. Un altro modulo di conversione restituisce in output la versione OQL della query.

Nel presente lavoro di tesi sono stati realizzati i moduli di interfacciamento al linguaggio OQL per ODB-QOptimizer. Il primo di questi moduli, rappresentato in figura 4.3 dal blocco “OQL–OCDL Translator”, legge in input una query in linguaggio OQL, individua le parti rappresentabili nel formalismo OCDL e le traduce in un'interrogazione per Type Expander. Il secondo modulo, “OCDL–OQL Translator”, riproduce in output l'interrogazione iniziale, in formato OQL, apportando i cambiamenti dovuti al processo di ottimizzazione. Si noti che nella figura i moduli di interfacciamento sono rappresentati come parte integrante di ODB-QOptimizer poichè attualmente fanno parte dello stesso programma eseguibile. In realtà

l'ottimizzatore può funzionare anche privo di questi moduli, interagendo con il mondo esterno esclusivamente attraverso il formalismo OCDL.

Le caratteristiche dell'interfaccia saranno illustrate in maniera dettagliata nei capitoli seguenti.

Capitolo 5

OQL e OCDL

In questo capitolo vengono presentate le espressioni tradotte dal linguaggio OQL a OCDL e viceversa. Nella prima parte vengono evidenziati i problemi incontrati nella traduzione delle espressioni OQL, le soluzioni individuate, le estensioni ed infine l'elenco completo delle espressioni tradotte ed il formalismo OCDL equivalente. Nella seconda parte viene descritto il metodo di traduzione delle espressioni OCDL in costrutti OQL.

5.1 La traduzione OQL-OCDL

Da un esame dei linguaggi OQL e OCDL si nota che non tutte le espressioni del primo possono essere tradotte nel secondo. I motivi per cui non è possibile una traduzione integrale si possono individuare principalmente nella limitata espressività del modello OCDL e nel differente orientamento dei due linguaggi.

Il primo aspetto deriva dal fatto che OCDL riesce a descrivere, in modo completo, soltanto gli aspetti strutturali di un modello di dati ad oggetti. Sono invece assenti le primitive necessarie ad esprimere gli aspetti dinamici del modello, in particolare i metodi. Questo implica che ogni interrogazione in cui compare l'invocazione di operazioni non possa essere tradotta, poichè i metodi corrispondenti non possono essere definiti nello schema OCDL della base di dati. Mancano inoltre alcuni operatori relazionali ("$\neq$") e booleani ("not" e "or") fondamentali. Per quanto riguarda l'operatore "or", di uso molto frequente, va detto che attualmente l'algoritmo di ottimizzazione semantica (presentato in [JBBV95, Vin94]) si applica esclusivamente a

interrogazioni di tipo congiuntivo, da cui l'impossibilità di esprimere questo operatore in OCDL.

La seconda importante differenza riguarda l'orientamento dei due linguaggi. OQL è un linguaggio "retrieval oriented", pensato cioè per essere utilizzato prevalentemente per estrarre oggetti da una base di dati. Per questo motivo OQL è fornito di una serie di operatori pensati specificatamente per *manipolare* collezioni e oggetti complessi, come gli operatori di conversione, le funzioni di aggregazione, i costruttori. OCDL è invece un linguaggio *descrittivo* che ha funzionalità e obiettivi diversi. In OCDL un'interrogazione, per poter essere sottoposta al controllo di coerenza e all'ottimizzazione semantica, viene rappresentata come una classe *virtuale* dello schema. Non ha quindi senso prevedere quell'insieme di operatori, presenti in OQL, che trovano un utilizzo soltanto di tipo "run-time" o che non sono significativi ai fini dell'ottimizzazione.

5.1.1 Esempi di interrogazioni OQL traducibili in OCDL

Nella precedente sezione è stato evidenziato come la traduzione dal linguaggio OQL a OCDL possa essere soltanto parziale. Si presentano allora due possibilità: consentire all'utente di esprimere le interrogazioni utilizzando soltanto i costrutti OQL traducibili, oppure accettare una qualunque interrogazione esprimibile in questo linguaggio e riconoscere le parti che possono essere tradotte. Sulla base del discorso introduttivo sull'ottimizzazione semantica effettuato in sezione 4.2, viene scelta la seconda soluzione, ovvero si effettua una separazione dell'interrogazione in una parte *clean* (esprimibile in OCDL e sulla quale è quindi effettuabile l'ottimizzazione semantica) e una parte *dirty* (non esprimibile in OCDL). In questo modo, si ottimizzano il maggior numero possibile di interrogazioni. Si pone dunque il problema di come riconoscere quelle parti della query che possono essere tradotte.

OQL è un linguaggio funzionale i cui costrutti sono visti come interrogazioni e possono essere liberamente composti. A causa di questa natura completamente ortogonale un operatore OQL che, preso singolarmente, può essere tradotto in OCDL, se viene composto con altri costrutti può dar luogo ad un'espressione non traducibile. Di conseguenza le regole generali seguite dal traduttore sono le seguenti:

- Esiste un insieme di costrutti OQL che, sotto certe condizioni, possono essere tradotti in OCDL.

- Se vengono composti due costrutti traducibili il risultato è di nuovo un'espressione traducibile.
- Se un costrutto traducibile viene composto con un costrutto non traducibile il risultato è un costrutto non traducibile.

Un'eccezione alle regole precedenti è costituita dall'operatore booleano `and`. Infatti nei casi in cui questo operatore non viene tradotto ma viene composto con un altro operatore `and` l'espressione ottenuta può risultare comunque traducibile in OCDL.

Per illustrare il metodo seguito utilizziamo inizialmente alcuni esempi. Gli esempi sono riferiti sempre al dominio Magazzino, il cui schema OCDL è riportato in Tabella 4.1. Inoltre, nel presente capitolo, con Q verrà indicata una classe virtuale dello schema (cioè $Q \in \mathbf{V}$): tale classe virtuale è appunto il risultato della trasformazione di una interrogazione originariamente formulata in linguaggio OQL.

Innanzitutto osserviamo che un'interrogazione OQL viene tradotta soltanto se è possibile individuare un nome di classe cui fare riferimento nella traduzione. In generale le interrogazioni tradotte sono allora di due tipi. Il primo, banale, è costituito da un semplice nome di classe:

```
Material
```

il secondo è costituito invece da un operatore `select-from-where` in cui la parte `from` contiene almeno un nome di classe, come nell'interrogazione seguente: Q_1 = "Seleziona i materiali con rischio superiore a 10"

```
select   *
from     Material M
where    M.risk > 10
```

L'individuazione di un nome di classe cui fare riferimento nella traduzione è indispensabile in quanto la query OQL viene tradotta in una descrizione di classe virtuale OCDL. La precedente interrogazione OQL viene infatti tradotta in:

$$\begin{aligned}\sigma(Q_1) \;=\; & \texttt{Material} \sqcap \\ & \triangle[\texttt{risk}\colon 10 \div \infty]\end{aligned}$$

È importante notare che una volta individuata la classe di riferimento, la traduzione in OCDL prescinde dall'eventuale proiezione sugli attributi di tale

classe. Ad esempio, se modifichiamo la precedente query OQL aggiungendo una proiezione sull'attributo name:

```
select   M.name
from     Material M
where    M.risk > 10
```

la sua traduzione in OCDL è sempre la stessa. Il motivo di questo comportamento risiede nell'obiettivo dell'ottimizzazione semantica, riassumibile sinteticamente nell'individuazione dei fattori che possono essere modificati, aggiunti o eliminati da una interrogazione. Il sistema in esame effettua tale compito considerando, attraverso una descrizione di classe OCDL, le condizioni espresse sugli oggetti dell'interrogazione, indipendentemente da quelli che sono gli attributi specificati nella select-list della query OQL. Si ricorda che in OCDL l'estensione di una classe è un insieme di oggetti (più precisamente di identificatori di oggetti); in OQL per ottenere un insieme di oggetti (con *oid*) da un filtro `select-from-where` deve essere specificata come proiezione la variabile collegata alla classe:

```
select   M
from     Material M
where    M.risk > 10
```

Di seguito introduciamo i vari costrutti tradotti attraverso alcuni esempi.

Operatore *and*

Per illustrare la traduzione dell'operatore `and` consideriamo la seguente interrogazione:

Q_2 = "Seleziona i materiali di nome "steel" con rischio inferiore a 10"

```
select   *
from     Material M
where    M.risk < 10 and
         M.name = "steel"
```

È facile constatare che l'interrogazione è interamente traducibile in linguaggio OCDL nel seguente modo:

$$\begin{aligned}\sigma(Q_2) &= \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}\colon \infty \div 10] \sqcap \\ &\quad \triangle[\texttt{name}\colon \texttt{"steel"}]\end{aligned}$$

Allo stesso modo è facile constatare che l'aggiunta alla precedente query di un fattore contenente l'operatore di disuguaglianza rende l'interrogazione non traducibile. È il caso della query seguente, che indicheremo con Q_3:

```
select   *
from     Material M
where    M.risk < 10 and
         M.name  = "steel" and
         M.code != "C 60"
```

In presenza di un fattore non traducibile noi perdiamo la possibilità di tradurre la query OQL in una classe virtuale, in quanto non è possibile rappresentare in OCDL tutte le condizioni necessarie e sufficienti di appartenenza di un oggetto alla query. D'altro canto la parte della query OQL traducibile in OCDL rappresenta delle condizioni necessarie *ma* non sufficienti di appartenenza di un oggetto alla query, pertanto possiamo rappresentare in OCDL tale parte con una classe *primitiva* ($Q_P \in \mathbf{C}$):

$$\begin{aligned}\sigma(Q_P) &= \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}\colon \infty \div 10] \sqcap \\ &\quad \triangle[\texttt{name}\colon \texttt{"steel"}]\end{aligned}$$

Per continuare a trattare in maniera uniforme le query del tutto traducibili e quelle non traducibili la precedente classe primitiva viene espressa in maniera equivalente attraverso la seguente classe virtuale

$$\begin{aligned}\sigma(Q_3) &= \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}\colon \infty \div 10] \sqcap \\ &\quad \triangle[\texttt{name}\colon \texttt{"steel"}] \sqcap \\ &\quad \overline{Q}\end{aligned}$$

dove $\overline{Q}$ è una particolare classe primitiva, detta *atomo fittizo*, senza alcuna descrizione (cioè $\overline{Q} \in \mathbf{C}$ e $\sigma(\overline{Q}) = \top$). In questo modo, le interrogazioni

rappresentabili soltanto parzialmente in linguaggio OCDL vengono espresse ancora come classi virtuali, ma nella loro descrizione viene introdotto un atomo fittizio che rappresenta la parte della query non traducibile.

Si noti che l'introduzione dell'atomo fittizio nella descrizione di una query non interamente traducibile è indispensabile ai fini della corretta classificazione della query. Infatti, siano Q e Q' due classi, derivate dalla traduzione di altrettante interrogazioni, con la seguente descrizione:

$$\begin{aligned}\sigma(Q) &= \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}: 10 \div \infty] \sqcap \overline{Q} \\ \sigma(Q') &= \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}: 30 \div \infty]\end{aligned}$$

Se la classe Q non avesse nella sua descrizione l'atomo fittizio sarebbe verificata la relazione di sussunzione $Q' \sqsubseteq Q$. Di fatto però l'interrogazione OQL corrispondente alla classe Q contiene altri fattori, non tradotti, per i quali la relazione di sussunzione trovata potrebbe non essere verificata.

Naturalmente è possibile che tutta la condizione espressa nella clausola `where` risulti non traducibile, come avviene nella seguente interrogazione:

```
select   M.name
from     Manager  M
where    M.level != 6
```

Tale interrogazione si traduce nella classe $\sigma(Q_4) = \texttt{Manager} \sqcap \overline{Q}$.

Operatore *for all*

L'operatore for all, il quantificatore universale di OQL, restituisce un valore vero (*true*) soltanto se *tutti* gli elementi di una collezione soddisfano la condizione specificata. Per illustrare la traduzione di quest'operatore consideriamo la seguente interrogazione :

Q_5 = "Seleziona i magazzini in cui tutti i materiali contenuti hanno un rischio maggiore di 10 e sono presenti in più di 100 unità"

```
select   *
from     Storage  S
where    for all X in S.stock: ( X.qty > 100 and
                                 X.item.risk > 10 )
```

la cui rappresentazione in OCDL assume la forma:

$$\begin{aligned}\sigma(Q_5) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{qty}: 100 \div \infty] \}] \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{item}: \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

Questo esempio si presta per fare alcune considerazioni sulla forma dell'espressione OCDL generata dal traduttore. Prima di tutto si noti che è possibile trovare per la classe virtuale Q altre descrizioni sintatticamente differenti dalla precedente ma ad essa equivalenti come, ad esempio, la seguente:

$$\begin{aligned}\sigma(Q_5) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{qty}: 100 \div \infty] \sqcap [\texttt{item}: \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

L'ottimizzatore semantico comprende un modulo per la riduzione di una generica espressione di tipo OCDL in un forma canonica interna, presentata in [BN94], in cui vengono esplicitati tutti gli attributi delle classi e, in particolare, vengono risolte le congiunzioni. Di conseguenza, nella traduzione da OQL a OCDL, si è scelto di non generare direttamente un'espressione in forma canonica poichè ciò avrebbe richiesto, dal punto di vista dell'implementazione, un dispendio di risorse (ad esempio strutture dati, tempo di elaborazione) per ottenere una funzionalità già presente nel modulo ottimizzatore.

Nella traduzione da OQL a OCDL è stato scelto come obiettivo quello di generare un tipo espresso come congiunzione di altri tipi che non contengono a loro volta l'operatore di congiunzione e utilizzano solo tipi ennupla con un unico attributo; come vedremo tra breve questo non è sempre possibile. La formalizzazione di questo discorso verrà presentata nella sezione 5.1.2.

Un'ulteriore considerazione che può essere fatta riguarda la traduzione delle espressioni di percorso di OQL, quale, ad esempio, `X.item.risk` nella precedente interrogazione. È evidente che in OCDL il tipo candidato per rappresentare espressioni di percorso di OQL è il tipo cammino; ad esempio, si potrebbe esprimere la precedente interrogazione utilizzando esclusivamente dei tipi cammino, nel modo seguente:

$$\begin{aligned}\sigma(Q_5) &= \texttt{Storage} \sqcap \\ &(\triangle.\texttt{stock}.\forall.\texttt{qty}: 10 \div \infty) \sqcap (\triangle.\texttt{stock}.\forall.\texttt{item}. \triangle .\texttt{risk}: 10 \div \infty)\end{aligned}$$

Tuttavia, è stato scelto di tradurre le espressione di percorso di OQL non tramite tipi cammino OCDL ma utilizzando gli altri costrutti del formalismo. Anche in questo caso si tratta di una scelta dovuta a ragioni di efficienza. Tradurre l'espressione utilizzando il tipo cammino OCDL sarebbe infatti risultata un'operazione inutile: il modulo ottimizzatore, per le sue caratteristiche di funzionamento, avrebbe in seguito esplicitato nuovamente l'espressione.

