Technical Report

UCAM-CL-TR-547
ISSN 1476-2986

Number 547

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Semantic optimization of OQL queries

Agathoniki Trigoni

October 2002

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 2001 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Pembroke College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Abstract

This work explores all the phases of developing a query processor for OQL, the Object Query Language proposed by the Object Data Management Group (ODMG 3.0). There has been a lot of research on the execution of relational queries and their optimization using syntactic or semantic transformations. However, there is no context that has integrated and tested all the phases of processing an object query language, including the use of semantic optimization heuristics. This research is motivated by the need for query execution tools that combine two valuable properties: i) the expressive power to encompass all the features of the object-oriented paradigm and ii) the flexibility to benefit from the experience gained with relational systems, such as the use of semantic knowledge to speed up query execution.

The contribution of this work is twofold. First, it establishes a rigorous basis for OQL by defining a type inference model for OQL queries and proposing a complete framework for their translation into calculus and algebraic representations. Second, in order to enhance query execution it provides algorithms for applying two semantic optimization heuristics: constraint introduction and constraint elimination techniques. By taking into consideration a set of association rules with exceptions, it is possible to add or remove predicates from an OQL query, thus transforming it to a more efficient form.

We have implemented this framework, which enables us to measure the benefits and the cost of exploiting semantic knowledge during query execution. The experiments showed significant benefits, especially in the application of the constraint introduction technique. In contexts where queries are optimized once and are then executed repeatedly, we can ignore the cost of optimization, and it is always worth carrying out the proposed transformation. In the context of adhoc queries the cost of the optimization becomes an important consideration. We have developed heuristics to estimate the cost as well as the benefits of optimization. The optimizer will carry out a semantic transformation only when the overhead is less than the expected benefit. Thus transformations are performed safely even with adhoc queries. The framework can often speed up the execution of an OQL query to a considerable extent.

To my family

Acknowledgements

I would like to thank my supervisor, Dr Ken Moody, for his invaluable guidance, sustained interest and encouragement during the three years of this research. In spite of his busy schedule, he always made himself readily available to discuss a variety of issues and to assist me in the shaping of my thesis. I am also grateful to Dr Gavin Bierman, my second supervisor, for teaching me type theory and formalization and for patiently helping me with a variety of difficult theoretical problems. The contribution of Dr Jean Bacon must also be acknowledged: working in a slightly different research area, she often provided me with a much needed external viewpoint. I also owe much to Robin Fairbarns and Graham Titmus who provided me with extensive software and hardware support.

This research would never have been possible without the kind and joint financial support from the Greek State Scholarships Foundation (IKY) and my employer, the National Bank of Greece (NBG). In particular, I wish to thank my IKY advisor, Dr Alexiou, for his insightful advice and encouraging progress reports at the end of each year. I am also much indebted to my superiors in the IT Division of NBG, K. Economopoulos, K. Marinakis and E. Synodinou, for promoting my application for graduate studies, and to the Human Resource Development Division for arranging my leave of absence.

I also feel compelled to thank my many friends and colleagues in the OPERA Group. I am grateful to Oliver and Chaoying for introducing me to the OPERA Group research and to Alan, Alexis, Andrei, Andras, Aras, Brian, Peter, Walt and Wei for the enjoyable experiences that we shared together.

On a more personal note, I would like to thank Constantin for making Cambridge seem like a wonderful place even at the worst of times. I greatly benefited from both his practical and emotional support. Back home, I am indebted to my friend Spyros for acting on my behalf on a variety of issues and for making Greece seem a little closer than it really was. I am also grateful to my flatmates and other friends in Cambridge for those precious relaxing moments away from my computer.

Finally, I owe a great deal to my family and in particular to my parents for their love and unceasing support. I may never have undertaken this thesis had they not always encouraged me to pursue my interests and inspired me with their idealistic view of research.

Contents

List of Figures

Chapter 1

Introduction

Query processing and its optimization have been two of the most popular areas of research in the database community. A lot of experience has accumulated, mostly on the execution of relational queries and their optimization using syntactic transformations. Recently, much interesting work has focused on semantic query optimization, but relatively few frameworks have been built to test the effectiveness of these techniques. To my knowledge, there is no research context that has developed and implemented all the phases of processing an object query language; further, relatively few attempts have been made towards using data mining techniques, e.g. association rules, in order to optimize queries semantically.

The comparative analysis of relational and object-oriented database systems is one of the most controversial database issues [CLC95, DD88]. The efficiency of data storage and query execution is the main advantage of the relational systems; however the advocates of the object-oriented world claim that their paradigm is more expressive. The vast experience of optimization techniques and index usage is a strong argument used by the supporters of RDBS; the advent of object-oriented programming languages and their natural connection with OODBS is an important counter-argument. Object-relational databases try to offer the best aspects of the two database paradigms [CMN95]. They allow the user to create persistent objects, but they use a relational physical organisation to store them.

The limited experience in developing purely object-oriented database systems means that many query processors have limited capabilities and several deficiencies. The work reported in this thesis was motivated by the lack of query engines that possess two valuable properties: i) the expressive power to encompass all the features of the object-oriented paradigm and ii) the flexibility to benefit from experience gained from relational systems, such as the use of semantic knowledge to speed up query execution.

The objective of this work is to explore all the phases of developing an object query processor

and to provide algorithms for applying two semantic optimization heuristics. The implementation of this framework enables us to measure the cost and the benefits of putting semantic knowledge in the service of query execution. It should be pointed out that the conclusions drawn from the experiments are equally applicable to relational query processors.

1.1 Relevant Research Areas

Developing a query processing framework and enriching it with semantic optimization techniques involves using techniques and resolving issues from disparate research areas.

Various aspects of type theory are studied in order to ensure the typability of a query. In particular, type checking is a means of exploring whether a certain query can be executed against a particular schema without causing any type-errors. Type inference is used to infer the type of a query without accessing any particular schema information. It can be used to identify the requirements that a schema should satisfy to be compatible with a certain query.

Part of this dissertation also involves work in translating a query into various intermediate representations before it actually gets executed. This entails not only translating a query from one form to another, but also optimizing it at each stage. A lot of work has been done in this area in the relational framework, but relatively few attempts have been made to do the same in an object-oriented context.

Semantic optimization is a third important aspect of our query processing framework. The underlying idea is to optimize a query based on semantic knowledge in the form of association rules. This first requires identifying the rules using standard data mining methods; these rules must then be combined into forms suitable for the optimization purposes.

Finally, assessing the performance benefits of semantic optimization requires estimating the cost and the scalability of the methods developed. In the last part of this thesis we look at the complexity of the algorithms involved.

1.2 Research Context

The query language used in our framework is OQL, the Object Query Language proposed as part of the Object Data Standard (version 3.0) published by the ODMG [CBB+00]. This standard provides specifications for storing and retrieving objects in databases. Its objective is to encompass all the important features of the OO paradigm and to ensure portability across database systems that comply with it. It consists of specifications of the following components:

- the Object Model defines the type system of the object paradigm,

- the Object Definition Language (ODL) is a specification language used to define the classes of a particular schema,
- the Object Interchange Format (OIF) is a specification language used to dump and load the state of the database to and from a file,
- the Object Query Language (OQL) is a functional language that allows the user to query objects,
- C++, Java, Smalltalk bindings enable developers to write code in the respective language to manipulate persistent objects.

A general criticism of a previous release (1.2) of the Object Standard can be found in [Ala97]. In this thesis, we focus on OQL, since it is the query language studied throughout this dissertation. It is a functional language, in which queries can be composed to an arbitrary depth, as long as their operands and results respect the type system defined in the Object Model. Many of the constructs of OQL are syntactically similar to those of SQL-92; however, OQL has a number of object-oriented features. First, it deals with complex objects, i.e. objects containing or being related to other objects, or containing collections (or structures) of objects. Secondly, it enables users to navigate over the objects using path expressions of arbitrary length. Thirdly, OQL allows the comparison of objects based on their object identity instead of their content values. Fourthly, it enables method invocations within a query. Fifthly, it manipulates a series of collections (sets, bags, lists, arrays, dictionaries) with the aid of additional language constructs. Sixthly, it handles polymorphic collections, i.e. collections with objects belonging to different classes. The late binding mechanism enables users to perform generic operations on the elements of these collections. The casting mechanism allows them to declare explicitly the class of an object, going down the class hierarchy.

The expressive power of OQL makes it an interesting query language to develop. In addition to its independent use as a query language, it can potentially be embedded in any object-oriented programming language. The object-oriented nature of OQL makes the gap between the query and the programming language much narrower than in the relational context. In certain cases, it allows the programmer to perform exactly the same operation by either using the commands available in the binding of the programming language, or by embedding an OQL query in it.

1.3 Existing Object-Oriented Database Systems

There currently exist several companies who claim to have designed OODB systems compliant with the ODMG standard. For instance, SunMicrosystems, POET, Object Design and Objectivity and

Versant have developed Java Blend, Poet Object Server, ObjectStore, Objectivity/DB and Versant Developer Suite respectively.

Java Blend enables users to write database applications for any database entirely in the Java programming language. Developers do not need to know how data are structured in a database, they should just be aware of the business logic underlying their application. The main idea behind building this product was to avoid using any query language resembling SQL.

ObjectStore enables the database to be accessed through two language bindings, for Java and for C++. It can either be used as a middle-tier Data Server that links to a conventional relational database, or as a real object-oriented database system. The rationale behind its development is similar to Java Blend. It does not provide an independent object query language, but enables programmers to query, update or delete objects using the language binding of their choice. Although this allows developers to access the database without switching to a query language, it deprives them of the expressiveness and the flexibility of OQL. Query processing in ObjectStore is studied in [OHMS92].

Objectivity/DB also includes C++ and Java bindings that conform to the second release of the ODMG standard (ODMG 2.0). In addition, it provides a separate query facility called SQL++. This is an ANSI-standard SQL engine extended with object-oriented features of Objectivity/DB. The design of SQL++ was intended to preserve the investment in traditional SQL-based tools; the effect of this was a loss of expressive power compared to OQL, such as the failure to handle multiple collections, late binding and class casting.

Versant Developer Suite (VDS) 6.0 is an object database management system that provides a tight binding between the application programming language (C++ or Java) and the underlying object database model. Versant has also its own proprietary Versant Query Language (VQL). VQL does not only have select query capabilities, but also supports insert, update and delete operations. Queries have an SQL-like syntax extended to include a number of object-oriented concepts. However, VQL does not have the expressive power of OQL.

A more detailed description is given for POET, since it has been used as the basis for developing the query framework of this research. POET has actually contributed in defining the ODMG standard. It provides two language bindings (for Java and C++), as well as an OQL query processor. However, only a subset of OQL constructs is implemented; neither method invocation nor the group-by operation are supported in queries. The type system of the ODMG object model is not entirely respected. For instance, the result of a query in POET is either of type `Object` or of type `Set(Object)`. According to the standard, it could be any simple type, class or collection like bag, list or array. The weaknesses of the POET OQL implementation motivated in part the development of a new OQL engine. Since in most other respects – object model, language bindings – POET conforms to the ODMG standard, its storage system and Java binding were selected as tools for building and testing the new query processor.

1.4 Type Checking vs Type Inference

The ODMG standard presents, rather informally, some details of a type system for checking OQL queries using type information about the classes, extents, named objects and query definitions from a given database schema. Recently, there have been some efforts to formalise this type system [Ala99, BT00]. Their common objective is to ensure that the query execution will not fail due to type errors. This is also reflected by the following statement in the ODMG standard:

> "OQL is a functional language where operators can freely be composed, as long as the operands respect the type system" ([CBB+00], page 89).

The core of the type checking model [BT00] is a set of typing rules which derive the type of OQL constructs based on the types of their constituent parts (operands or subqueries). If a subquery is just an identifier its type is looked up either in the query environment, if it is a local range variable, or in the database schema, if it is a named object, a query definition or an extent. Hence, knowledge of the schema type information of a database is essential for type checking.

On the other hand, type inference derives the most general type of a query without accessing any schema information. Whilst doing that, it establishes restrictions on the properties of a schema, in order that a certain query may be well-typed with respect to it.

This technique could be exploited in distributed database applications where we often need to check the typability of a query against multiple schemata. It would also be useful in order to tranform queries into equivalent forms that fit different schema specifications. Finally, a type inference model could be exploited in contexts where we have a limited knowledge of schema information (semi-structured or unstructured databases).

OQL type inference is particularly interesting, because of the complexity of the type model of the language. Besides class types and simple types, the type model includes structures, as well as multiple collection types (sets, bags, lists, arrays and dictionaries). The wide variety of OQL constructs leads to multiple combinations of these types, and thus to a rather complex type system. Type inference is studied in detail as a part of this thesis.

1.5 Calculus and Algebraic Representations

Most query languages, including OQL, are declarative, i.e. they declare what has to be provided as the result, without describing how this is going to be achieved. Intermediate representations aid in bridging the gap between the user-level language and the query engine programs ultimately used to produce the query result. There are calculus-style and algebra-style intermediate representations that serve different purposes, but both need to reflect the special characteristics of the query language. Calculus-style formalisms support the concept of tuple (for relational models) or object (for

object models) variables, while algebra-style formalisms are set- (resp. collection-) oriented. The former allow for nesting of expressions in a natural way; the latter use operators that reflect mutually independent (context free), but well defined execution steps [SdBB96].

In this thesis, the monoid comprehension calculus is adopted as the initial internal representation of OQL. This calculus provides a uniform way of expressing a great variety of queries; for instance, operations over multiple collection types, group-by operations, existential or universal quantification have very similar calculus representations. The calculus framework of this dissertation has extended previous research [Afs98, FM95, FM00], in order to deal with several weak points of OQL and to resolve a number of ambiguities. One of these extensions concerns the introduction of new normalization rules wrt [FM00]. Normalization is the transformation (unnesting) of monoid expressions to syntactically equivalent ones which are either more efficient, or provide opportunities for further calculus or algebraic optimization.

A set of algebraic operators (e.g. Reduce, Select, Nest) are proposed as the second internal representation of OQL. In order to translate a query from the calculus to its algebraic form, one needs to apply a series of translation rules. Having obtained the algebraic form of a query the next step is to provide transformation rules that convert an algebraic form to a more efficient form. Note that the term *translation* is used to express the conversion from one internal representation to another; the term *transformation* denotes the conversion of an expression to an optimized one within the same internal representation [SdBB96]. The algebraic operators presented in this thesis are similar to those proposed by Fegaras and Maier [FM00]; however, a great deal of the semantics and the applied transformations are different in the two contexts. In addition, the respective algorithms for the translation of calculus to algebraic forms differ to a great extent.

1.6 Semantic optimization using association rules

In general, optimization methods transform a query from one representation to a more efficient one, or to a form that provides opportunities for further optimization. Calculus normalization and algebraic optimization, which have traditionally dominated the query evaluation process, involve transformations that are valid for any database state. In contrast, semantic transformations are valid for any state satisfying a set of constraints (semantic rules) [SSS92]. These rules differ from database integrity constraints, in that they are not strictly imposed, that is they do not reject any update that invalidates them. Such an update would rather cause the relaxation or the deletion of the rules it invalidates.

Association rules reflect correlations between attribute values in a database. They have mainly been studied in the context of sales transactions databases. A lot of work has focused on the market-basket problem, i.e. on finding association rules of the form *if a set of items appears in a transaction, then another different set of items is likely to appear as well*. These *boolean* association rules have

been extended to deal with quantitative and categorical values.

Two measures – support and confidence – express the validity of association rules. Support is the fraction of data instances satisfying a certain rule, while confidence is the ratio of instances satisfying the rule to the instances satisfying its antecedent. Semantic rules can be viewed as association rules with 100% confidence. Semantic optimization could be achieved using association rules, as long as their exceptions are taken into account in the query transformations. We assume that these exceptions are maintained in the presence of updates [AFP00, CLK97]. Considering the exceptions of association rules results in valid but not necessarily advantageous semantic transformations.

Siegel has proposed a series of heuristics for the semantic optimization of queries [SSS92]. The following two are studied in detail in this thesis:

- if a selection on attribute A is implied by another selection condition on attribute B and A is not an index attribute, then the selection on A can be removed from the query;
- if a relation R in the query has a restricted attribute A and an unrestricted cluster index attribute B, then look for a rule where the restriction on A implies a restriction on B.

A number of algorithms proposed in this dissertation enable the application of both heuristics for the optimization of OQL queries. In addition, an experimental framework has been set up, in order to measure the complexity of the optimization algorithms and the performance benefits of the query transformations.

1.7 Dissertation outline

The rest of this thesis is structured as follows: Chapter 2 presents first a series of applications for which this research is particularly useful. These applications are complex enough to benefit from an object-oriented paradigm and are rich in meaningful correlations in order to provide the opportunity for semantic optimization. Further, this chapter reviews previous work related to the various phases of query processing and optimization. In doing so, it points out the main differences between previous work and the research presented in this thesis.

Chapter 3 focuses on the first step of query processing, that is the type inference of OQL queries. Note that there is no discussion about checking the type of an OQL query with respect to a certain schema. The type-checking algorithm is quite similar to the ones proposed for relational query languages and is formally presented in [BT00]. Instead this chapter focuses on the following problem: given an OQL query find its most general type as well as the type requirements that a schema should satisfy in order to be compatible with the query.

Chapter 4 discusses the translation of an OQL query to its first internal representation, the monoid comprehension calculus. We argue against the claim of Fegaras and Maier that nearly all OQL

expressions have a direct translation into their monoid calculus, with the exception of indexed OQL collections ([FM00],section 3). We extend the calculus syntax and modify the translation rules, in order to provide a full mapping of OQL queries to their calculus forms. Further, we present additional rules for the normalization of monoid calculus expressions taking into account the new calculus expressions.

Chapter 5 is the equivalent of chapter 4 on the algebraic level. That is, it first gives an overview of the algebraic operators used at this level and explains their functionality; it then presents the rules for the translation of the calculus representation to the algebraic one. The reader will notice the differences between the algebraic representation of this thesis and the one proposed in [FM00].

Chapter 6 introduces the interesting topic of semantic optimization. It gives a series of algorithms for applying two optimization heuristics proposed by Siegel et al. [SSS92]. It then discusses the contexts in which each of these techniques is expected to speed up query execution.

Chapter 7 evaluates the complexity of the semantic optimization techniques and assesses their benefits in query execution. These issues are first addressed theoretically, by reasoning about the cost of each step of the optimization algorithms. Complexity is then evaluated practically, by setting up a series of experiments that monitor the behaviour of the algorithms under different input conditions.

Chapter 8 discusses the insights and the experience gained from this research. It presents some weaknesses and suggests directions for future work. Finally, it concludes this dissertation by pointing out its principal contributions.

Chapter 2

Applications and Related Work

This chapter discusses two important issues relevant to this work. First, it presents application frameworks that can benefit from the algorithms developed in this research. These applications are expected to be rich in semantic associations and complex enough to benefit from their deployment in an object-oriented framework. Second, this chapter reviews previous research done on the various stages of query execution and optimization. It focuses mostly on the work that has directly influenced this research, and highlights the novel points of this dissertation.

2.1 Applications that motivate this research

The applications that are particularly interesting in our context are often labelled as 'data-rich'. This term has multiple interpretations: first, it requires that an application has large amounts of data that need to be managed in an intelligent way. Secondly, it implies that this data is inherently complex, and hence it would benefit from being structured in an object-oriented database. Thirdly, this term suggests that the data is rich in semantic correlations; the latter could be used either for query optimization or for decision-support purposes.

These applications can be found in both commercial and scientific contexts. Starting with the commercial world, most organisations are collecting increasing volumes of data about business processes and resources. Traditionally, they organise this data in relational databases, representing interconnections between tables by means of foreign keys. They query the information in order to produce long reports referring to aggregates of costs, earnings and various other statistical information. More complicated queries, aiming at risk analysis or prediction of future trends, are usually performed in a semi-manual way. Due to the volume of data, queries that range over entire tables (or extents) take a long time to evaluate; they are typically included in batch jobs and performed overnight.

For instance, consider a banking application that keeps data about all customers that have had a

loan contract with a bank. It would be very convenient if the bank information was organized in an object-oriented database, since one would be able to navigate easily from customers to their loans and deposits, their repayments and other obligations, as well as other customers they are related to, e.g. guarantors or partners in the same company, or suppliers. Apart from inverse relationships and path navigation, the banking application offers a lot of opportunity for inheritance, since products usually form hierarchies and so do customers or contracts. Further, this application provides many opportunities for identifying useful semantic knowledge. For example, in order to estimate whether a current customer is likely to repay his/her loan, it is sensible to look at the behaviour of previous customers with similar characteristics. This entails queries that range over large amounts of data relating the propensity of arrears of customers based on their loan application data. Identifying common patterns of customer behaviour, in the form of association rules, would be of great help to the execution of similar queries.

Moving to the scientific world, medical applications are representative examples that could benefit from the proposed methods. Given a database of patient histories (symptoms, diagnoses and treatments), interesting rules can be deduced from the association of exhibited symptoms and prescribed treatments with the evolution of a patient's condition. In order to reach slightly refined or more general conclusions, a doctor might want to address queries that are likely to be optimized by the application of the derived rules. It is worth representing the medical data in an object-oriented way for two reasons: first, symptoms, diseases and treatments have many attributes and two-way relationships with each other; doctors are particularly interested in these relationships and often write queries that navigate across them. In relational databases, this would require many expensive join operations, whilst in their object counterparts, it would involve pointer traversals. Second, the classification of symptoms or diseases under more general categories (using the inheritance mechanism of the object model) might enable high level correlations among medical data to be identified. This would enable the optimization of a wider variety of queries.

In general, an optimization is considered beneficial when the time saved by executing the optimized query is greater than the cost of applying the optimization. However, in applications where queries are executed frequently, it is possible to apply the optimization once at compile time and execute the optimized query several times without any further overhead. The semantic optimization methods proposed in this dissertation obey this general rule, thus favouring applications that apply a particular query repeatedly. An important consideration is the dynamics of the database; we should be able to monitor the validity of prespecified semantic transformations in the presence of database updates.

2.2 Previous work related to this research

As shown in the previous section, there are large families of commercial (e.g. banking) and scientific (e.g. medical) applications suitable for applying semantic optimization techniques. This fact has motivated a lot of research in the area of query processing and optimization that has been used as the basis of the current work. The remainder of this chapter outlines previous research efforts from the areas involved in this work: type theory, intermediate query representation, data mining and semantic optimization.

2.2.1 Ensuring the typability of an OQL query - Related Work

The first step towards processing an OQL query is checking its typability, i.e. ensuring that it can be executed without causing any compile-time type errors. This can be done either with respect to a specific schema (type checking) or just based on the structure of the query independent of any schema information (type inference). Both problems have been studied broadly for various languages during the last 30 years [BO96, Dam85, Gun92, Mil78, Sul00, Tof87]. Typability issues have been considered for query languages in both relational and object oriented frameworks.

A discussion about type-checking an OQL query can be found in the ODMG standard [CBB+00]. Unfortunately, this presentation of the type system is neither formal nor complete. There is obviously a need for a more formal, mathematical presentation; the informal English presentation may be eloquent but it leaves a lot of space for ambiguity and misinterpretation.

Riedel and Scholl set out to describe the type system underlying the ODMG object model formally [RS97]. However, their approach, reflected in the CROQUE-version of ODL, differs considerably from the type model as defined in the ODMG standard [CBB+00]. They propose a type-based object model, where object types need not be named, since they are distinguished uniquely by their set of functions (attributes, relationships and methods). Classes are not considered as the types of object instances, but rather as extents. Hence, they are treated in queries differently from the way that they are treated in programming languages. Although Riedel and Scholl define types and subtyping formally, they provide no formal means of checking the type of an OQL query, e.g. a complete set of type rules. Finally, they study a much earlier version of the Standard (version 1.2) so some of their work is no longer relevant.

Following this work, Alagić [Ala99] and Bierman and Trigoni [BT00] made two independent efforts towards checking the type of an OQL query. Alagić considers a formalisation of the OQL type system, and claims that 'several negative results are proved about the ability to type-check queries'. This claim is later questioned by the work of Bierman and Trigoni who give a set of typing rules adequate for checking the type of all OQL constructs. Alagić also demonstrates a much more serious problem with the Java language binding. He gives a type rule [[Ala99], Rule 26], which appears to

be a rule for typing Java directly, i.e. he considers OQL as a language extension of Java. Clearly there is a problem here as OQL has parameterised types (e.g. bag(Employee)), but Java does not (all it has is covariant arrays).

Using the host language type system to type queries has been the dream of OODBMS designers [ZM90]. However, nowhere in the Standard does the ODMG propose that the Java type system be used to type OQL. The type system of the Object Model should be the basis for OQL type checking. The Standard gives a number of language bindings that enable the user to invoke OQL queries within a programming language environment. It is the role of these bindings to provide the mapping between the programming language type system and that of the Object Model.

Bierman and Trigoni address the typing problem from a lower level [BT00]. They give a detailed account of the typing rules needed to check the type of all OQL constructs, and not only of the select queries (as is the case in Alagić's work). Hence they provide rules that deal with $\mathrm{grouping}$ and various collection operations (e.g. $\mathrm{union}, \mathrm{intersect}, \mathrm{except}$).

Regarding the order operation, the Standard does not specify what happens when the sort criterion in the order clause does not distinguish completely between results. Presumably the order of elements in the resulting list is system-dependent. To alleviate this problem, Alagić enforces the limitation that the ordered elements should implement an interface with an ordering method. Bierman and Trigoni suggest that the resultant type of a $\mathrm{select} \cdot \ldots \cdot \mathrm{group\ by}$ clause should be a list of bags of elements rather than a list of elements. Each bag should include all the elements with the same values for the sort criteria. If the sort criteria define an absolute order for the elements, then all the bags are singletons. An additional $\mathrm{order\_flatten}$ operator could be potentially added as part of the OQL language in order to flatten the list of bags of elements into a list of elements in a system-specific way.

As far as type inference is concerned, the canonical reference for work on type systems for database programming languages is the work of Buneman and Ohori [BO96]. The goals of their type inference algorithm are identical to those when inferring the type of OQL queries, as presented in the next chapter; both approaches infer the most general type of an expression (if one exists) without accessing any schema information, and in this sense determine the constraints placed on the schema by the query. However, the underlying languages, the type systems, and some parts of the inference algorithms differ to a considerable extent. Buneman and Ohori introduce kinded types to infer the type of a record based on selections of fields on this record. Instead the type inference work in this dissertation uses the notion of *general* types; in this way, it is possible to express general type information not only for records, but also for parametric collection types, dictionary types, structures and classes. Moreover, a wider variety of constraints are introduced and hence a different algorithm is proposed to resolve them.

The work most related to the object nature of OQL arises from studying type systems for object

oriented programming languages, e.g. [PS91, PC94, AH95]. Palsberg and Schwartzbach [PS91] present an algorithm that infers the type of an untyped object-oriented language resembling Smalltalk. They define *type* as a finite set of classes, and *induced type* as the set of classes of all possible non-nil values to which a concrete program may evaluate in any of its executions. There are finitely many classes in a program and a type of an expression refers to the set of possible classes the expression is an instance of. However, inferring the type of an OQL query without accessing a specific schema raises the need to deal with *class templates* rather than with specific classes. Moreover, the OQL type system includes several atomic, collection and structure types. Thus, a general type cannot be defined as a set of classes.

Gunter [Gun92] studies polymorphism and type inference for a calculus language called $\mathrm{ML\_0}$. He presents an algorithm W that infers the most general type of an $\mathrm{ML\_0}$ expression. The main point where his approach differs from that of the current work is the unification mechanism. He uses substitution as the mechanism to unify two types that should be equal. However, in the OQL setting, which supports subtyping, the need to unify two types is not simply equivalent to the need for their equality. In certain cases, it concerns comparing two types using subtype relations or their ability to be element types in the same collection. Thus, the notion of substitution is not sufficient in resolving constraints amongst OQL types.

The language Cardelli [Car87] considers for type inference is also a very simple lambda calculus with constants. He presents inference rules that are applied from left to right and bottom-up. Starting with initially broad assumptions, he gradually refines them using inference rules. In case an inference rule requires two types to be equal, he unifies them and in this way he specializes some of the types of the identifiers. The approach used in chapter 3 is very similar, except that the inference rules generate two categories of constraint. The first consists of constraints that relate the types of two different expressions that coexist in the same *context* (query construct). The second concerns constraints that relate different information about the type of the same expression. This information might have been inferred by analyzing different *contexts* (parts of the query).

2.2.2 Intermediate representations of OQL - Related Work

The next step after ensuring the typability of an OQL query is to transform it into its internal representations, i.e. the calculus and the algebraic form.

Fegaras and Maier present a standard of effectiveness for a database calculus, before proposing monoid calculus as an intermediate representation for OQL queries [Feg94, Feg97, Feg98, FM95, FM00]. They prove that monoid comprehensions conform to this standard by being expressive enough to represent OQL's constructs, manipulable (w.r.t. type checking, rewriting et cetera), evaluable (giving opportunities for alternative execution plans) and uniform in handling collection and primitive types (as monoids). Provided that certain types are viewed as monoids, a query can also

be considered as a map from a given monoid to another monoid. These maps are known as monoid homomorphisms. Thus, a framework is proposed in which OQL queries are translated to monoid comprehensions, which in turn are defined in terms of monoid homomorphisms. The calculus representation is eventually translated into a series of algebraic operators, like `join`, `select`, `unnest` and `reduce`.

Grust and Scholl also support the concept of monoid comprehension as a calculus representation [GS98a]. Like Fegaras and Maier, they also underline that monoid calculus expressions are amenable to normalization. The existence of normal forms constitutes a major advantage for two reasons. First it enables *pattern-matching*; secondly, it reduces the number of transformation rules needed to deal with the repertoire of OQL operators. In spite of that, normalisation should be treated carefully, because it can sometimes hide optimization opportunities [GS96]. Grust and Scholl map monoid comprehensions to a combinator algebra (including map, filter, cross, join, semijoin, antijoin and aggregation operators). The hybrid representation of monoid comprehensions and algebraic combinators is treated as pure syntactic sugar for a single basic recursion combinator `foldr`. The latter is used in combination with `build` in order to enable the derivation of streaming programs (cheap deforestation) [GS98b].

A hybrid representation of monoid comprehensions and a monoid-aware nested algebra (e.g. semijoins, nestjoins) is discussed in [GKG$^+$97]. The core of the adopted algebra is the operator $\texttt{fold}[\oplus;\texttt{f}](\texttt{E})$, which applies `f` to the elements of `E`. It then uses the merge operation $\oplus$ of some monoid `M` to collect the result. The algebraic operator `fold` along with certain variations like `crossfold` (for the evaluation of joins) are also supported in [Van92].

Buneman et al. propose a comprehension syntax as the starting point for the development of database languages [BLS$^+$94]. They claim that comprehensions provide a natural way of expressing structural recursion and a powerful paradigm for manipulating collection types.

Straube et al. also present a framework for the translation of an object query language to calculus and algebraic representations [SO95a]. A query in their object calculus is of the form $\{\texttt{o} \mid \psi(\texttt{o})\}$ where `o` is an object variable and $\psi(\texttt{o})$ is a formula built from range or predicate atoms. They discuss safety criteria for the computability of queries and they give an algorithm for the translation of safe calculus expressions to algebra operator trees. Finally, they present equivalence-preserving rewrite rules in order to optimize algebraic expressions. The expressiveness and the type system of the query language considered in their context are less complex than those of OQL.

A sensible question that might be raised at this point is the following. Why are the algebraic representations of extended relational and object-oriented query languages (`fold`, algebraic combinators and `foldr`, monoid homomorphisms) not as uniform as those concerning the respective calculus representation (monoid comprehensions)? It is tempting to highlight that the main difference of the algebraic approaches lies in the concept of structural recursion. On the one hand, `foldr` directly

corresponds to structural recursion on the insert representation (sri). It is, therefore, naturally associated with sequential processing of collections. On the other hand, `fold` and monoid homomorphisms express structural recursion on the union representation (sru) and are consequently associated with the parallel processing of collections.

Opting for translating comprehensions to monoid homomorphisms has the advantage that monoid homomorphisms have the potential for parallel processing via divide-and-conquer methods [Afs98, HIT97]. This option has already been proved successful for relational query languages in [Afs98]. The query evaluation framework - developed as a part of the architecture - makes use of monoid comprehensions which are first translated to monoid homomorphisms and then (after several rewriting steps) to algorithmic skeletons, e.g. hom, split, pipe and farm. The latter are constructs with well-defined parallel behaviour and, in particular, the hom skeleton models the behaviour of the monoid homomorphism and exhibits its potential for parallelism.

However, the use of monoid homomorphisms would prevent us from efficiently evaluating execution plans using valuable experience from the relational technology. Reordering join operations, selecting different join algorithms, applying various (index-based) access methods and achieving semantic optimization are some of the optimizations that are hard to apply using the homomorphic representation.

Hence, queries in this dissertation are first translated to the monoid calculus and then to a set of algebraic operators that also provide the potential for parallel processing. Although this approach is similar to the one taken by Fegaras and Maier, it departs from theirs in several respects.

First, the normalization rules within the calculus representation have been extended. Secondly, the semantics of the algebraic operators have been altered to better suit the needs of query representation and optimization. Thirdly, the rules for the translation of the calculus to the algebraic formalism differ to a considerable extent. The calculus and the algebraic representations as well as the transformations performed within each level are described in chapters 4 and 5.

2.2.3 Association Rules and Semantic Optimization - Related Work

Within the intermediate query representations, there are a lot of opportunities for syntactic calculus or algebraic transformations. A lot of experience has been gained applying these optimizations in the relational or object-oriented context [CS96, Cha98, GLR97, Hel98, Ioa96, KPH98, PS96, PS97, RS93, SdBB96, SO95a, SO95b, WM99]. This thesis focuses on semantic query optimization, i.e. query transformations based on semantic knowledge rather than syntactic equivalence. In particular, semantic knowledge is represented by association rules of the form:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow p_0 \ (\mathtt{Exc})$$

where $\mathtt{p_i}, \mathtt{i} = 0, \ldots, \mathtt{n}$ have the form `class_name.attr_name op value` and $\mathtt{op} \in \{=, <, \leq, >, \geq\}$. `Exc` is the set of exceptions associated with the rule; it includes all the objects of class `class_name` that satisfy the antecedent $\mathtt{p_1} \wedge \ldots \wedge \mathtt{p_n}$, without satisfying the consequent $\mathtt{p_0}$. Such an association rule is often characterized by two measures of interest, confidence and support; we redefine these measures to fit the object-oriented paradigm. Confidence is the fraction of objects of class `class_name` satisfying the antecedent, that also satisfy the consequent. Support is the percentage of objects in the extent of `class_name` satisfying both the antecedent and the consequent.

A lot of research has been devoted to mining association rules that relate values of attributes in a relational table. The same techniques can equally be used in an object-oriented framework. Agrawal et al [AIS93] present an algorithm for identifying boolean association rules, i.e. correlations between boolean attributes. Further, Srikant and Agrawal presented techniques for discovering rules over quantitative and categorical attributes [SA96]. Later on, Miller and Yang studied the problem of mining association rules over interval data [MY97] using the Birch clustering algorithm [ZRL96]. A lot of attempts have been made to improve or to alter slightly the basic algorithm underlying the approaches discussed above [ASY98, CA97, FMMT96, KMR$^+$94, NLH98, Par95, TUA$^+$98, WYY00]. Chapter 7 presents an experimental framework intended to demonstrate the benefits of semantic optimization using association rules. The programs developed to identify these rules are based on standard algorithms for identifying quantitative rules [SA96] and rules over interval data [MY97].

Semantic knowledge that has been discovered using data mining techniques can be used for query optimization [HHCF96, Hsu96, HK96, Hsu97, HK98, HK00, Kin81b, LS95, MC96, SHKC93, YSP97, YS89]. A number of heuristics have been proposed that take advantage of association rules in order to optimize queries semantically [Bel96, CGM90, GGMR97, GGZ01, Kin81a, SSS92]. We refer briefly to the heuristics presented by Siegel et al [SSS92].

- *If a relation R in the query has a restricted attribute A and an unrestricted cluster index attribute B, then look for a rule where the restriction on A implies a restriction on B.* This heuristic implies the introduction of the second constraint, which is indexed.

- *If the only restriction on relation R is a join condition and R does not have an index on the join attributes, then look for a selection restriction on this relation. The new selection condition should be deduced from the selection conditions on the joining relation.*

- *If a selection condition on attribute A is implied by another selection condition on attribute B and A is not an index attribute, then the selection on A can be removed from the query.*

- *If a query contains a restriction on a range attribute, then try to locate a constraint on that attribute that is a contradiction to the existing restriction.*

- *If a restriction on a range attribute implies the answer to a query then the latter can be answered without accessing the database*

All of these heuristics make use of association rules to add or remove constraints, or to find contradictions or tautologies. In this dissertation we focus on the application of the first and third conditions, although the algorithms proposed can be adapted to serve all of the heuristics.

Chakravarthy et al. [CGM90, CGM88] also study the use of semantic knowledge for the transformation of queries into more efficient forms. They use a set of heuristics similar to those described above and propose techniques that take advantage of integrity constraints in the form of rules. Bell [Bel96] has studied the use of constraints in order to identify whether the result of a query contains distinct values. This knowledge is used in order to optimize `select` · `distinct` ... queries or `group by` operations.

Beneventano et al. [BBSV97] have developed a semantic query optimizer as part of a broader set of object-oriented tools (ODB-tools). Their optimizer takes advantage of the constraint introduction and constraint elimination heuristics, as well as of additional ones, e.g. reducing the scope of a query to a smaller subclass. These heuristics are presented in detail by Grant et al. [GGMR97].

Cheng et al. describe the implementation of two semantic optimization techniques, namely predicate introduction and join elimination, in DB2 Universal Database [CGK$^+$99, GLQ99]. After an input query is converted to an intermediate form called *query graph model*, it is transformed by a query rewrite engine [PHH92, PLH97] into a more efficient form using a set of integrity constraints.

Most of the approaches above ignore weak association rules, i.e. rules that are not a hundred per cent valid. In most cases rules represent integrity constraints, allowing no updates that invalidate them. Godfrey et al [GGZ01] consider the use of *absolute soft constraints* the exceptions of which are stored in materialized views. However, they do not consider combining soft constraints (and their exceptions) to infer new constraints. In general, previous approaches take advantage only of direct correlations as expressed by single rules. They do not combine rules to identify indirect associations that might be of use to the semantic optimization process.

In this dissertation, a series of algorithms is given that deals with these two issues. First, these algorithms take into account the exceptions associated with each rule, so that the result of the optimized query is accurate. Secondly, they combine existing association rules to identify useful indirect correlations along with their exceptions. Both direct and indirect correlations are combined to a series of optimization solutions; the criteria according to which the best solution is selected depend on the optimization heuristic.

2.3 Extending the previous work - Discussion

This chapter provided an overview of the applications and the previous work relevant to this research. The variety of applications is definitely a strong motivation for extending the existing work to address the problem of query processing and semantic optimization in an object-oriented framework. Although a lot of research has already been done in these areas, little work has focused on integrating the various stages of query processing, namely type inference, translation to calculus/algebraic representation and semantic optimization using association rules. Very few attempts have actually tried to measure experimentally the benefits of semantic optimization with respect to the optimization cost. To our knowledge there has been no implementation of the semantic optimization techniques that take account of indirect association rules with exceptions.

Chapter 3

Type Inference of OQL

This chapter presents an inference algorithm for OQL which both identifies the most general type of a query in the absence of schema type information, and derives the minimum type requirements a schema should satisfy to be compatible with this query. It is the result of joint work with Dr G.M. Bierman, presented in the proceedings of BNCOD 2001 [TB01]. The proposed algorithm is useful in any database application where heterogeneity is encountered, for example, schema evolution, queries addressed against multiple schemata, inter-operation or reconciliation of heterogeneous schemata. The inference algorithm is technically interesting as it concerns an object functional language with a rich semantics and complex type system. More precisely, we have devised a set of constraints and an algorithm to resolve them. The resulting type inference system for OQL should be useful in any open distributed, or even semi-structured, database environment.

3.1 Introduction

The ODMG Standard [CBB+00] presents, rather informally, some details of a type system for checking OQL queries using type information about the classes, extents, named objects and query definitions from a given database schema. Recently there have been some efforts to formalise this type system [Ala99, BT00]. The work presented in this chapter builds on earlier work [BT00] and considers the problem of inferring the most general type of an OQL query in the absence of any schema information.

For example, consider the following OQL definition and query:

```
define Dept_Managers(dept) as
  select e
  from   Employees as e
  where  e.position="manager" and e.department=dept;
```

```
select d
from   Departments as d
where  count(Dept_Managers(d))>5
```

This query yields those departments that have more than five managers. It is interesting to notice that this information could be drawn by running the query against databases with significantly different schemata. For instance, consider schema A, which has two classes, `Employee` and `Department`, defined as follows.

```
class Employee (extent Employees)
{ attribute string     name;
  attribute string     position;
  attribute int        year_of_birth;
  attribute float      salary;
  attribute Department department;}

class Department (extent Departments)
{ attribute string id;}
```

On the other hand, consider a second schema B, which has a class `Employee` and a named collection object `Departments` of type `List(int)`.

```
class Employee (extent Employees)
{ attribute string name;
  attribute string position;
  attribute int    department;}
```

The query could potentially run against both A and B without causing any type errors. In the case of schema A, the result of the query would be a bag of `Department` objects. In a database with schema B, the result of the query would be a bag of integers. Two vital questions arise at this point. First, how we can draw limits, or put restrictions, on the properties of a schema, so that a certain query is well-typed with respect to it? Second, what information can we derive about the type of the result of the query, supposing that we have no specific schema in mind? Before addressing these questions in detail, we consider the setting where type inference could be important.

As one example, this information could be exploited in distributed database applications. Suppose time-critical queries are addressed against multiple schemata. If there are frequent updates on parts of these schemata, then many of the queries will inevitably fail to be executed. In order to avoid this situation, a user can register interest in specific updates of each schema –at least in those that would affect the critical queries– and resolve the type incompatibility at the time the queries get executed.

Type inference is equally useful in contexts where we need to achieve inter-operation between heterogeneous sources. There has been a lot of research on reconciling schemata with semantic heterogeneity [BHP94, Hul97]. One approach to this problem identifies the semantic inconsistencies of the ontologies in different domains and creates a global ontology that combines all of them. Another approach identifies the intersection of domains where the inconsistencies occur and tries to resolve them by introducing matching rules between them. In both cases, queries that are initially written to be executed on one domain need to be rephrased to fit the needs of more domains. Knowing the schema requirements of a query and the schema mappings to a (global or just different) ontology, the task of rephrasing queries becomes an automatic process. Suppose that a group of airline companies cooperate to create a single uniform system for booking tickets. In order to do that they define a global ontology that is very close to each of the distinct ontologies. Each query is initially phrased to conform to the global ontology and is then transformed to appropriate queries addressed to the individual schemata. The transformation is much easier to perform if besides the schema mappings (from one ontology to the other), we are aware of the query schema requirements. The latter effectively point out the exact mappings we need to use.

Finally, the work discussed in this chapter enables the use of *general* query definitions. Version 2.0 of the ODMG standard allowed the user to write query definitions without explicitly defining the types of their arguments. This feature was changed in version 3.0, presumably either to simplify the task of type-checking or to complete the latter as much as possible at compile time. However, the ability to define general queries, which possibly run against several schemata without any change, is lost. Our inference algorithm enables the use of query definitions with unspecified argument types. It can infer the most general types of these arguments in exactly the same way as it derives the schema requirements (class/extent information, named objects or query definitions).

3.2 Core OQL

As mentioned previously, the work in type inference is based on earlier work in checking the type of an OQL query wrt specific schema information [BT00]. This section gives an overview of the grammar, the types and the subtyping relation of a core OQL, i.e. a fragment of the language defined in the Standard, but which has the same expressive power [BT00].

An OQL **program** consists of a number (maybe zero) of named definitions followed by a query. The syntax for queries and definitions[1], as well as the OQL types are presented in figure 3.1. In what follows, $\mathrm{Col}(\sigma)$ denotes an arbitrary collection type (set, bag, list or array), with elements of type σ.

Implicit in the ODMG model is a notion of subtyping; the underlying idea is that σ is said to be a subtype of τ, if a value of type σ can be used in any context in which a value of type τ is expected.

[1] Naturally as we are interested in *inferring* types we drop the requirement that definition parameters be explicitly typed.

Queries $\mathtt{q}$ ::= $b \mid f \mid i \mid c \mid s$
| $\mathtt{x}$
| $\mathtt{bag}(\mathtt{q},\ldots,\mathtt{q}) \mid \mathtt{set}(\mathtt{q},\ldots,\mathtt{q}) \mid \mathtt{list}(\mathtt{q},\ldots,\mathtt{q}) \mid \mathtt{array}(\mathtt{q},\ldots,\mathtt{q})$
| $\mathtt{struct}(\ell\colon \mathtt{q},\ldots,\ell\colon \mathtt{q})$
| $\mathtt{C}(\ell\colon \mathtt{q},\ldots,\ell\colon \mathtt{q}) \mid \mathtt{q}.\ell \mid (\mathtt{C})\mathtt{q}$
| $\mathtt{q}[\mathtt{q}] \mid \mathtt{q}\ \mathtt{in}\ \mathtt{q} \mid \mathtt{q}() \mid \mathtt{q}(\mathtt{q},\ldots,\mathtt{q})$
| $\mathtt{forall}\ \mathtt{x}\ \mathtt{in}\ \mathtt{q}\colon \mathtt{q} \mid \mathtt{exists}\ \mathtt{x}\ \mathtt{in}\ \mathtt{q}\colon \mathtt{q}$
| $\mathtt{q}\ \mathit{binop}\ \mathtt{q} \mid \mathit{unop}(\mathtt{q})$
| $\mathtt{select}\ [\mathtt{distinct}]\ \mathtt{q}$
$\mathtt{from}\ (\mathtt{q}\ \mathtt{as}\ \mathtt{x},\cdots,\mathtt{q}\ \mathtt{as}\ \mathtt{x})$
$\mathtt{where}\ \mathtt{q}$
$[\mathtt{group\ by}\ (\ell\colon \mathtt{q},\cdots,\ell\colon \mathtt{q})]$
$[\mathtt{having}\ \mathtt{q}]$
$[\mathtt{order\ by}\ (\mathtt{q}\ \mathtt{asc}|\mathtt{desc},\cdots,\mathtt{q}\ \mathtt{asc}|\mathtt{desc})]$

Definitions d ::= $\mathtt{define}\ \mathtt{x}\ \mathtt{as}\ \mathtt{q}$
| $\mathtt{define}\ \mathtt{x}(\mathtt{x},\ldots,\mathtt{x})\ \mathtt{as}\ \mathtt{q}$

Here b, f, i, c, s range over booleans, floats, integers, characters and strings respectively, $\mathtt{x}$ is taken from a countable set of identifiers, ℓ is taken from a countable set of labels, and $\mathtt{C}$ ranges over a countable set of class names. We assume sets of unary and binary operators, ranged over by unop and binop respectively.

Types σ ::= $\mathtt{int} \mid \mathtt{float} \mid \mathtt{bool} \mid \mathtt{char} \mid \mathtt{string} \mid \mathtt{void}$
| $\sigma \times \cdots \times \sigma \to \sigma$
| $\mathtt{bag}(\sigma) \mid \mathtt{set}(\sigma) \mid \mathtt{list}(\sigma) \mid \mathtt{array}(\sigma)$
| $\mathtt{struct}(\ell\colon \sigma,\cdots,\ell\colon \sigma)$
| $\mathtt{C}$

We assume a distinguished class name `Object`.

Figure 3.1: Syntax and Types for Core OQL

Sub-typing

$$\frac{}{\mathtt{C} \leq \mathtt{Object}}\ \text{Top} \qquad \frac{\mathtt{C} \sqsubseteq \mathtt{C}'}{\mathtt{C} \leq \mathtt{C}'}\ \text{Sub-Class}$$

$$\frac{\sigma'_1 \leq \sigma_1 \cdots \sigma'_\mathtt{k} \leq \sigma_\mathtt{k} \qquad \tau \leq \tau'}{\sigma_1 \times \cdots \times \sigma_\mathtt{k} \to \tau \leq \sigma'_1 \times \cdots \times \sigma'_\mathtt{k} \to \tau'}\ \text{Sub-Fun} \qquad \frac{\sigma \leq \tau}{\mathrm{Col}(\sigma) \leq \mathrm{Col}(\tau)}\ \text{Sub-Coll}$$

$$\frac{\sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_\mathtt{k} \leq \tau_\mathtt{k}}{\mathtt{struct}(\ell_1 \colon \sigma_1, \ldots, \ell_\mathtt{k} \colon \sigma_\mathtt{k}, \ldots, \ell_{\mathtt{k}+\mathtt{n}} \colon \sigma_{\mathtt{k}+\mathtt{n}}) \leq \mathtt{struct}(\ell_1 \colon \tau_1, \ldots, \ell_\mathtt{k} \colon \tau_\mathtt{k})}\ \text{Sub-Struct}$$

$$\frac{}{\sigma \leq \sigma}\ \text{Sub-Refl} \qquad \frac{\sigma \leq \sigma' \qquad \sigma' \leq \sigma''}{\sigma \leq \sigma''}\ \text{Sub-Trans}$$

Figure 3.2: Subtyping for core OQL

This we shall write $\sigma \leq \tau$ and define as the least relation closed under the rules given in Figure 3.2.

The $\sqsubseteq$ symbol denotes single inheritance between two classes, referred to in the Standard as the "*derives from*" relation. To simplify the presentation we do not consider interfaces.

An interesting feature of our subtype relation is the treatment of structures. A structure type $\sigma = \mathtt{struct}(\ell_1 \colon \sigma_1, \ldots, \ell_\mathtt{m} \colon \sigma_\mathtt{m})$ is considered to be a subtype of $\tau = \mathtt{struct}(\ell_1 \colon \tau_1, \ldots, \ell_\mathtt{n} \colon \tau_\mathtt{n})$ if τ is obtained from σ by dropping some labels. (In fact, we generalise this a little and also allow subtyping between the label types). This so-called width-subtyping is an extension to the Standard, but we feel it offers considerable flexibility.

The type system and the subtype relation are given in detail in an earlier paper [BT00]. In that work, we aimed at deriving the type of an OQL query given specific schema information. In order to do that, we defined typing judgements of the form:

$$\mathcal{S}; \mathcal{D}; \mathcal{N}; \mathcal{Q} \vdash \mathtt{q} \colon \sigma$$

where $\mathcal{S}$ are the class definitions, $\mathcal{D}$ are the persistent query definitions and $\mathcal{N}$ are the named objects of a specific schema. $\mathcal{Q}$ represents the query typing environment, i.e. it contains the types of any free identifiers in $\mathtt{q}$. A simple example of the typing rules used to derive the type of a query is the following:

$$\frac{\mathcal{S}; \mathcal{D}; \mathcal{N}; \mathcal{Q} \vdash \mathtt{q} \colon \mathtt{list}(\sigma)}{\mathcal{S}; \mathcal{D}; \mathcal{N}; \mathcal{Q} \vdash \mathtt{first}(\mathtt{q}) \colon \sigma}\ \text{First-list}$$

In the context of inferring the type of an OQL query, we have no information about the classes, the query definitions, and the named objects, as we have no schema information. The problem addressed

in this chapter can thus be written $? \vdash \texttt{q}: ?$, i.e. given an arbitrary query q, can we infer its type and the type of any supporting schemata?

3.3 Type and Schema Inference

This section presents the extended type system behind our inference algorithm and a new relation between types called *Generalisation-Specialisation* relation. We also discuss the notions of *Least Upper Bound* and *Greatest Lower Bound* of two types that occur frequently in our inference algorithm.