Consideriamo ora una parte non traducibile all'interno dell'operatore `for all`:

Q_6 = "seleziona i magazzini in cui tutti i materiali contenuti hanno un rischio inferiore a quello massimo ammissibile per il magazzino e sono presenti in più di 50 unità".

```
select   *
from     Storage S
where    for all X in S.stock: (X.item.risk < S.maxrisk and
                                X.qty > 50)
```

È facile verificare che in OCDL non si può esprimere un confronto tra attributi, quindi, seguendo il discorso fatto in precedenza l'espressione tradotta è la seguente:

$$\begin{aligned}\sigma(Q_6) \quad = \quad & \texttt{Storage} \sqcap \\ & \triangle[\texttt{stock}: \{ [\texttt{qty}: 50 \div \infty] \}] \sqcap \\ & \overline{Q}\end{aligned}$$

L'esempio seguente mostra la traduzione nel caso di due predicati non traducibili:

```
select   *
from     Storage S
where    for all X in S.stock: (X.item.risk < S.maxrisk and
                                X.qty > 50) and
         S.category != "A2"
```

a questa query facciamo corrispondere l'espressione OCDL

$$\begin{aligned}\sigma(Q_7) \quad = \quad & \texttt{Storage} \sqcap \\ & \triangle[\texttt{stock}: \{ [\texttt{qty}: 50 \div \infty] \}] \sqcap \\ & \overline{Q}\end{aligned}$$

Si osserva immediatamente che viene generato un solo atomo fittizio. Per i nostri scopi è infatti sufficiente rappresentare i due predicati non traducibili con un unico atomo fittizio e quindi considerare l'espressione $\sigma(Q_7)$ data in precedenza.

Operatore *exists*

OQL mette a disposizione due quantificatori esistenziali. Di seguito viene presentata la forma "nativa" dell'operatore. L'espressione alternativa, che sarà trattata più accuratamente nel seguito, è invece una variazione sintattica della prima, ed è stata prevista soltanto per ragioni di compatibilità verso SQL. L'operatore `exists` restituisce un valore vero (*true*) se, nella collezione specificata, esiste almeno un elemento che soddisfa una data condizione. Consideriamo, ad esempio, la stessa forma di interrogazione fatta per l'operatore `for all`:

Q_8 = "Seleziona i magazzini in cui almeno uno dei materiali contenuti ha un rischio maggiore di 10 ed è presente in più di 100 unità"

```
select   *
from     Storage  S
where    exists X in S.stock: ( X.qty > 100 and
                                X.item.risk > 10 )
```

La traduzione in questo caso sarà del tipo:

$$\begin{aligned}\sigma(Q_8) \;&=\; \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}\colon \exists\{\,[\texttt{qty}\colon 100 \div \infty\,] \sqcap [\texttt{item}\colon \triangle[\texttt{risk}\colon 10 \div \infty]\,]\,\}\,]\end{aligned}$$

ovvero, in una forma che non rispetta le condizioni delineate in precedenza; infatti la seguente traduzione

$$\begin{aligned}\sigma(Q_8^{'}) \;&=\; \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}\colon \exists\{\,[\texttt{qty}\colon 100 \div \infty]\,\}\,] \sqcap \\ &\triangle[\texttt{stock}\colon \exists\{\,[\texttt{item}\colon \triangle[\texttt{risk}\colon 10 \div \infty]\,]\,\}\,]\end{aligned}$$

non è corretta; in particolare, la classe virtuale $Q_8^{'}$ fornisce un sovrainsieme degli oggetti della interrogazione originale.

Come ulteriore esempio di traduzione relativa all'operatore `exists`, consideriamo la seguente interrogazione contenente una parte non traducibile:

```
select   *
from     Storage  S
where    exists X in S.stock: ( X.qty > 100 and
                                X.item.risk < S.maxrisk )
```

Traduzione

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\quad \triangle[\texttt{stock}: \exists\{\,[\texttt{qty}: 100 \div \infty]\,\}\,] \sqcap \\ &\quad \overline{Q}\end{aligned}$$

Operatore *in*

L'operatore `in` esprime la condizione di appartenenza di un elemento ad un insieme[1]. L'espressione restituisce un valore vero (*true*) se la condizione è verificata. Perchè la traduzione sia possibile la collezione specificata deve essere costituita da un nome di classe oppure da un'espressione *select* che ha come proiezione la variabile definita nella clausola *from*. Infatti solo in questi casi l'insieme dato individua un insieme di oggetti ed è quindi rappresentabile come una descrizione di classe OCDL. Questo significa che la seguente query OQL

$Q_9 =$ "seleziona i magazzini il cui responsabile è un top-manager di livello superiore a 9"

```
select   *
from     Storage  S
where    S.managed_by in ( select   M
                           from     TManager M
                           where    M.level > 9 )
```

è interamente traducibile in OCDL

$$\begin{aligned}\sigma(Q_9) &= \texttt{Storage} \sqcap \\ &\quad \triangle[\texttt{managed_by}: \texttt{TManager}] \sqcap \\ &\quad \triangle[\texttt{managed_by}: \triangle[\texttt{level}: 9 \div \infty]\,]\end{aligned}$$

[1] In questo contesto per insieme si intende un qualunque tipo di collezione

Ovviamente l'operatore `in` esprime anche condizioni di appartenenza relative ad insiemi di valori, come avviene nella seguente query: "seleziona i magazzini il cui responsable ha il nome uguale a quello di un top-manager di livello superiore a 9"

```
select   *
from     Storage  S
where    S.managed_by.name in ( select   M.name
                                from     TManager M
                                where    M.level > 9 )
```

Si noti innanzitutto che la condizione espressa fa sì che questa interrogazione, rispetto alla precedente, restituisca un insieme di valori più ampio. La collezione ottenuta infatti contiene certamente i magazzini il cui responsabile è un top-manager di livello superiore a 9 ma contiene anche (ammesso che ne esistano) i magazzini diretti da una persona che ha lo stesso nome di almeno uno dei top-manager citati. Si noti inoltre che questo tipo di condizione non è esprimibile in OCDL; di conseguenza l'interrogazione precedente viene tradotta come: $\sigma(Q_{10}) = \texttt{Storage} \sqcap \overline{Q}$.

Operatore *intersect*

L'operatore `intersect` calcola l'insieme di elementi comuni tra due collezioni date. La traduzione di questo operatore avviene quando si può rappresentare come intersezione tra classi nell'espressione OCDL corrispondente. Consideriamo, ad esempio, la seguente interrogazione: Q_{11} = "seleziona i magazzini il cui responsabile è sia un manager di livello superiore a 5 che un top-manager"

```
select  *
from    Storage  S
where   S.managed_by in (
             TManager intersect ( select  M
                                  from    Manager M
                                  where   M.level > 5 ))
```

a cui corrisponde l'espressione OCDL

$$
\begin{aligned}
\sigma(Q_{11}) \;=\; & \texttt{Storage} \sqcap \\
& \triangle[\texttt{managed_by}\colon \texttt{TManager}] \sqcap
\end{aligned}
$$

$$\triangle[\texttt{managed_by}\colon \texttt{Manager}] \sqcap$$
$$\triangle[\texttt{managed_by}\colon \triangle[\texttt{level}\colon 5 \div \infty]\,]$$

Anche in questo caso i fattori generati dal traduttore non contengono l'operatore di congiunzione. Sarà in seguito il modulo ottimizzatore a calcolare l'intersezione tra le classi, generando un'espressione del tipo

$$\texttt{Storage} \sqcap \triangle[\texttt{managed_by}\colon \texttt{TManager} \sqcap \texttt{Manager} \sqcap \triangle[\texttt{level}\colon 5 \div \infty]\,]$$

L'operazione di *join*

La seguente interrogazione OQL:

```
select   *
from     Storage  S,
where    S.category  = "A2"   and
         S.managed_by.level > 5
```

effettua una selezione sulla classe `Storage`, ma la condizione `S.managed_by.level > 5` è espressa su oggetti di un'altra classe, la classe `Manager`, legata alla classe `Storage` attraverso la gerarchia di composizione: in questi casi si parla di *join implicito* tra le due classi. La traduzione in OCDL è la seguente:

$$\begin{aligned}\sigma(Q_{12}) \;=\;& \texttt{Storage} \sqcap \\ & \triangle[\texttt{category}\colon "A2"] \sqcap \\ & \triangle[\texttt{managed_by}\colon \triangle[\texttt{level}\colon 5 \div \infty]\,]\end{aligned}$$

L'esempio precedente, come tutti gli altri esempi illustrati finora, presenta nella parte `from` un'unica classe. D'altro canto in OQL è possibile specificare nella clausola `from` più classi (più espressioni), allo scopo di effettuare l'operazione di *join esplicito* tra queste classi, ovvero un join la cui condizione è espressa su generici attributi delle classi in questione, come avviene nel seguente esempio: Q_{13} = "seleziona le coppie di magazzini di categoria "A2" e materiali di tipo "steel" in cui il massimo rischio ammesso dal magazzino coincide con il rischio del materiale"

```
select   *
from     Storage  S,
         Material M
where    S.maxrisk   = M.risk and
         S.category  = "A2"   and
         M.name      = "steel"
```

È facile verificare che questa interrogazione non si può esprimere in OCDL. In questo caso si potrebbe cercare di ottimizzare comunque la query considerando separatamente le due condizioni locali poste rispettivamente su `Storage` e su `Material`, ovvero considerando le seguenti classi virtuali:

$$\begin{aligned}\sigma(Q_{13}) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{category}: "A2"] \sqcap \\ &\overline{Q}\end{aligned}$$

e

$$\begin{aligned}\sigma(Q'_{13}) &= \texttt{Material} \sqcap \\ &\triangle[\texttt{name}: "steel"] \sqcap \\ &\overline{Q}\end{aligned}$$

ed eseguendo per esse due processi di ottimizzazione semantica separati, per poi riportarne il risultato nella interrogazione originale. Dato che attualmente il sistema prevede, per una data query, un solo processo di ottimizzazione, è stato scelto, in alternativa al totale rifiuto della precedente query OQL da parte dell'ottimizzatore, il trattamento di una sola delle due sottointerrogazioni. In particolare, si considera la sottointerrogazione riferita alla classe che nella clausola `from` viene specificata per prima; ad esempio, nel precedente caso si considera solo $\sigma(Q_{13})$.

5.1.2 Individuazione dei costrutti traducibili

In questa sezione viene riportato l'elenco degli operatori OQL integralmente o parzialmente tradotti, specificando per ognuno la sintassi dell'espressione OCDL generata.

Espressione *select*

Espressione generale OQL

select [**distinct**]	query
from	query [[**as**]] Iterator_name
	{, query [[**as**] *Iterator_name*]}
[**where**	query]
[**group by**	partition_attributes]
[**having**	query]
[**order by**	sort_criterion {, sort_criterion}]

Le clausole `select`, `group by` e `order by` non possono essere rappresentate in OCDL. Qualunque sia l'espressione specficata in queste parti l'interrogazione viene comunque tradotta. Le clausole dell'espressione che vengono prese in considerazione ai fini della traduzione sono le seguenti:

select	Oql_Expression
from	Class_name [[**as**] Iterator_name] [, Oql_Expression]
[**where**	Condition]

Il termine `Oql_Expression` indica un'espressione OQL che può essere specificata dal punto di vista grammaticale ma che non viene tradotta. L'espressione `select` viene tradotta se sono verificate le seguenti condizioni:

- la clausola `from` deve specificare almeno un nome di classe.
- nel caso vengano specificate più classi o più espressioni soltanto la prima classe elencata viene presa in cosiderazione per la traduzione.
- i predicati espressi nella clausola `where` possono riferire la classe di apparteneza in maniera esplicita o implicita. Nel primo caso il predicato riferisce la classe attraverso una variabile. Nel secondo caso il predicato contiene un pathname che inizia con un attributo della classe. Il traduttore è quindi in grado di individuare la giusta correlazione tra pathname e classi.

La forma generale di un'espressione *select* traducibile può essere rappresentata nel modo seguente:

select	$\pi(c, e_1, e_2 \ldots, e_n)$
from	C as c,

$E_1(c)$ as e_1,
$E_2(c, e_1)$ as e_2,
...
$E_n(c, e_1, e_2, \ldots, e_{n-1})$ as e_n
where $F_1(c)$ and $F_2(c)$ and ... and $F_n(c)$ and
$\overline{F}_1(c, e_1, e_2, \ldots, e_n)$ and $\overline{F}_2(c, e_1, e_2, \ldots, e_n)$ and ... and
$\overline{F}_n(c, e_1, e_2, \ldots, e_n)$

dove
π identifica una generica proiezione OQL
C rappresenta un **nome di classe**
$E_1, E_2, \ldots, E_n$ sono generiche espressioni ammesse nella clausola from
$F_1, F_2, \ldots, F_n$ sono fattori che è possibile tradurre
$\overline{F}_1, \overline{F}_2, \ldots, \overline{F}_n$ sono fattori non traducibili

Si noti che tutti i fattori F_i traducibili sono funzione della variabile definita sulla classe C, oggetto dell'interrogazione. Indicando con F_i' le espressioni OCDL corrispondenti ai fattori F_i , la query OCDL avrà la forma seguente

$$\sigma(Q) \quad = \quad \begin{cases} C \sqcap F_1' \sqcap F_2' \ldots \sqcap F_n' & \text{oppure} \\ C \sqcap F_1' \sqcap F_2' \ldots \sqcap F_n' \sqcap \overline{Q} & \text{dove } \overline{Q} \text{ è un atomo fittizio} \end{cases}$$

L'espressione F_i', $0 \leq i \leq n$, è chiaramente un tipo OCDL definito in 4.3 ma ulteriormente vincolato a rispettare le condizioni definite dalla seguente sintassi:

$$S \quad \rightarrow \quad \top \mid B \mid C \mid [a : S] \mid \ \triangle\, S \mid \forall\{S\} \mid \exists\{S_1 \sqcap S_2 \ldots \sqcap S_m\}$$

In particolare, si noti che l'operatore di congiunzione $\sqcap$ è ammesso solo all'interno dell'operatore $\exists\{\}$ e le ennuple hanno un solo attributo.

Operatore *and*

Espressione generale OQL

query **and** query

L'operatore `and` viene espresso in OCDL attraverso la congiunzione ($\sqcap$) di fattori semplici. Indicando con F_1 ed F_2 gli operandi di *and*, con F_1' ed F_2' le rispettive traduzioni e con $\overline{Q}$ un atomo fittizio le regole di traduzione dell'operatore `and` sono le seguenti:

1. Se F_1 e F_2 sono espressioni traducibili viene generata la traduzione OCDL
$$F_1' \sqcap F_2'$$

2. Se soltanto F_1 (F_2) è traducibile l'espressione generata è
$$F_1' \sqcap \overline{Q}$$

3. Se nessuno dei due operandi è traducibile viene generato solamente un atomo fittizio
$$\overline{Q}$$

Operatori relazionali

Espressione generale OQL

query θ query $\qquad\qquad$ $\theta \in \{>, >=, <, <=, =\}$

Le espressioni che contengono operatori relazionali sono tradotte soltanto se rappresentano un confronto tra un attributo (*attribute*) e un tipo base (*value*)

attribute θ value $\qquad\qquad$ $\theta \in \{>, >=, <, <=, =\}$

Se $\theta \in \{>, >=\}$ viene generata l'espressione OCDL

[attribute : value $\div\infty$]

Se $\theta \in \{<, <=\}$ viene generata l'espressione OCDL

[attribute : $\infty\div$ value]

Se $\theta \in \{=\}$ viene generata l'espressione OCDL

[attribute : value]

Per le stringhe di caratteri viene tradotto soltanto l'operatore di uguaglianza.

Naturalmente è possibile specificare un cammino al posto di un semplice attributo. In questo caso nelle espressioni sopra riportate al posto del nome dell'attributo comparirà la traduzione OCDL del cammino. Questa considerazione è valida anche per gli altri operatori trattati in questa sezione.

Espressione *for all*

Espressione generale OQL

for all Iterator_name **in** query : query

Indicando con *attribute* un attributo a valori multipli della classe oggetto di interrogazione l'espressione che può essere tradotta è del tipo

for all Iterator_name **in** attribute : query

Per evitare di generare interrogazioni incoerenti la traduzione di questo costrutto è diversa a seconda della definizione data nello schema per l'attributo multivalore. Se *attribute* è definito come $\{\}$ oppure $\exists\{\}$ viene generata la traduzione :

[attribute : $\{\ldots\}$]

se invece *attribute* è definito nello schema come *sequenza* la traduzione diventa:

[attribute : $\langle\ldots\rangle$]

Espressione *exists*

Espressione generale OQL

exists Iterator_name **in** query : query

L'espressione che si può tradurre ha la seguente forma

exists Iterator_name **in** attribute : query

Se l'attributo multivalore è definito nello schema come $\{\}$ oppure $\exists\{\}$ viene generata la traduzione :

[attribute : $\exists\{\ldots\}$]

se invece *attribute* è definito nello schema come sequenza la traduzione non è attualmente possibile non essendo disponbile l'operatore $\exists\langle\ldots\rangle$.

Espressione *in*

Espressione generale OQL

query **in** query

L' operatore *in* viene tradotto nei casi seguenti:

attribute **in** Class_name

attribute **in** (select X from Class_name as X where ...)

Nel primo caso la traduzione OCDL generata è:

[attribute : Class_name]

mentre nel secondo caso la traduzione è:

[attribute : Class_name $\sqcap F_1 \sqcap F_2 \sqcap \ldots \sqcap F_n$]

Espressione *intersect*

Espressione generale OQL

query **intersect** query

L'operatore *intersect* viene tradotto quando si presenta nelle forme

(select X from Class_name X where ...)
intersect
(select Y from Class_name Y where ...)

oppure

Class_name intersect (select X from Class_name X where ...)

Gli operandi ammessi sono quindi

- un nome di classe.
- un'espressione *select* traducibile la cui proiezione è costituita dalla variabile definita nella clausola *from*.
- un'altra espressione *intersect* traducibile.

La traduzione generata è data dalla congiunzione delle classi interrogate e dei predicati tradotti nella clausola *where* (se specificata):

$$C_1 \sqcap F_1 \sqcap F_2 \sqcap \ldots \sqcap F_n \sqcap C_2 \sqcap F'_1 \sqcap F'_2 \sqcap \ldots \sqcap F'_n$$

C_1 indica la prima classe ed $F_1, F_2, \ldots, F_n$ indicano i rispettivi predicati tradotti
C_2 indica la seconda classe ed $F'_1, F'_2, \ldots, F'_n$ indicano i rispettivi predicati tradotti

5.1.3 Espressioni non tradotte

Dal discorso effettuato nelle sezioni precedenti, si deduce che è possibile tradurre interrogazioni congiuntive i cui fattori contengono gli operatori relazionali (ad eccezione dell'operatore di disuguaglianza), alcune forme ristrette di quantificazione universale ed esistenziale e l'operazione di join implicito. D'altra parte la limitazione della potenza espressiva del linguaggio di interrogazione sul quale viene effettuata l'ottimizzazione semantica in OCDL ha lo scopo di rendere efficiente il processo di ottimizzazione.

I principali operatori che non vengono trattati sono gli operatori booleani **or** e **not**:

query ::= query **or** query
query ::= query **not** query

In OCDL non è possibile esprimere l'operazione di *or* logico tra due espressioni poichè l'algoritmo di ottimizzazione semantica si applica esclusivamente a query di tipo congiuntivo. Per ovviare a tale mancanza si potrebbe pensare di suddividere un'interrogazione che contiene predicati in *or* in più query contenenti soltanto predicati in *and*. Le interrogazioni così ottenute potrebbero essere ottimizzate in fasi distinte e il risultato complessivo sarebbe rappresentato dall'unione delle singole query. L'operatore *not* non è disponibile in OCDL e di conseguenza non viene tradotto.

Un discorso a parte deve invece essere fatto per quei costrutti OQL che possono essere sostituiti, in maniera del tutto equivalente, da altri operatori dello stesso linguaggio. In questi casi è stato scelto di tradurre soltanto una delle forme equivalenti, e precisamente quella che meglio si adatta ad essere

traslata in OCDL. Il caso più evidente di questa situazione è rappresentato dall'operatore **exists**

query ::= **exists**(query)

Questa espressione è una variazione sintattica dell'operatore *exists* di OQL ed è prevista unicamente per ragioni di compatibilità verso SQL. Consideriamo, ad esempio, l'interrogazione seguente: Q_14 = "seleziona i magazzini in cui almeno un materiale è stoccato in più di 50 unità"

```
select   *
from     Storage S
where    exists( select   *
                 from     S.stock X
                 where    X.qty > 50 )
```

questa forma del quantificatore esistenziale non viene tradotta in OCDL; di conseguenza la query sarebbe rappresentata come $\sigma(Q_{14}) = \texttt{Storage} \sqcap \overline{Q}$. Se però l'interrogazione viene posta nella forma equivalente che segue

```
select   *
from     Storage S
where    exists X in S.stock: ( X.qty > 50 )
```

il quantificatore viene tradotto. L'espressione OCDL che si ottiene è la seguente:

$$\begin{aligned} \sigma(Q'_{14}) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}\colon \{\, [\texttt{qty}\colon 10 \div \infty] \,\}\,] \end{aligned}$$

5.1.4 Problemi aperti

In questa sezione vengono presentati due casi particolari di espressioni OQL che attualmente non vengono tradotte in OCDL. La particolarità di questi costrutti risiede nel fatto che essi si prestano ad essere trasformati in espressioni equivalenti che a loro volta possono essere trattate dal software realizzato. Questa trasformazione sintattica dovrebbe quindi avvenire *prima* che l'interrogazione venga esaminata dal traduttore. Di seguito vengono presentate le espressioni individuate durante il lavoro di tesi.

Operatore *for all*

Abbiamo già osservato che la natura puramente funzionale di OQL consente di comporre in qualunque modo i costrutti del linguaggio purchè venga rispettato il sistema dei tipi. Questa caratteristica, se da un lato consente di non limitare in alcun modo la capacità espressiva del linguaggio, dall'altro pone dei problemi nella sua traduzione nel formalismo OCDL.

Uno di questi problemi si manifesta a proposito di espressioni che si trovano all'interno di quantificatori (esistenziali e universali) pur non dipendendo dalla variabile associata al quantificatore. Per illustrare meglio il problema utilizziamo alcuni esempi che fanno sempre riferimento allo schema di tabella 4.1. Supponiamo inoltre che esista la nuova regola (in sintassi ODMG-93):

rule R5 **for all** `X` **in** `Storage` (`X.category` = "B4")
then `X.maxrisk` = 20

Consideriamo ora la seguente interrogazione, che indicheremo con Q_{15}, che fa uso dell'operatore "for all":

```
select *
from Storage S
where forall X in S.stock : ( S.category = "B4" and
                              X.qty > 10 )
```

In pratica questa interrogazione seleziona i magazzini in cui, per tutti gli elementi dello stock, è verificata una delle due seguenti condizioni:

1. ogni elemento è presente in più di 10 unità e la categoria del magazzino è "B4".

2. lo stock è vuoto. In questo caso l'operatore `for all` restituisce un valore booleano vero (*true*).

Osserviamo ora che la traduzione in linguaggio OCDL effettuata dal software realizzato è di questo tipo:

$$\sigma(Q_{15}) = \texttt{Storage} \sqcap \triangle[\texttt{stock}: \{ [\texttt{qty}: 10 \div \infty] \}] \sqcap \overline{Q}$$

Si vede immediatamente che il predicato `S.category` non è stato tradotto e che al suo posto è stato generato un atomo fittizio. Tale predicato infatti pur trovandosi all'interno del quantificatore `for all` non dipende dalla variabile `X`. Dipende invece dalla variabile `S` definita sulla classe `Storage`. Si noti inoltre che tale predicato non può essere tradotto in OCDL nel modo seguente

$$\begin{aligned}\sigma(Q_{15}) \; = \; & \texttt{Storage} \sqcap \\ & \triangle[\texttt{stock}: \{ [\texttt{qty}: 10 \div \infty] \}] \sqcap \\ & \triangle[\texttt{stock}: \{ [\texttt{category}: \; "B4"] \}]\end{aligned}$$

poichè si perderebbe il riferimento alla classe `Storage`. Per evitare questo problema si potrebbe però pensare di generare una traduzione di questo tipo

$$\begin{aligned}\sigma(Q_{15}) \; = \; & \texttt{Storage} \sqcap \\ & \triangle[\texttt{stock}: \{ [\texttt{qty}: 10 \div \infty] \}] \sqcap \\ & \triangle[\texttt{category}: \; "B4"] \}]\end{aligned}$$

In questo modo si avrebbe l'applicazione della regola R5, e di conseguenza si otterrebbe un'ottimizzazione dell'interrogazione.

In realtà vedremo nel seguito che ci sono delle complicazioni. A questo punto possiamo fare alcune considerazioni:

1. se per un magazzino la condizione `category = "B4"` non è verificata allora il `for all` resituirà un valore booleano falso (a meno che lo stock non sia vuoto) e il magazzino non verra' selezionato.

2. se invece la condizione `category = "B4"` è verificata allora il `for all` restituirà un valore vero se lo stock e' vuoto oppure se e solo se la condizione `qty > 10` è verificata per tutti gli elementi dello stock.

3. se lo stock è vuoto il magazzino farà in ogni caso parte dell'insieme selezionato dalla query.

Trascurando momentaneamente la possibilità di uno stock vuoto si osserva che i magazzini restituiti dalla query sono tutti di categoria "B4", indipendentemente dall'altro predicato presente nella condizione del "forall", e quindi, in base alla regola R5, hanno l'attributo `maxrisk` uguale a 20. Sotto queste ipotesi l'applicazione della regola non sembra porre problemi e allo stesso risultato si potrebbe pervenire formulando la query Q_{16}:

```
select *
from Storage S
where for all X in S.stock : ( X.qty > 10 ) and
      category = "B4"
```

Tale query si comporta, a meno di stock vuoto, secondo quanto esposto ai punti 1 e 2.

Le cose si complicano ammettendo l'eventualita' di stock vuoto. In questo caso la query equivalente a quella data (Q_{15}) deve essere espressa nella forma seguente, che indicheremo con Q_{16}:

```
select *
from Storage S
where (  for all X in S.stock : ( X.qty > 10 ) and  category = "B4")
      or for all X in S.stock : ( false )
```

Quello che cambia, in sostanza, e' la posizione nella query del predicato implicato dalla regola. La query OQL che si dovrebbe ottenere dopo l'applicazione della regola R5 dovrebbe essere:

```
select *
from Storage S
where (for all X in S.stock : ( X.qty > 10 ) and
        category = "B4" and
        maxrisk  = 20 )
      or for all X in S.stock : ( false )
```

Questa interrogazione seleziona i magazzini che

- possiedono uno stock in cui tutti gli elementi sono in quantita' maggiore di 10.
- sono di categoria "B4".
- hanno un rischio massimo uguale a 20.

oppure seleziona i magazzini che hanno lo stock vuoto.

La query finale dà quindi lo stesso risultato della query iniziale (Q_{14}). In questo modo il riposizionamento del fattore `category = "B4"` potrebbe essere effettuato *prima* del passo di traduzione della query in OCDL.

Questo genere di problemi non si presenta nel caso del quantificatore esistenziale poiche' l'estrazione di un predicato genera una espressione del tutto equivalente a quella data.

Espressioni di percorso nella *from clause*

Nel capitolo 3 abbiamo visto che è possibile utilizzare le espressioni di percorso nella clausola `from` del filtro `select-from-where`per navigare attraverso le associazioni tra classi o per raggiungere le proprietà di oggetti complessi. Consideriamo ad esempio l'interrogazione seguente:

```
select struct( STORAGE : S, RISK : X.item.risk )
from   Storage  S,
       S.stock  X
where  X.qty > 50
```

In questa query ci proproniamo di selezionare i magazzini che contengono soltanto materiali in numero superiore a 50 unità. Inoltre vogliamo ottenere in output l'oid di ogni magazzino assieme al rischio di ogni materiale stoccato. In precedenza abbiamo visto che il software realizzato traduce l'interrogazione in OCDL in questo modo

$$\sigma(Q) = \texttt{Storage} \sqcap \overline{Q}$$

poiché il predicato `X.qty > 50` non dipende dal primo nome specificato nella clausola `from`. Supponiamo ora di istanziare alcuni oggetti delle classi `Storage` e `Material` utilizzando la sintassi prevista da OQL:
Classe `Material`:

```
m1  ←  Material( risk : 5 )
m2  ←  Material( risk : 6 )
m3  ←  Material( risk : 7 )
```

Classe `Storage`:

```
s1  ←  Storage( stock: set( struct( qty : 20 , item : m1 ),
                            struct( qty : 40 , item : m1 ),
                            struct( qty : 60 , item : m2 ),
                            struct( qty : 70 , item : m3 )))

s2  ←  Storage( stock: set( struct( qty : 15 , item : m1),
                            struct( qty : 30 , item : m2)))

s3  ←  Storage( stock: set( struct( qty : 55 , item : m2 )))
```

dove m_1, m_2, m_3 sono gli identificatori delle istanze del tipo `Material`. $s_1, s_2, \ldots s_n$ sono gli identificatori delle istanze del tipo `Storage`.