3.3.1 Extended Type System

It turns out to be convenient to extend the notion of type given in Figure 3.1; an example should make this clear. Consider the following:

```
define q1 as select x from Students as x;
define q2 as set(first(Students));
q1 union q2
```

Considering the query `q1 union q2` first, we can infer immediately that `q1` and `q2` should be either sets or bags of elements. To represent this we introduce a new type constructor $\texttt{set/bag}(-)$. Moreover, the elements of this collection cannot be of a *function* type, since the Standard does not allow functions to be members of a collection; thus we introduce the types `nonfunctional` and `function`. From the definition `q1` we infer that `Students` is some collection (set, bag, list or array) of elements of any (non-functional) type. Considering the definition `q2` we can infer further information about `Students`. As it is the argument of a `first` operation it must be an *ordered* collection, i.e. a list or an array. Again we introduce a new type constructor $\texttt{list/array}(-)$. In summary, the algorithm should infer that `Students` is a list or an array of a nonfunctional member type τ, and that the query `q1 union q2` is of type $\texttt{bag}(\tau)$.[2]

The above example motivates our need to extend the initial *specific* types (given in Figure 3.1) with the following so-called *general* types.

$$
\begin{aligned}
&\{\texttt{any}, \texttt{nonfunctional}, \texttt{atomic}, \texttt{orderable}, \texttt{int/float}\} \cup \\
&\{\texttt{collection}(\tau), \texttt{set/bag}(\tau), \texttt{list/array}(\tau), \texttt{constructor}(\ell_i : \tau_i)\} \cup \\
&\{\texttt{all types from the core type system with at} \\
&\texttt{least one component type being a general type, e.g. set(any)}\}
\end{aligned}
$$

where τ, τ_i are *specific* or *general* types.

[2] The Standard [§4.10.11] states that merging a set and a bag results in a bag.

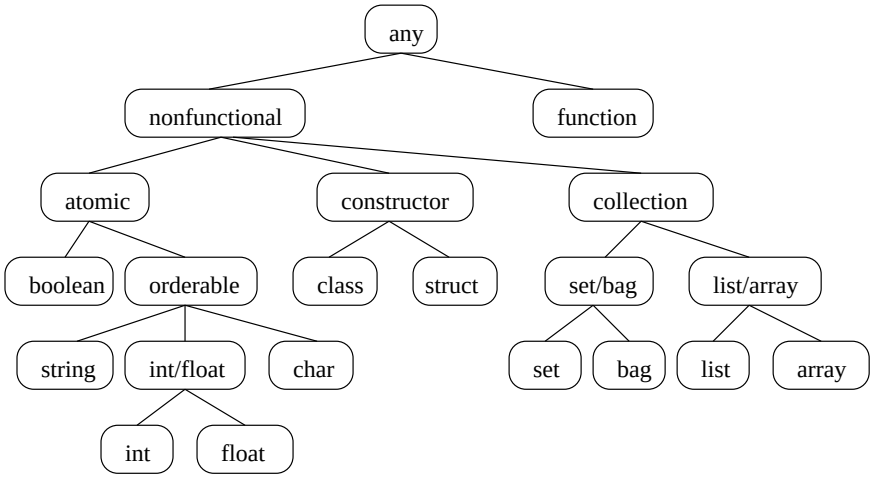

Figure 3.3: Type Hierarchy

Given these general types the resulting type system is then as follows:

$$
\begin{array}{rcl}
\sigma & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{string} \mid \texttt{void} \\
& \mid & \sigma \times \cdots \times \sigma \to \sigma \\
& \mid & \texttt{bag}(\sigma) \mid \texttt{set}(\sigma) \mid \texttt{list}(\sigma) \mid \texttt{array}(\sigma) \mid \texttt{struct}(\ell\colon \sigma, \cdots, \ell\colon \sigma) \\
& \mid & \texttt{C} \\
& \mid & \texttt{any} \mid \texttt{nonfunctional} \\
& \mid & \texttt{atomic} \mid \texttt{orderable} \mid \texttt{int/float} \\
& \mid & \texttt{constructor}(\ell\colon \sigma, \cdots, \ell\colon \sigma) \\
& \mid & \texttt{collection}(\sigma) \mid \texttt{set/bag}(\sigma) \mid \texttt{list/array}(\sigma)
\end{array}
$$

This extended type system is coupled with the type hierarchy illustrated in Figure 3.3. It is worth noting that the *general* types, which are the internal nodes of the tree, are not types that can be found in a database schema, but rather abstractions or families of types that encapsulate the common features of their children.

A type is said to be *specific*, if it can be derived by the type system given in Figure 3.1. Otherwise, it is said to be *general*. All the leaves of the hierarchy tree (in Figure 3.3) are specific types, if they are nullary (non parametric) types (`int`, `float`, `char`, `string`, `bool`) or if they are parametric types with all the parameter types being specific types (e.g. $\texttt{set}(\texttt{int})$).

Given this more general type system, we need to extend our notion of subtyping given earlier. We define a new relation, GSR (Generalisation-Specialisation Relationship). Given types σ, τ, we write

$\sigma \subseteq \tau$ to express that σ is *more specific* than τ.

$$\frac{\sigma \leq \tau}{\sigma \subseteq \tau}\,\mathrm{GSR-Type} \qquad \frac{\sigma'_1 \subseteq \sigma_1 \cdots \sigma'_{\mathtt{k}} \subseteq \sigma_{\mathtt{k}} \;\; \tau \subseteq \tau'}{\sigma_1 \times \cdots \times \sigma_{\mathtt{k}} \to \tau \subseteq \sigma'_1 \times \cdots \times \sigma'_{\mathtt{k}} \to \tau'}\,\mathrm{GSR-Fun}$$

$$\frac{\mathtt{coll_1\ is\ child\ of\ coll_2} \;\; \sigma \subseteq \tau}{\mathtt{coll_1}(\sigma) \subseteq \mathtt{coll_2}(\tau)}\,\mathrm{GSR-Coll}$$

$\mathtt{coll_i}$, $\mathtt{i} = 1, 2$ are nodes in the subtree (figure 3.3) with root `collection` and `is child of` signifies that $\mathtt{coll_2}$ is a direct or indirect parent of $\mathtt{coll_1}$ or that $\mathtt{coll_1}$ and $\mathtt{coll_2}$ are the same.

$$\frac{\mathtt{constr_1\ is\ child\ of\ constr_2} \;\; \sigma_1 \subseteq \tau_1 \cdots \sigma_{\mathtt{k}} \subseteq \tau_{\mathtt{k}}}{\mathtt{constr_1}(\ell_1 \colon \sigma_1, \ldots, \ell_{\mathtt{k}} \colon \sigma_{\mathtt{k}}, \ldots, \ell_{\mathtt{k+n}} \colon \sigma_{\mathtt{k+n}}) \subseteq \mathtt{constr_2}(\ell_1 \colon \tau_1, \ldots, \ell_{\mathtt{k}} \colon \tau_{\mathtt{k}})}\,\mathrm{GSR-Constr}$$

$\mathtt{constr_i}$, $\mathtt{i} = 1, 2$ are nodes in the subtree (figure 3.3) with root `constructor` and `is child of` signifies that $\mathtt{constr_2}$ is a direct parent of $\mathtt{constr_1}$ or $\mathtt{constr_1}$ and $\mathtt{constr_2}$ are the same.

$$\frac{\mathtt{atom_1\ is\ child\ of\ atom_2}}{\mathtt{atom_1} \subseteq \mathtt{atom_2}}\,\mathrm{GSR-Atomic}$$

$\mathtt{atom_i}$, $\mathtt{i} = 1, 2$ are nodes in the subtree (figure 3.3) with root `atomic` and `is child of` signifies that $\mathtt{atom_2}$ is direct or indirect parent of $\mathtt{atom_1}$ or $\mathtt{atom_1}$ and $\mathtt{atom_2}$ are the same.

$$\frac{}{\sigma \subseteq \sigma}\,\mathrm{GSR-Refl} \qquad \frac{\sigma_1 \subseteq \sigma_2 \;\; \sigma_2 \subseteq \sigma_3}{\sigma_1 \subseteq \sigma_3}\,\mathrm{GSR-Trans}$$

$$\frac{\sigma \neq \tau_1 \to \tau_2}{\sigma \subseteq \mathtt{nonfunctional}}\,\mathrm{GSR-NonFun} \qquad \frac{}{\sigma \subseteq \mathtt{any}}\,\mathrm{GSR-All}$$

Given this definition, we can define the *Greatest Lower Bound* (`GLB`) and the *Least Upper Bound* (`LUB`) of two types τ_1 and τ_2. Insight into these concepts can be gained through the following simple example. Consider the types $\tau_1 = \mathtt{set(atomic)}$ and $\tau_2 = \mathtt{set/bag(int)}$. The `GLB` of the two types is derived by taking the most specific of the collection constructors, `set`, and the most specific of the parameter types, `int`. Thus $\mathtt{GLB}(\tau_1, \tau_2) = \mathtt{set(int)}$. Likewise, for the `LUB`, we take the most general of the two collection constructors, `set/bag`, and the most general of the two parameter types, `atomic`. Thus, $\mathtt{LUB}(\tau_1, \tau_2) = \mathtt{set/bag(atomic)}$.

We may now formally present `GLB` and `LUB`. In the following definitions, we assume that $\mathtt{constr_1}$, $\mathtt{constr_2}$ are nodes in the subtree (figure 3.3) with root `constructor` and that $\mathtt{constr_1}$ is a child of $\mathtt{constr_2}$ or $\mathtt{constr_1} = \mathtt{constr_2}$. $\mathtt{coll_1}$ and $\mathtt{coll_2}$ are nodes in the subtree (figure 3.3) with root `collection` and $\mathtt{coll_1}$ is a child of $\mathtt{coll_2}$ or $\mathtt{coll_1} = \mathtt{coll_2}$. Although `LUB` is always defined, this is not the case for `GLB`. There is not always a type that is more specific than each possible pair of OQL types. For instance, it is not possible to define the `GLB` of a class and an atomic type.

$$\texttt{GLB}(\tau_1, \tau_2) \stackrel{\text{def}}{=} \tau_1 \texttt{ if } \tau_1 \subseteq \tau_2 \wedge \tau_2 \subseteq \texttt{atomic}$$

$$\texttt{LUB}(\tau_1, \tau_2) \stackrel{\text{def}}{=} \tau \text{ if } \tau_1 \subseteq \texttt{atomic} \wedge \tau_2 \subseteq \texttt{atomic} \wedge$$
$$(\text{there exists no } \tau' \text{ s.t. } (\tau' \neq \tau \wedge \tau_1 \subseteq \tau' \wedge \tau_2 \subseteq \tau' \wedge \tau' \subseteq \tau))$$

$$\texttt{GLB}(\texttt{constr}_1(\ell_1\colon \sigma_1, \ldots, \ell_\texttt{k}\colon \sigma_\texttt{k}), \texttt{constr}_2(\ell_1\colon \sigma'_1, \ldots, \ell_\texttt{k}\colon \sigma'_\texttt{k}, \ldots, \ell_{\texttt{k+n}}\colon \sigma'_{\texttt{k+n}}))$$
$$\stackrel{\text{def}}{=} \texttt{constr}_1(\ell_1\colon \texttt{GLB}(\sigma_1, \sigma'_1), \ldots, \ell_\texttt{k}\colon \texttt{GLB}(\sigma_\texttt{k}, \sigma'_\texttt{k}), \ldots, \ell_{\texttt{k+n}}\colon \sigma'_{\texttt{k+n}})$$

$$\texttt{LUB}(\texttt{constr}_1(\ell_1\colon \sigma_1, \ldots, \ell_\texttt{k}\colon \sigma_\texttt{k}), \texttt{constr}_2(\ell_1\colon \sigma'_1, \ldots, \ell_\texttt{k}\colon \sigma'_\texttt{k}, \ldots, \ell_{\texttt{k+n}}\colon \sigma'_{\texttt{k+n}}))$$
$$\stackrel{\text{def}}{=} \texttt{constr}_2(\ell_1\colon \texttt{LUB}(\sigma_1, \sigma'_1), \ldots, \ell_\texttt{k}\colon \texttt{LUB}(\sigma_\texttt{k}, \sigma'_\texttt{k}))$$

$$\texttt{GLB}(\texttt{coll}_1(\sigma_1), \texttt{coll}_2(\sigma_2)) \stackrel{\text{def}}{=} \texttt{coll}_1(\texttt{GLB}(\sigma_1, \sigma_2))$$

$$\texttt{LUB}(\texttt{coll}_1(\sigma_1), \texttt{coll}_2(\sigma_2)) \stackrel{\text{def}}{=} \texttt{coll}_2(\texttt{LUB}(\sigma_1, \sigma_2))$$

$$\texttt{GLB}(\tau_1 \times \cdots \times \tau_\texttt{k} \to \sigma, \tau'_1 \times \cdots \times \tau'_\texttt{k} \to \sigma') \stackrel{\text{def}}{=}$$
$$\texttt{LUB}(\tau_1, \tau'_1) \times \cdots \times \texttt{LUB}(\tau_\texttt{k}, \tau'_\texttt{k}) \to \texttt{GLB}(\sigma, \sigma')$$

$$\texttt{LUB}(\tau_1 \times \cdots \times \tau_\texttt{k} \to \sigma, \tau'_1 \times \cdots \times \tau'_\texttt{k} \to \sigma') \stackrel{\text{def}}{=}$$
$$\texttt{GLB}(\tau_1, \tau'_1) \times \cdots \times \texttt{GLB}(\tau_\texttt{k}, \tau'_\texttt{k}) \to \texttt{LUB}(\sigma, \sigma')$$

$$\texttt{LUB}(\sigma, \tau) \stackrel{\text{def}}{=} \texttt{any}, \texttt{ if } \sigma \subseteq \texttt{function} \wedge \tau \subseteq \texttt{nonfunctional}$$

$$\texttt{LUB}(\sigma_1, \sigma_2) \stackrel{\text{def}}{=} \texttt{nonfunctional}, \text{ if } \forall \tau_1, \tau_2 . \sigma_1 \subseteq \tau_1 \wedge \sigma_2 \subseteq \tau_2 \wedge \tau_1 \neq \tau_2 \wedge$$
$$\tau_1, \tau_2 \in \{\texttt{atomic}, \texttt{constructor}(), \texttt{collection}(\texttt{any})\}$$

3.4 Type compatibility - Constraints

The inference algorithm we present in section 3.7 analyses an OQL construct and infers the most general type of the query and the schema requirements that should be satisfied so that the query is well-typed. Before being able to present the inference algorithm, we first discuss an important mechanism that the algorithm is based upon—the generation of constraints. When the inference algorithm analyses a certain query construct, it often infers several relations or associations amongst the types of the query and its subqueries. These associations are given in the form of constraints.

For example, the analysis of a query $\texttt{q}_1$ `union` $\texttt{q}_2$ would generate the constraint that the type τ of the query is the merge result of the types τ_1 and τ_2 of the two subqueries ($\tau = \texttt{Merge\_Result}(\tau_1, \tau_2)$). In section 3.5, we show how this constraint is simplified and is *assimilated* in the set of the existing constraints.

Analysing the query `exists x in Customers: x.income` $> 40,000$ our algorithm generates the following constraints: first, it introduces the constraint $\tau_1 = \texttt{Member\_Type}(\tau_2)$, where τ_1 is the type of `x` and τ_2 is the type of `Customers`. The type of `x` should be the same as that of the members of the collection `Customers`. Second, the constraint $\tau_3 = \texttt{Constructor\_Member\_Type}(\tau_1, \texttt{income})$ is

generated, where τ_3 is the type of `x.income`. This signifies that `x` is a constructor type (a structure or a class) with at least one member `income` of type τ_3. Another interesting constraint is that the types of `x.income` (τ_3) and of the literal $40,000$ (`int`) should be compatible in the sense that they than can be compared for inequality (`Greater_Less_Than_Compatible`$(\tau_3, \texttt{int})$). Later, we will show how this constraint is simplified to the constraint $\tau_3 \subseteq$ `int/float`.

In section 3.6, we give a set of inference rules, one for each query construct. Each rule starts with an existing set of constraints and generates a number (possibly zero) of new constraints. The different kinds of constraints generated by the rules in our inference algorithm are given below:

1. `Equality_Compatible`$(\tau_1, \ldots, \tau_n)$	6. $\tau_0 =$ `Merge_Result`(τ_1, τ_2)
2. `Greater_Less_Than_Compatible`$(\tau_1, \ldots, \tau_n)$	7. $\tau_0 =$ `Distinct_Result`(τ_1)
3. $\tau =$ `Member_Type`(σ)	8. $\tau_1 = \tau_2$
4. $\tau =$ `Constructor_Member_Type`(σ, ℓ)	9. $\tau_1 \subseteq \tau_2$
5. $\tau_0 =$ `Arith_Result`(τ_1, τ_2)	

The constraint `Equality_Compatible`$(\tau_1, \ldots, \tau_n)$ is analysed in section 3.4.1. The second constraint `Greater_Less_Than_Compatible`$(\tau_1, \ldots, \tau_n)$ is useful for ensuring typability for queries like $q_1 < q_2$. $\tau =$ `Member_Type`(σ) implies that σ is a collection type and that τ is the type of its members. $\tau =$ `Constructor_Member_Type`(σ, ℓ) is used to denote that σ is a `class` or `struct` type with at least one member ℓ of type τ. If σ is a `class` type then ℓ can be any of its properties, relationships or methods. $\tau_0 =$ `Arith_Result`(τ_1, τ_2) is needed for the type inference of queries of the form $q_1 \, op \, q_2$ where $op \in \{+, -, /, *\}$. Likewise, $\tau_0 =$ `Merge_Result`(τ_1, τ_2) arises as a constraint from inferring the type of `union`, `intersect` or `except` expressions. $\tau_0 =$ `Distinct_Result`(τ_1) implies that both τ_0 and τ_1 are collection types and the collection constructor of τ_0 is the distinct equivalent of the collection constructor of τ_1. Finally, type equality and GSR (Generalisation-Specialisation Relationship) are handled by constraints 8 and 9.

3.4.1 Collections - Membership Type Compatibility

The first two constraints refer to type compatibility wrt equality or non-equality comparison. These constraints arise in OQL constructs that involve a merge of two or more elements, or a membership test. For instance, the first constraint results from considering a query of the form `set`$(q_1, \ldots, q_n)$, which involves the merge of n query results.

First of all, we should stress the fact that in order for two values (objects or literals) to be eligible as members of the same collection, they should be eligible for *equality comparison.* If two types are compatible (membership-wise), two values of these types may be members of a set. In order to insert an element into a set, we need to test if its value is equal to any existing value. Thus, we need to

ensure that these values have types which are compatible (equality-wise). Inversely, if two types are compatible equality-wise, then their values may be inserted into any collection, therefore these types are also compatible membership-wise.

The Standard [§4.10] defines recursively when two types are compatible, and thus when elements of these types can be put in the same collection. The Standard then defines the notion of least upper bound (LUB) of two types to derive the type of the collection elements. In a context where we need to check and derive the type of a query based on *specific* type information (from a schema), this approach is sufficient and straightforward. However, in our context, where we aim to infer the type of a query without any schema information, the compatibility issue becomes more complicated. The use of LUB to infer the type of a query like $\texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_\texttt{n})$ does not yield the appropriate result, for example consider the following.

```
define q1 as struct(x:12,y:30);
define q2 as element(select z from People as z where z.x=14);
set(q1,q2)
```

If we call the inference algorithm on `q1` and `q2` the inferred types would be $\texttt{struct}(\texttt{x} : \texttt{int}, \texttt{y} : \texttt{int})$ and $\texttt{constructor}(\texttt{x} : \texttt{int})$ respectively. Their least upper bound (LUB) is $\texttt{constructor}(\texttt{x} : \texttt{int})$. Thus, the inferred type of the query `set(q1,q2)` would be $\texttt{set}(\texttt{constructor}(\texttt{x} : \texttt{int}))$.

However, the correct inferred type should be $\texttt{set}(\texttt{struct}(\texttt{x} : \texttt{int}))$, since we may not merge objects and structures in the same collection, and therefore we know that the `constructor` should be a `struct` and not a `class`.

To overcome this problem we define another relation between types, namely CUB (*Compatible Upper Bound*). Intuitively, CUB combines the behaviour of both LUB and GLB (Greatest Lower Bound). In the previous example, the CUB of the two types $\texttt{IT}(\texttt{q}_1)$ and $\texttt{IT}(\texttt{q}_2)$ would be derived by taking the most specific of the two constructor types (`constructor` and `struct`), but the least general of the element types ($(\texttt{x} : \texttt{int})$ and $(\texttt{x} : \texttt{int}, \texttt{y} : \texttt{int})$). Before we define CUB, we define *compatibility* (Membership- or Equality- wise) for our typing system.

Compatibility is recursively defined as follows:

- τ is compatible with τ
- If $\texttt{coll}_1, \texttt{coll}_2 \in \{\texttt{collection}, \texttt{set/bag}, \texttt{list/array}, \texttt{set}, \texttt{bag}, \texttt{list}, \texttt{array}\}$ and either the collection constructors are the same or one is child of the other in the hierarchy tree and σ is compatible with τ then $\texttt{coll}_1(\sigma)$ is compatible with $\texttt{coll}_2(\tau)$.
- Any two class types $\texttt{class\_id}_1$ and $\texttt{class\_id}_2$ are compatible.
- If σ_i is compatible with $\tau_i, \forall\, i = 1, \ldots, n$ and $\texttt{constr}_1, \texttt{constr}_2 \in \{\texttt{constructor}, \texttt{struct}\}$ and no labels other than $\ell_1, \ldots, \ell_\texttt{n}$ are common in both constructor types then the constructor

type $\texttt{constr}_1(\ell_1\colon \sigma_1, \ldots, \ell_\texttt{n}\colon \sigma_\texttt{n}, \ell_{11}\colon \sigma_{11}, \ldots, \ell_{1\texttt{k}}\colon \sigma_{1\texttt{k}})$ is considered to be compatible with $\texttt{constr}_2(\ell_1\colon \tau_1, \ldots, \ell_\texttt{n}\colon \tau_\texttt{n}, \ell_{21}\colon \tau_{21}, \ldots, \ell_{2\texttt{m}}\colon \tau_{2\texttt{m}})$.

- If σ_i is compatible with τ_i, $\texttt{i} = 1, \ldots, \texttt{n}$ and $\texttt{Constructor\_Member\_Type}(\texttt{class\_id}, \ell_\texttt{i}) = \tau_\texttt{i}$, $\texttt{i} = 1, \ldots, \texttt{n}$, then class_id and $\texttt{constructor}(\ell_1\colon \sigma_1, \ldots, \ell_\texttt{n}\colon \sigma_\texttt{n}, \ell_{11}\colon \sigma_{11}, \ldots, \ell_{1\texttt{k}}\colon \sigma_{1\texttt{k}})$ are compatible types, provided that no labels other than $\ell_1, \ldots, \ell_\texttt{n}$ are common members in the two types.

- If $\sigma, \tau \in \{\texttt{atomic}, \texttt{orderable}, \texttt{int/float}, \texttt{int}, \texttt{float}, \texttt{char}, \texttt{string}, \texttt{bool}\}$ and either they are the same, or one is a child of the other in the hierarchy tree, or one is int and the other is float then σ is compatible with τ.

- If either of the types is nonfunctional and the other type is not a function type then these types are compatible.

- If either of two types is any, then these types are compatible.

Note that we do not define compatibility for function types, as no two function values may be members of the same collection or may be compared for equality. Only the results of function application may be considered for compatibility.

Given that two types are compatible (based on the recursive definition above), their CUB is defined recursively and in accordance with the compatibility category that they fall into.

- $\texttt{CUB}(\tau, \tau) = \tau$.

- If $\texttt{coll}_1(\sigma)$ is compatible with $\texttt{coll}_2(\tau)$ and $\texttt{coll}_1$ is a child of (or the same as) $\texttt{coll}_2$, then $\texttt{CUB}(\texttt{coll}_1(\sigma), \texttt{coll}_2(\tau)) = \texttt{coll}_1(\texttt{CUB}(\sigma, \tau))$.

- If the types τ_1 and τ_2 are class types, then $\texttt{CUB}(\tau_1, \tau_2)$ is the least common superclass of the two classes.

- If a constructor type $\sigma = \texttt{constr}_1(\ell_1\colon \sigma_1, \ldots, \ell_\texttt{n}\colon \sigma_\texttt{n}, \ell_{11}\colon \sigma_{11}, \ldots, \ell_{1\texttt{k}}\colon \sigma_{1\texttt{k}})$ is compatible with $\tau = \texttt{constr}_2(\ell_1\colon \tau_1, \ldots, \ell_\texttt{n}\colon \tau_\texttt{n}, \ell_{21}\colon \tau_{21}, \ldots, \ell_{2\texttt{m}}\colon \tau_{2\texttt{m}})$, where $\texttt{constr}_1, \texttt{constr}_2 \in \{\texttt{constructor}, \texttt{struct}\}$ and $\texttt{constr}_1$ is a child of (or the same as) $\texttt{constr}_2$ then $\texttt{CUB}(\sigma, \tau) = \texttt{constr}_1(\ell_1\colon \texttt{CUB}(\sigma_1, \tau_1), \ldots, \ell_\texttt{n}\colon \texttt{CUB}(\sigma_\texttt{n}, \tau_\texttt{n}))$.

- If $\sigma = \texttt{class\_id}_1$ and $\texttt{Constructor\_Member\_Type}(\texttt{class\_id}_1, \ell_\texttt{i}) = \sigma_\texttt{i}$, $\texttt{i} = 1, \ldots, \texttt{n}$ and $\tau = \texttt{constructor}(\ell_1\colon \tau_1, \ldots, \ell_\texttt{n}\colon \tau_\texttt{n}, \ell_{21}\colon \tau_{21}, \ldots, \ell_{2\texttt{m}}\colon \tau_{2\texttt{m}})$ then $\texttt{CUB}(\sigma, \tau)$ is the least superclass of $\texttt{class\_id}_1$, say $\texttt{class\_id}_2$, satisfying the following condition: For all $\ell'_\texttt{j}$, $\ell'_\texttt{j}$ is a property or a relationship of $\texttt{class\_id}_2$, if $\texttt{Constructor\_Member\_Type}(\texttt{class\_id}_2, \ell'_\texttt{j}) = \phi_\texttt{j}$ then there exists $\texttt{k}$, $1 \leq \texttt{k} \leq \texttt{n}$, s.t. $\ell'_\texttt{j} = \ell_\texttt{k} \wedge \texttt{CUP}(\tau_\texttt{k}, \sigma_\texttt{k}) \subseteq \phi_\texttt{j}$, $\forall\ \texttt{j} = 1, \ldots, \texttt{m}$, $\texttt{m} \leq \texttt{n}$.

- If σ is compatible with τ and $\sigma, \tau \subseteq \texttt{atomic}$ then

  1. if either of them is $\texttt{int/float}$ or one is $\texttt{int}$ and the other is $\texttt{float}$ then
     $\texttt{CUB}(\sigma, \tau) = \texttt{int/float}$
  2. else
     $\texttt{CUB}(\sigma, \tau) = \texttt{GLB}(\sigma, \tau)$.

- If $\sigma = \texttt{nonfunctional}$ and $\tau \subseteq \texttt{nonfunctional}$ then $\texttt{CUB}(\sigma, \tau) = \tau$.

- If $\sigma = \texttt{any}$ then, for any type τ, $\texttt{CUB}(\sigma, \tau) = \tau$.

As discussed earlier, the notion of CUB is used for the inference of the types of queries like $\texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_n)$. However, the OQL construct $\texttt{q}_1$ `in` $\texttt{q}_2$ raises another issue of a slighly different nature. The Standard [§4.10.8.3] states that if the type of $\texttt{q}_2$ is $\texttt{coll}(\tau)$ then the type of $\texttt{q}_1$ should be τ. We explain why this is not the case in our context. Suppose that the type of $\texttt{q}_2$ is inferred to be $\texttt{bag}(\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{string}))$; then according to the Standard $\texttt{q}_1$ should have the type $\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{string})$. Since a value of type $\texttt{struct}(\texttt{x}\colon \texttt{int})$ could potentially be added in the collection $\texttt{q}_2$ (that is, since $\texttt{struct}(\texttt{x}\colon \texttt{int})$ and $\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{float})$ are compatible types), there is no reason why $\texttt{q}_1$ could not be of type $\texttt{struct}(\texttt{x}\colon \texttt{int})$ or even $\texttt{struct}()$.

The same situation occurs when dealing with a collection of objects of different classes. Suppose $\texttt{lub\_class}(\ell_1\colon \sigma_1, \ldots, \ell_n\colon \sigma_n)$ is the LUB of all classes of the objects in the collection and `Object` is the most general class that all other classes derive from (the top of the class hierarchy). We should be able to check whether an object of type $\texttt{object\_class}(\ell'_1\colon \sigma'_1, \ldots, \ell'_m\colon \sigma'_m)$ is a member of the collection, even if its class is not a subclass of $\texttt{lub\_class}(\ell_1\colon \sigma_1, \ldots, \ell_n\colon \sigma_n)$. This allows more queries to (safely) type-check, for example:

```
select x
from   People as x
where  x.father in School_Teachers
```

This query does not type-check according to the Standard. In order to be type-correct, there should be an explicit type cast `(School_Teacher)x.father`. The problem is that this query, despite being well-typed, can generate a run-time error (if the cast does not succeed). We choose not to enforce that the type of `x.father` should be more specific than the member type or the collection `School_Teachers`. Instead we ensure that `x.father` could potentially be a member of `School_Teachers`, by adding the constraint $\texttt{Equality\_Compatible}(\sigma, \tau)$, where σ is the type of `x.father` and τ is the member type of `School_Teachers`.

3.5 Resolving constraints

Now that we have studied the kinds of constraints that are generated by our inference algorithm, we can discuss how these constraints are resolved. When a constraint is generated by an inference rule, it is added to the set of existing constraints. If this was a simple insertion procedure, we would end up having a huge set of constraints, that would include redundant and often incomprehensible type information; there is obviously a need to resolve the inserted constraints. Due to the complexity of the type system and the expressiveness of the language, we have a wide variety of constraints, that cannot be solved using a standard unification mechanism alone [Rob65]. In our system, the insertion of a new constraint in a set of existing constraints may have one of the following effects:

- A constraint is deleted, if it is always satisfied, e.g. the constraint
  $\texttt{Equality\_Compatible}(\texttt{set}(\texttt{int}), \texttt{set}(\texttt{float}))$
  is always true, so it does not need to be maintained.

- A constraint raises a type error or exception, if it is never satisfied, e.g.
  $\texttt{set}(\texttt{Employee}(\texttt{name} : \texttt{string})) \subseteq \texttt{list/array}(\texttt{Employee}(\texttt{name} : \texttt{string}))$.

- A constraint might be maintained as it is. This usually occurs when some of the types involved are *general*; it may be that when refined, these types no longer satisfy the constraint. Therefore, they must be preserved as required schema information. For instance, if $\tau_1 \subseteq \texttt{set/bag}(\tau)$ and $\tau_2 \subseteq \texttt{set/bag}(\tau)$ are existing constraints, the constraint $\tau_0 = \texttt{Merge\_Result}(\tau_1, \tau_2)$ needs to be preserved.

- A constraint is often simplified, i.e. replaced by one or more simpler constraints. For instance, the constraint $\texttt{set}(\sigma) = \texttt{set}(\tau)$ is replaced by the simpler one $\sigma = \tau$.

- A constraint occasionally implies one or more constraints. The latter need to be added to the set of constraints already produced. For example, $\texttt{Greater\_Less\_Than\_Compatible}(\tau_1, \tau_2)$ is inserted as a new constraint along with the implied constraints $\tau_1 \subseteq \texttt{orderable}$ and $\tau_2 \subseteq \texttt{orderable}$.

The effect varies depending on the constraint kind, the types involved in the constraint and the already existing constraints on these types.

It is worth pointing out that the resolution of constraints could take place either at the time each constraint is generated (*gradual resolution*) or at the time all the constraints have been produced (*accumulative resolution*).

The *gradual resolution* is very simple, since it usually concerns the insertion of a few constraints whose simplification (*resolution*) is straightforward. If their simplification produces new constraints then these are simplified as well, until no more constraints are produced.

The *accumulative resolution* starts from the constraints of the form $\tau_1 = \tau_2$. It simplifies them to constraints of the form $\texttt{type\_var} = \tau$ and replaces `type_var` by τ in all other constraints that involve `type_var`. Then it proceeds to simplify all other kinds of constraints. If a simplification leads to more constraints, the latter are added to the set of unprocessed constraints and are simplified in due course.

3.6 Type Inference Rules

Having explained the type system underlying our inference model and the various constraints generated and resolved by our algorithm, we are now in a position to present the backbone of our work, the *inference rules*. Note that there is a single rule for each OQL construct, and, therefore, the use of the rules by the inference algorithm is syntax driven. In this section, we present the inference rules for a substantial part of OQL; the rules for the remaining OQL constructs are given in Appendix A.

In the following rules, $\mathcal{H}$ signifies the *type environment*, that is $\mathcal{H} = \{var_i : \tau_i\}$, and $\mathcal{C}$ denotes the *constraints* added so far. The inference rules for the literal and the identifier queries are given first:

$$\frac{}{\mathcal{H};\mathcal{C} \vdash b\colon \texttt{bool} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash i\colon \texttt{int} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash f\colon \texttt{float} \Rightarrow \mathcal{C}}$$

$$\frac{}{\mathcal{H};\mathcal{C} \vdash c\colon \texttt{char} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash s\colon \texttt{string} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H} \cup \{\texttt{x}\colon \sigma\};\mathcal{C} \vdash \texttt{x}\colon \sigma \Rightarrow \mathcal{C}}$$

There are several rules to deal with various collections (sets, bags, lists, arrays). We just give one representative rule, that concerns the query construct $\texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_\texttt{n})$. As expected, the rule generates a constraint that ensures that the types of the queries are compatible equality-wise (or membership-wise). We also give the rules for accessing the first, last or i-th member of an ordered collection, as well as checking whether an element belongs in a certain collection. The constraint $\texttt{Member\_Type}(\sigma) = \tau$ denotes that σ is a collection (set, bag, list or array) with members of type τ.

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1\colon \sigma_1 \Rightarrow \mathcal{C}_1 \ \ldots\ \mathcal{H};\mathcal{C}_{n-1} \vdash \texttt{q}_\texttt{n}\colon \sigma_\texttt{n} \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_\texttt{n})\colon \texttt{set}(\texttt{CUB}(\sigma_1, \ldots, \sigma_\texttt{n})) \Rightarrow \mathcal{C}_n \wedge \{\texttt{Equality\_Compatible}(\sigma_1, \ldots, \sigma_\texttt{n})\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \texttt{first}(\texttt{q}_1)\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\texttt{Member\_Type}(\sigma) = \phi\} \wedge \{\sigma \subseteq \texttt{list/array}(\phi)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1\colon \sigma \Rightarrow \mathcal{C}_1 \ \ \mathcal{H};\mathcal{C}_1 \vdash \texttt{q}_2\colon \tau \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1[\texttt{q}_2]\colon \phi \Rightarrow \mathcal{C}_2 \wedge \{\tau = \texttt{int}\} \wedge \{\texttt{Member\_Type}(\sigma) = \phi\} \wedge \{\sigma \subseteq \texttt{list/array}(\phi)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma \Rightarrow \mathcal{C}_1 \;\; \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q_2} : \tau \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1\ in\ q_2} : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\mathtt{Equality\_Compatible}(\sigma, \mathtt{Member\_Type}(\tau))\}}$$

The rules for constructing a structure or an object, as well as for accessing a member of a structure or an object are given below. As is explained earlier, the constraint $\mathtt{Constructor\_Member\_Type}(\sigma, \ell) = \tau$ denotes that type σ is a class or a structure with a member called ℓ of type τ.

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma_1 \Rightarrow \mathcal{C}_1 \;\; \ldots \;\; \mathcal{H};\mathcal{C}_{n-1} \vdash \mathtt{q_n} : \sigma_n \Rightarrow \mathcal{C}_n}{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{class\_name}(\ell_1 : \mathtt{q_1}, \ldots, \ell_n : \mathtt{q_n}) : \mathtt{class\_name} \Rightarrow \mathcal{C}_n \wedge \\ \{\sigma_1 \subseteq \mathtt{Constructor\_Member\_Type}(\mathtt{class\_name}, \ell_1)\} \wedge \ldots \wedge \\ \{\sigma_n \subseteq \mathtt{Constructor\_Member\_Type}(\mathtt{class\_name}, \ell_n)\}\end{array}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma_1 \Rightarrow \mathcal{C}_1 \;\; \ldots \;\; \mathcal{H};\mathcal{C}_{n-1} \vdash \mathtt{q_n} : \sigma_n \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \mathtt{struct}(\ell_1 : \mathtt{q_1}, \ldots, \ell_n : \mathtt{q_n}) : \mathtt{struct}(\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n) \Rightarrow \mathcal{C}_n}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \tau \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1}.\ell : \sigma \Rightarrow \mathcal{C}_1 \wedge \{\mathtt{Constructor\_Member\_Type}(\tau, \ell) : \sigma\}}$$

The inference rules for the existential and the universal quantification follow. It is worth noting that the variable x is bound in query $\mathtt{q_2}$ to the member type of the collection $\mathtt{q_1}$.

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma_1 \Rightarrow \mathcal{C}_1 \;\; \mathcal{H} \cup \{\mathtt{x} : \phi\};\mathcal{C}_1 \vdash \mathtt{q_2} : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{exists\ x\ in\ q_1} : \ \mathtt{q_2} : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\sigma_2 = \mathtt{bool}\} \wedge \{\mathtt{Member\_Type}(\sigma_1) = \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma_1 \Rightarrow \mathcal{C}_1 \;\; \mathcal{H} \cup \{\mathtt{x} : \phi\};\mathcal{C}_1 \vdash \mathtt{q_2} : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{forall\ x\ in\ q_1} : \ \mathtt{q_2} : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\sigma_2 = \mathtt{bool}\} \wedge \{\mathtt{Member\_Type}(\sigma_1) = \phi\}}$$

An interesting set of rules concerns the application of methods with or without parameters. The inferred types of the queries used as arguments are not constrained to be the same as the types of the parameters of the method involved. They only need to be their subtypes.

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1}() : \phi \Rightarrow \mathcal{C}_1 \wedge \{\sigma = \mathtt{unit} \rightarrow \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_0} : \sigma_0 \Rightarrow \mathcal{C}_0 \;\; \ldots \;\; \mathcal{H};\mathcal{C}_{n-1} \vdash \mathtt{q_n} : \sigma_n \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_0}(\mathtt{q_1}, \ldots, \mathtt{q_n}) : \phi \Rightarrow \mathcal{C}_n \wedge \{\sigma_0 = \tau_1 \times \ldots \times \tau_n \rightarrow \phi\} \wedge \{\sigma_1 \subseteq \tau_1\} \wedge \ldots \wedge \{\sigma_n \subseteq \tau_n\}}$$

The rules concerning the query constructs q binop q or unop q are omitted for space reasons.

We finish by giving the rule for a simple `select` query; the judgements dealing with the remaining OQL expressions are presented in appendix A.

$$\frac{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{q_1} : \sigma_1 \Rightarrow \mathcal{C}_1 \\ \mathcal{H} \cup \{\mathtt{x_1} : \tau_1\};\mathcal{C}_1 \vdash \mathtt{q_2} : \sigma_2 \Rightarrow \mathcal{C}_2 \\ \ldots \\ \mathcal{H} \cup \{\mathtt{x_1} : \tau_1, \ldots, \mathtt{x_{n-1}} : \tau_{n-1}\};\mathcal{C}_{n-1} \vdash \mathtt{q_n} : \sigma_n \Rightarrow \mathcal{C}_n \\ \mathcal{H} \cup \{\mathtt{x_1} : \tau_1, \ldots, \mathtt{x_n} : \tau_n\};\mathcal{C}_n \vdash \mathtt{q_{01}} : \sigma_{01} \Rightarrow \mathcal{C}_{01} \\ \mathcal{H} \cup \{\mathtt{x_1} : \tau_1, \ldots, \mathtt{x_n} : \tau_n\};\mathcal{C}_{01} \vdash \mathtt{q_{00}} : \sigma_{00} \Rightarrow \mathcal{C}_{00}\end{array}}{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{select}\ \mathtt{q_{00}}\ \mathtt{from}\ \mathtt{q_1}\ \mathtt{as}\ \mathtt{x_1}, \ldots, \mathtt{q_n}\ \mathtt{as}\ \mathtt{x_n}\ \mathtt{where}\ \mathtt{q_{01}} : \mathtt{bag}(\sigma_{00}) \Rightarrow \mathcal{C}_{00} \wedge \\ \{\mathtt{Member\_Type}(\sigma_1) = \tau_1\} \wedge \ldots \wedge \{\mathtt{Member\_Type}(\sigma_n) = \tau_n\} \wedge \{\sigma_{01} = \mathtt{bool}\}\end{array}}$$

3.7 Inference Algorithm

Having given an overview of the type system, the constraints and the rules involved in our inference model, we may now present the core of our work, which is the inference algorithm. The algorithm takes as input a query `q` and returns its inferred type, as well as a pair $(\mathcal{H}, \mathcal{C})$ of a type environment and its constraints. This pair is a synopsis of the requirements a schema should satisfy so that the query `q` can be executed against it without any type-errors.

1. For each free variable `var` in the query q, $\mathcal{H} = \mathcal{H} \cup \{\mathtt{var} : \mathtt{new\_type\_var}\}$. Initially $\mathcal{C} = \{\}$.

2. Based on the construct of the query `q` recursively apply the appropriate inference rule.

3. Depending on the unification strategy, either simplify the constraints as soon as they are produced (gradual resolution) or simplify them all in the end after having applied all the inference rules. If the resolution process produces a type-error then the query is not typable and the algorithm is interrupted.

4. The final $\mathcal{H}, \mathcal{C}$ include the requirements a schema should have to be compatible with the query `q`. The type of q, which is the type inferred by the outer inference rule, also satisfies the constraints $\mathcal{C}$.

3.8 Discussion

Much research on schema evolution, schema inter-operation, distributed or semi-structured database applications has pointed out that there is a need to run queries in the presence of changing or heterogeneous schemata, or even in the absence of specific schema information [BDHS96]. The problem is addressed by proposing an inference algorithm for the ODMG query language OQL. This algorithm

infers the most general type of an OQL query and derives the schema information required so that the query can be executed against it without any type errors. In contrast to other work, we deal with a rather complex type system, which includes atomic types, structures, classes, various (parameterised) collection types (set, bag, list, array) and function types. This, in connection with the rich semantics of OQL, results in the generation of a wide variety of constraints by the inference rules. We discuss the semantics of these constraints and provide a mechanism for their solution. Finally, we present a set of inference rules for OQL, which is the core of our type inference algorithm. Based on our experience, this algorithm, as well as all the formalisms prior to it, are easy to implement, and hence, we believe that they could prove to be useful in many applications.

Chapter 4

OQL calculus representation

This chapter presents the translation of OQL into its first intermediate representation the monoid comprehension calculus. This calculus formalism has a number of advantages. Firstly, it has the expressive power to cover all OQL constructs. Secondly, being a calculus-style formalism, it supports the concept of object variables and it allows for nesting of expressions in a similar way to OQL. It would be hard to translate OQL directly into a collection-oriented algebraic representation. Thirdly, using forms known as monoid comprehensions, the monoid calculus captures OQL queries in a uniform way. This feature enables the optimization of calculus expressions using only a few rules.

The calculus representation adopted in this dissertation is based on the one proposed by Fegaras and Maier [FM00]. It has also been influenced by the work of Afshar [Afs98] in establishing a theoretical basis for his monoid query language (MQL). However, the calculus formalism adopted in this thesis addresses certain important issues that have not been considered previously. It highlights certain weak points of OQL and it adds new calculus expressions to deal with them; it proposes different ways of mapping certain OQL queries into their calculus representation (wrt [FM00]); finally, it introduces a series of new rules to transform calculus expressions into more efficient forms and gives a detailed algorithm that determines the order in which these rules must be applied.

4.1 Monoid Comprehension Calculus

This section gives an overview of the monoid calculus representation, as presented in previous work [Afs98, FM00]. It defines the notions of monoid, free monoid, monoid comprehension and monoid homomorphism and presents examples of how OQL is mapped to its calculus representation. The role of this section is to set up the theoretical basis for a formal description of our approach regarding the calculus intermediate representation of OQL.

4.1.1 Monoids

One of the benefits of using the monoid comprehension calculus is the uniform representation of OQL queries. For instance, this feature is obvious in the manipulation of multiple collection types. OQL is an expressive language that allows for the use of not only set collections, but also of bags, lists and arrays. The monoid calculus abstracts the similarities of these collections and allows for their uniform representation as collection monoids. Likewise, OQL includes a variety of operations on simple types, like integers and floats. In the context of our calculus, these can also be considered as primitive monoid types and hence they can be treated accordingly.

Definition 1 *A monoid of type* $\mathtt{T}$ *is a pair* $(\oplus, \mathcal{Z}_\oplus)$*, where* $\oplus$ *is an associative function of type* $\mathtt{T} \times \mathtt{T} \rightarrow \mathtt{T}$ *and* $\mathcal{Z}_\oplus$ *is the left and right identify of* $\oplus$ [FM00].

An alternative notation used for monoids in [Afs98] is $(\mathtt{T}, \oplus, \mathcal{Z})$. Function $\oplus$ is known as the *merge* operation that takes as parameters two elements of type $\mathtt{T}$ and returns an element of type $\mathtt{T}$. The value $\mathcal{Z}_\oplus$ is the zero element of the monoid $(\oplus, \mathcal{Z}_\oplus)$, i.e. the value such that $\mathcal{Z}_\oplus \oplus \mathtt{x} = \mathtt{x} \oplus \mathcal{Z}_\oplus = \mathtt{x}$ for every $\mathtt{x}$. For instance, integers can be represented by either of the monoids $(+, \mathtt{0})$ or $(\times, \mathtt{1})$. The first monoid merges integers based on the addition operation; the zero element is the integer 0. The second monoid merges integers by multiplying them and the zero (neutral) element is therefore the integer 1. Likewise booleans can be represented by two different monoids, merging boolean values using the $\wedge$ or the $\vee$ operations. These monoids on simple types are known as primitive monoids.

In order to represent collections, we need to define the notion of collection monoids [FM00]. These are monoids for which we need to define an additional *unit* function $\mathcal{U}_\oplus$. This function takes as parameter an element $\mathtt{x}$ and returns a singleton collection containing $\mathtt{x}$. Hence, a collection monoid is denoted by a triple $(\oplus, \mathcal{Z}_\oplus, \mathcal{U}_\oplus)$. It may represent any collection parametric type $\mathtt{T}_\oplus(\tau)$, such that τ supports comparison for equality. For instance, the monoid $(\cup, \{\}, \lambda\mathtt{x}.\{\mathtt{x}\})$ corresponds to the collection type $\mathtt{set}(\mathtt{a})$, where $\mathtt{a}$ is the type of the elements $\mathtt{x}$.

Monoids are said to be commutative and/or idempotent, if their merge operations are commutative and/or idempotent respectively. For instance, the set monoid $(\cup, \{\}, \lambda\mathtt{x}.\{\mathtt{x}\})$ is both commutative and idempotent, the integer monoid $+, \mathtt{0}$ is just commutative, whilst the list monoid $(++, [], \lambda\mathtt{x}.[\mathtt{x}])$ has neither of the two properties.

Based on these properties, Afshar distinguishes three monoid categories:

1. the (unrestricted) category of monoids, $\mathtt{M}$;
2. the category of commutative monoids, $\mathtt{CM}$;
3. the category of commutative and idempotent monoids, $\mathtt{CIM}$.

For the purposes of translating OQL queries to calculus formalism, we need to define an additional category, that of idempotent (but not necessarily commutative) monoids (IM).

Fegaras and Maier define a partial order $\preceq$ between monoids based on their properties. They define a mapping ψ from monoids to the set $\{\mathtt{C}, \mathtt{I}\}$ as: $\mathtt{C} \in \psi(\oplus)$ if and only if $\oplus$ is commutative and $\mathtt{I} \in \psi(\oplus)$ if and only if $\oplus$ is idempotent. The partial order $\preceq$ is defined as follows:

$$\otimes \preceq \oplus \equiv \psi(\otimes) \subseteq \psi(\oplus)$$

As it will be explained later on, both the monoid categories by Afshar, and the partial order by Fegaras and Maier are useful in order to determine when a mapping from one monoid to another, i.e. a monoid homomorphism, is well-defined.

4.1.2 Monoid Homomorphisms

Homomorphisms are defined in [Afs98] and [FM95] in slightly different ways. The definition by Afshar [Afs98] is given below.

Definition 2 *A monoid homomorphism between a monoid* $(\mathcal{T}, \times, \mathtt{e})$ *and a monoid* $(\mathcal{M}, *, \mathtt{I})$ *is a function* $\mathtt{f} : \mathcal{T} \rightarrow \mathcal{M}$ *which satisfies the following* [Afs98]*:*

$$\mathtt{f}\ \mathtt{e} = \mathtt{I}$$
$$\mathtt{f}(\mathtt{a} \times \mathtt{b}) = \mathtt{f}(\mathtt{a}) * \mathtt{f}(\mathtt{b})$$

Fegaras and Maier [FM95] define homomorphisms only on collection monoids. They regard homomorphisms as means of expressing OQL queries, none of which corresponds to a homomorphism from a primitive monoid. Hence although the previous definition by Afshar is more general [Afs98], the following one by Fegaras and Maier is sufficient in the context of translating OQL [FM95].

Definition 3 *A homomorphism* $\mathtt{hom}[\oplus, \otimes](\mathtt{f})\mathtt{A}$ *from the collection monoid* $(\oplus, \mathcal{Z}_\oplus, \mathcal{U}_\oplus)$ *to any monoid* $(\otimes, \mathcal{Z}_\otimes)$*, where* $\oplus \preceq \otimes$ *is defined by the following inductive equations* [FM00]*:*

$$\mathtt{hom}[\oplus, \otimes](\mathtt{f})(\mathcal{Z}_\oplus) = \mathcal{Z}_\otimes$$
$$\mathtt{hom}[\oplus, \otimes](\mathtt{f})(\mathcal{U}_\oplus(\mathtt{a})) = \mathtt{f}(\mathtt{a})$$
$$\mathtt{hom}[\oplus, \otimes](\mathtt{f})(\mathtt{x} \oplus \mathtt{y}) = (\mathtt{hom}[\oplus, \otimes](\mathtt{f})(\mathtt{x})) \otimes (\mathtt{hom}[\oplus, \otimes](\mathtt{f})(\mathtt{y}))$$

In the latter definition, the function $\mathtt{hom}[\oplus, \otimes](\mathtt{f})$ maps $\mathcal{Z}_\oplus$ to $\mathcal{Z}_\otimes$, $\mathcal{U}_\oplus$ to f and the merge operation $\oplus$ to $\otimes$. The third inductive equation reflects the "divide-and-conquer" mechanism implicit in homomorphisms. That is, one may apply the function f on a collection $\mathtt{x} \oplus \mathtt{y}$, by applying it separately on its parts x and y and merging the results using the operation $\otimes$.