Con queste istanze l'interrogazione precedente restituisce come risultato le seguenti coppie di valori:

STORAGE: s_1 RISK: 5
STORAGE: s_1 RISK: 6
STORAGE: s_3 RISK: 7

Si può constatare che soltanto due istanze del tipo `Storage`, e precisamente s_1 ed s_3, vengono selezionate. Si osservi inoltre la molteplicità dell'istanza s_1 dovuta al prodotto cartesiano effettuato nella `from` clause [2].

Consideriamo ora l'interrogazione seguente, che fa uso del quantificatore esistenziale:

```
select S, X.item.risk
from   Storage  S,
       S.stock  X
where  exists Y in S.stock: ( Y.qty > 50 )
```

Si noti innanzitutto che questa query può essere parzialmente tradotta in OCDL dal software realizzato nel modo seguente:

$$\sigma(Q) = \texttt{Storage} \sqcap \triangle[\texttt{stock}: \{[\texttt{qty}: 50 \div \infty]\}]$$

Osserviamo inoltre che in generale questa query restituisce un risultato diverso dalla precedente:

STORAGE: s_1 RISK: 5
STORAGE: s_1 RISK: 5
STORAGE: s_1 RISK: 6
STORAGE: s_1 RISK: 7
STORAGE: s_3 RISK: 6

La differenza consiste nella molteplicità dell'istanza s_1 nel risultato, ed è dovuta alla diversa condizione espressa dall'operatore `exists`. Si noti però che le istanze selezionate sono sempre le stesse, cioè s_1 ed s_3. A questo punto possiamo fare una considerazione: l'obiettivo dell'ottimizzazione semantica è quello di ottenere l'interrogazione più specializzata tra quelle semanticamente

[2] Una definizione rigorosa della semantica dell'operatore `select` è data in [Cat94].

equivalenti a quella data, riducendo così il numero di istanze da controllare per determinare il risultato. Nelle due query presentate abbiamo osservato che le istanze del tipo `Storage` recuperate sono sempre le stesse, anche se con diversa molteplicità. Dal punto di vista del formalismo OCDL, anche per le considerazioni sulla `select-list` effettuate in sezione 5.1.1, le due interrogazioni potrebbero essere pensate come equivalenti e, di conseguenza, si potrebbe cercare di rendere traducibile la prima query trasformandola, a priori, nella seconda forma. Questa potrebbe essere fornita in seguito al traduttore OQL→OCDL.

5.2 La traduzione da OCDL a OQL

Un'altra funzionalità fondamentale dell'interfaccia realizzata nel presente lavoro di tesi è la traduzione dell'interrogazione ottimizzata dal formalismo OCDL al linguaggio OQL. Questo passaggio è in generale sempre possibile.

Per illustrare il metodo seguito ricorriamo ad alcuni esempi. In questa sezione indicheremo con Q l'interrogazione OCDL fornita in input all'ottimizzatore e con Q' la query restituita in output, cioè ottimizzata.

Innanzitutto osserviamo che l'espressione OCDL non viene tradotta in una query OQL vera e propria. In realtà vengono soltanto sostituite quelle parti dell'interrogazione originaria interessate dal processo di ottimizzazione. Consideriamo, ad esempio, l'interrogazione seguente:

```
select   M
from     Material M
```

Secondo le regole esposte nella sezione precedente la query viene tradotta in $\sigma(Q)$ = `Material`. È facile verificare che, poichè nessuna regola può essere applicata, l'espressione ottimizzata sarà del tutto simile a quella iniziale e precisamente $\sigma(Q')$ = `Material`. Di conseguenza l'interrogazione OQL risultante sarà in questa forma:

```
select   M
from     Material M
```

La parte evidenziata rappresenta l'espressione tradotta dal formalismo OCDL al linguaggio OQL. Essa *sostituisce* nel testo dell'interrogazione il nome di classe specificato inizialmente dall'utente. La parte non evidenziata è

invece rimasta inalterata rispetto alla query originaria. Naturalmente si possono verificare casi in cui, oltre a sostituire le parti di interrogazione esaminate dall'ottimizzatore, vengono aggiunti nuovi predicati (costrutti) introdotti dall'applicazione delle regole. Ad esempio, nella seguente interrogazione

```
select   S
from     Storage S
where    S.managed\_by.level > 6  and
         S.managed\_by.level < 12
```

la cui traduzione in OCDL si può individuare in

$$\begin{aligned}\sigma(Q) \;&=\; \texttt{Storage} \sqcap \triangle[\texttt{managed_by}: \triangle[\texttt{level}: 6 \div \infty]]\\ &\;\triangle[\texttt{managed_by}: \triangle[\texttt{level}: \infty \div 12]]\end{aligned}$$

le condizioni imposte sull'attributo `level` causano l'applicazione della regola **R3**. Di conseguenza l'interrogazione ottimizzata diventa:

$$\sigma(Q^{'}) \;=\; \texttt{Storage} \sqcap \triangle[\texttt{category}: "A2", \texttt{managed_by}: \texttt{Manager} \sqcap \triangle[\texttt{level}: 6 \div \infty]]$$

L'indicazione della classe `Manager`verrà introdotta in uno dei paragrafi successivi; per adesso ci interessa sottolineare che il fattore `category`: "$A2$", introdotto dal processo di ottimizzazione, viene riportato nella query OQL finale in questo modo:

```
select   S
from     Storage S
where    S.managed_by.level > 6 and
         S.managed_by.level < 12 and
         S.category = "A2"
```

Consideriamo ora l'interrogazione Q_1 presentata nella sezione precedente. Al termine dell'espansione semantica, ottenuta applicando la regola **R1**, si ottiene l'espressione seguente:

$$\sigma(Q_1^{'}) = \texttt{SMaterial} \;\sqcap\; \triangle\,[\texttt{risk}: 10 \div \infty]$$

In particolare si nota che ora l'interrogazione è riferita alla classe `SMaterial`. La query OQL corrispondente assume allora la forma:

```
select   M
from     SMaterial M
where    M.risk > 10
```

È immediato verificare che il nome di classe `Material`è stato sostituito con `SMaterial`, mentre il predicato `M.risk > 10` è stato sostituito con un'espressione identica. Le parole chiave del linguaggio e la `select-list` sono invece rimaste inalterate. Questo modo di operare consente di raggiungere due obiettivi:

- l'interrogazione ottimizzata che viene presentata all'utente è del tutto simile, nelle parti non tradotte, alla query OQL iniziale. In questo modo i cambiamenti apportati dall'ottimizzatore diventano immediatamente individuabili.
- in output è possibile riprodurre anche le espressioni OQL che, nella prima fase di traduzione, non era stato possibile rappresentare nel formalismo OCDL. È il caso, ad esempio, della `select-list`.

Un'altra osservazione, di carattere generale, che possiamo fare è che il nome della classe cui fa riferimento l'interrogazione OCDL sostituisce quella specificato inizialmente nella clausola `from`. La seconda osservazione è che la descrizione OCDL della classe ($[\texttt{risk}\colon 10 \div \infty]$) viene invece riprodotta nella clausola `where`.

Procedendo per esempi illustriamo di seguito la traduzione dei vari costrutti OCDL. Per non appesantire inutilmente la trattazione riutilizzeremo alcune delle interrogazioni presentate nella sezione precedente.

Tipo Ennupla

Per illustrare la traduzione del tipo ennupla riprendiamo l'interrogazione Q_2:

$$\begin{aligned}\sigma(Q_2) \;=\; & \texttt{Material} \sqcap \\ & \triangle[\texttt{risk}\colon \infty \div 10] \sqcap \triangle[\texttt{name}\colon \texttt{"steel"}]\end{aligned}$$

È facile vedere che nella fase di espansione semantica non può essere applicata alcuna regola, di conseguenza l'espressione restituita dall'ottimizzatore è la seguente

$$\sigma(Q_2') = \texttt{Material} \ \sqcap \ \triangle[\texttt{risk}\colon \infty \div 10, \texttt{name}\colon \texttt{"steel"}]$$

Ciò che possiamo notare è che la sintassi della descrizione è differente da quella generata dal traduttore OQL–OCDL. L'ottimizzatore infatti, pur non applicando regole, ha risolto le congiunzioni ottenendo un'unica ennupla con gli attributi `risk` e `name`. La traduzione in OQL, analogamente alle precedenti interrogazioni, assume la forma:

```
select   *
from     Material M
where    M.risk < 10
         M.name = "steel"
```

L'interrogazione Q_3 si presta ad evidenziare la trattazione dell'atomo fittizio. Ricordiamo che la forma OQL iniziale di tale interrogazione era traducibile soltanto parzialmente nel formalismo OCDL, e di conseguenza l'espressione generata dal traduttore era la seguente:

$$\begin{aligned}\sigma(Q) \ &= \ \texttt{Material} \sqcap \\ &\quad \triangle[\texttt{risk}\colon \infty \div 10] \sqcap \triangle[\texttt{name}\colon \texttt{"steel"}] \sqcap \overline{Q}\end{aligned}$$

Ovviamente anche in questo caso non si può applicare nessuna regola per cui l'espressione ottenuta al termine dell'espansione semantica è la seguente

$$\sigma(Q_3') = \texttt{Material} \ \sqcap \ \triangle[\texttt{risk}\colon \infty \div 10, \texttt{name}\colon \texttt{"steel"}] \sqcap \overline{Q}$$

da cui si ottiene l'espressione OQL

```
select   *
from     Material M
where    M.risk < 10 and
         M.name = "steel" and
         M.code = "C 60"
```

È facile vedere che le parti di interrogazione sostituite sono le stesse dell'interrogazione Q_2. L'atomo fittizio quindi non viene preso in considerazione in questa fase della traduzione: la parte dell'interrogazione OQL che non era stato possibile tradurre viene semplicemente riprodotta, tale e quale, nella query finale.

Quantificatore universale

Per illustrare la traduzione del quantificatore universale utilizziamo l'interrogazione seguente

```
select   *
from     Storage  S
where    for all X in S.stock: ( X.qty > 100 and
                                 X.item.risk < 10 )
```

la cui traduzione in linguaggio OCDL assume la forma

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}\colon\{\,[\texttt{qty}\colon 100 \div \infty]\,\}\,] \sqcap \\ &\triangle[\texttt{stock}\colon\{\,[\texttt{item}\colon \triangle[\texttt{risk}\colon \infty \div 10]\,]\,\}\,]\end{aligned}$$

Tale interrogazione, dopo il processo di espansione semantica, durante il quale non viene applicata nessuna regola, assume la forma

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}\colon\{\,[\texttt{qty}\colon 100 \div \infty\ ,\texttt{item}\colon \texttt{Material} \sqcap \triangle[\texttt{risk}\colon \infty \div 10]\,]\,\}\,]\end{aligned}$$

Possiamo osservare che, rispetto all'interrogazione iniziale, l'ottimizzatore ha risolto le congiunzioni dando luogo ad una descrizione che contiene un unico quantificatore universale con all'interno un'unica ennupla con gli attributi `risk` e `qty`. Si noti inoltre che in questa espressione si specifica (viene cioè esplicitato) che l'attributo `item` si riferisce ad oggetti della classe `Material`. Poiché quest'informazione è specificata esplicitamente nello schema della base di dati non viene riportata nella traduzione in linguaggio OQL della query. Un apposito modulo software, descritto nel capitolo 6 si occupa di identificare questa eventualità.

Il quantificatore OCDL viene tradotto utilizzando l'operatore `for all`, nella cui condizione sono introdotti i predicati su `risk` e `qty`:

```
select   *
from     Storage S
where    for all X in S.stock:  X.qty > 100 and
                                X.item.risk < 10 )
```

Anche in questo caso le parti tradotte da OCDL a OQL e sostituite nel testo dell'interrogazione originale sono evidenziate.

Al fine di illustrare la trattazione dell'atomo fittizio nel caso dell'operatore `for all` riprendiamo l'interrogazione Q_6. È immediato verificare che a questa interrogazione l'ottimizzatore non applicherà nessuna regola, restituendo quindi l'espressione

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{qty}: 50 \div \infty] \}] \sqcap \overline{Q}\end{aligned}$$

che verrà infine convertita nella query OQL seguente

```
select   *
from     Storage S
where    for all X in S.stock:  X.item.risk < S.maxrisk and
                                X.qty > 50 )
```

Si nota che il predicato `X.item.risk < S.maxrisk`, che nella traduzione della query iniziale aveva dato origine ad un atomo fittizio, viene semplicemente riprodotto in output senza alcuna modifica.

Quantificatore esistenziale

Per illustrare la traduzione del quantificatore esistenziale utilizziamo l'interrogazione Q_8 presentata nella sezione precedente. L'espressione OCDL generata dal traduttore era

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \exists\{ [\texttt{qty}: 100 \div \infty] \sqcap [\texttt{item}: \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

Al termine della fase di ottimizzazione, in cui non viene applicata nessuna regola, si ottiene l'espressione:

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \exists\{ [\texttt{qty}: 100 \div \infty \text{ , } \texttt{item}: \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

Anche in questo caso l'ottimizzatore ha semplicemente risolto le congiunzioni, producendo all'interno del quantificatore esistenziale un'unica ennupla che presenta gli attributi `risk` e `qty`. Questa espressione viene convertita nella query OQL seguente, in cui il quantificatore esistenziale di OCDL viene tradotto con l'operatore `exists` di OQL:

```
select   *
from     Storage S
where    exists X in S.stock:  X.qty > 100 and
                               X.item.risk > 10 )
```

Come negli esempi precedenti le parti evidenziate rappresentano ciò che è stato tradotto da linguaggio OCDL a OQL, mentre le parti non evidenziate sono semplicemente riprodotte in output.

Indicazione di classe

Per illustrare la traduzione di questo costrutto consideriamo la query Q_9 presentata nella sezione precedente. L'espressione restituita dall'ottimizzatore in risposta a questa interrogazione assume la forma:

$$\sigma(Q) = \texttt{Storage} \sqcap \triangle[\texttt{managed_by}: \texttt{TManager} \sqcap \texttt{Manager} \sqcap \triangle[\texttt{level}: 9 \div \infty]]$$

Si nota immediatamente che il processo di espansione semantica ha generato la congiunzione delle classi `Manager`e `TManager`, poichè nello schema della base di dati l'attributo `managed_by` è riferito alla classe `Manager`. Dato però che il nostro obiettivo è l'ottimizzazione della query viene riportata in uscita soltanto la classe `TManager`, poichè si tratta della classe più specializzata. L'espressione `managed_by`: `TManager` viene tradotta utilizzando l'operatore "`in`" di OQL nel seguente modo

```
select   *
from     Storage S
where    S.managed_by in ( selectM
                           from  TManager M
                           where M.level > 9 )
```

Si noti che anche nella sottointerrogazione vengono sostituite soltanto alcune parti, e precisamente quelle che derivano dalla traduzione del formalismo OCDL.

Consideriamo ora un esempio più complesso, costituito dalla query Q_5 della sezione precedente. La traduzione in OCDL di tale interrogazione era stata individuata in

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{qty}: 100 \div \infty] \}] \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{item}: \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

Durante il processo di espansione semantica vengono applicate le regole **R1** ed **R2** consentendo di ottenere la nuova interrogazione

$$\begin{aligned}\sigma(Q) &= \texttt{Storage} \sqcap \texttt{SStorage} \sqcap \\ &\triangle[\texttt{stock}: \{ [\texttt{qty}: 100 \div \infty \ , \texttt{item}: \texttt{Material} \sqcap \texttt{SMaterial} \sqcap \triangle[\texttt{risk}: 10 \div \infty]] \}]\end{aligned}$$

Per prima cosa possiamo osservare che, rispetto all'interrogazione iniziale, l'ottimizzatore ha risolto le congiunzioni dando luogo ad una descrizione che contiene un unico quantificatore universale ed un'unica ennupla con gli attributi `risk` e `qty`. In secondo luogo notiamo che le classi `Storage` e `Material` sono state specializzate, rispettivamente, nelle classi `SStorage` ed `SMaterial`. Di conseguenza viene generata in output la query OQL seguente

```
select   *
from     SStorage S
where    for all X in S.stock:(
                      X.item in (
                                select   Y
                                from     SMaterial
                                where    Y.risk > 10 ) and
                      X.qty > 100 )
```

Si nota che in questo caso viene introdotta un'intera sottoquery, in cui viene riportato, nella clausola `where`, il predicato `risk` > 10.

5.2.1 Traduzione dei costrutti OCDL

In questa sezione viene riportato un elenco riassuntivo dei vari costrutti OCDL tradotti unitamente alla corrispondente espressione OQL generata.

Operatore di congiunzione

Per l'operatore di congiunzione occorre distinguere tra i casi seguenti:

- quando viene applicato a due nomi di classe compatibili [3] la traduzione generata corrisponde al nome della classe più specializzata. Ad esempio l'espressione

  `Material`$\sqcap$ `SMaterial`

  viene tradotta nell'espressione OQL `SMaterial`. Nel formalismo OCDL è però possibile avere anche congiunzioni di classi che nel linguaggio OQL sarebbero considerate incompatibili. In questo caso la traduzione generata cerca comunque di rappresentare la situazione con l'operatore `intersect`. As esempio, l'espressione

  `Material`$\sqcap$ `Storage`

  viene rappresentata con il costrutto OQL `Materialintersect Storage`.

- quando tale operatore si trova tra un nome di classe e la sua descrizione non viene tradotto. Semplicemente si riporta il nome di classe nella clausola `from` e la descrizione, sotto la forma di predicati booleani, nella clausola `where`. Non riportiamo esempi di questo caso poiché basta osservare le interrogazioni già esposte in precedenza.

- nei casi rimanenti la congiunzione viene tradotta con l'operatore `and` di OQL. Ad esempio l'espressione

  [`risk`: $\infty \div 10$] $\sqcap$ [`name`: `"steel"`]

  viene tradotta in

  ```
  risk < 10 and name = "steel"
  ```

Ennupla

Espressione OCDL

[attribute$_1$: ... , attribute$_2$: ...]

La traduzione degli attributi di un'ennupla avviene utilizzando l'operatore `and`:

attribute$_1$ ... **and** attribute$_2$ ...

[3] la compatibilità tra tipi è intesa nel senso specificato da OQL

Quantificatore universale

Espressione OCDL
[attribute: $\{\ldots\}$]
Il quantificatore universale viene tradotto in OQL attraverso l'operatore `for all`:

for all Iterator_name **in** attribute: (...)

Sequenza

Espressione OCDL
[attribute: $\langle\ldots\rangle$]
Il tipo sequenza viene tradotto in OQL attraverso l'operatore `for all`:

for all Iterator_name **in** attribute: (...)

Quantificatore esistenziale

Espressione OCDL
[attribute: $\exists\{\ldots\}$]
Il quantificatore esistenziale viene tradotto in OQL attraverso l'operatore `exists`:

exists Iterator_name **in** attribute: (...)

Indicazione di classe

Espressione OCDL
[attribute: $Class_name$]
L'indicazione di una classe viene tradotta in OQL utilizzando l'operatore "in":

attribute **in** Class_name

Capitolo 6

Il software

Nel presente capitolo viene descritta l'architettura del software realizzato che consente di controllare la correttezza di una query OQL e di determinare le parti che possono essere tradotte in linguaggio OCDL. In particolare il software è in grado di:

- determinare la correttezza sintattica e semantica di una interrogazione OQL;
- rilevare le parti dell'interrogazione che possono essere tradotte nel formalismo OCDL;
- generare la corrispondente struttura di memoria centrale che rappresenta la query da fornire in input al modulo ottimizzatore;
- tradurre la struttura di memoria centrale che costituisce l'output del modulo ottimizzatore in un insieme di espressioni e predicati in linguaggio OQL;
- sostituire tali espressioni nella query originale, evidenziando le parti ottimizzate.

Il capitolo è organizzato come segue: inizialmente vengono presentati i concetti fondamentali su cui si basa il traduttore realizzato, di seguito vengono illustrati i principali aspetti progettuali ed infine viene mostrata l'architettura generale del sistema unitamente ad un esempio di codifica.

6.1 Concetti Base

La funzione principale svolta dall'interfaccia tra OQL e OCDL è quella di tradurre un file di testo contenente una query in linguaggio OQL in una struttura di memoria centrale che rappresenta l'interrogazione in formato OCDL. Di seguito vengono quindi illustrati quegli aspetti basilari del funzionamento di un compilatore (traduttore) che hanno trovato applicazione nel software realizzato.

Ogni traduttore deve svolgere due compiti fondamentali: l'*analisi* del programma sorgente (in questo caso la query) e la *sintesi* del codice (nel caso specifico una struttura di memoria). In quasi tutti i compilatori moderni questo processo è diretto dalla struttura sintattica del programma sorgente, opportunamente riconosciuta da un modulo detto *parser*. Si parla allora di *compilatori diretti dalla sintassi*. La traduzione può essere suddivisa nei seguenti passi

- *analisi lessicale*. Per riconoscere la sintassi del programma sorgente bisogna innanzitutto individuare le singole parole chiave del linguaggio ed i simboli utilizzati. Questo compito è svolto da un modulo, detto *analizzatore lessicale* o *scanner*, che produce in output una sequenza di *token*. Ogni token attribuisce un significato ad un gruppo di caratteri del testo del programma sorgente. Se una sequenza di caratteri non è riconosciuta viene generato un errore lessicale.

- *analisi sintattica*. Il parser riceve in input la sequenza di token e, analizzando il modo in cui questi sono combinati, riconosce i costrutti del linguaggio sorgente. Quando una particolare sequenza non corrisponde a nessuna delle regole grammaticali programmate nel parser viene rilevato un errore di sintassi.

- *generazione del codice*. Ogni volta che il parser riconosce un costrutto invoca un'apposita operazione il cui codice è definito dal programmatore del traduttore. Questa operazione è detta *azione semantica* e presenta un duplice aspetto: in primo luogo completa il compito di analisi semantica effettuando ulteriori controlli di correttezza (ad esempio regole di visibilità dei nomi); il secondo aspetto riguarda invece la generazione del codice. A questo livello vengono rilevati gli errori semantici.

- *ottimizzazione*. Il codice prodotto necessita in genere di essere ottimizzato. Questa fase può essere svolta separatamente dalle altre oppure può essere compresa direttamente nella fase di generazione del codice. Nel traduttore realizzato questa fase non è presente in quanto non necessaria.

Un traduttore può essere scritto in un qualunque linguaggio di programmazione che presenti sufficienti caratteristiche di capacità espressiva e flessibilità. Nella maggioranza dei casi però risulta essere un compito piuttosto complesso, indipendentemente dal linguaggio utilizzato. Per questo motivo sono stati studiati e realizzati appositi strumenti software, detti *generatori di parser*, che facilitano la scrittura dei compilatori. Nella prossima sezione vengono presentati due di questi strumenti, utilizzati nel lavoro di tesi.

6.2 Generatori di parser

Flex e Bison sono due strumenti software che facilitano la scrittura di programmi in linguaggio C per l'analisi e l'interpretazione di sequenze di caratteri che costituiscono un dato testo sorgente. Entrambi questi strumenti, partendo da opportuni file di specifica, generano direttamente il codice in linguaggio C, che può quindi essere trattato allo stesso modo degli altri moduli sorgenti di un programma.

Flex Flex legge un file di specifica che contiene delle espressioni regolari per il riconoscimento dei token (si veda la sezione precedente) e genera una funzione, chiamata **yylex()**, che effettua l'analisi lessicale del testo sorgente. La funzione generata estrae i caratteri in sequenza dal fusso di input. Ogni volta che un gruppo di caratteri soddisfa una delle espressioni regolari viene riconosciuto un token e, di conseguenza, viene invocata una determinata azione, definita opportunamente dal programmatore. Tipicamente l'azione non fa altro che rendere disponibile il token identificato al riconoscitore sintattico. Per spiegare meglio il meccanismo di funzionamento ricorriamo ad un esempio: l'individuazione, nel testo sorgente, di un numero intero

```
[0-9]+            {
                    sscanf( yytext, "%d", &yylval );
                    return( INTEGER );
                  }
```

l'espressione regolare `[0-9]+` rappresenta una sequenza di una o più cifre comprese nell'intervallo `0-9`. La parte compresa tra parentesi {...} specifica invece, in linguaggio C, l'azione che deve essere eseguita. In questo caso viene restituito al parser il token `INTEGER` poiché è stato riconosciuto un numero intero.

Bison BISON è un programma in grado di generare un parser in linguaggio C partendo da un file di specifica che definisce un insieme di regole grammaticali. In particolare BISON genera una funzione, chiamata **yyparse()**, che interpreta una sequenza di token e riconosce la sintassi definita nel file di input. La sequenza di token può essere generata da un qualunque analizzatore lessicale; di solito però BISON viene utilizzato congiuntamente a FLEX.

I parser generati da questo strumento sono in grado di riconoscere soltanto un certo sottoinsieme di grammatiche dette *non contestuali*. Questa però non è una limitazione reale poiché i linguaggi di programmazione presentano, in generale, una sintassi definita da una grammatica non contestuale.

Il vantaggio principale che deriva dall'utilizzo di BISON è la possibilità di ottenere un vero e proprio parser semplicemente definendo, in un apposito file, la sintassi da riconoscere. Ciò avviene utilizzando una notazione molto simile alla *Bakus-Naur Form* (BNF). Occorre però notare che i parser generati in questo modo sono in grado di riconoscere soltanto un certo sottoinsieme di grammatiche, dette *non contestuali*. A prima vista ciò potrebbe sembrare una limitazione; in realtà questo tipo di grammatica è in genere sufficiente[1] per definire la sintassi di un linguaggio di programmazione.

Per illustrare meglio il funzionamento di questo software utilizziamo un esempio di un possibile input per BISON:

```
var_declaration:    VAR var_list ':' type_name ';' ;
variable_list:      variable_name |
                    variable_list ',' variable_name ;
variable_name:      STRING ;
type_name:          INTEGER | FLOAT | BOOLEAN ;
```

Ogni regola consiste di un nome, o simbolo non terminale, seguito da una definizione, che presenta a sua volta uno o più simboli terminali o non terminali (ovvero nomi di altre regole). I simboli terminali, rappresentati nell'esempio

[1] una trattazione più dettagliata e formale è data in [ASU86, FRJL88]

in carattere maiuscolo, sono i token ottenuti dal riconoscitore lessicale. Il riconoscimento della grammatica avviene con un procedimento di tipo *bottom-up*[2], includendo ogni regola che viene riconosciuta in regole più generali, fino a raggiungere un particolare simbolo terminale che include tutti gli altri. A questo punto il testo sorgente è stato completamente riconosciuto e l'analisi sintattica è terminata.

In realtà un parser deve svolgere anche altri compiti, come l'analisi semantica e la generazione del codice. Per questo motivo BISON consente al programmatore di definire un segmento di codice, detto *azione*, per ogni regola grammaticale. Ogni volta che una regola viene riconosciuta il parser invoca l'azione corrispondente, permettendo, ad esempio, di inserire i nomi delle variabili nella symbol table durante l'analisi della sezione dichiarativa di un linguaggio:

```
var_declaration:      VAR var_list ':' type_name ';'
                      {
                          Push( $2 );
                      }
                      ;
```

Nell'esempio illustrato `Push()` è una funzione il linguaggio C che si occupa di inserire una lista di variabili nella `symbol table`. Il codice che si occupa della traduzione vera e propria può allora essere integrato nel parser attraverso il meccanismo delle azioni semantiche.

6.3 Progettazione

In questa sezione vengono illustrati gli argomenti che hanno richiesto una particolare attenzione dal punto di vista progettuale.

6.3.1 Informazioni semantiche

Un'importante caratteristica della progettazione del traduttore è la definizione delle strutture dati su cui operano le azioni semantiche. L'approccio seguito è quello di associare un *record semantico* ad ogni simbolo grammaticale riconosciuto dal parser. In questo modo si fa corrispondere ad ogni

[2]descritto ampiamente in [MB91]

elemento della grammatica una struttura che contiene le informazioni appropriate per quel simbolo. Ogni volta che il parser riconosce una regola viene invocata l'azione semantica corrispondente, i cui parametri sono gli elementi che costituiscono la parte destra della regola. I record semantici dei simboli terminali contengono valori atomici derivati dal testo del token associato e vengono in genere creati direttamente dall'analizzatore lessicale. I record dei simboli non terminali sono invece creati dall'azione semantica corrispondente. Per spiegare meglio questo meccanismo ricorriamo ad un esempio[3]):

```
sum_expression:      operand '+' operand
                     {
                        /* azione semantica:       */
                        /* "somma degli operandi"  */
                        ...
                     }
```

Ad ogni simbolo `operand` è associato un record che contiene le informazioni che lo descrivono ai fini dell'analisi semantica (ad esempio, il tipo, il valore ecc.). Al simbolo terminale `'+'` è invece associato un record che contiene solamente il token che rappresenta tale operatore. Come detto in precedenza, quando questa regola viene riconosciuta il parser attiva l'azione associata. I parametri di questa routine sono i record semantici appena descritti; il valore di ritorno è un'altra struttura che contiene le informazioni che descrivono il simbolo `sum_expression` (ad esempio, il tipo del risultato). Ulteriori esempi di azioni semantiche saranno illustrati in sezione 6.3.2.

6.3.2 Type Checking

Come detto in precedenza il controllo di correttezza di un'interrogazione non può essere esclusivamente sintattico ma deve riguardare anche la semantica. Il tipo di controllo che si può fare al momento della traduzione è detto di tipo *statico*, e si differenzia dal controllo che viene fatto a run-time, detto controllo *dinamico*. In questa sezione viene illustrata una forma di verifica statica detta *type-checking* (controllo di tipo).