In order for a monoid homomomorphism to be well-defined, Fegaras and Maier require the constraint $\oplus \preceq \otimes$ be hold. For instance, a homomorphism may map a list to a bag or to a set, but it

could not possibly map a bag to a list or a set to a bag. Fegaras and Maier set additional restrictions to ensure that homomorphisms are well-defined. If a monoid $(\oplus, \mathcal{Z}_\oplus)$ is not commutative it should be anti-commutative, i.e. $\forall \mathtt{x} \neq \mathtt{y} \neq \mathcal{Z}_\oplus : \mathtt{x} \oplus \mathtt{y} \neq \mathtt{y} \oplus \mathtt{x}$. Likewise, if a monoid is not idempotent it should be anti-idempotent, i.e. $\forall \mathtt{x} \neq \mathcal{Z}_\oplus : \mathtt{x} \oplus \mathtt{x} \neq \mathtt{x}$. However, these restrictions are not always satisfied in their framework. For instance, the primitive monoid $(\times, 1)$ is not idempotent. However there exists an element $\mathtt{x} = 0 \neq 1$ such that $0 \times 0 = 0$.

In this thesis, we use the notions of monoid category and free monoid (formally presented in [Afs98]) in order to express the necessary (but not sufficient) conditions for the well-definedness of monoid homomorphisms. A slightly rephrased definition of free monoids (found in [[Afs98]),ch.6] is given below:

Definition 4 *A free Monoid $(\mathcal{F}, *, \mathtt{e})$, where $\mathcal{F}$ is generated by the injection map $\mathtt{i} : \mathcal{S} \rightarrow \mathcal{F}$, is such that* [Afs98]*:*

$\forall \mathtt{f} : \mathcal{S} \rightarrow \mathcal{M}$ *– where* $\mathtt{f}$ *is a set function and* $\mathcal{M}$ *is a monoid*

$\exists \mathtt{h} : \mathcal{F} \rightarrow \mathcal{M}$ *– where* $\mathtt{h}$ *is a monoid homomorphism*

s.t. $\mathtt{h} \circ \mathtt{i} = \mathtt{f}$

In order for a homomorphism $\mathtt{f} : \mathcal{M} \rightarrow \mathcal{N}$ to be well-formed, $\mathcal{N}$ should lie in the same category as $\mathcal{M}$. In addition to that, if $\mathcal{N}$ is a free monoid then $\mathcal{M}$ should also be free.

4.1.3 Monoid Comprehensions

Fegaras and Maier use homomorphisms to define formal semantics for monoid comprehensions. A monoid comprehension has the form $\oplus\{\mathtt{e} \parallel \mathtt{q}_1, \ldots, \mathtt{q}_n\}$, where $\oplus$ is called the accumulator of the comprehension, $\mathtt{e}$ is the head and $\mathtt{q}_i$ are the qualifiers of the comprehension. A qualifier takes either of the following forms:

- $\mathtt{u} \leftarrow \mathtt{e}'$: a *generator* binds elements of the collection $\mathtt{e}'$ to the variable $\mathtt{u}$
- $\mathtt{u}' \equiv \mathtt{e}''$: a *binding* binds the variable $\mathtt{u}'$ to the value of the expression $\mathtt{e}''$
- $\mathtt{e}'''$: a *filter* may include variables bound in previous generators or bindings.

Definition 5 *A monoid comprehension over a primitive or collection monoid $\oplus$ is defined by the following inductive equations: ($\otimes$ is the collection monoid associated with the expression* $\mathtt{u}$ *and is possibly different from $\oplus$)* [FM95]

$$\oplus\{\mathtt{e} \parallel\} = \begin{cases} \mathcal{U}_\oplus(\mathtt{e}) & \text{for a collection monoid } \oplus \\ \mathtt{e} & \text{for a primitive monoid } \oplus \end{cases}$$

$$\oplus\{\mathtt{e} \parallel \mathtt{x} \leftarrow \mathtt{u}, \bar{\mathtt{q}}\} = \mathtt{hom}[\otimes, \oplus](\lambda \mathtt{x}. \oplus \{\mathtt{e} \parallel \bar{\mathtt{q}}\})\mathtt{u}$$

$$\oplus\{\mathtt{e} \parallel \mathtt{pred}, \bar{\mathtt{q}}\} = \mathtt{if\ pred\ then} \oplus \{\mathtt{e} \parallel \bar{\mathtt{q}}\} \mathtt{\ else\ } \mathcal{Z}_\oplus$$

where $\bar{\mathtt{q}} = \mathtt{q}_1, \ldots, \mathtt{q}_n,\ \mathtt{n} \geq 0$.

An example is given that transforms a `select` query into its calculus counterpart.

```
select x
from Countries as x
where x.population > 50,000,000
```

The OQL query above is translated to the following comprehension:

$$\cup\{\mathtt{x} \parallel \mathtt{x} \leftarrow \mathtt{Countries}, \mathtt{x.population} > 50,000,000\}$$

Fegaras and Maier present the syntax of their monoid comprehension calculus, as well as the translation rules for most OQL constructs. The interested reader may refer to [FM95, FM00] for the detailed rules that map queries into their calculus representation.

The calculus adopted in this thesis departs from their proposal in certain respects. Hence some of the translation rules from OQL to calculus form and certain optimization rules are different. The remaining sections highlight the points at which the current approach differs from their work and explains our reasons for proposing a number of modifications and extensions.

4.2 Extended Calculus: syntax and monoids

4.2.1 Calculus Syntax

The grammar of the monoid comprehension calculus adopted in this thesis is given in figure 4.1. The calculus expressions that are new in this framework (wrt [FM00]) are:

$$< \mathtt{class\_name} > (\ell_1 = \mathtt{e}_1, \ldots, \ell_k = \mathtt{e}_k),$$
$$\mathtt{e1[e2]},\ \mathtt{not}\ (\mathtt{e}),$$
$$\mathtt{bagof}(\mathtt{e}),\ \mathtt{listof}(\mathtt{e}),\ \mathtt{arrayof}(\mathtt{e})$$
$$\mathtt{orderof}(\mathtt{e}),\ \mathtt{elementof}(\mathtt{e})$$

The construct $< \mathtt{class\_name} > (\ell_1 = \mathtt{e}_1, \ldots, \ell_k = \mathtt{e}_k)$ has been added in order to deal with the construction of objects. The second one ($\mathtt{e1[e2]}$) accesses the $\mathtt{e}_2$-th element of the collection $\mathtt{e}_1$, whilst $\mathtt{not}(\mathtt{e})$ obviously handles boolean negation. The last four expressions $\mathtt{bagof}(\mathtt{e})$, $\mathtt{listof}(\mathtt{e})$, $\mathtt{arrayof}(\mathtt{e})$ and $\mathtt{orderof}(\mathtt{e})$ have been added in order to avoid the problem of ill-formed comprehensions. The partial function $\mathtt{elementof}(\mathtt{e})$ is the inverse of `unit`, that is, it takes a singleton collection and returns its element. If the collection has no elements or more than one element, an error is generated. If `e` has type $\mathtt{coll}(\tau)$, then the types of $\mathtt{bagof}(\mathtt{e})$, $\mathtt{listof}(\mathtt{e})$, $\mathtt{arrayof}(\mathtt{e})$, $\mathtt{orderof}(\mathtt{e})$ and $\mathtt{elementof}(\mathtt{e})$ are $\mathtt{bag}(\tau)$, $\mathtt{list}(\tau)$, $\mathtt{array}(\tau)$, $\mathtt{list}(\tau)$ and τ respectively.

$$
\begin{array}{rcl}
\mathtt{e}, \mathtt{e}_1, \ldots, \mathtt{e}_n & ::= & \mathtt{literal}\ (integer, float, boolean, char, string) \\
& | & < \mathtt{id} > \ (\mathtt{db\ object/class/extent\ name}) \\
& | & \mathtt{zero}[< \mathtt{monoid} >] \\
& | & \mathtt{unit}[< \mathtt{monoid} >](\mathtt{e}) \\
& | & \mathtt{merge}[< \mathtt{monoid} >](\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
& | & < \mathtt{monoid} > \{\mathtt{e} \parallel \mathtt{qualifiers}\} \\
& | & \mathtt{struct}(\ell_1 = \mathtt{e}_1, \ldots, \ell_k = \mathtt{e}_k) \\
& | & < \mathtt{class\_name} > (\ell_1 = \mathtt{e}_1, \ldots, \ell_k = \mathtt{e}_k) \\
& | & \mathtt{e}_1 \ < \mathtt{comparison\_operator} > \ \mathtt{e}_2 \\
& | & \mathtt{e}_1 \ < \mathtt{arithmetic\_operator} > \ \mathtt{e}_2 \\
& | & \mathtt{e}. < \mathtt{attr\_rltp\_method} > \\
& | & \mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_k) \\
& | & \mathtt{e1}[\mathtt{e2}] \\
& | & \mathtt{not}(\mathtt{e}) \\
& | & \mathtt{bagof}(\mathtt{e}) \\
& | & \mathtt{listof}(\mathtt{e}) \\
& | & \mathtt{arrayof}(\mathtt{e}) \\
& | & \mathtt{orderof}(\mathtt{e}) \\
& | & \mathtt{elementof}(\mathtt{e})
\end{array}
$$

where

- $< \mathtt{comparison\_operator} > \in \{<, >, \leq, \geq, =, \neq\}$,
- $< \mathtt{arithmetic\_operator} > \in \{+, -, \times, /, \mathtt{mod}\}$,
- $< \mathtt{attr\_rltp\_method} >$ is the name of an attribute, relationship or method of a class,
- $< \mathtt{monoid} >$ is either a primitive or a collection monoid,
- $\mathtt{zero}[< \mathtt{monoid} >]$, $\mathtt{unit}[< \mathtt{monoid} >](\mathtt{e})$, and $\mathtt{merge}[< \mathtt{monoid} >](\mathtt{e}_1, \ldots, \mathtt{e}_n)$ are alternative representations of the zero element $\mathcal{Z}_\oplus$, the unit function $\mathcal{U}_\oplus$ and the merge operation $\oplus$ of some $< \mathtt{monoid} >$ $(\mathcal{Z}_\oplus, \oplus)$.

Figure 4.1: Syntax of the calculus OQL representation

The ODMG standard defines that every collection in the `from` clause of a `select` query is converted into a bag ([CBB+00],4.10.9). For instance, consider the query:

```
select x
from Students as x
where x.grade = 10
```

Although the extent `Students` has type `set(Student)`, the result of the `from` clause is a bag of students. Hence it would be wrong to translate the query above to the calculus form:

$$\texttt{bag}\{\texttt{x} \parallel \texttt{x} \leftarrow \texttt{Students}, \texttt{x.grade} = 10\}$$

This is an ill-formed comprehension, since there is no homomorphic mapping from a set to a bag. To avoid this problem, we first apply the non-homomorphic function `bagof`$(-)$ to the set `Students`. The resulting calculus expression is:

$$\texttt{bag}\{\texttt{x} \parallel \texttt{x} \leftarrow \texttt{bagof}(\texttt{Students}), \texttt{x.grade} = 10\}$$

.

The non-homomorphic function `listof`$(-)$ is useful in a number of contexts. First it deals with the non-determinism introduced when ordering the elements of a collection based on insufficient sort criteria. The Standard [CBB+00] states the following:

> "The order by a set of functions ($\texttt{f}_1, \ldots, \texttt{f}_\texttt{n}$) is performed as follows: First sort according to function $\texttt{f}_1$, then for all elements having the same $\texttt{f}_1$ value sort them according to $\texttt{f}_2$, and so on ([CBB+00],ch 4.10.9.4)".

The question that arises is "what if there are several elements with the same values of $\texttt{f}_1, \ldots, \texttt{f}_\texttt{n}$?". For instance, the result of the following query is not determined in the case that at least two students have the same grade.

```
select x.name
from Students as x
order by x.grade
```

To this question, the ODMG standard does not give an adequate solution. It is implied that whether the ordering is *complete* is decided at run-time. If it is not complete, then either the query produces a non-deterministic result, or alternatively, it raises an error and it is rejected. In the former case, it is possible that the user gets different results every time s/he submits the same query; in the latter case, the user needs to wait until execution time to be notified about a possible error.

We suggest a different way of handling non-determinism. First we define the collection monoid `oset`, as a means of handling an ordered collection of elements with no duplicates. `oset` is defined

over a parametric type τ, such that the values of this type can be compared for equality. Its zero element and the unit and merge operations are defined as follows:

- $\texttt{zero}[\texttt{oset}] = \| \, \|$
- $\texttt{unit}[\texttt{oset}](\texttt{e}) = \| \, \texttt{e} \, \|$
- $\texttt{merge}[\texttt{oset}](\| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|, \| \, \texttt{e} \, \|)$ is evaluated as follows:
  - If there exists i, s.t. $\texttt{e}_\texttt{i} = \texttt{e}$, then $\texttt{merge}[\texttt{oset}](\| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|, \| \, \texttt{e} \, \|) = \| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|$
  - Otherwise, $\texttt{merge}[\texttt{oset}](\| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|, \| \, \texttt{e} \, \|) = \| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n}, \texttt{e} \, \|$
- $\texttt{merge}[\texttt{oset}](\| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|, \| \, \texttt{e}'_1, \ldots, \texttt{e}'_\texttt{m} \, \|) =$
  $\texttt{merge}[\texttt{oset}](\texttt{merge}[\texttt{oset}](\| \, \texttt{e}_1, \ldots, \texttt{e}_\texttt{n} \, \|, \| \, \texttt{e}'_1 \, \|), \| \, \texttt{e}'_2, \ldots, \texttt{e}'_\texttt{m} \, \|)$

Inserting an existing element into an ordered set (oset) has no effect. Inserting a new element in an ordered set is identical to inserting it into a list. Thus, oset is idempotent, but not commutative. oset is free in the idempotent category of monoids (IM).

The following proposal for handling non-determinism departs from the Standard significantly:

- A $\texttt{select} \cdot \texttt{distinct} \cdot \ldots \cdot \texttt{order by}\ \texttt{e}_1, \ldots, \texttt{e}_\texttt{n}$ clause yields an oset of bags of elements. Each bag contains elements having the same values for all sort criteria $\texttt{e}_1, \ldots, \texttt{e}_\texttt{n}$. If no two elements have the same values for all the criteria, then the result is an oset of singleton bags.
- A $\texttt{select} \cdot \ldots \cdot \texttt{order by}\ \texttt{e}_1, \ldots, \texttt{e}_\texttt{n}$ clause (without distinct) yields a list of bags of elements. Each bag contains elements having the same values for all sort criteria $\texttt{e}_1, \ldots, \texttt{e}_\texttt{n}$. If no two elements have the same values for all the criteria, then the result is a list of singleton bags.

Intuitively, the result of a $\texttt{select} \cdot [\texttt{distinct}\cdot] \ldots \cdot \texttt{order by} \ldots$ query is expected to be a list or an ordered set of elements. However, the solution proposed above returns a list or an ordered set of bags. Thus, we need a new OQL construct order_flatten that flattens this result into the expected collections. Non-determinism is inevitably introduced by flattening a list of non-singleton bags into a list of elements. The ordering of elements in the same bag is determined by system-dependent algorithms ($\texttt{listof}(\texttt{set}/\texttt{bag}(\texttt{elements}))$), and is therefore considered arbitrary in an open distributed world. Although this approach does not avoid non-determinism, it has the advantage that it avoids applying the user's sort criteria and the arbitrary sorting algorithm in one step. The user can see first the result of his sorting expression. On his conscious request (by the use of order_flatten), he can then get a flattened list or ordered set, especially for presentation or printing purposes.

The following table illustrates the result of applying the order_flatten on different nested collections. It is defined in a way corresponding to the discussion about flatten in the ODMG standard [CBB+00]. In what follows the notation $\texttt{coll}_1/\texttt{coll}_2$ (e.g. set/bag) denotes alternatives, not general types.

$\texttt{coll}_1$	$\texttt{coll}_2$	$\texttt{order\_flatten}(\texttt{coll}_1(\texttt{coll}_2(\texttt{elem})))$
set/bag	set/bag	$\texttt{coll}_2(\texttt{elem})$
list/oset/array	set/bag	$\texttt{coll}_1(\texttt{elem})$
set/bag	list/oset/array	$\texttt{coll}_1(\texttt{elem})$
list/oset/array	list/oset/array	$\texttt{coll}_2(\texttt{elem})$

This section presented the calculus expressions used in our framework, and discussed the role of the nonhomomorphic functions bagof and listof. The functionality of arrayof and orderof remains to be discussed in section 4.2.3. In order to translate OQL queries into this calculus representation, we first need to define the monoids involved in the translation. In the following two sections, we pay particular attention to the new monoids (wrt [FM00]) that have been introduced in this dissertation, e.g. the group and the dictionary monoids. We also present several translation rules from OQL to calculus that involve these monoids. The complete set of translation rules is given in Appendix B.

4.2.2 Monoids

The primitive and collection monoids used in our context are the following:

Primitive Monoids	Collection Monoids
sum (commutative)	set (commutative, idempotent)
avg (commutative)	bag (commutative)
some (commutative, idempotent)	list
all (commutative, idempotent)	array
max (commutative, idempotent)	oset (idempotent)
min (commutative, idempotent)	dictionary
	group (commutative)

Some of the monoids above are identical to those used in relevant calculus frameworks. The interested reader may refer to [FM00] for their descriptions. The remainder of this section defines only those monoids that have been introduced in the current framework.

The group monoid

The group monoid in our context is defined over an unordered collection of structures of type $\texttt{struct}(< \ell_1 >: \tau_1, \ldots, < \ell_n >: \tau_n, \texttt{partition} : \texttt{bag}(\tau))$, such that types τ_i, are comparable types

and τ is any nonfunctional type. Informally, group is used to manipulate structures with at least one label called partition and at least one other label. All labels except partition are called *group-by attributes*. All structures in the collection have the same labels. In order to insert a new structure in the group monoid, we check whether there exists a structure such that its *group-by* attributes have the same values as the corresponding attributes of the new structure. If there exists such a structure, then its partition (which is a bag) is updated, by being merged with the partition of the new structure. Otherwise, the new structure is inserted, as it would be normally inserted into a bag.

More formally,

- $\mathtt{zero}[\mathtt{group}] = \ll\gg$
- $\mathtt{unit}[\mathtt{group}](\mathtt{struct} < \ell_1 = \mathtt{e}_1, \ldots, \ell_j = \mathtt{e}_j, \mathtt{partition} = \mathtt{e} >) =$
  $\ll \mathtt{struct} < \ell_1 = \mathtt{e}_1, \ldots, \ell_j = \mathtt{e}_j, \mathtt{partition} = \mathtt{e} > \gg$
- Consider $\mathtt{struct}_i = \mathtt{struct} < \ell_1 = \mathtt{v}_{i1}, \ldots, \ell_j = \mathtt{v}_{ij}, \mathtt{partition} = \mathtt{v}_i >, i = 1, \ldots, n$ and $\mathtt{struct}' =< \ell_1 = \mathtt{v}_{k1}, \ldots, \ell_j = \mathtt{v}_{kj}, \mathtt{partition} = \mathtt{v} >$.

  - If $1 \leq k \leq n$, $\mathtt{merge}[\mathtt{group}](\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_n \gg, \ll \mathtt{struct}' \gg) =$
    $\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_{k-1}, \mathtt{struct} < \ell_1 = \mathtt{v}_{k1}, \ldots, \ell_j = \mathtt{v}_{kj}, \mathtt{partition} =$
    $\mathtt{merge}[\mathtt{bag}](\mathtt{v}_k, \mathtt{v}) >, \mathtt{struct}_{k+1}, \ldots, \mathtt{struct}_n > \gg$.

  - Otherwise, if none of the existing structures has the same values for the group-by attributes as $\mathtt{struct}'$,
    $\mathtt{merge}[\mathtt{group}](\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_n \gg, \ll \mathtt{struct}' \gg) =$
    $\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_n, \mathtt{struct}' \gg$.
- $\mathtt{merge}[\mathtt{group}](\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_n \gg, \ll \mathtt{struct}'_1, \ldots, \mathtt{struct}'_m \gg) =$
  $\mathtt{merge}[\mathtt{group}](\mathtt{merge}[\mathtt{group}](\ll \mathtt{struct}_1, \ldots, \mathtt{struct}_n \gg, \ll \mathtt{struct}'_1 \gg),$
  $\ll \mathtt{struct}'_2, \ldots, \mathtt{struct}'_m \gg)$

group is commutative, since the order in which we merge structures does not make any difference. We just compare values of the group-by attributes, and if they match we merge their partitions. The latter are simple bags, so their merging is a commutative operation. group is not idempotent, since if we merge two singleton group monoids with identical structures, the partition of the result contains twice the elements of each initial partition.

group is more specific than the corresponding $\mathtt{grouped}(\mathtt{V} \times \mathtt{N})$ monoid defined in [Afs98]. It accumulates all elements with the same grouping values into a bag, whilst $\mathtt{grouped}(\mathtt{V} \times \mathtt{N})$ groups on V values into any monoid N. In the case of group the elements are structures, the grouping values are

paired with corresponding labels and all elements having the same grouping values are merged into a bag with label `partition`. Instead, the merge operation of $\texttt{grouped}(\texttt{V} \times \texttt{N})$ accumulates tuples, that is neither the grouping values nor the grouped elements have corresponding labels. This is not possible in OQL, the type system of which does not support tuples.

The `dictionary` monoid

`dictionary` represents an unordered collection of structures of type $\texttt{struct}(\texttt{key} : \sigma, \texttt{value} : \tau)$, such that values of type σ can be compared for equality and τ is any nonfunctional type. Before inserting a new structure into a dictionary, we check whether the latter already contains a structure with the same key. If this is the case, the existing structure $\texttt{struct}(\texttt{key} : \texttt{k}, \texttt{value} : \texttt{old\_value})$ is replaced by the new one $\texttt{struct}(\texttt{key} : \texttt{k}, \texttt{value} : \texttt{new\_value})$. Otherwise, the latter is inserted as it would be inserted into a bag. The notation $(\texttt{k}, \texttt{v})$ is often used as an alternative representation of the expression $\texttt{struct}(\texttt{key} : \texttt{k}, \texttt{value} : \texttt{v})$. More formally,

- $\texttt{zero}[\texttt{dict}] = \prec\succ$
- $\texttt{unit}[\texttt{dict}]((\texttt{k}, \texttt{v})) = \prec (\texttt{k}, \texttt{v}) \succ$
- If there exists $i,\ 1 \leq i \leq n$ such that $\texttt{k}_\texttt{i} = \texttt{k}$, then
  $\texttt{merge}[\texttt{dict}](\prec (\texttt{k}_1, \texttt{v}_1), \ldots, (\texttt{k}_\texttt{i}, \texttt{v}_\texttt{i}), \ldots, (\texttt{k}_\texttt{n}, \texttt{v}_\texttt{n}) \succ, \prec (\texttt{k}, \texttt{v}) \succ)$
  $= \prec (\texttt{k}_1, \texttt{v}_1), \ldots, (\texttt{k}_\texttt{i}, \texttt{v}), \ldots, (\texttt{k}_\texttt{n}, \texttt{v}_\texttt{n}) \succ$
- If there is no $i,\ 1 \leq i \leq n$ such that $\texttt{k}_\texttt{i} = \texttt{k}$, then
  $\texttt{merge}[\texttt{dict}](\prec (\texttt{k}_1, \texttt{v}_1), \ldots, (\texttt{k}_\texttt{n}, \texttt{v}_\texttt{n}) \succ, \prec (\texttt{k}, \texttt{v}) \succ)$
  $= \prec (\texttt{k}_1, \texttt{v}_1), \ldots, (\texttt{k}_\texttt{n}, \texttt{v}_\texttt{n}), (\texttt{k}, \texttt{v}) \succ$

The `merge` operation of the monoid `dictionary` is idempotent. That is, for each pair $(\texttt{k}, \texttt{v})$ $\texttt{merge}[\texttt{dict}](\prec (\texttt{k}, \texttt{v}) \succ, \prec (\texttt{k}, \texttt{v}) \succ) = \prec (\texttt{k}, \texttt{v}) \succ$. `dictionary` is not a commutative monoid, since $\texttt{merge}[\texttt{dict}](\prec (1, 2) \succ, \prec (1, 3) \succ) = \prec (1, 3) \succ$, whilst $\texttt{merge}[\texttt{dict}](\prec (1, 3) \succ, \prec (1, 2) \succ) = \prec (1, 2) \succ$.

Notice that the `merge` operation of `dictionary` resembles the corresponding operation of the monoid `array`, defined by Fegaras and Maier [FM00]. A significant difference is that the `key` in their context is always integer, since it reflects the position of the `value` in the array.

4.2.3 Translation rules

The previous subsections defined the monoids that are novel in this calculus framework. This section uses these monoids to present a set of translation rules from OQL constructs to their calculus representation. In fact, it only presents those translation rules that differ from the corresponding ones in

OQL Expressions[1]	Calculus Expressions
select [distinct] f from q_1 as $x_1, \ldots, q_k$ as x_k where p	bag [|set\]{|f|* ‖ $x_1 \leftarrow$ bagof(|q_1|*), ..., $x_k \leftarrow$ bagof(|q_k|*), |p|*}
select [distinct] f from q_1 as $x_1, \ldots, q_k$ as x_k where p order by o_1 [asc/desc], ..., o_j [asc/desc]	list [|oset\]{v_1.partition ‖ $v_1 \leftarrow$ orderof(group{ struct(asc/desc$_1$ = |o_1|*, ..., asc/desc$_j$ = |o_j|*, partition = unit[bag](|f|*)) ‖ $x_1 \leftarrow$ bagof(|q_1|*), ..., $x_k \leftarrow$ bagof(|q_k|*), |p|*})}

select [distinct] f
from q_1 as $x_1, \ldots, q_k$ as x_k
where p
group by $\ell_1 : g_1, \ldots, \ell_j : g_j$
having h
order by o_1 [asc/desc],
..., o_n [asc/desc]
	list [|oset\]{v_1.partition ‖ $v_1 \leftarrow$ orderof(group{
	struct(asc/desc$_1$ = |o_1|*, ..., asc/desc$_n$ = |o_n|*,
	partition = unit[bag](f)) ‖ $v_2 \leftarrow$ bagof(group{
	struct(ℓ_1 = |g_1|*, ..., ℓ_j = |g_j|*, partition =
	unit[bag](struct($x_1 = x_1, \ldots, x_k = x_k$)) ‖ $x_1 \leftarrow$
	bagof(|q_1|*), ..., $x_k \leftarrow$ bagof(|q_k|*), |p|*}), $\ell_1 \equiv v_2.\ell_1$,
	..., $\ell_j \equiv v_2.\ell_j$, partition $\equiv v_2$.partition, |h|*})}

Figure 4.2: Translation of `select` queries to their calculus representation

other calculus frameworks [FM00, Afs98]. The complete set of translation rules from OQL to the monoid calculus is given in Appendix B.

We use the notation $|q|^*$ to denote the calculus equivalent of query q ($|-|^* : \texttt{OQL} \rightarrow \texttt{Calculus}$).

First the calculus expression $e_1[e_2]$ enables accessing ordered collections. In order to translate the query last(q), the primitive monoid sum is used. Since this monoid is not idempotent, the elements of the collection q are counted after transforming it into a bag.

$$|\texttt{first}(q)|^* = |q|^*[0]$$
$$|\texttt{last}(q)|^* = |q|^*[\texttt{sum}\{1 \,\|\, x \leftarrow \texttt{bagof}(|q|^*)\} - 1]$$
$$|q_1[q_2]|^* = |q_1|^*[|q_2|^*]$$

Second, structures and classes are translated to calculus as follows:

$$|\texttt{struct}(\ell_1 : q_1, \ldots, \ell_n : q_n)|^* = \texttt{struct}(\ell_1 = |q_1|^*, \ldots, \ell_n = |q_n|^*)$$
$$|C(\ell_1 : q_1, \ldots, \ell_n : q_n)|^* = C(\ell_1 = |q_1|^*, \ldots, \ell_n = |q_n|^*)$$

The translation of a dictionary expression is straightforward.

[1] f, g_i, h, p, q_i and o_i represent OQL queries in figure 4.2.

$$|\texttt{dictionary}((\texttt{q}_1, \texttt{q}'_1), \ldots, (\texttt{q}_\texttt{n}, \texttt{q}'_\texttt{n}))|^* = \texttt{merge}[\texttt{dict}](\texttt{unit}[\texttt{dict}]((|\texttt{q}_1|^*, |\texttt{q}'_1|^*)), \ldots, \texttt{unit}[\texttt{dict}]((|\texttt{q}_\texttt{n}|^*, |\texttt{q}'_\texttt{n}|^*)))$$

The translation of `select` queries is given in figure 4.2. All collections in the `from` clauses are first converted to bags. Moreover, the monoid `group` and a new function `orderof` are used to translate queries with a `group by` and an `order by` clause. The function `orderof` operates on a collection of structures of the form:

$$\texttt{struct}(< \texttt{sc}_1 >= \texttt{v}_1, \ldots, < \texttt{sc}_\texttt{n} >= \texttt{v}_\texttt{n}, \texttt{partition} = \texttt{v})$$

where $< \texttt{sc}_\texttt{i} > ::= \texttt{asc}_\texttt{i} | \texttt{desc}_\texttt{i}$ and the types $\tau_\texttt{i}$ of $\texttt{v}_\texttt{i}$ are sortable types, for all $\texttt{i} = 1, \ldots, \texttt{n}$.

All structures in the collection have the same labels. The labels $< \texttt{sc}_\texttt{i} >$ are called *sort criteria*. By convention, their names start with `asc` or `desc` and are followed by the serial number of the corresponding criterion. For instance, if there are two sort criteria in the structures, the first being ascending and the second descending, their names would be $\texttt{asc}_1$ and $\texttt{desc}_2$ respectively. The function `orderof` sorts these structures based on their sort criteria. It is assumed that there are no structures with the same values for the sort criteria. The reason why is that `orderof` always applies to a `group` comprehension, such that the group-by attributes coincide with the sort criteria. Being a nonhomomorphic function, it waits until all the elements of its operand are calculated, before it starts ordering them.

The non-homomorphic function `bagof` is applied to the result of `group`, since there is no homomorphic function from `group` to `bag` or between two `group` monoids with different grouping attributes (labels).

The result of a `select` query with an `order by` clause is not a `list` of elements, but a `list` or `oset` of bags of elements. As is explained in section 4.2.1 a new OQL construct `order_flatten` is introduced to flatten the result into a `list` or `oset` of elements. The calculus forms of the new construct are given below:

$$|\texttt{order\_flatten}(\texttt{q})|^* = \texttt{list}\{\texttt{x} \parallel \texttt{s} \leftarrow |\texttt{q}|^*, \texttt{x} \leftarrow \texttt{listof}(\texttt{s})\} \quad \texttt{q} : \texttt{list}(\texttt{bag}(-))$$
$$|\texttt{order\_flatten}(\texttt{q})|^* = \texttt{oset}\{\texttt{x} \parallel \texttt{s} \leftarrow |\texttt{q}|^*, \texttt{x} \leftarrow \texttt{listof}(\texttt{s})\} \quad \texttt{q} : \texttt{oset}(\texttt{bag}(-))$$

The ODMG defines that the OQL construct `distinct` converts a bag into a set. However, the result of the `distinct` operation on a list remains a list. In this framework, `distinct` converts lists or arrays into ordered sets (`osets`).

$$|\texttt{distinct}(\texttt{q})|^* = \texttt{set}\{\texttt{x} \parallel \texttt{x} \leftarrow |\texttt{q}|^*\} \quad \texttt{q} : \texttt{set/bag}$$
$$|\texttt{distinct}(\texttt{q})|^* = \texttt{oset}\{\texttt{x} \parallel \texttt{x} \leftarrow |\texttt{q}|^*\} \quad \texttt{q} : \texttt{list/oset}$$
$$|\texttt{distinct}(\texttt{q})|^* = \texttt{oset}\{\texttt{x} \parallel \texttt{x} \leftarrow \texttt{listof}(|\texttt{q}|^*)\} \quad \texttt{q} : \texttt{array}$$

When applied on an ordered collection, the non-homomorphic function `listof` converts its operand into a list respecting the prespecified order of its elements.

Another point at which this framework extends the Standard is handling collection operations, i.e. `union`, `except` and `intersect`. The Standard allows only sets and bags to be the operands of such constructs. It is interesting to look at a possible extension of the Standard, that allows the `union` operation between lists, arrays and ordered sets. However, we do not attempt to define the operations `except` and `intersect` between lists, ordered sets or arrays. The reason why is that when $\mathtt{c}_1, \mathtt{c}_2$ are ordered collections, it is difficult to satisfy the following property:

$$\mathtt{union}(\mathtt{except}(\mathtt{c}_1, \mathtt{c}_2), \mathtt{intersect}(\mathtt{c}_1, \mathtt{c}_2)) = \mathtt{c}_1 \tag{4.1}$$

The following table presents the result of a union operation over different collections.

$$\begin{aligned}
\mathtt{union} \;&:\; \mathtt{set(a)} * \mathtt{set(a)} \rightarrow \mathtt{set(a)} \\
&:\; \mathtt{set(a)/bag(a)} * \mathtt{bag(a)} \rightarrow \mathtt{bag(a)} \\
&:\; \mathtt{oset(a)} * \mathtt{oset(a)} \rightarrow \mathtt{oset(a)} \\
&:\; \mathtt{array(a)} * \mathtt{array(a)} \rightarrow \mathtt{array(a)} \\
&:\; \mathtt{oset(a)/list(a)/array(a)} * \mathtt{list(a)} \rightarrow \mathtt{list(a)} \\
&:\; \mathtt{oset(a)} * \mathtt{array(a)} \rightarrow \mathtt{list(a)}
\end{aligned}$$

In addition, if sets or bags are merged with lists, osets or arrays, lists and arrays are converted into bags and osets into sets ($\mathtt{set(a)} * \mathtt{list(a)} \Leftrightarrow \mathtt{set(a)} * \mathtt{bag(a)}$). If arrays are merged with other collection types, they are converted into lists. The only issue that needs to be resolved at this point is how the union of two arrays is evaluated.

Manipulating arrays

The Standard does not define a union operation between arrays. However, it allows flattening an array of arrays by merging the elements of the nested arrays. It is not clear whether this implies a list concatenation, where `NULL` values are preserved, or a merge operation ($\Box$) of vectors, as defined by Fegaras and Maier ([FM00], section 5.3). In the first case, two arrays $[1, \mathtt{NULL}, 3]$ and $[2, 5, \mathtt{NULL}, 12, \mathtt{NULL}]$ would be merged into $[1, \mathtt{NULL}, 3, 2, 5, \mathtt{NULL}, 12, \mathtt{NULL}]$. This would make the use of arrays trivial, since their manipulation in the context of OQL would be exactly the same as that of lists. In the second case [FM00], vectors (or arrays) are represented as sets of pairs $(\mathtt{i}, \mathtt{e})$, where `i` is the index of element `e`. Merging the arrays of the previous example, represented as $\{(1,1),(3,3)\}$ and $\{(1,2),(2,5),(4,12)\}$, would result in the array $\{(1,2),(2,5),(3,3),(4,12)\}$. That is, a new value in position `i` of the second array overwrites the old value in the same position of the first array. Although this approach reflects the way vectors are manipulated in programming languages (assigning a value to a vector position), it creates practical problems in the context of OQL.

Consider for instance the OQL query $\texttt{exists x in } (\texttt{q}_1 \texttt{ union } \texttt{q}_2) : \texttt{p(x)}$, where $\texttt{q}_1, \texttt{q}_2$ are arrays. The calculus form of this query is expected to be:

$$\texttt{some}\{|\texttt{p(x)}|^* \,\|\, \texttt{x} \leftarrow \texttt{bagof}(\texttt{M}_\square(|\texttt{q}_1|^*, |\texttt{q}_2|^*))\}$$

The function bagof is used to ensure that the arrays are merged completely, before the elements of the resulting array are bound to x. Since some is idempotent, one could possibly use a similar function setof; in practice this would make no difference, since there are no duplicate pairs in the resulting array. x is bound to elements of the form $(\texttt{i}, \texttt{e})$ whereas we expect it intuitively to be bound to e. The practical problem that arises is that we would have to introduce different translation rules for the same OQL expression. For instance, $\texttt{exists x in } \texttt{q}_1 : \texttt{q}_2$, would be translated to two different calculus forms, depending on whether $\texttt{q}_1$ is an array or not.

In order to avoid this problem, we define that when the non-homomorphic function $\texttt{bagof}(-)$ (or $\texttt{listof}(-)$) applies to an array, it creates a bag (or list) with elements the values e corresponding to each pair $(\texttt{i}, \texttt{e})$ in the initial array. Hence, in this thesis we adopt the monoid introduced in [FM00] for the manipulation of vectors and we refer to it as array. This monoid is idempotent, but not commutative. By adjusting the definitions of bagof and listof to handle arrays, we manage to manipulate arrays and other collections in a uniform way. Finally, we introduce the function $\texttt{arrayof} : \texttt{list/oset(a)} \rightarrow \texttt{array}$, which takes each element e in position i of its operand, creates a pair $(\texttt{i}, \texttt{e})$ and adds it to the resulting array.

Sections 4.2.1 and 4.2.2 presented the main novel points of the calculus representation adopted in this thesis. As has already been pointed out, calculus forms can easily be transformed into simpler and more efficient forms. The next section presents a set of rules for *normalizing* queries; these rules are used to *unnest* complex queries in a number of examples; finally a normalization algorithm determines the order in which the rules must be applied.

4.3 Normalization of calculus expressions

4.3.1 Normalization Rules

Fegaras and Maier have proposed ten rewrite rules that can be used to transform calculus expressions into canonical forms:

$$\oplus\{\texttt{e} \,\|\, \texttt{u}_1 \leftarrow \texttt{path}_1, \ldots, \texttt{u}_\texttt{n} \leftarrow \texttt{path}_\texttt{n}, \texttt{pred}\}$$

where $\texttt{path}_\texttt{i}$ is a bound variable, the name of a class extent, the name of an object, or an expression $\texttt{path}'\texttt{.name}$. Their set of rewrite rules is extended with new rules that handle the additional features of the current calculus representation.

This section presents the set of normalization rules for calculus expressions and pays particular attention to the rules first introduced in this thesis. In the following rules the letters $\mathtt{M}$ and $\mathtt{N}$ represent monoids.

1. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{zero}[\mathtt{N}], \bar{\mathtt{s}}\} \Rightarrow \mathtt{zero}[\mathtt{M}]$

2. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{unit}[\mathtt{N}](\mathtt{e}'), \bar{\mathtt{s}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \equiv \mathtt{e}', \bar{\mathtt{s}}\}$

3. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{e}_1, \mathtt{e}_2), \bar{\mathtt{s}}\} \Rightarrow$
   $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{e}_1, \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{e}_2, \bar{\mathtt{s}}\})$

   For this transformation to be valid, either $\bar{\mathtt{q}}$ should be empty or $\mathtt{M}$ should be commutative. Say for instance that both $\mathtt{N}$ and $\mathtt{M}$ are list monoids and $\bar{\mathtt{q}}$ evaluates to a list with two elements $[\mathtt{v}_1, \mathtt{v}_2]$. The left part of the rule first joins $\mathtt{v}_1$ with the elements from $\mathtt{merge}[\mathtt{N}](\mathtt{e}_1, \mathtt{e}_2)$ and then joins $\mathtt{v}_2$ with each one of them. The right part of the rule first joins $\mathtt{v}_1$ with the elements from $\mathtt{e}_1$, then $\mathtt{v}_2$ with the elements from $\mathtt{e}_1$, then $\mathtt{v}_1$ with the elements from $\mathtt{e}_2$ and finally $\mathtt{v}_2$ with the elements from $\mathtt{e}_2$. The left part of the rule obviously evaluates to a different list than the right part. This problem does not occur if $\bar{\mathtt{q}}$ is empty. However, for reasons of confluence, i.e. to ensure that the order of applying normalization rules exhaustively does not affect the final result, we only apply this rule when $\mathtt{M}$ is commutative.

4. $(\mathtt{struct}(\mathtt{A}_1 = \mathtt{e}_1, \ldots, \mathtt{A}_n = \mathtt{e}_n)).\mathtt{A}_i \Rightarrow \mathtt{e}_i$

5. $(\mathtt{class\_name}(\mathtt{A}_1 = \mathtt{e}_1, \ldots, \mathtt{A}_n = \mathtt{e}_n)).\mathtt{A}_i \Rightarrow \mathtt{e}_i$

6. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{N}\{\mathtt{e}' \parallel \bar{\mathtt{r}}\}, \bar{\mathtt{s}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \bar{\mathtt{r}}, \mathtt{u} \equiv \mathtt{e}', \bar{\mathtt{s}}\}$

7. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{some}\{\mathtt{pred} \parallel \bar{\mathtt{r}}\}, \bar{\mathtt{s}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \bar{\mathtt{r}}, \mathtt{pred}, \bar{\mathtt{s}}\}$

   Notice that $\mathtt{M}$ should be idempotent. For example, if for some $\mathtt{q}_i$ derived from $\bar{\mathtt{q}}$ there are only two elements $\mathtt{r}_1, \mathtt{r}_2$ from $\bar{\mathtt{r}}$ that satisfy $\mathtt{pred}$, then $\mathtt{q}_i$ will be considered once in the initial expression, but twice in the transformed one. If $\mathtt{M}$ is idempotent then this will make no difference in the final result.

8. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{merge}[\mathtt{some}](\mathtt{p}_1, \mathtt{p}_2), \bar{\mathtt{s}}\} \Rightarrow$
   $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{p}_1, \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{p}_2, \bar{\mathtt{s}}\})$

   For this transformation to be valid $\mathtt{M}$ should be idempotent (as in rule 7).

9. $\mathtt{M}\{\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{r}}\} \parallel \bar{\mathtt{s}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{s}}, \bar{\mathtt{r}}\}$

   The monoid $\mathtt{M}$ should be primitive. If for example $\mathtt{M}$ was a $\mathtt{set}$ monoid, the result of the initial expression would be a set of sets of elements, whilst the result of the transformed one would just be a set of elements.

10. $\mathtt{M}\{\mathtt{merge}[\mathtt{M}](\mathtt{e}_1, \mathtt{e}_2) \parallel \bar{\mathtt{s}}\} \Rightarrow \mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e}_1 \parallel \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e}_2 \parallel \bar{\mathtt{s}}\})$

    For this transformation to be valid M should be commutative and primitive (as in rules 3 and 9 respectively).

11. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \equiv \mathtt{e}', \bar{\mathtt{q}}\} \Rightarrow \mathtt{M}\{\mathtt{e}[\mathtt{e}'/\mathtt{u}] \parallel \bar{\mathtt{p}}, \bar{\mathtt{q}}[\mathtt{e}'/\mathtt{u}]\}$

12. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{merge}[\mathtt{all}](\mathtt{p}_1, \ldots, \mathtt{p}_n), \bar{\mathtt{r}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{p}_1, \ldots, \mathtt{p}_n, \bar{\mathtt{r}}\}$

13. $\mathtt{zero}[\mathtt{all}] \Rightarrow \mathtt{true}$
    $\mathtt{zero}[\mathtt{some}] \Rightarrow \mathtt{false}$
    $\mathtt{zero}[\mathtt{sum}] \Rightarrow 0$

14. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{true}, \bar{\mathtt{r}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \bar{\mathtt{r}}\}$

15. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{false}, \bar{\mathtt{r}}\} \Rightarrow \mathtt{zero}[\mathtt{M}]$

16. $\mathtt{M}\{\mathtt{e} \parallel \} \Rightarrow \mathtt{unit}[\mathtt{M}](\mathtt{e})$, if M is a collection monoid.
    $\mathtt{M}\{\mathtt{e} \parallel \} \Rightarrow \mathtt{e}$, if M is a primitive monoid.

17. $\mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}}, \mathtt{zero}[\mathtt{M}]) \Rightarrow \mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}})$
    $\mathtt{merge}[\mathtt{M}](\mathtt{m}_i) \Rightarrow \mathtt{m}_i$
    $\mathtt{merge}[\mathtt{some}](\bar{\mathtt{m}}, \mathtt{true}) \Rightarrow \mathtt{true}$
    $\mathtt{merge}[\mathtt{all}](\bar{\mathtt{m}}, \mathtt{false}) \Rightarrow \mathtt{false}$
    $\mathtt{merge}[\mathtt{some}](\bar{\mathtt{m}}, \mathtt{false}) \Rightarrow \mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}})$
    $\mathtt{merge}[\mathtt{all}](\bar{\mathtt{m}}, \mathtt{true}) \Rightarrow \mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}})$

18. $\mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}}, \mathtt{merge}[\mathtt{M}](\bar{\mathtt{n}})) \Rightarrow \mathtt{merge}[\mathtt{M}](\bar{\mathtt{m}}, \bar{\mathtt{n}})$

19. $\mathtt{not}(\mathtt{not}\ \mathtt{q}) \Rightarrow \mathtt{q}$

20. $\mathtt{not}\ \mathtt{true} \Rightarrow \mathtt{false}$ and $\mathtt{not}\ \mathtt{false} \Rightarrow \mathtt{true}$.

21. $\mathtt{not}\ (\mathtt{q}_1\ \mathtt{comp\_op}\ \mathtt{q}_2) \Rightarrow \mathtt{q}_1\ \mathtt{comp\_op}'\ \mathtt{q}_2$,
    where $\mathtt{comp\_op}, \mathtt{comp\_op}' \in \{<, >, \leq, \geq, =, \neq\}$ and
    $\forall \mathtt{e}_1, \mathtt{e}_2.((\mathtt{e}_1\ \mathtt{comp\_op}\ \mathtt{e}_2) \parallel (\mathtt{e}_1\ \mathtt{comp\_op}'\ \mathtt{e}_2) = \mathtt{true})$.

22. $\mathtt{not}\ (\mathtt{merge}[\mathtt{all}](\mathtt{p}_1, \ldots, \mathtt{p}_n)) \Rightarrow \mathtt{merge}[\mathtt{some}](\mathtt{not}\ \mathtt{p}_1, \ldots, \mathtt{not}\ \mathtt{p}_n)$
    $\mathtt{not}\ (\mathtt{merge}[\mathtt{some}](\mathtt{p}_1, \ldots, \mathtt{p}_n)) \Rightarrow \mathtt{merge}[\mathtt{all}](\mathtt{not}\ \mathtt{p}_1, \ldots, \mathtt{not}\ \mathtt{p}_n)$

23. $\mathtt{not}\ (\mathtt{all/some}\{\mathtt{pred} \parallel \bar{\mathtt{r}}\}) \Rightarrow \mathtt{some/all}\{\mathtt{not}\ \mathtt{pred} \parallel \bar{\mathtt{r}}\}$

24. $\texttt{bagof}(\texttt{zero}[\texttt{M}]) \Rightarrow \texttt{zero}[\texttt{bag}]$
$\texttt{bagof}(\texttt{unit}[\texttt{M}](\texttt{e})) \Rightarrow \texttt{unit}[\texttt{bag}](\texttt{e})$
$\texttt{bagof}(\texttt{merge}[\texttt{M}](\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})) \Rightarrow \texttt{merge}[\texttt{bag}](\texttt{bagof}(\texttt{e}_1), \ldots, \texttt{bagof}(\texttt{e}_\texttt{n}))$,
if $\texttt{M} \in \{\texttt{bag}, \texttt{list}\}$
$\texttt{bagof}(\texttt{M}\{\texttt{e} \parallel \bar{\texttt{r}}\}) \Rightarrow \texttt{bag}\{\texttt{e} \parallel \bar{\texttt{r}}\}$, if $\texttt{M} \in \{\texttt{bag}, \texttt{list}\}$
$\texttt{bagof}(\texttt{e}) \Rightarrow \texttt{bag}\{\texttt{u} \parallel \texttt{u} \leftarrow \texttt{e}\}$, if e has either of the forms id or $\texttt{e}_1.\texttt{id}$ or $\texttt{e}_1[\texttt{e}_2]$ and e is a list of elements
$\texttt{bagof}(\texttt{e}) \Rightarrow \texttt{e}$, if e has either of the forms id or $\texttt{e}_1.\texttt{id}$ or $\texttt{e}_1[\texttt{e}_2]$ and e is a bag of elements
$\texttt{bagof}(\texttt{bagof}(\texttt{e})) \Rightarrow \texttt{bagof}(\texttt{e})$
$\texttt{bagof}(\texttt{listof}(\texttt{e})) \Rightarrow \texttt{bagof}(\texttt{e})$
$\texttt{bagof}(\texttt{arrayof}(\texttt{e})) \Rightarrow \texttt{bagof}(\texttt{e})$
$\texttt{bagof}(\texttt{orderof}(\texttt{e})) \Rightarrow \texttt{bagof}(\texttt{e})$

25. $\texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{bagof}(\texttt{merge}[\texttt{N}](\texttt{e}_1, \texttt{e}_2)), \bar{\texttt{s}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{merge}[\texttt{N}](\texttt{e}_1, \texttt{e}_2), \bar{\texttt{s}}\}$, if M is idempotent and N is either set or oset.

$\texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{bagof}(\texttt{N}\{\texttt{e}' \parallel \bar{\texttt{r}}\}), \bar{\texttt{s}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{N}\{\texttt{e}' \parallel \bar{\texttt{r}}\}, \bar{\texttt{s}}\}$, if M is idempotent and N is either set or oset.

$\texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{bagof}(\texttt{e}'), \bar{\texttt{s}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{q}}, \texttt{u} \leftarrow \texttt{e}', \bar{\texttt{s}}\}$, if M is idempotent and $\texttt{e}'$ has either of the forms id or $\texttt{e}_1.\texttt{id}$ or $\texttt{e}_1[\texttt{e}_2]$ and $\texttt{e}'$ is a set or oset of elements .

26. $\texttt{listof}(\texttt{zero}[\texttt{M}]) \Rightarrow \texttt{zero}[\texttt{list}]$
$\texttt{listof}(\texttt{unit}[\texttt{M}](\texttt{e})) \Rightarrow \texttt{unit}[\texttt{list}](\texttt{e})$
$\texttt{listof}(\texttt{merge}[\texttt{list}](\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})) \Rightarrow \texttt{merge}[\texttt{list}](\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})$
$\texttt{listof}(\texttt{list}\{\texttt{e} \parallel \bar{\texttt{r}}\}) \Rightarrow \texttt{list}\{\texttt{e} \parallel \bar{\texttt{r}}\}$
$\texttt{listof}(\texttt{e}) \Rightarrow \texttt{e}$, if e has either of the forms id or $\texttt{e}_1.\texttt{id}$ or $\texttt{e}_1[\texttt{e}_2]$ and e is a list of elements
$\texttt{listof}(\texttt{listof}(\texttt{e})) \Rightarrow \texttt{listof}(\texttt{e})$
$\texttt{listof}(\texttt{arrayof}(\texttt{e})) \Rightarrow \texttt{listof}(\texttt{e})$
$\texttt{listof}(\texttt{orderof}(\texttt{e})) \Rightarrow \texttt{orderof}(\texttt{e})$

27. $\texttt{M}\{\texttt{e} \parallel \bar{\texttt{s}}, \texttt{u} \leftarrow \texttt{listof}(\texttt{e}'), \bar{\texttt{r}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{s}}, \texttt{u} \leftarrow \texttt{bagof}(\texttt{e}'), \bar{\texttt{r}}\}$, if $\texttt{M} \in \{\texttt{set}, \texttt{bag}, \texttt{group}\}$ (monoid degeneration)
$\texttt{M}\{\texttt{e} \parallel \bar{\texttt{s}}, \texttt{u} \leftarrow \texttt{orderof}(\texttt{e}'), \bar{\texttt{r}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{s}}, \texttt{u} \leftarrow \texttt{bagof}(\texttt{e}'), \bar{\texttt{r}}\}$, if $\texttt{M} \in \{\texttt{set}, \texttt{bag}, \texttt{group}\}$ (monoid degeneration)

28. $\texttt{M}\{\texttt{e} \parallel \bar{\texttt{p}}, \texttt{x} \leftarrow \texttt{bagof}(\texttt{group}\{\texttt{struct}(\ell_1 = \texttt{e}_1, \ldots, \ell_\texttt{n} = \texttt{e}_\texttt{n}, \texttt{partition} = \texttt{e}') \parallel \bar{\texttt{q}}\}), \bar{\texttt{r}}, \texttt{y} \leftarrow \texttt{x}.\texttt{partition}, \bar{\texttt{s}}\} \Rightarrow \texttt{M}\{\texttt{e} \parallel \bar{\texttt{p}}, \bar{\texttt{q}}, \bar{\texttt{r}}, \texttt{y} \leftarrow \texttt{e}', \bar{\texttt{s}}\}$, if $\texttt{x}, \ell_1, \ldots, \ell_\texttt{n}$ are not free variables in $\texttt{e}, \texttt{r}, \texttt{s}$ and M is commutative, that is $\texttt{M} \in \{\texttt{set}, \texttt{bag}, \texttt{group}\}$.

29. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{x} \leftarrow \mathtt{bagof}(\mathtt{merge}[\mathtt{group}](\mathtt{group}\{\mathtt{struct}(\ell_1 = \mathtt{e}_1, \ldots, \ell_n = \mathtt{e}_n, \mathtt{partition} = \mathtt{e}') \parallel \bar{\mathtt{q}}_1, \mathtt{u} \leftarrow \mathtt{q}_{21}, \bar{\mathtt{q}}_3\},\ \mathtt{group}\{\mathtt{struct}(\ell_1 = \mathtt{e}_1, \ldots, \ell_n = \mathtt{e}_n, \mathtt{partition} = \mathtt{e}') \parallel \bar{\mathtt{q}}_1, \mathtt{u} \leftarrow \mathtt{q}_{22}, \bar{\mathtt{q}}_3\})),$
$\bar{\mathtt{r}}, \mathtt{y} \leftarrow \mathtt{x.partition}, \bar{\mathtt{s}}\} \Rightarrow \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \bar{\mathtt{q}}_1, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{bag}](\mathtt{q}_{21}, \mathtt{q}_{22}), \bar{\mathtt{q}}_3, \bar{\mathtt{r}}, \mathtt{y} \leftarrow \mathtt{e}', \bar{\mathtt{s}}\},$
if $\mathtt{x}, \ell_1, \ldots, \ell_n$ are not free variables in $\mathtt{e}, \mathtt{r}, \mathtt{s}$ and $\mathtt{M}$ is commutative, i.e. $\mathtt{M} \in \{\mathtt{set}, \mathtt{bag}, \mathtt{group}\}$.

30. $\mathtt{elementof}(\mathtt{unit}[\mathtt{M}](\mathtt{e})) = \mathtt{e}$
$\mathtt{elementof}(\mathtt{set/bag/list/array}(\mathtt{e})) = \mathtt{e}$

The rules presented above contain eight of the ten transformations proposed by Fegaras and Maier (1,2,3,4,6,7,9,11). One of the omitted rules concerns the beta reduction: $(\lambda \mathtt{u}.\mathtt{e}_1)\mathtt{e} \Rightarrow \mathtt{e}_1[\mathtt{e}/\mathtt{u}]$. In this context, method application is denoted by using the name of a method followed by an optional set of arguments. The second transformation that has been excluded from the normalization rules is:

$$\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow (\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3), \bar{\mathtt{s}}\} \Rightarrow$$
$$\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{e}_1, \mathtt{u} \leftarrow \mathtt{e}_2, \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{not}(\mathtt{e}_1), \mathtt{u} \leftarrow \mathtt{e}_3, \bar{\mathtt{s}}\}$$

This rule is valid if $\mathtt{M}$ is commutative or $\bar{\mathtt{q}}$ is empty. There is no equivalent of this rule, since ODMG OQL statements do not use or imply the `if` statement.

4.3.2 Examples of transforming calculus expressions

We give an example that applies the normalization rules for the optimization of a query with an `order by` clause. Assume that the initial OQL query ($\mathtt{Q}_1$) is:

```
order_flatten
 (select q
 from q1 as x1, ..., qk as xk
 where q'
 order by q''1, ..., q''n)
```

This would be translated to the following monoid calculus form:

$$\begin{aligned}&\mathtt{list}\{\mathtt{u} \parallel \mathtt{v}_1 \leftarrow \mathtt{list}\{\mathtt{v}_2.\mathtt{partition} | \mathtt{v}_2 \leftarrow \mathtt{orderof}(\mathtt{group}\{\mathtt{struct}(\\ &\quad \mathtt{asc}_1 = |\mathtt{q}''_1|^*, \ldots, \mathtt{asc}_n = |\mathtt{q}''_n|^*, \mathtt{partition} = \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*)) \parallel \\ &\quad \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*), \ldots, \mathtt{x}_k \leftarrow \mathtt{bagof}(|\mathtt{q}_k|^*), |\mathtt{q}'|^*\})\}, \mathtt{u} \leftarrow \mathtt{listof}(\mathtt{v}_1)\}\end{aligned} \tag{4.2}$$

Based on transformations 6 and 11, expression 4.2 is converted to the following form:

$$\begin{aligned}&\mathtt{list}\{\mathtt{u} \parallel \mathtt{v}_2 \leftarrow \mathtt{orderof}(\mathtt{group}\{\mathtt{struct}(\mathtt{asc}_1 = |\mathtt{q}_1''|^*, \ldots,\\ &\quad \mathtt{asc}_\mathtt{n} = |\mathtt{q}_\mathtt{n}''|^*, \mathtt{partition} = \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*)) \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*),\\ &\quad\quad \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*\}), \mathtt{u} \leftarrow \mathtt{listof}(\mathtt{v}_2.\mathtt{partition})\}\end{aligned} \tag{4.3}$$

Consider the query $\mathtt{Q}_2$:

$$\begin{aligned}&\mathtt{select}\ \mathtt{x}\\ &\mathtt{from}\ \mathtt{Q}_1\ \mathtt{as}\ \mathtt{x}\end{aligned}$$

Based on 4.3, $\mathtt{Q}_2$ takes the following calculus form:

$$\begin{aligned}&\mathtt{bag}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \mathtt{bagof}(\mathtt{list}\{\mathtt{u} \parallel \mathtt{v}_2 \leftarrow \mathtt{orderof}(\mathtt{group}\{\mathtt{struct}(\mathtt{asc}_1 = |\mathtt{q}_1''|^*,\\ &\quad \ldots, \mathtt{asc}_\mathtt{n} = |\mathtt{q}_\mathtt{n}''|^*, \mathtt{partition} = \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*)) \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*),\\ &\quad\quad \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*\}), \mathtt{u} \leftarrow \mathtt{listof}(\mathtt{v}_2.\mathtt{partition})\})\}\end{aligned} \tag{4.4}$$

After applying the transformations 6, 11 and 24, expression 4.4 becomes:

$$\begin{aligned}&\mathtt{bag}\{\mathtt{u} \parallel \mathtt{v}_2 \leftarrow \mathtt{orderof}(\mathtt{group}\{\mathtt{struct}(\mathtt{asc}_1 = |\mathtt{q}_1''|^*, \ldots,\\ &\quad \mathtt{asc}_\mathtt{n} = |\mathtt{q}_\mathtt{n}''|^*, \mathtt{partition} = \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*)) \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*),\\ &\quad\quad \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*\}), \mathtt{u} \leftarrow \mathtt{listof}(\mathtt{v}_2.\mathtt{partition})\}\end{aligned} \tag{4.5}$$

Expression 4.5 is transformed through 27 and 24 into 4.6:

$$\begin{aligned}&\mathtt{bag}\{\mathtt{u} \parallel \mathtt{v}_2 \leftarrow \mathtt{bagof}(\mathtt{group}\{\mathtt{struct}(\mathtt{asc}_1 = |\mathtt{q}_1''|^*, \ldots,\\ &\quad \mathtt{asc}_\mathtt{n} = |\mathtt{q}_\mathtt{n}''|^*, \mathtt{partition} = \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*)) \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*),\\ &\quad\quad \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*\}), \mathtt{u} \leftarrow \mathtt{v}_2.\mathtt{partition}\}\end{aligned} \tag{4.6}$$

This is the point at which transformation 28 is useful. In a commutative monoid comprehension, if a group monoid forms partitions that are used only to be unnested in later generators, then group may be replaced by bag. Grouping is avoided in the first place. Thus, expression 4.6 is transformed to the simplified form:

$$\mathtt{bag}\{\mathtt{u} \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*), \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*, \mathtt{u} \leftarrow \mathtt{unit}[\mathtt{bag}](|\mathtt{q}|^*))\} \tag{4.7}$$

Rules 2 and 11 finally enable the transformation of 4.7 into the final form 4.8.

$$\mathtt{bag}\{|\mathtt{q}|^* \parallel \mathtt{x}_1 \leftarrow \mathtt{bagof}(|\mathtt{q}_1|^*), \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(|\mathtt{q}_\mathtt{k}|^*), |\mathtt{q}'|^*\} \tag{4.8}$$

The normalized calculus form (expression 4.8) of query $\mathtt{Q}_2$ is far simpler than the initial expression 4.4.

```
case Query of
                ...  :  ...
            not e    :  normalize_not
                ...  :  ...
 M{head ‖ q1,...,qn}  :  normalize_compr
                ...  :  ...
```

Figure 4.3: Normalization Algorithm

4.3.3 Normalization Algorithm

In order to transform a calculus expression into a simpler form, it is not sufficient to know the rules that guide the transformations. It is important to determine the order in which the rules must be triggered. The rules that are eligible to be applied depend on the structure of the query. The normalization algorithm consists of a single *case* step that considers all possible constructs of OQL (see figure 4.3).

Figure 4.4 presents the normalization algorithm for the construct `not (e)`.

Most of the other OQL constructs are normalized in a similar way. Only the case of monoid comprehension expressions is slightly more complicated and therefore needs to be explained in detail (see figure 4.5).

`optimize_from_qualifier(i)` optimizes the comprehension starting from qualifier q_i to the last qualifier and then to the head of the comprehension. The transformations applicable to each qualifier depend on whether the latter is a filter, a generator, or a binding. We study these cases separately:

- Suppose qualifier q_i is a generator $u \leftarrow e$. The rules that may be applied are 1, 2, 3, 6, 25, 27, 28 and 29. The algorithm that normalizes a comprehension from its generator in position `i` is presented in 4.6.
- Suppose qualifier q_i is a filter `e`. The rules that may be applied are 7, 8, 12, 14 and 15. The algorithm that normalizes a comprehension from its filter in position `i` is presented in figure 4.7.
- Suppose qualifier q_i is a binding $u \equiv e$. The only applicable rule is 11. The algorithm that normalizes a comprehension from its binding in position `i` is given in figure 4.8.

When all qualifiers have been considered the algorithm invokes a process to optimize the head of the monoid comprehension. The applicable rules in this case are 9, 10 and 16. Details of the algorithm are given in figure 4.9.

```
normalize_not (Rules 19,20,21,22,23)

 Input: not(e)
 Algorithm

 opt_e =optimize e
 case opt_e has the construct
                    not pred  :  apply rule 19
                  true, false :  apply rule 20
          e1 < comp_oper > e2 :  apply rule 21
     merge[all/some](p1,...,pn) :  apply rule 22
           all/some{pred || ...} :  apply rule 23

 if any rule was applied
    optimize the result of the transformation
    return the result of the optimization
 else
    annotate not (opt_e) as normalized
    return not (opt_e)
```

Figure 4.4: Normalization of the OQL construct not (e)

```
normalize compr
(Rules 1,2,3,6,7,8,9,10,11,12,14,15,16,25,27,28,29)

 Input: M{head || q1,...,qn}
 Algorithm

 if the comprehension has qualifiers (n > 0)
    optimize_from_qualifier(1)
 else
    optimize_head
 annotate result as normalized and return it
```

Figure 4.5: Normalization of monoid comprehensions

```
if e is annotated as normalized
   opt_e = e
else
   opt_e = optimize e
case opt_e has the construct
                  zero[N]  :  apply rule 1;
               unit[N](e1) :  apply rule 2;
                              optimize_from_qualifier(i);
           merge[N](e1, e2) :  try rule 3
                              if rule 3 was applicable
                               optimize its result;
              N{e' || r} :  :  apply rule 6;
                              optimize_from_qualifier(i);
  bagof(merge[N](e1,...,en)) :  try rule 25;
                              if rule 25 was applicable
                               optimize_from_qualifier(i);
     bagof(set/oset{e' || r}) :  try rule 25;
                              if rule 25 was applicable
                               optimize_from_qualifier(i);
        listof/orderof(e) :  try rule 27;
                              if rule 27 was applicable
                               optimize_from_qualifier(i);
         bagof(group{...}) :  try rule 28;
                              if rule 28 was applicable
                               optimize_from_qualifier(i);
    bagof(merge[group](...)) :  try rule 29;
                              if rule 29 was applicable
                               optimize_from_qualifier(i);
 if no transformation was applied in the case clause
    optimize_from_qualifier(i + 1)
    (or optimize head if i + 1 > n)
```

Figure 4.6: Normalizing comprehensions from qualifier q_i, where q_i is a generator ($u \leftarrow e$)

```
if e is annotated as normalized
   opt_e = e
else
   opt_e = optimize e
case opt_e has the construct
            true  :  apply rule 14;
                     optimize_from_qualifier(i);
           false  :  apply rule 15; optimize result;
     some{...|...}  :  try rule 7;
                     if rule 7 was applicable
                      optimize_from_qualifier(i);
   merge[some](...)  :  try rule 8;
                     if rule 8 was applicable
                      optimize its result;
    merge[all](...)  :  apply rule 12;
                     optimize_from_qualifier(i);
if no transformation was applied in the case clause
   optimize_from_qualifier(i + 1)
   (or optimize_head if i + 1 > n)
```

Figure 4.7: Normalizing comprehensions from qualifier q_i, where q_i is a filter (e)

```
if e is annotated as normalized
   opt_e = e
else
   opt_e = optimize e
apply rule 11
optimize_from_qualifier(i)
```

Figure 4.8: Normalizing comprehensions from qualifier q_i, where q_i is a binding ($u \equiv e$)

```
if e is annotated as normalized,
   opt_head = head
else
   opt_head = optimize head
if opt_head has the form M{...} (where M is a primitive monoid)
   apply rule 9
else if opt_head has the form merge[M](...)
(where M is a primitive and commutative monoid)
   apply rule 10
else if there are no qualifiers
   apply rule 16
if any transformation was applied
   optimize its result
   return optimized result
else
   return the comprehension as optimized so far
```

Figure 4.9: Normalizing the head of a comprehension

The normalization algorithm has the following properties: i) it always terminates; ii) it yields the same result as the exhaustive iterative application of rules and iii) it yields this result in the most efficient way.

We first provide an informal proof that the algorithm always terminates. Normalization rules perform either of the following tasks.

- They push up merge operations (Rules 3, 8, 10, 18 and 22). Neither of the remaining rules pushes down nor creates new merge operators. Hence, rules 3, 8, 10, 18 and 22 are applicable only a finite number of times.

- They unnest inner comprehensions pushing up their head and qualifiers (Rules 6, 7 and 9). The number of inner comprehensions can only be increased finitely by either of the following rules

  1. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{merge}[\mathtt{some}](\mathtt{p}_1, \mathtt{p}_2), \bar{\mathtt{s}}\} \Rightarrow$
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{p}_1, \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{p}_2, \bar{\mathtt{s}}\})$

     $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{e}_1, \mathtt{e}_2), \bar{\mathtt{s}}\} \Rightarrow$
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{e}_1, \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{q}}, \mathtt{u} \leftarrow \mathtt{e}_2, \bar{\mathtt{s}}\})$

$\mathtt{M}\{\mathtt{merge}[\mathtt{M}](\mathtt{e}_1, \mathtt{e}_2) \parallel \bar{\mathtt{s}}\} \Rightarrow \mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e}_1 \parallel \bar{\mathtt{s}}\}, \mathtt{M}\{\mathtt{e}_2 \parallel \bar{\mathtt{s}}\})$

In this case, the number of additional comprehensions is bounded by the initial number of `merge` operators.

2. $\mathtt{bagof}(\mathtt{e}) \Rightarrow \mathtt{bag}\{\mathtt{u} \parallel \mathtt{u} \leftarrow \mathtt{e}\}$, if $\mathtt{e}$ has either of the forms `id` or $\mathtt{e}_1.\mathtt{id}$ or $\mathtt{e}_1[\mathtt{e}_2]$ and $\mathtt{e}$ is a list of elements. Based on this rule, the number of comprehensions can be increased at most by the initial number of nonhomomorphic functions that could be transformed to `bagof`.

3. $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \equiv \mathtt{e}', \bar{\mathtt{q}}\} \Rightarrow \mathtt{M}\{\mathtt{e}[\mathtt{e}'/\mathtt{u}] \parallel \bar{\mathtt{p}}, \bar{\mathtt{q}}[\mathtt{e}'/\mathtt{u}]\}$, where $\mathtt{e}$ is a comprehension. In this case, the number of comprehensions can be increased at most by the number of occurences of range variables in the tree. This is finite since the number of range variables is increased only when pushing up `merge` operators.

- They eliminate, push down or simplify nonhomomorphic functions (Rules 24, 25, 26, 27 and 30). These rules are applicable a finite number of times, since the number of nonhomomorphic functions is finite and the depth of the tree increases finitely using rules 3, 8 and 11.

- They avoid unnecessary grouping by pushing up the qualifiers of nested `group` comprehensions (Rules 28 and 29).

- They push down not operators (Rules 19, 20, 21, 22 and 23).

- They transform calculus expressions to much simpler forms (Rules 1, 2, 4, 5, 12, 13, 14, 15, 16, 17), e.g. by pushing down the zero element (Rule 1).

There is no rule that performing one of these actions causes the reverse effect of another. Hence, since each one can be applied a finite number of times, the algorithm will eventually not be able to apply any of the rules and will terminate. The same applies for the exhaustive iterative application of rules.

We show that our algorithm is also exhaustive in the sense that when it terminates no normalization rule is applicable in any of the subexpressions. The algorithm can be viewed as a Depth-First unnesting algorithm where the children of a node are first normalized, before the node itself is considered. In a monoid comprehension, each child (qualifier) is normalized as an independent expression first, and then with respect to the outer comprehension (parent node). If normalization of a qualifier with respect to the outer comprehension alters only the qualifier, the qualifier is renormalized. If no transformation takes place, the following qualifier is visited. Finally, if the normalization process results in a completely different outer expression, which is no longer a comprehension, the new expression is normalized from the beginning. Hence, the next qualifiers are considered after the current

one, or all qualifiers are visited as subexpressions of a different calculus expression. In this way, all subexpressions are eventually normalized.

The question that arises is whether the actual order of applying rules affects the result of the exhaustive iterative algorithm. If we proved that the exhaustive application of rules (in a random iterative way) is confluent, we could deduce that our algorithm also yields the same result as the random iterative process. The reason is that since our algorithm is also exhaustive, it could be viewed as one of the possible executions of the random iterative process.

All rules (except for Rule 11[2]) modify at most the first two levels of a subtree (subexpression). Hence, if two transformations do not occur close to each other, the order in which they are applied does not affect the final result. However, there are two cases in which applying one rule might affect the result of the other, or might even hide the opportunity to apply the second rule.

1. If the possible transformations occur in two consecutive edges in a path from the root to a leaf.
2. If the possible transformations occur in two sibling edges, i.e. in two edges starting from the same node.

To prove confluence, we need to consider applying all possible pairs of rules in each one of the cases above. For space reasons, we only study the application of one pair in each case.

1. Consider the calculus expression $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{q}, \mathtt{merge}[\mathtt{N}](\mathtt{r}, \mathtt{s})), \bar{\mathtt{t}}\}$. If M is commutative then both rules 3 and 18 are applicable in two consecutive edges of the calculus tree.

   - If we applied rule 3 first then the expression would be transformed to:
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{q}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{r}, \mathtt{s}), \bar{\mathtt{t}}\}$.
     Another application of rule 3 would then yield the expression:
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{q}, \bar{\mathtt{t}}\}, \mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{r}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{s}, \bar{\mathtt{t}}\}))$.
     By applying rule 12, we would get the final result:
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{q}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{r}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{s}, \bar{\mathtt{t}}\})$.
   - If we applied rule 18 first then the expression would be transformed to the following form:
     $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{q}, \mathtt{r}, \mathtt{s}), \bar{\mathtt{t}}\}$.
     Using rule 3 we would then get the same final result:
     $\mathtt{merge}[\mathtt{M}](\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{q}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{r}, \bar{\mathtt{t}}\}, \mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{s}, \bar{\mathtt{t}}\})$.

2. Consider the calculus expression $\mathtt{M}\{\mathtt{e} \parallel \bar{\mathtt{p}}, \mathtt{u} \leftarrow \mathtt{merge}[\mathtt{N}](\mathtt{q}, \mathtt{r}), \mathtt{v} \leftarrow \mathtt{zero}[\mathtt{M}]\}$. In this case we can apply either of the rules 1 or 3 on two sibling edges.

[2] Rule 11 is just a replacement of a variable by a subexpression and the order in which it is applied should not matter as long as the order in which the remaining rules are applied does not alter the final result.

- If rule 1 was applied first then the result would be $\texttt{zero}[\texttt{M}]$.
- If rule 3 was applied first then the expression would take the form:
  $\texttt{merge}[\texttt{M}](\texttt{M}\{\texttt{e} \parallel \bar{\texttt{p}}, \texttt{u} \leftarrow \texttt{q}, \texttt{v} \leftarrow \texttt{zero}[\texttt{M}]\}, \texttt{M}\{\texttt{e} \parallel \bar{\texttt{p}}, \texttt{u} \leftarrow \texttt{r}, \texttt{v} \leftarrow \texttt{zero}[\texttt{M}]\})$.
  If any of the possible rule sequences $[1, 1, 17, 17]$ or $[1, 17, 1, 17]$ were then applied the result would be $\texttt{zero}[\texttt{M}]$.

Since the result does not depend on the order in which we apply rules on consecutive, sibling or isolated edges, the random iterative application of rules is confluent and yields the same result as our algorithm. Confluence is proven with the aid of a tree representation of rule applications.

Assume that $\texttt{S}_\texttt{i} = [\texttt{r}_{\texttt{i1}}, \ldots, \texttt{r}_{\texttt{in}_\texttt{i}}]$ represents a sequence of rule applications on a calculus expression $\texttt{e}$, such that, after it is used, no other rule is applicable. If $\texttt{S} = \{\texttt{S}_1, \ldots, \texttt{S}_\texttt{m}\}$ is the set of all possible sequences applicable to $\texttt{e}$, $\texttt{S}$ could be represented as a tree. The head of the tree is the expression $\texttt{e}$, the leaves represent the final forms of $\texttt{e}$ and each sequence $\texttt{S}_\texttt{i}$ is a complete path from the head ($\texttt{e}$) to the $\texttt{i}^{\texttt{th}}$ final form of $\texttt{e}$. The internal nodes represent intermediate forms of $\texttt{e}$, whilst each edge can be viewed as a rule application.

We navigate the tree upwards starting from internal nodes $\texttt{e}_{\texttt{1i}}$, $\texttt{i} = 1, \ldots, \texttt{k}_1$, whose direct or indirect children are either leaves or nodes with one child. All leaves under such an intermediate expression $\texttt{e}_{\texttt{1i}}$ should have the same form; the reason is that no matter which rule was applied first to $\texttt{e}_{\texttt{1i}}$, we maintain the possibility of getting the same result (proof about consecutive, sibling or isolated transformations). We then consider internal nodes $\texttt{e}_{\texttt{2i}}$, $\texttt{i} = 1, \ldots, \texttt{k}_2$, whose children are either leaves or internal nodes with one child or nodes in $\{\texttt{e}_{11}, \ldots, \texttt{e}_{1\texttt{k}_1}\}$. In the same way as before, we prove that the leaves of each one of these nodes $\texttt{e}_{\texttt{2i}}$ have the same form. The same procedure continues until we reach the root of the tree, which also has the same property. Hence, all possible exhaustive sequences of rule applications finally yield the same result. We conclude that the random iterative process is confluent and uields the same result as our algorithm.

When a calculus expressions is normalized (using our algorithm), it is annotated as such, and needs not be considered again. In addition to that, we consider normalization of monoid comprehensions, every time we visit a subexpression (qualifier) and thus we identify normalization opportunities as early as possible. Another approach would be to normalize all qualifiers first, before proceeding to consider the outer comprehension. However, this might burden the normalization process with redundant steps. By annotating subexpressions as normalized and unnesting expressions as early as possible, the algorithm normalizes a calculus expression in the least number of steps. Our algorithm triggers the execution of one of the optimal sequences of rules that could possibly be executed based on a random exhaustive process.

In this section, we provided a complete framework for normalizing OQL queries. This framework has been implemented and has been tested for a series of complex OQL queries.

4.4 Discussion

This chapter gave an overview of the first intermediate representation of OQL, the monoid calculus. In particular, it presented how OQL queries are translated into calculus expressions, and how the latter are transformed into more efficient forms. This framework is based on work by Afshar [Afs98] and Fegaras and Maier [FM00]. However, it has extended the latter with new monoids, several different translation rules, new normalization rules, and an algorithm that determines the order of applying these rules. It has also pointed out several weak points of the ODMG standard and has proposed ways of resolving them.

Chapter 5

OQL algebraic representation

The previous chapter described how OQL queries are translated into the monoid calculus representation; it also presented several transformations within this representation. This chapter presents the second intermediate representation of OQL queries, referred to as the object algebra. We do not use monoid homomorphisms as the basis of this algebra, since they are difficult to optimize and their natural implementation involves inefficient nested loops. Instead, we adopt a set of algebraic operators for two reasons: i) these operators are easy to rearrange and ii) they can be mapped into efficient physical operators used broadly in the relational systems.

The algebraic operators are similar to those proposed by Fegaras and Maier [FM00]. However, the mechanism for translating queries from calculus to algebraic form is significantly different. This chapter presents the translation algorithm in detail, highlights its advantages and gives several examples that demonstrate its functionality.

5.1 Algebraic Operators

We give an overview of the algebraic operators and explain their functionality. First, we present those operators that have also been defined in [FM00] and point out subtle differences in the two approaches. We then define a set of algebraic operators that are novel in this dissertation and explain their role in the translation process. The complete syntax of the algebraic representation of OQL is given in Figure 5.1. In particular, `Group A` (figure 5.1) contains the expressions that have a very similar calculus equivalent, whilst `Group B` consists of the algebraic operators described below.

A subtle point in which the current algebraic operators differ from the corresponding ones in [FM00] is the specification of the additional information `mon` (monoid). In general, operators iterate over the elements of a collection, process them in some way, and accumulate the results. The monoid field specifies the kind of merge operation used to accumulate the results. Fegaras and Maier specify

$$
\begin{array}{lll}
\texttt{e} & ::= & /* \texttt{ Group A } */ \\
 & & < \texttt{literal} > \ (\texttt{integer}, \texttt{float}, \texttt{boolean}, \texttt{char}, \texttt{string}) \\
 & | & < \texttt{id} > \ (\texttt{db object/extent name}) \\
 & | & \texttt{zero}[< \texttt{monoid} >] \\
 & | & \texttt{unit}[< \texttt{monoid} >](\texttt{e}) \\
 & | & \texttt{merge}[< \texttt{prim\_mon} >](\texttt{e}_1, \ldots, \texttt{e}_n) \\
 & | & \texttt{struct}(\texttt{l}_1 = \texttt{e}_1, \ldots, \texttt{l}_k = \texttt{e}_k) \\
 & | & \texttt{class\_name}(\texttt{l}_1 = \texttt{e}_1, \ldots, \texttt{l}_k = \texttt{e}_k) \\
 & | & \texttt{e}_1 \ < \texttt{comp\_op} > \ \texttt{e}_2 \\
 & | & \texttt{e}_1 \ < \texttt{arithm\_op} > \ \texttt{e}_2 \\
 & | & \texttt{e}_1.\texttt{attr\_rltp\_method} \\
 & | & \texttt{e}(\texttt{e}_1, \ldots, \texttt{e}_n) \\
 & | & \texttt{e}_1[\texttt{e}_2] \\
 & | & \texttt{not}\ (\texttt{e}) \\
 & | & -\texttt{e} \\
 & & /* \texttt{ Group B } */ \\
 & | & \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}}{\pi}}\ \overset{\texttt{projection}}{\texttt{operand}} \\
 & | & \texttt{operand}_1 \ \overset{\texttt{mon}}{\underset{\overline{\texttt{v}_1},\overline{\texttt{v}_2}\ \texttt{predicate}}{\bowtie}}\ \texttt{operand}_2 \\
 & | & \texttt{operand}_1 \ \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}_1},\overline{\texttt{v}_2}\ \texttt{predicate}}{=\!\!\bowtie}}\ \texttt{operand}_2 \\
 & | & \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}}{\Gamma}}\ \overset{\texttt{gmon}}{\underset{\texttt{predicate}}{\texttt{gname}}}\ \overset{\texttt{gproject}}{\underset{\overline{\texttt{gv}}}{}}\ \texttt{operand} \\
 & | & \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}}{\sigma}}\ \overset{\texttt{predicate}}{\texttt{operand}} \quad | \quad \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}}}{=\!\!\sigma}}\ \overset{\overline{\texttt{ov}}}{\texttt{predicate}}\ \texttt{operand} \\
 & | & \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}\ \texttt{v}'}{\mu}}\ \overset{\texttt{path}}{\underset{\texttt{pred}}{}}\ \texttt{operand} \quad | \quad \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}}\ \texttt{v}'}{=\!\!\mu}}\ \overset{\texttt{path}}{\underset{\texttt{pred}}{}}\ \texttt{operand} \\
 & | & \overset{\texttt{mon}}{M}\ \overset{\texttt{mon}'}{(\texttt{e}_1, \ldots, \texttt{e}_n)} \quad | \quad \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}\ \texttt{v}'}{=\!\!M}}\ \overset{\texttt{mon}'}{\underset{\texttt{parts}}{}}\ \texttt{operand} \\
 & | & \overset{\texttt{mon}}{B}\ \overset{\texttt{function}}{\texttt{operand}} \quad | \quad \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}\ \texttt{v}'}{=\!\!B}}\ \overset{\texttt{function}}{\underset{\texttt{argument}}{}}\ \texttt{operand}
\end{array}
$$

Figure 5.1: Syntax of the algebraic OQL representation

this information (accumulator) only for the Reduce and Nest operators. We claim that it would be equally useful for the other operators.

First, it would enable the optimization of the execution plan, as will be explained later on. Second, it may reduce the number of operators in the algebraic plan. For instance, to filter the elements of a bag into a set, based on a predicate, one would need a Select operator to apply the predicate, and a Reduce operator to transform the bag into a set. In our context, it suffices to use a Select operator with monoid (set).

In the remainder of this section we present the operators in detail.

1. $\overset{\texttt{mon}\ \ \texttt{projection}}{\underset{\overline{\texttt{v}}}{\pi}}\ \texttt{operand}$

   Project : This operator iterates over the elements of operand. It binds (the parts of) each element to the variables $\overline{\texttt{v}}$ and calculates the expression projection. It corresponds to the following calculus expression:

   $$\texttt{mon}\{\texttt{projection}|\texttt{u} \leftarrow \texttt{operand}, \texttt{v}_1 \equiv \texttt{u}.\texttt{v}_1, \ldots, \texttt{v}_\texttt{n} \equiv \texttt{u}.\texttt{v}_\texttt{n}\}$$

   where $\overline{\texttt{v}} = \{\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}\}$

   The Project operator differs from the similar Reduce operator ([FM00]) in that it considers no predicate. That is the Project operator does not perform any filtering on the elements of operand.

   Hereafter the notation $\overline{\texttt{v}} \leftarrow \texttt{operand}$ is used as a short form for the binding:

   $$(\texttt{u} \leftarrow \texttt{operand}, \texttt{v}_1 \equiv \texttt{u}.\texttt{v}_1, \ldots, \texttt{v}_\texttt{n} \equiv \texttt{u}.\texttt{v}_\texttt{n})$$

2. $\overset{\texttt{mon}\ \ \texttt{predicate}}{\underset{\overline{\texttt{v}}}{\sigma}}\ \texttt{operand}$

   Select : This operator iterates over the operand and filters its elements using predicate. Those elements of operand that satisfy predicate are accumulated using the merge operation of monoid mon. The operator corresponds to the calculus expression:

   $$\texttt{mon}\{\texttt{struct}(\texttt{v}_1 = \texttt{v}_1, \ldots, \texttt{v}_\texttt{n} = \texttt{v}_\texttt{n})|\overline{\texttt{v}} \leftarrow \texttt{operand}, \texttt{predicate}\}$$

   where $\overline{\texttt{v}} = \{\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}\}$.

   The question that arises at this point is how to identify the monoid mon. In the case of Project the monoid is defined from the calculus expression that we translate, in a straightforward way. The same holds for the Nest operator which is defined later on. The following example demonstrates how the monoid information is derived for all other operators, including Select.

Say that the query to be executed is

$$\texttt{exists j in Priority\_List : j.priority} = 6$$

where Priority_List is an ordered collection (list) containing objects of class Job. This query would be translated into the calculus form

$$\texttt{some}\{\texttt{true}|\texttt{j} \leftarrow \texttt{Priority\_List}, \texttt{j.priority} = 6\}$$

One would expect the algebraic equivalent of this expression to include a Select operator that tests the predicate j.priority $= 6$ iterating over the collection Priority_List. If this operator has no knowledge about the monoid information, it respects the ordered nature of the operand and therefore it filters its elements in the prespecified order. However, knowing that the result monoid of the comprehension (some) is a commutative monoid, it may consider Priority_List as a bag, and rearrange its elements or make use of indices in an optimal way for the filtering. In fact, since some is an idempotent monoid, the operator could even decide to remove duplicates from Priority_List before filtering its elements. Thus, the physical implementation of Select could benefit from the information mon $=$ set/bag. Whilst optimizing the algebraic plan, it is possible to refine the monoid information of several operators, e.g. from set/bag to set, instead of leaving this decision to the physical implementation.

The example illustrated how the monoid information of Select is derived by looking at the result monoid of the monoid comprehension. The following table presents this relation in detail.

Comprehension Monoid	Operator Monoid
$<$ IC monoid $>$	set/bag
$<$ C monoid $>$	bag
$<$ I monoid $>$	oset/list
$<$ other $>$	list

where,

- IC is a notation for monoids both idempotent and commutative.
- C stands for idempotent but not commutative monoids.
- I stands for commutative but not idempotent monoids.
- other is the category for monoids that are neither commutative nor idempotent.

The table above is used to derive the monoid information for all algebraic operators, excluding Project and Nest.

3. $\underset{\overline{\mathtt{v}}\ \mathtt{v}'}{\overset{\mathtt{mon}}{\mu}}\ \overset{\mathtt{path}}{\underset{\mathtt{pred}}{\mathtt{oper}}}$

   `Unnest` : This operator iterates over the operand `oper` and for each element, it *unfolds* the path expression. Each element of the path expression is bound to the new variable $\mathtt{v}'$. The name of this variable should be different from the names of $\overline{\mathtt{v}}$. Hence the result of this operation is a collection of structures with labels named as the variables in $\overline{\mathtt{v}}$ or as $\mathtt{v}'$. In fact, only the structures that satisfy `pred` are included in the result. The monoid information (`mon`) is inferred in the same way as in the operator `Select`.

   Notice that it is not necessary to include the predicate in the `Unnest` operator. Instead one could use a `Select` operator having `Unnest` as its operand. However, it is a good decision to include a predicate, since `Unnest` often generates a much bigger collection than its operand. Hence, it is worthwhile to prune this collection before sending it to the output. The same applies for the `Join` operator.

   The `Unnest` operator corresponds to the calculus expression:

$$\mathtt{mon}\{\mathtt{struct}(\mathtt{v_1} = \mathtt{v_1}, \ldots, \mathtt{v_n} = \mathtt{v_n}, \mathtt{v}' = \mathtt{v}') | \overline{\mathtt{v}} \leftarrow \mathtt{oper}, \mathtt{v}' \leftarrow \mathtt{path}, \mathtt{pred}\}$$

   where $\overline{\mathtt{v}} = \{\mathtt{v_1}, \ldots, \mathtt{v_n}\}$.

4. $\mathtt{operand_1} \underset{\overline{\mathtt{v_1}},\overline{\mathtt{v_2}}\ \mathtt{predicate}}{\overset{\mathtt{mon}}{\bowtie}} \mathtt{operand_2}$

   `Join` : This operator takes the cartesian product of the two operands and selects those elements that satisfy `predicate`. It accumulates the results using the merge operation of the monoid `mon`. The latter is derived as in the case of the `Select` operator. The physical implementation of `Join` may benefit from the `mon` information. Indeed, if $\mathtt{mon} = \mathtt{set}/\mathtt{bag}$, then even if the operands are both lists, they can be considered as bags or sets. Should they be considered as bags, they can be rearranged (sorted) to facilitate their joining; if they have many duplicates, it is even more efficient to convert them to sets before joining them. In the latter case `mon` should be refined to `set`. If $\mathtt{mon} = \mathtt{list}$, the optimizer can use metainformation regarding the ordering of elements in the operands in order to apply optimization techniques studied in sequence query processing [SLR94].

   The `Join` operator corresponds to the calculus expression:

$$\begin{aligned}&\mathtt{mon}\{\mathtt{struct}(\mathtt{v_{11}} = \mathtt{v_{11}}, \ldots, \mathtt{v_{1n}} = \mathtt{v_{1n}}, \mathtt{v_{21}} = \mathtt{v_{21}}, \ldots, \mathtt{v_{2m}} = \mathtt{v_{2m}}) |\\ &\overline{\mathtt{v_1}} \leftarrow \mathtt{operand_1}, \overline{\mathtt{v_2}} \leftarrow \mathtt{operand_2}, \mathtt{predicate}\}\end{aligned}$$

   where $\overline{\mathtt{v_1}} = \{\mathtt{v_{11}}, \ldots, \mathtt{v_{1n}}\}$ and $\overline{\mathtt{v_2}} = \{\mathtt{v_{21}}, \ldots, \mathtt{v_{2m}}\}$.

5. $\overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}}\ \texttt{v}'}{\Rightarrow\!\!\!\!\!\!\mu}}\ \overset{\texttt{path}}{\underset{\texttt{pred}}{}}\ \texttt{operand}$

   `OuterUnnest` : This operator is used to *unfold* path expressions in nested (inner) comprehensions. Its functionality is similar to `Unnest` with a single important difference. If for an element `e` in `operand`, the collection `path` is empty, or none of its elements satisfy `pred`, then `e` is still included in the result, paired with the value `NULL` [FM00]. There should be no confusion between this `NULL` value and the `UNDEFINED` value; the latter is defined in the ODMG Standard ([CBB+00],chapter 4.6) as the result of accessing a property of the `nil` object. `UNDEFINED` is a valid value for any literal or object type.

   Notice that `OuterUnnest` is annotated with information about two monoids. The first monoid $\texttt{mon}_1$ denotes the `merge` operation used to accumulate the results. The second one $\texttt{mon}_2$ gives information about how `path` should be unnested. For instance, if $\texttt{mon}_1 = \texttt{bag}$ and $\texttt{mon}_2 = \texttt{list}$, then although the result is a `bag`, the elements of collection `path` should be annotated with a serial number, based on their order. Hence, if we need to group these elements later on into a different list, we would be able to know their order despite their being members of a `bag`. Since $\texttt{mon}_1 = \texttt{bag}$, the elements of `operand` should be annotated with information that distinguishes its duplicates (see discussion about dot equality in `Nest`).

   `OuterUnnest` is used to translate path expressions that are generators in nested (inner) comprehensions. Hence, $\texttt{mon}_1$ is derived based on the monoid of the current (outer) comprehension, whilst $\texttt{mon}_2$ is inferred by the monoid of the nested (inner) comprehension.

6. $\texttt{operand}_1\ \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}_1},\overline{\texttt{v}_2}\ \texttt{predicate}}{\Rightarrow\!\!\!\bowtie}}\ \texttt{operand}_2$

   `OuterJoin` : This operator is useful in the case of nested comprehensions. It differs from `Join` in that if an element from $\texttt{operand}_1$ cannot be matched with at least one element from $\texttt{operand}_2$, it is preserved in the result by being paired with the `NULL` value.

   Notice that `OuterJoin` is annotated with information about two monoids. The first monoid $\texttt{mon}_1$ denotes the `merge` operation used to accumulate the results. The second one $\texttt{mon}_2$ gives information about the nature of $\texttt{operand}_2$. They are derived in a similar way as in the case of `OuterUnnest`.

7. $\overset{\texttt{mon}}{\underset{\overline{\texttt{v}}}{\Gamma}}\ \overset{\texttt{gmon}}{\underset{\texttt{predicate}}{\overset{\texttt{gname}}{}}}\ \overset{\texttt{gproject}}{\overline{\texttt{gv}}}\ \texttt{operand}$

   `Nest` : This operator groups elements based on certain grouping variables ($\overline{\texttt{gv}}$), or to be accurate, on dot equality between the values of these variables [FM00]. This operator handles the translation of inner monoid comprehensions. We give an example that shows the importance of dot equality in the context of the `Nest` operator.

Consider the translation of the following monoid calculus expression:
$\texttt{bag}\{\texttt{x}|\texttt{x} \leftarrow \texttt{X}, \texttt{all}\{\texttt{y.a}|\texttt{y} \leftarrow \texttt{Y}, \texttt{p}(\texttt{x},\texttt{y})\}\}$

The translation algorithm from calculus to algebra first creates an `OuterJoin` operator, to join `X` and `Y` based on $\texttt{p}(\texttt{x},\texttt{y})$. The reason why it would not use a `Join` operation is that `Y` is in an inner comprehension wrt `X`. For each `x` in `X` the algorithm then gathers (through the monoid `all`) all the values `y.a`. This is done using the `Nest` operation with:

- `mon` (monoid): bag
- `operand`: $\texttt{X} \overset{\texttt{bag set/bag}}{\underset{\texttt{x,y p(x,y)}}{=\!\bowtie}} \texttt{Y}$
- `gmon` (group monoid): `all`
- `gname` (group name): `new_var`
- `gproject` (group project): `y.a`
- $\overline{\texttt{gv}}$ (grouping variables): `x`
- $\overline{\texttt{v}}$ (vars): `x`, `y`
- `predicate`: `true`

There is a subtle point here that needs special attention. If there are two pairs $(\texttt{x}_1, \texttt{y}_1),\ (\texttt{x}_2, \texttt{y}_2)$ in the `OuterJoin` result, such that $\texttt{x}_1 = \texttt{x}_2$, they may have been derived in two ways: i) by combining the same element `x` in `X` with two different elements $\texttt{y}_1, \texttt{y}_2$ in `Y`; ii) by combining two different elements $\texttt{x}_1$ and $\texttt{x}_2$ in `X` (s.t. $\texttt{x}_1 = \texttt{x}_2$) with the elements $\texttt{y}_1, \texttt{y}_2$ respectively. The `Nest` operator groups together two (or more) different values $\texttt{y}_1\texttt{.a}, \texttt{y}_2\texttt{.a}$ if they correspond to $\texttt{x}_1, \texttt{x}_2$, such that $\texttt{x}_1 = \texttt{x}_2$ and $\texttt{x}_1, \texttt{x}_2$ were the same element `x` in `X`. If both conditions are satisfied then dot equality holds between $\texttt{x}_1$ and $\texttt{x}_2$ ($\texttt{x}_1 \doteq \texttt{x}_2$) [FM00]. The values `y.a` that correspond to the same `x` (dot equality wise) are merged using $\wedge$, i.e. the merge operation of the group monoid `all`. In fact, they are merged only if they satisfy the predicate of the `Nest` operation.

The result of `Nest` is a collection of structures with labels the grouping variables $\{\texttt{x}\}$ and the group name `new_var`. The `OuterJoin` operator between `X` and `Y` has the monoid information `mon` = `bag` and the same applies for the `Nest` operator.

The operators defined so far are similar to those proposed by Fegaras and Maier. The next three operators are new in our context.

8. $\overset{\texttt{mon}_1\ \texttt{mon}_2}{=\!\sigma_{\overline{\texttt{v}}}}\ \overset{\overline{\texttt{ov}}}{\texttt{predicate}}\ \texttt{operand}$

`OuterSelect` : This operator is similar to `Select`. The only difference is that if `predicate` is not satisfied, then the values of the outer variables $\overline{\mathtt{ov}}$ are preserved; they are appended with `NULL` values for the local variables in $(\overline{\mathtt{v}} - \overline{\mathtt{ov}})$. $\mathtt{mon}_1$ indicates the way elements of the outer comprehension are accumulated, whilst $\mathtt{mon}_2$ corresponds to the local (inner) comprehension. The usefulness of this operator will become evident in the rules for the translation of calculus to algebraic expressions.

9. $\overset{\mathtt{mon}}{M}\ \underset{(\mathtt{e}_1,\ldots,\mathtt{e}_n)}{\overset{\mathtt{mon}'}{}}$

`Merge` : This operator merges the operands $\mathtt{e}_1, \ldots, \mathtt{e}_n$ using the `merge` operation of $\mathtt{mon}'$. The results are accumulated using the `merge` operation of `mon`. For instance, say we have the monoid comprehension $\mathtt{set}\{\mathtt{e}(\mathtt{x}) | \mathtt{x} \leftarrow \mathtt{merge}[\mathtt{bag}](\mathtt{X}_1, \mathtt{X}_2), \ldots\}$, where both $\mathtt{X}_1$ and $\mathtt{X}_2$ are bags. The algebraic expression corresponding to the first generator is:

$$\overset{\mathtt{set/bag}}{M}\ \underset{(\mathtt{X}_1,\mathtt{X}_2)}{\overset{\mathtt{bag}}{}}$$

10. $\underset{\overline{\mathtt{v}}\ \mathtt{v}'}{\overset{\mathtt{mon}}{I\!M}}\ \underset{\mathtt{operand}}{\overset{\mathtt{mon}'}{\mathtt{parts}}}$

`NestedMerge` : This operator iterates over the `operand` and for each element, it merges the `parts` using the merge operation of monoid $\mathtt{mon}'$. The result of merging them is bound to the new variable $\mathtt{v}'$. Each structure in `operand` is extended with the new label $\mathtt{v}'$. The resulting structures are accumulated using the merge operation of `mon`.

11. $\overset{\mathtt{mon}}{B}\ \underset{\mathtt{operand}}{\overset{\mathtt{function}}{}}$

`Bulk` : This operator applies `function`, e.g. `listof`, `bagof`, `arrayof` or `orderof`, to `operand`. It accumulates the elements of the resulting collection using the `merge` operation of `mon`. This operator applies non-homomorphic bulk functions. For instance, if `operand` is a set of elements then the function $\mathtt{bagof}(\mathtt{operand})$ waits until the input is complete (all the elements have been received) and generates a bag containing these elements, but without any duplicates. `Bulk` operators are expected to create bottlenecks in any execution plan based on a pipeline mechanism.

12. $\underset{\overline{\mathtt{v}}\ \mathtt{v}'}{\overset{\mathtt{mon}}{I\!B}}\ \underset{\mathtt{operand}}{\overset{\mathtt{function}\ \mathtt{argument}}{}}$

`NestedBulk` : This operator is similar to `Bulk`. However, instead of applying `function` to `operand`, it applies it to the collection `argument`, which is evaluated for each element in

Monoid Calculus Form	Algebraic Form
$< \texttt{literal} >$	$< \texttt{literal} >$
$\texttt{zero}[< \texttt{monoid} >]$	$\texttt{zero}[< \texttt{monoid} >]$
$\texttt{unit}[< \texttt{monoid} >](\texttt{e})$	$\texttt{unit}[< \texttt{monoid} >](\lvert\texttt{e}\rvert^{\#})$
$\texttt{merge}[< \texttt{prim\_mon} >](\texttt{e}_1, \ldots, \texttt{e}_\texttt{n})$	$\texttt{merge}[< \texttt{prim\_mon} >](\lvert\texttt{e}_1\rvert^{\#}, \ldots, \lvert\texttt{e}_\texttt{n}\rvert^{\#})$
$\texttt{struct}(\texttt{l}_1 = \texttt{e}_1, \ldots, \texttt{l}_\texttt{k} = \texttt{e}_\texttt{k})$	$\texttt{struct}(\texttt{l}_1 = \lvert\texttt{e}_1\rvert^{\#}, \ldots, \texttt{l}_\texttt{k} = \lvert\texttt{e}_\texttt{k}\rvert^{\#})$
$\texttt{C}(\texttt{l}_1 = \texttt{e}_1, \ldots, \texttt{l}_\texttt{k} = \texttt{e}_\texttt{k})$	$\texttt{C}(\texttt{l}_1 = \lvert\texttt{e}_1\rvert^{\#}, \ldots, \texttt{l}_\texttt{k} = \lvert\texttt{e}_\texttt{k}\rvert^{\#})$
$\texttt{e}_1 \ < \texttt{comp\_op} > \ \texttt{e}_2$	$\lvert\texttt{e}_1\rvert^{\#} \ < \texttt{comp\_op} > \ \lvert\texttt{e}_2\rvert^{\#}$
$\texttt{e}_1 \ < \texttt{arithm\_op} > \ \texttt{e}_2$	$\lvert\texttt{e}_1\rvert^{\#} \ < \texttt{arithm\_op} > \ \lvert\texttt{e}_2\rvert^{\#}$
$\texttt{e.attr\_rltp\_method}$	$\lvert\texttt{e}\rvert^{\#}.\texttt{attr\_rltp\_method}$
$\texttt{e}(\texttt{e}_1, \ldots, \texttt{e}_\texttt{k})$	$\lvert\texttt{e}\rvert^{\#}(\lvert\texttt{e}_1\rvert^{\#}, \ldots, \lvert\texttt{e}_\texttt{k}\rvert^{\#})$
$\texttt{e}_1[\texttt{e}_2]$	$\lvert\texttt{e}_1\rvert^{\#}[\lvert\texttt{e}_2\rvert^{\#}]$
$\texttt{not}\ (\texttt{e})$	$\texttt{not}\ (\lvert\texttt{e}\rvert^{\#})$
$-\texttt{e}$	$-\lvert\texttt{e}\rvert^{\#}$

Figure 5.2: Translation of calculus to algebraic constructs in Group A (figure 5.1)

operand. Each element in operand is extended with the resulting collection value of the bulk function, which is bound to the variable $\texttt{v}'$. The resulting structures are accumulated using the merge operation of mon.

In this section, we presented the syntax of the algebraic representation of OQL. We reviewed the algebraic operators proposed by Fegaras and Maier, proposed slight changes and introduced five new operators. We are now in a position to discuss the translation process from the monoid calculus to the algebraic level.

5.2 Translating monoid calculus to algebraic expressions

The translation of calculus expressions that have a similar algebraic equivalent (in Group A of figure 5.1) is straightforward. All we do is replace the constituent parts of these expressions by their algebraic counterparts. This is shown in table 5.2. The function $|-|^{\#} : \texttt{calculus} \rightarrow \texttt{algebra}$ is the generic translation algorithm that takes a monoid calculus expression and converts it to an algebraic form. Its implementation depends on the calculus expression.

The remaining calculus forms to be considered are the nonhomomorphic functions bagof, listof, arrayof and orderof, as well as the merge operation and monoid comprehensions. If the argument of a nonhomomorphic function (e.g. bagof) has no free variables, its algebraic equivalent is derived

as follows:

$$|\texttt{function(e)}|^{\#} = \overset{\texttt{mon}}{B}\ \overset{\texttt{function}}{\texttt{e}}$$

where $\texttt{function} \in \{\texttt{bagof}, \texttt{listof}, \texttt{arrayof}, \texttt{orderof}\}$ and mon is the monoid corresponding to the comprehension which has function(e) as a generator.