Nel capitolo 3 è stata descritta una caratteristica fondamentale di OQL per la quale gli operatori del linguaggio possono essere liberamente composti purché venga sempre rispettato il sistema dei tipi. Il traduttore realizzato,

[3]la sintassi utilizzata è quella del file di specifica di Bison.

per poter verificare questa condizione, possiede un meccanismo di controllo di compatibilità dei tipi. Questo meccanismo verifica che il tipo di un costrutto corrisponda a quello richiesto dal contesto in cui compare, o, più semplicemente, verifica che ogni operatore sia applicato ad operandi di tipo compatibile[4]. Di conseguenza, per la progettazione del type-checker, deve essere dato un insieme di regole che permetta di:

1. verificare la compatibilità dei tipi.

2. dedurre il tipo di un'espressione da quello delle sottoespressioni.

Per spiegare più chiaramente i concetti esposti riportiamo dal manuale di OQL la regola relativa all'operatore `for all`:

"se `x` è un nome di variabile, e_1 ed e_2 sono espressioni[5], e_1 denota una collezione, ed e_2 è un'espressione di tipo booleano, allora `for all x in` e_1: e_2 è un'espressione di tipo booleano."

Si noti che nella regola citata è implicito che ogni espressione deve essere associata ad un tipo. A questo punto si pone il problema di come ottenere questa informazione. Nel caso del linguaggio OQL si possono individuare le situazioni seguenti:

- l'espressione rappresenta un tipo base: ad esempio

  `50`, `"steel"`, `true`

  In questo caso il tipo dell'espressione è immediatamente deducibile.

- il tipo dell'espressione non è deducibile dalle informazioni presenti nel testo della query e non può nemmeno essere ricavato dalle regole di compatibilità di OQL. Come esempio riportiamo la seguente espressione di percorso

  `S.stock.qty`

  In questo caso l'unico modo per risalire all'informazione inerente il tipo è quello di consultare lo **schema** della base di dati. Il procedimento da seguire prevede innanzitutto di ottenere, attraverso la symbol table, il tipo dell'oggetto riferito dalla variabile `S` (in questo caso supponiamo

[4] La definizione di compatibilità è riportata in sezione 3.3.1.

[5] in questo caso il termine "espressione" non si riferisce al *tipo* di un costrutto bensì è sinonimo di *interrogazione*.

si tratti della classe `Storage`). Successivamente bisogna navigare attraverso lo schema delle classi fino a raggiungere l'attributo `qty`, e con esso l'informazione cercata.

- il tipo dell'espressione può essere dedotto utilizzando le regole previste da OQL. Ad esempio

  `S.stock.qty` $>$ `50`

  In questo caso la regola relativa agli operatori relazionali prevede che il risultato sia un valore booleano. In altri casi, che non riportiamo per brevità, la regola può essere notevolmente complessa (ad esempio per il filtro `select-from-where`).

Un aspetto rilevante che emerge dalle considerazioni effettuate finora è che nel testo di una query OQL non compare in alcun modo l'informazione inerente il tipo. Le motivazioni di questa scelta sono state illustrate nel capitolo introduttivo di questo lavoro di tesi, e precisamente in sezione 1.2.2.

Per terminare il discorso sul type-checking illustriamo un altro esempio di regola di compatibilità di OQL, e precisamente quella del costruttore `set()`:

"se $e_1, e_2, \ldots, e_n$ sono espressioni di tipi compatibili $t_1, t_2, \ldots, t_n$, allora $set(e_1, e_2, \ldots, e_n)$ è un'espressione di tipo `set(t)`, dove $t = lub(t_1, t_2, \ldots, t_n)$. Il risultato è un'istanza di un insieme contenente gli elementi $e_1, e_2, \ldots, e_n$."

La funzione `lub()`, presentata in sezione 3.3.1, calcola il *least upper bound* dei tipi forniti come argomenti. Anche per la realizzazione di questa funzione bisogna ricorrere ad informazioni memorizzate nello schema della base di dati. Nello schema OCDL, in particolare, ogni classe o tipo è associato ad un insieme, presentato in [BN94], detto Generalization Set (GS) che viene calcolato dall'algoritmo di sussunzione e denota l'insieme dei tipi generalizzazione di un dato tipo. Il traduttore utilizza questo insieme per verificare la compatibilità tra tipi.

6.3.3 Symbol Table

Per implementare le regole di visibilità dei nomi previste in OQL è stata utilizzata una semplice *symbol table* che tiene traccia delle variabili che è possibile riferire in un dato punto della query. Una trattazione esauriente di questo argomento, piuttosto complesso, non rientra negli scopi di questa tesi;

pertanto saranno presentati di seguito soltanto gli aspetti inerenti al software realizzato.

Una symbol table è un meccanismo che associa dei nomi a dei valori. In generale, all'interno di un programma, un nome viene di solito definito soltanto in un punto, detto *dichiarazione*, ma può essere usato in molte altre parti del testo sorgente. La funzione della symbol table è quella di memorizzare, per ogni nome, le informazioni semantiche disponibili al momento della dichiarazione. In questo modo, ogni volta che un nome viene utilizzato nel programma, il parser può accedere alle informazioni semantiche associate servendosi di questa struttura dati.

Prima di descrivere le scelte progettuali compiute presentiamo l'insieme di regole definite da OQL che è stato necessario implementare.

Regole di visibilità dei nomi

La clausola `from` di un operatore `select-from-where` introduce, in maniera esplicita od implicita, una variabile per ogni collezione specificata. Un esempio di dichiarazione esplicita è il seguente

```
select ... from Storage S ...
```

mentre una dichiarazione implicita si presenta in questa forma

```
select ... from Storage ...
```

Il campo di visibilità di queste variabili si estende su tutte le parti del filtro `select-from-where`, incluse anche le query innestate. Una variabile, all'interno della propria sezione, quando denota un oggetto complesso, può essere usata per costruire espressioni di percorso:

```
S.maxrisk
```

È possibile inoltre utilizzare direttamente il nome di un attributo, senza fare riferimento ad una variabile:

```
maxrisk > 10
```

In questo caso però, tra gli oggetti denotati dalle variabili visibili in quel punto dell'interrogazione, deve esisterne soltanto uno per cui è definito l'attributo specificato.
Per spiegare meglio le regole esposte ricorriamo al seguente esempio:

```
select   S1
from     Storage as X,
         Material
where    exists(  select   S2
                  from     SStorage as Y
                           S3       as Z )
```

In sezione S_1 sono visibili i nomi: `Storage`, `Material`, `X`, tutte le proprietà delle classi `Storage` e `Material` purchè non abbiano lo stesso nome e non si chiamino "`Storage`", "`Material`" oppure "`X`".
In sezione S_2 sono visibili i nomi: `SStorage`, `Y`, `Z` e gli stessi nomi visibili in sezione S_1, eccetto `Y` e `Z`, se esistono.
In sezione S_3 sono visibili i nomi: `SStorage`, `Y` e gli stessi nomi visibili in sezione S_1, eccetto `Y` se esiste.
Si noti inoltre che nelle sezioni S_2 ed S_3 non è possibile accedere direttamente agli attributi delle classi `SStorage` e `Storage`, dato che, come si può facilmente verificare dallo schema, le proprietà di queste classi hanno lo stesso nome.

Realizzazione

Le tecniche di realizzazione di una symbol table sono molteplici, a seconda del numero dei nomi da memorizzare e delle prestazioni desiderate: lista (ordinata o disordinata), albero binario, hash table. Le scelte progettuali compiute sono le seguenti:

1. Nel software realizzato è stata implementata una gestione a lista disordinata. Non si è ritenuto necessario utilizzare una tecnica più sofisticata (e quindi più efficiente) poichè le prestazioni del metodo scelto rimangono accettabili fino ad una ventina di nomi memorizzati. Questa limitazione non è stata giudicata rilevante date le dimensioni medie di un'interrogazione.

2. Come esposto in precedenza, le regole di visibilità dei nomi previste da OQL permettono l'innestamento delle parti dichiarative. Per questo motivo la symbol table è stata strutturata a blocchi. Ogni blocco (o frame) corrisponde ad una parte dichiarativa della query, ovvero ad una clausola `from` di un filtro `select-from-where`. Quando il traduttore inizia l'analisi di questa parte, viene creato un nuovo frame in cui

vengono inseriti i nuovi nomi. Quando l'analisi dell'intero operatore `select-from-where` termina, il frame creato viene eliminato. Un'apposito stack si occupa di mantenere l'indice dei blocchi visibili ad un dato istante durante il processo di analisi semantica. In questo modo, ogni volta che in un'espressione compare, ad esempio, un nome di variabile, le informazioni necessarie vengono ottenute ricercando il nome nello stack, iniziando dall'ultimo frame creato e procedendo verso il primo. In figura 6.1 è riportato uno schema di massima del sistema adottato.

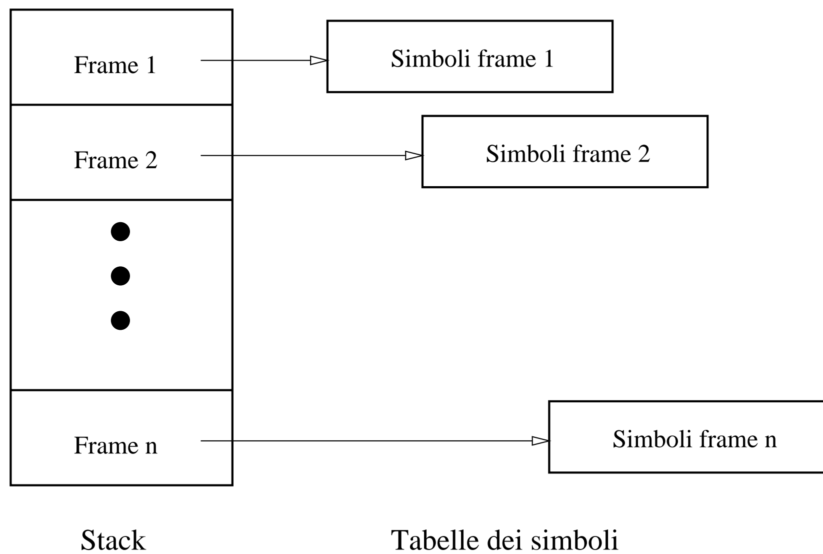

Figura 6.1: Symbol Table.

6.3.4 Errori

Una funzionalità essenziale del traduttore è quella di segnalare il tipo e la posizione degli errori individuati. Come scelta progettuale è stato deciso di arrestare il processo di analisi (sintattica e semantica) al primo errore individuato. Il messaggio di errore contiene il numero di linea del sorgente in cui si è verificato l'errore. Se l'errore è di tipo sintattico viene riprodotto in output il simbolo grammaticale errato. Se l'errore è invece di tipo semantico

viene visualizzato il costrutto impiegato impropriamente, unitamente a una breve descrizione del problema incontrato. In entrambi i casi viene visualizzato il numero di riga in cui si è verificato l'errore, in modo che sia semplice individuarlo.

Nella progettazione è inoltre stato curato il problema della propagazione degli errori durante la fase di analisi semantica. Quando l'azione associata ad un record semantico rileva un'espressione errata "marca" il record con un valore particolare, che segnala alle azioni successive di arrestare il processo di analisi. Se ciò non avvenisse l'errore si propagherebbe quasi certamente anche alle espressioni collegate a quella errata. In output verrebbero quindi riportati più messaggi, rendendo difficile individuare l'espressione sbagliata.

6.4 L'interfaccia OQL-OCDL

L'interfaccia OQL-OCDL è stata sviluppata in linguaggio ANSI C utilizzando il compilatore *gcc* versione 2.7.2 della GNU (Free Software Fundation, Inc.). La piattaforma hardware utilizzata è una SUN Sparc 20 con sistema operativo Solaris 2.x.

In figura 6.2 è riportato uno schema generale dell'interfaccia che illustra i moduli principali del software realizzato e la loro interazione con l'ottimizzatore ODB-QOPTIMIZER.

Il blocco OQL→OCDL compie l'analisi sintattica e semantica dell'interrogazione in linguaggio OQL e genera in output la rappresentazione OCDL delle parti traducibili. Questo blocco crea inoltre una struttura dati, che chiameremo *rappresentazione intermedia*, che serve a tenere conto anche delle parti non traducibili dell'interrogazione. L'analisi lessicale e sintattica della query sono effettuate dalle funzioni `yylex()` e `yyparse()`, realizzate attraverso gli strumenti BISON e FLEX. L'analisi semantica è invece eseguita da un modulo a parte che riceve in input le strutture dati generate da `yyparse()`. Per controllare la correttezza dell'interrogazione questo blocco deve accedere alle informazioni contenute nello schema della base di dati.

Il blocco OQL→OCDL ha il compito di riprodurre in uscita l'interrogazione, evidenziando le parti cambiate dal processo di ottimizzazione. Il modulo "Analisi OCDL" riceve in input la query in formato OCDL, converte i predicati nella sintassi OQL e aggiorna la rappresentazione intermedia. Quest'ultima, infine, viene convertita in una query OQL dal modulo "Generazione OQL", che può in questo modo riprodurre anche le parti della

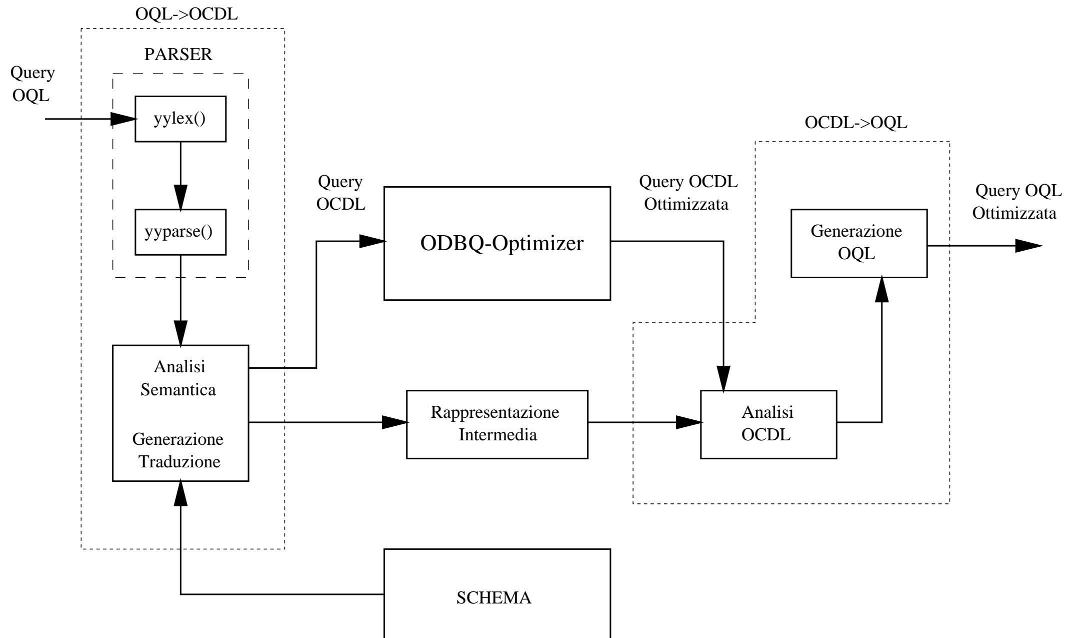

Figura 6.2: Architettura.

query iniziale che non erano state tradotte in OCDL-Designer dal blocco OQL→OCDL.

Per descrivere nei dettagli le funzionalità del software realizzato introduciamo brevemente il modello DFD (Data Flow Diagram).

6.4.1 Il modello DFD

Il modello DFD è uno strumento dell'ingegneria del software utilizzato in una delle prime fasi della progettazione per descrivere le funzionalità del prodotto che si deve sviluppare. Il DFD appartiene alla categoria dei modelli *semi-formali*, ovvero sistemi di rappresentazione basati su costrutti grafici che permettono di schematizzare, in modo non ambiguo, solo una certa parte della conoscenza acquisita. La parte restante viene invece descritta in modo informale oppure utilizzando un metodo di descrizione alternativo. Il modello DFD, in particolare, utilizza un formalismo estremamente semplice ed efficace, permettendo di dettagliare la rappresentazione, in fasi successive, fino a raggiungere il livello desiderato. D'altro canto non offre però alcuna

descrizione delle strutture dati e non è in grado di rappresentare l'ordine di esecuzione delle funzioni schematizzate. Per questo motivo in una delle sezioni seguenti verranno illustrate le principali strutture dati utilizzate dal software.

Gli elementi grafici del modello DFD sono: gli agenti esterni, i flussi di dati, i processi e i depositi di dati. In figura 6.3 sono visualizzati i simboli utilizzati.

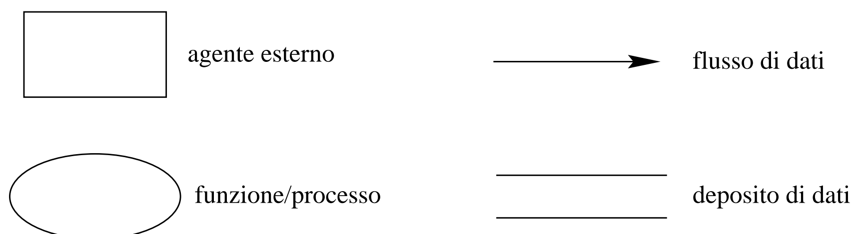

Figura 6.3: Simboli del modello DFD.

6.4.2 Architettura

In questa sezione viene illustrata l'architettura dell'interfaccia mediante i diagrammi DFD. Per limitare il numero e la complessità dei diagrammi preferiamo non raggiungere il livello di dettaglio della singola funzione software. Ci limitiamo ad indicare nel corso della descrizione le routine più importanti utilizzate.

In figura 6.4 viene riportato lo schema DFD generale del sistema costituito dal modulo di interfaccia e da ODB-QOptimizer. La funzione *interfaccia*, posta tra l'utente e il modulo ottimizzatore, gestisce la conversione delle interrogazioni tra il linguaggio OQL e il formalismo OCDL e viceversa. Per svolgere questo compito la funzione accede a due depositi di dati: il primo, denominato *schema*, rappresenta lo schema della base di dati ed è utilizzato per ottenere informazioni sul tipo degli oggetti coinvolti nell'interrogazione.

Il secondo deposito, denominato *symbol table*, contiene la descrizione dei nomi (simboli) che compaiono nel testo della query.

L'ordine di esecuzione delle funzioni rappresentate nello schema può essere riassunto nel modo seguente: inizialmente l'interfaccia converte il flusso di

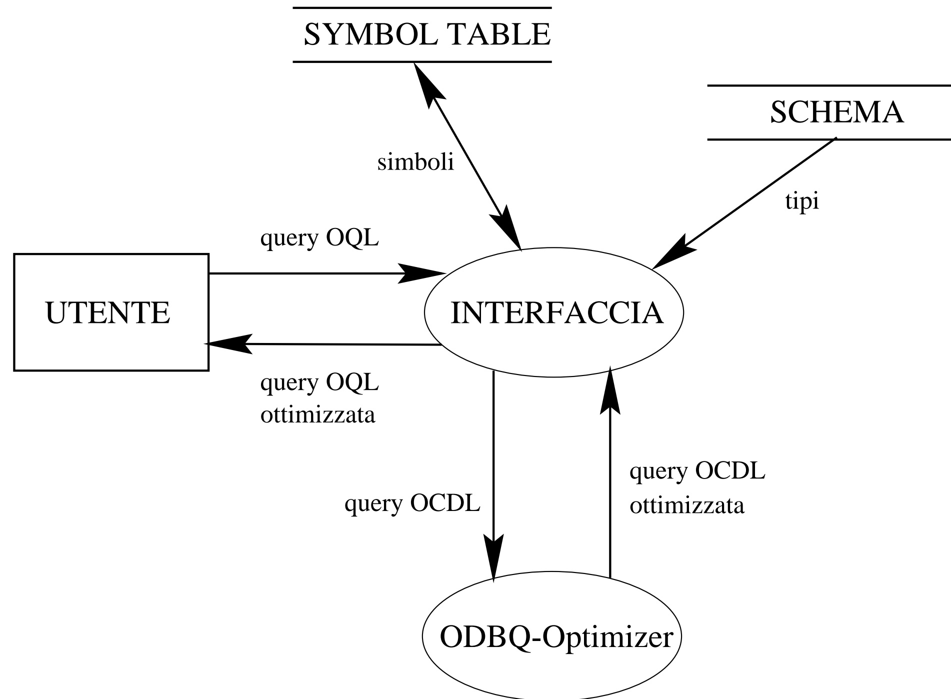

Figura 6.4: Schema DFD dell'architettura del sistema

dati *query OQL* nel formalismo OCDL ed invia l'interrogazione all'ottimizzatore attraverso il flusso *query OCDL*. Il modulo *ODBQ-Optimizer* restituisce l'interrogazione ottimizzata (flusso *query OCDL ottimizzata*) che viene nuovamente convertita in OQL dall'interfaccia e presentata all'utente attraverso il flusso di dati *query OQL ottimizzata*.

Esaminiamo ora nei dettagli la funzione d'interfacciamento, esplosa nello schema di figura 6.5. Innanzitutto notiamo che è stato introdotto un nuovo deposito di dati, denominato *rappresentazione intermedia*, che viene utilizzato per rappresentare l'interrogazione OQL nelle fasi che precedono e seguono l'attivazione dell'ottimizzatore. Questa struttura dati è necessaria poiché vi possono essere delle parti di query non traducibili in OQL e quindi non trattabili dall'ottimizzatore; di conseguenza determinate informazioni relative all'interrogazione iniziale necessitano di essere memorizzate temporaneamente in un'apposita struttura.

Descriviamo ora i nuovi flussi di dati rappresentati nello schema:

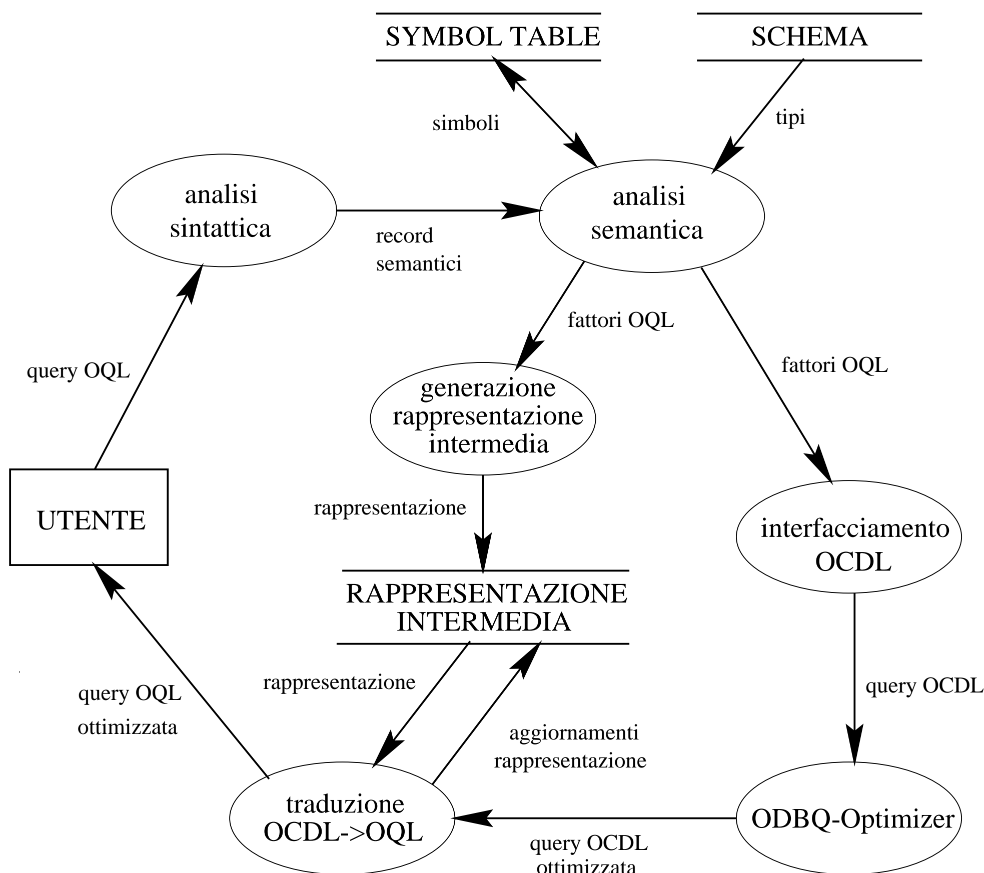

Figura 6.5: Schema DFD della funzione "interfaccia".

- *record semantici:* rappresentazione dell'interrogazione attraverso un'insieme di record.
- *fattori OQL:* elementi che corrispondono alle parti tradotte e non tradotte dell'interrogazione OQL. Nel primo caso contengono la rappresentazione OCDL del costrutto.
- *rappresentazione:* Descrizione completa dell'interrogazione OQL contenente informazioni sia sulle parti tradotte che su quelle non tradotte.
- *aggiornamenti rappresentazione:* parti di interrogazione OQL ottenuti dalla conversione della query OCDL ottimizzata. Sono utilizzati

per introdurre nella rappresentazione intermedia i cambiamenti dovuti all'ottimizzazione.

- *rappresentazione:* rappresentazione intermedia aggiornata della query, pronta per essere convertita nel testo finale OQL.

La funzione *analisi sintattica*, realizzata dalle routine software `yylex()` ed `yyparse()`, traforma il testo dell'interrogazione in una struttura di memoria composta da record semantici. Il processo *traduzione OQL→OCDL* utilizza in seguito tale struttura per effettuare l'analisi semantica e per tradurre, quando possibile, i singoli costrutti OQL. La principale funzione software utilizzata in questa fase, `ParseSigma()`, esamina tutti i record semantici generati dal processo precedente. Il metodo utilizzato verrà approfondito nelle sezioni seguenti.

Al termine di questa fase vengono attivati due nuovi processi. Il primo, *generazione rappresentazione intermedia* ha il compito di costruire una descrizione dell'interrogazione che tenga conto delle parti tradotte e non tradotte. Il secondo, *interfacciamento OCDL*, collega le parti di interrogazione rappresentate in OCDL organizzandole in una query vera e propria. Le funzioni software principali utilizzate da questi processi sono, rispettivamente, `OqlListFactorization()` e `BuildQuery()`.

Il processo *traduzione OCDL→OQL*, infine, riproduce in uscita l'interrogazione in linguaggio OQL introducendo i cambiamenti dovuti all'ottimizzazione. Esaminiamo ora nei dettagli questa funzione (illustrata in figura 6.6). Il processo *fattorizzazione* si occupa di scomporre in fattori l'interrogazione OCDL ottimizzata. In seguito la funzione *classificazione* ha il compito di distinguere quali tra questi fattori devono essere riportati in output. Al termine di questa fase viene attivata la funzione *traduzione OQL* che genera la rappresentazione OQL dei fattori da riprodurre in uscita. A questo punto la rappresentazione intermedia viene modificata dalla funzione *aggiorna rappresentazione intermedia* e in seguito viene convertita in una stringa di caratteri dal processo *generazione testo OQL*. La funzione *indentazione*, infine, migliora la leggibilità del testo generato. Tra le principali funzioni software utilizzate dai processi illustrati citiamo `ClassFactor()` per la classificazione dei fattori, `AttachPredicate()` e `AttachText()` per l'aggiornamento della rappresentazione ed infine `CopyQuery()` per la sostituzione nel testo originale delle parti ottimizzate.

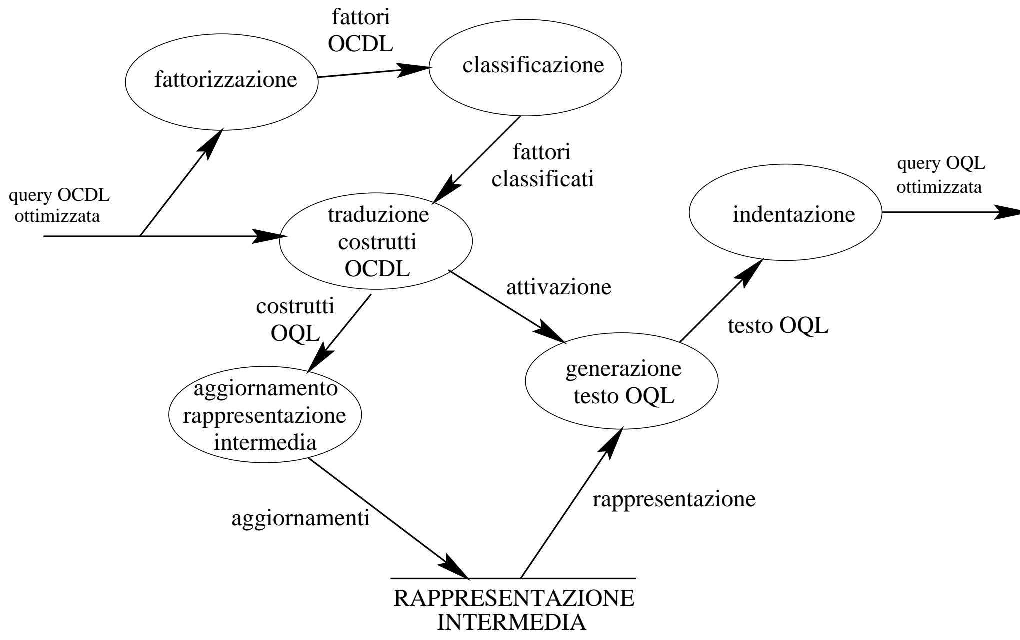

Figura 6.6: Funzione "traduzione OCDL→OQL".