Likewise, if the arguments of a merge calculus expression have no free variables, the expression is translated to the algebraic form:

$$|\texttt{merge}[< \texttt{coll\_mon} >](\texttt{e}_1, \ldots, \texttt{e}_n)|^{\#} = \overset{\texttt{mon}}{M}\ \overset{\texttt{coll\_mon}}{|\texttt{e}_1|^{\#}, \ldots, |\texttt{e}_n|^{\#}}$$

where mon is derived as explained in the case of function(e), if merge is a generator. Otherwise $\texttt{mon} = \texttt{coll\_mon}$.

If the calculus expressions translated so far are parts of a comprehension and not all of their variables are free, then their translation is not so straightforward. It is tackled below in the algorithm for the translation of monoid comprehensions.

5.2.1 Translation of Monoid Comprehensions

Intuitively, the algorithm builds an algebraic expression by traversing the qualifiers first and the head afterwards. Initially, the algebraic expression is null. As the qualifiers are processed, it gradually grows bigger, until it takes its final form at the point it *absorbs* the head. Before giving a formal description of the algorithm we demonstrate its functionality through a simple example:

$$(\texttt{set}\{\texttt{x.address}|\texttt{x} \leftarrow \texttt{People}, \texttt{x.name} = \texttt{"Clare"}\})\ () \Rightarrow$$

$$(\texttt{set}\{\texttt{x.address}|\ \}_{\texttt{x}})\ \left(\overset{\texttt{set/bag}}{\underset{\texttt{x}}{\sigma}}\ \overset{\texttt{x.name="Clare"}}{(\texttt{People})} \right) \Rightarrow$$

$$()\ \left(\overset{\texttt{set}}{\underset{\texttt{x}}{\pi}}\ \overset{\texttt{x.address}}{\left(\overset{\texttt{set/bag}}{\underset{\texttt{x}}{\sigma}}\ \overset{\texttt{x.name="Clare"}}{(\texttt{People})} \right)} \right)$$

As we process the qualifiers and the head of the comprehension, we distinguish between the range variables that have been encountered in an outer comprehension, and those bound in previous generators of the same comprehension. The former are called outer_vars whilst the latter local_vars. Likewise, we distinguish the algebraic query produced so far by the qualifiers of the outer comprehensions, from the one generated by the previous qualifiers of the same comprehension. The first is called

outer_query and the second local_query. This is a characteristic point at which our algorithm differs from that of Fegaras and Maier. Their algorithm gradually develops a single outer_query, while we insist on maintaining a local context of the nested comprehension too. Later on, we will explain the implications of the two different approaches. At certain critical points, the local_query is merged with the outer_query. The local_query becomes empty and the local_vars are included in the outer_vars. However, we keep storing the initial *outer* variables in a vector called ext_vars. It is worth noting that if a predicate refers to a combination of outer_vars and local_vars, we defer its evaluation until the local- and outer- queries are merged. Meanwhile, we maintain it in a vector named predicates.

The input of the translation algorithm consists of the outer_query already formed and the outer_vars encountered. Obviously, if the comprehension is an external one, both input values are null, empty respectively. The algorithm has a naturally functional implementation, going along with recursive descent of a calculus syntax tree. In the remainder of this section, we present a set of rules that formally define the translation algorithm for monoid comprehensions. These rules are preceded by the necessary notation.

Notation

Each side of the rules has the following constituent elements:

- the ext_vars (external variables), i.e. the variables from outer comprehensions,
- a monoid calculus expression (which has not been converted yet to its algebraic representation),
- the local_query already formed,
- the local_vars already encountered,
- the outer_query already formed,
- the outer_vars already encountered,
- the set of predicates that combine local and outer range variables, but have not been absorbed into either of the algebraic queries (local_query or outer_query).

Thus, each side of a rule has the form:

$$< \texttt{ext\_vars} > < \texttt{calculus} > \begin{matrix} < \texttt{local\_query} > & < \texttt{outer\_query} > \\ < \texttt{local\_vars} > & < \texttt{outer\_vars} > \end{matrix} \texttt{predicates}$$

In the monoid calculus expressions, on the bottom of each qualifier, we denote its dependency wrt the outer and the local variables. If all variables in the qualifier are bound, then $\texttt{dep} = \texttt{null}$. If the

free variables are all included in the local_vars, then dep = local. However, if any of its free variables is included in the outer_vars, then dep = outer.

In the translation rules from calculus to algebra we broadly use the function $\mathcal{S}$. This function applies on a pair of queries, one algebraic and one calculus-like. It has a different implementation depending on the calculus (the second) query. Its goal is to simplify the calculus expression, i.e. to remove any nested queries from it, by incorporating their algebraic equivalent to the initial algebraic query.

We only use this function when some of the free variables of the calculus expression are in the range variables over the algebraic query. Hence, $\mathcal{S}$ takes three input parameters: i) an algebraic expression (AE), ii) a calculus one (CE) and iii) the range variables over the algebraic expression ($\overline{\mathtt{v}}$). It returns an extended algebraic expression (AE′), a simplified calculus one (CE′) and an updated set of range variables ($\overline{\mathtt{v}}'$) over AE′.

$$\mathcal{S}\left\{ \begin{array}{c} \mathtt{AE} \\ \mathtt{CE} \end{array} \right\}_{\overline{\mathtt{v}}} = \left\{ \begin{array}{c} \mathtt{AE}' \\ \mathtt{CE}' \end{array} \right\}_{\overline{\mathtt{v}}'}$$

If expression CE is a comprehension, the extended algebraic query AE′ is derived by calling the function $|-|^{\#}$ on the comprehension with $\mathtt{outer\_vars} = \overline{\mathtt{v}}$ and $\mathtt{outer\_query} = \mathtt{AE}$. The resulting algebraic query will be a Nest operator and the resulting variables $\overline{\mathtt{v}}' = \overline{\mathtt{v}} \cup \{\mathtt{g}\}$, where g is the name of the group (gname) in the Nest operator. Also, $\mathtt{CE}' = \mathtt{g}$. This will become evident to the reader after studying the translation rules in subsections 5.2.1, 5.2.1 and 5.2.1.

If the only expression that allowed nested queries was the monoid comprehension form, then we would not need to introduce the $\mathcal{S}$ function. All we would need is the recursive application of the translation rules. However, in our context, we need to simplify all kinds of nested queries, beyond monoid comprehensions. Figure 5.3 shows how $\mathcal{S}$ works for various calculus expressions.

We are now in a position to present the rules for the translation of monoid comprehensions to their algebraic forms.

Generators without any dependency from local or outer variables

$$\begin{array}{r} < \mathtt{ext\_vars}_1 > < \mathtt{M}\{\mathtt{head}|\mathtt{u} \underset{\mathtt{dep=null}}{\leftarrow} \mathtt{e}, \overline{\mathtt{r}}, \mathtt{p}\} > \begin{array}{cc} < \mathtt{loc\_query}_1 > & < \mathtt{out\_query} > \\ < \mathtt{loc\_vars}_1 > & < \mathtt{out\_vars} > \end{array} \mathtt{pred}_1 \\ \Rightarrow < \mathtt{ext\_vars}_2 > < \mathtt{M}\{\mathtt{head}|\overline{\mathtt{r}}, \mathtt{p}[\hat{\mathtt{u}}]\} > \begin{array}{cc} < \mathtt{loc\_query}_2 > & < \mathtt{out\_query} > \\ < \mathtt{loc\_vars}_2 > & < \mathtt{out\_vars} > \end{array} \mathtt{pred}_2 \end{array}$$

where

- p is the calculus expression in which all filters of the comprehension have been assembled, $\mathtt{p}[\hat{\mathtt{u}}]$ is what remains from p after removing the parts of it that refer to u only (denoted as

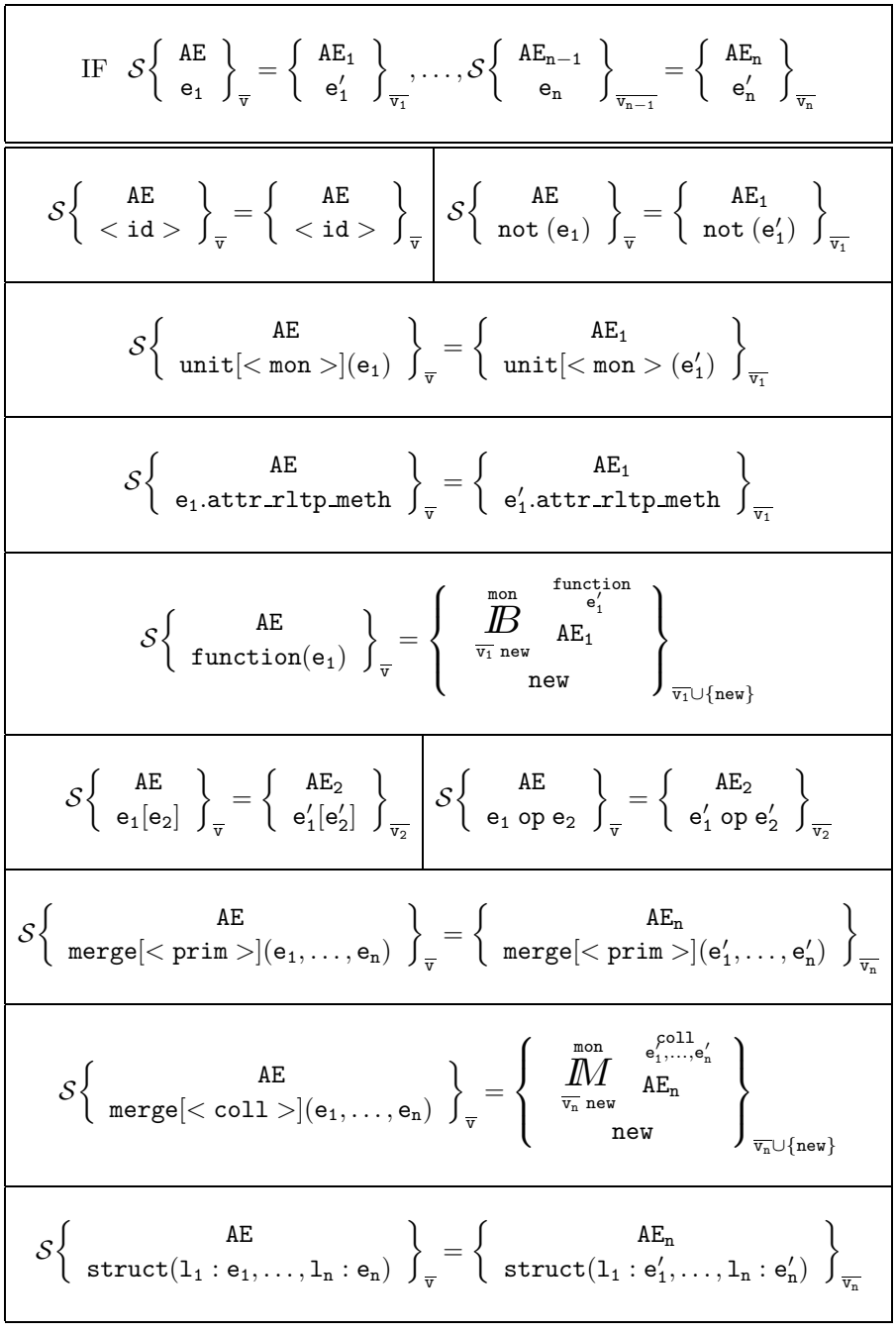

Figure 5.3: Implementing $\mathcal{S}$ for various calculus constructs

$\mathtt{p}[\mathtt{u}]$), or to u and any of the $\mathtt{loc\_vars}_1$ (denoted as $\mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1]$), or to u and any of the $(\mathtt{loc\_vars}_1 \cup \mathtt{out\_vars})$ (denoted as $\mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1, \mathtt{out\_vars}]$).

- if there exists $\mathtt{p}[\mathtt{u}]$ such that $\mathtt{p}[\mathtt{u}] \neq \mathtt{true}$ then

  – $\mathtt{gener\_query} = \overset{\mathtt{mon}}{\underset{\overline{\mathtt{v}}}{\sigma}}\ ^{\mathtt{simpl\_p[u]}}\ \mathtt{e}'$

  where $\mathcal{S}\left\{ \begin{array}{c} |\mathtt{e}|^{\#} \\ \mathtt{p}[\mathtt{u}] \end{array} \right\}_{\{\mathtt{u}\}} = \left\{ \begin{array}{c} \mathtt{e}' \\ \mathtt{simpl\_p[u]} \end{array} \right\}_{\overline{\mathtt{v}}}$

  otherwise,

  – $\mathtt{gener\_query} = |\mathtt{e}|^{\#}$

- $\mathtt{pred}_2 = \mathtt{pred}_1 \cup \mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1, \mathtt{outer\_vars}]$

- $\mathtt{loc\_vars}_2 = \mathtt{loc\_vars}_1 \cup \{\mathtt{u}\} = \overline{\mathtt{v}_1}$

- if $\mathtt{loc\_query}_1 \neq \mathtt{null}$ then

  – if there exists $\mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1] \neq \mathtt{true}$ then

$$\mathcal{S}\left\{ \begin{array}{c} \mathtt{loc\_query}_1 \underset{\overline{\mathtt{v}_1}, \mathtt{true}}{\overset{\mathtt{mon}}{\bowtie}} \mathtt{gener\_query} \\ \mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1] \end{array} \right\}_{\overline{\mathtt{v}_1}} = \left\{ \begin{array}{c} \mathtt{alg\_query}_1 \\ \mathtt{simpl\_pred}_1 \end{array} \right\}_{\overline{\mathtt{v}_2}}$$

$$\mathtt{loc\_query}_2 = \overset{\mathtt{mon}}{\underset{\overline{\mathtt{v}_2}}{\sigma}}\ ^{\mathtt{simpl\_pred}_1}\ \mathtt{alg\_query}_1$$

  – Otherwise, if there is no $\mathtt{p}[\mathtt{u}, \mathtt{loc\_vars}_1]$,

$$\mathtt{loc\_query}_2 = \mathtt{loc\_query}_1 \underset{\overline{\mathtt{v}_1}, \mathtt{true}}{\overset{\mathtt{mon}}{\bowtie}} \mathtt{gener\_query}$$

  Otherwise, if $\mathtt{loc\_query}_1 = \mathtt{null}$, then

  – $\mathtt{loc\_query}_2 = \mathtt{gener\_query}$

- $\mathtt{ext\_vars}_2 = \left\{ \begin{array}{ll} \mathtt{ext\_vars}_1, & \mathtt{if\ ext\_vars}_1 \neq \mathtt{empty\ vector} \\ \mathtt{out\_vars}, & \mathtt{if\ ext\_vars}_1 = \mathtt{empty\ vector} \end{array} \right.$

Generators dependent on local (but not outer) variables

$$< \texttt{ext\_vars}_1 > < \texttt{M}\{\texttt{head}|\texttt{u} \underset{\texttt{dep=local}}{\leftarrow} \texttt{e}, \overline{\texttt{r}}, \texttt{p}\} > \begin{matrix} < \texttt{loc\_query}_1 > & < \texttt{out\_query} > \\ < \texttt{loc\_vars}_1 > & < \texttt{out\_vars} > \end{matrix} \texttt{pred}_1$$

$$\Rightarrow \; < \texttt{ext\_vars}_2 > < \texttt{M}\{\texttt{head}|\overline{\texttt{r}}, \texttt{p}[\hat{\texttt{u}}]\} > \begin{matrix} < \texttt{loc\_query}_2 > & < \texttt{out\_query} > \\ < \texttt{loc\_vars}_2 > & < \texttt{out\_vars} > \end{matrix} \texttt{pred}_2$$

where

- $\mathcal{S}\left\{ \begin{matrix} \texttt{loc\_query}_1 \\ \texttt{e} \end{matrix} \right\}_{\texttt{loc\_vars}_1} = \left\{ \begin{matrix} \texttt{alg\_query}_1 \\ \texttt{e}' \end{matrix} \right\}_{\overline{\texttt{v}}}$
- $\texttt{alg\_query}_2 = \overset{\texttt{mon}}{\underset{\overline{\texttt{v}}\;\texttt{u}}{\mu}} \overset{\texttt{e}'}{\underset{\texttt{true}}{}} \texttt{alg\_query}_1$
- Say $\texttt{p}[\widehat{\texttt{out\_vars}}] = \texttt{merge}[\texttt{all}](\texttt{p}[\texttt{u}], \texttt{p}[\texttt{u}, \texttt{loc\_vars}_1])$.

  If $\mathcal{S}\left\{ \begin{matrix} \texttt{alg\_query}_2 \\ \texttt{p}[\widehat{\texttt{out\_vars}}] \end{matrix} \right\}_{\overline{\texttt{v}} \cup \{\texttt{u}\}} = \left\{ \begin{matrix} \texttt{alg\_query}_3 \\ \texttt{simpl\_p}[\widehat{\texttt{out\_vars}}] \end{matrix} \right\}_{\overline{\texttt{v}_1}}$

  then $\texttt{loc\_query}_2 = \overset{\texttt{mon}}{\underset{\overline{\texttt{v}_1}}{\sigma}} \overset{\texttt{simpl\_p}[\widehat{\texttt{out\_vars}}]}{} \texttt{alg\_query}_3$
- $\texttt{loc\_vars}_2 = \overline{\texttt{v}_1}$
- $\texttt{pred}_2 = \texttt{pred}_1 \cup \texttt{p}[\texttt{u}, \texttt{out\_vars}]$
- $\texttt{ext\_vars}_2 = \begin{cases} \texttt{ext\_vars}_1, & \texttt{if ext\_vars}_1 \neq \texttt{empty vector} \\ \texttt{outer\_vars}, & \texttt{if ext\_vars}_1 = \texttt{empty vector} \end{cases}$

Generators dependent (among others) on outer variables

$$< \texttt{ext\_vars}_1 > < \texttt{M}\{\texttt{head}|\texttt{u} \underset{\texttt{dep=outer}}{\leftarrow} \texttt{e}, \overline{\texttt{r}}, \texttt{p}\} > \begin{matrix} < \texttt{loc\_query}_1 > & < \texttt{out\_query}_1 > \\ < \texttt{loc\_vars}_1 > & < \texttt{out\_vars}_1 > \end{matrix} \texttt{pred}_1$$

$$\Rightarrow \; < \texttt{ext\_vars}_2 > < \texttt{M}\{\texttt{head}|\overline{\texttt{r}}, \texttt{p}[\hat{\texttt{u}}]\} > \begin{matrix} < \texttt{loc\_query}_2 > & < \texttt{out\_query}_2 > \\ < \texttt{loc\_vars}_2 > & < \texttt{out\_vars}_2 > \end{matrix} \texttt{pred}_2$$

where

- If $\texttt{loc\_query}_1 \neq \texttt{null}$ then

  $$- \; \mathcal{S}\left\{ \begin{matrix} \texttt{out\_query}_1 \overset{\texttt{mon}_1\;\texttt{mon}_2}{\underset{\overline{\texttt{v}},\;\texttt{true}}{\Longrightarrow\!\!\bowtie}} \texttt{loc\_query}_1 \\ \texttt{pred}_1 \end{matrix} \right\}_{\overline{\texttt{v}}} = \left\{ \begin{matrix} \texttt{alg\_query} \\ \texttt{simpl\_pred}_1 \end{matrix} \right\}_{\overline{\texttt{v}_1}}$$

  where $\overline{\texttt{v}} = \texttt{out\_vars}_1 \cup \texttt{loc\_vars}_1$

– $\texttt{alg\_query}_1 = \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}_1}}{=\!=\!\sigma}}\ \overset{\texttt{ext\_vars}_1}{\texttt{simpl\_pred}_1}\ \texttt{alg\_query}$

Otherwise,

– $\texttt{alg\_query}_1 = \texttt{out\_query}_1$ and $\overline{\texttt{v}_1} = \texttt{out\_vars}_1$.

- Say $\mathcal{S}\left\{ \begin{array}{c} \texttt{alg\_query}_1 \\ \texttt{e} \end{array} \right\}_{\overline{\texttt{v}_1}} = \left\{ \begin{array}{c} \texttt{alg\_query}_2 \\ \texttt{simpl\_e} \end{array} \right\}_{\overline{\texttt{v}_2}}$ and

  $\texttt{alg\_query}_3 = \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}_2}\ \texttt{u}}{=\!=\!\mu}}\ \overset{\texttt{simpl\_e}}{\texttt{true}}\ \texttt{alg\_query}_2$

- Let $\texttt{preds\_on\_u} = \texttt{merge}[\texttt{all}](\texttt{p}[\texttt{u}], \texttt{p}[\texttt{u}, \texttt{loc\_vars}_1], \texttt{p}[\texttt{u}, \texttt{loc\_vars}_1, \texttt{out\_vars}_1])$.

$$\mathcal{S}\left\{ \begin{array}{c} \texttt{alg\_query}_3 \\ \texttt{preds\_on\_u} \end{array} \right\}_{\overline{\texttt{v}_2}\cup\{\texttt{u}\}} = \left\{ \begin{array}{c} \texttt{alg\_query}_4 \\ \texttt{simpl\_p} \end{array} \right\}_{\overline{\texttt{v}_3}}$$

- $\texttt{out\_query}_2 = \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}_3}}{=\!=\!\sigma}}\ \overset{\texttt{ext\_vars}_2}{\texttt{simpl\_p}}\ \texttt{alg\_query}_4$
- $\texttt{out\_vars}_2 = \overline{\texttt{v}_3}$
- $\texttt{loc\_vars}_2 = \{\}$
- $\texttt{loc\_query}_2 = \texttt{null}$
- $\texttt{pred}_2 = \{\}$
- $\texttt{ext\_vars}_2 = \begin{cases} \texttt{ext\_vars}_1, & \text{if } \texttt{ext\_vars}_1 \neq \text{empty vector} \\ \texttt{out\_vars}, & \text{if } \texttt{ext\_vars}_1 = \text{empty vector} \end{cases}$

Absorbing remaining predicates and head of the comprehension

$$\begin{array}{rl} < \texttt{ext\_vars}_1 > < \texttt{M}\{\texttt{head}|\} > & \begin{array}{cc} < \texttt{loc\_query}_1 > & < \texttt{out\_query}_1 > \\ < \texttt{loc\_vars}_1 > & < \texttt{out\_vars}_1 > \end{array}\ \texttt{pred}_1 \\ \Rightarrow\ < \texttt{ext\_vars}_2 > <> & \begin{array}{cc} < \texttt{loc\_query}_2 > & < \texttt{out\_query}_2 > \\ < \texttt{loc\_vars}_2 > & < \texttt{out\_vars}_2 > \end{array}\ \texttt{pred}_2 \end{array}$$

where

- If $\texttt{loc\_query}_1 \neq \texttt{null}$ and $\texttt{out\_query}_1 \neq \texttt{null}$ then

$$\mathcal{S}\left\{ \texttt{out\_query}_1\ \overset{\texttt{mon}_1\ \texttt{mon}_2}{\underset{\overline{\texttt{v}},\ \texttt{true}}{=\!=\!\bowtie}}\ \texttt{loc\_query}_1 \right\}_{\overline{\texttt{v}}} = \left\{ \begin{array}{c} \texttt{alg\_query}_1 \\ \texttt{simpl\_pred}_1 \end{array} \right\}_{\overline{\texttt{v}_1}}$$

where $\overline{\mathtt{v}} = \mathtt{out\_vars}_1 \cup \mathtt{loc\_vars}_1$.

Otherwise, if $\mathtt{out\_query}_1 \neq \mathtt{null}$
$\mathtt{alg\_query}_1 = \mathtt{out\_query}_1$ and $\overline{\mathtt{v_1}} = \mathtt{out\_vars}_1$

Otherwise, if $\mathtt{loc\_vars}_1 \neq \mathtt{null}$
$\mathtt{alg\_query}_1 = \mathtt{loc\_query}_1$ and $\overline{\mathtt{v_1}} = \mathtt{loc\_vars}_1$.

- If $\mathtt{pred}_1 \neq \{\}$
  - $\mathtt{alg\_query}_2 = \overset{\mathtt{mon}_1\ \mathtt{mon}_2}{\underset{\overline{\mathtt{v_1}}}{=\!\!\sigma}}\ \overset{\mathtt{ext\_vars}_1}{\mathtt{simpl\_pred}_1}\ \mathtt{alg\_query}_1$

  Otherwise,

  - $\mathtt{alg\_query}_2 = \mathtt{alg\_query}_1$.
- $\mathcal{S}\left\{ \begin{array}{c} \mathtt{alg\_query}_2 \\ \mathtt{head} \end{array} \right\}_{\overline{\mathtt{v_1}}} = \left\{ \begin{array}{c} \mathtt{alg\_query}_3 \\ \mathtt{simpl\_head} \end{array} \right\}_{\overline{\mathtt{v_2}}}$

- If $\mathtt{ext\_vars}_1 = \{\}$ then the resulting algebraic expression is:
  - $\underset{\overline{\mathtt{v_2}}}{\overset{\mathtt{M}}{\pi}}\ \overset{\mathtt{simpl\_head}}{\mathtt{alg\_query}_2}$

  Otherwise (if $\mathtt{ext\_vars}_1 \neq \{\}$), the resulting algebraic expression takes the following form:
  - $\underset{\overline{\mathtt{v_2}}}{\overset{\mathtt{mon}}{\Gamma}}\ \overset{\mathtt{M}\quad \mathtt{simpl\_head}}{\mathtt{group\_id}\quad \mathtt{ext\_vars}_1}\ \underset{\mathtt{null}}{\quad}\ \mathtt{alg\_query}_2$

5.3 Benefits of the algebraic representation: Examples

This section presents the advantages of this algebraic framework with respect to the one proposed by Fegaras and Maier [FM00]. A number of translation examples are given to show in practice where the novel points of the current approach are applicable.

1. This framework addresses the non-homomorphic transformations involved in OQL queries, e.g. `bagof` and `listof`. To do that, it introduces the operators `Bulk` and `NestedBulk`.

Consider, for instance the following OQL query:

$$\begin{array}{l}\texttt{select x.C} + \texttt{y.D}\\ \texttt{from X as x,}\\ \quad (\texttt{select distinct z.A}\\ \qquad \texttt{from x.B as z}) \texttt{ as y}\end{array} \tag{5.1}$$

If X is a bag or list and x.B is a set, bag, list, or oset ($\texttt{x} \in \texttt{X}$), the query above corresponds to the following calculus expression:

$$\texttt{bag}\{\texttt{x.C} + \texttt{y.D} | \texttt{x} \leftarrow \texttt{X}, \texttt{y} \leftarrow \texttt{bagof}(\texttt{set}\{\texttt{z.A} | \texttt{z} \leftarrow \texttt{x.B}\})\}$$

The calculus expression above translates to the algebraic form:

$$\overset{\texttt{bag}}{\underset{\texttt{x,y}}{\pi}}{}^{\texttt{x.C+y.D}} \overset{\texttt{bag}}{\underset{\texttt{x,g'\ y}}{\mu}}{}^{\texttt{g'}}_{\texttt{true}} \overset{\texttt{bag}}{\underset{\texttt{x,g\ g'}}{\not\!B}}{}^{\texttt{bagof}}_{\texttt{g}} \overset{\texttt{bag}}{\underset{\texttt{x,z}}{\Gamma}}{}^{\texttt{set}}_{\texttt{g}}{}^{\texttt{z.A}}_{\texttt{x}} {}_{\texttt{true}} \left(\overset{\texttt{bag set/bag}}{\underset{\texttt{x z}}{=\!\!\not\mu}}{}^{\texttt{x.B}}_{\texttt{true}} (\texttt{X}) \right)$$

2. Nested comprehensions are processed locally as much as possible, before being merged with the algebraic expressions generated in outer comprehensions. This leaves space for further algebraic optimization.

   Consider, for example, the query:

$$\begin{array}{l}\texttt{select x.B}\\ \texttt{from X as x}\\ \texttt{where}\\ \quad \texttt{min}\,(\texttt{select } \texttt{y}_1\texttt{.A from } \texttt{Y}_1 \texttt{ as } \texttt{y}_1\\ \qquad \texttt{where exists } \texttt{y}_2 \texttt{ in } \texttt{Y}_2 : \ \texttt{y}_1\texttt{.C}_1 < \texttt{y}_2\texttt{.C}_2\\ \qquad \texttt{and x.C} < \texttt{y}_2\texttt{.C}_2) \ > \ 4\end{array} \tag{5.2}$$

   If $\texttt{Y}_1$ and $\texttt{Y}_2$ are bags, the normalized calculus representation of the previous query is:

$$\texttt{bag}\{\texttt{x.B} | \texttt{x} \leftarrow \texttt{X}, \texttt{min}\{\texttt{y}_1\texttt{.A} | \texttt{y}_1 \leftarrow \texttt{Y}_1, \texttt{y}_2 \leftarrow \texttt{Y}_2, \texttt{y}_1\texttt{.C}_1 < \texttt{y}_2\texttt{.C}_2, \texttt{x.C} < \texttt{y}_2\texttt{.C}_2\} > 4\}$$

   This calculus expression takes the algebraic form:

$$\overset{\texttt{bag}}{\underset{\texttt{x,g}}{\pi}}{}^{\texttt{x.B}} \overset{\texttt{bag}}{\underset{\texttt{x,g}}{\sigma}}{}^{\texttt{g>4}} \overset{\texttt{bag}}{\underset{\texttt{x,y}_1\texttt{,y}_2}{\Gamma}}{}^{\texttt{min}}_{\texttt{g}}{}^{\texttt{y}_1\texttt{.A}}_{\texttt{x}} {}_{\texttt{true}} \left(\texttt{X} \overset{\texttt{bag set/bag}}{\underset{\texttt{x,y}_1\texttt{,y}_2\ \texttt{x.C<y}_2\texttt{.C}_2}{=\!\!\bowtie}} \left(\texttt{Y}_1 \overset{\texttt{set/bag}}{\underset{\texttt{y}_1\texttt{,y}_2\ \texttt{y}_1\texttt{.C}_1\texttt{<y}_2\texttt{.C}_2}{\bowtie}} \texttt{Y}_2 \right) \right)$$

Based on the translation rules by Fegaras and Maier [FM00], the sub-expression

$$\mathtt{E} = \mathtt{X} \overset{\mathtt{bag\ set/bag}}{\underset{\mathtt{x,y_1.C_1,y_2.C_2\ x.C<y_2.C_2}}{\Longrightarrow\!\bowtie}} \left(\mathtt{Y_1} \overset{\mathtt{set/bag}}{\underset{\mathtt{y_1,y_2\ y_1<y_2}}{\bowtie}} \mathtt{Y_2} \right)$$

would be replaced by:

$$\mathtt{E'} = \left(\mathtt{X} \underset{\mathtt{x,y_1\ true}}{\Longrightarrow\!\bowtie} \mathtt{Y_1} \right) \underset{\mathtt{x,y_1,y_2\ x.C<y_2.C_2 \wedge y_1.C_1<y_2.C_2}}{\Longrightarrow\!\bowtie} \mathtt{Y_2}$$

Notice that we have preserved the local context of the inner comprehension by using `Join` instead of `OuterJoin`. `E` is expected to be more efficient than `E'`, because the intermediate join between $\mathtt{Y_1}$ and $\mathtt{Y_2}$ has a filter that reduces the size of the result. In `E'` the first `OuterJoin` is likely to generate a big collection. Even if it was restricted by some predicate, being an `OuterJoin` it would preserve the values of the left operand independent of whether they matched any elements of the right operand. Moreover, `E'` cannot be transformed to `E`, since the two expressions are not syntactically equivalent.

3. This framework handles the `merge` operations of collection monoids, e.g. `union`.

   Consider the query:

   $$\begin{array}{l} \mathtt{select\ x.C + y.D} \\ \mathtt{from\ X\ as\ x,\ (x.A\ union\ x.B)\ as\ y} \end{array} \tag{5.3}$$

   Assume that both attributes `A` and `B` of the extent `X` are set collections. The calculus representation of this query is:

   $$\mathtt{bag}\{\mathtt{x.C + y.D} | \mathtt{x} \leftarrow \mathtt{X}, \mathtt{y} \leftarrow \mathtt{bagof}(\mathtt{merge}[\mathtt{set}](\mathtt{x.A}, \mathtt{x.B}))\}$$

   This query cannot be translated in contexts where operators like `Merge` or `OuterMerge` are not supported (e.g. [FM00]). In this framework, it takes the algebraic form:

   $$\overset{\mathtt{bag}}{\underset{\mathtt{x,new,new',y}}{\pi}} \overset{\mathtt{x.C+y.D}}{\overset{\mathtt{bag}}{\underset{\mathtt{x,new,new'\ y}}{\mu}}} \overset{\mathtt{new'}}{\underset{\mathtt{x,new\ new'}}{\mathbb{B}}}_{\mathtt{true}} \overset{\mathtt{bagof}}{\underset{\mathtt{x\ new}}{\mathbb{M}}}_{\mathtt{new}} \overset{\mathtt{set}}{\underset{\mathtt{x.A,x.B}}{(\mathtt{X})}}$$

4. The `OuterSelect` operator allows for filtering the results of a nested comprehension before processing its head. This is not possible in other contexts, when the predicate is not simple, but needs to be unnested.

Consider the OQL query:

$$
\begin{array}{l}
\texttt{select x.D} \\
\texttt{from X as x} \\
\texttt{where min (select } \mathtt{y_1.A_1 + y_2.A_2} \\
\quad \texttt{from } \mathtt{Y_1} \texttt{ as } \mathtt{y_1},\ \mathtt{Y_2} \texttt{ as } \mathtt{y_2} \\
\quad \texttt{where (forall } \mathtt{y_3} \texttt{ in } \mathtt{Y_3} : \ \mathtt{y_1.B_1 < x.B + y_3.B_3}) \texttt{ and} \\
\qquad (\texttt{exists z in x.E} : \ \mathtt{z.C > y_2.C_2})) \ > \ 4
\end{array}
\tag{5.4}
$$

If X is a bag or list and Y_1, Y_2, Y_3 and x.E are sets, bags or lists, this query corresponds to the following normalized calculus form:

$$
\begin{array}{l}
\mathtt{bag}\{\mathtt{x.D} | \mathtt{x} \leftarrow \mathtt{X}, \mathtt{min}\{\mathtt{y_1.A_1 + y_2.A_2} | \mathtt{y_1} \leftarrow \mathtt{Y_1}, \mathtt{y_2} \leftarrow \mathtt{Y_2}, \\
\quad \mathtt{all}\{\mathtt{y_1.B_1 < x.B + y_3.B_3} | \mathtt{y_3} \leftarrow \mathtt{Y_3}\}, \mathtt{z} \leftarrow \mathtt{x.E}, \mathtt{z.C > y_2.C_2}\} > 4\}
\end{array}
$$

This calculus form is translated to the algebraic expression:

$$
\overset{\text{bag}}{\underset{\text{x,g}_1}{\pi}}{}^{\text{x.D}} \ \overset{\text{bag}}{\underset{\text{x,g}_1}{\sigma}}{}^{\text{g}_1>4} \ (\text{E}_1)
$$

where

$$
\text{E}_1 = \overset{\text{bag}}{\underset{\text{x,y}_1,\text{y}_2}{\Gamma}}{}^{\text{min}}_{\text{g}_1}{}^{\text{y}_1.\text{A}_1+\text{y}_2.\text{A}_2}_{\text{x}} \ {}^{\text{true}} \ \overset{\text{bag set/bag}}{\underset{\text{x,y}_1,\text{y}_2\ \text{z}}{=\!\!\!\!\!\!\mu}}{}^{\text{x.E}}_{\text{z.C}>\text{y}_2.\text{C}_2} \ \overset{\text{bag bag}}{\underset{\text{x,y}_1,\text{y}_2,\text{g}}{=\!\!\!\!\!\!\sigma}}{}^{\text{x}}_{\text{g}} \ (\text{E}_2)
$$

where

$$
\text{E}_2 = \overset{\text{bag}}{\underset{\text{x,y}_1,\text{y}_2,\text{y}_3}{\Gamma}}{}^{\text{all}}_{\text{g}}{}^{\text{y}_1.\text{B}_1<\text{x.B}+\text{y}_3.\text{B}_3}_{\text{x,y}_1,\text{y}_2} \ {}^{\text{true}} \left(\left(\text{X} \ \overset{\text{bag set/bag}}{\underset{\text{x,y}_1,\text{y}_2}{=\!\!\!\!\!\!\bowtie}} \ \left(\text{Y}_1 \ \overset{\text{set/bag}}{\underset{\text{y}_1,\text{y}_2}{\bowtie}} \ \text{Y}_2 \right) \right) \ \overset{\text{bag set/bag}}{\underset{\text{x,y}_1,\text{y}_2,\text{y}_3}{=\!\!\!\!\!\!\bowtie}} \ \text{Y}_3 \right)
$$

5. Annotating operators with monoid information allows for the optimization and the efficient execution of the physical plan.

   This has been discussed in section 5.1.

6. Annotating the Outer· operators with two monoids allows for unnesting non-commutative collections in a commutative context. This is an issue that has not been considered in previous research.

Say that the outer comprehension of a calculus expression has a commutative monoid, e.g. `bag`. Assume also that it has a predicate, part of which is a nested comprehension with a non-commutative monoid, say `list`. This inner comprehension includes variables bound in the generators of the outer comprehension. At some point, the generators of the inner comprehension are combined with those of the outer comprehension, through an `OuterJoin` or an `OuterUnnest` operator. In our context, these two operators are annotated with two monoids, one for the outer and one for the inner comprehension. This means that the elements of the inner comprehension are not simply unnested or joined. They are annotated with a serial number denoting their order in the original nested collections. This information is essential at the time a subsequent `Nest` operator groups these values again into a non-commutative monoid. If the `Outer`$\cdot$ operators flattened everything, say into a bag, then the `Nest` operator would try to merge elements of a bag into a list in an arbitrary way.

5.4 Conclusion

This chapter presents a framework for the translation of calculus expressions to algebraic forms. This work is based on related work by Fegaras and Maier. However, the current appoach has extended their work in several respects. First, new algebraic operators have been proposed to address the translation of calculus expressions that have not been considered in other contexts. The latter expressions include `bagof`$(\mathtt{e})$, `listof`$(\mathtt{e})$, `arrayof`$(\mathtt{e})$, `orderof`$(\mathtt{e})$ and `merge`$[<\mathtt{coll\_mon}>]$. Second, operators that have been used in other contexts are annotated with monoid information. This is useful for three reasons: i) the optimization of the algebraic plan, ii) the efficient execution of physical operators and iii) the ability to outerjoin or outer-unnest elements of a nested comprehension even if its monoid has different properties from the monoid of the external comprehension. Third, the translation algorithm from calculus to algebra has been modified in two respects: i) generators of nested comprehensions are processed locally as long as possible and ii) predicates relating variables of inner and outer comprehensions are pushed down (using `OuterSelect`); this is not possible in [FM00] for predicates that contain nested queries.

Most of the issues that are addressed in this chapter are not found in relational contexts. Many of them are due to the manipulation of multiple collection types. This feature of OQL adds complexity to the algebraic framework and requires careful manipulation of the operators borrowed from the relational algebra.

Chapter 6

Using Association Rules to Optimize Queries Semantically

In chapters 3, 4 and 5, we studied various stages of OQL processing, namely type inference, translation to calculus form, normalization and translation to algebraic form. Unlike the calculus level, where the focus was on syntactic transformations (normalization rules), the objective at the algebraic level is to identify opportunities for optimizing queries semantically. This chapter presents joint work with Dr Ken Moody, which has also been reported in the SSDBM conference [TM01]. The goal of this work is to study the use of association rules to add or eliminate constraints in the `where` clause of a `select` query. In terms of the algebraic representation of OQL, the goal is to add or remove constraints in the predicate of a `Select` operator. We take advantage of the following heuristics presented by Siegel et al. [SSS92]: i) if a selection on attribute A is implied by another selection condition on attribute B and A is not an index attribute, then the selection on A can be removed from the query; ii) if a relation R in the query has a restricted attribute A and an unrestricted cluster index attribute B, then look for a rule where the restriction on A implies a restriction on B. The contribution of our work is twofold. First, we present detailed algorithms that apply these heuristics. Hence, our ideas are easy to implement. Second, we discuss conditions under which it is worth applying these optimization techniques, and we show the extent to which they speed up query execution.

6.1 Introduction

Rules that correlate values of data attributes in large databases have been investigated for two main purposes. The first is to aid management and decision-making in large companies with great amounts of data. The second relates to query optimization, by exploiting the semantic knowledge expressed in rules [CGM90, GGMR97, SSS92, Yoo94, YSP95]. Siegel et al. [SSS92] have proposed several

heuristics to guide rule generation and so help to transform queries to more efficient forms. In this chapter we investigate algorithms for applying two of these heuristics to optimize range queries. Although we are interested in optimizing OQL, our algorithms and optimization techniques are also applicable in any relational or object-relational database system. The form of queries under consideration is:

```
select e(x)
from Extent_X as x
where p1(x) and ... and pn(x)
```

The first heuristic (H1) aims to eliminate as many predicates (or constraints) $\mathtt{p_i(x)}$ as possible, when they are implied by other existing predicates. The second heuristic (H2) finds implications from one or more existing predicates to a new predicate on a cluster index attribute. If the latter is added to the `where` clause of the query and is tested first, then the existing predicates will be tested on a smaller number of objects. In order to take advantage of either of these heuristics, we use association rules that are relevant to `Extent_X`.

In our context, association rules express semantic correlations between the values of the attributes of all objects belonging to a certain extent. They consist of the antecedent, which has one or more constraints, and the consequent, which has just one constraint: $\mathtt{constr_1} \wedge \ldots \wedge \mathtt{constr_n} \Rightarrow \mathtt{constr_0}$. The constraints $\mathtt{constr_i}$ are of the form $\mathtt{class\_name.attribute_i\ comp\_op_i\ value_i}$, where for all $\mathtt{0 <= i <= n}$, $\mathtt{comp\_op_i} \in \{<, >, <=, >=, =, \neq\}$. Association rules are typically ranked by two measures of interest: confidence and support. Confidence indicates how accurate or reliable a rule is, since it is defined as the fraction of all objects satisfying the antecedent that also satisfy the consequent. Support measures the popularity of a rule, since it reflects the percentage of objects of the extent satisfying both the antecedent and the consequent. A number of algorithms have been proposed for mining association rules over categorical or quantitative data [AIS93, SA96, MY97, Par95, TUA+98]. In the remainder of the chapter, we assume that the process of mining rules has been completed and that we already have a warehouse with a set of rules for each extent in the database. Further, in this warehouse we store the exceptions to each rule, i.e. those objects that satisfy its antecedent but not its consequent. We also assume that the rules and their exceptions are correctly maintained when the database is updated.

The automatic use of these rules by the OQL optimizer raises many issues. The first concerns mapping the predicates in the `where` clause of a `select` query to constraints that could potentially be found in the antecedent (or the consequent) of association rules. This is a step towards identifying the templates of association rules that could help in our optimizations; this task is slightly different for heuristics H1 and H2. The next step is to present a common graph algorithm that goes through all existing association rules and discovers the ones that fit the templates along with their exceptions. Another graph algorithm should combine these rules to discover the possible optimization solutions,

i.e. the sets of constraints that could be omitted from (H1) or added to (H2) the query. Using the optimal optimization solution, we then present the actual transformations of OQL queries for each of the two optimization techniques. Finally, we investigate the conditions under which it is worth applying the optimizations and show the extent to which they are expected to speed up query execution.

6.2 Converting query predicates to rule constraints

The first step towards applying either of the heuristics is to map the predicates found in the `where` clause of a `select` query to constraints found in the existing association rules. Consider the following OQL query:

$$\begin{aligned}
&\texttt{select x from Employees as x}\\
&\texttt{where x.salary} > 35,000 \texttt{ and x.salary} < 55,000 \texttt{ and}\\
&\texttt{x.position} = \text{“}\texttt{associate}\text{”} \texttt{ and}\\
&\texttt{x.birthYear} <= 1975 \texttt{ and x.birthYear} >= 1955\\
&\texttt{and x.emplYear} <= 1990 \texttt{ and}\\
&\texttt{x.emplYear} > 1970
\end{aligned} \tag{6.1}$$

We first need to gather all predicates that refer to the same attribute into one predicate, and form constraints of the form `attribute_name in range`. The resulting constraints in our example are given below:

$$\begin{aligned}
&\texttt{salary in } \{(35,000, 55,000)\} && C_A\\
&\texttt{position in } \{\text{“}\texttt{associate}\text{”}\} && C_B\\
&\texttt{birthYear in } \{[1955, 1975]\} && C_C\\
&\texttt{emplYear in } \{(1970, 1990]\} && C_D
\end{aligned}$$

Based on these query constraints, the next step is to search for *similar* constraints in the association rules. Suppose that the warehouse contains the association rules illustrated in figure 6.1, which are relevant to the class `Employee`. The constraints in the antecedent and the consequent of the rules are labeled with short names, e.g. $C_{k.R_{kj}}$ is a constraint limiting the values of attribute `k` to the range R_{kj}. Here a *range* is a specified set of values of some attribute. The set of ranges associated with a particular attribute is partially ordered under inclusion in the usual way.

To distinguish constraints that are derived from the `where` clause of our query from those in the antecedent or consequent of the association rules, we refer to the former as *query constraints*, and to the latter as *rule constraints*. For each heuristic we look for rule constraints corresponding to the query constraints.

$$
\begin{array}{rcl}
\texttt{birthYear in } \{[1955, 1975]\}(\texttt{C}_{1.\texttt{R}_{11}}) & \Rightarrow & \texttt{salary in } \{[35,000, 50,000]\}(\texttt{C}_{2.\texttt{R}_{21}}) \\
\texttt{position in } \{\text{“associate”}\}(\texttt{C}_{4.\texttt{R}_{41}}) & \Rightarrow & \texttt{salary in } \{[30,000, 60,000]\}(\texttt{C}_{2.\texttt{R}_{22}}) \\
\texttt{position in } \{\text{“manager”}\}(\texttt{C}_{4.\texttt{R}_{42}}) & \Rightarrow & \texttt{birthYear in } \{[1955, 1980]\}(\texttt{C}_{1.\texttt{R}_{12}}) \\
\texttt{birthYear in } \{[1950, 1970]\}(\texttt{C}_{1.\texttt{R}_{13}}) & \Rightarrow & \texttt{position in } \{\text{“associate”}\}(\texttt{C}_{4.\texttt{R}_{41}}) \\
\texttt{emplYear in } \{[1975, 1990]\}(\texttt{C}_{5.\texttt{R}_{53}}) & \Rightarrow & \texttt{position in } \{\text{“associate”}\}(\texttt{C}_{4.\texttt{R}_{41}}) \\
\texttt{emplYear in } \{[1965, 1985]\}(\texttt{C}_{5.\texttt{R}_{52}}) & \Rightarrow & \texttt{efficiency in } \{[3, 6]\}(\texttt{C}_{3.\texttt{R}_{31}}) \\
\left.\begin{array}{r}\texttt{salary in } \{[30,000, 65,000]\}(\texttt{C}_{2.\texttt{R}_{23}}) \\ \texttt{position in } \{\text{“manager”}\}(\texttt{C}_{4.\texttt{R}_{42}})\end{array}\right\} & \Rightarrow & \texttt{emplYear in } \{[1970, 1985]\}(\texttt{C}_{5.\texttt{R}_{51}}) \\
\left.\begin{array}{r}\texttt{salary in } \{[30,000, 60,000]\}(\texttt{C}_{2.\texttt{R}_{22}}) \\ \texttt{efficiency in}\{[3, 6]\}(\texttt{C}_{3.\texttt{R}_{31}})\end{array}\right\} & \Rightarrow & \texttt{birthYear in } \{[1955, 1975]\}(\texttt{C}_{1.\texttt{R}_{11}}) \\
\left.\begin{array}{r}\texttt{salary in } \{[30,000, 60,000]\}(\texttt{C}_{2.\texttt{R}_{22}}) \\ \texttt{efficiency in}\{[3, 6]\}(\texttt{C}_{3.\texttt{R}_{31}})\end{array}\right\} & \Rightarrow & \texttt{position in } \{\text{“associate”}\}(\texttt{C}_{4.\texttt{R}_{41}}) \\
\left.\begin{array}{r}\texttt{birthYear in } \{[1955, 1980]\}(\texttt{C}_{1.\texttt{R}_{12}}) \\ \texttt{salary in } \{[35,000, 50,000]\}(\texttt{C}_{2.\texttt{R}_{21}})\end{array}\right\} & \Rightarrow & \texttt{efficiency in}\{[3, 6]\}(\texttt{C}_{3.\texttt{R}_{31}})
\end{array}
$$

Figure 6.1: Association rules relevant to the class Employee

6.2.1 Heuristic H1

We first discuss rule constraints relevant to the constraint elimination heuristic. For each constraint C on attribute_name with range R, identify the following rule constraints:

Category 1 - relaxed constraints First, identify the constraints whose ranges are the next more general than (or the same as) the range of the original constraint C. If there exists such a constraint $\texttt{C}_0$ with a range $\texttt{R}_0$, then for all other constraints $\texttt{C}_\texttt{i}$ i $\neq 0$ on attribute_name with ranges $\texttt{R}_\texttt{i}$, if $\texttt{R} \subseteq \texttt{R}_\texttt{i}$ then $\texttt{not}(\texttt{R}_\texttt{i} \subseteq \texttt{R}_0)$. That is, $\texttt{R}_0$ is one of the least supersets of R among all ranges of constraints referring to the same attribute.

Category 2 - relaxed combinations of constraints Second, identify minimal combinations of constraints which have the property that the union of their ranges is a superset of the range of the original constraint C. The constraints in any combination must satisfy the following conditions: firstly, none of the constraints can be the same as (or with a wider range than the range of) any constraint identified previously in category 1; secondly, no combination must include all the constraints of a smaller combination; thirdly, if either of two constraints on the same attribute may take part in a combination, and the range of the one is a subset of the range of the other, then the one with the smaller range is selected.

Category 3 - tightened constraints Third, identify the constraints whose ranges are the next more specific than (or the same as) the range of the original constraint. If there exists such a constraint C_0 with a range R_0, it means that for all other constraints C_i i $\neq 0$ on attribute_name with ranges R_i, if $R \supseteq R_i$ then not$(R_i \supseteq R_0)$. That is, R_0 is one of the greatest subsets of R among all ranges of constraints referring to the same attribute.

Category 4 - tightened combinations of constraints Fourth, identify maximal combinations of constraints which have the property that the intersection of their ranges is a subset of the range of the original constraint C. The constraints in any combination must satisfy the following conditions: firstly, none of the constraints can be the same as (or with a range contained in the range of) any constraint identified previously in category 3: secondly, no combination must include all the constraints of a smaller combination: thirdly, if either of two constraints on the same attribute may take part in a combination, and the range of the one is a superset of the range of the other, then the one with the wider range is selected.

In the example of query 6.1, the query constraints C_A, C_B, C_C and C_D correspond to the following rule constraints:

	Relaxed Constraints	*Tightened Constraints*
C_A	salary in $\{[30,000, 60,000]\}(C_{2.R_{22}})$	salary in $\{[35,000, 50,000]\}(C_{2.R_{21}})$
C_B	position in $\{$"associate"$\}(C_{4.R_{41}})$	position in $\{$"associate"$\}(C_{4.R_{41}})$
C_C	birthYear in $\{[1955, 1975]\}(C_{1.R_{11}})$	birthYear in $\{[1955, 1975]\}(C_{1.R_{11}})$
C_D		emplYear in $\{[1970, 1985]\}(C_{5.R_{51}})$
		emplYear in $\{[1975, 1990]\}(C_{5.R_{53}})$
	Relaxed Combinations of Constraints	*Tightened Combinations of Constraints*
C_C		$\{$birthYear in $\{[1955, 1980]\}(C_{1.R_{12}})$
		birthYear in $\{[1950, 1970]\}(C_{1.R_{13}})\}$
C_D	$\{$emplYear in $\{[1970, 1985]\}(C_{5.R_{51}})$	
	emplYear in $\{[1975, 1990]\}(C_{5.R_{53}})\}$	

For each tightened combination of constraints we generate a new constraint whose range is the intersection of the ranges of the constraints in the combination. We add this new constraint to the category of tightened constraints and create a temporary rule with the initial constraints in the antecedent and the new tightened constraint in the consequent. For instance, based on the tightened combination $\{$birthYear in $\{[1955, 1980]\}(C_{1.R_{12}})$, birthYear in $\{[1950, 1970]\}(C_{1.R_{13}})\}$, we create the constraint $\{$birthYear in $\{[1955, 1970]\}(C_{1.R_{1.23}})$. Repeating this procedure for all tightened combinations of constraints, we generate a new set of tightened constraints.