6.4.3 Azioni semantiche

Nella sezione precedente è stato evidenziato che l'analisi sintattica e quella semantica avvengono in due fasi separate. Ciò è dovuto ad una precisa scelta progettuale: se i due processi fossero stati condensati in un'unica fase, l'ordine di esecuzione delle azioni semantiche sarebbe stato inevitabilmente legato all'ordine di riconoscimento delle regole grammaticali da parte della funzione `yyparse()`. Questo modo di procedere presenta però dei problemi, sia per l'implementazione dele regole di visibilità dei nomi illustrate in sezione 6.3.3, sia per l'analisi dell'operatore `select`. Per questa espressione, in particolare, risulta funzionale trattare prima la parte `from`, in cui vengono dichiarate le variabili, poi le clausole `where`, `order by` e `group by` ed infine la `select-list`. Con questo ordine di analisi è possibile dichiarare (ed introdurre nella symbol table) le variabili prima che siano utilizzate all'interno dei vari costrutti di OQL, facilitando notevolmente sia il controllo di correttezza dell'interrogazione sia il riconoscimento delle espressioni traducibili.

Per rendere possibile questo procedimento durante l'analisi sintattica ven-

gono creati appositi record semantici ai quali viene collegata l'azione semantica appropriata, tramite un puntatore a funzione. In generale, quindi, record diversi saranno associati ad azioni diverse. Nel caso dell'operatore `select`, ad esempio, la funzione collegata esamina prima le parte `from` e poi le altre clausole.

6.4.4 Eliminazione dei fattori

Come accennato in precedenza nel processo di espansione semantica eseguito da ODB-QOptimizer vengono esplicitati tutti gli attributi delle classi interessate dall'interrogazione. Occorre dunque individuare quali proprietà (fattori) devono essere riportate in output e quali devono invece essere ignorate. Per effettuare questa selezione è stato sviluppato, precedentemente a questo lavoro di tesi, un apposito modulo, indicato in fig 6.6 con la funzione *fattorizzazione*. L'interfaccia realizzata si serve di questo modulo, durante la fase di output dei risultati, per distinguere le espressioni OCDL, generate dall'ottimizzatore, di cui occorre effettuare la traduzione in OQL. Per distinguere tra i vari cambiamenti introdotti dall'ottimizzatore (specializzazione di classe, introduzione di un nuovo predicato ecc.) è stata realizzata la funzione `ClassFactor()` che integra il modulo citato.

È importante notare che la distinzione effettuata è di natura puramente sintattica, non si riferisce cioè all'eliminazione dei fattori *ridondanti* dell'interrogazione, problema che è tuttora oggetto di studio.

6.4.5 Strutture Dati

In questa sezione vengono illustrate alcune delle principali strutture dati utilizzate.

OQL è un linguaggio che possiede una grande varietà di tipi di dati. I record semantici devono quindi riuscire a memorizzare tipi di informazione molto diversi, a seconda del simbolo grammaticale a cui sono associati. D'altra parte vi è l'esigenza implementativa di organizzare i record semantici in una struttura di memoria omogenea, composta da elementi tutti uguali tra loro. Da queste considerazioni appare chiaro che la struttura dati destinata alla rappresentazione dei record deve avere una parte fissa, in cui sono memorizzate le informazioni comuni a tutti i record, e una parte variabile che contiene le informazioni specifiche di ogni simbolo della grammatica. In

particolare il record utilizzato dal traduttore, riportato in tabella 6.1 è strutturato nel modo seguente: la parte variabile è costituita dal campo `Rec` della struttura, nel quale viene riservato spazio sufficiente per memorizzare le informazioni specifiche di qualunque tipo di record. In pratica l'utilizzo della `union` consente di combinare strutture differenti (ad esempio `BaseTypeRec` e `ComparisonRec`) in un unico campo. In tabella, per brevità, sono state riportate solo alcune delle strutture contenute da tale campo.

```
typedef struct SemanticRecT {
  UINT                  uiLine;
  eSrTypeT              eRecType;
  UINT                  uiOdmgType;
  char                  *sObjName;
  union {
    BaseTypeT           BaseTypeRec;
    NameRecT            NameRec;
    IterRecT            IterRec;
    SelectRecT          SelectRec;
    ComparisonRecT      ComparisonRec;
    ...
    ...    /* altri tipi */
  } Rec;
  void                  (*Action)();
  struct OqlFactorT     *pofFactor;
  struct L_sigma        *lsSchema;
  struct SemanticRecT   *psrNext;
  struct SemanticRecT   *psrNested;
} SemanticRecT;
```

Tabella 6.1: Record semantico utilizzato dal traduttore.

La parte fissa è costituita dagli altri campi del tipo `SemanticRecT` e precisamente:

uiLine: numero di linea del sorgente in cui è stato rilevato il simbolo grammaticale a cui corrisponde il record.

eRecType: tipo del record semantico; i possibili valori che può assumere questo campo sono riportati nel file `ootypes.h`.

uiOdmgType: costante che indica il tipo ODMG–93 dell'elemento rappresentato:

- `ODMG_UNDEFINED` = tipo non definito
- `ODMG_long` = tipo intero
- `ODMG_float` = tipo reale
- `ODMG_string` = tipo stringa
- `ODMG_boolean` = tipo booleano
- `ODMG_char` = tipo carattere
- `ODMG_list` = tipo lista
- `ODMG_set` = tipo set
- `ODMG_bag` = tipo bag
- `ODMG_array` = tipo array
- `ODMG_struct` = tipo struttura
- `ODMG_object` = tipo oggetto con oid
- `ODMG_class` = tipo classe

il tipo classe, che non fa parte del sistema dei tipi di ODMG–93, serve a distinguere tra intensione ed estensione di una classe.

sObjName: nome di tipo valore o tipo classe nello schema OCDL.

Action: puntatore all'azione semantica.

pofFactor: puntatore al fattore OQL della rappresentazione intermedia.

lsSchema: puntatore ad un elemento dello schema OCDL.

psrNext: puntatore al prossimo record nella lista.

psrNested: puntatore ad un ulteriore livello di innestamento.

Come esempio di struttura contenuta nella parte variabile riportiamo quella utilizzata per gli operatori relazionali:

```
typedef struct ComparisonRecT {
  int                   iOperator;
  int                   iQuantifier;
  struct SemanticRecT   *Op1;
  struct SemanticRecT   *Op2;
} ComparisonRecT;
```

I campi utilizzati sono:

iOperator: indica il tipo di operatore utilizzato ($>$, $\geq$, ecc.).

iQuantifier: indica se è stato specificato un quantificatore (`any`, `some`, `all`).

Op1: puntatore al primo operando.

Op2: puntatore al secondo operando.

Come si vede le informazioni rappresentate da questa parte del record riguardano in maniera specifica gli operatori relazionali.

La rappresentazione intermedia deve contenere le informazioni necessarie a riprodurre la query in linguaggio OQL tenendo conto dei cambiamenti introdotti dall'ottimizzatore. In particolare questa struttura, poichè deve rispecchiare in qualche modo l'interrogazione iniziale, viene creata durante la fase di analisi semantica e di conseguenza i suoi elementi vengono anche utilizzati per memorizzare la rappresentazione OCDL dei costrutti tradotti. Al termine di questo processo le parti tradotte vengono assemblate in'unica struttura di memoria che rappresenta l'interrogazione OCDL. In base ai risultati della fase di ottimizzazione i nodi vengono successivamente aggiornati con il testo OQL da riportare in uscita, oppure l'albero può essere modificato per l'introduzione di nuovi predicati. La struttura utilizzata ha la forma seguente:

```
typedef struct OqlFactorT {
  eSrTypeT              eType;
  eStatusTypeT          eStatus;
  char                  *sItName;
  struct BoxT           *pboxBoxList;
  struct L_sigma        *lsSigma;
  struct L_sigma        *lsSigmaLink;
  char                  *pcOptText;
```

```
  struct fattori2       *pfOcdlFactor;
  struct OqlFactorT     *pofSubFactor;
  struct OqlFactorT     *pofSubFactor2;
} OqlFactorT;
```

I campi della struttura hanno il seguente significato:

eType: indica il tipo del fattore rappresentato (classe, quantificatore, operatore relazionale ecc.). L'elenco completo dei tipi è riportato nel file `ootypes.h` in appendice.

eStatus: stato del fattore (modificato, nuovo, non modificato ecc.)

sItName: nome della variabile utilizzata dai quantificatori e dalle classi. Serve a riprodurre in uscita gli stessi nomi di variabile utilizzati nella query iniziale.

pboxBoxList: puntatore alla struttura che memorizza la posizione nel testo del fattore.

lsSigma: puntatore alla testa di una lista di tipo `L_sigma`. Usato durante la fase di analisi semantica per memorizzare la traduzione OCDL.

lsSigmaLink: puntatore alla coda di una lista di tipo `L_sigma`. Usato durante la fase di analisi semantica per memorizzare la traduzione OCDL.

pcOptText: puntatore alla stringa di testo in linguaggio OQL che rappresenta una parte ottimizzata da riprodurre in output.

pfOcdlFactor: puntatore a un fattore OCDL. Utilizzato per identificare il nodo da aggiornare.

pofSubFactor puntatore al sottoalbero di sinistra.

pofSubFactor2 puntatore al sottoalbero di destra.

Per le strutture dati utilizzate da ODB-QOPTIMIZERsi rimanda a [Vin94].

Un esempio di codifica

Illustriamo infine un esempio di codifica della struttura di record per l'analisi semantica. L'interrogazione rappresentata è la seguente:

```
select   M.name
from     Material M
where    M.risk > 10
```

In corrispondenza di questa query il parser genera una struttura di record semantici del tipo di quella riportata in figura 6.7. La figura si riferisce al-

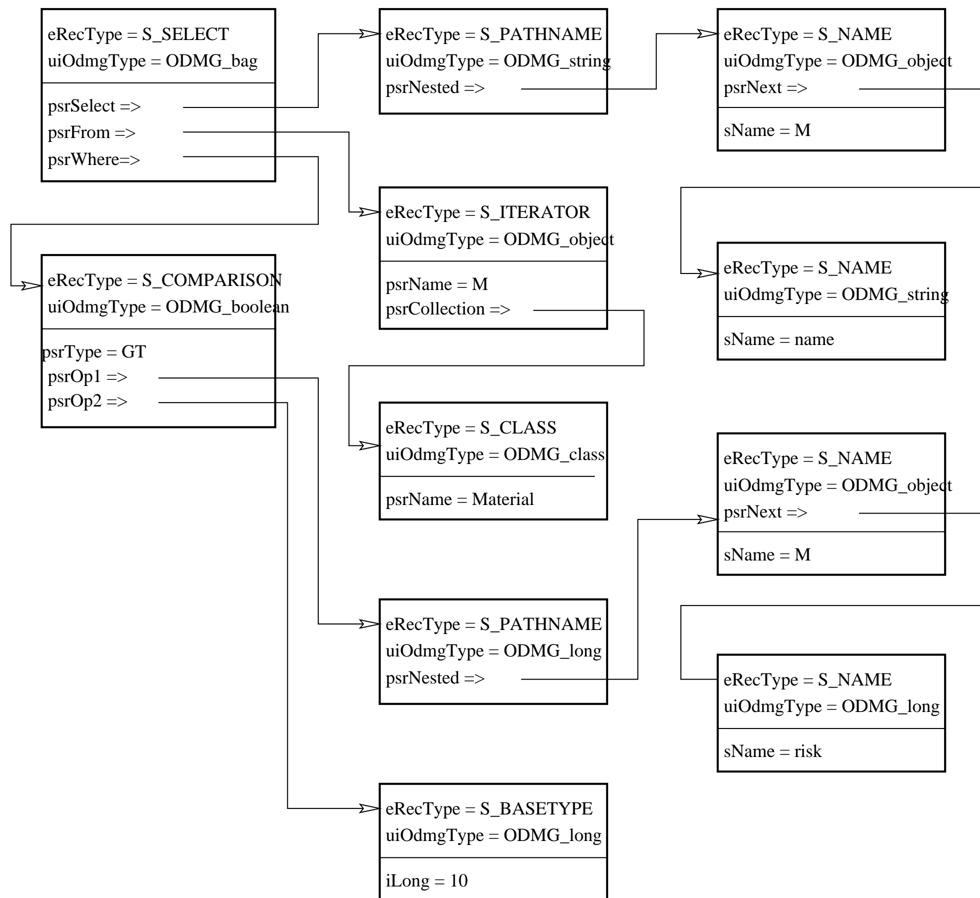

Figura 6.7: Esempio di codifica.

lo stato della struttura dati dopo che è stata effettuata l'analisi semantica

e sono stati individuati i tipi dei dati coinvolti nella query. Nella semplice notazione utilizzata i record sono rappresentati mediante rettangoli, suddivisi in due parti, contenenti i campi di maggior interesse della struttura. La parte superiore rappresenta la struttura fissa del record, quella inferiore rappresenta la parte variabile. Quest'ultima è assente da alcuni dei record più semplici perchè non svolge alcuna funzione. Si nota inoltre un record "radice", corrispondente all'operatore `select`, dal quale ha inizio il processo di analisi semantica.

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.

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley, 1986.

[Atz93] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects.* Springer-Verlag: LNCS n. 701, Heidelberg - Germany, 1993.

[BB96] D. Beneventano and S. Bergamaschi. Incoherence and subsumption for recursive views and queries in object-oriented data models. *Data & Knowledge Engineering*, 1996. To appear.

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

[Ben94] 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.

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

[BN94] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

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

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93, Release 1.1.* Morgan Kaufmann Publishers, Inc., 1994.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler.* The Benjamin/Cumming Publishing Company, Inc., 1988.

[Gar95] A. Garuti. *OCDL-Designer: un componente software per il controllo di consistenza di schemi di basi di dati ad oggetti con vincoli di integrità.* Tesi di Laurea, Facoltà di Scienze MM. FF. NN., Corso di Laurea in Scienze dell'Informazione, Università di Bologna, Bologna, 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.

[JBBV95] D. Beneventano J.P. Ballerini, S. Bergamaschi, and M. Vincini. Odb-qoptimizer: un ottimizzatore semantico di interrogazioni per oodb. In A. Albano, S. Salerno, F. Arcelli, M. Gaeta, S. Rizzo, and G. Vantini, editors, *Convegno su Sistemi Evoluti per Basi di Dati*, June 1995.

[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.

[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.

[MB91] T. Mason and D. Brown. *Lex & Yacc*. O'Reilly & Associates Inc., 1991.

[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.

[SSS92] M. Siegel, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Transactions on Database Systems*, 17(4):563–600, December 1992.

[Vin94] M. Vincini. *ODBQ-Optimizer: un sistema per l'ottimizzazione delle interrogazioni nelle basi di dati ad oggetti complessi.* Tesi di Laurea, Facoltà di Ingegneria, Corso di Laurea in Ingegneria Informatica, Università di Modena, Modena, 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.