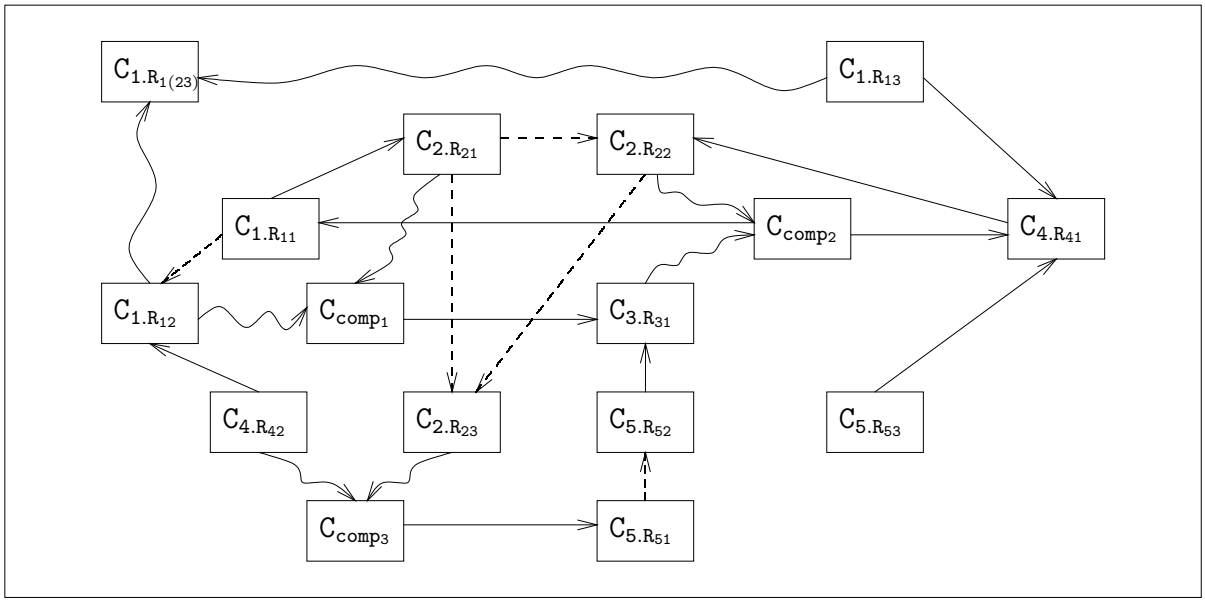

Figure 6.2: Graph of Constraints

The rationale behind finding the constraints in categories 1 to 3 is the following. By definition, the query constraints imply their corresponding relaxed constraints. Likewise, the tightened constraints imply the initial query constraints. Hence if we can find association rules that relate constraints in the first two categories to those in the third, it is equivalent to finding rules that correlate the initial query constraints with each other.

6.2.2 Heuristic H2

For the purposes of applying the constraint introduction heuristic, we identify the following rule constraints:

Category 1 - relaxed constraints These are identical to the relaxed constraints described in section 6.2.1.

Category 2 - relaxed combination of constraints This category is the same as its counterpart in section 6.2.1.

Category 3 - index constraints If the extent has a cluster index on attribute $\texttt{attr}_i$, then this category includes all the constraints on this attribute that occur as consequents in the existing rules. Assume that the cluster index of the extent `Employees` is the attribute `efficiency`. Based on the rules of figure 6.1, the only index constraints is the constraint `efficiency` $\texttt{in}\{[3, 6]\}$.

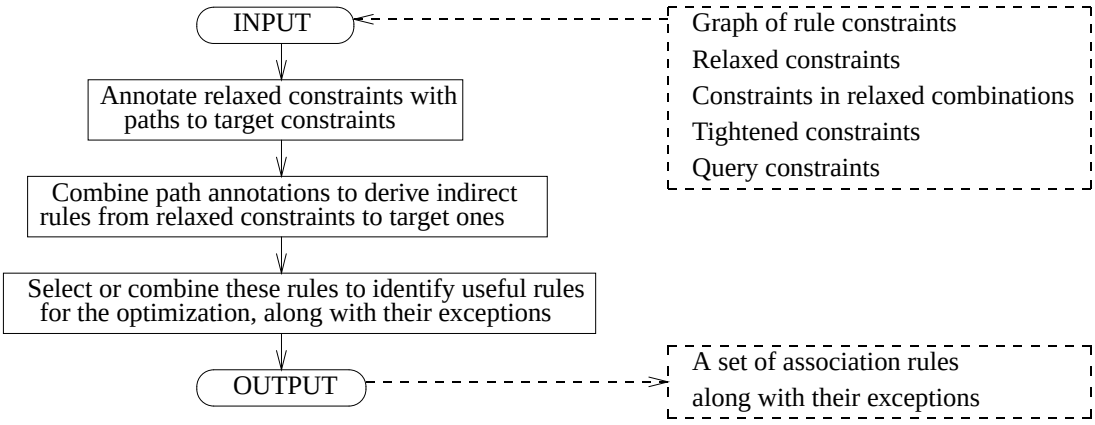

Figure 6.3: An abstract view of the algorithm

Rule constraints imply relaxed (combinations of) constraints, by definition. If we succeed in finding paths from relaxed (combinations of) constraints to an index constraint, then we can add the latter to the `where` clause of the query. This allows us to reduce the disc access overhead without altering the results of the query.

6.2.3 Discussion

Constraints in categories 1 and 2 play an identical role for heuristics H1 and H2. However, the other categories play a different role in the two cases. Tightened constraints are similar to existing query constraints that can potentially be eliminated. Index constraints are new constraints that can potentially be added to the `where` clause of the query. In the next section we present a graph algorithm that finds associations from the relaxed constraints to the tightened (H1) or the index (H2) constraints respectively. Since the algorithm is common to both heuristics, tightened and index constraints are hereafter referred to as *target constraints*.

6.3 Identifying associations between constraints

In this section, we present an algorithm that goes through the association rules in the warehouse and identifies direct or indirect correlations between the relaxed and the target constraints discussed in the previous section. Before presenting this algorithm, we discuss an alternative graphical representation of the association rules. Consider for example the association rules in figure 6.1. They can be represented using a directed graph of constraints, see figure 6.2.

A rule having a single constraint in its antecedent (e.g. $\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}}$) is represented by two vertices, labeled with the constraints ($\mathtt{C}_{1.\mathtt{R}_{11}}$ and $\mathtt{C}_{2.\mathtt{R}_{21}}$) and linked together by an edge from the an-

tecedent constraint ($C_{1.R_{11}}$) to the consequent constraint ($C_{2.R_{21}}$). A rule with more than one constraint in its antecedent uses an additional vertex to denote the conjunction. The link $\rightsquigarrow$ denotes the fact that a constraint in the antecedent of a rule implies the consequent only in combination with the other conjuncts forming the antecedent. The dashed edges link pairs of nodes that represent constraints on the same attribute, with the source constraint implying the destination constraint.

The notion of path in our context requires one extension to the standard notion for a directed graph. A path from a constraint C_{source} to a constraint C_{dest} is any sequence of vertices linked by directed edges, provided that no vertex represents a composite constraint. If a path contains a composite constraint, say $C = C_1 \wedge \ldots \wedge C_n$, then there must be n paths from C_{source}, one to each of the conjuncts C_i. These subpaths merge at the node representing the composite constraint, and from there the path continues towards C_{dest}.

We consider also paths from a set of constraints $\{C_1, \ldots C_n\}$ to a single constraint. Such a path consists of subpaths starting from the source constraints $\{C_1, \ldots C_n\}$ which merge at various composite constraints before reaching the destination constraint.

We can now develop an algorithm that navigates over a graph of constraints, finds paths linking relaxed (combinations of) constraints to target constraints and combines them to derive association rules useful for the purposes of optimization. It consists of three basic steps as shown in figure 6.3. The algorithm is discussed in detail in section 6.3.1. At the end of each step we give an example to show how it can be applied for heuristic H1. It is equally applicable to heuristic H2.

Relaxed constraints and constraints in relaxed combinations often play the same role in the following algorithm, and we use the term *source constraint* in such contexts.

6.3.1 Algorithm

For each constraint $C_{k.R_{kj}}$ in the target constraints take the following steps:

Step 1

Navigate backwards from $C_{k.R_{kj}}$ in the graph of constraints, i.e. navigate against the direction of the edges and annotate any source constraints encountered with information about the reverse path traversed so far. On encountering a composite constraint navigate backwards from *all* of its conjuncts. If a particular constraint is encountered more than once, or if there is no incoming link, then backtracking from the current constraint terminates.

Step 1 may be implemented as a recursive method `backtrack` on a class `Constraint`. We demonstrate its functionality by an example. Consider the target constraint $C_{4.R_{41}}$ in the constraint graph (figure 6.2). We recursively apply `backtrack` to all the constraints leading to it, i.e. $C_{1.R_{13}}$, $C_{5.R_{53}}$ and C_{comp_2}. The argument for the method is the path traversed so far, i.e. $\{C_{4.R_{41}}\}$. $C_{1.R_{13}}$ is

Relaxed Constraints	*Path Annotations*
$C_{2.R_{22}}$	$\{C_{2.R_{22}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
	$\{C_{2.R_{22}} \rightarrow C_{2.R_{23}} \rightsquigarrow C_{comp_3} \rightarrow C_{5.R_{51}} \rightarrow C_{5.R_{52}} \rightarrow C_{3.R_{31}} \rightsquigarrow$ $\rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
$C_{4.R_{41}}$	
$C_{1.R_{11}}$	$\{C_{1.R_{11}} \rightarrow C_{2.R_{21}} \rightarrow C_{2.R_{22}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
	$\{C_{1.R_{11}} \rightarrow C_{2.R_{21}} \rightarrow C_{2.R_{22}} \rightarrow C_{2.R_{23}} \rightsquigarrow C_{comp_3} \rightarrow C_{5.R_{51}} \rightarrow$ $\rightarrow C_{5.R_{52}} \rightarrow C_{3.R_{31}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
	$\{C_{1.R_{11}} \rightarrow C_{2.R_{21}} \rightsquigarrow C_{comp_1} \rightarrow C_{3.R_{31}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
	$\{C_{1.R_{11}} \rightarrow C_{1.R_{12}} \rightsquigarrow C_{comp_1} \rightarrow C_{3.R_{31}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
Constraints in Relaxed Combinations	*Path Annotations*
$C_{5.R_{51}}$	$\{C_{5.R_{51}} \rightarrow C_{5.R_{52}} \rightarrow C_{3.R_{31}} \rightsquigarrow C_{comp_2} \rightarrow C_{4.R_{41}}\}$
$C_{5.R_{53}}$	$\{C_{5.R_{53}} \rightarrow C_{4.R_{41}}\}$

Figure 6.4: Paths from relaxed constraints to the target constraint $C_{4.R_{41}}$

not a source constraint, so no path annotation is assigned to it. No constraint leads to it (there is no incoming link), so there is no further recursive call from it. However, $C_{5.R_{53}}$ is a constraint in a relaxed combination, so it is annotated with the path $\{C_{4.R_{41}}\}$. No recursive call occurs from it either. C_{comp_2} is a composite constraint, so we recursively call `backtrack` on each of the conjuncts $C_{2.R_{22}}$ and $C_{3.R_{31}}$, passing as argument the updated path traversed $\{C_{comp_2} \rightarrow C_{4.R_{41}}\}$. Recursive calls of `backtrack` continue until the constraint on which a call is made has no incoming link or the constraint appears in the argument path annotation. In figure 6.4, we present the path annotations assigned to source constraints by the time Step 1 is completed for the target constraint $C_{4.R_{41}}$.

Discussion of step 2

The objective of step 2 is to combine the annotations of source constraints to identify *complete* paths from sets of these constraints to a target constraint $C_{k.R_{kj}}$. The first step of the algorithm annotated the source constraints with complete or incomplete paths from them to a particular target constraint. For instance, $C_{1.R_{11}}$, which is a relaxed constraint, is annotated with four paths, as shown in figure 6.4. Paths that include $\rightsquigarrow$ links are called incomplete; they are useful only in combination

A(e)	B(e)	C(e)	e in E(A ⇒ B)	e in E(B ⇒ C)	$e\ in$ E(A ⇒ C)
T	T	T	F	F	F
T	T	F	F	T	T
T	F	T	T	F	F
T	F	F	T	F	T
F	T	T	F	F	F
T	T	F	F	T	F
F	F	T	F	F	F
F	F	F	F	F	F

Figure 6.5: Truth table which shows the correctness of formula 6.2

with other incomplete paths that include the related conjuncts. One of the objectives of step 2 is to combine incomplete paths in order to identify (complete) paths from a source constraint like $\mathtt{C}_{1.\mathtt{R}_{11}}$ to a target constraint like $\mathtt{C}_{4.\mathtt{R}_{41}}$. In fact, we can combine paths from more than one source constraint to a destination constraint.

Each edge $\rightarrow$ stands for an association rule which possibly has a list of exceptions E. A series of consecutive edges $\rightarrow$ corresponds to an indirect association from the source constraint to the destination one. The exceptions to this rule are evaluated from the exceptions of the association rules involved. In particular, if constraint A leads to constraint B based on a rule with exceptions $\mathtt{E}_{\mathtt{A}\Rightarrow\mathtt{B}}$, and B leads to C based on a rule with exceptions $\mathtt{E}_{\mathtt{B}\Rightarrow\mathtt{C}}$, then the exception set of the indirect path $\mathtt{A} \rightarrow \mathtt{C}$ (or the indirect rule $\mathtt{A} \Rightarrow \mathtt{C}$) is:

$$\begin{aligned} \mathtt{E} \;=\; & \{\mathtt{exc} | \mathtt{exc} \leftarrow \mathtt{E}_{\mathtt{A}\Rightarrow\mathtt{B}},\ \mathtt{not}\ \mathtt{C}(\mathtt{exc})\} \\ & \cup \{\mathtt{exc} | \mathtt{exc} \leftarrow \mathtt{E}_{\mathtt{B}\Rightarrow\mathtt{C}},\ \mathtt{A}(\mathtt{exc})\} \end{aligned} \tag{6.2}$$

Note that $\mathtt{E}_{\mathtt{A}\Rightarrow\mathtt{B}} \cap \mathtt{E}_{\mathtt{B}\Rightarrow\mathtt{C}} = \{\}$, since all exceptions of $\mathtt{E}_{\mathtt{A}\Rightarrow\mathtt{B}}$ satisfy A, but do not satisfy B. Therefore they are not exceptions of $\mathtt{E}_{\mathtt{B}\Rightarrow\mathtt{C}}$. The correctness of formula 6.2 is proved by the truth table in figure 6.5.

Step 2

Path annotations derived from step 1 are combined to form complete paths by repeating steps 2.1 and 2.2:

Step 2.1 For all path annotations we omit all the initial constraints, until the first constraint which is followed by a $\rightsquigarrow$ edge (this constraint is maintained). To omit these initial constraints, we must

first evaluate the exceptions involved in the rules until the $\rightsquigarrow$ link (equation 6.2). The resulting paths are either empty or start with sub-constraints leading to composite constraints.

When this step is executed for the first time the resulting paths in our example are the following:

$$
\begin{array}{ll}
\mathtt{C}_{2.\mathtt{R}_{22}} & \mathtt{P}_1\colon \{\mathtt{C}_{2.\mathtt{R}_{22}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_1 = \{\} \\
\mathtt{C}_{2.\mathtt{R}_{22}} & \mathtt{P}_2\colon \{\mathtt{C}_{2.\mathtt{R}_{23}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_3} \rightarrow \mathtt{C}_{5.\mathtt{R}_{51}} \rightarrow \mathtt{C}_{5.\mathtt{R}_{52}} \rightarrow \mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_2 = \mathtt{E}(\mathtt{C}_{2.\mathtt{R}_{22}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{23}}) = \{\} \\
\mathtt{C}_{1.\mathtt{R}_{11}} & \mathtt{P}_3\colon \{\mathtt{C}_{2.\mathtt{R}_{22}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_3 = \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}}),\ \mathtt{not}\ \mathtt{C}_{2.\mathtt{R}_{22}}(\mathtt{e})\} \\
\mathtt{C}_{1.\mathtt{R}_{11}} & \mathtt{P}_4\colon \{\mathtt{C}_{2.\mathtt{R}_{23}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_3} \rightarrow \mathtt{C}_{5.\mathtt{R}_{51}} \rightarrow \mathtt{C}_{5.\mathtt{R}_{52}} \rightarrow \mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_4 = \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}}),\ \mathtt{not}\ \mathtt{C}_{2.\mathtt{R}_{22}}(\mathtt{e}),\ \mathtt{not}\ \mathtt{C}_{2.\mathtt{R}_{23}}(\mathtt{e})\} \\
& \quad = \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}}),\ \mathtt{not}\ \mathtt{C}_{2.\mathtt{R}_{23}}(\mathtt{e})\} \\
\mathtt{C}_{1.\mathtt{R}_{11}} & \mathtt{P}_5\colon \{\mathtt{C}_{2.\mathtt{R}_{21}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_1} \rightarrow \mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_5 = \mathtt{E}(\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}}) \\
\mathtt{C}_{1.\mathtt{R}_{11}} & \mathtt{P}_6\colon \{\mathtt{C}_{1.\mathtt{R}_{12}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_1} \rightarrow \mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_6 = \{\} \\
\mathtt{C}_{5.\mathtt{R}_{51}} & \mathtt{P}_7\colon \{\mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_7 = \{\} \cup \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_{5.\mathtt{R}_{52}} \Rightarrow \mathtt{C}_{3.\mathtt{R}_{31}}), \mathtt{C}_{5.\mathtt{R}_{51}}(\mathtt{e})\} \\
\mathtt{C}_{5.\mathtt{R}_{53}} & \mathtt{P}_8\colon \{\} \\
& \mathtt{E}_8 = \mathtt{E}(\mathtt{C}_{5.\mathtt{R}_{53}} \Rightarrow \mathtt{C}_{4.\mathtt{R}_{41}})
\end{array}
$$

If a path annotation is empty, it means that there is a complete path from the source constraint (e.g. $\mathtt{C}_{5.\mathtt{R}_{53}}$) to the destination one ($\mathtt{C}_{4.\mathtt{R}_{41}}$). A new rule is created from the former to the latter ($\mathtt{C}_{5.\mathtt{R}_{53}} \Rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}$) and the exceptions evaluated so far for this path annotation ($\mathtt{E}_8$) are assigned to the new association rule. The path annotation and its exceptions are deleted. If an empty annotation corresponds to more than one source constraint then the rule that is generated has a composite antecedent, i.e. it is of the form: $\mathtt{C}_1 \wedge \ldots \wedge \mathtt{C}_\mathtt{n} \Rightarrow \mathtt{C}_0$.

When using equation 6.2 to determine exceptions during step 2.1 we must take note of the exception sets arising from earlier iterations of steps 2.1 and 2.2.

Step 2.2 For each composite constraint $\mathtt{C}_{\mathtt{comp}}$ at the second position of some path annotation, we try to find a set of path annotations having at their first positions the component constraints of $\mathtt{C}_{\mathtt{comp}}$. These path annotations are combined into new path annotations as follows: i) the subconstraints at their first positions are omitted; ii) the resulting exceptions of the remaining paths are the union of the exceptions evaluated so far for each combined path that satisfy the

source constraints of the other combined paths. If $\mathtt{C} = \mathtt{C}_1 \wedge \mathtt{C}_2$,

$$\begin{aligned}\mathtt{E}(\mathtt{C}_\mathtt{A} \wedge \mathtt{C}_\mathtt{B} \Rightarrow \mathtt{C}) = \\ \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_\mathtt{A} \Rightarrow \mathtt{C}_1), \mathtt{C}_\mathtt{B}(\mathtt{e})\} \cup \\ \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_\mathtt{B} \Rightarrow \mathtt{C}_2), \mathtt{C}_\mathtt{A}(\mathtt{e})\}\end{aligned} \tag{6.3}$$

When step 2.2 is executed for the first time in our example, the following path annotations are generated:

$$\begin{array}{ll}
\mathtt{C}_{2.\mathtt{R}_{22}}, \mathtt{C}_{5.\mathtt{R}_{51}} & \mathtt{P}_9 \colon \{\mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_9 = \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}_7,\ \mathtt{C}_{2.\mathtt{R}_{22}}(\mathtt{e})\} \\
\mathtt{C}_{1.\mathtt{R}_{11}}, \mathtt{C}_{5.\mathtt{R}_{51}} & \mathtt{P}_{10} \colon \{\mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_{10} = \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}_3,\ \mathtt{C}_{5.\mathtt{R}_{51}}(\mathtt{e})\} \\
& \quad \cup \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}_7,\ \mathtt{C}_{1.\mathtt{R}_{11}}(\mathtt{e})\} \\
\mathtt{C}_{1.\mathtt{R}_{11}} & \mathtt{P}_{11} \colon \{\mathtt{C}_{\mathtt{comp}_1} \rightarrow \mathtt{C}_{3.\mathtt{R}_{31}} \rightsquigarrow \mathtt{C}_{\mathtt{comp}_2} \rightarrow \mathtt{C}_{4.\mathtt{R}_{41}}\} \\
& \mathtt{E}_{11} = \{\mathtt{e}|\mathtt{e} \leftarrow \mathtt{E}_5,\ \mathtt{C}_{1.\mathtt{R}_{11}}(\mathtt{e})\} \\
& \quad = \mathtt{E}(\mathtt{C}_{1.\mathtt{R}_{11}} \Rightarrow \mathtt{C}_{2.\mathtt{R}_{21}})
\end{array}$$

If two path annotations which are combined do not have the same paths after the composite constraint (after the second position), then two combinations of paths should be generated, one for each annotation.

We retain path annotations $\mathtt{P}_1$ to $\mathtt{P}_8$ in order to combine them with new annotations at a later execution of step 2.2.

Substeps 2.1 and 2.2 are repeated until neither of them has any effect on the path annotations. This happens when all paths that have not been converted to rules in step 2.1 cannot be further combined in step 2.2.

Step 3

Not all the rules derived in step 2.1 are useful for optimization purposes. The aim of this step is to filter and combine the rules derived from the previous step, referred to as $\mathtt{R}_\mathtt{A}$, in order to generate rules relating query constraints with each other (H1), or query constraints to new index constraints (H2). The resulting set of rules is referred to as $\mathtt{R}_\mathtt{B}$. We first give a detailed account of step 3 in the context of heuristic H1. We then point out a detail that should be changed so that the step is also applicable for H2.

We look for (sets of) rules in $\mathtt{R}_\mathtt{A}$ of the following kinds:

- A rule from a relaxed constraint to the target constraint $\mathtt{C}_\mathtt{i}$, i.e. of the form $\mathtt{C} \Rightarrow \mathtt{C}_0$. Such a rule is used to generate a new rule $\mathtt{C}' \Rightarrow \mathtt{C}'_0$, such that $\mathtt{C}'$, $\mathtt{C}'_0$ are the query constraints that refer to

the same attributes as C and C_0 respectively, C is one of the relaxed constraints of C' and C_0 is one of the target constraints of C'_0. Indeed,

$$\left.\begin{array}{l} C' \Rightarrow C \\ C \Rightarrow C_0 \text{ (existing rule)} \\ C_0 \Rightarrow C'_0 \end{array}\right\} C' \Rightarrow C'_0$$

If $C \Rightarrow C_0$ has exceptions E, the exceptions of the new rule $C' \Rightarrow C'_0$ are $E' = \{e | e \leftarrow E,\ C'(e)\}$.

- A set of rules of the form $C_i \Rightarrow C_0, i = 1, \ldots, n$, such that C_i are all constraints in the same relaxed combination of size n. This set of rules is used to generate the new rule $C' \Rightarrow C'_0$, in which C', C'_0 are the query constraints that refer to the same attribute as the C_i and C_0 respectively. If the corresponding exception lists of the rules $C_i \Rightarrow C_0, i = 1, \ldots, n$ are E_i, the exceptions of the new rule are $E = \{e | e \leftarrow E_1 \cup \ldots \cup E_n,\ C'(e)\}$.

- A rule from a set of relaxed constraints to a target one, i.e. $C_1 \wedge \ldots \wedge C_n \Rightarrow C_0$. This rule is converted to the corresponding rule $C'_1 \wedge \ldots \wedge C'_n \Rightarrow C'_0$, such that $C'_0, \ldots, C'_n$ are the query constraints on the same attributes as $C_0, \ldots, C_n$ respectively. The exceptions of the new rule are those of the initial rule that satisfy $C'_1 \wedge \ldots \wedge C'_n$, i.e. $E' = \{e | e \leftarrow E,\ C'_1(e),\ \ldots,\ C'_n(e)\}$.

- A set of rules whose antecedents contain both relaxed constraints and constraints in relaxed combinations is useful if it consists of all rules of the form:

$$\overbrace{C_1 \quad \ldots \quad C_k}^{\text{Relaxed constraints}} \quad \overbrace{C_{(k+1).j_1} \quad \ldots \quad C_{(k+m).j_m}}^{\text{Constraints in relaxed combinations}} \quad \Rightarrow \quad C_0$$

  where $1 \leq j_i \leq n_i$ for all $1 \leq i \leq m$.

  Constraints $C_{(k+i).1}, \ldots, C_{(k+i).n_i}$ form the relaxed combination i. The set contains $N = n_1 \times \ldots \times n_m$ rules, such that each constraint in a relaxed combination occurs in all possible combinations with constraints from the other relaxed combinations. In some of the rules some (or all) of the relaxed constraints $C_1 \ldots C_n$ may not be present.

  If such a set of rules occurs in R_A we derive $C'_1 \wedge \ldots \wedge C'_k \wedge C'_{k+1} \wedge \ldots \wedge C'_{k+m} \Rightarrow C'_0$, such that $C'_1, \ldots, C'_{k+m}, C'_0$ are the query constraints that refer to the same attributes as $C_1, \ldots, C_{(k+m).j_m}, C_0$ respectively. If the original rules have exception lists $E_1, \ldots, E_N$, then the resulting rule has exceptions $E = \{e | e \leftarrow E_1 \cup \ldots \cup E_N,\ C'_1(e),\ \ldots,\ C'_{k+m}(e)\}$.

  Evaluating exceptions appears very expensive in this case. However, in the common situation that $m = 1$, the cost of finding exceptions to the new rule is similar to the cost in the second case of step 3.

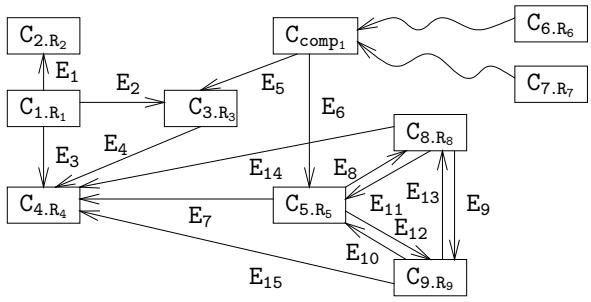

Figure 6.6: Correlations of query constraints

Note that in the context of heuristic H2, C_0 is a constraint on the cluster index attribute; it does not correspond to any of the existing query constraints. Hence, step 3 is applicable to H2, provided that in all four cases above, $C'_0 = C_0$.

6.4 Identifying optimization solutions

In the previous section, we gave an algorithm that navigates over a constraint graph and extracts a set of useful association rules along with their exceptions. This section combines these rules to identify all possible solutions to the constraint elimination or constraint introduction problem.

Consider an OQL `select` query with say 15 predicates. We combine all predicates referring to the same attribute into a single constraint $\texttt{attr}_i \texttt{ in } \texttt{range}_i\ (C_{i.R_i})$. Assume that we derive the following set of constraints: $C_{1.R_1}, \ldots, C_{12.R_{12}}$, say. Since there is just one range for each attribute, ranges are presented as R_i instead of R_{ij}. We first identify the rule constraints of possible interest for H1 or H2, then apply the graph algorithm presented in section 6.3. The next step is discussed separately for each heuristic.

6.4.1 Heuristic H1

Say that the graph algorithm (section 6.3) results in the associations illustrated in figure 6.6. Not all query constraints need be related; in our example only the first nine are. Note that the association graph (figure 6.6) is transitive; if there is a direct link from constraint A to constraint B, and another one from B to C, then A and C are also directly connected. This is a property of the graph algorithm in section 6.3. We now define an algorithm that produces a set of constraint elimination solutions $\{(M_i, E_i)\}$, where M_i, E_i stand for Maintained Constraints and Exceptions respectively. Each pair (M_i, E_i) is a solution with the following explanation:

We may eliminate all but the constraints $\mathtt{M_i}$ from the `where` clause of the query, given that we take into account the exceptions $\mathtt{E_i}$.

Algorithm

Step 1 Identify all constraints that do not have any incoming links. These constraints should be maintained (not eliminated), since no other constraint implies them. In figure 6.6, these constraints are $\mathtt{C}_{1.\mathtt{R}_1}, \mathtt{C}_{6.\mathtt{R}_6}, \mathtt{C}_{7.\mathtt{R}_7}$.

Step 2 Identify all cyclic paths that have no incoming link from any external constraint. If two cyclic paths have at least one common constraint, they are considered as a single cyclic path. This case does not occur in our example. Form combinations of constraints by choosing one constraint from every cyclic path identified. In our example there is just one such path containing the constraints $\mathtt{C}_{5.\mathtt{R}_5}, \mathtt{C}_{8.\mathtt{R}_8}, \mathtt{C}_{9.\mathtt{R}_9}$; therefore the combinations consist of just one element: $\{\mathtt{C}_{5.\mathtt{R}_5}\}$, $\{\mathtt{C}_{8.\mathtt{R}_8}\}$, $\{\mathtt{C}_{9.\mathtt{R}_9}\}$. Extend each of these combinations with the constraints without incoming links, identified in step 1. The resulting combinations are $\{\mathtt{C}_{5.\mathtt{R}_5}, \mathtt{C}_{1.\mathtt{R}_1}, \mathtt{C}_{6.\mathtt{R}_6}, \mathtt{C}_{7.\mathtt{R}_7}\}$, $\{\mathtt{C}_{8.\mathtt{R}_8}, \mathtt{C}_{1.\mathtt{R}_1}, \mathtt{C}_{6.\mathtt{R}_6}, \mathtt{C}_{7.\mathtt{R}_7}\}$ and $\{\mathtt{C}_{9.\mathtt{R}_9}, \mathtt{C}_{1.\mathtt{R}_1}, \mathtt{C}_{6.\mathtt{R}_6}, \mathtt{C}_{7.\mathtt{R}_7}\}$. Each combination is a minimal set of constraints that implies all the remaining constraints in the graph (figure 6.6).

Step 3 For each combination evaluate the exceptions that are involved in removing the implied constraints. If a constraint is implied by more than one constraint in a combination, then consider the exceptions of the strongest implication (the implication with the fewest exceptions). For example, the exceptions corresponding to the combination $\{\mathtt{C}_{5.\mathtt{R}_5}, \mathtt{C}_{1.\mathtt{R}_1}, \mathtt{C}_{6.\mathtt{R}_6}, \mathtt{C}_{7.\mathtt{R}_7}\}$ are evaluated by forming the union of the exception lists: $\mathtt{E}_1$ for the elimination of $\mathtt{C}_2$; $\mathtt{min\_set}\{\mathtt{E}_2, \mathtt{E}_5\}$ for the elimination of $\mathtt{C}_3$; $\mathtt{min\_set}\{\mathtt{E}_3, \mathtt{E}_6, \mathtt{E}_7\}$ for the elimination of $\mathtt{C}_4$; $\mathtt{E}_8$ for the elimination of $\mathtt{C}_8$; $\mathtt{E}_{12}$ for the elimination of $\mathtt{C}_9$.

Step 4 Form the final solutions to the elimination problem by adding to each combination the remaining query constraints that are not related to each other $(\mathtt{C}_{10.\mathtt{R}_{10}}, \mathtt{C}_{11.\mathtt{R}_{11}}, \mathtt{C}_{12.\mathtt{R}_{12}})$. The exceptions corresponding to each solution are filtered so that they satisfy all the maintained constraints, i.e. all the constraints in the combination.

Note that all the solutions identified in the algorithm have the same number of constraints. We assume that the cardinality of the extent against which the query is run is far greater than any of the exception lists annotating the optimization solutions. Therefore, there is no advantage in eliminating only a subset of the constraints in a solution in order to decrease the number of exceptions involved.

6.4.2 Heuristic H2

Assume that in the context of H2 the graph algorithm (section 6.3) results in the associations illustrated in figure 6.7. The constraints of the form $\mathtt{C}_{\mathtt{i}.\mathtt{R}_\mathtt{i}}$ are existing query constraints, while the

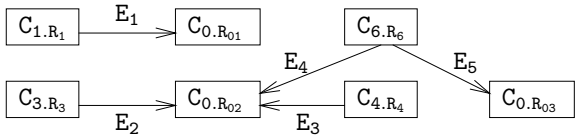

Figure 6.7: Correlations between query constraints and new indexed constraint

constraints $C_{0.R_{0j}}, j = 1, \ldots, 3$ are new constraints on the cluster index attribute. The additional subscript j is needed because the constraints on the index attribute may have different ranges. We identify all possible constraint introduction solutions as follows:

Algorithm

For each constraint on the index attribute $C_{0.R_{0j}}$

Step 1 find the least expensive association from a query constraint to $C_{0.R_{0j}}$, i.e. find the incoming link $C_{i.R_i} \Rightarrow C_{0.R_{0j}}$ with the fewest exceptions. For example if the cardinalities of $E_1, \ldots, E_5$ are $n_1, \ldots, n_5$ and $n_4 \leq n_2 \leq n_3$, then the least expensive rule for $C_{0.R_{0j}}$ is $C_{6.R_6} \Rightarrow C_{0.R_{02}}(E = E_4)$.

Step 2 filter out all the exceptions that do not satisfy at least one of the constraints $C_{1.R_1}, \ldots, C_{12.R_{12}}$. We do not need to test the exceptions for the antecedent of the corresponding rule, since they satisfy it by definition.

The constraint introduction solutions identified in our example are the following:

Introduced Constraints	*Exceptions*
$C_{0.R_{01}}$	$\{e \mid e \leftarrow E_1, C_{2.R_2}(e), \ldots, C_{12.R_{12}}(e)\}$
$C_{0.R_{02}}$	$\{e \mid e \leftarrow E_4, C_{1.R_1}(e), \ldots, C_{5.R_5}(e), C_{7.R_7}(e), \ldots, C_{12.R_{12}}(e)\}$
$C_{0.R_{03}}$	$\{e \mid e \leftarrow E_5, C_{1.R_1}(e), \ldots, C_{5.R_5}(e), C_{7.R_7}(e), \ldots, C_{12.R_{12}}(e)\}$

6.5 Optimizing OQL queries

In the previous sections, a series of algorithms were given to find a collection of constraint elimination or constraint introduction solutions. In this section, we show how the original query is transformed to its optimized form using the optimal solution. Consider the OQL query:

```
select x
from Extent_X as x
where C1.R1 and ... and Cn.Rn
```

6.5.1 Heuristic H1

Let $< \{C_{i1}, \ldots, C_{im}\}, E_i >$ be the maintained constraints and the exceptions of a solution i. Only the maintained constraints of the optimization solution should be tested on the objects of the whole extent; however, all the constraints should be tested on the exception cases and the objects that satisfy the maintained constraints but not the ones omitted should be removed from the result. Hence, the original query should be converted to the following one:

```
(select x from Extent_X as x
 where C_i1 and ... and C_im)
    except E_i
```

If we assume that tests on the query constraints take roughly the same time, the optimal solution is the one with fewest exceptions E_i.

6.5.2 Heuristic H2

Let $< C_{0.R_{0i}}, E_i >$ be the index constraint and the corresponding exceptions of a constraint introduction solution. Instead of testing the query constraints $C_{1.R_1}, \ldots, C_{n.R_n}$ on the entire extent Extent_X, we apply them only on the results of the subquery

```
select x
from Extent_X as x          (Q_0i)
where C_0.R_0i
```

Since $C_{0.R_{0i}}$ is a constraint on a cluster index attribute, the select operation is expected to be quite fast. The original query is transformed to its more efficient form:

```
(select x
 from Q_0i as x
 where C_1.R_1 and ... and C_n.R_n)
    union E_i
```

The exceptions E_i are merged to the result because they satisfy the query constraints, but not the new index constraint $C_{0.R_{0i}}$. The optimal solution is the one that introduces the index constraint with the highest selectivity and the least exceptions, i.e. the solution that minimizes $|Q_{0i}| + |E_i|$.

6.6 Discussion

We now look at two different scenarios, and estimate the extent to which heuristics H1 and H2 speed up query execution.

The first scenario concerns frequently executed queries. Assume that the association rules which are used by algorithm 6.3 are not modified. We may optimize a query once at compilation time, then execute its optimized form. It is worth optimizing provided that the execution time of the optimized query is less than the execution time of the original query. For heuristic H1 this happens only if the time saved by omitting some constraints is greater than the time needed to remove the exceptions from the result (`except` operation). The more the eliminated constraints and the fewer the exceptions, the better the optimization. It is worth pointing out that the time saved by the elimination of constraints is CPU-related. Since query execution is dominated by data access time, this optimization is not expected to alter performance significantly. It would help only in contexts rich in associations with few exceptions, in which users express many constraints in their queries.

Heuristic H2 is expected to bring more significant benefits. Firstly, this optimization involves a `union` operation, which is much cheaper than the `except` operation used in H1. Secondly, instead of retrieving all the objects of an extent from the database, we need only look at the subset retrieved through an indexed constraint. Hence, we save a considerable amount of data access time, spending a negligible amount of CPU time in evaluating the additional constraint.

The second scenario concerns queries which are executed only once. In this case, the time required for optimization is important. This time depends on the algorithm that finds associations between relaxed constraints and tight constraints (see section 6.3), since this is the most expensive step in the optimization process. This algorithm finds paths in a directed graph, and combines the exceptions associated with each edge of the path to derive the total exceptions for the path. In chapter 7 we propose a number of optimizations for this algorithm and study its complexity in detail.

6.7 Conclusion

The use of association rules for query optimization is relevant to both relational and object-oriented database systems. There has been a lot of research on generating association rules and maintaining them in the presence of updates. Research has also focused on finding heuristics that take advantage of rules in order to optimize a query. Most of this work ([CGM90, GGMR97]) has considered integrity rules, rather than association rules with exceptions. Semantic optimization heuristics were also applied without considering indirect associations. In this chapter, we propose algorithms that apply two optimization heuristics presented by Siegel et al., taking account of both exceptions and indirect associations. We show how to use these heuristics to optimize an OQL query.

The complexity of the optimization process is closely related to the complexity of the constraint graph, which represents the set of association rules in the data. It also depends on the number of exceptions associated with each rule. In chapter 7, we study these dependencies in detail and set up an experimental framework to assess the benefits of the semantic optimization techniques. We

also propose a number of changes to the proposed algorithms that are expected to reduce the cost of semantic optimization significantly.

Chapter 7

Complexity and Optimization

The constraint introduction and elimination heuristics suggest the use of association rules to optimize queries semantically (chapter 6). In this chapter we assess the complexity of the algorithm that navigates over the existing association rules and identifies indirect paths from sources to target constraints along with their exceptions. In particular, we look at its first two steps, since the cost of the third step is negligible. Recall that step 1 backtracks from a target constraint in order to find incomplete paths from sources to targets. The path annotations resulting from step 1 are then combined in step 2 to form (complete) direct or indirect paths along with their exceptions.

Studying the cost of these steps does not only tell us whether it is worth executing the algorithm. It also helps us to locate expensive parts that could possibly be optimized, deferred, or even completely avoided. In this chapter, we propose a number of optimizations that reduce the cost of the algorithm significantly.

Having implemented the optimized version of the algorithm, we set up a number of experiments to measure the benefits and the cost of exploiting semantic knowledge during query execution. In our experimental framework, we use the algorithm in order to add index constraints in the `predicate` of a `Select` algebraic operator. Consequently, we are interested in associations that relate the constraints of a query with any constraint on the existing index attribute. By running a variety of queries and changing the cluster index attribute on which data organisation is based, we manage to monitor the behaviour of the algorithm under different input conditions.

7.1 Experimental Framework

The theoretical analysis of the cost of steps 1 and 2 is supported by experimental results of running OQL queries in the following context. Queries are executed on a Pentium III processor, running at 450 MHz with 256 Mbytes of SDRAM. The OQL processor, which has been developed for the

purposes of this thesis, consists of the following components:

- a type checker that ensures the typability of a query against a database schema [BT00],
- a module that converts OQL to calculus representation (chapter 4),
- a module that normalizes calculus expressions based on a set of rules (chapter 4),
- a module that converts calculus expressions to their algebraic forms (chapter 5),
- a semantic optimizer that takes advantage of direct or indirect association rules (with their exceptions) in order to add or eliminate constraints from a predicate of a `Select` algebraic operator (chapter 6),
- a physical execution module that takes the optimized algebraic plan and executes it.

This OQL processor has been implemented on top of the storage manager of POET OSS 6.1.

The data set to which queries are addressed has been downloaded from the site of the U.S. Census Bureau (Department of Commerce). Each object represents a respondent to a census survey with 48 attributes. We have downloaded the survey data in the period 1973-1994 which consists of approximately 64,000 objects.

We also implemented a rule mining module based on the algorithms by Agrawal et al. [AIS93] and Srikant et al. [SA96]. This module identifies association rules of the form

$$\mathtt{attr}_1 \in \mathtt{R}_1 \wedge \ldots \wedge \mathtt{attr}_n \in \mathtt{R}_n \Rightarrow \mathtt{attr}_0 \in \mathtt{R}_0 \ (\mathtt{Exc})$$

That is, for all objects (except those in `Exc`) such that the value of $\mathtt{attr}_1$ is in range $\mathtt{R}_1$ and ... and the value of $\mathtt{attr}_n$ is in range $\mathtt{R}_n$, it is expected that the value of $\mathtt{attr}_0$ be in $\mathtt{R}_0$. The objects that do not satisfy the rule above, i.e. that satisfy the antecedent condition without satisfying the consequent, are included in the set of exceptions `Exc`.

Each rule has a certain confidence and support, which reflect its strength and popularity in the dataset. When running the rule mining algorithm, we may specify the minimum values of support and confidence for the derived rules. Depending on these values, the rule mining algorithm generates different sets of association rules along with their exceptions. For instance, low values of confidence and support are expected to yield numerous rules with many exceptions on average, and high connectivity of the graph of constraints that represents the association rules.

For each experiment to be presented in the remainder of this chapter, we specify the confidence and the support that were passed as parameters to the rule mining algorithm, as well as the number of resulting association rules. Both the strength and the number of existing association rules affect significantly the cost of semantic optimization of OQL queries, as will become evident in the complexity analysis.

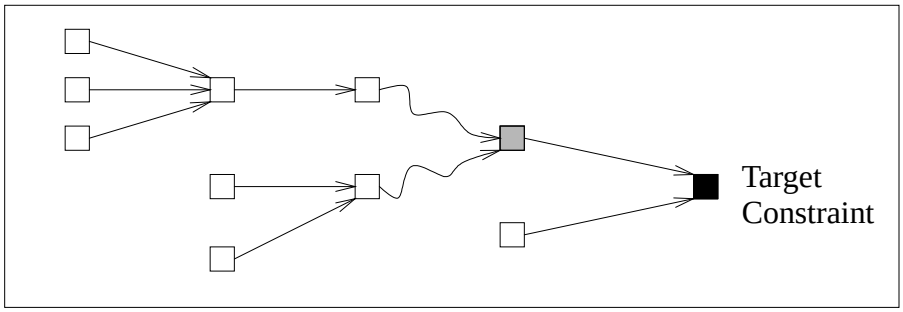

Figure 7.1: Graph of constraints forming a tree towards the target constraint

The rule mining algorithm and the OQL processor have been implemented in Java (JDK 1.2.2). In order to evaluate the cost of the semantic optimizer, we use a Java profiling tool called OptimizeIt. This tool provides the means of monitoring both the memory and the CPU usage of a Java program. We use its CPU profiler to measure the cost of running the OQL processor, paying particular attention to the semantic optimizer. The monitored times are slightly greater than the real times, because of the small overhead imposed by the CPU profiler. Time measurements are also slightly inconsistent in multiple executions of the same query. This is mainly due to the random effect of the garbage collection mechanism of the Java virtual machine and the caching mechanism of the object storage manager. In order to handle this problem, we execute each query at least five times. Hence, each time measurement in the experiments of this chapter is the average of the times monitored during multiple executions of the same query.

Despite the inaccuracy of time measurements the experimental results provide sufficient evidence to support the theoretical claims about the complexity of the semantic optimization algorithm.

7.2 Complexity of step 1

7.2.1 Initial version of step 1

This step backtracks from each target constraint, in order to find all (incomplete) paths from sources to targets. This problem is equivalent to finding all maximum-length acyclic paths leading to the target constraints.

If the graph of constraints had the form of a tree as illustrated in figure 7.1, then step 1 would access the edges of the graph (E) at most once for each target constraint. The target constraints are at most as many as the constraints of the graph (V). Hence, in the worst case, the complexity of this step would simply be $O(V * E)$.

However, in the general case, the backtracking algorithm traverses certain edges of the graph a

number of times. In order to evaluate the cost of step 1, we need to evaluate the maximum number of edge traversals given a graph with V vertices and E edges.

In appendix C we show that given a number of vertices V, the maximum number of edge traversals is given by the following formula:

$$\mathtt{TE} = \sum_{\mathtt{i}=1}^{\mathtt{V}-1}(\prod_{\mathtt{j}=1}^{\mathtt{i}}(\mathtt{V}-\mathtt{j})) \tag{7.1}$$

We also show that given a number of edges E, the maximum number of edge traversals is:

$$\mathtt{TE} = (\mathtt{m}+1) + (\mathtt{m}+1)(\mathtt{m}) + (\mathtt{m}+1)(\mathtt{m})(\mathtt{m}-1) + \ldots + (\mathtt{m}+1)! \tag{7.2}$$

where $\lceil \mathtt{m} = (-3+\sqrt{1+8\mathtt{E}})/2 \rceil$.

The maximum number of edge traversals given both V and E is derived based on formulae 7.1 and 7.2.

$$\mathtt{TE}(\mathtt{V},\mathtt{E}) \leq \mathtt{min}\{\mathtt{TE}_1(\mathtt{V}), \mathtt{TE}_2(\mathtt{E})\} \tag{7.3}$$

The cost of step 1 is linear in the number of traversed edges (TE). At each edge of the graph of constraints a path annotation of average length ℓ is propagated, i.e. it is copied and extended with one more constraint. Since $\ell \ll \mathtt{TE}$, we may deduce that the cost of step 1 is $\mathtt{O}(\mathtt{TE})$.

In practice, the number of edges traversed in step 1 (TE) is much smaller than the theoretical maximum given in 7.3. TE does not depend only on the number of edges and the number of vertices; it is very strongly related to the index (target) constraints and their connectivity to the rest of constraints. For instance, consider a graph of constraints with 51 vertices and 223 edges, that corresponds to 186 association rules with minimum confidence 73% and support 31.25%. Using as index the attribute `race` led to step 1 traversing 1,670,103 edges. However, using as index the attribute `spwrkslf` resulted in traversing 2,098,510 edges. Given a set of association rules, the number of TE varies with the selected index, but remains the same for each query execution. Hence, the optimizer can estimate the complexity of step 1 by optimizing a single query, and select a different index attribute in case the cost is not acceptable.

7.2.2 Optimization of Step 1

The great number of traversed edges makes the execution of step 1 prohibitively expensive. The original version of step 1 is inefficient since it may traverse a certain edge of the graph of constraints multiple times (at most as many as the number of path annotations). Another reason is that it only stops the backtracking procedure when it finds a constraint already included in the path or a constraint

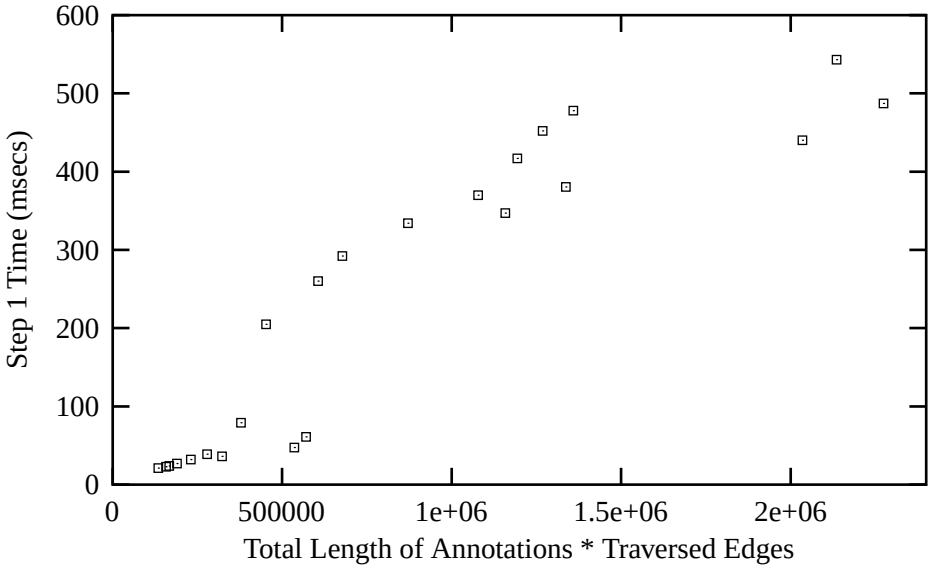

Figure 7.2: Cost of finding incomplete paths from sources to targets in step 1

without any incoming links. However, it would be sufficient to stop when a source constraint is encountered.

Based on these two remarks, we propose an optimized version of step 1. We start from a target constraint C_T and we apply the following steps recursively.

1. Annotate the current constraint C_{cur} as *visited*.
2. If C_{cur} is a source constraint, annotate it with a path consisting only of itself.
3. Otherwise, for each constraint C_i such that there is an edge from C_i to C_{cur},
   - if C_i is not visited apply the procedure on it recursively. This results in C_i being annotated with zero or more paths from sources to C_i. (If C_i has already been visited then it has already been annotated with these paths.)
   - Extend each one of these paths with the current constraint C_{cur} and add the new paths to the annotations of C_{cur}.

The algorithm above shows that backtracking is interrupted when a source constraint is encountered. The use of the flag *visited* prevents us from traversing the same edge more than once. The downside is that all constraints in the paths from sources to targets are annotated with paths from sources to themselves. These annotations are first created and stored in the source constraints; they are gradually extended and propagated towards the targets.

Let us consider the cost of the new version of step 1. This does not depend only on the number of traversed edges, but also on the number and the length of path annotations propagated across the graph of constraints. Propagating a number of annotations across an edge implies copying them and extending each one of them with an additional constraint. The cost of copying the annotations at each stage (across a certain edge) is bounded by the total length of path annotations derived as a result of step 1 (TLPA). If TE is the number of edges traversed during the execution of step 1, the cost of the latter should be $O(TLPA * TE)$. Figure 7.2 shows the cost of the optimized version of step 1 for 23 queries, which are optimized by adding constraints on different index attributes. The graph of constraints is generated by 186 association rules with minimum confidence 73% and support 31.25%. The optimization of the same 23 queries have been used to monitor times relevant to following experiments (figures 7.2, 7.3, 7.4, 7.5, 7.6, 7.7). The optimized version of step 1 is considerably faster than the original one.

7.3 Complexity of Step 2

As a result of the first (backtracking) step the source constraints are annotated with paths leading to the target constraints. Based on these path annotations, the second step of the algorithm performs two operations iteratively. It first absorbs all simple constraints in the paths until a pair of simple-composite constraints is reached. It then finds combinations of annotations having the same composite constraint, such that the preceding simple constraints are all the conjuncts of the composite one. If new annotations result from the combinations they are added to the existing annotations and the two substeps are repeated until no more combinations are generated.

An important issue that should be stressed is the handling of exceptions during the iteration of the two substeps. Substep 1 absorbs the simple constraints in the path annotations, by going through the edges (i.e. the direct associations) that link them. The exception sets associated with each edge are taken into consideration whilst going through the edges. If constraint A leads to constraint B based on a rule with exceptions $E_{A \Rightarrow B}$, and B leads to C based on a rule with exceptions $E_{B \Rightarrow C}$, then the exception set of the indirect path $A \rightarrow C$ (or the indirect rule $A \Rightarrow C$) is:

$$\begin{aligned} E \;=\; & \{exc | exc \leftarrow E_{A \Rightarrow B},\ \mathtt{not}\ C(exc)\} \\ & \cup \{exc | exc \leftarrow E_{B \Rightarrow C},\ A(exc)\} \end{aligned} \tag{7.4}$$

Substep 2 combines path annotations that pass from the same composite constraint through different conjuncts. Doing so, it also combines the exception sets evaluated so far for each of the annotations; that is, it creates the union of the exceptions evaluated so far for each combined path that

satisfy the source constraints of the other combined paths. If $\mathtt{C} = \mathtt{C}_1 \wedge \mathtt{C}_2$,

$$\begin{aligned}\mathtt{E}(\mathtt{C}_\mathtt{A} \wedge \mathtt{C}_\mathtt{B} \Rightarrow \mathtt{C}) = \\ \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_\mathtt{A} \Rightarrow \mathtt{C}_1), \mathtt{C}_\mathtt{B}(\mathtt{e})\} \cup \\ \{\mathtt{e} | \mathtt{e} \leftarrow \mathtt{E}(\mathtt{C}_\mathtt{B} \Rightarrow \mathtt{C}_2), \mathtt{C}_\mathtt{A}(\mathtt{e})\}\end{aligned} \tag{7.5}$$

The original algorithm implies the application of the two formulae (7.4, 7.5) at each iteration of the substeps 2.1 and 2.2.. This results in filtering many exception sets against various constraints, even if the resulting path annotations do not eventually lead to complete paths from a source to a target constraint. It is indicative to say that measuring the time of step 2 in the context of rules with 90 percent confidence, the prevailing cost of both substeps was the filtering of exceptions. This is because filtering implies resolving the references of the exceptions, fetching the actual instances from the database and checking their values against certain constraints.

To avoid this overhead we optimize step 2 by deferring the filtering of exceptions as much as possible. Instead of applying formulae 7.4 and 7.5, we just accumulate the sets of exceptions both while absorbing simple constraints (substep 2.1) and while combining path annotations (substep 2.2). This is a relatively cheap operation since it does not involve resolving exceptions, and therefore accessing the database.

The resolution and filtering of exceptions will take place only for those exceptions that are relevant to the complete paths from sources to target constraints. Hence, annotations that do not form complete paths should not burden the optimization process with exception filtering. Further, the exceptions do not need to be tested against the intermediate constraints of the path annotations, as is implied by the formulae 7.4 and 7.5, but only against the sources and the target constraints.

The optimization of step 2 - by deferring exception filtering and actually doing it only against the sources and the target constraints - led to significant performance improvements. In the remainder of this section, we reason about the complexity of step 2 (in its optimized form) and test its performance by running a series of queries.

7.3.1 Substep 2.1 - Absorbing simple constraints in path annotations

The nature of substep 2.1 denotes that its cost should be linear to the number of absorbed constraints. This is explained by the fact that absorbing each constraint, by traversing a simple edge, involves a constant cost as is explained below. Each annotation is associated with a set of exception sets which is initially empty. Absorbing a constraint involves i) traversing a simple edge (i.e. a direct association rule) and ii) adding the exception set of this edge to the set (of exception sets) associated with the annotation.

The linearity of the relation between the number of absorbed constraints and the cost of step 2.1 is demonstrated in figure 7.3.

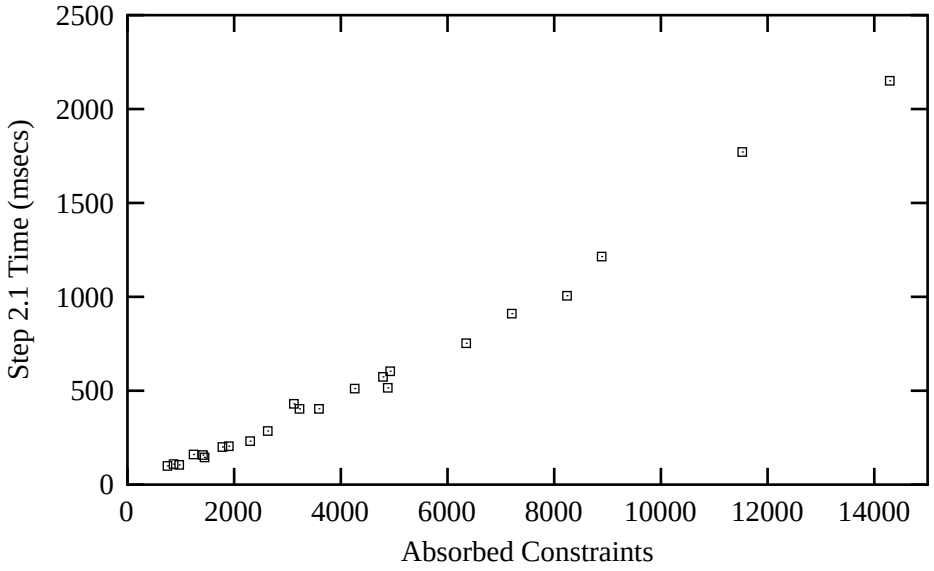

Figure 7.3: Relating the cost of step 2.1 to the number of absorbed constraints

Proposition 1 *The number of absorbed constraints is not bounded by the total length of path annotations.*

Proof. This proposition is easy to prove by the use of a simple example. Consider six path annotations containing the same composite constraint. In two of them the composite constraint is preceded by its first conjunct; the other four annotations reach the composite constraint through its second conjunct.

$$\mathtt{C}_1 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_1} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$
$$\mathtt{C}_2 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_1} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$
$$\mathtt{C}_3 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_2} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$
$$\mathtt{C}_4 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_2} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$
$$\mathtt{C}_5 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_2} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$
$$\mathtt{C}_6 \rightarrow \ldots \mathtt{C}_{\mathtt{conj}_2} \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_7 \rightarrow \mathtt{C}_8 \rightarrow \mathtt{C}_9$$

Combining these annotation results in eight new annotations. Hence, the edges following the composite constraint will be traversed eight instead of six times. The total traversed edges will obviously be more than the total length of path annotations.

Proposition 2 *The number of absorbed constraints is bounded by the following formula:*

$$(\mathtt{New\_Annotations} + \mathtt{Initial\_Annotations}) \times \mathtt{Composite\_Distance},$$

where `Composite_Distance` *is the maximum number of consecutive simple constraints preceding or following a composite constraint.*

Proof. `New_Annotations` is the number of paths generated by combining existing annotations in step 2.2. These along with the `Initial_Annotations`, generated by the backtracking algorithm in step 1, are all the annotations to which the absorbing algorithm may apply. The maximum number of constraints absorbed in each annotation is `Composite_Distance`. Hence, the number of absorbed constraints never exceeds

$$(\texttt{Combined\_Annotations} + \texttt{Initial\_Annotations}) \times \texttt{Composite\_Distance}$$

7.3.2 Substep 2.2 - Combining path annotations

Substep 2.2 involves the following costs:

Sorting Cost: The path annotations resulting from substep 2.1 have a simple constraint at their first position and a composite one at their second. It would be convenient to sort these annotations in order to facilitate their combination. The initial implementation of step 2.2 sorted paths based first on the composite and then on the simple constraints. It then found combinations of annotations satisfying two conditions: i) their initial composite constraint should be the same and ii) the simple constraints preceding the common composite constraint should form all its conjuncts. This method had two basic disadvantages. First it resulted in great numbers of new annotations that needed to be sorted and then combined. Second, the new annotations contained many duplicates that needed to be removed. In short, this method made the cost of step 2.2 prohibitively expensive.

A simple observation optimized the implementation of step 2.2 significantly. *A number of paths should be combined only if they are identical from their first composite constraint onwards.* For instance, say we have the following path annotations:

$$\begin{aligned}
&\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_3 \rightarrow \mathtt{C}_4 \rightarrow \mathtt{C}_5\\
&\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_3 \rightarrow \mathtt{C}_4 \rightarrow \mathtt{C}_5\\
&\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_5\\
&\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_5\\
&\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_6 \rightsquigarrow \mathtt{C}'_{\mathtt{comp}} \rightarrow \mathtt{C}_5\\
&\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_6 \rightsquigarrow \mathtt{C}'_{\mathtt{comp}} \rightarrow \mathtt{C}_5
\end{aligned}$$

In the original implementation the first annotation would be combined with the second, the third and the sixth, since they have the same first composite constraint and different simple constraints (conjuncts) preceding the composite one. However in the current implementation, the first annotation is combined only with the second, because it has the same body from the first composite constraint onwards. This method of combining annotations yields a smaller number of new annotations, without limiting the ability to find indirect association rules. It also avoids the generation of duplicate new annotations.

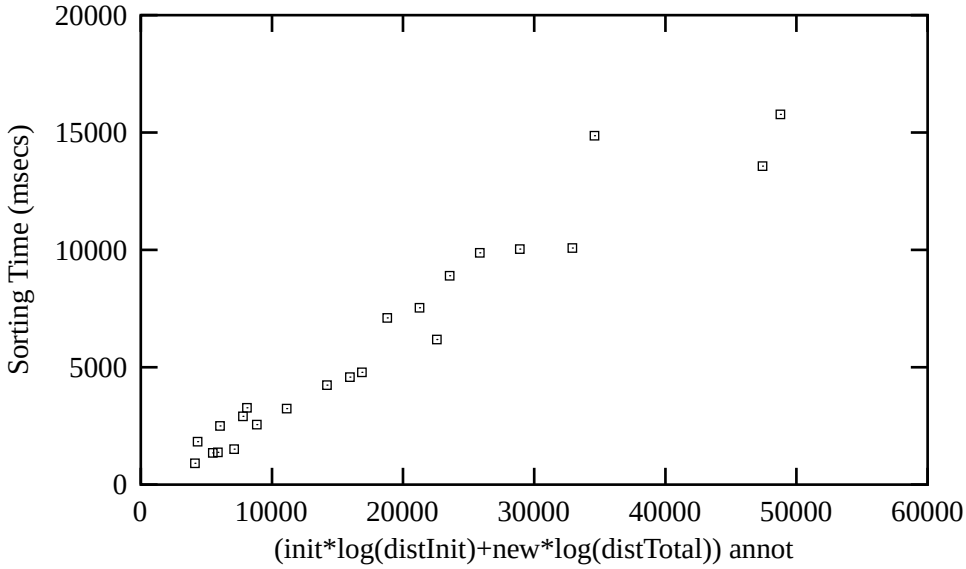

Figure 7.4: Cost of sorting initial and new path annotations

In order to facilitate combining paths, we sort them based on:

1. the number of composite constraints in each of them (desc)
2. their body from the first composite constraint onwards (asc)
3. the simple constraint preceding the first composite constraint (asc)

The paths of the previous example would be sorted as follows:

$$\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_6 \rightsquigarrow \mathtt{C}'_{\mathtt{comp}} \rightarrow \mathtt{C}_5$$
$$\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_6 \rightsquigarrow \mathtt{C}'_{\mathtt{comp}} \rightarrow \mathtt{C}_5$$
$$\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_3 \rightarrow \mathtt{C}_4 \rightarrow \mathtt{C}_5$$
$$\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_3 \rightarrow \mathtt{C}_4 \rightarrow \mathtt{C}_5$$
$$\mathtt{C}_1 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_5$$
$$\mathtt{C}_2 \rightsquigarrow \mathtt{C}_{\mathtt{comp}} \rightarrow \mathtt{C}_5$$

Consider the general case in which Step 1 yields $\mathtt{IA}$ initial annotations. At the first iteration, step 2.1 absorbs the single constraints preceding the first composite constraint of each path. The resulting annotations are inserted into a sorted vector and duplicate paths are eliminated. This results in a vector of $\mathtt{DA}$ distinct ordered annotations. The cost of inserting/sorting the initial annotations is $\mathtt{O}(\mathtt{IA} * \mathtt{log}(\mathtt{DA}))$. Step 2.2 then combines these annotations into new ones that are also inserted in the

ordered vector. The total number of annotations in the final sorted vector is $\mathtt{TA} = \mathtt{DA} + \mathtt{NA}$, where $\mathtt{NA}$ are the new annotations generated during all iterations of step 2.2. The cost of inserting/sorting the new annotations is therefore $\mathtt{O}(\mathtt{NA} * \mathtt{log}(\mathtt{TA}))$.

The total sorting cost of step 2.2 is $\mathtt{O}(\mathtt{IA} * \mathtt{log}(\mathtt{DA}) + \mathtt{NA} * \mathtt{log}(\mathtt{TA}))$(fig 7.4). We must take into account that comparing path annotations, in order to sort them, resembles comparing strings. The maximum number of operations per comparison is bounded by the maximum length of path annotations. If this is a constant $\mathtt{k} \ll \mathtt{IA} * \mathtt{log}(\mathtt{DA}) + \mathtt{NA} * \mathtt{log}(\mathtt{TA})$ then we may still assume that

$$\mathtt{Sorting\_Time} = \mathtt{O}(\mathtt{IA} * \mathtt{log}(\mathtt{DA}) + \mathtt{NA} * \mathtt{log}(\mathtt{TA})).$$

In order to predict the sorting cost of step 2, we need to evaluate $\mathtt{IA}, \mathtt{DA}, \mathtt{NA}$ and $\mathtt{TA}$. The number of initial annotations ($\mathtt{IA}$) is determined at the end of step 1, whilst the number of distinct annotations ($\mathtt{DA}$) is known by the end of the first iteration of step 2.1. If we managed to predict the number of new annotations ($\mathtt{NA}$), then the total anotations would be $\mathtt{TA} = \mathtt{DA} + \mathtt{NA}$.

Assume that after sorting the initial path annotations, we have

- $\mathtt{p}_1$ annotations containing ℓ_1 composite constraints,
- $\mathtt{p}_2$ annotations containing ℓ_2 composite constraints,
- ...
- $\mathtt{p}_\mathtt{n}$ annotations containing $\ell_\mathtt{n}$ composite constraints,

where $\mathtt{DA} = \mathtt{p}_1 + \ldots + \mathtt{p}_\mathtt{n}$.

We assume for simplicity that composite constraints consist of two conjuncts. If $\mathtt{s}$ is the number of source constraints, then each composite constraint can be reached by at most $2^\mathtt{s} - 1$ combinations of sources. The minimum number of paths that can generate $2^\mathtt{s} - 1$ combinations is $2 * \sqrt{2_\mathtt{s} - 1}$ ($\sqrt{2_\mathtt{s} - 1}$ paths reach the composite constraint through each one of its conjuncts). Hence, by combining $\mathtt{p}_1$ annotations, we derive at most $(\mathtt{p}_1 * \sqrt{2^\mathtt{s} - 1})/2$ new paths.

If $\ell_2 = \ell_1 - 1$ then the new paths $(\mathtt{p}_1 * \sqrt{2^\mathtt{s} - 1})/2$ are added to $\mathtt{p}_2$. Combining the annotations with ℓ_2 composite constraints ($\mathtt{p}'_2 = \mathtt{p}_2 + (\mathtt{p}_1 * \sqrt{2^\mathtt{s} - 1})/2$) results in at most $(\mathtt{p}'_2 * \sqrt{2^\mathtt{s} - 1})/2$ new paths.

If $\ell_2 < \ell_1 - 1$ then only $\mathtt{p}'_1 = (\mathtt{p}_1 * \sqrt{2^\mathtt{s} - 1})/2$ paths are combined into $(\mathtt{p}'_1 * \sqrt{2^\mathtt{s} - 1})/2$ new annotations. These have $\ell_1 - 2$ composite constraints.

This procedure continues in a similar way. If $\ell_1 = \ell_2 + 1 = \ldots = \ell_\mathtt{n} + \mathtt{n} - 1$ then

$$\mathtt{NA} = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{New\_Annot}_\mathtt{i} \tag{7.6}$$

where $\mathtt{New\_Annot}_\mathtt{i} = (\mathtt{p}'_\mathtt{i} * \sqrt{2^\mathtt{s} - 1})/2$ and $\mathtt{p}'_\mathtt{i} = \mathtt{p}_\mathtt{i} + \mathtt{New\_Annot}_{\mathtt{i}-1}$.

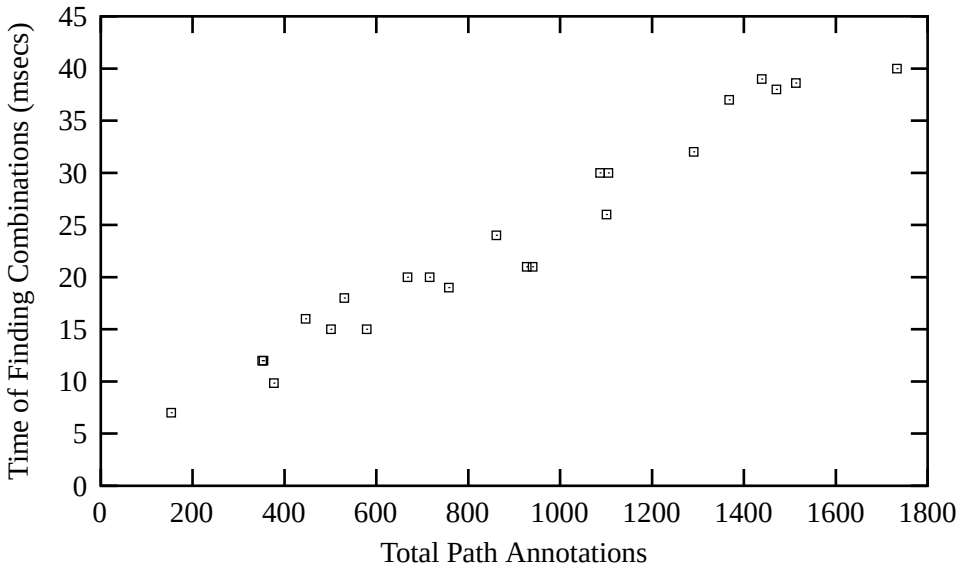

Figure 7.5: Cost of finding new combinations in step 2.2

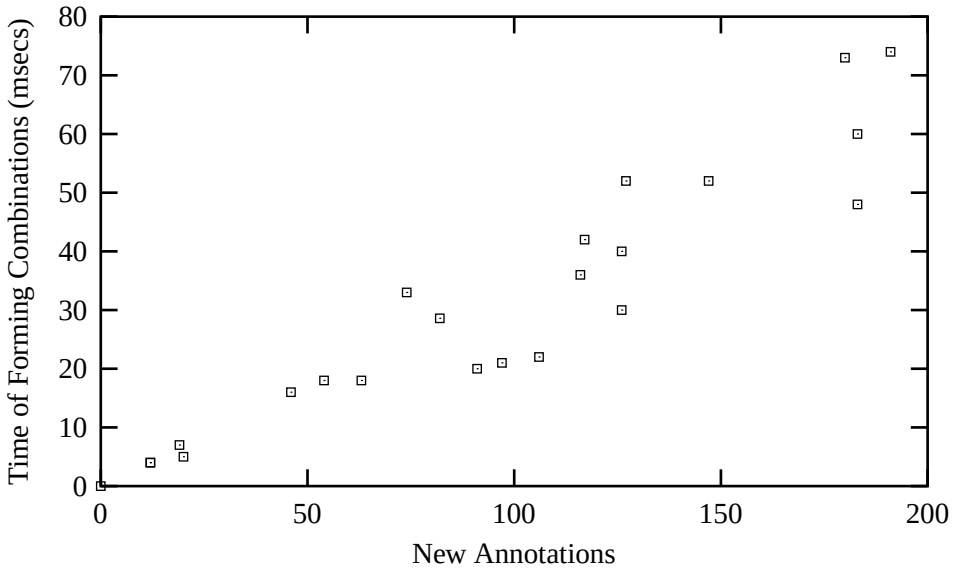

Figure 7.6: Cost of forming combinations in step 2.2

Cost of Combining Annotations: This cost involves iterating over the ordered vector of paths in order to combine annotations with the same initial composite constraint, different conjuncts preceding it and the same body from the composite constraint onwards. This is easy to do since the vector is already sorted based on these three criteria.

Consider a composite constraint as having $\mathtt{n}$ positions, one for each conjunct. By browsing the sorted vector of annotations, we can linearly determine which paths can fill the first, second, or $\mathtt{n}^{\mathtt{th}}$ position of the composite constraint. Groups of $\mathtt{n}$ annotations that can fill all different positions of a composite constraint are combined to form a new path annotation. Hence the cost of finding combinations is linear in the total number of path annotations $\mathtt{TA}$ (figure 7.5).

The total number of combinations equals the number of new annotations. For each combination the sources of the constituent paths are checked that they have compatible ranges. The exceptions of these paths are then merged to form the exceptions of the new path. The cost of forming combinations is therefore linear in the number of new annotations $\mathtt{NA}$ (figure 7.6).

The total cost of combining sorted annotations is given by a function of the form:

$$\mathtt{Combining\_Cost} = \mathtt{a} * \mathtt{TA} + \mathtt{b} * \mathtt{NA} + \mathtt{c},\ \mathtt{a}, \mathtt{b}, \mathtt{c} > 0$$

Based on this equation, we can predict the worst-case cost of combining annotations by predicting $\mathtt{NA}$ and setting $\mathtt{TA} = \mathtt{DA} + \mathtt{NA}$.

7.4 Combining steps 1 and 2 optimizes the algorithm significantly

Extending and propagating path annotations across the graph of constraints is the most expensive operation of step 1. Likewise, sorting path annotations in order to facilitate their combination is the dominant cost of step 2. It is possible to avoid both of these expensive operations by combining the functionality of steps 1 and 2.

In the new optimized algorithm, we avoid storing in each constraint $\mathtt{C_{cur}}$ (step 1) long paths from source constraints. Instead we store pairs of the form $(\mathtt{C_s}, \mathtt{SE})$, which means that there is a complete (indirect) path from the source constraint $\mathtt{C_s}$ to $\mathtt{C_{cur}}$ that involves the sets of exceptions $\mathtt{E_i} \in \mathtt{SE}$. Here $\mathtt{E_i}$, $\mathtt{i} = 1, \ldots, \mathtt{n}$ correspond to the sets of exceptions of the direct edges (rules) that form the path $\mathtt{C_s} \rightarrow \ldots \rightarrow \mathtt{C_{cur}}$. Based on this remark, we present the combined version of steps 1 and 2.

We start from a target constraint $\mathtt{C_T}$ and we apply the following steps recursively:

1. Annotate the current constraint $\mathtt{C_{cur}}$ as *visited*.

2. If $\mathtt{C_{cur}}$ is a source constraint, annotate it with the pair $(\mathtt{C_{cur}}, \{\})$.

3. Otherwise, if $\mathtt{C_{cur}}$ is a simple constraint (not a conjunction of constraints)

- for each constraint C_i such that there is an edge from C_i to C_{cur},
  - If C_i is not visited apply the procedure to it recursively. This results in C_i being annotated with zero or more pairs of the form C_s, SE (if C_i has already been visited then it has already been annotated with these pairs).
  - Let E_i be the set of exceptions of the association rule $C_i \rightarrow C_{cur}$.
  - Take copies of the annotations $\{(C_j, SE_j)\}$, $j = 1, \ldots, m$ of C_i and update their sets of exceptions with E_i: $SE'_j = SE_j \cup \{E_i\}$, $j = 1, \ldots, m$. Add the updated copies $\{(C_j, SE'_j)\}$ to the annotations of C_{cur}.
- Remove duplicate annotations of C_{cur}

4. Otherwise, (if C_{cur} is a composite constraint)

- for each conjunct C_i, $i = 1, \ldots, n$ such that $C_i \rightsquigarrow C_{cur}$,
  - If C_i is not visited apply the procedure to it recursively. This results in C_i being annotated with zero or more pairs of the form C_s, SE (if C_i has already been visited then it has already been annotated with these pairs).
- Combine the annotations of conjuncts C_i to form new annotations $\{(C'_j, SE'_j)\}$ that reflect complete paths that pass through all conjuncts of C_{cur}. For instance, if each conjunct C_i is annotated with a single pair (C_{s_i}, SE_i), these pairs are combined to the new annotation $(C_{s_1} \wedge \ldots \wedge C_{s_n}, SE_1 \cup \ldots \cup SE_n)$.
- Remove duplicate annotations of C_{cur}

The algorithm above propagates pairs of the form $(\texttt{source}, \texttt{exception\_sets})$ instead of long path annotations that include all intermediate constraints. There is also no need to sort path annotations before combining them, since combinations are formed on the fly at the composite constraint, as soon as the indirect paths from sources to its conjuncts have been identified. The cost of the algorithm depends on the number of edges traversed in the graph of constraints (TE). Note that TE is bounded by the total number of edges E in the graph. The length of path annotations no longer plays an important role (as in the previous version of step 1), since for each indirect path we store only its source and target constraint. The number of tuples $(\texttt{source}, \texttt{exception\_sets})$ propagated across each edge (or combined in composite constraints) is very small compared to the number of traversed edges. These tuples represent complete paths from sources to a particular constraint and are considerably fewer than the incomplete paths propagated across the graph in the previous version of the algorithm.

To be more specific, if s is the number of source constraints, the maximum number of annotations propagated across an edge (or stored in a constraint) is $2^s - 1$. For instance, given two sources C_1

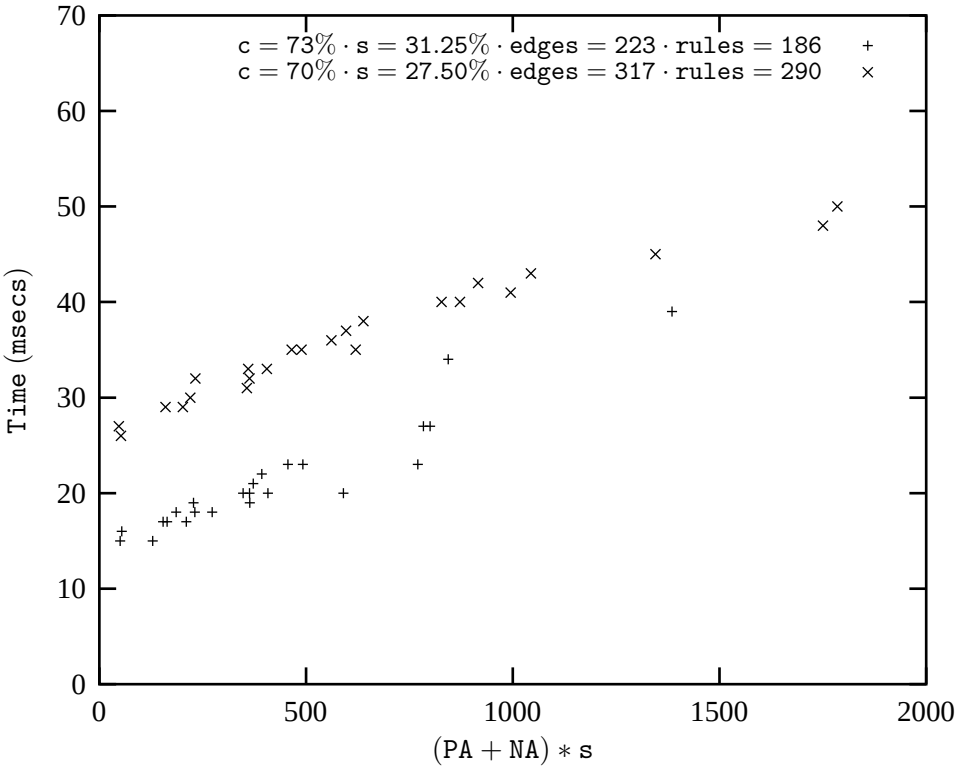

Figure 7.7: Cost of optimized algorithm

and C_2, the annotations of any other constraint can be at most $2^2 - 1 = 3$:

$$(C_1,\ E_1) \qquad (C_2,\ E_2) \qquad (C_1 \wedge C_2,\ E_3)$$

Consider PA as the total number of annotations propagated towards simple (non-composite) constraints. Assume that NA is the number of new annotations that are formed in composite constraints by combining the annotations of the conjuncts. For each one of the $PA + NA$ annotations, we evaluate the corresponding sets of exceptions. Then we add each tuple (C, E) to the annotations of a constraint ensuring that we eliminate duplicates. Since the maximum number of annotations per constraint is $2^s - 1$, the cost of forming annotations and removing duplicate ones is $O((PA + NA) * \log(2^s - 1)) = O((PA + NA) * s)$ (figure 7.7).

We monitored the cost of the algorithm by optimizing 23 queries (the same ones used in previous experiments) using first 186 association rules and then 290 rules (figure 7.7). The first set of rules was generated by running the rule mining algorithm with confidence 73% and support 31.25%. The second set was generated based on 70% confidence and 27.5% support. Optimizing queries using 186 association rules is cheaper than using 290 rules, since the generated graph of constraints has

fewer edges in the first case (223 edges) than in the second (317 edges). This implies that there are more complete paths from source constraints to a certain constraint C, i.e. C has more annotations. The cost of removing duplicate annotations, and hence the total cost of the algorithm, is increased. This explains why optimizing queries with the same value for $(\mathtt{PA} + \mathtt{NA}) * \mathtt{s}$ is more expensive when the number of association rules, and thus the number of edges, is increased. The factor s provides an upper bound for the cost of removing duplicates at each constraint, but in practice this factor depends on the connectivity of the graph.

If E is the number of edges in the graph of constraints then the total number of annotations propagated towards simple constraints is:

$$\mathtt{PA} \leq \mathtt{E} * (2^{\mathtt{s}} - 1)$$

If the graph contains comp composite constraints, each one having at most conj conjuncts, then the total number of annotations generated in composite constraints is:

$$\mathtt{NA} \leq \mathtt{comp} * (2^{\mathtt{s}} - 1)^{\mathtt{conj}}$$

Hence, the complexity of the new optimized algorithm is the following:

$$\mathtt{Cost} = \mathtt{O}((\mathtt{E} * (2^{\mathtt{s}} - 1) + \mathtt{comp} * (2^{\mathtt{s}} - 1)^{\mathtt{conj}}) * \mathtt{s}) \tag{7.7}$$

The number of sources s reflects the number of predicates in the where clause of the initial query. Hence, formula 7.7 shows that the complexity of the query affects the cost of optimization exponentially. The number of edges E, composite constraints comp and conjuncts per composite constraint conj reflect the connectivity of the graph of constraints. Hence, the optimization cost also depends on the number and the nature of association rules which are represented in the graph of constraints. For a particular query, we can bound the cost of the algorithm by selecting an appropriate set of association rules as the basis for the optimization.

7.5 Overall performance of optimizing queries semantically

In this chapter, we studied the complexity of the algorithm that navigates over a graph of constraints and finds direct or indirect paths along with their exceptions. Having implemented various versions of this algorithm, we used it in order to optimize queries semantically. Our intention was to identify association rules that would help us add index constraints in the where clauses of the queries. A series of time measurements on the various (sub)steps of the algorithm were carried out in order to shed light on the analysis of complexity.

Studying the complexity of the algorithm led us to a number of optimizations that improved its performance significantly. In the final version of the algorithm, the cost of steps 1 and 2 is negligible,

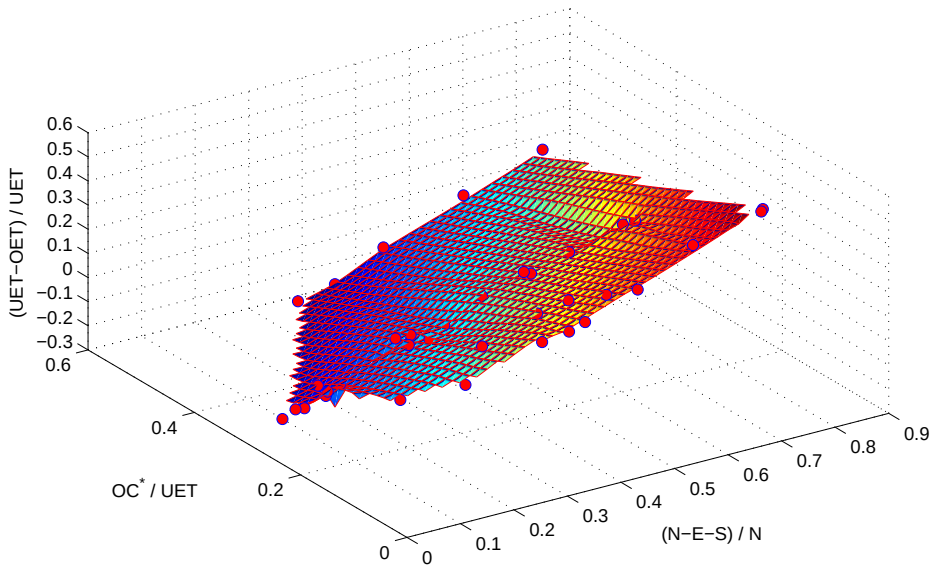

Figure 7.8: Relating the query execution benefits to the optimization cost and the evaluation benefits (top view)

compared to the operations that fetch objects from the database. These operations include i) fetching the association rules in order to generate the graph of constraints and ii) filtering the exceptions involved in the (indirect) paths that contribute to the query optimization. The cost of the first operation is ignored, since the OQL processor can load the association rules into a graph of constraints once in order to avoid burdening the execution of each individual query. Since the cost of resolving and filtering exceptions is linear in the number of exceptions, the optimizer decides whether to optimize a query semantically, as soon as it evaluates the number of exceptions $\mathtt{E_i}$ corresponding to each useful indirect association $\mathtt{i}$. The consequent (index) constraint of an association $\mathtt{i}$ can be used to retrieve $\mathtt{S_i}$ instead of $\mathtt{N}$ objects from the extent. The optimal association is the one that minimizes $\mathtt{E_i} + \mathtt{S_i}$. The optimizer applies this association to transform a query semantically only provided that $\mathtt{N} > \mathtt{E_{opt}} + \mathtt{S_{opt}}$. Otherwise, the optimization algorithm is interrupted and the query is executed in its initial form.

The surface in figure 7.8 demonstrates the benefits of the algorithm in the optimization of OQL queries. The x-axis $((\mathtt{N} - \mathtt{E} - \mathtt{S})/\mathtt{N})$ represents the percentage of objects that we avoided retrieving from the extent by introducing the index constraint. It shows the benefits of the optimization solution at evaluation time. The y-axis represents the fraction of the optimization cost to the unoptimized total execution time ($\mathtt{OC}^*/\mathtt{UET}$). Note that the optimization cost ($\mathtt{OC}^*$) does not include the cost of resolving exceptions from the database. It only involves loading the association rules into

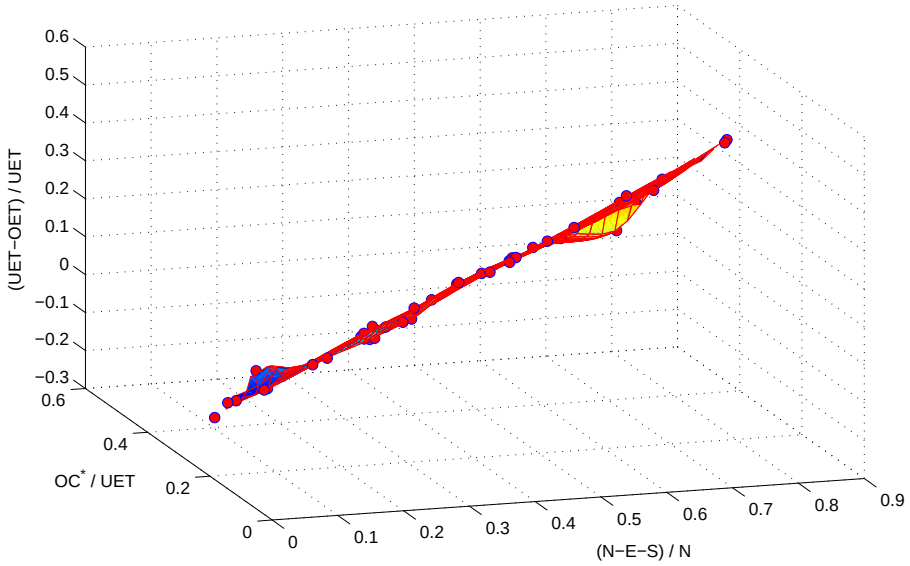

Figure 7.9: Relating the query execution benefits to the optimization cost and the evaluation benefits (1st side view)

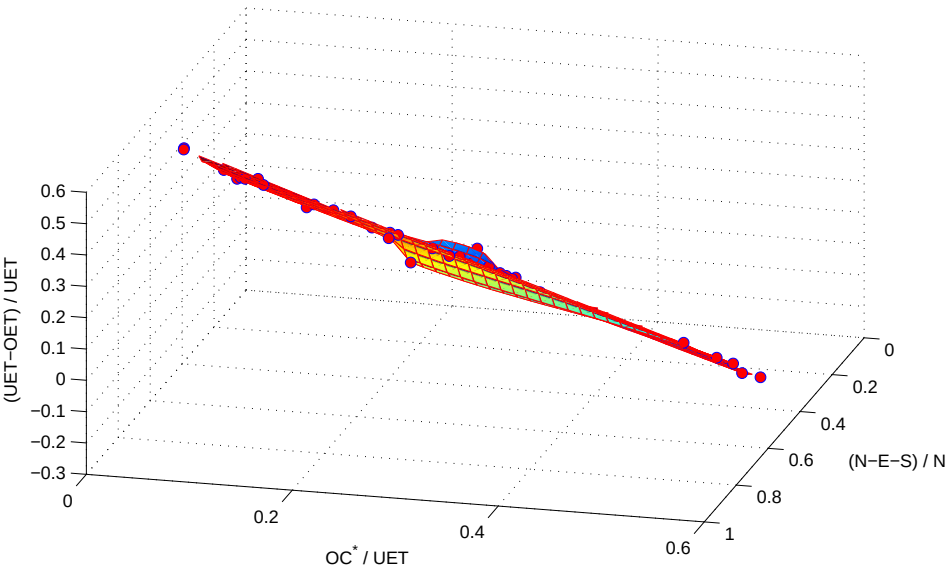

Figure 7.10: Relating the query execution benefits to the optimization cost and the evaluation benefits (2nd side view)

a graph of constraints and finding the constraint introduction solution. $\mathtt{OC}^*$ should be distinguished from the optimization cost measured in figure 7.7, which does not include the cost of loading rules in memory. The z-axis represents the percentage of the total execution time saved by the optimization $(\mathtt{UET} - \mathtt{OET})/\mathtt{UET}$. It is evident that as we decrease the cost factor ($\mathtt{OC}^*/\mathtt{UET}$) and increase the evaluation benefit $(\mathtt{N} - \mathtt{E} - \mathtt{S})/\mathtt{N}$, the total benefit of the optimization $(\mathtt{UET} - \mathtt{OET})/\mathtt{UET}$ is increased.

We optimized 40 queries using a variety of index constraints with different selectivities. We also used different sets of association rules by running the rule mining algorithm with the following values of confidence and support $\{(\mathtt{c} = 50\%, \mathtt{s} = 10\%),\ (\mathtt{c} = 60\%, \mathtt{s} = 10\%),\ (\mathtt{c} = 70\%, \mathtt{s} = 18.75\%),\ (\mathtt{c} = 73\%, \mathtt{s} = 25\%),\ (\mathtt{c} = 73\%, \mathtt{s} = 31.25\%)\}$. Although queries were evaluated over a small extent (4000 objects) the benefits of applying the constraint introduction heuristic were often significant. For large extents the cost factor $\mathtt{OC}^*/\mathtt{UET}$ is expected to be negligible and the total execution benefit to be linear in the evaluation benefit ($(\mathtt{N} - \mathtt{E} - \mathtt{S})/\mathtt{N}$). This is also the case for queries that are optimized once at compile time, and are then executed multiple times.

The side views of the same surface (figures 7.9 and 7.10) show that its shape is almost planar, as is expected theoretically:

$$\begin{aligned}
&\mathtt{Total_Optimization_Benefit} = \mathtt{Evaluation_Benefit} - \mathtt{Semantic_Optimization_Cost}\\
&\Rightarrow \mathtt{UET} - \mathtt{OET} = \mathtt{a} * (\mathtt{N} - \mathtt{E} - \mathtt{S}) - \mathtt{OC}^*\\
&\Rightarrow (\mathtt{UET} - \mathtt{OET})/\mathtt{UET} = \mathtt{a} * (\mathtt{N} - \mathtt{E} - \mathtt{S})/\mathtt{UET} - \mathtt{OC}^*/\mathtt{UET}\\
&\Rightarrow (\mathtt{UET} - \mathtt{OET})/\mathtt{UET} = \mathtt{a} * (\mathtt{N} - \mathtt{E} - \mathtt{S})/(\mathtt{a} * \mathtt{N} + \mathtt{b}) - \mathtt{OC}^*/\mathtt{UET}
\end{aligned}$$

where a reflects the time of fetching and processing a single object and b reflects the cost of checking the type of a query, translating it into its intermediate representations and applying syntactic transformations. If b is much smaller than the evaluation time ($\mathtt{a} * \mathtt{N}$), we may assume that $\mathtt{UET} \approx \mathtt{a} * \mathtt{N}$. Hence,

$$(\mathtt{UET} - \mathtt{OET})/\mathtt{UET} \approx (\mathtt{N} - \mathtt{E} - \mathtt{S})/\mathtt{N} - \mathtt{OC}^*/\mathtt{UET}$$

The anomalies of the surface in figures 7.8, 7.9 and 7.10 are partly caused by the inaccuracy of time measurements, as well as by the assumption $\mathtt{UET} \approx \mathtt{a} * \mathtt{N}$. The surface is the result of interpolating the irregularly sampled data to a regularly spaced grid and then plotting the interpolated and the non-uniform data (using MATLAB).

There are cases when the optimization has a negative effect on the query execution time. This may be due to either of the following reasons: i) the optimizer may have decided that the optimization is not worth applying, but cannot undo the optimization cost incurred so far; ii) the optimization was worth applying, but its benefits were not greater than the optimization cost. In both cases the negative effect is negligible, provided that the optimization cost does not include fetching the association rules from the database at each query execution.

7.6 Discussion

In chapter 6, we proposed an algorithm that navigates through a set of existing association rules, and identifies indirect associations between certain source and target constraints. This algorithm was proposed as part of the overall algorithm for optimizing queries using the constraint introduction (or elimination) heuristic. In this chapter, we reasoned about the complexity of the algorithm by looking at its functionality step by step. The complexity analysis highlighted aspects of the algorithm that needed to be improved and hence contributed to its optimization. Having implemented various versions of the algorithm we tested its performance by running a series of OQL queries. The benefits of applying the constraint introduction heuristic depend on the selectivity of the index constraint and the exceptions involved in the useful associations. Our experience is that this heuristic often reduces the time of query execution significantly. The constraint elimination heuristic is worth applying only when the associations involved have practically no exceptions.

Chapter 8

Conclusion and Future Work

In this dissertation we developed an OQL processing framework that encompasses all features of the object-oriented paradigm and is flexible enough to benefit from semantic knowledge in the form of association rules. This was achieved by extending previous work on type inference, calculus and algebraic intermediate representations, as well as on the use of semantic optimization heuristics. The proposed OQL framework is useful for a number of commercial or scientific applications that handle large amounts of complex data rich in semantic correlations (chapter 2). We give a summary of the contributions of this dissertation in the various stages of OQL processing.

8.1 Summary

We first proposed a type inference model (chapter 3) in order to ensure that an OQL query is well-typed. The inference algorithm, which is the core of this model, identifies the most general type of an OQL query in the absence of schema information; it also derives the minimum type requirements that a schema should satisfy to be compatible with this query. The expressiveness and the complex type system of OQL raised issues that could not be handled by previous work on type inference. Hence, by developing a type inference model for OQL, we gained a lot of experience in dealing with polymorphic collection types, structures and classes all being related through a complex subtype relationship and being manipulated using a variety of constructs. It was shown that the unification mechanism used to resolve constraints in previous models is not sufficient in our context. The inference rules defined in this thesis generate a variety of constraints specific to OQL and a mechanism is provided to resolve them. The proposed inference model should be useful in any distributed or even semi-structured database environment.

Having ensured the typability of an OQL query, we then investigated its translation into the monoid calculus intermediate representation (chapter 4). In the translation process, we identified

several weak points of ODMG OQL [CBB+00], e.g. the nondeterministic behaviour of the `order by` clause, and proposed ways of dealing with them. We pointed out that not all OQL queries reflect homomorphic mappings, and hence, their translation using monoid comprehensions is not always possible. Nonhomomorphic functions like `bagof`, `listof`, `arrayof` and `sortof` were introduced in order to resolve this problem. The resulting translation rules, which cover all OQL constructs, differ from those defined in related contexts [Afs98, FM00]. In [FM00], a set of normalization rules is defined in order to simplify calculus expressions. In this dissertation, we extended this set with additional rules and we proposed a normalization algorithm that determines the order of their application.

We then investigated the translation of calculus expressions to the monoid algebra (chapter 5). Following the approach of Fegaras and Maier [FM00], we adopted a set of algebraic operators (e.g. `Select`, `Join`) similar to the ones used in relational contexts. However, we realised that new operators should be introduced in order to translate calculus expressions containing nonhomomorphic functions. We also pointed out that nested comprehensions should be processed locally before being merged with the algebraic expressions generated in outer comprehensions. Our approach provides opportunities for further optimization compared to the approach in [FM00]. New algebraic operators were added in order to handle merge operations of collections. We highlighted the need for the `OuterSelect` operator, which enables the OQL processor to filter the results of a nested comprehension before processing its head. At the physical level, this implies reducing the volume of data to be processed at an early stage. Finally, by annotating all algebraic operators with monoid information, we provided opportunities for optimizing the physical execution plan. The translation rules proposed in this thesis provide a complete mapping from the monoid calculus to the algebraic query representation.

In chapter 6, we proposed an algorithm that uses association rules in order to optimize queries semantically. This algorithm is the main step towards applying the constraint introduction and elimination heuristics presented in [SSS92]. It is assumed that a rule mining algorithm is used in order to generate a set of association rules along with their exceptions. The number of derived rules depends on the minimum support and confidence that are passed as parameters. The exceptions to the rules, i.e. the objects that satisfy the antecedent but not the consequent, are maintained properly in the presence of database updates. Given a particular query, we showed how query predicates are mapped into source or target constraints found in association rules. We then proposed an algorithm that identifies direct or indirect correlations between source and target constraints along with their exceptions. Using these correlations, a set of optimization solutions is identified, and the optimal one is selected for the query transformation. Previous work on semantic optimization has mainly considered integrity rules rather than association rules with exceptions [CGM90, GGMR97]. There has also been no previous effort to identify indirect correlations that could be useful for semantic transformations. The

algorithm proposed in this thesis has two valuable properties: i) it is flexible enough to take advantage of weak or indirect correlations and ii) it can be used to apply a variety of heuristics besides the ones that suggest the introduction or elimination of constraints. For instance, the proposed algorithm is equally useful for the following two heuristics proposed by Siegel et al. [SSS92]:

- If a query contains a restriction on a range attribute, then try to locate a constraint on that attribute that is a contradiction to the existing restriction (contradiction heuristic).

- If a restriction on a range attribute implies the answer to a query then the latter can be answered without accessing the database (tautology heuristic).

In order to assess the benefits of semantic optimization, we evaluated the cost of the algorithm that finds indirect correlations between source and target constraints. The complexity of the algorithm was estimated from a theoretical point of view and was confirmed by optimizing a number of queries using the constraint introduction heuristic. The complexity analysis led us to identify expensive tasks of the algorithm that could be changed, deferred or avoided. A number of optimizations were proposed that reduced the cost of the algorithm considerably.

Our experience from implementing the algorithm showed that a significant amount of time could be saved if the association rules are loaded in memory (into a graph of constraints) once and not every time a query is optimized. Given a graph of constraints, the semantic optimizer navigates over its edges and identifies indirect associations that could be used to apply the constraint introduction heuristic. We decided that the exceptions involved in the indirect associations should be accumulated but not resolved, until the OQL processor decides on whether to apply the optimization. A query is optimized, if the sum of exceptions and objects retrieved using the index constraint is smaller than the cardinality of the extent. Hence, instead of resolving all objects from the original extent, the OQL processor only fetches a subset of them, and thus saves significant data-access time. If the optimization is not applicable, the query execution is burdened with a negligible CPU-related cost. It is shown that this cost is bounded by the number of edges in the graph of constraints.

The constraint elimination heuristic is worth applying when the number of exceptions involved in the optimization solution is very small. In a relational context, the cost of resolving these exceptions may easily outbalance the CPU-related benefits of applying the heuristic. However, in an object-oriented context, where constraints may potentially include expensive method invocations, the optimizer may decide to apply the heuristic despite the great number of exceptions involved.

In the case of the tautology and contradiction heuristics, a potential optimization solution is beneficial independent of the number of exceptions involved. The optimized queries can be executed by resolving only the exceptions of the solution, which are always fewer than the cardinality of the extent to which the query is addressed.

8.2 Future work

The areas studied in this dissertation include type inference of OQL, calculus representation, normalization of calculus expressions, algebraic representation and semantic optimization using association rules. This work can be extended in the last two phases of OQL processing, namely the algebraic level and the semantic optimization techniques.

In particular, in this dissertation we defined a complete mapping from calculus expressions to the monoid algebra. However, we did not consider syntactic transformations at the algebraic level. It would be interesting to identify a set of rules that optimize the algebraic plan, taking into account the new algebraic operators introduced in this thesis. Further work could focus on optimizing the physical execution plan based on the monoid information associated with the algebraic operators.

Future research might also concern the semantic optimization of OQL queries. In this dissertation we presented in detail all the steps of applying the constraint introduction and elimination heuristics (chapter 6). The most important step is an algorithm that navigates over a graph of constraints and finds direct or indirect associations along with their exceptions. The heuristics considered so far are applicable to both relational and object-oriented systems. Future work could investigate the use of the proposed algorithm to apply heuristics specific to the object-oriented paradigm. For instance, the query optimizer could benefit from association rules with the following kinds of constraints:

$$
\begin{array}{rcl}
\mathtt{C} & \equiv & \mathtt{x_1} \in \mathtt{extent} \\
& | & \mathtt{x_1.relationship} = \mathtt{x_2} \\
& | & \mathtt{x_1} \in \mathtt{x_2.relationship} \\
& | & \mathtt{x_1.attribute_1} < \mathtt{comp\_op} > \mathtt{x_2.attribute_2} \\
& | & \mathtt{x_1.method}(\mathtt{y_1}, ..., \mathtt{y_k}) < \mathtt{comp\_op} > \mathtt{x_2.attr} \\
& | & \mathtt{x_1.method}(\mathtt{y_1}, ..., \mathtt{y_k}) < \mathtt{comp\_op} > \mathtt{value}
\end{array}
$$

where $\mathtt{x_1}, \mathtt{x_2}$ represent objects and $\mathtt{y_1}, \ldots, \mathtt{y_k}$ may be instances of any OQL type (chapter 3). It is assumed that in the beginning of a rule, each variable $\mathtt{x_i}$ (or $\mathtt{y_i}$) is annotated with the most general type for which the rule is applicable.

Examples of rules containing constraints on the attributes, methods, relationships or extents of one or more objects are given below:

λ x : Employee.
x.salary $> 50,000 \Rightarrow$ x $\in$ Managers

λ x : Company.
x.revenues $>$ x.expenses $\Rightarrow$ x.profits() > 0

λ x : Person, y : Person.
x $\in$ Unemployed $\wedge$ y $\in$ Employees $\Rightarrow$ x.expenses $<$ y.expenses

λ x : Hotel, month : integer.
x.stars $= 3 \wedge$ month $< 9 \wedge$ month $> 5 \Rightarrow$ x.revenues(month) $> 200,000$

In order to benefit from such semantic knowledge, new data mining algorithms should be devised to identify correlations between the values of attributes, relationships and methods of objects belonging to the same or different classes. Mechanisms like the use of join indices should be considered in order to optimize the execution of mining algorithms [ST95, JS97]. Standard procedures for defining semantics about method invocations (e.g. relating parameters to their results or restricting the range of the results) are also worth studying.

Assuming that we have the means of deriving *complex* association rules, the next issue that arises is how we could possibly generate a graph of constraints that reflects these associations. For instance, information about the class hierarchy of a particular schema is necessary in order to add a link from a constraint x $\in$ $\texttt{extent}_1$ towards x $\in$ $\texttt{extent}_2$, where $\texttt{extent}_1 \subseteq \texttt{extent}_2$ ($\texttt{class}_1$ corresponding to $\texttt{extent}_1$ is a subclass of $\texttt{class}_2$ corresponding to $\texttt{extent}_2$).

Association rules have traditionally been used to define *positive* correlations between the values of data. For instance, if the values of an attribute A are in a range $\texttt{R}_1$ the values of attribute B are in range $\texttt{R}_2$. However, query optimization could take advantage of association rules that contain negations and thus express the fact that certain ranges of attributes never appear together [EGLM01]. It would be interesting to study how we could identify indirect associations with exceptions in the presence of such rules.

We might wish to define indirect relationships between objects, for example, if x.father $=$ y and y.father $=$ z then x.grandfather $=$ z. This is a typical rule that expresses intensional information in a deductive database. It could be implemented as a deductive method attached to a class in an object-oriented system [Sam00, SP00]. Hence, a node which represents the constraint x.grandfather $=$ z might be an abstraction of a subgraph which needs to be traversed. The nodes of the subgraph would be variables instead of constraints, whilst the edges would denote the relationships traversed. Combining known traversal techniques for the subgraphs [Pul96] with the traversal

techniques proposed in this thesis, we could take advantage of associations involving intentional and extensional information.

Given that the previous problems are solved, the proposed algorithm navigates over the graph of constraints and finds paths from source to target constraints. Identifying which constraints are used as sources and targets is another open problem, the answer to which depends on the optimization heuristic.

Finally, future research could consider combining data mining with warehousing techniques. If we were able to identify association rules that involve aggregate values (stored in a warehouse), then it would be possible to optimize expensive `select ... group by` queries [AHLS00].

8.3 Conclusion

This dissertation presents a framework for the efficient execution of OQL queries. Starting from inferring the type of an OQL query, we proceed to translate it into a calculus form, normalize it at this level and translate it into an algebraic representation. We then demonstrate how OQL queries can be optimized semantically with the aid of association rules. Many optimization heuristics, e.g. the constraint introduction and elimination heuristics, suggest the use of semantic knowledge expressed in (direct) integrity rules. We point out that optimization opportunities would be increased if we used indirect associations that are not necessarily 100% valid. An algorithm is proposed that navigates over existing association rules and identifies indirect associations along with their exceptions. We assess the cost and the benefits of this algorithm in the context of two optimization heuristics applicable in both object-oriented and relational systems. It is expected that the algorithm be equally useful for applying heuristics specific to the object-oriented paradigm. We implemented and tested the ideas reported in this thesis by developing an OQL engine and enhancing it with a semantic optimizer.

Appendix A

Inference Rules

This appendix presents the inference rules for OQL that have not been discussed in chapter 3 of the dissertation:

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \tau \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 \ \mathtt{and}\ \mathtt{q}_2 : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\sigma = \mathtt{bool}\} \wedge \{\tau = \mathtt{bool}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 \ \mathtt{union}\ \mathtt{q}_2 : \sigma \Rightarrow \mathcal{C}_2 \wedge \{\sigma = \mathtt{Merge\_Result}(\sigma_1, \sigma_2)\} \wedge \\ \{\mathtt{Equality\_Compatible}(\mathtt{Member\_Type}(\sigma_1), \mathtt{Member\_Type}(\sigma_2))\}\end{array}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 = \mathtt{q}_2 : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\mathtt{Equality\_Compatible}(\sigma_1, \sigma_2)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 < \mathtt{q}_2 : \mathtt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\mathtt{Greater\_Less\_Than\_Compatible}(\sigma_1, \sigma_2)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 + \mathtt{q}_2 : \sigma_0 \Rightarrow \mathcal{C}_2 \wedge \{\sigma_0 = \mathtt{Arith\_Result}(\sigma_1, \sigma_2)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \mathtt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \mathtt{q}_1 \ \mathtt{mod}\ \mathtt{q}_2 : \mathtt{int} \Rightarrow \mathcal{C}_2 \wedge \{\sigma_1 = \mathtt{int}\} \wedge \{\sigma_2 = \mathtt{int}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash -\mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1 \wedge \{\sigma \subseteq \mathtt{int/float}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{not\ q}\colon \mathtt{bool} \Rightarrow \mathcal{C}_1 \wedge \{\sigma = \mathtt{bool}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{abs(q)}\colon \sigma \Rightarrow \mathcal{C}_1 \wedge \{\sigma \subseteq \mathtt{int/float}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{min(q)}\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\phi \subseteq \mathtt{orderable}\} \wedge \{\mathtt{Member\_Type}(\sigma) = \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{sum(q)}\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\phi \subseteq \mathtt{int/float}\} \wedge \{\mathtt{Member\_Type}(\sigma) = \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{avg(q)}\colon \mathtt{float} \Rightarrow \mathcal{C}_1 \wedge \{\sigma \subseteq \mathtt{collection(int/float)}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{count(q)}\colon \mathtt{int} \Rightarrow \mathcal{C}_1 \wedge \{\sigma \subseteq \mathtt{collection(nonfunctional)}\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{distinct(q)}\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\phi = \mathtt{Distinct\_Result}(\sigma)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{listtoset(q)}\colon \mathtt{set}(\phi) \Rightarrow \mathcal{C}_1 \wedge \{\sigma = \mathtt{list}(\phi)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{element(q)}\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\mathtt{Member\_Type}(\sigma) = \phi\}}$$

Appendix B

Translating OQL to the monoid calculus

This appendix presents the full mapping from OQL constructs to their calculus forms. The notation $|\mathtt{q}|^*$ stands for the calculus equivalent of an OQL query $\mathtt{q}$ ($|-|^* : \mathtt{OQL} \rightarrow \mathtt{Calculus}$).

OQL Expressions	Calculus Expressions
$\mathtt{id}$	$\mathtt{id}$
$\mathtt{literal}$	$\mathtt{literal}$
$\mathtt{q.id}$	$\lvert\mathtt{q}\rvert^*.\mathtt{id}$
$\mathtt{q}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$	$\lvert\mathtt{q}\rvert^*(\lvert\mathtt{q}_1\rvert^*, \ldots, \lvert\mathtt{q}_n\rvert^*)$
$\mathtt{q}_1$ union $\mathtt{q}_2$	$\mathtt{merge}[\mathtt{M}](\lvert\mathtt{q}_1\rvert^*, \lvert\mathtt{q}_2\rvert^*)$, (to determine M see 4.2.3)
$\mathtt{q}_1$ intersect $\mathtt{q}_2$, ($\mathtt{q}_1$ or $\mathtt{q}_2 : \mathtt{bag}(-)$)	$\mathtt{bag}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*), \mathtt{some}\{\mathtt{x} = \mathtt{y} \parallel \mathtt{y} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_2\rvert^*)\}\}$
$\mathtt{q}_1$ intersect $\mathtt{q}_2$, ($\mathtt{q}_1$ and $\mathtt{q}_2 : \mathtt{set}(-)$)	$\mathtt{set}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \lvert\mathtt{q}_1\rvert^*, \mathtt{some}\{\mathtt{x} = \mathtt{y} \parallel \mathtt{y} \leftarrow \lvert\mathtt{q}_2\rvert^*\}\}$
$\mathtt{q}_1$ except $\mathtt{q}_2$ ($\mathtt{q}_1$ or $\mathtt{q}_2 : \mathtt{bag}(-)$)	$\mathtt{bag}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*), \mathtt{all}\{\mathtt{x} \neq \mathtt{y} \parallel \mathtt{y} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_2\rvert^*)\}\}$
$\mathtt{q}_1$ except $\mathtt{q}_2$ ($\mathtt{q}_1$ and $\mathtt{q}_2 : \mathtt{set}(-)$)	$\mathtt{set}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \lvert\mathtt{q}_1\rvert^*, \mathtt{all}\{\mathtt{x} \neq \mathtt{y} \parallel \mathtt{y} \leftarrow \lvert\mathtt{q}_2\rvert^*\}\}$
$\mathtt{first}(\mathtt{q})$	$\lvert\mathtt{q}\rvert^*[0]$
$\mathtt{last}(\mathtt{q})$	$\lvert\mathtt{q}\rvert^*[\mathtt{sum}\{1 \parallel \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\} - 1]$
$\mathtt{q}_1[\mathtt{q}_2]$	$\lvert\mathtt{q}_1\rvert^*[\lvert\mathtt{q}_2\rvert^*]$

OQL Expressions	Calculus Expressions
$\mathtt{set}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$	$\mathtt{merge}[\mathtt{set}](\mathtt{unit}[\mathtt{set}](\lvert\mathtt{q}_1\rvert^*), \ldots, \mathtt{unit}[\mathtt{set}](\lvert\mathtt{q}_n\rvert^*))$
$\mathtt{bag}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$	$\mathtt{merge}[\mathtt{bag}](\mathtt{unit}[\mathtt{bag}](\lvert\mathtt{q}_1\rvert^*), \ldots, \mathtt{unit}[\mathtt{bag}](\lvert\mathtt{q}_n\rvert^*))$
$\mathtt{list}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$	$\mathtt{merge}[\mathtt{list}](\mathtt{unit}[\mathtt{list}](\lvert\mathtt{q}_1\rvert^*), \ldots, \mathtt{unit}[\mathtt{list}](\lvert\mathtt{q}_n\rvert^*))$
$\mathtt{array}(\mathtt{q}_1, \ldots, \mathtt{q}_n)$	$\mathtt{arrayof}(\mathtt{merge}[\mathtt{list}](\mathtt{unit}[\mathtt{list}](\lvert\mathtt{q}_1\rvert^*), \ldots, \mathtt{unit}[\mathtt{list}](\lvert\mathtt{q}_n\rvert^*)))$
$\mathtt{dictionary}((\mathtt{q}_1, \mathtt{q}'_1), \ldots, (\mathtt{q}_n, \mathtt{q}'_n))$	$\mathtt{merge}[\mathtt{dict}](\mathtt{unit}[\mathtt{dict}](\lvert\mathtt{q}_1\rvert^*, \lvert\mathtt{q}'_1\rvert^*), \ldots, \mathtt{unit}[\mathtt{dict}](\lvert\mathtt{q}_n\rvert^*, \lvert\mathtt{q}'_n\rvert^*))$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{col}_1(\mathtt{set}(-)))$	$\mathtt{set}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*), \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{col}_1(\mathtt{bag}(-)))$	$\mathtt{bag}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*), \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{set}(\mathtt{list/oset/array}(-)))$	$\mathtt{set}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{bagof}(\mathtt{s})\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{bag}(\mathtt{list/oset/array}(-)))$	$\mathtt{bag}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{bagof}(\mathtt{s})\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{list/oset}(\mathtt{oset}(-)))$	$\mathtt{oset}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{array}(\mathtt{oset}(-)))$	$\mathtt{oset}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \mathtt{listof}(\lvert\mathtt{q}\rvert^*), \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{list}(\mathtt{list}(-)))$	$\mathtt{list}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{oset/array}(\mathtt{list}(-)))$	$\mathtt{bag}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \mathtt{list}(\lvert\mathtt{q}\rvert^*), \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{list/oset}(\mathtt{array}(-)))$	$\mathtt{array}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \mathtt{arrayof}(\lvert\mathtt{q}\rvert^*), \mathtt{x} \leftarrow \mathtt{bagof}(\mathtt{s})\}$
$\mathtt{flatten}(\mathtt{q}),\ (\mathtt{q} : \mathtt{array}(\mathtt{array}(-)))$	$\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{s}\}$
$\mathtt{order\_flatten}(\mathtt{q}),\ \mathtt{q} : \mathtt{list}(\mathtt{bag}(-))$	$\mathtt{list}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{listof}(\mathtt{s})\}$
$\mathtt{order\_flatten}(\mathtt{q}),\ \mathtt{q} : \mathtt{oset}(\mathtt{bag}(-))$	$\mathtt{oset}\{\mathtt{x} \parallel \mathtt{s} \leftarrow \lvert\mathtt{q}\rvert^*, \mathtt{x} \leftarrow \mathtt{listof}(\mathtt{s})\}$
$\mathtt{distinct}(\mathtt{q})\ (\mathtt{q} : \mathtt{set/bag}(-))$	$\mathtt{set}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \lvert\mathtt{q}\rvert^*\}$
$\mathtt{distinct}(\mathtt{q})\ (\mathtt{q} : \mathtt{oset/list}(-))$	$\mathtt{oset}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \lvert\mathtt{q}\rvert^*\}$
$\mathtt{distinct}(\mathtt{q})\ (\mathtt{q} : \mathtt{array}(-))$	$\mathtt{oset}\{\mathtt{x} \parallel \mathtt{x} \leftarrow \mathtt{listof}(\lvert\mathtt{q}\rvert^*)\}$
$\mathtt{q}_1\ <\mathtt{comp\_op}>\mathtt{q}_2$ $(<\mathtt{comp\_op}> \in \{<, >, \leq, \geq, =, \neq\})$	$\lvert\mathtt{q}_1\rvert^*\ <\mathtt{comp\_op}>\lvert\mathtt{q}_2\rvert^*$
$\mathtt{q}_1\ <\mathtt{comp\_op}>\ \mathtt{all}\ \mathtt{q}_2$ $(<\mathtt{comp\_op}> \in \{<, >, \leq, \geq, =, \neq\})$	$\mathtt{all}\{\lvert\mathtt{q}_1\rvert^*\ <\mathtt{comp\_op}>\mathtt{x}_2 \parallel \mathtt{x}_2 \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_2\rvert^*)\}$
$\mathtt{q}_1\ <\mathtt{comp\_op}>\ \mathtt{any/some}\ \mathtt{q}_2$ $(<\mathtt{comp\_op}> \in \{<, >, \leq, \geq, =, \neq\})$	$\mathtt{some}\{\lvert\mathtt{q}_1\rvert^*\ <\mathtt{comp\_op}>\mathtt{x}_2 \parallel \mathtt{x}_2 \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_2\rvert^*)\}$
$\mathtt{q}_1\ <\mathtt{arithm\_op}>\ \mathtt{q}_2$ $(<\mathtt{arithm\_op}> \in \{+, -, *, /, \mathtt{mod}\})$	$\lvert\mathtt{q}_1\rvert^*\ <\mathtt{arithm\_op}>\ \lvert\mathtt{q}_2\rvert^*$

OQL Expressions	Calculus Expressions
$-\mathtt{q}$	$-\lvert\mathtt{q}\rvert^*$
$\mathtt{not}\ (\mathtt{q})$	$\mathtt{not}\ (\lvert\mathtt{q}\rvert^*)$
$\mathtt{q}_1\ \mathtt{and}\ \mathtt{q}_2$	$\mathtt{merge[all]}(\lvert\mathtt{q}_1\rvert^*, \lvert\mathtt{q}_2\rvert^*)$
$\mathtt{q}_1\ \mathtt{or}\ \mathtt{q}_2$	$\mathtt{merge[some]}(\lvert\mathtt{q}_1\rvert^*, \lvert\mathtt{q}_2\rvert^*)$
$\mathtt{element}(\mathtt{q})$	$\mathtt{elementof}(\lvert\mathtt{q}\rvert^*)$
$\mathtt{listtoset}(\mathtt{q})$	$\mathtt{set}\{\mathtt{x} \,\|\!|\, \mathtt{x} \leftarrow \mathtt{q}\}$
$\mathtt{max}(\mathtt{q})$	$\mathtt{max}\{\mathtt{x} \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\}$
$\mathtt{min}(\mathtt{q})$	$\mathtt{min}\{\mathtt{x} \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\}$
$\mathtt{avg}(\mathtt{q})$	$\mathtt{unit[avg]}(< \mathtt{avg\_sum} = \mathtt{sum}\{\mathtt{x} \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\},$ $\mathtt{avg\_count} = \mathtt{sum}\{1 \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\} >\}$
$\mathtt{count}(\mathtt{q})$	$\mathtt{sum}\{1 \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\}$
$\mathtt{sum}(\mathtt{q})$	$\mathtt{sum}\{\mathtt{x} \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\}$
$\mathtt{q}_1\ \mathtt{in}\ \mathtt{q}_2$	$\mathtt{some}\{\mathtt{x} = \lvert\mathtt{q}_1\rvert^* \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_2\rvert^*)\}$
$\mathtt{for\ all\ x\ in}\ \mathtt{q}_1 : \mathtt{q}_2$	$\mathtt{all}\{\lvert\mathtt{q}_2\rvert^* \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*)\}$
$\mathtt{exists\ x\ in}\ \mathtt{q}_1 : \mathtt{q}_2$	$\mathtt{some}\{\lvert\mathtt{q}_2\rvert^* \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*)\}$
$\mathtt{exists}(\mathtt{q})$	$\mathtt{some}\{\mathtt{true} \,\|\!|\, \mathtt{x} \leftarrow \lvert\mathtt{q}\rvert^*\}$
$\mathtt{unique}(\mathtt{q})$	$\mathtt{sum}\{1 \,\|\!|\, \mathtt{x} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}\rvert^*)\} = 1$
$\mathtt{struct}(\ell_1 : \mathtt{q}_1, \ldots, \ell_\mathtt{n} : \mathtt{q}_\mathtt{n})$	$\mathtt{struct}(\ell_1 = \lvert\mathtt{q}_1\rvert^*, \ldots, \ell_\mathtt{n} = \lvert\mathtt{q}_\mathtt{n}\rvert^*)$
$\mathtt{C}(\ell_1 : \mathtt{q}_1, \ldots, \ell_\mathtt{n} : \mathtt{q}_\mathtt{n})$	$\mathtt{C}(\ell_1 = \lvert\mathtt{q}_1\rvert^*, \ldots, \ell_\mathtt{n} = \lvert\mathtt{q}_\mathtt{n}\rvert^*)$
$\mathtt{select}\ [\mathtt{distinct}]\ \mathtt{q}'$ $\mathtt{from}\ \mathtt{q}_1\ \mathtt{as}\ \mathtt{x}_1, \ldots, \mathtt{q}_\mathtt{k}\ \mathtt{as}\ \mathtt{x}_\mathtt{k}$ $\mathtt{where}\ \mathtt{q}''$	$\mathtt{bag}\ [\lvert\mathtt{set}]\{\lvert\mathtt{q}'\rvert^* \,\|\!|\, \mathtt{x}_1 \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*), \ldots,$ $\mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_\mathtt{k}\rvert^*), \lvert\mathtt{q}''\rvert^*\}$
$\mathtt{select}\ [\mathtt{distinct}]\ \mathtt{q}'$ $\mathtt{from}\ \mathtt{q}_1\ \mathtt{as}\ \mathtt{x}_1, \ldots, \mathtt{q}_\mathtt{k}\ \mathtt{as}\ \mathtt{x}_\mathtt{k}$ $\mathtt{where}\ \mathtt{q}''$ $\mathtt{order\ by}\ \mathtt{q}'''_1\ [\mathtt{asc/desc}],$ $\ldots, \mathtt{q}'''_\mathtt{j}\ [\mathtt{asc/desc}]$	$\mathtt{list}\ [\lvert\mathtt{oset}]\{\mathtt{v}_1.\mathtt{partition} \,\|\!|\, \mathtt{v}_1 \leftarrow \mathtt{orderof}(\mathtt{group}\{$ $\mathtt{struct}(\mathtt{asc/desc}_1 = \lvert\mathtt{q}'''_1\rvert^*, \ldots, \mathtt{asc/desc}_\mathtt{j} = \lvert\mathtt{q}'''_\mathtt{j}\rvert^*,$ $\mathtt{partition} = \mathtt{unit[bag]}(\lvert\mathtt{q}'\rvert^*)) \,\|\!|\, \mathtt{x}_1 \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*),$ $\ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_\mathtt{k}\rvert^*), \lvert\mathtt{q}''\rvert^*\})\}$
$\mathtt{select}\ [\mathtt{distinct}]\ \mathtt{q}'$ $\mathtt{from}\ \mathtt{q}_1\ \mathtt{as}\ \mathtt{x}_1, \ldots, \mathtt{q}_\mathtt{k}\ \mathtt{as}\ \mathtt{x}_\mathtt{k}$ $\mathtt{where}\ \mathtt{q}''$ $\mathtt{group\ by}\ \ell_1 : \mathtt{q}'''_1, \ldots, \ell_\mathtt{j} : \mathtt{q}'''_\mathtt{j}$ $\mathtt{having}\ \mathtt{q}''''$ $\mathtt{order\ by}\ \mathtt{q}'''''_1\ [\mathtt{asc/desc}],$ $\ldots, \mathtt{q}'''''_\mathtt{n}\ [\mathtt{asc/desc}]$	$\mathtt{list}\ [\lvert\mathtt{oset}]\{\mathtt{v}_1.\mathtt{partition} \,\|\!|\, \mathtt{v}_1 \leftarrow \mathtt{orderof}(\mathtt{group}\{$ $\mathtt{struct}(\mathtt{asc/desc}_1 = \lvert\mathtt{q}'''''_1\rvert^*, \ldots, \mathtt{asc/desc}_\mathtt{n} = \lvert\mathtt{q}'''''_\mathtt{n}\rvert^*,$ $\mathtt{partition} = \mathtt{unit[bag]}(\mathtt{v}_2)) \,\|\!|\, \mathtt{v}_2 \leftarrow \mathtt{bagof}(\mathtt{group}\{$ $\mathtt{struct}(\ell_1 = \lvert\mathtt{q}'''_1\rvert^*, \ldots, \ell_\mathtt{j} = \lvert\mathtt{q}'''_\mathtt{j}\rvert^*, \mathtt{partition} =$ $\mathtt{unit[bag]}(\mathtt{struct}(\mathtt{x}_1 = \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k} = \mathtt{x}_\mathtt{k})) \,\|\!|\, \mathtt{x}_1 \leftarrow$ $\mathtt{bagof}(\lvert\mathtt{q}_1\rvert^*), \ldots, \mathtt{x}_\mathtt{k} \leftarrow \mathtt{bagof}(\lvert\mathtt{q}_\mathtt{k}\rvert^*), \lvert\mathtt{q}''\rvert^*\}), \ell_1 \equiv \mathtt{v}_2.\ell_1,$ $\ldots, \ell_\mathtt{j} \equiv \mathtt{v}_2.\ell_\mathtt{j}, \mathtt{partition} \equiv \mathtt{v}_2.\mathtt{partition}, \lvert\mathtt{q}''''\rvert^*\})\}$

Appendix C

Edge traversals in the backtracking algorithm

C.1 Edge traversals given a graph with V vertices

Given a number of vertices V, the more connected the graph, the more the edge traversals in step 1. Hence, the worst case scenario occurs when the graph is fully connected. For a directed graph, this implies that $\mathtt{E} = \mathtt{V} * (\mathtt{V} - 1)$. Each constraint has $\mathtt{V} - 1$ incoming links from all other constraints and $\mathtt{V} - 1$ outgoing edges towards them.

Assume that backtracking starts from a certain target constraint. Backtracking can be implemented both as a depth-first-search (DFS) and as a breadth-first-search (BFS) method, both having the same complexity. Starting from the target constraint, the first level of BFS traverses $\mathtt{V} - 1$ edges. The paths traversed so far consist of two constraints or vertices. The second time, each one of the $\mathtt{V} - 1$ vertices traverses $\mathtt{V} - 2$ edges, thus forming acyclic paths containing three vertices. This procedure continues until the $\mathtt{V} - 1$ level of BFS, that is, until the traversed paths have V constraints. The table below shows the number of edges traversed at each level.

Level	TraversedEdges
1	$(\mathtt{V} - 1)$
2	$(\mathtt{V} - 1)(\mathtt{V} - 2)$
3	$(\mathtt{V} - 1)(\mathtt{V} - 2)(\mathtt{V} - 3)$
4	$(\mathtt{V} - 1)(\mathtt{V} - 2)(\mathtt{V} - 3)(\mathtt{V} - 4)$
...	...
$\mathtt{n} = \mathtt{V} - 1$	$(\mathtt{V} - 1)(\mathtt{V} - 2)(\mathtt{V} - 3) \ldots (\mathtt{V} - (\mathtt{V} - 1))$

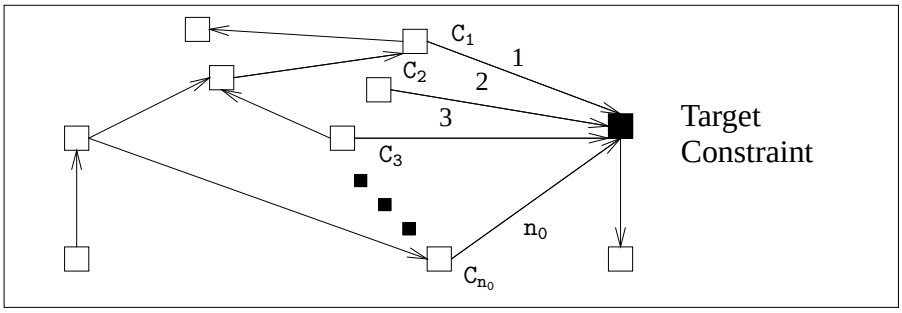

Figure C.1: The target constraint has $\mathtt{n}_0$ incoming links

The total number of traversed edges is:

$$\mathtt{TE} = \sum_{\mathtt{i}=1}^{\mathtt{V}-1}[(\mathtt{V}-1)(\mathtt{V}-2)\ldots(\mathtt{V}-\mathtt{i})]$$

$$\Rightarrow \quad \mathtt{TE} = \sum_{\mathtt{i}=1}^{\mathtt{V}-1}(\prod_{\mathtt{j}=1}^{\mathtt{i}}(\mathtt{V}-\mathtt{j})) \tag{C.1}$$

C.2 Edge traversals given a graph with E edges

Let us consider the backtracking algorithm for a target constraint $\mathtt{C}_0$. Assume that $\mathtt{C}_0$ has $\mathtt{n}_0$ incoming links, as illustrated in figure C.1. The maximum number of edges traversed backwards starting from $\mathtt{C}_0$, in a graph of E edges, is denoted as $\mathtt{TE}(\mathtt{C}_0, \mathtt{E})$, or simply $\mathtt{TE}(\mathtt{E})$. It is the sum of the following edges:

- $\mathtt{n}_0$ (the incoming links to $\mathtt{C}_0$)
- $\sum_{\mathtt{i}=1}^{\mathtt{n}_0} \mathtt{TE}(\mathtt{E}-\mathtt{n}_0)$ (the traversed edges from the $\mathtt{C}_\mathtt{i},\ \mathtt{i}=1,\ldots,\mathtt{n}_0$ backwards).

Notice that the traversed edges from the constraints $\mathtt{C}_\mathtt{i},\ \mathtt{i}=1,\ldots,\mathtt{n}_0$ do not consider the initial $\mathtt{n}_0$ edges. The latter will never be traversed again, since they are incoming links to a vertex that has already been encountered. We refer to $\mathtt{C}_\mathtt{i},\ \mathtt{i}=1,\ldots,\mathtt{n}_0$ as first-level constraints, because they would be encountered at the first level in a breadth-first-search implementation of the algorithm.

Assume that the first-level constraints have on average $\mathtt{n}_1$ incoming links. It can be easily inferred that

$$\mathtt{TE}(\mathtt{E}) = \mathtt{n}_0 + \textstyle\sum_{\mathtt{i}_0=1}^{\mathtt{n}_0}(\mathtt{TE}(\mathtt{E}-\mathtt{n}_0)) = \mathtt{n}_0 + \sum_{\mathtt{i}_0=1}^{\mathtt{n}_0}(\mathtt{n}_1 + \sum_{\mathtt{i}_1=1}^{\mathtt{n}_1}(\mathtt{TE}(\mathtt{E}-\mathtt{n}_0-\mathtt{n}_1))) \Rightarrow$$

$$\begin{aligned}\mathtt{TE} = \mathtt{n}_0 + \textstyle\sum_{\mathtt{i}_0=1}^{\mathtt{n}_0}(\mathtt{n}_1) + \ldots + \sum_{\mathtt{i}_0=1}^{\mathtt{n}_0}\sum_{\mathtt{i}_1=1}^{\mathtt{n}_1}\cdots\sum_{\mathtt{i}_{\mathtt{m}-1}=1}^{\mathtt{n}_{\mathtt{m}-1}}(\mathtt{n}_\mathtt{m}) + \\ + \textstyle\sum_{\mathtt{i}_0=1}^{\mathtt{n}_0}\sum_{\mathtt{i}_1=1}^{\mathtt{n}_1}\cdots\sum_{\mathtt{i}_\mathtt{m}=1}^{\mathtt{n}_\mathtt{m}}(\mathtt{TE}(\mathtt{E}-\mathtt{n}_0-\ldots-\mathtt{n}_\mathtt{m}))\end{aligned}$$

where the last term $\sum_{i_0=1}^{n_0}\sum_{i_1=1}^{n_1}\ldots\sum_{i_m=1}^{n_m}(TE(E-n_0-\ldots-n_m))=0$. Hence,

$$TE = n_0 + n_0 * n_1 + n_0 * n_1 * n_2 + \ldots + n_0 * \ldots * n_m \tag{C.2}$$

$$E = n_0 + n_1 + \ldots + n_m \tag{C.3}$$

The value of TE in formula C.2 is maximized when $n_0 > n_1 > \ldots > n_m$ and in particular when $n_i = n_{i-1} + 1$. If we set $n = n_0$, then the formulae C.2, C.3 take the following forms:

$$TE = n + n(n-1) + \ldots + n(n-1)\ldots(n-m) \tag{C.4}$$

$$E = n(m+1) - m(m+1)/2 \tag{C.5}$$

The question that arises at this point is which is the combination of n and m that maximizes the bound TE on the number of traversed edges. The example in the following table shows that TE is maximized when m is maximized.

$Edges = 15$

n_0	n_1	n_2	n_3	n_4	$\sum n_i$	TE
5	4	3	2	1	15	325
6	5	4			15	156
8	7				15	64

Hence, TE is maximized if $n = m + 1$. This enables the following transformations:

$$E = (m+1)(m+1) - m(m+1)/2 \Rightarrow$$
$$m^2 + 3m + 2(1-E) = 0 \Rightarrow$$
$$m = (-3 + \sqrt{1+8E})/2$$

The formula that calculates the traversed edges becomes:

$$TE = (m+1) + (m+1)(m) + (m+1)(m)(m-1) + \ldots + (m+1)! \tag{C.6}$$

where $m = \lceil(-3 + \sqrt{1+8E})/2\rceil$.

For instance, when $E = 3$, $m = (-3 + \sqrt{1+24})/2 = 1$ and $TE = 2 + 2 * 1 = 4$. In fact, when $E = 3$ the maximum number of traversed edges is $3 \leq 4$.

When $E = 6$, $m = (-3 + \sqrt{1+48})/2 = 2$ and $TE = 3 + 3 * 2 + 3 * 2 * 1 = 15$. Indeed, when $E = 6$ the maximum number of traversed edges is $9 \leq 15$.

Notice that it is wrong to assume that the maximum number of traversed edges occurs when the edges form a fully connected graph. If this was true, then we would calculate the number of vertices using the formula $V(V-1) = E$; given V, the maximum traversed edges would be evaluated as in section C.1.

Appendix D

Implementation of OQL Processor

This appendix gives an overview of the OQL processor developed to test the ideas and to perform the experiments discussed in this thesis.

An OQL interpreter was developed that performs the following tasks:

- lexical analysis
- parsing
- type checking and type inference
- translation of OQL into calculus representation
- normalization at the calculus level
- translation of calculus into algebraic representation
- semantic optimization
- query evaluation

The query interpreter uses the object storage manager and the Java binding of POET OSS 6.1.

The first step towards developing an OQL interpreter is to write a specification file for OQL terminals (OQL keywords, atomic literals and variables). This file is used by JLex, a lexical analyzer generator for Java, in order to produce a class that takes a query and whenever invoked, returns the next terminal encountered.

A set of Java classes are then defined corresponding to the various OQL constructs. For instance, the classes `Select_Query`, `Ref_Access_Query`, `Index_Access_Query` correspond to the `select`

clause, `e.label` and $\texttt{e}_1[\texttt{e}_2]$ respectively. The class `Main_Query` corresponds to the outer OQL expression, which potentially includes query definitions. All classes inherit from the general class `Query`.

A second specification file includes the grammar of OQL, i.e. the query expressions that are syntactically acceptable. Each OQL expression is paired with the action that should be taken when the expression is encountered. CUP, an LALR Parser Generator for Java, takes as input the OQL grammar specification and produces a parser for it. The generated parser takes a query string and, with the aid of the lexical analyzer, it generates a `Query` object for each query subexpression encountered. When the whole query is parsed a `Main_Query` object is created and its method `process` is invoked. This method checks the type of the query, translates it to calculus form, normalizes it, translates it to algebraic form, optimizes it semantically and evaluates its result.

All subclasses of `Query`, including `Main_Query`, define the methods `checkType`, `inferTypes` and `toMonoidCalculus`. Their implementation varies depending on the specific OQL construct that they represent.

In the `process` method of `Main_Query`, we first invoke the method `checkType`. In order to derive the type of a query, we need to identify the types of its operands; hence, `checkType` is recursively called on its subqueries. Access to the database schema is eventually required when `checkType` is invoked on objects that represent identifiers that are not range variables. Since the Java binding of POET OSS 6.1 does not provide an interface to the schema information, we needed to define and use a method `getType` for all classes in the schema. The method `inferTypes` is also called recursively, but it is used only when the input OQL expression includes query definitions with untyped parameters. Only the types of the parameters are inferred, whilst the remaining identifiers are type-checked in the usual way.

After ensuring that the input query is well-typed, the method `toMonoidCalculus` is invoked in order to translate the query into its monoid calculus form. The latter is invoked recursively on the subqueries (operands or attributes), in a similar way as in the case of `checkType`. For each monoid calculus expression, we define a corresponding class, whose name starts with `MC_`, e.g. `MC_Comprehension`, `MC_Ref_Access`, `MC_Identifier`. All `MC_` classes inherit from the general monoid calculus class `MC_Query`. Hence, the method `toMonoidCalculus` returns an object of type `MC_Query`, referred to hereafter as the calculus object.

All subclasses of `MC_Query` implement the method `normalize`, based on the normalization algorithm defined in chapter 4. They also define a method `toALG` for the translation of calculus constructs to algebraic representation (chapter 5). The implementation of both methods depends on the calculus construct (subclass).

We first invoke the method `normalize` on the calculus object and then the method `toALG` on the normalized calculus object. For each algebraic expression we define a corresponding class whose

name starts with ALG_, e.g. ALG_Selection, ALG_Nest. All algebraic classes inherit from the general class ALG_Query. Hence, the method toALG returns an object of type ALG_Query, referred to as the algebraic object.

A method that performs semantic optimization is defined in the class ALG_Query and has been implemented to apply the constraint introduction or elimination heuristic whenever possible. Hence, it is only implemented in subclasses of ALG_Query that represent algebraic expressions with predicates, e.g. ALG_Selection. This method uses the association rules that refer to the operand of the algebraic object and finds direct or indirect associations (with exceptions) in order to add a constraint on an index attribute, or to eliminate any redundant non-index constraints.

We have implemented a rule mining module based on the algorithms by Agrawal et al. [AIS93] and Srikant et al. [SA96]. This module identifies association rules corresponding to a specific extent, which are stored in the database along with their exceptions. The input values of the rule mining algorithm are the minimum support and the minimum confidence required for the resulting rules. In order to find rules between quantitative attributes of an extent, we first apply a clustering algorithm on their values. The Birch algorithm [ZRL96] has been implemented to distribute the values of an attribute into a number of clusters. For each quantitative attribute we define a mirror attribute in order to store cluster identifiers. Hence, instead of finding association rules between individual values by looking at the quantitative attributes, we discover rules relating clusters. That is, we run the rule mining algorithm over the corresponding mirror attributes that hold cluster information.

After an algebraic query has been optimized semantically, it is evaluated using the Java binding to access the POET object database. All algebraic operators (queries) implement the following method.

```
public void exec (ALG_Basket initiator_basket, Thread initiator_exec_thread)
{
    this.start_resource_input ();
    this.consume_resources (initiator_basket, initiator_exec_thread);
}
```

The initiator_basket is the structure where the current operator is expected to put its results. This structure is initialized by the parent operator. The first method (start_resource_input) initializes the baskets of the operands; the latter may then start populating these baskets with objects that the current operator will use as input. The next method (consume_resources) is invoked on the current operator in order to start processing the objects retrieved from the baskets of the operands. The output of this processing is accumulated into the initiator_basket.

If the operand of an algebraic query is an extent, a database name or an index constraint on an extent, the method consume_resources accesses the database using the Java binding of POET. In case the operand is an extent, an iterator is started and the method Iterator.next() is used to fetch the next object from the database. Indices are implemented as objects containing a list of

values and a list of objects. Hence, accessing an extent using an index implies resolving the index object using the method `Transaction.lookup(index_name)` and then invoking on it the method `Index.select(range)`, that fetches from the database all objects for which the value of the index attribute is in the given range. Named objects are also resolved from the database using the built-in method `Transaction.lookup(object_name)`.

The implementation of the method `consume_resources` depends on the algebraic construct (select, join, project, nest etc). Although the focus of this dissertation was not on syntactic algebraic optimization and efficient physical execution plans, this framework provides the opportunity to implement the method `consume_resources` efficiently. For instance, the algebraic operator `Join` could be implemented as a `SemiJoin`, i.e. it could store only the instances of the left operand in the result; this would be possible if the right operand is only referred to only in the predicate of `Join` and in none of the parent operators.

If algebraic operators (objects) are viewed as nodes and their relationships to operand objects as edges, the outer algebraic object can be viewed as the root of an algebraic tree that can potentially be rearranged and optimized using well-known techniques from relational databases. However, the use of new and monoid-aware algebraic operators in our context provides opportunities for further research in order to identify algebraic syntactic transformations and efficient execution plans for OQL.

Bibliography

[AFP00] M. Akhtar Ali, A.A.A. Fernandes, and N.W. Paton. Incremental maintenance of materialized OQL views. In *ACM Intl Workshop on Data Warehousing and OLAP*, pages 41–48, 2000.

[Afs98] M. Afshar. *An Open Parallel Architecture for Data-intensive Applications*. PhD thesis, University of Cambridge, Computer Laboratory, 1998.

[AH95] O. Agesen and U. Holzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 91–107, 1995.

[AHLS00] J. Albrecht, W. Hümmer, W. Lehner, and L. Schlesinger. Query optimization by using derivability in a data warehouse environment. In *ACM Intl Workshop on Data Warehousing and OLAP*, pages 49–56, 2000.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Intl Conf on Management of Data*, pages 207–216, 1993.

[Ala97] S. Alagić. The ODMG object model: Does it make sense? In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 253–270, 1997.

[Ala99] S. Alagić. Type checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems*, 24(3):319–360, September 1999.

[ASY98] C.C. Aggarwal, Z. Sun, and P.S. Yu. Online algorithms for finding profile association rules. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 86–95, 1998.

[BBSV97] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in OODB. In *Intl Conf on Data Engineering (ICDE)*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM SIGMOD Intl Conf on Management of Data*, pages 505–516, 1996.

[Bel96] S. Bell. Deciding distinctness of query results by discovered constraints. In *Intl Conf on the Practical Application of Constraint Technology (PACT)*, pages 399–417, 1996.

[BHP94] M.W. Bright, A.R. Hurson, and S. Pakzad. Automated resolution of semantic heterogeneity in multidatabases. *ACM Transactions on Database Systems*, 19(2), 1994.

[BLS+94] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[BO96] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.

[BT00] G.M. Bierman and A. Trigoni. Towards a formal type system for ODMG OQL. Technical Report 497, University of Cambridge, Computer Laboratory, October 2000.

[CA97] K.C.C. Chan and W-H Au. Mining fuzzy association rules. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 209–215, 1997.

[Car87] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[CBB+00] R.G.G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[CGK+99] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 UDB. In *Intl Conf on Very Large Databases (VLDB)*, pages 687–698, 1999.

[CGM88] U.S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In *Foundations of Deductive Databases and Logic Programming (DDLP)*, pages 243–273, 1988.

[CGM90] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, 1990.

[Cha98] S. Chaudhuri. An overview of query optimization in relational systems. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 34–43, 1998.

[CLC95] J-Y. Chung, Y-J. Lin, and D.T Chang. Object and relational databases. In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA Addendum)*, pages 164–169, 1995.

[CLK97] D.W. Cheung, S.D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Intl Conf on Database Systems for Advanced Applications (DASFAA)*, pages 185–193, 1997.

[CMN95] M.J. Carey, N.M. Mattos, and A.K. Nori. Object-relational database systems (tutorial); principles, products and challenges. In *ACM SIGMOD Intl Conf on Management of Data*, pages 455–459, 1995.

[CS96] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Conf on Extending Database Technology (EDBT)*, pages 167–182, 1996.

[Dam85] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Edinburgh University, 1985.

[DD88] J. Duhl and C. Damon. A performance comparison of object and relational databases using the Sun Benchmark. In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 153–163, 1988.

[EGLM01] J. Edmonds, J. Gryz, D. Liang, and R.J. Miller. Mining for empty rectangles in large data sets. In *Intl Conf on Database Theory (ICDT)*, pages 174–188, 2001.

[Feg94] L. Fegaras. A uniform calculus for collection types. Technical Report 94-030, Oregon Graduate Institute, 1994.

[Feg97] L. Fegaras. An experimental optimizer for OQL. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997.

[Feg98] L. Fegaras. Query unnesting in object-oriented databases. In *ACM SIGMOD Intl Conf on Management of Data*, pages 49–60, 1998.

[FM95] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *ACM SIGMOD Intl Conf on Management of Data*, pages 47–58, 1995.

[FM00] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 2000.

[FMMT96] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 182–191, 1996.

[GGMR97] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Intl Conf on Data Engineering (ICDE)*, pages 444–453, 1997.

[GGZ01] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *ACM SIGMOD Intl Conf on Management of Data*, pages 582–592, 2001.

[GKG+97] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M.H. Scholl. Query evaluation in CROQUE - calculus and algebra coincide. In *15th British National Conference on Databases (BNCOD)*, pages 84–100, 1997.

[GLQ99] J. Gryz, L. Liv, and X. Qian. Semantic query optimization techniques in IBM DB2: initial results. Technical Report CS-1999-01, York University, 1999.

[GLR97] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–74, March 1997.

[GS96] T. Grust and M.H. Scholl. Translating OQL into monoid comprehensions — stuck with nested loops? Technical report, University of Konstanz, Dept. of Mathematics and Computer Science, 1996.

[GS98a] T. Grust and M.H. Scholl. Hybrid strategies for query translation and optimization. Technical report, University of Konstanz, Dept. of Mathematics and Computer Science, 1998.

[GS98b] T. Grust and M.H. Scholl. Query deforestation. Technical report, University of Konstanz, Dept. of Mathematics and Computer Science, 1998.

[Gun92] C.A. Gunter. *Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1992.

[Hel98] J.M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[HHCF96] J. Han, Y. Huang, N. Cercone, and Y. Fu. Intelligent query answering by knowledge discovery techniques. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):373–390, 1996.

[HIT97] Z. Hu, H. Iwasaki, and M. Takechi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HK96] C.N. Hsu and C. Knoblock. Using inductive learning to generate rules for semantic query optimization. In *Advances in Knowledge Discovery and Data Mining*, pages 425–445, 1996.

[HK98] C.N. Hsu and C. A. Knoblock. Discovering robust knowledge from databases that change. *Data Mining and Knowledge Discovery*, 2(1):69–95, 1998.

[HK00] C.N. Hsu and C. A. Knoblock. Semantic query optimization for query plans of heterogeneous multidatabase systems. *Knowledge and Data Engineering*, 12(6):959–978, 2000.

[Hsu96] C.N. Hsu. *Learning effective and robust knowledge for semantic query optimization*. PhD thesis, Dept of Computer Science, University of Southern California, 1996.

[Hsu97] C.N. Hsu. Tradeoff in rule induction for semantic query optimization. In *AAAI Workshop on Building Resource-Bounded Reasoning Systems*, pages 61–68, 1997.

[Hul97] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *ACM Symposium on Principles of Database Systems (PODS)*, 1997.

[Ioa96] Y.E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[JS97] V. Josifovski and S.Y.W. Su. Incorporating association pattern and operation specification in ODMG's OQL. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 332–339, 1997.

[Kin81a] J.J. King. *Query optimization by semantic reasoning*. PhD thesis, Stanford University, 1981.

[Kin81b] J.J. King. Quist: A system for semantic query optimization in relational databases. In *Intl Conf on Very Large Databases (VLDB)*, pages 510–517, 1981.

[KMR+94] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 401–407, 1994.

[KPH98] J. Kröger, S. Paul, and A. Heuer. Query optimization: On the ordering of rules. Technical report, CS Department, University of Rostock, 1998.

[LS95] A.Y. Levy and Y. Sagiv. Semantic query optimization in datalog programs (extended abstract). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 163–173, 1995.

[MC96] I. Min and A. Chen. Query answering using discovered rules. In *Intl Conf on Data Engineering (ICDE)*, pages 402–411, 1996.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems Sciences*, 17:348–375, 1978.

[MY97] R.J. Miller and Y. Yang. Association rules over interval data. In *ACM SIGMOD Intl Conf on Management of Data*, 1997.

[NLH98] R.T. Ng, L.V.S. Lakshmanan, and J. Han. Exploratory mining and pruning optimizations of constrained association rules. In *ACM SIGMOD Intl Conf on Management of Data*, pages 13–24, 1998.

[OHMS92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in ObjectStore database system. In *ACM SIGMOD Intl Conf on Management of Data*, pages 403–412, 1992.

[Par95] J.S. Park. An effective hash-based algorithm for mining association rules. In *ACM SIGMOD Intl Conf on Management of Data*, pages 175–186, 1995.

[PC94] J. Plevyak and A.A. Chien. Precise concrete type inference for object-oriented languages. In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 324–340, 1994.

[PHH92] H. Pirahesh, J.M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *ACM SIGMOD Intl Conf on Management of Data*, pages 39–48, 1992.

[PLH97] H. Pirahesh, T.Y.C. Leung, and W. Hasan. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In *Intl Conf on Data Engineering (ICDE)*, pages 391–400, 1997.

[PS91] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *ACM Conf on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 146–161, 1991.

[PS96] A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *VLDB Journal*, 5(2):119–132, 1996.

[PS97] A. Poulovassilis and C. Small. Formal foundations for optimising aggregation functions in database programming languages. In *Workshop on Database Programming Languages (DBPL)*, pages 299–318, 1997.

[Pul96] E. Pulido. Recursive query processing using graph traversal techniques. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 37–44, 1996.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RS93] C. Rich and M.H. Scholl. Query optimization in oodbms. In *BTW*, pages 266–284, 1993.

[RS97] H. Riedel and M.H. Scholl. A formalization of ODMG queries. In *Seventh IFIP 2.6 Working Conference on Database Semantics (DS-7)*, 1997.

[SA96] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *ACM SIGMOD Intl Conf on Management of Data*, pages 1–12, 1996.

[Sam00] P.R.F. Sampaio. *Design and Implementation of a Deductive Query Language for ODMG Compliant Object Databases*. PhD thesis, University of Manchester, 2000.

[SdBB96] H.J. Steenhagen, R.A. de By, and H.M. Blanken. Translating OSQL-queries into efficient set expressions. In *Conf on Extending Database Technology (EDBT)*, pages 183–200, 1996.

[SHKC93] S. Shekar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):950–964, 1993.

[SLR94] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *ACM SIGMOD Intl Conf on Management of Data*, pages 430–441, 1994.

[SO95a] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, 1995.

[SO95b] D.D. Straube and M.T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):210–227, 1995.

[SP00] P.R.F. Sampaio and N.W. Paton. Query processing in DOQL: A deductive database language for the ODMG model. *Data and Knowledge Engineering*, 2000.

[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.

[ST95] A. Shrufi and T. Topaloglou. Query processing for knowledge bases using join indices. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 158–166, 1995.

[Sul00] M. Sulzmann. A general type inference framework for Hindley/Milner style systems. Technical Report 2000/15, Dept. of Computer Science and Software Engineering, The University of Melbourne, July 2000.

[TB01] A. Trigoni and G.M. Bierman. Inferring the principal type and the schema requirements of an OQL query. In *18th British National Conference on Databases (BNCOD)*, pages 185–201, 2001.

[TM01] A. Trigoni and K. Moody. Using association rules to add or eliminate query constraints automatically. In *Intl Conf on Scientific and Statistical Database Management (SSDBM)*, pages 124–133, 2001.

[Tof87] M. Tofte. *Operational semantics and polymorphic type inference.* PhD thesis, University of Edinburgh, 1987.

[TUA+98] D. Tsur, J. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *ACM SIGMOD Intl Conf on Management of Data*, pages 1–12, 1998.

[Van92] B. Vance. Towards an object-oriented query algebra. Technical report, Oregon Graduate Institute, 1992.

[WM99] L.B. Warshaw and D.P. Miranker. Rule-based query optimization, revisited. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 267–275, 1999.

[WYY00] W. Wang, J. Yang, and P.S. Yu. Efficient mining of weighted association rules (war). In *ACM SIGKDD Intl Conf on Knowledge Discovery and Data Mining (KDD)*, pages 270–274, 2000.

[Yoo94] S.C. Yoon. Intelligent query answering in deductive and object-oriented databases. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 244–251, 1994.

[YS89] C.T. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3), 1989.

[YSP95] S.C. Yoon, I.Y. Song, and E.K. Park. Semantic query processing in object-oriented databases using deductive approach. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 150–157, 1995.

[YSP97] S.C. Yoon, I.Y. Song, and E.K. Park. Intensional query processing using data mining approaches. In *ACM Intl Conf on Information and Knowledge Management (CIKM)*, pages 201–208, 1997.

[ZM90] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[ZRL96] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: an efficient data clustering method for very large databases. In *ACM SIGMOD Intl Conf on Management of Data*, pages 103–114, 1996.